

Игорь
Облаков

obla@bigpost.com



Inside the HP-UX Operating System

Student Workbook

Version D.00

H5081S

Printed in USA 02/99

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ERRORS CONTAINED HERIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS IN CONNECTION WITH THE FURNISHING, PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT, OR OTHER LEGAL THEORY).

Some states do not allow the exclusion of implied warranties or the limitations or exclusions of liability for incidental or consequential damages, so the above limitations and exclusions may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior consent of Hewlett-Packard Company.

UNIX ® is a registered trademark of The Open Group.

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

HP Education
100 Mayfield Avenue
Mountain View, CA 94043 U.S.A.

© Copyright 1999 by the Hewlett-Packard Company

Table of Contents

Module 1: Overview

Module Objectives	1-1
HP-UX Structural Overview	1-2
Kernel Entry.....	1-4
Primary Subsystems	1-8
Process Management Data Structures	1-10
Example : Viewing the Kernel Data Structures	1-15
Example: Accessing the Process Table	1-16
Example: Accessing the Pregion List	1-17
Example: Accessing the File Descriptor Information	1-18
Example: Access the Thread Structures	1-19
Example: Accessing the Uarea.....	1-20
Memory Management Data Structures	1-22
File System Data Structures	1-24
The Big Picture.....	1-28
New Features of HP-UX 11.0	1-30

Module 2: System Architecture

Module Objectives.....	2-1
PA-RISC Architecture Overview	2-2
Processor Architecture	2-4
PA-RISC Processor Versions.....	2-9
HP-UX 11.0 Architectural Support	2-13
Register Context	2-18
General Registers	2-20
Space Registers	2-24
Control Registers	2-26
Instruction Address Queues.....	2-30
PA-RISC 2.0 Processor Status Word (PSW).....	2-32
Virtual Memory Concepts	2-41
Virtual Memory Layout: PA-RISC 1.1 and 2.0 Narrow Mode.....	2-43
Virtual Memory Layout: PA-RISC 2.0 Wide Mode	2-47
32-Bit Address Space Layout.....	2-49
64-Bit Address Space Layout.....	2-51
32- vs 64-Bit HP-UX Address Layouts	2-53
Long and Short Pointers: PA-RISC 1.1 and 2.0 Narrow Mode	2-55
Explicit and Implicit Pointers: PA-RISC 2.0 Wide Mode	2-57
Address Swizzling.....	2-59
PA-RISC 2.0 Global Virtual Addresses (GVA).....	2-61
PA-RISC 2.0 GVA Formation: Narrow Mode.....	2-63
PA-RISC 2.0 GVA Formation: Wide Mode	2-65
Virtual to Physical Address Translation	2-67
Lab: Virtual Addresses: PA-RISC 1.1.....	2-71
Lab: Implicit pointers: PA-RISC 2.0	2-73
Address Translation Components	2-79
Address Translation Through TLB and Cache	2-81
Cache Entry	2-83
Searching the Cache.....	2-85
Direct Mapped Caches.....	2-87
Assist Cache	2-91
4-Way Associative Set Caches	2-94
Translation Lookaside Buffer	2-96
Searching the TLB.....	2-102
Hardware TLB Miss Handler (PA 1.1 Only).....	2-104
Access Control.....	2-108
TLB/Cache Summary	2-114
Instruction Pipelining	2-116
Superscalar Pipelined Execution.....	2-120
Interrupts During Instruction Execution.....	2-122
Interrupt Groups	2-124

Interrupt Vector Table..... 2-130
 Interrupt State Save 2-132
 The Processor Status Word and Interrupt Handling..... 2-134
 Architecture Summary 2-138
 Lab: Exploring Cache & TLB Behavior 2-140

Module 3: Kernel Management

Module Objectives 3-1
 Kernel Services Overview 3-2
 The System Call Interface 3-4
 Argument Coercion (64-bit only)..... 3-11
 System Call Return Path..... 3-12
 Kernel Timeout Services..... 3-16
 The Callout Structures 3-18
 Scheduling a New Timeout..... 3-22
 How Timeouts Expire 3-24
 Kernel Memory Allocation Buckets 3-26
 Bucket Contents 3-28
 When the Bucket is Empty: The **sysmap**..... 3-30
 A Populated **sysmap** 3-32
 Lab: System Calls..... 3-34
 Lab: Callout Structures 3-35

Module 4: Process Management

Module Objectives.....	4-1
Introduction	4-2
Program, Process and Thread Definitions	4-4
System Calls for a Process's Lifecycle	4-8
Process Creation: fork() orvfork()	4-10
Process Creation: fork1()	4-16
Process Creation: newproc()	4-22
Process Creation: procdup()	4-26
Process Creation: vfork() Example.....	4-28
Process Creation: vfork() Calls exec() Example	4-30
Process Creation: exec()	4-34
execve : Call to getxfile()	4-36
Process Termination: exit()	4-38
Process Termination: wait()	4-42
Process Termination: freeproc() and freethread()	4-46
Process Structure Layout.....	4-48
32-Bit User Mode Address Space Layout.....	4-52
SHARE_MAGIC or DEMAND_MAGIC: The Global Perspective.....	4-58
Limitations of 32-Bit Memory Map	4-60
Allowing Larger Data Areas	4-64
EXEC_MAGIC: The The Global Perspective	4-66
Allowing More Shared Memory	4-68
SHMEM_MAGIC: The Global Perspective	4-70
Shared Memory Windows	4-72
64-Bit Kernel Mode and User Mode Address Space Layout	4-74
Process Structure: Virtual Layout Overview	4-76
Process Structure: The Process Table	4-78
Process Management: Threads	4-80
Process Management: Virtual Address Space (VAS).....	4-82
Process Management: pregions	4-84
pregion Skip Lists.....	4-86
Adding a New pregion	4-88
Process Management: User Area (UAREA)	4-90
Lab: Process Table Lab.....	4-94
Process and Thread States.....	4-98
Process Flags (p_flag).....	4-100
Thread Flags (kt_flag).....	4-102
Priority Values (Internal to Kernel)	4-104
Priority Values (External to Kernel)	4-106
Signal Handling.....	4-108
RTSCHED Run Queue	4-110
SCHED RTPRIO Queue	4-116

SCHED_TIMESHARE Queue.....	4-118
SCHED_RTPRIO and SCHED_TIMESHARE Queue Details	4-122
Run Queue Initialization: rqinit()	4-124
Thread Scheduling: Timeline	4-128
Time-shared Thread Priority Behavior.....	4-132
Priority Degradation for CPU-Intensive Applications.....	4-134
Priority Degradation for Interactive Programs.....	4-136
Thread Scheduling: hardclock()	4-138
Thread Scheduling: setpri()	4-140
Thread Scheduling: schedcpu()	4-142
Remove a Thread From a Queue: remrq()	4-146
Add a Thread to a Queue: setrq()	4-148
Adjusting a Thread Priority: Review	4-150
Context Switching.....	4-154
Thread Scheduling - swtch()	4-158
Thread Scheduling - sleep()	4-162
Thread Scheduling - wakeup()	4-166
Process Scheduling Lab	4-170
Process Structures Lab	4-172

Module 5: Multiprocessor Systems

Module Objectives	5-1
Symmetric Multiprocessor Definition	5-2
Hardware Overview	5-4
MP Data Structures	5-6
Per Processor Data Structures	5-10
MP Run Queues	5-14
Uniprocessor Data Contention	5-18
Process Synchronization	5-20
Locking Strategies.....	5-22
Spinlocks.....	5-24
Spinlock Data Structures	5-26
Load and Clear Word Instruction	5-30
Semaphores	5-32
Processor Scheduling	5-34
SPU Load Balancing.....	5-36
Processor Control	5-38
MP Problems	5-42
Lab: Multiprocessor Data Structure	5-44

Module 6: InterProcess Communication (IPC)

Module Objectives.....	6-1
IPC Introduction.....	6-2
IPC Facilities	6-6
Process Level Semaphores	6-8
System V Semaphores: Data Structures	6-10
System V Semaphores: Undo Structure.....	6-12
System V Semaphores: Kernel Parameter Limits.....	6-14
Message Queues	6-18
Message Queues: Data Structures.....	6-20
Message Queues: Allocation of Space.....	6-22
Shared Memory and Global Virtual Address Space	6-26
Shared Memory Management Data	6-28
Data Structures When Using Shared Memory	6-30
32-Bit and 64-Bit Shared Memory Coexistence.....	6-32
Controlling Space Allocation for Shared Memory.....	6-34
Memory Mapped Files.....	6-38
Memory Mapped Files and Global Virtual Address Space.....	6-40
Signals.....	6-42
Signal Delivery Methods	6-44
Signal Anticipation: sigvec()	6-46
Signal Handler: setsigvec()	6-48
Signal Anticipation: Proc and Thread Adjustment.....	6-50
Signal Causation: kill()	6-52
Signal Delivery: psignal()	6-54
Signal Recognition: issig()	6-58
Signal Handling: psig()	6-60
Signal Handler Commencement: sendsig()	6-62
Signal Handler Return: sigcleanup()	6-66
Lab: Signals	6-70
Lab: Shared Memory and Semaphores	6-71
Lab: Signal Handler Return: sigcleanup()	6-72
Lab: Message Queues	6-92
Lab: Message Queue Structures.....	6-95
Lab: Semaphore Structures	6-98

Module 7: Memory Management

Module Objectives	7-1
Data Structure Overview	7-2
Physical RAM Underlying Virtual Memory	7-4
Calculating an Index into the HTBL	7-6
Mapping a Virtual Address to a Physical Address	7-8
When Multiple Addresses Hash to the Same HTBL Entry	7-10
The hpde Structure	7-12
Physical to Virtual Address Translation	7-18
Mapping Multiple Virtual Addresses to One Physical Address	7-20
Hardware Independent Page Information Table (pfdat)	7-22
How Processes Reference Pages	7-26
Pregion Structures and Linked Lists	7-28
Region Structures	7-30
The Root of the btree : broot Structure	7-36
Managing VFD/DBD Pairs	7-40
Virtual Frame Descriptors	7-42
Disk Block Descriptors	7-44
fork() - Duplicating preions with Shared Regions	7-46
fork() - Duplicating preions with Private Regions	7-48
fork() - Copy-On-Write First Read Mechanics	7-52
fork() - Copy-On-Write First Write Mechanics	7-54
Virtual Memory and exec()	7-56
Virtual Memory and exit()	7-60
Bringing in a Page the First Time (Demand Paging)	7-62
Reclaiming Pages from the Free List	7-64
Retrieving Pages from Disk	7-66
Reserving Swap Space	7-68
When to Push Pages Out	7-70
vhand - the Page Daemon	7-74
The Two-Handed Clock Algorithm	7-76
How Much of the preion to Age and Steal	7-78
Choosing a Swap Location	7-80
The swaptab/swapmap Structure	7-84
Swapping Using the Pager	7-88
Lab: Forking and Address Spaces	7-90
Lab: Looking at the Text Area	7-93
Lab: Looking at the Data Area	7-100
Lab: Connecting Physical Page Numbers (PPN) to Virtual Addresses	7-102
Lab: Looking at the Child	7-104
Lab: Reading Translations from the Page Directory	7-106
Lab: Data Access, Parent Reads First	7-108
Lab: Child Accesses the Data	7-110

Table of Contents

Lab: Data Access, Parent Writes First..... 7-112
Lab: True Copy-On-Write, with exec_magic Text Areas..... 7-113
Lab: Data Access, Child Writes 7-120
Lab: Memory Structures 7-122

Module 8: File Systems

Module Objectives	8-1
File System Overview	8-2
The Bell File System	8-3
Problems with the Bell File System	8-5
The HFS File System	8-7
HFS Disk-Resident Data Structure	8-12
The Superblock	8-14
The Cylinder Group	8-22
Blocks and Fragments	8-25
Inodes	8-27
Inodes and Data Blocks	8-30
Inode Allocation	8-32
Block Allocation	8-36
Problems with UFS	8-43
The Veritas Extended File System	8-45
JFS Disk Layout	8-48
The VxFS Superblock	8-49
The Object Location Table	8-52
Filesets and Headers	8-57
Inode Allocation Units	8-59
Free Space Management	8-63
The Extent Allocation Unit State File	8-65
The Extent Allocation Unit Summary File	8-67
The Extent Allocation Unit Freemap File	8-69
Vx Inodes Arrangement	8-72
The Intent Log	8-84
Logging Levels	8-88
Completion Records	8-90
Extended Inode Operations	8-92
Directories	8-96
The Current Usage Table	8-100
Kernel-Based Data Structures	8-102
Virtual File System Structures	8-104
The Mount Structure for UFS File Systems	8-107
The vx_vfs Structure for VxFS File Systems	8-108
File Access Data Structures	8-113
The File Table	8-115
The vnode Structure	8-118
Inodes and the Inode Table	8-121
Locating Incore Inodes	8-126
vx_inodes	8-128
The buffer Cache, Overview	8-130

Table of Contents

Buffer Cache Structures.....	8-132
Dynamic Buffer Cache.....	8-137
Accessing Data Through the Buffer Cache	8-139
Labs: VFS Structures	8-142
Lab: VxFS Structures.....	8-146
Lab: VxFS.....	8-151

Module 9: Logical Volume Manager

Module Objectives.....	9-1
LVM Overview	9-2
LVM Architecture.....	9-4
LVM Disk Layout: Bootable.....	9-8
LVM Disk Layout: Non-Bootable.....	9-10
LVM Disk Structures: PVRA and BDRA	9-12
LVM Disk Structures: VGRA (Header/Trailer)	9-16
LVM Disk Structures: VGRA (PVOL/LVOL Structures)	9-18
Lab: LVM Disk Structures.....	9-24
Memory Resident Structures	9-26
LVMTAB File.....	9-30
Mirror Write Cache	9-34
Mirror Write Cache Data Structures	9-36

Module 10: I/O Subsystem

Module Objectives	10-1
Example topology of a PA-RISC System.....	10-2
I/O and Processor Dependent Code.....	10-6
System Address Space: 32-bit.....	10-10
System Address Space: 64-bit.....	10-14
Modes of I/O Operation.....	10-16
Device Files and the Switch Tables	10-20
The ioconfig File	10-22
The Converged I/O System	10-24
Overview: General I/O and Context Dependent I/O.....	10-26
Overview: General I/O System (GIO).....	10-28
Context Dependent I/O Module (CDIOs).....	10-30
GIO Objects	10-32
General I/O Tree Objects	10-34
I/O Tree Node State Translation.....	10-38
GIO Class Objects	10-40
Kernel Device Table (KDT) Objects	10-42
Switch Tables	10-44
GIO-CDIO Interface.....	10-48
Inter-CDIO Communications	10-52
Kernel Generation Structure.....	10-54
WSIO CDIO	10-56
SIO CDIO.....	10-58
PA CDIO	10-62
CORE CDIO	10-64
EISA CDIO	10-66
Lab: The I/O Subsystem.....	10-69
Lab: I/O Structures	10-73

Table of Contents

Module 11: System Initialization

Module Objectives.....	11-1
The Beginning of System Initialization.....	11-2
Overview - "The Big Picture"	11-3
Firmware - PDH/PDC/IODC.....	11-5
Firmware - Monarch Selection.....	11-7
PDC - Initial Memory Module.....	11-9
IPL - Handoff From PDC	11-11
HPUXBOOT - Handoff From IPL.....	11-13
Kernel Initialization	11-15
I/O Configuration Process - Level 1	11-18
I/O COnfiguration Process - Level 2.....	11-20
Completed I/O Tree	11-22

Appendix A: Q4 Reference Guide

B: Format Log (JFS)
C: Raw Notes
D: RM Log

Module 1 Overview

“I’m a bear of very little brain, and big words bother me.”

Winnie-the-Pooh ch 4, A.A.Milne

Objectives :

- Understand the role of the kernel in the structure of HP-UX
- Familiarity with the purpose and primary structures of major kernel subsystems.

Slide: HP-UX Structural Overview

Every user on an HP-UX system is interacting with the kernel through commands and applications. To complete a requested task, most commands/applications will need to request information or action from the kernel.

The **primary purpose** of the kernel is to

- provide access to system resources
- manage/control system resources (CPU scheduling, RAM, disk space, etc)

The kernel itself is made up of individual components such as

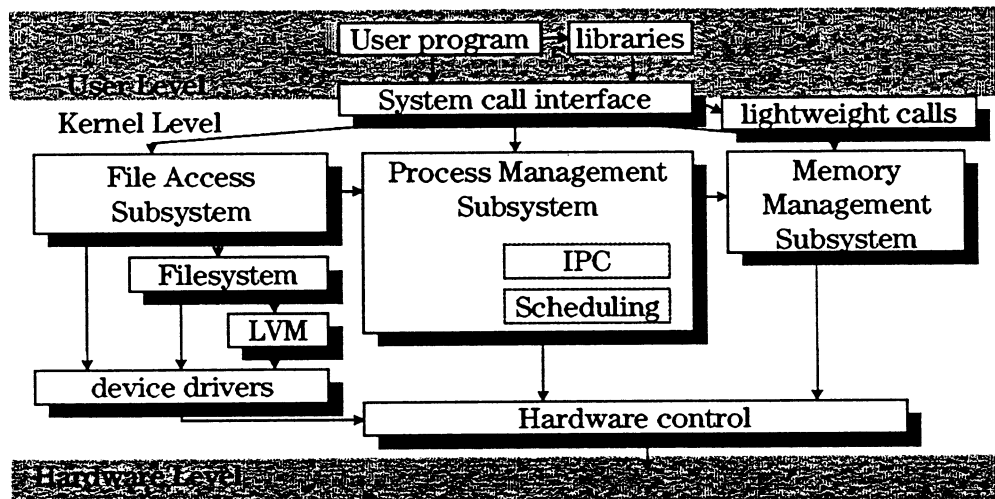
- **Kernel Processes** such as swapper, vhand, init, and statdaemon, *UxFS*
He имеет user memory space
- **Threads** are streams of execution within a process that share the process' address space. While the scheduling of threads lies mostly within the process management subsystem, it touches all areas of the kernel. *Сервисизация изменений совместных. Cache coherency?*
- **Device Drivers** that handle I/O to specific types of devices
- **Privileged library routines** that deal with processes, memory, the file system, and the I/O system.
- **Timeout routines** which are responsible for monitoring other parts of the kernel such as process scheduling and free memory.

Multiple **subsystems** make up the kernel. For the purpose of this course we will primarily deal with the *Process Management, Memory Management, Filesystem, and I/O* subsystems. There are many other areas of interest in the kernel which will be covered but these four are primarily responsible for basic system activity.

Multithread — с 10.30 (пенум где разработчики)
успешно не расширяются

Slide: Kernel Entry

Kernel Entry



a69613

Notes:

Slide: Kernel Entry

A process requests access into the kernel or action by the kernel through a system call interface. The system call interface serves as the bridge between user level code and the kernel.

System calls are made like normal C function calls but get mapped to lower level routines which are actually in the kernel. How this is done is discussed in detail in Kernel Services module.

System calls interact with a particular area of the kernel. For example there are

- calls for file access through the file system (*open, read, write*)
- calls for process info and control (*kill, nice, getpid*)
- calls to allocate/deallocate memory (*mmap, mprotect, swapon*).

Once inside the kernel, individual subsystems may interact with other subsystems to complete the requested task. For example, a call to open a file will go through the file system to actually get the file but interact with the Process and Memory Management systems to allocate pages in memory for the file and to attach the file to the given process.

The list of system calls are found in `/usr/include/sys/scall_define.h`

(определены коды системных вызовов)

В HP-UX сист. вызов не TRAP, а
branch → gate (переходиме управление)
это много дешевле, это важно для
легковесных вызовов. (getppid и т.п.)

Module 1 — Overview

Example

The action of getting from user mode to kernel mode is fairly complex as we will later see but it can be shown quite simply. Copying a file involves reading from the source file and writing to the target one. Both reading and writing involve making systems calls. Using the **dd** command the size of the read and write operations and therefore their number can be varied. The following quick example demonstrates the time concerns with making system calls.

```
root@tiger[] dd if=/stand/vmunix of=/dev/null bs=64k # read the file into the buffer cache
```

```
235+1 records in  
235+1 records out
```

```
root@tiger[] timex dd if=/stand/vmunix of=/dev/null bs=64k
```

```
235+1 records in  
235+1 records out
```

```
real 0.10  
user 0.01  
sys 0.10
```

```
root@tiger[] timex dd if=/stand/vmunix of=/dev/null bs=64
```

```
241360+1 records in  
241360+1 records out
```

```
real 10.82  
user 1.57  
sys 7.80
```

Уз-за калібражних расхэгоў
іа аўтэнтывіе вэзубаў

From this simple experiment we can see that on this system (a C200) that each read or write system call takes about 20us which does not sound a lot but when large numbers of calls are being made it can soon add up.

The time command gives three values for the time, real which is the elapsed time, user and system. Both of these relate to the amount of CPU time used, the user time relates to the amount of time spent in user code, and the sys values is the time spent in kernel mode, running things like system calls, although the kernel might need to do work on our behalf at other times as well.

Module 1 — Overview

Left blank intentionally

Slide: Primary Subsystems

Primary Subsystems

- Process Management
 - Manage all running processes and threads on the system
- Memory Management
 - Virtual addressing
 - Each process has own virtual address space
- File Systems
 - Manage data on different types of disks
 - VNODE based system
- I/O
 - Define principles for device access

a69614

Notes:

Slide: Primary Subsystems

There are four primary subsystems covered in this course which are responsible for the majority of the kernel activity.

Process Management

This subsystem is responsible for

- managing all of the running processes and threads on the system
- maintaining information about process and thread resources
- scheduling threads on the cpu

Memory Management

Memory is managed by the kernel in a way that allows each process to have its own virtual address space. Traditionally, this address space is private to a particular process but HP-UX has evolved to the point where much of this address space is shared. Kernel threads alter the privacy of process data but the traditional view is as stated. Other memory resources are global to all processes.

All of these memory structures are managed in such a way that allows the sum of process address space to be greater than the amount of physical memory in the system. This is the theory behind the virtual address space concept.

File Systems

The file system is responsible for managing data on various types of disks.

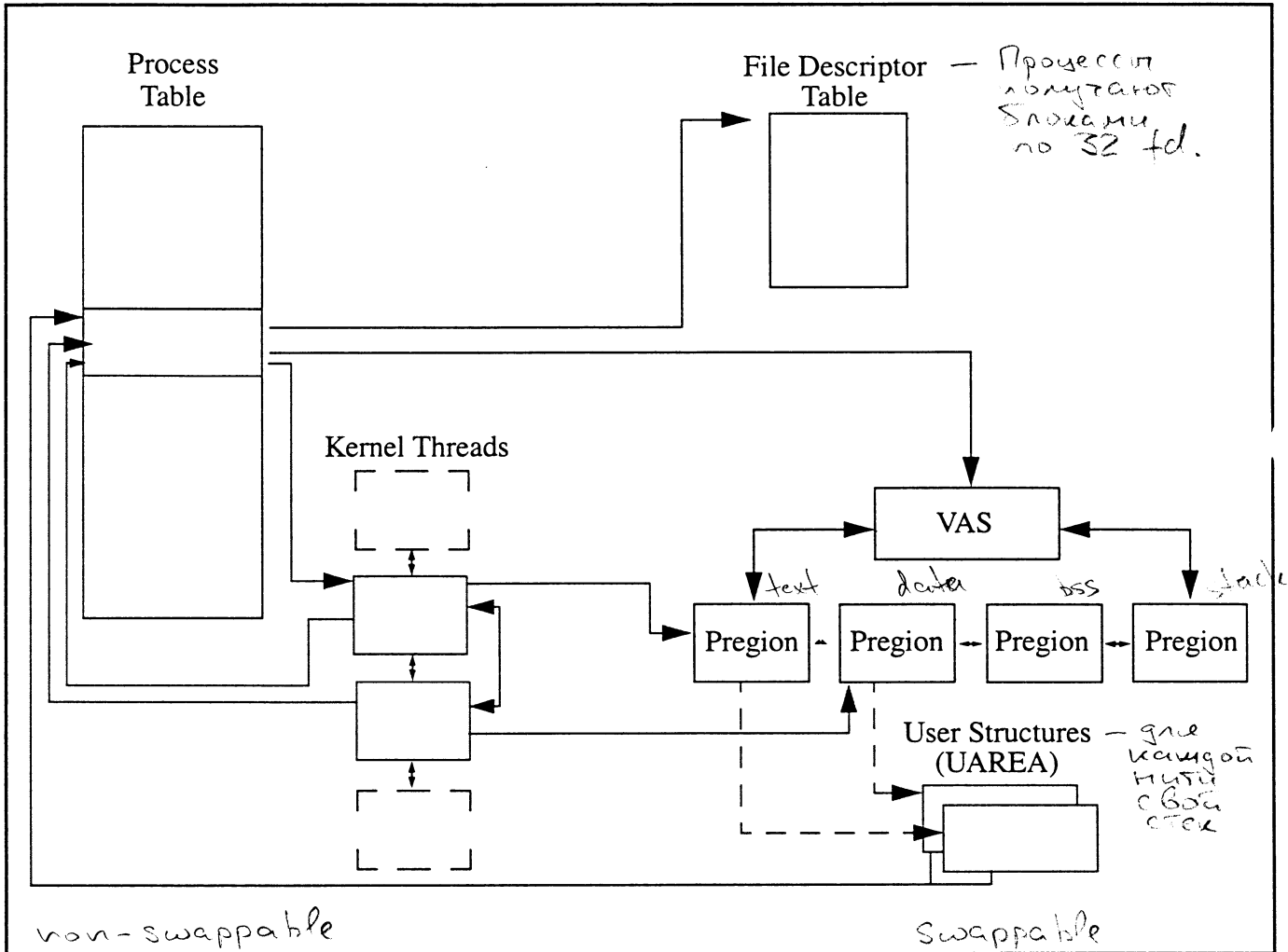
The file system has provisions to manage different types of file system disk layouts such as UFS, NFS, CDFS, VxFS. The ability to do this comes from a level of abstraction known as the vnode layer.

I/O

The I/O system is primarily made up of device drivers which are specifically designed for a particular type of device or interface. The I/O system defines general principles of how interrupt-driven devices can talk to the system and how to determine which driver is to be called for a given task.

In the I/O module we will be mostly concerned with the converged aspects of Workstation I/O (WSIO) and Server I/O (SIO).

Slide: Process Management Data Structures



Notes:

Ключевые образы и начисление
 ресурсов аппаратуры VA → PA

В UAREA конфигурируются параметры
 системных вызовов

Slide: Process Management Data Structures

The Unix kernel uses many data structures to describe and manage all the things happening on the system. Many of these data structures are organised into tables. As data structures, tables have many advantages for the programmer, for instance they can be treated as arrays and so any element within the table can be very easily and efficiently accessed.

Tables do however have a major drawback as data structures, when a table is created, it is created at a certain size, and can not normally then be resized dynamically. As a result of this most kernel tables have a sizing parameter to govern how large to make them. These sizing parameters are then kernel parameters which need to be set in the `/stand/system` file and are used to build the kernel. In the slide the process table has a kernel parameter `NPROC` to specify its size. At HP-UX 11 where process can have a variable number of threads then the threads are also described using a table, the `kthread` table and this is sized using the `NKTHREAD` kernel parameter.

Not all data structures within the kernel use tables, many data structures are dynamically allocated by the kernel when they are required and are then organised using a variety of techniques such as the linked lists shown with pre-queue structures on the slide. With a linked list one data structure in a set points to the next element. Whilst these dynamically created linked lists have the advantage of not having a fixed size, they can be much slower to access as they need to be accessed sequentially, and their organisation makes it difficult to optimise the data accesses in today's cache and TLB based CPUs (see the system architecture module).

All jobs running on the system run as a process. Each process on the system has an entry in the system **Process Table**. This table contains data that is shared by all of the kernel threads within a given process. Examples of fields in the process table include the process id, user and group id, and pointers to process' address space and open file descriptors.

Originally each process could only perform one task at a time, this is known as single threading. Many applications, however, benefit from being able to multitask. Whilst Unix always allowed multitasking, it was at the process level, the kernel scheduled the various processes, but each individual process was then, at least as far as the kernel was concerned only single tasking. IE the kernel only scheduled one part of it and only in respect to one event.

From HP-UX 9 a pthread library has been available for use with HP-UX, this allowed processes to multitask internally, but this did not operate at the kernel level so if one part of the process were to go to sleep, then the whole process would sleep. Since this implementation is not managed by the kernel it is known as **user threads**.

In order for the kernel to manage and schedule multiple tasks (known as threads) within a process its data structures need to be re-organised to separate out the information that is global to all the threads within a process from the scheduling information about the individual threads. This re-organisation starts at HP-UX 10.10, but for both 10.10 and 10.20 the multi-threading facility is not available for use, with these releases there will be one thread per process.

Once the kernel is re-organised into separate structures for processes and threads, and the thread structures now contain the scheduling information, we consider that it is the threads that now run and not the processes, processes if you like become a container for the threads.

Slide: Process Management Data Structures

With HP-UX 11, however each process may have multiple threads and so true multitasking is possible within a single process.

Each process may have one or more kernel threads. Each thread is a uniquely scheduled entity with its own Kernel and User stacks. Thread-specific information such as scheduling and related attributes (i.e. priority, cpu usage, and thread state) are held in the **Thread Table**.

The **User Structure**, also known as the **UAREA**, contains information unique to the individual thread that is swappable. The user structure contains thread information such as the Process Control Block(PCB), and system call information as well as the Kernel stack.

The processes and thread tables are held permanently in the RAM since the kernel might require access to any of the information they contain at any time. Some information is only required whilst a thread is actually running (such as information about the current system call) or when it is about to be scheduled or de-scheduled (such as the Process Control Block, which holds the CPU register contents whilst not actually running). Since this information is only required during, or near, running, and not at other times it would be possible to push it out of memory if the thread was not going to be running and there was a major shortage of memory. Hence the thread information is divided into these two separate areas, the thread table (kthread structure) which is permanently memory resident, and the UAREA (user structure) which can be swapped out.

Each process has a 4GB or 16TB virtual address space. Obviously not all of this is going to be in use. In reality processes only tend to use a small amount of their virtual address space but they use a number of discreet pieces in the overall range. The kernel needs to describe each of these individual pieces.

To describe the **virtual address space (VAS)** for a process, a pointer in the process table links the entry to the **VAS Structure** for the process. The VAS is the head of a linked list of **pregions**. Together the VAS and the **pregions** represent the virtual address space for the process. Each **pregion** will represent a different type or piece of address space, such as stack, data, or text.

Open files for the processes are maintained in the **File Descriptor Table**. This table contains one entry for each open the process performs. Since there can be a wide variation in the number of file descriptors different process might use, a simple table structure would either be very limiting, or very inefficient (tables have a fixed size, make it small and it would limit those programs that would like a large number of open files, make it large to accommodate them and it would be very wasteful for the majority of process that do not need many files open). So a two stage table is used, where an initial table just holds pointers to the second level, where entries are created as needed, each holding information about up to 32 file descriptors.

Instructors Notes

The following quick walk through a few of the kernel data structures in q4 is entirely optional, It would probably depend upon the groups of students.

The reason for putting these examples here, is that this is the one place in the course where a set of kernel road maps appear and I wanted the instructions to live with the road maps. However only this slide has accompanying q4 instructions in this module. The others might require a little too much theory or examples.

For the example of reading the file descriptor information you need to be very careful between 32bit kernels and 64bit kernels.

This example came from a C200 running a 64bit kernel.

WARNING: The version of q4 that ships with HP-UX 11 9808, q4 1.79a, attempts to pxdb the kernel file if it has not previously been prepared. At least on the 64bit kernel this does not appear to work and can leave the kernel file unusable.

Instructors Notes

Example

Viewing the kernel data structures

The kernel crash dump debugger **q4** can be used to view the kernel data structures.

анализатор CRASH для Solaris и Sun SCD

NOTE: q4 is primarily intended for use by Hewlett-Packard personnel and is **not supported**, however it does allow us to view the various kernel structures discussed in this class.

In order to use q4 it might be necessary to prepare your kernel file /stand/vmunix. This can be performed without the need to reboot the system. The command to prepare the kernel is /usr/contrib/bin/q4pxdb.

All the various data structures discussed will be described in later modules in the class. This example is just to show that these structures can be viewed.

```
root@tigger[] q4pxdb /stand/vmunix
q4pxdb64: /stand/vmunix is already preprocessed
PXDB aborted.
root@tigger[] ied -h -/.q4_history q4 /stand/vmunix /dev/mem
@(#) q4 $Revision: 1.79a $ $Date: 97/09/08 12:00:22 $ 0
q4: (warning) no modules in the crashdump or no INDEX file
q4: (warning) q4 will try to read /dev/kmem for kernel access
Reading kernel symbols ...
Reading kernel data types ...
/dev/mem: can't validate: expected size or checksum not available.
Dump data may not be correct.
Initialized PA-RISC 2.0 address translator ...
Initializing stack tracer ...
Get the latest q4 news by typing "news".
```

prepare the kernel file

ied is just an input editor

добавляет в модуль программы регистры командной строки.

q4 is designed primarily to read crash dumps

Example

Accessing the process table

```
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
```

load structures of type
struct proc from the pointer
also call proc (which points to
the process table) and nproc
entries.

```
q4> keep p_stat != 0
kept 73 of 276 struct proc's, discarded 203
```

The process table is always
nproc entries in size,
regardless of how many
processes are actual running.
This just keeps the entries
in use.

```
q4>
q4> print p_pid p_ppid p_uid p_comm | more
  0  0  0 "swapper"
  1  0  0 "init"
  2  0  0 "vhand"
  3  0  0 "statdaemon"
  4  0  0 "unhashdaemon"
...
1516 1  0 "cron"
1874 1873 0 "sh"
1987 1986 0 "q4"
944 1  0 "inetd"
...
q4> keep p_pid == 1874
kept 1 of 73 struct proc's, discarded 72
q4>
q4>
```

Printout the process id,
parent process id,
user id and command name

Just keep the entry for my sh

Example

Accessing the pregon list

```
q4>
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4> load struct pregon from va_ll.lle_prev next p_ll.lle_prev max 1000
loaded 7 struct pregon's as a linked list (stopped by loop)
q4>
```

- Т.к. pregon в
списке, а не в
таблице

The VAS structure can be accessed via the pointer p_vas from a process table entry

NOTE: q4 can only load from a pointer in a structure when a single structure is loaded, hence the keep p_pid == ... operation to get down to a single process table entry.

Once the VAS structure is loaded then the pregon list can be accessed via the va_ll.lle_prev field. Normally q4 only loads single structures, as with example of loading the vas structure above. Q4 can also load sets of structures if a max clause is given, by default it then loads the structures as a table, but linked lists can also be loaded by specifying which field in the new structure provides the link to the next structure.

$32 + 32 = 64$ Space + vaddr

```
q4> print p_type p_count %5d p_space %#8x p_vaddr %#8x
p_type p_count p_space p_vaddr
PT_UAREA 7 0xdc23c00 0xffffce000
PT_STACK 6 0xafd2000 0x7f7e6000
PT_MMAP 1 0xafd2000 0x7f7e5000
PT_DATA 39 0xafd2000 0x40001000
PT_TEXT 80 0xbc1f800 0x1000
PT_NULLDREF 1 0xbc1f800 0
0 12240896 0x1 0xbac800
q4>
```

Pregons, as we will see in the memory management module, come in different types, to hold different areas of the address space such as the text area of data area. The **p_count** field gives the overall size of the area of address space, and each area has an address, on HP-UX this is given in two parts the **p_vaddr** field is the normal part of the address, but HP-UX has the concept of global virtual addresses and the **p_space** part is location with this global virtual address space.

Q4's print command allows the use of C style formats after a field, **%#8x**, tells q4 to print the value in hex indicated with the normal **0x** prefix and have at least 8 characters.

The last line is effectively garbage, the vas structure is in the linked list, but is not a pregon, q4 can't know this, and had attempted to load a pregon from the area of memory that is holding the vas structure.

forget - забыть номерную результат q4
и вернуться на шаг назад

Example

Accessing the thread structures

The linked list of threads is referenced from the process table as well.

```
q4> load struct kthread from p_firstthreadp next kt_link
loaded 1 struct kthread as a linked list (stopped by max count)
q4>
```

Threads are described by the kthread structure, and the process points to the list using the fields `p_firstthreadp` and also `p_lastthreadp`. The threads are then linked together using the fields `kt_link` and `kt_rlink` (for the reverse list).

```
q4> print kt_pri kt_wchan %#x kt_stat
kt_pri kt_wchan kt_stat
 670 0xf6f9c0 TSSLEEP      7.e. fusb 0x45
q4>
```

Many of the fields that are normally viewed using `ps` can be seen from the process and thread structures, such as the current priority (`ps` shows this value less 512), wait channel and status.

Example

Accessing the Uarea

The uarea belongs to the thread but does not live in the kernel's main data virtual address space. Remember that the uarea can be swapped out, and so is described by a pregon.

```
q4> load struct pregon from kt_upreg
loaded 1 struct pregon as an array (stopped by max count)
q4> load struct user from tospace(p_space) | p_vaddr
loaded 1 struct user as an array (stopped by max count)
q4>
```

From the kthread structure the fields `kt_upreg` points to the pregon that describes the uarea. Where structures need to be loaded from full global virtual addresses then the `conststruct`

```
tospace(p_space) | p_vaddr — c080pam bypr. aggp.
```

is used.

```
q4> print u_syscall u_arg[0] u_arg[1] %#x u_arg[2] %#x
u_syscall u_arg[0] u_arg[1] u_arg[2]
200 -1 0x7f7e6954 0x2
q4>
```

As an example of information that might be found in the uarea, let's print out information about the current system call.

The field `u_syscall` gives the system call number that is used to identify which call is being made. The header file `/usr/include/sys/scall_define.h` mentioned previously lists all of the calls and their numbers. System call 200 is the `waitpid` call.

The array `u_arg`, holds the arguments for the system call. The manual page for the system call gives details of these.

```
root@tiger[] man 2 waitpid
```

```
wait(2)                                wait(2)

NAME
    wait, waitpid - wait for child process to stop or terminate
```

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *stat_loc);
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Since our three arguments were: -

```
q4> print u_syscall u_arg[0] u_arg[1] %#x u_arg[2] %#x
u_syscall u_arg[0] u_arg[1] u_arg[2]
```


Example

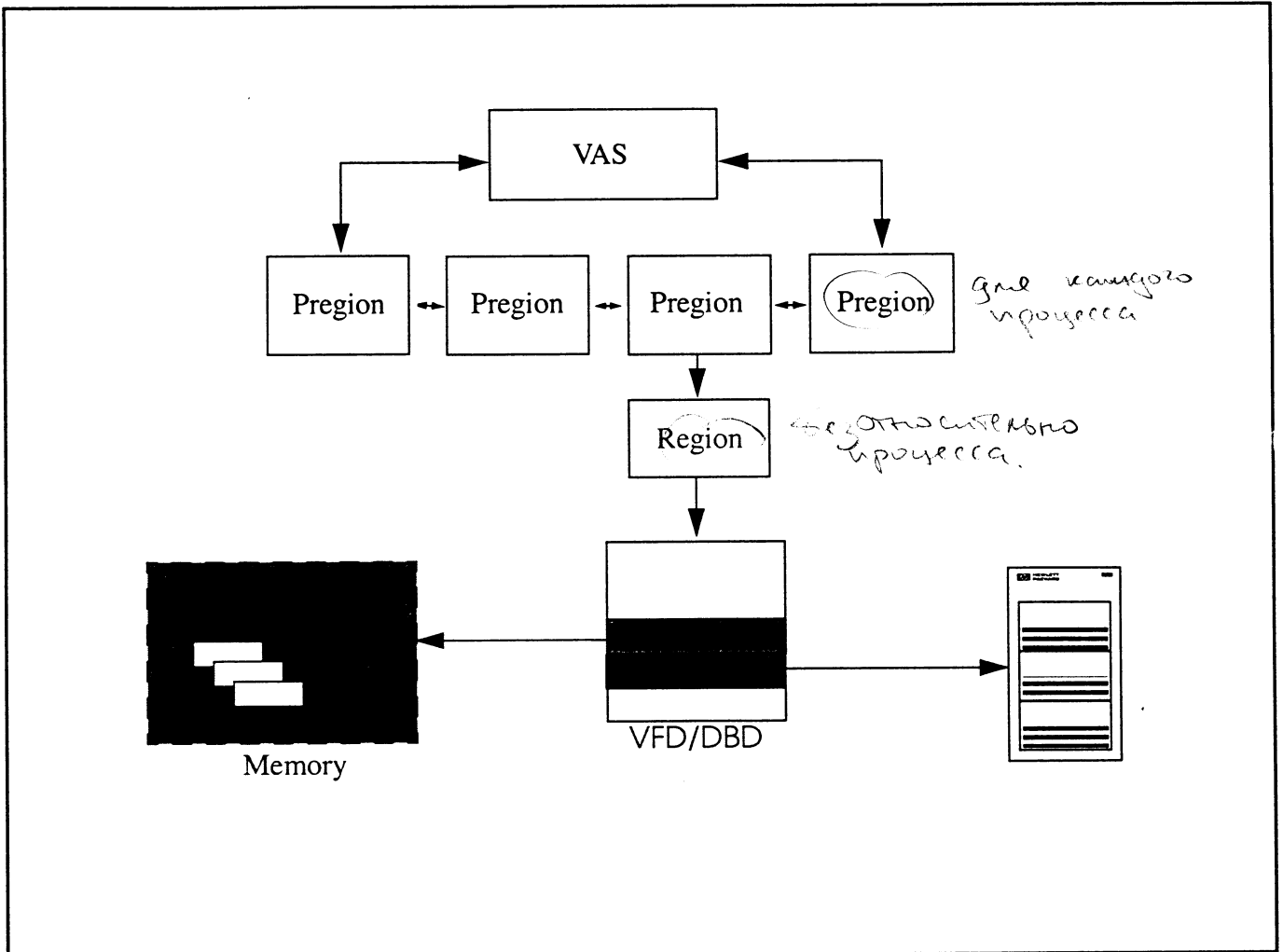
```
200  -1 0x7f7e6954  0x2
q4>
```

The pid that the shell was waiting for was -1, which indicates any process.

The status should be saved somewhere on the stack, at address 0x7f7e6954.

The options were 2, which would require reading the `<sys/wait.h>` header file. It actual means that the shell would like to know when its children stop or suspend as well as when they terminate.

Slide: Memory Management Data Structures



Notes:

Slide: Memory Management Data Structures

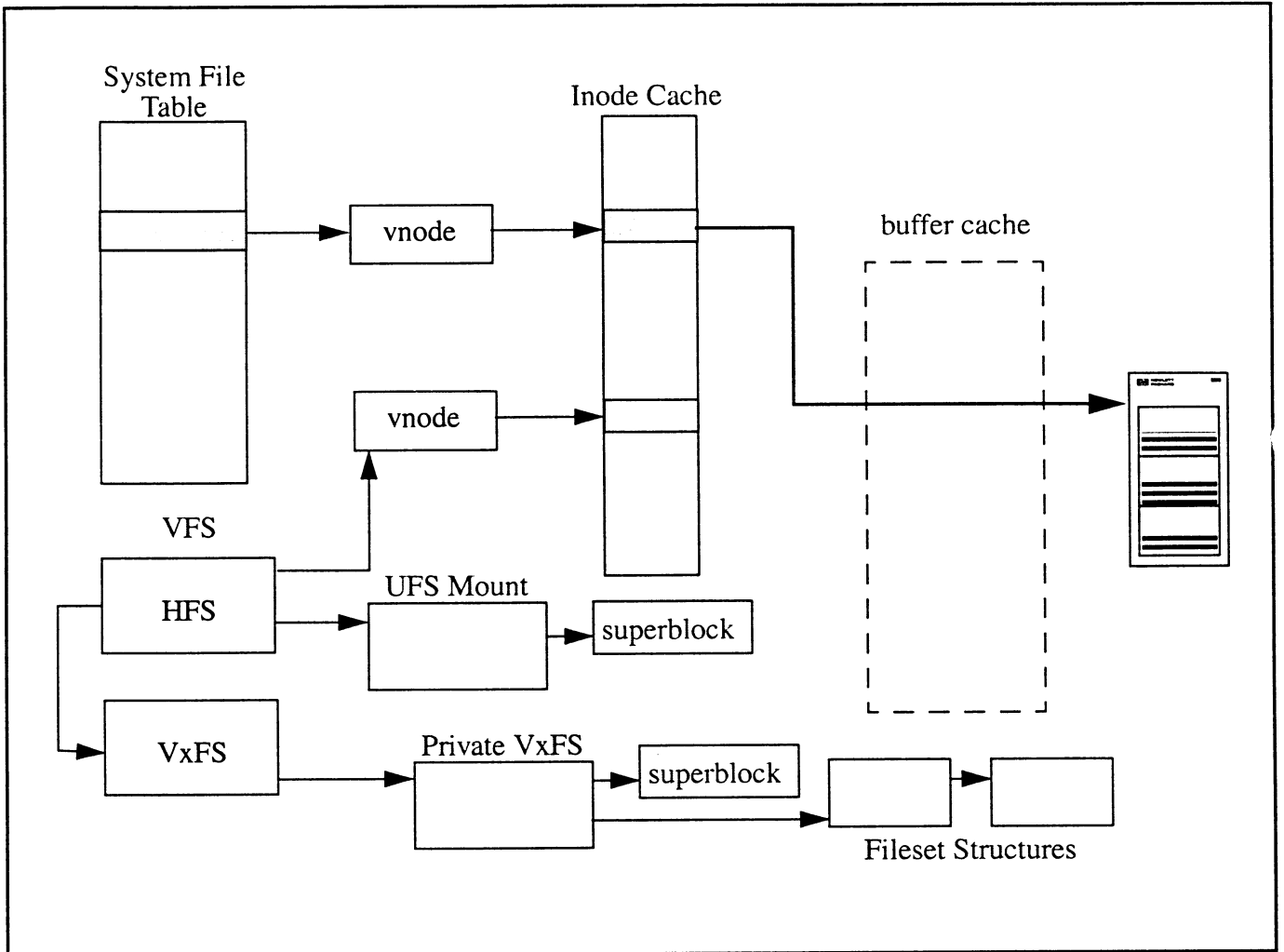
The memory management data structures pick up at the VAS/region level. This view is however 'private' to the process. Many areas of memory however can be shared between different process. For instance the text area (the machine code instructions) of a process is shared with each of the other processes running the same program. So as well as having a process private view of the memory areas, the kernel also requires a system wide view, which is provided using the region structure.

Each region is linked to a **region** that has further information about access to a particular range of addresses within an address space. Below the region, a **Virtual Frame Descriptor** (VFD) describes a page in that address space that is currently in memory. If a page is not in memory, then a **Disk Block Descriptor** (DBD) describes the location of the pages on disk. There exists a VFD/DBD pair for each page of the region which indicates whether the page is in memory or on disk and where in memory or on disk the page can be found.

Since the number of pages in a region can vary widely, even with 32bit applications, the number of pages in a single region can vary from 1 to nearly 1/2 a million. A highly flexible data structure needs to be used to manage these page level structures (the VFD and DBDs). A further complication is that the current set of pages described by the VFDs and DBDs can be sparse; that is, it can contain holes. So the mapping of the regions down to the VFD/DBD pairs needs to be able to cope efficiently with a large number of different situations. The data structure that is used here as we will see in the memory management part of the class is a **BTREE**.

Module 1 — Overview

Slide: File System Data Structures



Notes:

give VxFS the *uwp* *←* *inode*

Slide: File System Data Structures

The file system maintains a **System File Table** which contains an entry for every open that is performed on the system. A system file table entry points to a **vnode** for the file which has data specific to the type of file (NFS, device file, regular file). In the case of a regular file, the vnode points to an entry in the **inode cache**. The inode cache contains one entry for every file that is opened on the system and contains access and disk location information about the file. The **buffer cache** serves as an intermediate area between disk and processes to provide buffering and synchronization.

Each file system type has a set of data structures specifically related to how data is represented within the file system. To allow the kernel to access all file system types in a generic fashion, there is a **Virtual File System (VFS)** structure present for each mounted file system. This VFS structure will point to the private data for the file system represented. In the case of a UFS file system, the VFS will link to a mount entry. For VxFS it will be a VxFS structure.

The file descriptors we have seen with the process table reference entries in the '**System File Table**'. As with any table the system file table needs a sizing parameter, which is **NFILE**. Every open on the system allocates an entry from the system file table. The first job for these entries is to act as a switch between file and network access. The Berkeley sockets API uses file descriptors in much the same way as normal Unix file access. For files the file table entry contains information relevant to the current usage of the file such as the offset within the file where read and write operations will be performed.

The file table also points to the next data structure used in file access the **vnode**. It is possible for the same file to be open many times by different processes, and each open will result in using a slot in the file table. With vnodes however there will only ever be a single vnode for a file in the kernel.

The vnode was introduced as a data structure into the Unix kernel by Sun Microsystems to allow the support of multiple filesystem types, originally both UFS and NFS. The vnode *layer* acts as a switch allowing any software at a higher level, particularly the user level, to not have to know what type of filesystem they are accessing. A single read function can be used on all types of filesystem, there is no need to have an UFS_read, an NFS_read and a VxFS_read function inside your programs. When the access reaches the vnode the kernel is directed to correct routines, thus allowing higher level code to not need to worry about the issue.

The vnodes major job then is to act as the switch between different filesystem types. It is also used to hold any information about the file that is not reliant on its type, such as pointers to any entries in the buffer cache associated with the file.

The vnode then points to filesystem specific data structures:

- For UFS the vnode references an **inode** structure within the incore inode table (also know as the inode cache). Since this is a table the **NINODE** kernel parameter is needed to control its size. It is worth stressing that this table is used only for UFS filesystem and not for NFS or VxFS.
- For NFS the vnode references an **rnode** structure.
- For VxFS the vnode references a **vx_inode** structure. Vx_inodes are not held within a table and are dynamically allocated.

Slide: File System Data Structures

As well as the need to manage access for individual files the kernel also needs to manage the mounted file systems. Again, here there is the issue of having several varieties of file system to chose from so the kernel makes use of a data structure called a **vfs**, or virtual file system, to act as the switch between the different types.

The **vfs** structures are held on a linked list pointed to by **rootvfs**. The **vfs** structures then reference filesystem specific structures: -

- UFS, uses mount structures,
- NFS, uses mntinfo structures,
- VxFS, uses vx_vfs structures.

These structure will then need to reference further file system specific structures for the individual file system types. For the local file system these will include copies of the file system superblocks. For VxFS there will be a list of fileset structures.

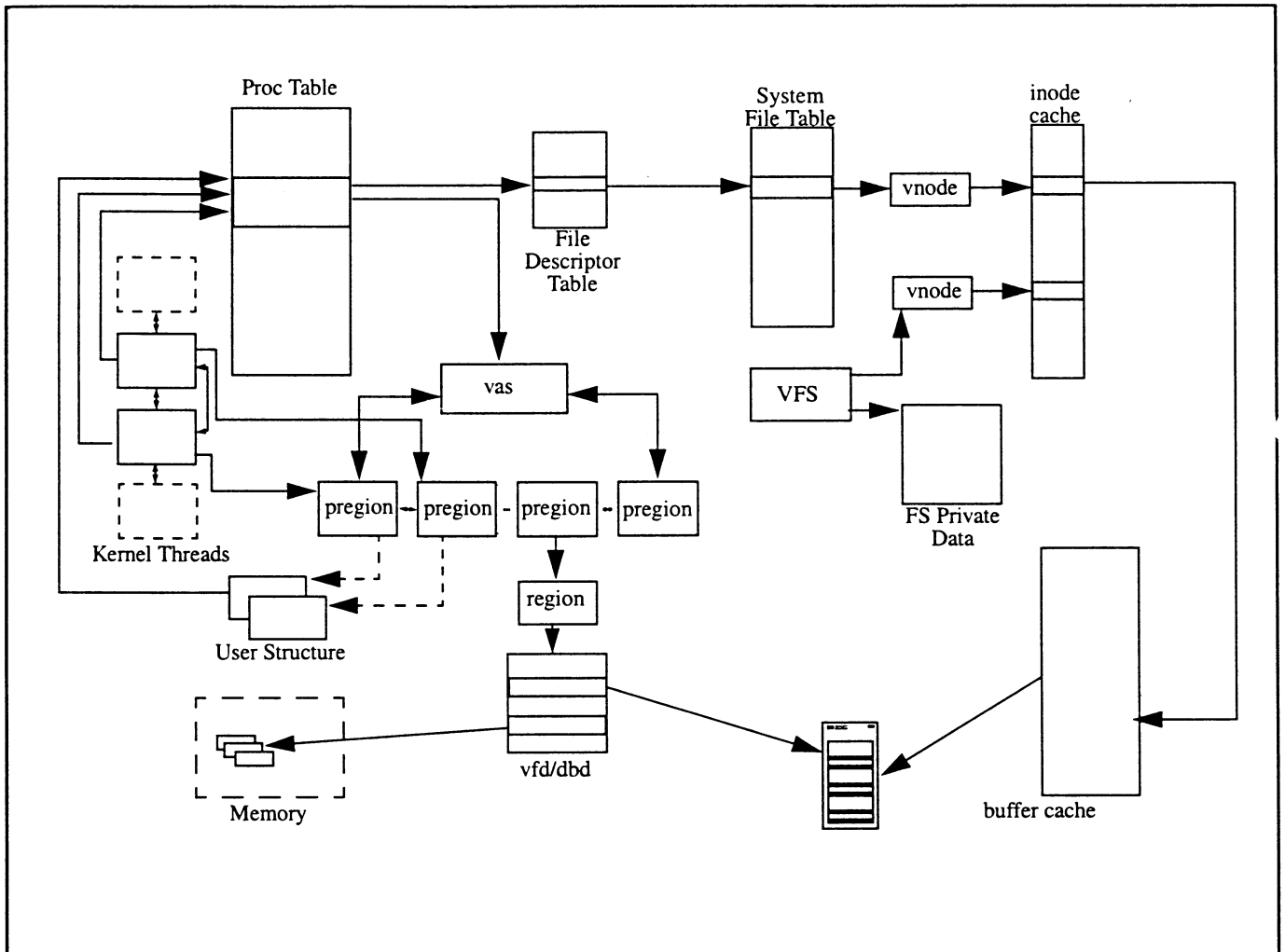
Module 1 — Overview

Left blank intentionally

22

Module 1 — Overview

Slide: The Big Picture



Notes:

Slide: The Big Picture

This slide shows the “Big Picture” of some of the primary kernel data structures. Previous slides have already shown how the process management and memory structures are linked together through the VAS and pregion. Here we also see how the process management structures are linked to the file system structures through the File Descriptor Table. Each entry in this table is mapped to an entry in the System File Table. Once in the System File Table we can complete the picture of the process through the file system tables.

Each of these structures will be discussed in greater detail in the individual modules of this course.

Slide: New Features of HP-UX 11.0

New Features of HP-UX 11.00

- PA-RISC 2.0 (64-bit) Architecture
- Kernel Threads
- DLKM
- Memory Windowing
- Variable-sized Pages — т.к. VM может быть очень большой

но у нас в 11.0 — 16кб (4к = 64М)
в 7000 PA-RISC — 4кб

но может быть какой-то
другой способ, например chatr

a69619

Notes:

в одном процессе до 4к до 64М на ар.
кэш (TLB) трансляции вирт. в физич. адр.
— маленькое АЗУ

Slide: New Features of HP-UX 11.0

Among some of the new features of HP-UX 11.0 discussed in this class are:

PA-RISC 2.0 Architecture

HP-UX 11.0 allows support of existing systems using PA-RISC 1.1 architecture (the 32-bit PA-7X00 family of processor), and PA-RISC 2.0 architecture (the 64-bit PA-8X00 family of processors). PA-RISC 2.0 is capable of running in both narrow (32-bit) and wide (64-bit) mode.

Kernel Threads

HP-UX 11.0 supports multiple kernel threads per process. Each thread is a unique execution of the program and its own scheduling entity.

Dynamically Loadable Kernel Modules (DLKM) — *временные модули ядра системы*

Allows for addition and removal of kernel modules, kernel module administration, and inactive kernel modules.

Memory Windowing

Allow more overall shared memory space by allowing various applications their own shared memory “window” of 1GB.

Variable-sized Pages

PA-RISC 2.0 supports variable-sized pages versus the static 4K physical page size used by PA-RISC 1.1.

Module 1 — Overview

Left blank intentionally

Module 2

System Architecture

“At the source of every error which is blamed on the computer you will find at least two human errors, including the error of blaming it on the computer.”

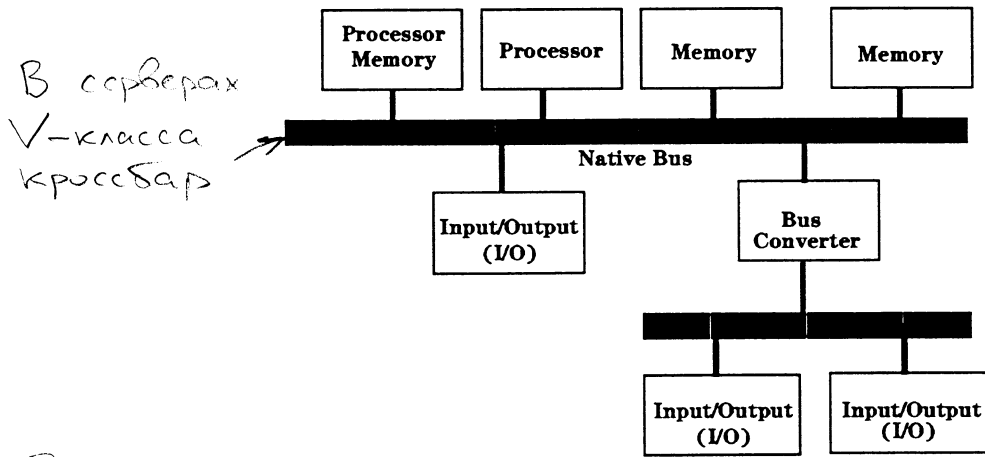
-- Unknown

Objectives :

- Understand PA-RISC 1.1 and 2.0 concepts of virtual addressing
- Understand the role of the TLB and Cache in accessing memory pages
- Understand levels of interruptions and how they are handled by PA-RISC hardware

Slide: PA-RISC Architecture Overview

PA-RISC Architecture Overview



В серверах
V-класса
кроссбар

Процессор.
CC: NUMA

(S, X-классы)

a69620

Notes:

Slide: PA-RISC Architecture Overview

When we talk about Computer Architecture, we are discussing the ideas and models for a computer system. We are not concerned with specific hardware differences among various PA-RISC 1.1 or 2.0 systems or processors, but rather the functional concepts that make them alike. For reference, there are tables at the end of this module that list some of the hardware differences.

PA-RISC is distinguished from other computer architectures by two important features:

- Reduced Instruction Set Architecture - PA-RISC systems are composed of simple, frequently used instructions each of which are designed to complete in one machine cycle. This allows for higher performance and simplified hardware design.
- Architectural Extensions - This includes features such as virtual memory, memory mapped I/O, support for multiprocessors and coprocessors, and instruction pipelining.

There are three primary subsystems in the PA-RISC design:

- Processor
- Memory
- Input/Output(I/O)

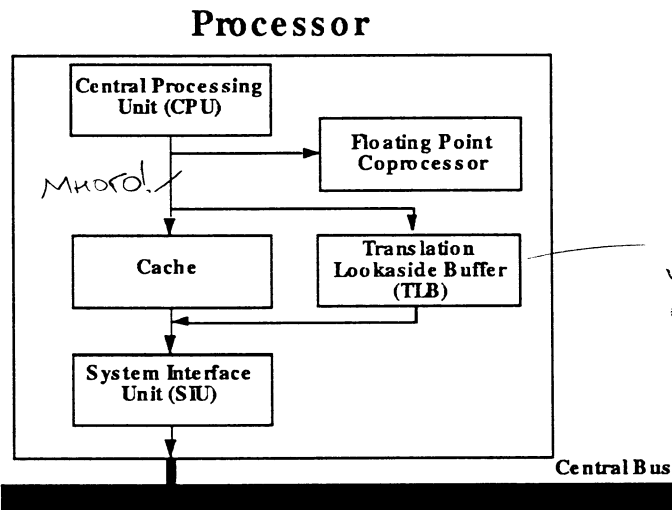
The slide shows a generic design for a PA-RISC system with several modules. **Modules** are hardware components of the PA-RISC system with a specialized function. Each of these modules are connected via different types and levels of buses.

A bus that follows the specifications of the PA-RISC I/O architecture is known as a **native bus**. Non-native buses require specialized hardware to convert them to the PA-RISC specifications.

The information in this module is based on both PA-RISC 1.1 and PA-RISC 2.0, which supports both 64-bit (Wide Mode) and 32-bit addressing (Narrow Mode). When necessary these architectures are presented on sequential slides. Examples are based on the PCX-U processor.

Slide: Processor Architecture

Processor Architecture



a69621

Notes:

Parue FPU u cache totu asozovno

Slide: Processor Architecture

In this module, we are primarily concerned with the architecture of the processor and its interaction with main memory. The slide shows a common processor and its major components.

Central Processing Unit

The key component in any processor is the Central Processing Unit (CPU). The CPU is the component which is given the primary task of reading program and data from memory, and executing the program instructions. Within the CPU there is:

- **Control Hardware** which coordinates the activity of the CPU by carrying out the fetch and decode of instructions to generate control signals for appropriate CPU hardware.
- **Execution Hardware** to perform the actual arithmetic, logic, and shift operations. Execution Hardware can take on many specialized tasks but most common are the **Arithmetic and Logic Unit (ALU)** and the **Shift Merge Unit (SMU)**.
- **Registers** which are held in very fast memory within the processor. This memory is much faster than conventional main memory but it is also much more expensive. For that reason this small amount of memory is partitioned off for specific purposes. The PA-RISC register context will be discussed in detail shortly.

Instruction and Data Cache

The **cache** is also a portion of high speed memory intended to reduce the amount of time needed for the CPU to access data and instructions. This is accomplished by keeping the most recently accessed data in the cache. All data going to the CPU from main memory passes through the cache first.

When the CPU requires data from main memory, it first checks the cache for the requested data. If the data is present in the cache, a **cache hit** occurs and the data is sent to the CPU. If the data does not exist in cache, then a **cache miss** occurs and the CPU must wait while the data is brought into the cache from main memory.

For some PA-RISC systems, the CPU may have separate cache for data and instructions. By doing this we are able to obtain better locality of data and instructions within the cache, thus increasing performance as a result of higher hit rates.

To process a piece of data it needs to be held within a CPU register, if it is not, it needs to be loaded from outside, when the data can come from cache, main memory or the disk.

Module 2 — System Architecture

Slide: Processor Architecture

	100MHz K-Series		440Mhz CPU V-class	
location	cycles	time	cycles	time
register	0	0	0	0
cache	1	10ns	1	2.27ns
memory	25	250ns	220	500ns
disk	1,000,000	10ms	4,400,000	10ms

oombua?

From this quick comparison we can see that as CPUs get faster (and memory does not, correspondingly) the role of the cache in supplying data becomes more important.

Translation Lookaside Buffer (TLB)

The CPU is performing all data access through virtual addresses. The **Translation Lookaside Buffer (TLB)** serves two purposes for the CPU:

1. Translate the virtual address to physical address.
2. Check access rights to grant access to instructions, data, or I/O only if the requesting process has proper authorization.

The concepts of virtual address, virtual address translation, and access rights will all be discussed in this module. They are mentioned here to give an overview of the primary tasks for each CPU component.

Assist Processors and Bus Circuitry

Completing the CPU are components which will not be dealt with in detail in this course. The remaining components are the **Assist Processors** or **Coprocessors** which exist to carry out specialized tasks for the CPU. A common example present in most PA-RISC systems is a Floating Point Coprocessor.

Also a part of the CPU is the **System Interface Unit (SIU)** which is the bus circuitry that allows the CPU to communicate on the Central/Native bus.

Cache Coherence — no separate cache for CPU, no a ID!

Because we may have multiple processors, each with their own cache, there needs to be a way to handle cache coherence.¹ There is special hardware in the cache controller that exists to control the cache consistency.

In a uniprocessor system, every load or store will cause the CPU to ask the cache controller whether a cache line is in the cache and what its state is. Depending on whether it is in the cache (and whether it is

1. Issues related to cache coherence will be discussed in Module 6 - Multi Processing

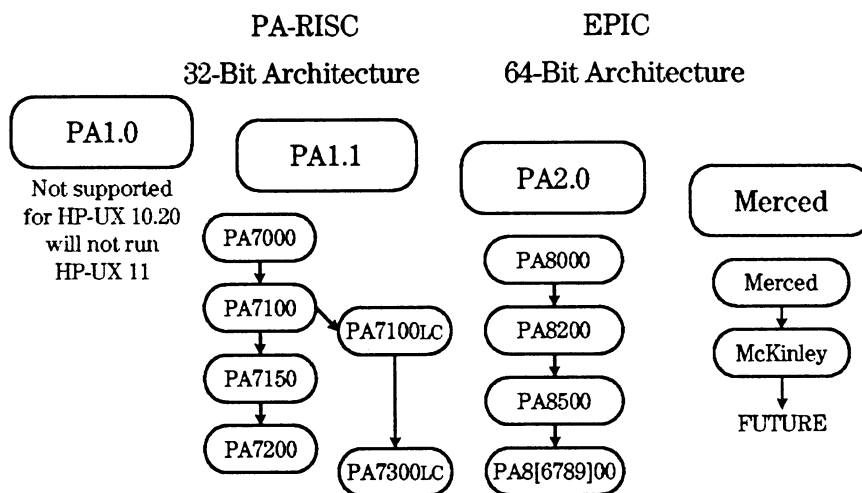
Slide: Processor Architecture

dirty or clean), the cache controller will go out on the bus and issue a transaction or just manipulate the cache directly (read the value out of the cache or store a new value in the cache).

In a multi-processor system, in addition to loads and stores from within the processor, the cache controller must watch transactions on the bus to see if those transactions are attempting access to lines which are in this processor's cache. Again, the subsequent behavior will depend on the whether the line is present and whether it is clean or dirty.

Slide: PA-RISC Processor Versions

PA-RISC Processor Versions



a69622

Notes:

Module 2 — System Architecture

Slide: PA-RISC Processor Versions

From its inception in the early 1980s there have been many versions of PA-RISC processors from the earliest used on the 840, which was built from discrete TTL chips through to the latest PA8500 used in the V2500 and C360 systems, which contains over 140,000,000 transistors.

The initial versions PA1.0 used on the

840

825,835,845

850,855,860,865,870/X00

815,808

822,832,842,852

890

Systems was obsoleted from HP-UX support from 10.20 and HP-UX 11 is compiled such that it will not boot on these systems.

With the advent of the firstly the 700 series and then the Nova 800 series the first major revision of PA was released, PA1.1

There have been a number of versions of PA1.1 processors

Version	internal name	external name	Clock Speed MHz	
PA1.1a	PCX-S	PA7000	33,50,66	720,730,750,710,705 807-877(FGHI 10-40)
PA1.1b	PCX-T	PA7100	33,50,75,90,99	715,725,735,755 887,897(GHI50-70), T500
		PA7150	120,125	735,755 T520
PA1.1c	PCX-L	PA7100LC	48,60,80,100	712,715,725 Es, D
PA1.1d	PCX-T'	PA7200	100,120	J,C Kx[012]0,D
PA1.1e	PCX-L2	PA7300LC	133,180	B A,D

Slide: PA-RISC Processor Versions

All of the PA1.X processors are basically 32-bit designs, in that their general registers are all 32 bits wide, but they are, as we shall see, capable of using larger virtual addresses. In the case of PA1.1 they also have 64-bit floating point units, with full 64-bit data paths to main memory.

PA2.0 then moves to the new 64-bit architecture. These processors are supported firstly under HP-UX 10.20, but since this is 32-bit operating system many of the new processors capabilities are not yet usable.

PA-RISC 2.0

PA-RISC 2.0 represents the first time that user-visible changes have been made to the core integer architecture. The following areas have been added or enhanced for PA-RISC 2.0.

64-Bit Extensions

ногдогодого обратора обмектуносту.

PA-RISC 1.x supported a style of 64-bit addressing known as “segmented” addressing. In this style, many of the benefits of 64-bit addressing were obtained without requiring the integer database to be larger than 32 bits. However, this did not easily provide the simplest programming model for single data objects (mapped files or arrays) larger than 4 GB.

Support for such objects calls for larger than 32-bit “flat” addressing, that is, pointers longer than 32 bits which can be the subject of larger than 32-bit indexing operations. PA-RISC 2.0 provides full 64-bit support with 64-bit registers and data paths. Most operations use 64-bit data operands and the architecture provides a flat 64-bit virtual address space.

Cache Prefetching

огробр. вобсер. 4 операциу
уз 56 вобсер. 4x, операциу
костроуу вобсер. 10 операциу

Because processor clock rates are increasing faster than main memory speeds, modern pipelined processors become more and more dependent upon caches to reduce the average latency of memory accesses. However, caches are only effective to the extent that they are able to anticipate the data and consequent processor stall while waiting for the required data or instruction to be obtained from the much slower main memory.

The key to reducing such effects is to allow optimizing compilers to communicate what they know (or suspect) about a program’s future behavior far enough in advance to eliminate or reduce the “surprise” penalties. PA-RISC 2.0 integrates a mechanism that supports encoding of cache prefetching opportunities in the instruction stream to permit significant reduction of these penalties.

Branch Prediction

ВКГ - таблица завокувануе
переходов: 2000

A “surprise” also occurs when a conditional branch is mispredicted. In this case, even if the branch target is already in the cache, the falsely predicted instructions already in the pipeline must be discarded. In a typical high-speed superscalar processor, this might result in a lost opportunity to execute more than a dozen instructions.

PA-RISC 2.0 contains several features that help compilers signal future data and likely instruction needs to the hardware. An implementation may use this information to anticipate data needs or to predict branches more successfully, thus avoiding the performance penalties.

HP-UX 10.20
10.20.0

Slide: PA-RISC Processor Versions

Memory Ordering — *нормально укажать, нужно ли на самом деле упорядочивать байты.*

When cache misses cannot be avoided, it is important to reduce the resultant latencies. The PA-RISC 1.x architecture specified that all loads and stores be performed “in order,” a characteristic known as “strong ordering.”

Future processors are expected to support multiple outstanding cache misses while simultaneously performing loads and stores to lines already in the cache. In most cases this effective reordering of loads and stores causes no inconsistency, and permits faster execution. The later model is known as “weak ordering,” and is intended to become the default model in future machines.

Of course, strongly ordered variants of loads and stores must be defined to handle contexts in which ordering must be preserved. This need for strong ordering is mainly related to synchronization among processors or with I/O activities.

Coherent I/O

As the popularity and pervasiveness of multiprocessor systems increase, the traditional PA-RISC model of I/O transfers to and from memory without cache coherence checks has become less advantageous. Multiprocessor systems require that processors support cache coherence protocols. By adding similar support to the I/O subsystem, the need to flush caches before and/or after each I/O transfer can be eliminated. As disk and network bandwidths increase, there is increasing motivation to move to such a cache coherent I/O model. The incremental impact on the processor is small and is supported in PA-RISC 2.0.

Multimedia Extensions — *SIMD*

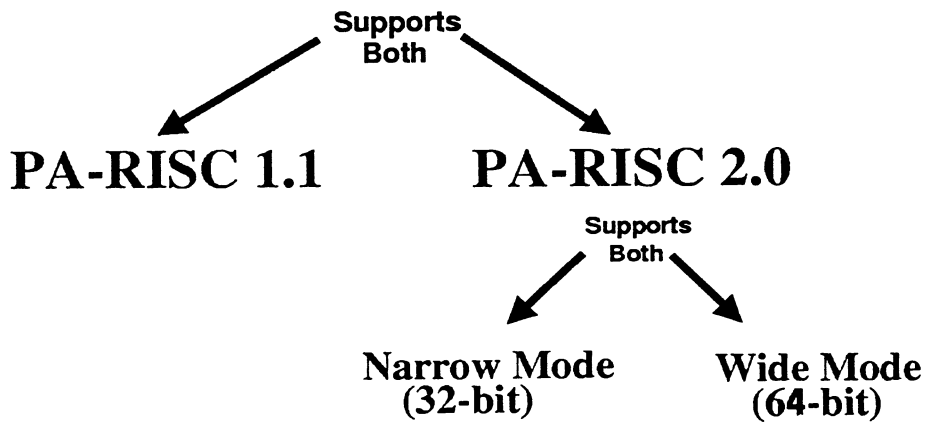
PA-RISC 2.0 contains a number of features which extend the arithmetic and logical capabilities of PA-RISC to support parallel operations on multiple 16-bit subunits of a 64-bit word. These operations are especially useful for manipulating video data, color pixels, and audio samples, particularly for data compression and decompression.

*Ког. массив 32-х элементов — 32/64
(то только 64-битово, с — не работает)*

Slide: HP-UX 11.0 Architectural Support

HP-UX 11.0 Architectural Support

HP-UX 11.0



Notes:

Указана поддержка 32

Slide: HP-UX 11.0 Architectural Support

HP-UX 11.0 Supports PA-RISC 1.1 and PA-RISC 2.0

HP-UX 11.0 runs on PA-RISC 1.1 and PA-RISC 2.0 systems. Because PA-RISC 2.0 supports two modes, 64-bit mode (called Wide Mode) and 32-bit mode (Narrow Mode), this module covers all three possibilities: PA-RISC 1.1, PA-RISC 2.0 Narrow Mode and PA-RISC 2.0 Wide Mode.

When appropriate the PA-RISC 1.1, 2.0 Narrow Mode, and 2.0 Wide Mode implementations are discussed on sequential slides. Often, because PA-RISC 1.1 and PA-RISC 2.0 Narrow Mode are sometimes very similar, one slide will discuss both.

Note: Two documents, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and *PA-RISC 2.0 Architecture* contain details about their respective versions of PA-RISC architecture. These books are referenced several times throughout this module. **When there are differences between 1.1 and 2.0, both books will be cited.** When the functionality being discussed has not changed from 1.1 to 2.0, only the *PA-RISC 2.0 Architecture* text will be cited.

Slide: HP-UX 11.0 Architectural Support

PA-RISC 2.0 Requirements

PA-RISC 2.0 was designed to meet the following requirements:

Support for Large High-End Applications

One key feature of PA-RISC 2.0 is the extension the PA-RISC architecture to a word size of 64-bits, for integers, physical addresses, and flat virtual addresses. This feature is necessary because 32-bit general registers and addresses with a maximum of 2^{32} byte objects become limiters as physical memories larger than 4 GB become practical. Some high-end applications already exceed the 4 GB working set size.

The table below summarizes some of the PA-RISC 2.0 features that provide 64-bit support.

New PA-RISC 2.0 Feature	Reason for Feature
Processor Status Word W-bit	Provides 32-bit versus 64-bit pointers
Variable sized pages	More flexible intra-space management and fewer TLB entries
Larger protection identifiers	More flexible protection regions
More protection identifier registers	More efficient management of protection identifiers
load/store double (64-bits)	64-Bit memory access

Зачем
но память

Binary Compatibility

Another PA-RISC 2.0 requirement is to maintain complete binary compatibility with PA-RISC 1.1. That is, the binary representation of existing PA-RISC 1.1 software programs must run correctly on PA-RISC 2.0 processors. The transition to 64-bit architectures is unlike the previous 32-bit microprocessor transition which was driven by an application pull. By the time that technology enabled cost-effective 32-bit processors, many applications had already outgrown 16-bit size constraints, and were “coping” with the 16-bit environment by awkward and inefficient means.

With the 64-bit transition, fewer applications need the extra capabilities and many applications will choose to forgo the transition. In many cases, due to cache memory effects, if an application does not need the extra capacities of a 64-bit architecture, it can achieve greater performance by remaining a 32-bit application. Yet 64-bit architectures are a necessity since some crucial applications, databases and large-scale engineering programs, and the operating system itself need this extra capacity.

Therefore, 32-bit applications are very important and must not be penalized when running on the 64-bit architecture. 32-bit applications will remain a significant portion of the execution profile and should also benefit from the increased capabilities of the 64-bit architecture without being ported to a new environment. Of course, it is also a requirement to provide full performance for 64-bit applications and the extended capabilities that are enabled by a wider machine.

Slide: HP-UX 11.0 Architectural Support

Mixed-Mode Execution

Another binary compatibility requirement in PA-RISC 2.0 is mixed-mode execution. This refers to the mixing of 32-bit and 64-bit applications or to the mixing of 32-bit and 64-bit data computations in a single application. In the transition from 32-bits to 64-bits, this ability is a key compatibility requirement, and is fully supported by the new architecture. The W bit in the Processor Status Word is changed from 0 (Narrow Mode) to 1 (Wide Mode) to enable the transition from 32-bit pointers to 64-bit pointers.

Performance Enhancements

Providing significant performance enhancements is another requirement. This is especially true for new computing environments that will become common during the lifetime of PA-RISC 2.0. For example, the shift in the workloads of both technical and business computations to include an increasing amount of multimedia processing led to the Multimedia Acceleration eXtensions (MAX) which are part of the PA-RISC 2.0 architecture. (Previously, a subset of these multimedia instructions were included in an implementation of PA-RISC 1.1 architecture as implementation-specific features.)

The table below summarizes some of the PA-RISC 2.0 performance features.

New PA-RISC 2.0 Feature	Reason for Feature
Weakly ordered memory accesses	Enables higher performance memory systems
Cache hint: Spatial locality	Prevent cache pollution when data has no reuse
Cache line prefetch	Reduce cache miss penalty, and prefetch penalty by disallowing TLB miss

Integrity of Basic Architecture

A final requirement was to add the 64-bit extensions to PA-RISC without disrupting the user community's understanding of the basic architecture. It was very important to build on how mechanisms work in PA-RISC 1.1 and naturally extend that definition.

*сгус. SDK, возборозытн на 32 маунтл
созгаш 64 сур нпурозытн.*

Slide: Register Context

Register Context

32 General Registers

8 Space Registers

32 Control Registers

64 Floating Point Registers — *непомножили* *грустно!*

7 Shadow Registers

2 Instruction Address Queues

1 Processor Status Word

a60624

Notes:

Slide: Register Context

One of the components of the CPU that we mentioned previously is the register set. **Registers** are high speed memory that is defined for specific uses. A PA-RISC system has the following types of registers

- General Registers
- Space Registers
- Control Registers
- Floating Point Registers
- Shadow Registers
- Instruction Address Queues
- Processor Status Word

PA-RISC architecture takes advantage of the speed of these registers by making its operations register intensive. All computations are performed between registers or between a register and a constant. This minimizes the need to access main memory or code.

The **Floating Point Registers** and **Shadow Registers** are discussed below. The remaining registers are discussed in the next several slides. You may find it beneficial to reference the register set diagram on page 2-73.

Floating Point Registers

The Floating Pointer Registers are not actually part of the CPU but are part of the Floating Point Coprocessor. Since this coprocessor is present on most systems, these registers make up a common part of the register context.

There are a total of **32 64-bit** floating point registers. They can be treated as such or as **64 32-bit** registers depending on what types of instructions they are accessed with (double or single-word load/stores). For the PA2.0 processors they can also be treated as **16 128-bit** registers.

Most of the floating point registers are simply data registers used to hold computations. Registers FP-0L through FP-3L are partitioned into 8 32-bit registers. The left word of FP-0L is the status register, and next seven 32-bit registers are exception registers. The **status register** controls arithmetic rounding modes, enables traps, indicates exceptions that have occurred, indicates the results of comparison, and identifies the coprocessor implementation. The seven 32-bit **exception registers** contain information on floating point operations that have completed execution and have caused a delayed trap.

For details of the floating point registers refer to *PA-RISC 2.0 Architecture*.

Shadow Registers — сохранение контекста при прерывании

PA-RISC processors also have seven shadow registers, numbered SHR 0 through SHR 6. The shadow registers are used to store the contents of general registers 1, 8, 9, 16, 17, 24, and 25 on interrupt. The same general registers are restored from these shadow registers on return from interrupt.

Прерывание не может прервать обработку
выполнение, пока не сохранит и не разгрузит

Slide: General Registers

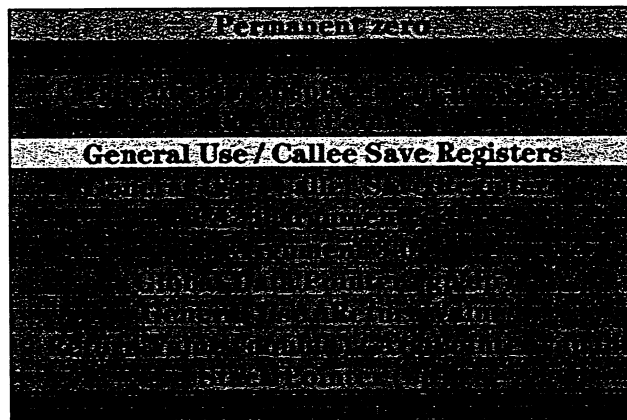
General Registers

GR0
GR1
GR2

GR3-18
GR19-26

GR19-22
GR23-26

GR27
GR28
GR29
GR30
GR31



PA1.1 provides 32-bit registers
PA2.0 provides 64-bit registers

a60625

Notes:

Slide: General Registers

General Registers

There are a total of 32 general registers, each 32-bits in size for PA-RISC 1.1 and 64-bits in size for PA-RISC 2.0. Only 4 of the registers have a special function defined by the PA-RISC architecture. These registers are GR0, GR1, GR2 and GR31. GR0 will always be zero, but GR1, GR2, and GR31 may also be used as general registers. The remaining registers are defined as general registers by the architecture, however HP-UX defines a special purpose for these registers.

These registers are used as working areas to hold immediate results or data that is accessed frequently. All data loaded from memory or stored to memory by the processor is done using a General Register. A common use of general registers is the passing of parameters. HP-UX expects that once parameters are loaded for a procedure call, they will be accessed rather quickly by the called procedure. So it makes sense to put the parameters in registers for fast access rather than storing them solely in the user stack.

Below is a summary of the General Register usage as defined by PA-RISC and HP-UX. Many of the registers have a special meaning ONLY in the context of a procedure call. **Procedure Calling Conventions** are discussed further in Module 3. Shaded rows indicate changes for PA-RISC 2.0.

GR0	Permanent Zero
GR1	ADDIL Target Address An ADDIL instruction will always deposit its result here. This register is also available for general use.
GR2	Target for long displacement of B,L / Return Pointer When a branch is taken, this register contains the instruction offset of the instruction to return to (the instruction following the branch)
GR3-GR18	Callee Save Registers / General Usage Callee Save Registers are saved by a Called Procedure upon entry to the procedure and restored prior to returning to the Calling procedure. <i>These registers are available for general use when not in the context of a procedure call.</i>
GR19-GR26	Caller Save Registers Procedure Arguments Caller Save Registers are saved by a Calling Procedure before branching to the Called Procedure and restored after returning from the Called Procedure. These registers are also used for passing procedure arguments to the Called Procedure. <i>These registers are available for general use when not in the context of a procedure call.</i>
GR19	PA 1.1: General Usage PA 2.0: Argument 1
GR20	PA 1.1: General Usage PA 2.0: Argument 2

←
 COPIED FROM
 KSD
 COPY PASTED
 PERMITS

Slide: General Registers

GR21	[PA 1.1: General Usage] [PA 2.0: Argument 5]
GR22	[PA 1.1: General Usage] [PA 2.0: Argument 4]
GR23	Argument 3
GR24	Argument 2
GR25	Argument 1
GR26	Argument 0
GR27	Global Data Pointer (gp/dp) For HP-UX the kernel's <i>dp</i> value (also know as <i>gp</i>) is stored in a kernel variable named <i>\$global\$</i> .
GR28	Return Value
GR29	[PA 1.1: Return Value (double)] If the return value is a double word, this will be word 0 and GR28 will contain word 1. [PA 2.0: Argument pointer] The Argument Pointer (AP) is used for referencing parameters stored in process's stack. <i>This register is available for general use when not in the work- ing set of the microarchitecture.</i>
GR30	Stack Pointer (sp) This register is the address of the current "top" of stack.
GR31	Link Register for BLE or General Use Instruction address offset link register for the base relative interspace procedure call instruction.

Left blank intentionally

Slide: Space Registers

Space Registers

	Link code space ID	32-bit apps	64-bit apps
SR0	General Use		
SR1	General Use		
SR2	General Use		
SR3	General Use		
SR4			
SR5			
SR6			
SR7			

PA1.1 defines 32-bit registers, but only provides 16 bits
 PA2.0 defines 64-bit registers, but only provides 32 bits

*pacu...
 VM na 4 casu.*

a60626

Notes:

Slide: Space Registers

Space Registers

The space registers are used to hold the Space Identifiers. Space IDs are used in conjunction with an offset in a general register to form a Virtual Address. Virtual addresses are discussed further beginning on page 2-39.

PA-RISC 1.1 defines 32-bit space registers. Combined with a 32-bit offset stored in a General Register, this forms a 64-bit virtual address. However, the actual implementation uses only 16 bits for the Space ID. Thus for PA-RISC 1.1 systems, a virtual address is actually 48 bits.

PA-RISC 2.0 defines 64-bit space registers. With PA2.0 the space registers and general registers are overlapped by 32 bits when combining to form global virtual addresses. Thus combined with a 64-bit offset stored in a General Register, this forms a 96-bit virtual address. However, the actual implementation uses only 32 bits for the Space ID. Thus for PA-RISC 2.0 systems, a virtual address is actually 64 bits.

It is very confusing to discuss what is architected by PA-RISC and what is actually implemented. We will commonly refer to Space Registers being 32 bits on PA-RISC 1.1 and 64 bits on PA-RISC 2.0. However, remember that our implementation only uses half of these bits.

There are a total of eight space registers named SR0 through SR7. SR0 is the instruction address space link register used for BRANCH AND LINK EXTERNAL instructions. Because processes commonly access certain data frequently (such as process code or text, process stack, global data, shared objects), the Space IDs for these areas are kept in SR4 through SR7. By using a concept of **short pointer addressing** or **implicit pointers**, a process can access a 64-bit virtual address (on PA Risc 1.1) using a 32-bit offset. The Space ID is located in SR4-SR7 depending on 2 Space Register Selection bits found in the offset. SR1-SR3 are used to construct **Long (explicit)** pointers. Short (implicit) and Long (explicit) pointers are discussed further beginning on page 2-53.

Slide: Control Registers

Control Registers

CR0	Recovery Counter
CR1-7	Reserved
CR8,9,12,13	Protection IDs
CR10	Coprocessor Configuration Register
CR11	Shift amount register
CR14	Interrupt vector address
CR15	External Interrupt Enable Mask
CR16	Interval timer
CR17,18	Interrupted Instruction Address Space and Offset Queue
CR19	Interrupted Instruction Register
CR20,21	Interrupted Space and Offset Registers
CR22	Interrupted Processor Status Word
CR23	External Interrupt Request Register
CR24-32	Temporary registers

PA1.1 uses mostly 32-bit registers
 PA2.0 uses mostly 64-bit registers

a69627

Notes:

Slide: Control Registers

Control Registers

The CPU also defines 32 Control Registers. PA-RISC 1.1 has 32-bit registers; PA-RISC 2.0 has 64-bit registers. Control Registers are used to reflect different states of the system. Most of the registers have very specific purposes related primarily to interrupt handling.

CR0	Recovery Counter This is a 32-bit register (even on PA-RISC 2.0). This register is decremented during the execution of each non-nullified instruction for which the PSW R-bit is 1. When the left-most bit of the Recovery Counter is 1, a recovery count trap occurs.
CR1-CR7	Reserved
CR8	[PA 1.1] Protection ID 1 [PA 2.0] Protection ID 1 and 2 When translation is enabled, the protection identifiers are compared with a page access identifier in the TLB entry to validate access.
CR9	[PA 1.1] Protection ID 2 [PA 2.0] Protection ID 3 and 4
CR10	Coprocessor Configuration (CCR) and SFU Configuration Register. Right-most 8 bits (CCR) indicate presence and usability of coprocessors. The preceding 8 bits (SCR) indicates which Special Function Units (SFUs) are enabled.
CR11	Shift Amount Register (SAR) 6-bit register used by variable shift, extrace, deposit and branch on bit instructions. It specifies the number of bits a quantity is to be shifted.
CR12	[PA 1.1] Protection ID 3 [PA 2.0] Protection ID 5 and 6
CR13	[PA 1.1] Protection ID 4 [PA 2.0] Protection ID 7 and 8
CR14	Interrupt Vector Address Contains the absolute address of the base of an array of interrupt service procedures.
CR15	External Interrupt Enable Mask (EIEM) One bit per each type of external interrupt. An external interrupt whose corresponding bit is set to 0 in the EIEM will be held off.

Slide: Control Registers

CR16	<p>Interval Timer Actually consists of 2 internal registers. One continually counts up by 1. The other register is set by writing to the register. When the low-order 32-bits are the same, an External Interrupt is generated.</p>
CR17	<p>Interrupt Instruction Address Space Queue Stores the contents of the Instruction Address Space Queue at the time of an interruption</p>
CR18	<p>Interrupt Instruction Address Offset Queue Stores the contents of the Instruction Address Offset Queue at the time of an interruption</p>
CR19	<p>Interrupt Instruction Register Used to pass executing instruction at the time of an interrupt to an interrupt handler. (PA-RISC 2.0: upper 32 bits reserved; lower 32 bits of register used for IIR)</p>
CR20	<p>Interruption Space Register Used with CR21 to pass a virtual address to an interruption handler.</p>
CR21	<p>Interruption Offset Register Used with CR20 to pass a virtual address to an interruption handler.</p>
CR22	<p>Interruption Processor Status Word Holds the value of the Processor Status Word when an interruption occurs.</p>
CR23	<p>External Interrupt Request Register (EIRR) Contains a bit for each type of external interrupt. When 1, a bit designates that an interruption is pending for the corresponding external interrupt. Both the PSW I-bit and the corresponding bit in the EIEM must be 1 for an interruption to occur.</p>
CR24-CR31	<p>Temporary Registers These registers provide space to save contents of general registers for interrupt handlers. CR24 is also known as the Per-Processor Data Pointer (PPDP), which is a pointer to the processor's <i>mpinfo</i> structure (discussed further in Module 6)</p>

Left blank intentionally

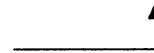
Slide: Instruction Address Queues

Instruction Address Queues

Instruction Address Offset Queue (IAOQ)

IAOQ_front	00
IAOQ_back	00

2-bit Privilege Level



Instruction Address Space Queue (IASQ)

IASQ_front
IASQ_back

RWX, Level
Bezo 4 ypybne -
- cpa6peue jonyua.
UcnoMuuuu
Honye coRITe

a66628

Notes:

Slide: Instruction Address Queues

The Instruction Address Queues (IAQ) hold the instruction address for the currently executing instruction. There are two instruction address queues:

- Instruction Address Space Queue (IASQ)
- Instruction Address Offset Queue (IAOQ)

Each of these queues are actually composed of a pair of registers (two 32-bit registers in PA-RISC 1.1; two 64-bit registers in PA-RISC 2.0) creating what we call a front and a back element. Using the front values from each queue we construct the virtual address for the currently executing instruction. Using the back values we get the following or next instruction.

From this, the virtual address for the current instruction is:

IASQ_front.IAOQ_front

and the virtual address for the next instruction is:

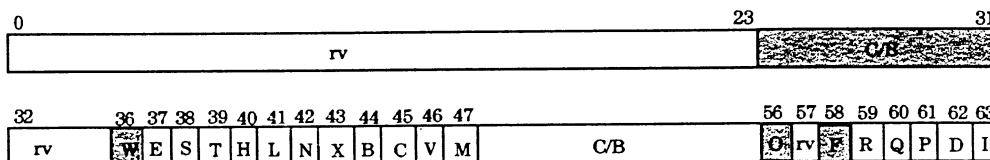
IASQ_back.IAOQ_back

Maintaining a front and back queue aids in the pipelining of instructions.

The instruction offset address is a word offset extracted from all but the lower 2 bits of the IAOQ. Thus, for PA-RISC 1.1 this word offset is 30 bits; for PA-RISC 2.0 it is 62 bits. Since all instructions are aligned on a 4-byte (word) boundary, the last 2 bits will always be zero. To avoid wasting bits, the low order 2 bits specify the privilege level of the instruction. Thus, the byte offset is formed by masking off the 2 privilege level bits. There are four possible privilege levels (0 - 3) but HP-UX recognizes only 0 for kernel and 3 for user.

Slide: PA-RISC 2.0 Processor Status Word (PSW)

PA-RISC 2.0 Processor Status Word (PSW)



- | | | | |
|-----|---------------------------------------|---|------------------------------------|
| rv | Reserved Bits | C | Code Address Translation |
| C/B | Carry/Borrow Bits | V | Divide Step Correction |
| W | Wide 64-bit address formation enable | M | High Priority Machine Check Mask |
| E | Little Endian Memory Access Enable | O | Ordered References |
| S | Secure Interval Timer | F | Performance Monitor Interrupt Mask |
| T | Taken Branch Trap Enable | R | Recovery Counter Enable |
| H | Higher-Privilege Transfer Trap Enable | Q | Interruption State Collection Mask |
| L | Lower-Privilege Transfer Trap Enable | P | Protection ID Validation Enable |
| N | Nullify Instruction | D | Data Address Translation Enable |
| X | Data Memory Break Disable | I | External Interrupt, Power Failure |
| B | Taken Branch | | Interrupt and LPMC Unmask |

Code VM



PA RISC 1.0 - always big endian
 ≥ 1.1 - can be big or little (с точки зрения го справуєт)

a69629

Notes:

Slide: PA-RISC 2.0 Processor Status Word (PSW)

The CPU has a register called the **Processor Status Word (PSW)** that contains the current processor state. This register is 32 bits in PA-RISC 1.1 and 64 bits in PA-RISC 2.0. The slide shows the 64-bit PSW. Whenever an interrupt occurs, the current PSW is saved into the **Interrupt Processor Status Word (IPSW)** to maintain the state and then is restored when returning from the interrupt.

The low order five bits of the PSW are known as the **system mask** because they mask/unmask specific types of system conditions. Bits in the PSW are defined as **mask/unmask bits** or **enable/disable bits**. Interrupts that are disabled by the corresponding bit in PSW are ignored by the processor. Interrupts that are masked remain pending until unmasked.

Bits 0-23, 32-35, and 57 are reserved bits (**rv**) in the PSW. The remaining bits are defined in the table below. Shaded rows indicate bit is new for PA-RISC 2.0:

W	Wide 64-bit address formation enable When 1, full 64-bit offset addressing is enabled. When 0, addresses are truncated to 32-bit offsets, for compatibility with existing PA-RISC 1.0 and 1.1 applications.
E	Little Endian Memory Access Enable When 0, all memory references are big endian. When 1, all memory references are little endian.
S	Secure Interval Timer When 1, the Interval Timer is readable only by code at the highest privilege level. When 0, it is readable by anyone.
T	Taken Branch Trap Enable When 1, any taken branch is terminated with a taken branch trap.
H	Higher-Privilege Transfer Trap Enable When 1, a higher privilege transfer trap occurs whenever the instruction following is of higher privilege.
L	Lower-Privilege Transfer Trap Enable When 1, a lower privilege transfer trap occurs whenever the instruction following is of lower privilege.
N	Nullify Instruction The current instruction is nullified (ignored) when this bit is 1. The bit is set by an instruction that nullifies the following instruction, such as a branch.

Module 2 — System Architecture

Slide: PA-RISC 2.0 Processor Status Word (PSW)

X	Data Memory Break Disable When set to 1, data memory break traps are disabled. This bit provides a way to control trapping on individual data store instructions.
B	Taken Branch Set to 1 on any taken branch instruction and set to 0 otherwise.
C	Code Address Translation Enable When set to 1, instruction addresses are translated and access rights are checked. A value of 0 is indication that we are operating in real addressing mode.
V	Divide Step Correction Set for DIVIDE STEP instruction to specify the set of conditions to use, thus providing an integer division primitive.
M	High Priority Machine Check (HPMC) Enable When 1, HPMCs are masked. Only set to 1 after an HPMC and set to 0 after all other interruptions.
C/B	Carry/Borrow Bits Carry Borrow bits for arithmetic instructions. See individual instructions for details on how bits are set. In general a 1 in a given bit indicates a carry/borrow for that particular digit.
O	Ordered References When 1, virtual memory references in pages with the corresponding TLB valid, and all absolute memory references, are ordered. When 0, memory references (even those explicitly marked as ordered or strongly ordered) may be weakly ordered. Note that references in I/O address space, references to pages with no TLB valid, single-precision instructions, and MMX/PAVE instructions are always strongly ordered.
I	Performance Monitor Interrupt Unmask When 1, the performance monitor interrupt is unmasked and can cause an interrupt.
R	Recovery Counter Enable <i>debugging mode</i> When 1 recovery counter traps are enabled. The bit also enables decrementing of the recovery counter.
Q	Interruption State Collection Mask When 1 IIAQ, IIR, ISR, and IOR are saved on interruption.

Slide: PA-RISC 2.0 Processor Status Word (PSW)

P	Protection ID Validation Enable When this bit and the C bit are both 1, instructions are checked for valid protection identifier(PID). When this bit and the D bit are both 1, data references are checked for valid PID.
D	Data Address Translation Enable When 1, data addresses are translated and access rights are checked.
I	External Interrupt, Power Failure Interrupt, and Low Priority Machine Check (LPMC) Unmask When set to 1, these interruptions are unmasked and can cause an interruption.

With the discussion of the PSW, we now have discussed the entire register context of the PA-RISC 2.0 processor. The entire register set together for use as a reference can be found on page 2-73 through page 2-75.

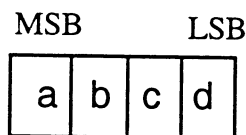
Some of the conventions listed on these pages are software conventions rather than architecture but the two are listed together to make a complete reference.

Byte Ordering

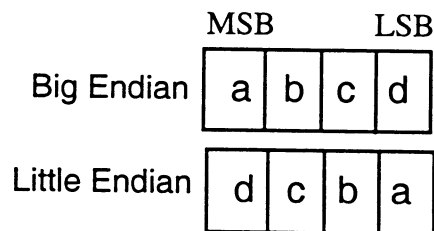
The optional E-bit in the PSW controls whether the ordering of bytes on loads and stores are **big endian** or **little endian**. When the E-bit is 0, all ordering is big endian, meaning that lower addressed bytes in memory correspond to high order bytes in the register. When the E-bit is 1, ordering is little endian in which the lower addressed bytes in memory correspond to lower order bytes in the register.

The diagram below shows the difference between the two types of byte ordering where MSB = Most Significant Byte; LSB = Least Significant Byte. HP-UX does all of its byte ordering as big endian, however it is possible to have non-HP I/O cards (such as on the workstation EISA interface) that use little endian ordering. In this case the driver for the card would need to handle the byte translation for HP-UX .

Memory Contents



Register Contents



Slide: PA-RISC 2.0 Processor Status Word (PSW)

Ordered References

This PA-RISC 2.0 option allows implementation of strongly or weakly ordered memory references. PA-RISC 1.1 specified that all loads and stores be performed “in order.” This is known as “strong ordering.”

PA-RISC 2.0 supports “weak ordering.” This is because modern processors can reduce latencies from cache misses by simultaneously performing loads and stores to lines already in the cache. This effective reordering of loads and stores causes no inconsistency in most cases and permits faster execution. Of course, strongly ordered loads and stores must be allowed to handle contexts in which ordering must be preserved. This need for strong ordering is mainly related to synchronization among processors or with I/O activity. For more information about the memory ordering model, see Appendix G in *PA-RISC 2.0 Architecture*.

For Your Reference

The upcoming lab exercise has complete summaries of PA-RISC 1.1 and 2.0 Register Contexts and Processor Status Word contents.

Slide: PA-RISC 2.0 Processor Status Word (PSW)

PA-RISC 1.1 Register Context

(32-bit Registers)

32 General Registers

GR-0	Permanent Zero
GR-1	ADDIL Targe Address
GR-2	Return Pointer (rp)
GR-3	
GR-4	General Usage
GR-5	...
GR-6	...
GR-7	...
GR-8	...
GR-9	...
GR-10	...
GR-11	...
GR-12	...
GR-13	...
GR-14	...
GR-15	...
GR-16	...
GR-17	...
GR-18	General Usage
GR-19	General Usage
GR-20	...
GR-21	...
GR-22	General Usage
GR-23	Argument 3 (arg3)
GR-24	Argument 2 (arg2)
GR-25	Argument 1 (arg1)
GR-26	Argument 0 (arg0)
GR-27	Data Pointer (dp)
GR-28	Return Value / Frame Ptr
GR-29	Return Value (double) Static Link
GR-30	Stack Pointer (sp)
GR-31	Link for BLE or General Use

Callee Save Registers

Caller Save Registers

2 Instruction Queues

IASQ_Front	Current Instr Space
IASQ_Back	Next Instr Space

IAOQ_Front	Current Instr Offset*
IAOQ_Back	Next Instr Offset*

* Last two bits of IAOQ are Privilege Level (0,1,2, or 3)

8 Space Registers

SR-0	Space Link Register for BLE
SR-1	
SR-2	
SR-3	
SR-4	User Text/Kernel Text & Data
SR-5	User Data and Stack
SR-6	Shared Text/ Data
SR-7	Shared Text/Data

32 Floating Point Regs (64-bits each)

FP-0	Status Register/Exptn Reg 1
FP-1	Exception Register 2/3
FP-2	Exception Register 4/5
FP-3	Exception Register 6/7

64-bit Floating Point
Data Registers

32 Control Registers

CR-0	Recovery Counter
CR-1	Reserved
CR-2	...
CR-3	...
CR-4	...
CR-5	...
CR-6	...
CR-7	Reserved
CR-8	Protection ID 1
CR-9	Protection ID 2
CR-10	Coprocessor Configuration
CR-11	Shift Amount Register
CR-12	Protection ID 3
CR-13	Protection ID 4
CR-14	Interrupt Vector Address
CR-15	EIEM
CR-16	Interval Timer
CR-17	PC Space Queue
CR-18	PC Offset Queue
CR-19	Interrupt Instruction Register
CR-20	Interrupt Space Register
CR-21	Interrupt Offset Register
CR-22	Interrupt PSW
CR-23	EIRR
CR-24	Per Processor Data Ptr PDIR Address Temporary Register 0
CR-25	Hash Table Address Temporary Register 1
CR-26	Temporary Register 2
CR-27	Temporary Register 3
CR-28	Temporary Register 4
CR-29	Temporary Register 5
CR-30	Temporary Register 6
CR-31	Temporary Register 7

Slide: PA-RISC 2.0 Processor Status Word (PSW)

PA-RISC 2.0 Register Context

(64-bit Registers)

32 General Registers

GR-0	Permanent Zero
GR-1	ADDIL Target Address
GR-2	Return Pointer (rp)
GR-3	
GR-4	General Usage
GR-5	...
GR-6	...
GR-7	...
GR-8	...
GR-9	...
GR-10	...
GR-11	...
GR-12	...
GR-13	...
GR-14	...
GR-15	...
GR-16	...
GR-17	...
GR-18	General Usage
GR-19	Argument 7 (arg7)
GR-20	Argument 6 (arg6)
GR-21	Argument 5 (arg5)
GR-22	Argument 4 (arg4)
GR-23	Argument 3 (arg3)
GR-24	Argument 2 (arg2)
GR-25	Argument 1 (arg1)
GR-26	Argument 0 (arg0)
GR-27	Data Pointer (gp/dp)
GR-28	Return Value / Frame Ptr
GR-29	Arg Pointer (ap)
GR-30	Stack Pointer (sp)
GR-31	Link for BLE or General Use

Callee Save Registers

Caller Save Registers

Differences from PA 1.1

2 Instruction Queues

IASQ_Front	Current Instr Space
IASQ_Back	Next Instr Space

IAOQ_Front	Current Instr Offset*
IAOQ_Back	Next Instr Offset*

* Last two bits of IAOQ are Privilege Level (0,1,2, or 3)

8 Space Registers

SR-0	Kernel Text and Data
SR-1	Not used in wide
SR-2	Not used in wide
SR-3	Not used in wide
SR-4	32/64-bit globals/Kernel data
SR-5	User Data and Stack
SR-6	Process Data
SR-7	64-bit Globals

32 Floating Point Regs (64-bits each)

FP-0	Status Register/Exptn Reg 1
FP-1	Exception Register 2/3
FP-2	Exception Register 4/5
FP-3	Exception Register 6/7

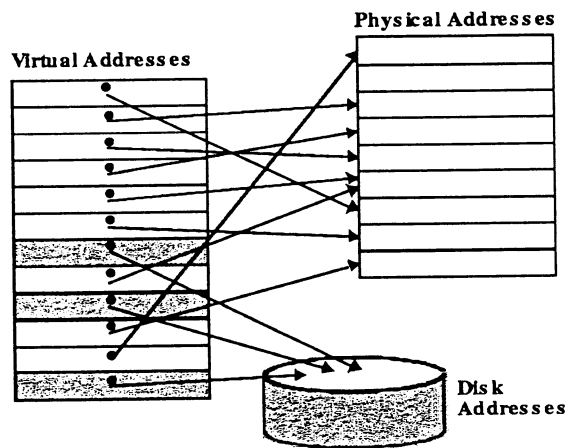
64-bit Floating Point Data Registers

32 Control Registers

CR-0	Recovery Counter
CR-1	Reserved
CR-2	...
CR-3	...
CR-4	...
CR-5	...
CR-6	...
CR-7	Reserved
CR-8	Protection ID 1/2
CR-9	Protection ID 3/4
CR-10	Coprocessor Configuration
CR-11	Shift Amount Register
CR-12	Protection ID 5/6
CR-13	Protection ID 7/8
CR-14	Interrupt Vector Address
CR-15	EIEM
CR-16	Interval Timer
CR-17	PC Space Queue
CR-18	PC Offset Queue
CR-19	Interrupt Instruction Register
CR-20	Interrupt Space Register
CR-21	Interrupt Offset Register
CR-22	Interrupt PSW
CR-23	EIRR
CR-24	Per Processor Data Ptr PDIR Address Temporary Register 0
CR-25	Hash Table Address Temporary Register 1
CR-26	Temporary Register 2
CR-27	Thread data (tp)
CR-28	Temporary Register 4
CR-29	Temporary Register 5
CR-30	Temporary Register 6
CR-31	Temporary Register 7

Slide: Virtual Memory Concepts

Virtual Memory Concepts



a68630

Notes:

Slide: Virtual Memory Concepts

Virtual Memory

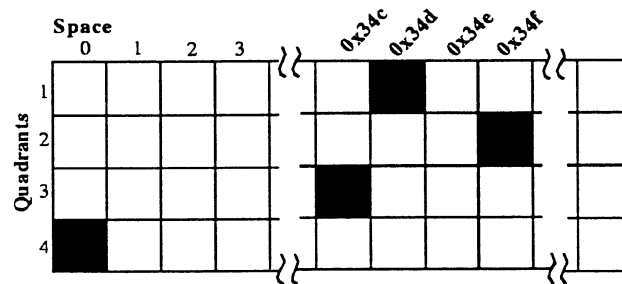
In virtual memory, pages are mapped from one set of addresses, virtual addresses, to another set, physical addresses. The processor generates virtual addresses while the memory is accessed using physical addresses.

Both the virtual memory and the physical memory are broken into pages, so that a virtual page is really mapped to a physical page. Of course, it is also possible for a virtual page to be absent from physical memory and not be mapped to a physical address, residing instead on disk. Program addresses are virtual and are divided by the hardware into a virtual page number and an offset into the page.

Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

Slide: Virtual Memory Layout: PA-RISC 1.1 and 2.0 Narrow Mode

Virtual Memory Layout: PA-RISC 1.1 and 2.0 Narrow Mode



rebaggart

Process VA Space

Text	0x34d.xxxxxxxx	} 4 Gbyte Total
Data	0x34f.xxxxxxxx	
Shared 2	0x34c.xxxxxxxx	
Shared 1	0x000.xxxxxxxx	

Space d *radde*

a6631

Notes:

Slide: Virtual Memory Layout: PA-RISC 1.1 and 2.0 Narrow Mode

This slide discusses the PA-RISC 1.1 and the 2.0 Narrow Mode virtual memory layout (the W bit is 0).

Process Address Space

In general, the address space of a process comprises all virtual memory locations that the program may reference. At any instant, the address space, along with the process's register context, reflects the current state of the program.

On a PA-RISC system, every page of physical memory has a physical address which is a **physical page number**. All access to these pages is done through virtual addresses. These virtual addresses represent an address into a large collection of imaginary memory regions. These virtual addresses are mapped to physical addresses that represent physical pages in memory.

For the purpose of addressing, virtual memory is partitioned into **quadrants** and **spaces**. A virtual memory address is comprised of a Space ID and offset within the space. In PA-RISC 1.1 and 2.0 Narrow Mode, each space represents a 4 Gbyte unit (2^{32}) of virtual memory. The offset portion of a virtual address is the offset into this space.

The format of an HP-UX PA-RISC 1.1 or 2.0 Narrow Mode virtual address is:

Space ID (32 bits)	Offset (32 bits)
-----------------------	---------------------

Current PA-Risc 1.1 implementations only use 16 bits for the space IDs. In PA-Risc 2.0 Narrow Mode, the space ID can be 32 bits. We will see later, however, that only the high-order 22-bits are actually used for the Space ID. The low-order 10-bits will always be zero.

Each 4Gbyte space is divided equally into four 1 Gbyte units called **quadrants**. The offset portion of the virtual address determines what quadrant of the given space we are in.

	Virtual Address Range
Quadrant 1	0x00000000 to 0x3FFFFFFF
Quadrant 2	0x40000000 to 0x7FFFFFFF
Quadrant 3	0x80000000 to 0xBFFFFFFF
Quadrant 4	0xC0000000 to 0xFFFFFFFF

Each process running on a PA-RISC processor has its own unique virtual address range. When created the process has allotted to it a 4 Gbytes virtual address range. This is not memory that is physically allocated but rather just address ranges that are available. Each quadrant will likely belong to a different space.

In the example shown in the slide, a particular process has been assigned Space ID 0x34d for its text, 0x34f for its data, 0x34c and 0x34e for shared text and data. So, to access the base of the process's user stack in quadrant two, a virtual address of 0x34f.0x7b03a000 would be used (offset 0x7b03a000 is a

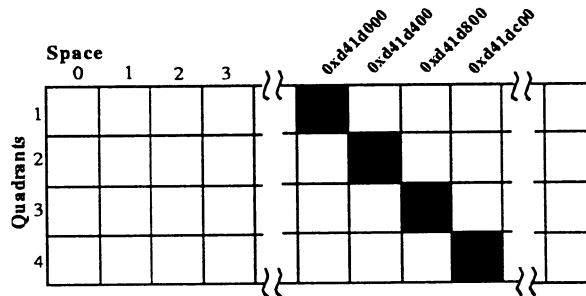
Slide: Virtual Memory Layout: PA-RISC 1.1 and 2.0 Narrow Mode

typically start of a process's user stack). These numbers are used only for the purposes of showing an example. Normally, you would not expect a process to be assigned sequential space ids such as this.

Left blank intentionally

Slide: Virtual Memory Layout: PA-RISC 2.0 Wide Mode

Virtual Memory Layout: PA-RISC 2.0 Wide Mode



Process VA Space

Shared Objects	0xd41d000.0xxxxxxxx xxxxxxxx	} 16 TB Total
Text	0xd41d400.4xxxxxxxx xxxxxxxx	
Data	0xd41d800.8xxxxxxxx xxxxxxxx	
Shared Objects	0xd41dc00.cxxxxxxxx xxxxxxxx	

a69632

Notes:

Slide: Virtual Memory Layout: PA-RISC 2.0 Wide Mode

In Wide Mode concepts of physical pages and virtual addresses are the same as for Narrow Mode. Differences exist in the formation and format of the virtual address and in the size range and layout of the address space.

The format of a PA-RISC 2.0 Wide Mode virtual Address is:

Space ID (64 bits)	Offset (64 bits)
-----------------------	---------------------

Current PA-Risc 2.0 implementations only use 32 bits for the space IDs. We will see later, however, that only the high-order 22-bits are actually used for the Space ID. The low-order 10-bits will always be zero. Also, the 64-bit offset is comprised of 2 Space register selection bits, followed by 20-bits set to zero, and a 42-bit offset within the space.

In Wide Mode each space is divided as follows into four 4 TB quadrants. In the Virtual Address Ranges below, note the high order 2 bits are the Space Register Selection bits and the next 20 bits are set to zero.

	Virtual Address Range
Quadrant 1	0x00000000 00000000 to 0x000003FF FFFFFFFF
Quadrant 2	0x40000000 00000000 to 0x400003FF FFFFFFFF
Quadrant 3	0x80000000 00000000 to 0x800003FF FFFFFFFF
Quadrant 4	0xC0000000 00000000 to 0xC00003FF FFFFFFFF

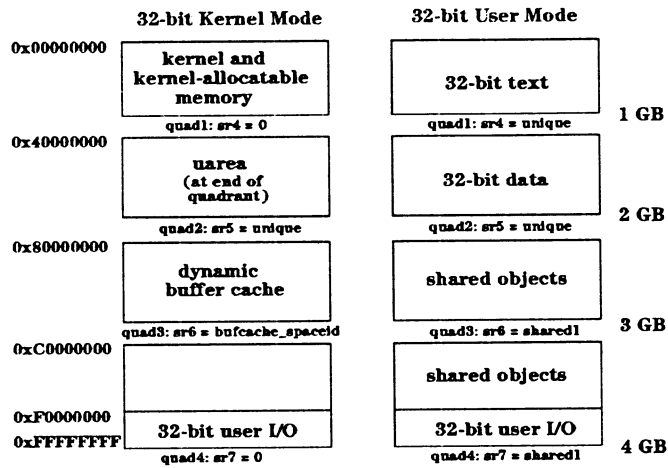
In Wide Mode, when created the process has allotted to it a 16 TB virtual address range. This is not memory that is physically allocated but rather just address ranges that are available. The 16 TB of virtual address space will not be all within one space.

In the example shown in the slide, a particular process has been assigned SID 0x41d400 for its text, 0x41d800 for its data, and 0x41d000 and 0x41dc00 for shared objects. Note the low-order 10 bits in the space IDs are set to zero. These numbers are used only for the purposes of showing an example. Normally you would not expect a process to be assigned sequential space ids such as this.

Note that the process virtual address space layout in Wide Mode is different than in Narrow Mode. More details about these differences are covered in the next two slides.

Slide: 32-Bit Address Space Layout

32-Bit Address Space Layout



supra-allocated
address
св. пространство

разомно 425та в убагране
4TB x 4 = 16 TB

a69633

Notes:

разомно space RG - 32bit

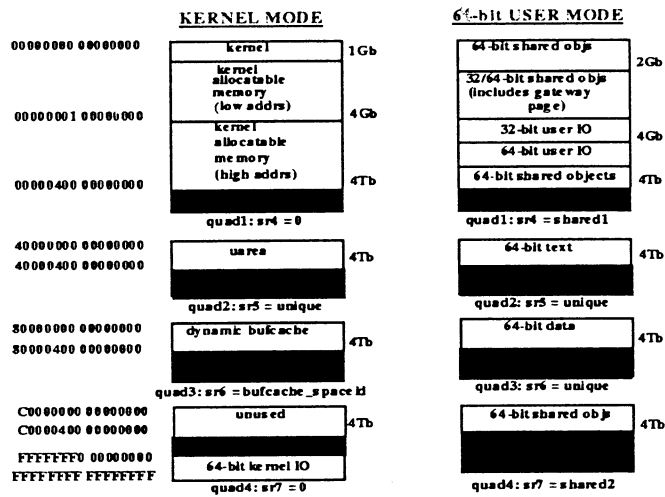
Slide: 32-Bit Address Space Layout

When operating in Narrow Mode or PA-RISC 1.1, processes are limited to a 32-bit address space. The address space layout for a Narrow Mode process is shown on the slide. The layout is the same as it is on HP-UX 10.x.

Note that the 32-bit Kernel Address Space Layout is only for PA-RISC 1.1.

Slide: 64-Bit Address Space Layout

64-Bit Address Space Layout



1-4 kb. shared. (32/64 - unique memory base)

a60634

Notes:

Slide: 64-Bit Address Space Layout

This slide illustrates the address space layout for a 64-bit kernel and for 64-bit users.

Note that one of the requirements for the 64-bit address space layout is to allow 32-bit and 64-bit applications to access the same shared object. This type of sharing is referred to as mixed-mode access.

For a 32-bit process, shared objects reside in Quadrant 3 and Quadrant 4. For a 64-bit process, mixed mode shared objects (i.e. those objects which can be accessed by 64-bit and 32-bit applications concurrently) reside in the third and fourth Gigabyte of Quadrant 1.

All other 64-bit shared objects reside in the remaining address space of Quadrants 1 and 4.

Slide: 32- vs 64-Bit HP-UX Address Layouts

32- vs 64-Bit HP-UX Address Layouts

Narrow Address Space Layouts

32-bit narrow applications (or PA-RISC 1.x)

32-bit kernel

- Might be installed on "64-bit capable" system because 64-bit features are not needed

Wide Address Space Layouts

64-bit wide applications

- Only when running 64-bit kernel

64-bit kernel

- Only on supported (PA8x00) systems with 64-bit kernel (11.0+) installed

a69635

Notes:

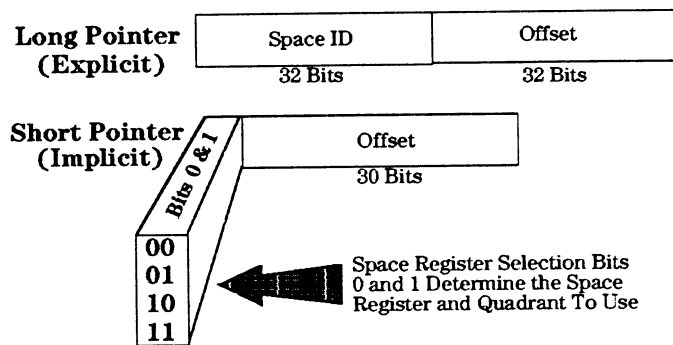
Slide: 32- vs 64-Bit HP-UX Address Layouts

Narrow Mode (32-bit) applications will run on any PA-RISC architecture version with either a 32-bit or a 64-bit HP-UX kernel.

Wide Mode (64-bit) applications require a 64-bit HP-UX kernel, which is only supported on certain PA-8x00-based PA-RISC 2.0 systems. The table at the end of this module contains information about which systems provide this capability.

Long and Short Pointers: PA-RISC 1.1 and 2.0 Narrow Mode in “32bit” node

Virtual Address Pointers: Narrow Mode



a69636

Notes:

Slide: Long and Short Pointers: PA-RISC 1.1 and 2.0 Narrow Mode

PA-RISC 1.1 or Narrow Mode 2.0 systems form virtual addresses using either **long (explicit) pointers** or **short (implicit) pointers**. For long pointers we have a 32-bit space id (again, only 16-bits are used in current PA 1.1 implementations) and a 32-bit offset.

Short pointers are 32-bits long and use the high order two bits to determine the space identifier. Short pointers are only used in the context of a process and access space identifiers stored in space registers 4 through 7.

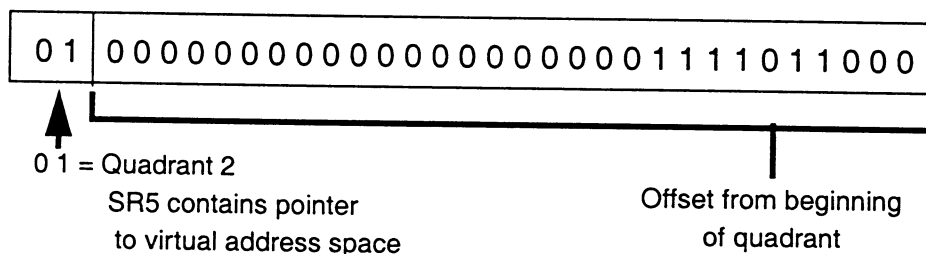
The space identifier for a short pointer is determined by taking the high order 2-bit value and adding 4 to it to determine the space register to use. The SID in that space register is combined with the offset to form the full 64-bit virtual address.

The advantage of using short pointers is by keeping the space ID of commonly accessed data in designated space registers, we can perform loads and stores to memory by simply accessing the 32-bit offset without constantly loading the space ID into a space register. A long pointer reference would first have to load the proper space ID in a space register, then load the offset in to a general register, then access the memory through the virtual address.

This table shows the relationships between Space Register Selection bits, space registers, and quadrants.

Bits 0-1	Space Register	Quadrant
00	SR4	Quad 1
01	SR5	Quad 2
10	SR6	Quad 3
11	SR7	Quad 4

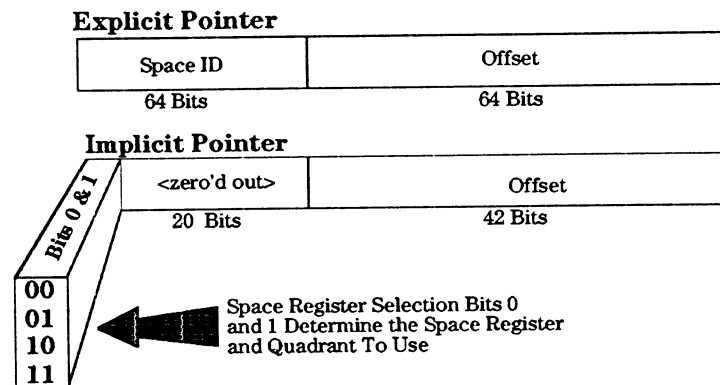
As an example, the diagram below shows the breakdown of a short pointer with the value of 0x400003D8.



Note: In Narrow mode, when dealing with the context of the current process, short pointers are useful. Since we know that the space registers SR4-SR7 contain the context of that process, short pointers will pull the appropriate SID from these registers. When the processor does not hold the context of the process we are interested in, long pointers must be constructed.

Explicit and Implicit Pointers: PA-RISC 2.0 Wide Mode

Virtual Address Pointers: Wide Mode



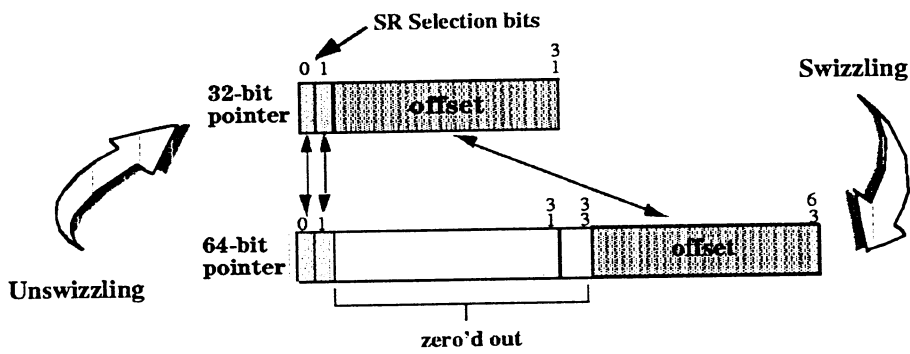
a69637

Notes:

Slide: Address Swizzling

Address Swizzling

Approximate 32 to 64 mapping



a69638

Notes:

Slide: Address Swizzling

Address “swizzling” refers to the conversion of a 32-bit address by copying the top two bits of a 32-bit address to the top two bits of a 64-bit address and the offset to the bottom 30 bits with zero-filled bits in between. The reverse process is referred to as “unswizzling.”

Address swizzling is used by the HP-UX 64-bit operating system when going between kernel and user modes for a Narrow Mode process.

Slide: PA-RISC 2.0 Global Virtual Addresses (GVA)

PA-RISC 2.0 Global Virtual Addresses



Global Virtual Addresses

a69639

Notes:

Slide: PA-RISC 2.0 Global Virtual Addresses (GVA)

The HP-UX 11.0 Global Virtual Address

The term Global Virtual Address (or GVA) is new to PA Risc 2.0. It combines the space ID and offset into a single component to describe the complete virtual address. Note that a 128-bit or even a 96-bit virtual address can not be easily managed on a system with 64-bit registers. We also need a unique method of describing a virtual address regardless of whether we are operating in wide or narrow mode.

For Narrow Mode processes, the 2.0 GVA is the same as the virtual address. The 32-bit space ID is concatenated with the 32-bit offset to form a 64-bit GVA. For Wide Mode processes however, the GVA is not the same as the virtual address. The Wide Mode virtual address is conceptually 128 bits, a 64-bit Space ID and a 64-bit offset. The PA-RISC 2.0 Architecture book documents a 96-bit GVA and implements a 64-bit space ID. However, HP-UX only implements a 32-bit space ID, thus allowing the GVA to fit in 64-bits. Selective bits from the space ID and offset are extracted and ORed together to form a unique 64-bit GVA.

The fact that HP-UX 11.0 uses only a 64-bit GVA offers a distinct advantage: the kernel can greatly simplify its management of the GVA pool by keeping all this data in an easily-managed 64-bit object.

To accomplish this Wide Mode 64-bit GVA, the application's address space is divided into quadrants as in earlier HP-UX releases (quadrants are discussed in detail later in this module) but only the lower 42-bits of each quadrant are allocated as usable addresses. For Wide Mode, this gives each of the four quadrants a 4TB pool for an overall process address space of 16TB. While not the broadest possible use of all the bits available, each of the four quadrants is 1000 times larger than was possible in the entire 32-bit virtual address model.

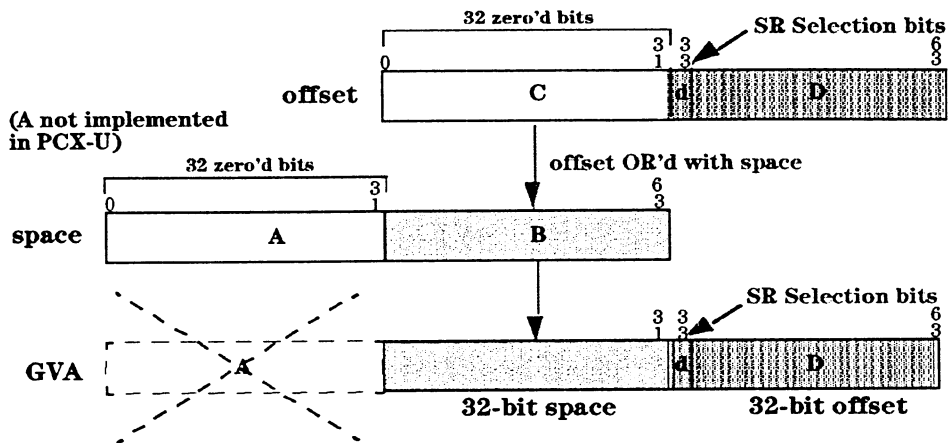
By restricting the virtual memory offsets to 4TB per quadrant, HP-UX allows itself the highest 22 bits of a 32-bit word in which to allocate a space ID (Currently implementation of PA-RISC 1.1 only used 16-bits for a Space ID).

The GVA is then formed by logically ORing the space ID with the offset. The next two slides contain details of the formation of both Narrow and Wide Mode PA-RISC 2.0 GVAs.

As we will see later, the GVA is used to determine the Virtual Page Number used to access the TLB and various internal memory structures such as the Page Directory (PDIR) Table.

Slide: PA-RISC 2.0 GVA Formation: Narrow Mode

PA-RISC 2.0 GVA Formation: Narrow Mode



a69640

Notes:

Slide: PA-RISC 2.0 GVA Formation: Narrow Mode

For PA-RISC 2.0 Narrow Mode the space identifier is combined with the offset to form a complete global virtual address. In general for both Wide and Narrow Mode, the offset and space portions are aligned as shown, and the low 32 bits of the space identifier are ORed together with the top 32 bits of the offset to form the GVA.

In Narrow Mode, bits 32 and 33 of the offset (“d” in the example) are space register selection bits for use in short pointer addressing. (Short pointers are discussed in more detail later in this module.)

When operating in Narrow Mode (the PSW W bit is 0), bits 0 to 31 of the offset are zeroed out. Then the GVA is formed by ORing the top 32-bits offset with the lower 32-bits of the space. Since the offset contains zeros, this is equivalent to concatenating the lower 32 bits of the space with the lower 32 bits of the offset.

Note that the first 32 bits of the space (bits 0 - 31; “A” in the slide) are not implemented by current PA-RISC 2.0 systems and are not used by HP-UX.

For example, below is a virtual address in the typical *space.offset* notation:

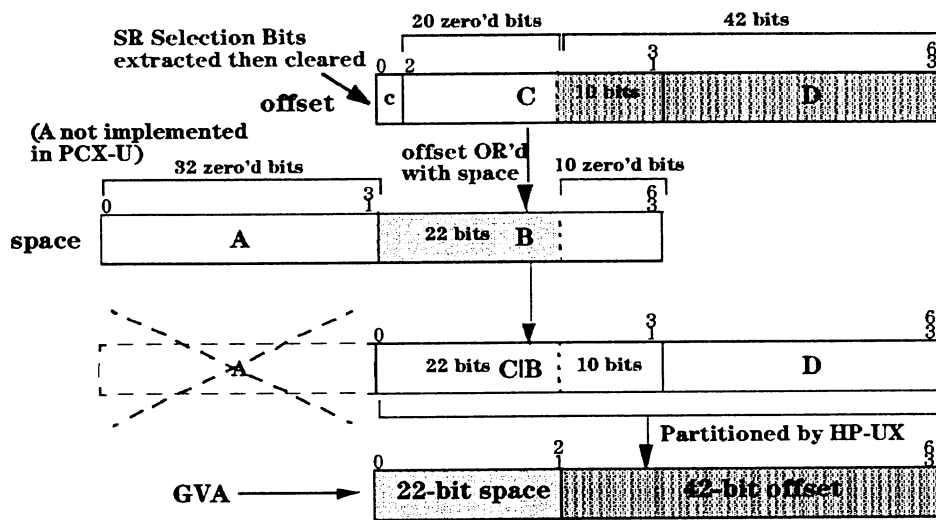
0x0e352c00.0xffffc8200

First we align the space and offset as follows and OR them together:

offset	0x00000000'7ffe6000
space	0x0e352c00
	=====
GVA	0x0e352c00'7ffe6000

Note there is no noticeable difference in the GVA and the “virtual address”.

PA-RISC 2.0 GVA Formation: Wide Mode



a69641

Notes:

Module 2 — System Architecture

Slide: PA-RISC 2.0 GVA Formation: Wide Mode

In general, the Wide Mode the GVA is created in the same way as in Narrow Mode. However, there are some differences.

First, the offset and space are aligned as shown. In Wide Mode bits 0 and 1 of the offset are space register selection bits. These bits are first extracted and then cleared from the offset. These bits are not needed in the GVA since they are only used to determine the appropriate space register for the Space ID. The top 32 bits of the modified offset (C) are then OR'd with the low 32-bit space register (B) in the same manner as for narrow mode, producing a 64-bit GVA.

As in Narrow Mode, the first 32 bits of the space (bits 0 - 31; "A" in the slide) are not implemented by all PA-RISC 2.0 systems and are not used by HP-UX.

Because the top 32-bits of the remaining 64-bit GVA are created in wide mode by ORing the space with the top 32-bits of the offset, it is the software's responsibility to partition the space ID and offset. (If this soft partition was not implemented, reconstruction of the initial space and offset from the GVA would be significantly more complex.)

Note that before ORing the offset and space, bits 2 to 21 (20 bits) of the offset and bits 54 to 63 (10 bits) of the space are zeroed out. This means that ORing the space and offset effectively concatenates bits 32 to 53 (22-bits) of the space with bits 22 to 63 (42-bits) of the offset.

HP-UX thus neatly partitions the top 22 bits of the 64-bit GVA for the space and the remaining 42 bits for the offsets. This results in 42 bits (4 Terabytes) of room in each quadrant. Since each process has access to 4 space registers, the total address space per process is 16 TB.

For example, below is a virtual address in the typical *space.offset* notation:

0x0624b400.0x400003ff'fffc8200

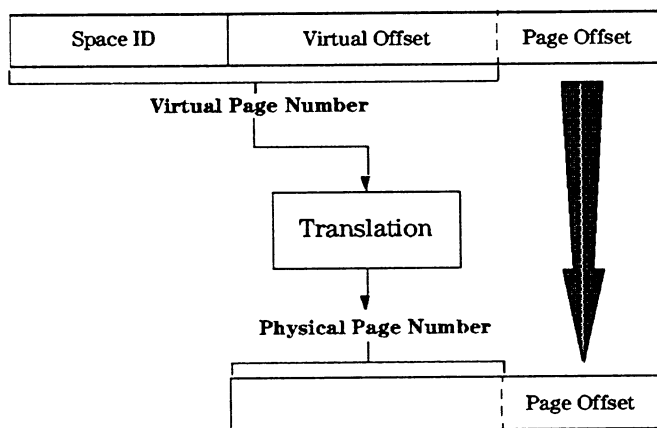
First we clear the space register selection bits since we know what the space ID is already, then we align the space and offset as follows and OR them together:

offset	0x000003ff'fffc8200
space	0x0624b400
	=====
GVA	0x0624b7ff'fffc8200

Note that once the space ID and offset are aligned, its easy to OR the 2 values. Each byte is essentially brought down except the byte represented by bits 20-23 in the GVA as outlined above. Now we have a global virtual address (GVA) that can be held in a single 64-bit register.

Slide: Virtual to Physical Address Translation

Virtual to Physical Address Translation



a60642

Notes:

Left blank intentionally

Module 2 — System Architecture

LAB: Virtual Addresses: PA-RISC 1.1

Estimated Time: 20 minutes, do either the PA1.1 example or the PA2.0 example

The register context for a process is shown below followed by a list of virtual addresses and a quadrant/space map. For each of the virtual address given, determine which space register will be used to evaluate the address and mark where the address will lie in the quadrant/space map.

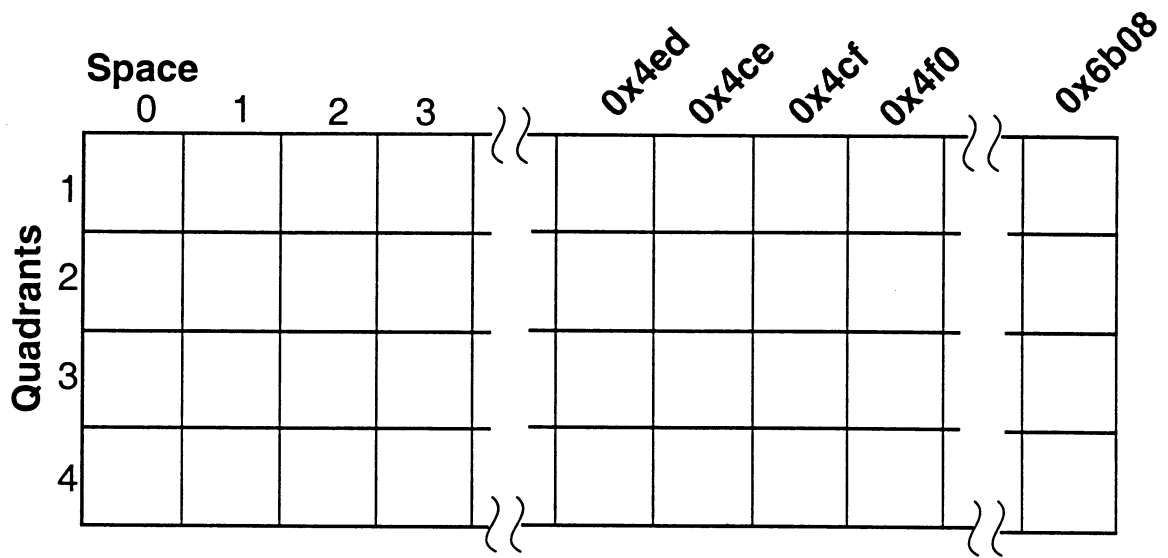
sp	=0x7ffe6b78	rp_flag	=0x00000001	gr0	=0x00000000
gr1	=0x00000001	rp	=0x00227b68	gr3	=0x00001770
gr4	=0x00000000	gr5	=0x00511c38	gr6	=0x02894ab8
gr7	=0x0265b860	gr8	=0x028b23b8	gr9	=0x012abcc0
gr10	=0x00000001	gr11	=0x000000f2	gr12	=0x00000000
gr13	=0x0000004d	gr14	=0x000000fd	gr15	=0xf0100000
gr16	=0xf0000fe0	gr17	=0xf0000074	gr18	=0xf000006c
gr19	=0x00511df8	gr20	=0x00000000	gr21	=0xffffffff00
gr22	=0x00444b20	arg3	=0x0000000f	arg2	=0x00000000
arg1	=0x7ffe6af8	arg0	=0x0002e264	dp	=0x00489a48
ret0	=0xffffffff0	ret1	=0x00000000	sp	=0x7ffe6b78
gr31	=0x00511a48	cr0	=0x00000000	cr1	=0x00000000
cr2	=0x00000000	cr3	=0x00000000	cr4	=0x00000000
cr5	=0x00000000	cr6	=0x00000000	cr7	=0x00000000
cr8	=0x00000000	cr9	=0x0000d610	cr10	=0x000000c0
cr11	=0x00000000	cr12	=0x00000000	cr13	=0x00000000
cr14	=0x00017800	cr15	=0xffffffff0	cr16	=0x19347dad
pcsqh	=0x00000000	pcoqh	=0x00000000	cr19	=0x00000000
cr20	=0x00000000	cr21	=0x00000000	cr22	=0x00000000
cr23	=0x00000000	cr24	=0x00444b20	cr25	=0x00208000
cr26	=0xffffffff0	cr27	=0x7ffe6bb8	cr28	=0x000001c0
cr29	=0x00000000	cr30	=0x40018e60	cr31	=0x0000ffff
sr0	=0x00000000	sr1	=0x0000537b	sr2	=0x00000000
sr3	=0x00000000	sr4	=0x00000000	sr5	=0x00006b08
sr6	=0x000004cf	sr7	=0x000004ce	IIA space	=0x00000000

Virtual Addresses:

- | | <u>Space Registers</u> | <u>Space ID</u> |
|----|------------------------|-----------------|
| A) | 0x400D9830 | |
| B) | 0x824D9320 | |
| C) | 0x7B03A000 | |
| D) | 0x3994A000 | |
| E) | 0x0.0000F000 | |
| F) | 0xC0000008 | |
| G) | 0x4F0.FFFFFFFF4 | |

Module 2 — System Architecture

Mark where each of the previous virtual addresses lie in the following quadrant/space map.



Module 2 — System Architecture

Lab: Implicit Pointers: PA-RISC 2.0

The register context for a process is shown below followed by a list of implicit pointers and a quadrant/space map. For each of the implicit pointer given, determine which space register will be used to evaluate the address and mark where the address will lie in the quadrant/space map.

r0 /r1 /r2	0'00000000	0'00000001	0'002776ac
r3 /r4 /r5	0'00000009	0'00000003	0'012b0048
r6 /r7 /r8	0x400003ff'ffced0	0'00000009	0'000338ac
r9 /r10/r11	0x400003ff'ffcf2a0	0xffffffff'ffffffff	0'00000009
r12/r13/r14	0'000003ec	0x7'00000000	0'00000000
r15/r16/r17	0'00000001	0'00000080	0'40001270
r18/r19/r20	0'00000000	0'00000009	0'005d1e80
r21/r22/r23	0'00000001	0'005cc600	0'0800000f
r24/r25/r26	0'00000000	0'0000000c	0'00509778
r27/r28/r29	0'0063c060	0'00000003	0x400003ff'ffcf540
r30/r31/r32	0x400003ff'ffcf550	0'00000009	
sr0/sr1/sr2	0'0b0cac00	0'0d41d800	0'0d41d800
sr3/sr4/sr5	0'00000000	0'03532c00	0'0d41d800
sr6/sr7/sr8	0'08948800	0'03040000	

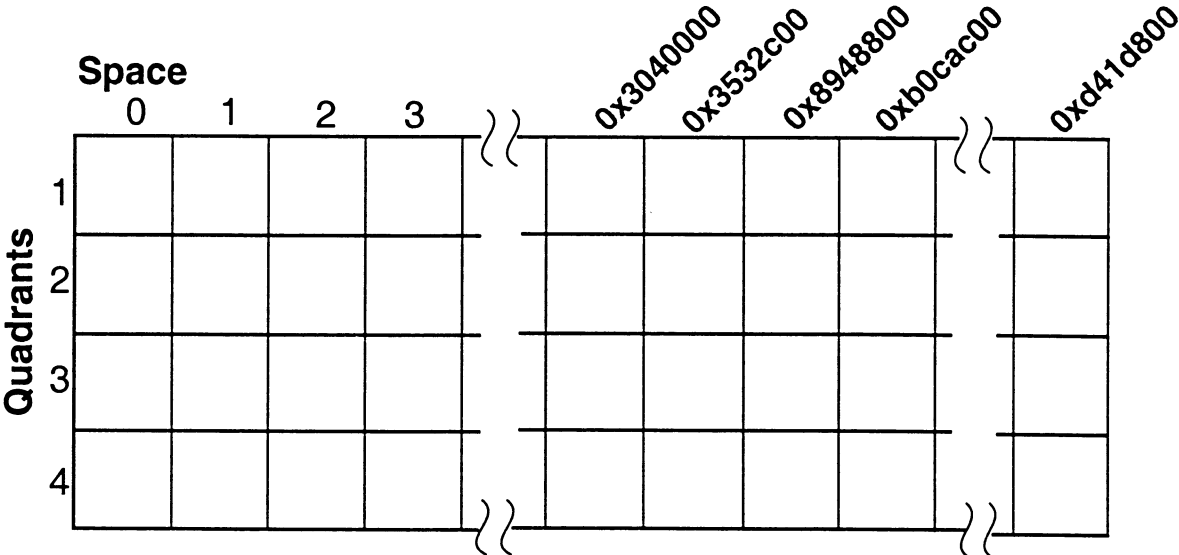
Implicit Pointers

	<u>Space Registers</u>	<u>Space ID</u>
A)	0x00000378'00ff4532	
B)	0x80000000'000002c4	
C)	0x40000220'a1374f0c	
D)	0xc0000243'0a452e1c	

Module 2 — System Architecture

Lab: Implicit Pointers: PA-RISC 2.0

Mark where each of the previous virtual addresses lie in the following quadrant/space map.



Module 2 — System Architecture

Solution: Virtual Addressing: PA-RISC 1.1

Estimated Time: 20 minutes

The register context for a process is shown below followed a list of virtual addresses and a quadrant/space map. For each of the virtual address given, determine which space register will be used to evaluate the address and mark where the address will lie in the quadrant/space map.

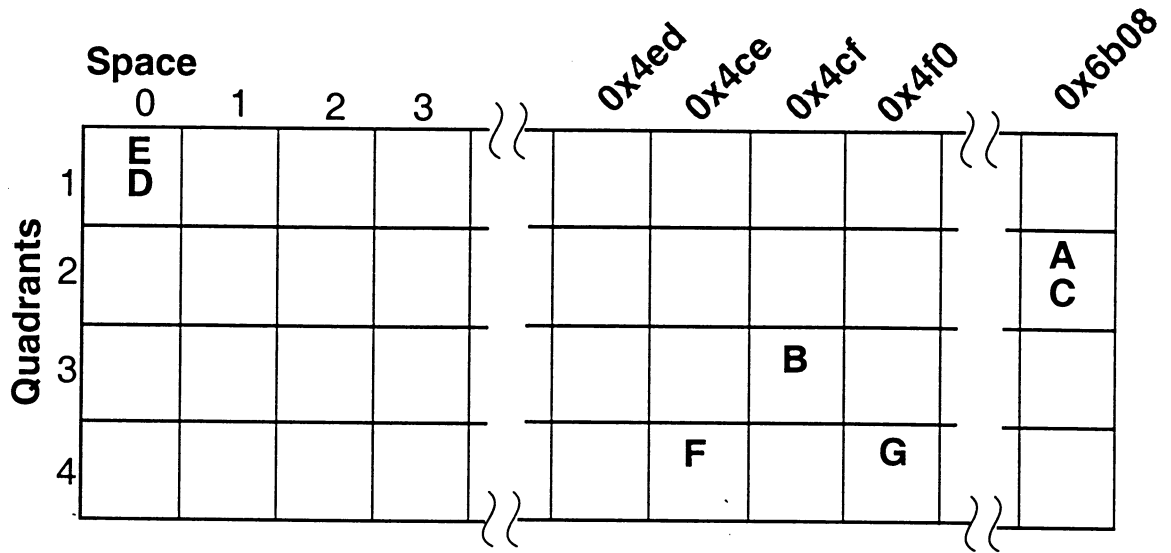
sp	=0x7ffe6b78	rp_flag	=0x00000001	gr0	=0x00000000
gr1	=0x00000001	rp	=0x00227b68	gr3	=0x00001770
gr4	=0x00000000	gr5	=0x00511c38	gr6	=0x02894ab8
gr7	=0x0265b860	gr8	=0x028b23b8	gr9	=0x012abcc0
gr10	=0x00000001	gr11	=0x000000f2	gr12	=0x00000000
gr13	=0x0000004d	gr14	=0x000000fd	gr15	=0xf0100000
gr16	=0xf0000fe0	gr17	=0xf0000074	gr18	=0xf000006c
gr19	=0x00511df8	gr20	=0x00000000	gr21	=0xffffffff00
gr22	=0x00444b20	arg3	=0x0000000f	arg2	=0x00000000
arg1	=0x7ffe6af8	arg0	=0x0002e264	dp	=0x00489a48
ret0	=0xffffffff0	ret1	=0x00000000	sp	=0x7ffe6b78
gr31	=0x00511a48	cr0	=0x00000000	cr1	=0x00000000
cr2	=0x00000000	cr3	=0x00000000	cr4	=0x00000000
cr5	=0x00000000	cr6	=0x00000000	cr7	=0x00000000
cr8	=0x00000000	cr9	=0x0000d610	cr10	=0x000000c0
cr11	=0x00000000	cr12	=0x00000000	cr13	=0x00000000
cr14	=0x00017800	cr15	=0xffffffff0	cr16	=0x19347dad
pcsqh	=0x00000000	pcoqh	=0x00000000	cr19	=0x00000000
cr20	=0x00000000	cr21	=0x00000000	cr22	=0x00000000
cr23	=0x00000000	cr24	=0x00444b20	cr25	=0x00208000
cr26	=0xffffffff0	cr27	=0x7ffe6bb8	cr28	=0x000001c0
cr29	=0x00000000	cr30	=0x40018e60	cr31	=0x0000ffff
sr0	=0x00000000	sr1	=0x0000537b	sr2	=0x00000000
sr3	=0x00000000	sr4	=0x00000000	sr5	=0x00006b08
sr6	=0x000004cf	sr7	=0x000004ce	IIA space	=0x00000000

Virtual Addresses:

		<u>Space Registers</u>	<u>Space ID</u>
A)	0x400D9830	SR5	0x6b08
B)	0x824D9320	SR6	0x4cf
C)	0x7B03A000	SR5	0x6b08
D)	0x3994A000	SR4	0x0
E)	0x0.0000F000	None	0x0
F)	0xC0000008	SR7	0x4ce
G)	0x4F0.FFFFF40	None	0x4f0

Solution: Virtual Addressing: PA-RISC 1.1

Mark where each of the previous virtual addresses lie in the following quadrant/space map.



Module 2 — System Architecture

Solution: Lab: Implicit Pointers: PA-RISC 2.0

Estimated Time: 20 minutes

The register context for a process is shown below followed by a list of implicit pointers and a quadrant/space map. For each of the implicit pointers given, determine which space register will be used to evaluate the address and mark where the address will lie in the quadrant/space map.

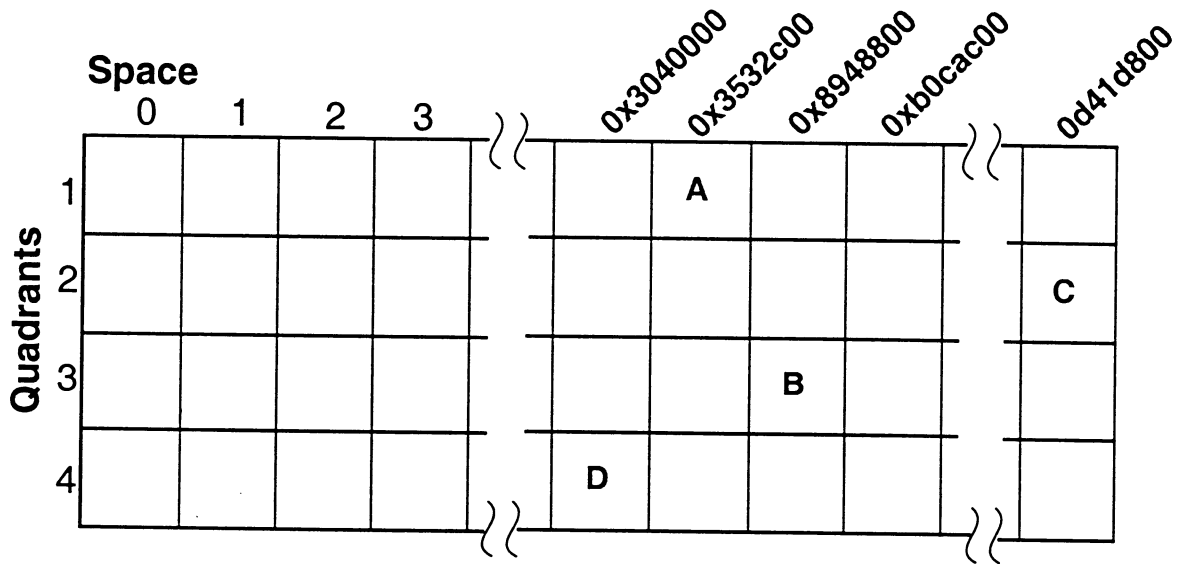
r0 /r1 /r2	0'00000000	0'00000001	0'002776ac
r3 /r4 /r5	0'00000009	0'00000003	0'012b0048
r6 /r7 /r8	0x400003ff'fffcede0	0'00000009	0'000338ac
r9 /r10/r11	0x400003ff'fffcf2a0	0xffffffff0'ffffffff	0'00000009
r12/r13/r14	0'000003ec	0x7'00000000	0'00000000
r15/r16/r17	0'00000001	0'00000080	0'40001270
r18/r19/r20	0'00000000	0'00000009	0'005d1e80
r21/r22/r23	0'00000001	0'005cc600	0'0800000f
r24/r25/r26	0'00000000	0'0000000c	0'00509778
r27/r28/r29	0'0063c060	0'00000003	0x400003ff'fffcf540
r30/r31/r32	0x400003ff'fffcf550	0'00000009	
sr0/sr1/sr2	0'0b0cac00	0'0d41d800	0'0d41d800
sr3/sr4/sr5	0'00000000	0'0e532c00	0'0d41d800
sr6/sr7/sr8	0'08948800	0'03040000	

Implicit Pointers

	<u>Space Registers</u>	<u>Space ID</u>
A) 0x00000378'00ff4532	SR4	0'0e532c00
B) 0x80000000'000002c4	SR6	0'08948800
C) 0x40000220'a1374f0c	SR5	0'0d41d800
D) 0xc0000243'0a452e1c	SR7	0'03040000

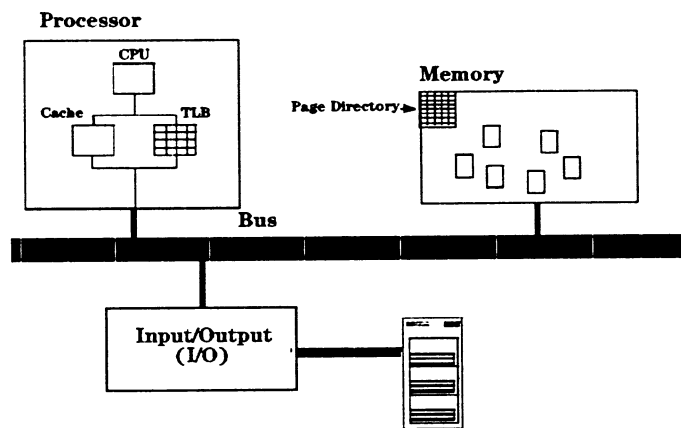
Solution: Lab: Implicit Pointers: PA-RISC 2.0

Mark where each of the previous virtual addresses lie in the following quadrant/space map.



Slide: Address Translation Components

Address Translation Components



a69643

Notes:

Slide: Address Translation Components

Virtual address translation is a multi-stage process involving several different components of the processor and the operating system. The bulk of data in the system is kept on secondary storage such as a disk. As pages of data are needed they must be brought in from disk to memory in such a way that the CPU knows the location of the pages.

The operating system maintains a table in memory called the **Page Directory (PDIR)** which keeps track of all the pages currently in memory. A subset of the PDIR is visible by the hardware.¹ When a page is brought into memory from disk, it is allocated an entry in the PDIR. The PDIR is what links a physical page in memory to its virtual address.

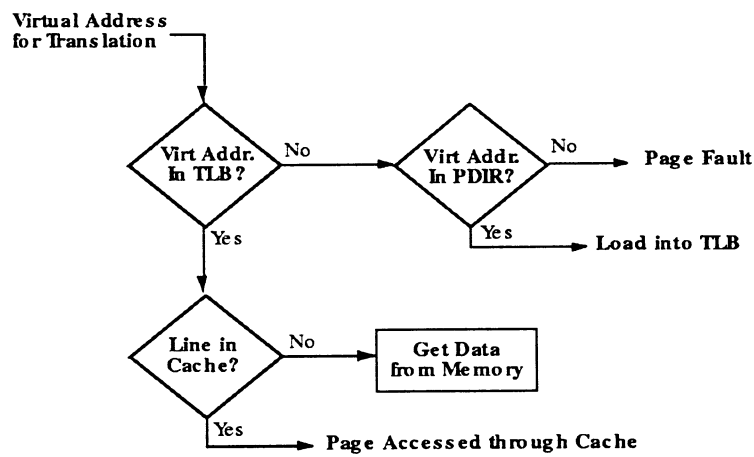
Within the processor the TLB holds a subset of entries from the PDIR. The TLB can be thought of as a cache for address translations. Only the most recently and hopefully most frequently used translations are kept in the TLB.

Address translation is handled from the top of this hierarchy hitting the fastest component first (the TLB) and then moving on to the PDIR and secondary storage.

1. The hardware visible portion of the PDIR (PA-RISC 1.1 only) will be discussed later as part of the TLB miss handler.

Slide: Address Translation Through TLB and Cache

Address Translation Through TLB & Cache



a69644

Notes:

Slide: Address Translation Through TLB and Cache

When a virtual address is requested by the CPU, the TLB, Cache, and PDIR are used to

- Translate the virtual address to a physical address
- Obtain the contents of the address.

First, the TLB is searched for a match of the virtual address. If the VPN is in the TLB we have a **TLB hit**, if not it is called a **TLB miss**. On a TLB hit, we know the worst case is that the data is in memory but not in cache. The cache is searched for the corresponding cache line containing the desired data. If it is in the cache, we use it. If not in cache, we bring the data into cache from memory.

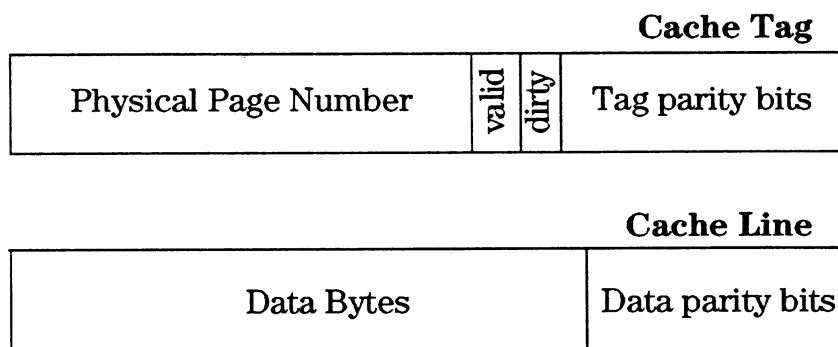
On a TLB miss, the PDIR is checked. If the VPN is found in the PDIR we know the page is in memory and update the TLB accordingly. If the VPN is not in the PDIR, then we page fault to bring the page in from secondary storage.

All memory access is granted through the TLB and cache. I/O access goes through the TLB but not the cache. On any TLB or Cache miss, the TLB and cache are updated from the PDIR or memory, respectively. Then the search process is restarted from the beginning.

We will continue to discuss details of virtual address translation through the TLB and Cache. Details of the PDIR and page faulting will be discussed in the Memory Management module since they are software services.

Slide: Cache Entry

Cache Entry



a69645

Notes:

Slide: Cache Entry

Cache Organization

Cache memory is divided into blocks called **cache lines**. The cache lines define the size of the unit of data that is passed between cache and main memory. For example, a PCX-U processor has a cache line of 32 bytes.

For each cache line, there is an associated **cache tag** to describe the contents. The tag typically contains

- **Physical Page Number:** Identifies page in memory that the data came from
- **Flags:**

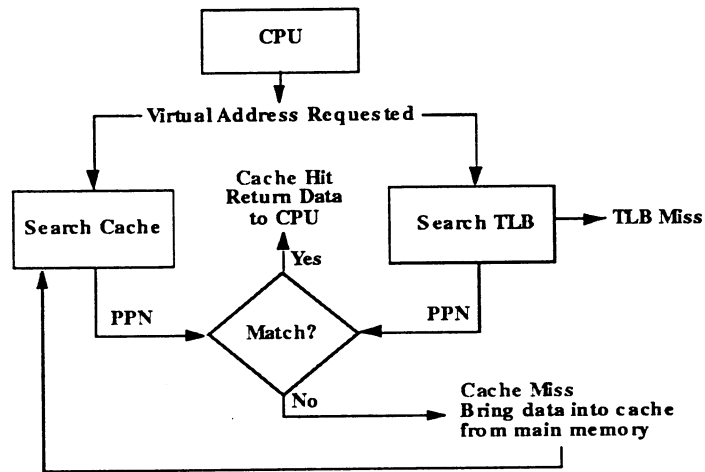
Valid	Indicates whether cache line contains valid data
Dirty	Indicates that CPU has modified contents of cache line
Others	Specific to implementation

Both the cache tag and cache line typically have parity bits associated with them as well. The cache line and cache tag together make up a cache entry.

The cache tags are used to check whether a particular cache line with the desired data is present in the cache. Like the TLB, a processor may have a unified cache or separate caches for instructions and data.

Slide: Searching the Cache

Searching the Cache



a69646

Notes:

Slide: Searching the Cache

Since the cache contains copies of only some of the data from main memory, it is necessary to search the cache to see whether the requested data is present.

There are a number of different techniques used to search with different PA processors. The great majority of PA processors have single level caches implemented externally to the CPU. This has had the advantage of allowing large caches to be implemented, and also cache sizes to be varied with system cost.

External caches however need to be simple in their design in order to be able to run at the speed of the processor, and also to avoid excessive pin counts on the processor chip.

The PA7200 processor implements two caches with different designs in parallel, as we shall see this allows the caches to cover-up for each other weaknesses.

The PA7300LC implements a small on chip first level cache to allow low cost systems to be built without the added complexity of needing to implement the cache on the main PCB. Although it allows a larger second level cache to be added to increase performance.

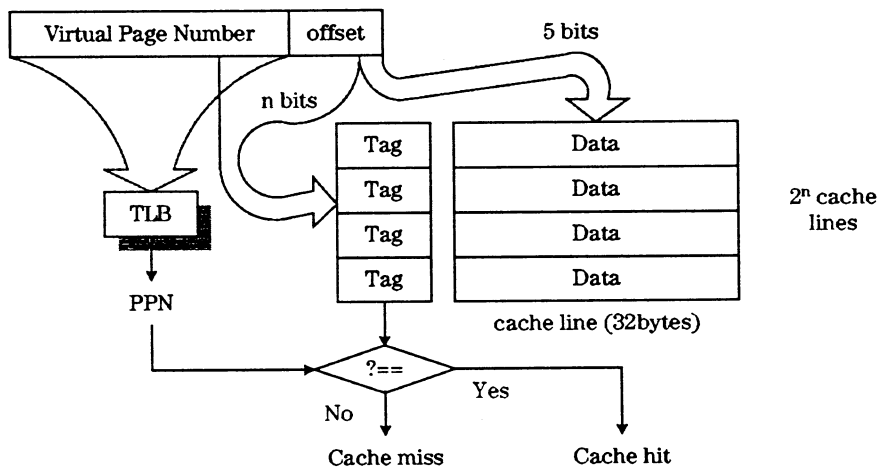
The PA8500 implements a large (1MB for data & 0.5MB for instructions) on chip cache since it runs at clock speeds that make building an off chip cache that is able to run at the full CPU speed impractical.

Over the next few slides we will take a look at the different cache designs that are used and discuss their strengths and weaknesses.

Slide: Direct Mapped Caches

Direct Mapped Caches

Supra-architeta
Bapucant



a69647

Notes:

Slide: Direct Mapped Caches

With a direct mapped cache a number of the bits of the virtual address are used as an index into the cache. These bits select the cache line to be searched. The physical address from the tag associated with this cache line is then compared with the physical address return from the TLB. A cache hit results if these two physical addresses match.

The virtual address is used as the index as this allows the searching if the cache and the TLB to be started in parallel, which saves time.

The physical address is used in the tag so that it is possible implement address aliasing whereby more than one virtual page in mapped to the same physical page¹.

The virtually indexed, physically tagged direct mapped cache has the advantage of simplicity, and speed. It has therefore allowed the cache to be implemented separately from the CPU. This allows large for very large caches (up to 2*2MB in the case of the PA8200) be implemented without the need to build massive processor chips.

The design does however have some weaknesses. Since the clock speeds of processor chips are increasing far faster than the access times of memory circuits there is a limit on how fast the processor can be. The PA8500 processor exceeds this limit and must therefore use a on chip cache. Moving the cache on chip not only allows for faster access but also makes a more complex design practical since there is not the same limitation on the number of connections.

The biggest weakness of the direct mapped design is that by using part of the virtual address to index a single cache entry, any memory access that has the same value of bits in that part will always attempt to access the same cache entry. This can be a major problem if the CPU is working on more than one area of memory and the address in these areas share the same sets bits in the range used by the cache indexing. This would happen if a program were to copy an area of memory and the source and target addresses were an integer multiple of the size of the cache apart.

```
EG      #define CACHESIZE (2*1024*1024)
        char buffer1[CACHESIZE],buffer2[CACHESIZE];
        ...
        ...
        for (i=0; i<CACHESIZE; i++) buffer1[i]=buffer2[i];
```

In this example, every attempt to write to buffer1 will need to go through the same cache entry as the preceding read from buffer2.

1. For more details of the use of address aliasing see the discussion on memory maps in the process management chapter and also the memory management chapter.

Slide: Direct Mapped Caches

Whilst this causes major performance problems¹ it is not difficult for the programmer, once aware of the problem to program their way around the it. A simple solution is to add some padding between the two buffers, so that corresponding entries in buffer1 and buffer2 no longer map to the same lines within the cache.

A better solution is to vectorise the code so that instead of processing it byte by byte (somewhat of a waste of a 32 or 64 bit CPU), processing it in whole cache lines. If the code were to read say two whole cache lines from buffer2 and then write them to buffer1 the conflict over the cache line would not matter since by the time the data was written to buffer1, buffer2's data would no longer be needed in the cache as would not be re-accessed.

Avoiding these problems relies on the programmer: -

- Being aware that the these problems might exist, and many programmer prefer to see the CPU as a magic black box the carries out their instructions, without having to worry about how.
- Knowing how to avoid the problems.
- Knowing exactly how the cache is implemented in the users system
- Knowing how many CPU registers can efficiently be used.

Программист может легко
устранить такую проблему,
так что это стоит
иметь в виду.

1. See the cache0 example program if your system has a direct mapped cache.

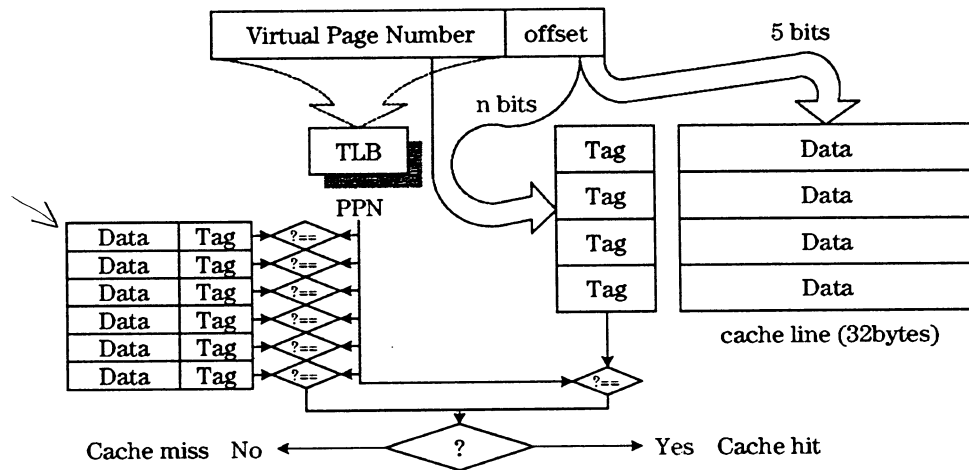
Left blank intentionally

Slide: Assist Cache

Assist Cache

PA7200

A3y
2x8



a69648

Notes:

Slide: Assist Cache

The other extreme of cache design from the directed mapped cache is the fully associative design. In the direct mapped design one a single cache line has it's tag compared to ascertain whether a cache hit has been achieved. This is simple to implement as only a single comparator is needed.

In the fully associative design every single entry with the cache has it's tag compared. If there was only a single comparator circuit available then the cache would need to be searched serially and this would be very slow. So to implement this design a comparator is provided for each line within the cache. This leads to the cache design being much more complex, but avoids the conflicts that happen with the direct mapped cache. Since now any slot in the cache will do to hold any piece of data.

In practice the increased complexity of implementing a fully associative design tend to severely limit their size, and in so doing increase the chances of a cache move as the cache can not hold as much data.

The PA7200 uses both of these designs in parallel. It implements an on chip fully associative cache called the assist cache. This is only 2KBytes in size. The main cache is then still an external direct mapped design.

If we now look at how the previous example would behave in the this environment we will see that the assist cache is able to deal with the inherent weakness of the direct mapped design.

Buffer2 data is read from memory and stored in the external cache.

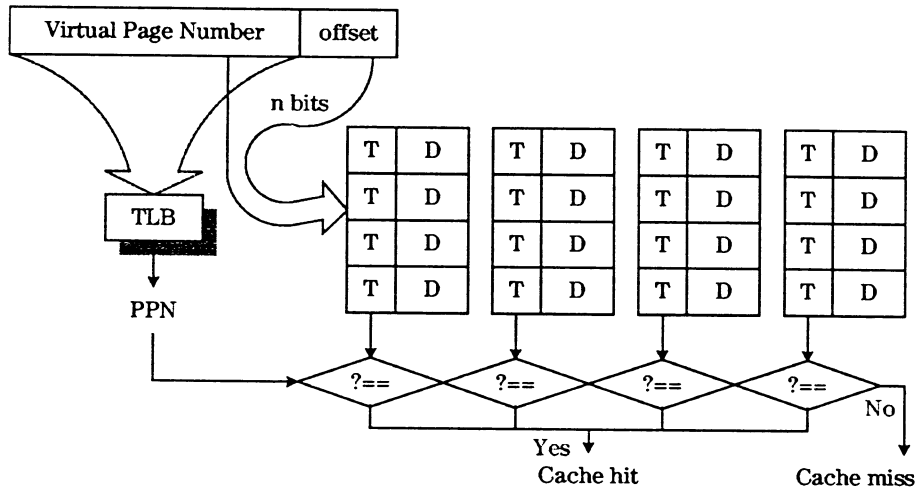
The write to the buffer1 location would then go to the same slot in the cache, causing the buffer2 data to be expelled. In this situation however instead of this data being lost, and therefore needing to be reread from memory next time around the loop, the data is moved into the assist cache.

The Next read from buffer2 can be satisfied from the assist cache, and therefore does not disturb the buffer1 data now stored in that location of the external. This saves the need to store the now dirty (modified) buffer1 data from the external cache, as well as needing to re-read the buffer2 data from memory. Hence saving two real memory accesses per iteration of the loop.

Slide: 4-way Associative Set Caches

DA8300

4-way Associative Set Caches



Membase A30
nozd gamebal
game eam LIX
ne wabld.

a69649

Notes:

Slide: 4-way Associative Set Caches

Where the cache is implemented internally it is easier to use more complex designs than the direct mapped one, but the complexity of the fully associative design still makes it impractical for all but the smallest caches.

A half way house solution is still to use part of the virtual address for the index, but to then compare more than one line. This is known as an associative set design. With the current processors, only the PA7300LC and PA8500 use this technique, and they both use a 4way associative set, that is there are 4 comparators and 4 cache lines are checked.

This then allows the simple case such as memory copies to avoid cache conflicts, as the source and target can be in different sets. It can not deal with all problem case so that: -

```
target[] = source1[] + source2[] + source3[] + source4[]
```

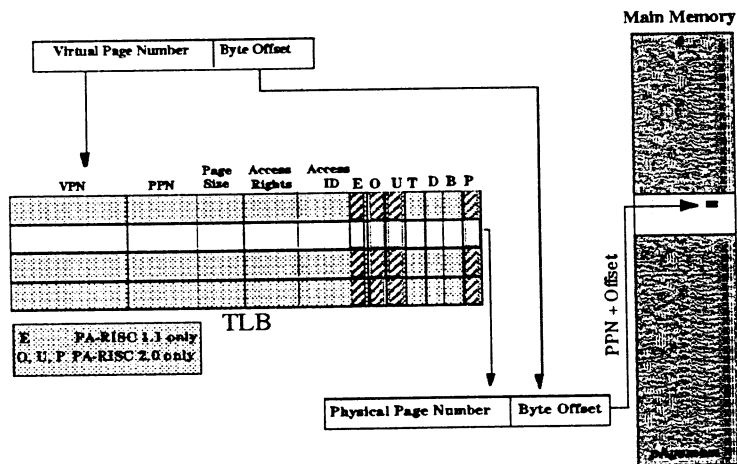
where each of the arrays were aligned in memory on a multiple of the cache size/(number of associative sets).

The memory copy situation is however far more common.

One possible solution is to
use a 4-way associative cache
with 4 comparators, but solving
this problem is very
difficult

Slide: Translation Lookaside Buffer

Translational Lookaside Buffer



a69650

Notes:

Slide: Translation Lookaside Buffer

The TLB is a hardware unit that serves to convert a Virtual Page Number (VPN) to a Physical Page Number (PPN). In an ideal case our TLB would be large enough to hold translations for every page of physical memory. This would be expensive so instead the TLB holds only a subset of all translations, and in practice the most recently accessed ones.

Given a Virtual Page Number and byte offset, the TLB is searched for a match and the Physical Page Number associated with the VPN is returned. This PPN combined with the byte offset locate the desired data in memory.

Because the purpose of the TLB is to satisfy virtual to physical address translation, the TLB is only searched when memory access is requested while in virtual mode, which is indicated by the D-bit in the PSW (or the I-bit for instruction access).

TLB Organization

There are two ways of organizing the TLB on the processor

- **Unified TLB** - A single TLB holds translations for both data and instructions.
- **Split Data and Instruction TLB** - There are two TLB units in the processor each of which hold translations specifically for data or instructions.

At one time many systems were being designed with split Data TLB (DTLB) and Instruction TLB (ITLB). This was to account for the fact that data and instructions have different characteristics in terms of their locality so that instruction translations would not flush frequently accessed data translations.

The purpose between having a split data and instruction TLB is to compensate for the different locality characteristics of each. By splitting the TLB the frequent random access of data is not affected by the relatively sequential single usage of instructions. Over time cost factors have allowed the inclusion of much larger TLBs on processors which has lessened the disadvantages of a unified TLB. As a result you are likely to see the unified organization in many newer processors.

TLB Entries

Since the TLB's purpose is to satisfy the translation from virtual to physical addresses, each entry contains both the Virtual Page Number (VPN) and the Physical Page Number (PPN).

Along with the VPN and PPN the TLB contains **Page Size**, **Access Rights**, an **Access Identifier**, and several **flags**.

The TLB entries are not physically viewable so the physical layout is not of much use to us. For reference the size of each field is as follows

	PA-RISC 1.1	PA-RISC 2.0
Virtual Page Number	36 bits	52 bits
Physical Page Number	20 bits	52 bits
Page Size	N/A	4 bits
Access Rights	7 bits	7 bits

Slide: Translation Lookaside Buffer

Access ID	15-18 bits	15-31 bits
Flags	6 bits	6 Bits

TLB Flags

For PA-RISC 1.1 there are 4 flags in each TLB entry:

- E** Entry Valid bit
- T** Page Reference bit
- D** Dirty Bit
- B** Break

The T, D, and B flags are only present in data or unified TLBs.

For PA-RISC 2.0 there are 6 flags in each TLB entry:

- O** Ordered
- U** Uncacheable
- T** Page Reference bit
- D** Dirty Bit
- B** Break
- P** Prediction

The O, U, T, D, and B flags are only present in data or unified TLBs.

Each flag is described in more detail below:

The **E bit** is the **valid bit** (PA-RISC 1.1 only) for each entry. If set, this bit indicates that the TLB entry reflects the current attributes of the physical page in memory. On PA-RISC 2.0, all entries in the TLB are assumed to be “valid” unless they have been “removed”. An entry in the TLB is **removed** when some action causes it to be inaccessible, such as a PURGE DATA TLB instruction. Exactly how an entry is removed is implementation specific. When the **Page Reference bit** (T) is set, any access to this page will cause a reference trap to be handled either by hardware or software trap handlers.

The **Ordered Bit** (PA-RISC 2.0 only) when 0 indicates that data memory references using this translation (except those explicitly marked as ordered or strongly ordered) may be weakly ordered. When 1 and the PSW[O] bit is 1, data memory references using this translation are ordered.

PA-RISC 1.1 specified that all loads and stores be performed in order (“strong ordering”). Future processors are expected to support multiple outstanding cache misses while simultaneously performing loads and stores to lines already in the cache. In most cases this effective reordering of loads and stores (“weak ordering”) causes no inconsistency and permits faster execution. Strongly ordered variants of loads and stores must be available to handle contexts in which ordering must be preserved (synchronization among processors, I/O activities).

The **Uncacheable bit** (PA-RISC 2.0 only) controls whether reference to this page in memory causes the page to be moved into the cache. This bit is commonly set to 1 for pages which map to I/O address space to prevent them from being brought into cache.

Slide: Translation Lookaside Buffer

When the **Page Reference bit** is set, any access to this page will cause a reference trap to be handled either by hardware or software trap handlers.

The **Dirty bit** when set, indicates that the associated page in cache is different than the same page in physical memory. This is the indication that the page must be flushed before being invalidated.

The **Break bit** causes a trap on any instruction that is capable of writing to this page.

When the **Prediction bit** (PA-RISC 2.0 only) is 0, branch prediction is performed based on static prediction hints encoded in the instructions. When 1, prediction is performed based on dynamic prediction hardware, on implementations so equipped. This bit functions solely as a performance hint. Implementation of the P-bit is optional. Since the P bit controls branch prediction, it is only implemented in the ITLB or combined TLB.

TLB Page Size Support

The TLB supports a range of page sizes, in multiples of four, from 4 Kbytes to 64 Mbytes. Each page is aligned to an address which is an integer multiple of its size. The page size is inserted into the TLB with each translation, and is encoded as shown in the following table. TLB purge instructions can also specify a page size, allowing a large contiguous address range to be purged in a single instruction.

Encoding	Page size
0	4 KB
1	16 KB
2	64 KB
3	256 KB
4	1 MB
5	4 MB
6	16 MB
7	64 MB
8 - 15	Reserved

TLB Page Protection

Since we are going to be satisfying translations from the TLB, we also need a way to control this access and ensure that a user process only sees data it is privileged to receive. The TLB contains access rights and protection identifiers. The **Access Rights** are stored in a seven bit field containing the access type allowed for the page and the two privilege levels.

The **Access Type** field specifies read, write, and execute permissions for the page. *PA-RISC 2.0*

Slide: Translation Lookaside Buffer

Architecture lists all the possible values allowed by the architecture. In the Memory Management module we will discuss details of those allowed by HP-UX.

All PA-RISC instructions execute at one of four privilege levels (0,1,2,3) with 0 being the most privileged. This **privilege level** is stored in the least significant two bits of each executing instruction. The privilege level for the executing instruction is compared to one of the privilege levels in the Access Rights of the TLB.

The first **Privilege Level** (PL1) is checked for read access and the second (PL2) is checked for write access. For each of these the privilege level of the executing instruction must be at least as high (0 being the highest) as the appropriate PL field in the TLB. For execute access the instruction's privilege level must be at least as high as PL1 but not higher than PL2.

For HP-UX, only two privilege levels are implemented. Privilege level 0 represents kernel execution and privilege level 3 represents user execution.

To allow for privilege level promotion PA-RISC includes a **BRANCH instruction** with the GATE (for gateway) completer. This instruction is used to branch to perform an interspace branch while increasing the privilege level. The most common example of this in HP-UX is a system call. This instruction is used to branch to the procedure associated with the system call and change the privilege level from user to system.

The Access Rights are only part of the privilege check in the TLB. The process must also pass a check of the **Protection Identifiers**.

Each process has a number of protection IDs associated with it. PA-RISC allows up to four (PA-RISC 1.1) or eight (PA-RISC 2.0) of these IDs to be held in Control Registers 8, 9, 12 and 13. The protection IDs are 32 bits each with the least significant bit being a Write Disable (WD) bit. If this bit is zero, writes that match the protection ID are allowed.

When accessing a page referenced in the TLB, all protection IDs are compared with the Access ID in the TLB entry. If any of the protection IDs match, the check is validated. If requesting a write, the Write Disable bit of one of the matching IDs must also be zero.

Block TLB (PA-RISC 1.1 only)

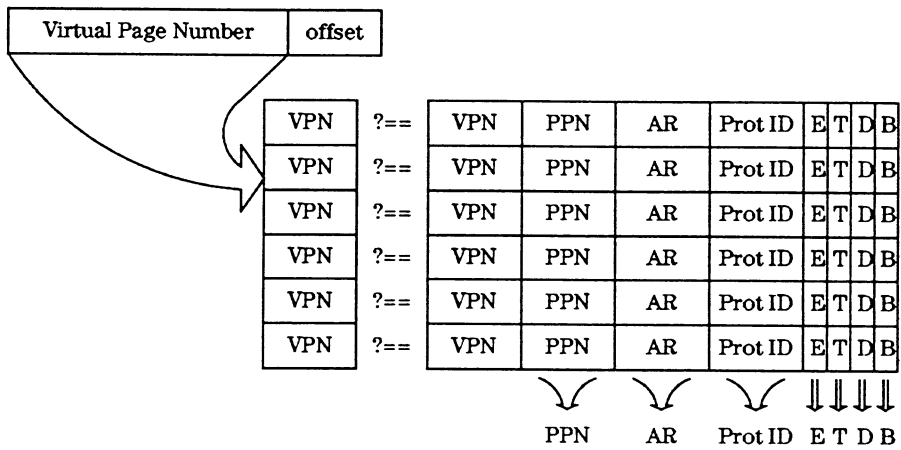
In addition the standard TLB which maps each entry to a single page of memory, many processors also have a block TLB. The block TLB is used to map entries to virtual address ranges larger than a single page and thus increases the overall address range of the TLB.

Since the operating system moves data in/out of memory by pages, a range of pages referenced by a block TLB entry are locked in memory and cannot be paged out. In the case of HP-UX, block TLB entries are used to reference kernel memory that remains resident and the framebuffer of the graphics cards.

Left blank intentionally

Slide: Searching the TLB

Searching the TLB



a69651

Notes:

Slide: Searching the TLB

The original PA1.0 processors used direct mapped design for the TLB, this allowed the TLB be made externally to the CPU from cache RAM chips, and to be large in size. With the TLB however it was not simple to program around the address conflicts that arise with the direct mapped designs so that in the move to PA1.1, this design was abandoned and a fully associative design was selected.

Whilst this design has the advantage of not suffering from conflicts its complexity necessitates a move to an on chip implementation and its size is severally limited.

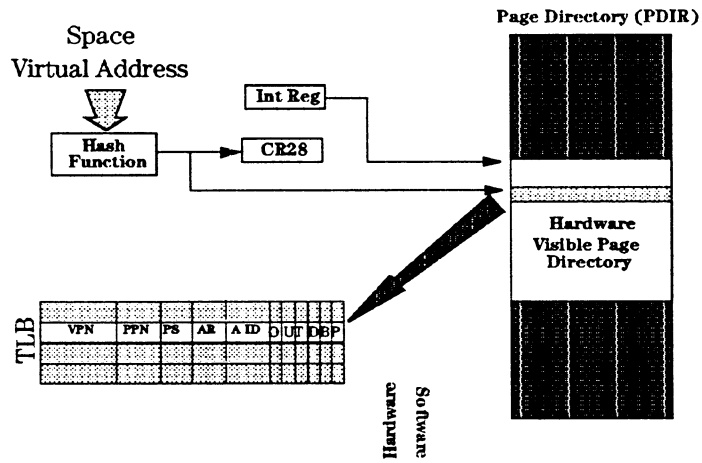
Now therefore more TLB misses occur dues to the limited range of the TLB.

There are several options for reducing the problem of TLB misses.

- improve the performance of handling TLB misses.
 - Interrupt handling features introduced with PA1.1 such as the shadow registers help here.
 - The PA1.1 processors are able to handle some TLB misses in hardware, see the later slide.
- increase the number of TLB entries implemented. The amount of space available on the chip is very limited, so this is not always feasible. However as can be seen from the PA2.0 processor line, attempts are made to do this, so that PA8000 has 96 entries, PA8200 120, and PA8500 expands this to 160 entries.
- Increase the page size that a entry maps.
 - PA1.1 increase the page size from 2K to 4K, so doubling the amount of data that can be held on the pages described in the TLB.
 - The PA1.1 also introduce the block TLB entries previously discussed, but these are not general purpose entries.
 - PA2.0 uses a variable page size for its translations, this allows a much greater range of memory to be accessed without sustaining TLB misses. For example with a 4K page size and a 120 entry TLB only 480K of data can be accessed through the TLB, whereas with a 64MB pages size 7.68GB can be accessed.

Slide: Hardware TLB Miss Handler (PA 1.1 Only)

Hardware TLB miss handler (PA1.1 Only)



a60652

Notes:

*Ma 2.0 TLB неперезумасел
неперезумасел*

Slide: Hardware TLB Miss Handler (PA 1.1 Only)

When an address translation is not found in the TLB, then we have encountered a **TLB Miss**. On a TLB Miss, the TLB must be updated with information about the desired page. This updating can be handled in one of two ways

- **Hardware TLB Miss Handler**
- **Software TLB Miss Handler**

Some PA-RISC systems have hardware miss handlers built in. These handlers can be enabled or disabled by software through PDC calls. In the case of HP-UX, the `enable_hw_tlb_miss()` is called to enable the hardware miss handling.

In the hardware miss handler, the hardware attempts to find the virtual to physical page translation in the Page Directory. If the hardware is unable to locate the translation in the PDIR, either because it does not exist or the hardware search algorithm is not exhaustive, then a trap occurs so that the software can carry out the process.

Accessing Page Directory (Hardware Walker)

In order for the hardware to handle the TLB Miss, the PDIR must be visible to the hardware. To make this possible the base address of the hardware visible portion of the PDIR is stored in an internal diagnostic register. We will see in the Memory Management module that the PDIR is accessed via a hash algorithm that may create collisions. Only the first level entries are visible to hardware. The hardware miss handler will not follow links for any hash collisions.

The hardware miss handler hashes the missing space and virtual address to obtain an offset into PDIR. This offset is merged with the value in the internal register (the PDIR base) to get the 32-bit real address of the desired entry. The cache line associated with this entry is brought into cache for validation.

Validating Page

The full structure of the PDIR is shown in the Memory Management module. As part of this structure we store a **Valid bit**, indicating the page is in memory, a VPN and PPN, and a **Reference bit**.

The PDIR entry located is considered valid and inserted into the TLB if

- The valid bit is set
- The VPN in the entry matches the desired VPN
- The reference bit is set (we do this to facilitate generating a trap for the VM system when the reference bit is not set).

If any of these checks fail, control is passed to software and the address of the hashed PDIR entry is

Slide: Hardware TLB Miss Handler (PA 1.1 Only)

placed in CR28. This is so that software need not go through the overhead of hashing again.

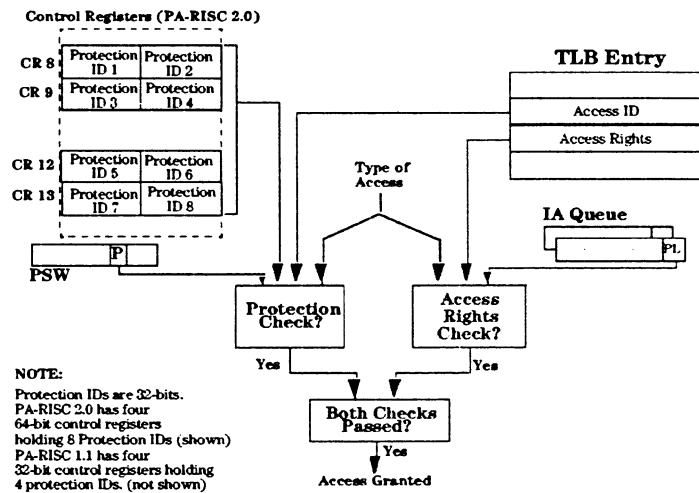
Notice that while satisfying the TLB Miss, no access or protection checks have been done. Because of this we must retranslate the original address through the TLB and perform all these checks. All protection and access traps will therefore occur from entries in the TLB.

The hardware walker for handling TLB misses, whilst it significantly improves the performance of handling first level page directories searches, is not felt to be sufficiently deterministic and so is not used in Stratus' fault tolerant HP-UX implementation. Also it gets complicated in a multiprocessor implementation. It has not been brought forward to the PA2.0 designs.

Left blank intentionally

Slide: Access Control

Access Control



a69653

Notes:

Slide: Access Control

The diagram in this slide shows the entire process for controlling access to a page through the TLB. Notice that there are two groups of checks that must be done -- the Protection Check and the Access Rights Check. If both checks pass (allow access) then access to the page is granted.

Access Rights Check

The **access rights** for a page indicate what type of access is allowed. The access rights from the TLB are compared to the type of access being requested to determine if the access can be granted.

The second part of the access rights check is to check the **privilege level** of the executing instruction. Recall from the previous discussion that the access rights field in the TLB records two privilege levels. The decision of which one to check is based on the access type.

If both the protection check and the access rights check pass, then access is granted to the page referenced by the TLB.

Protection Check

The first step in the Protection Check is to check the P-bit in the PSW, which is the **Protection ID Validation Enable** bit. If this bit is not set, we bypass all protection checking on the page and grant access (assuming the access rights check passes). If the protection ID validation is enabled, the **access ID** from the TLB entry is compared to the **protection IDs** in Control Registers CR-8, CR-9, CR-12, and CR-13.

The type of access is factored in because if we are requesting write access to the page, then the **Write Disable** bit in the Access ID must also be checked.

Protection IDs and Shared memory

Если процесс использует
много областей SHMEM,
то необходимо иметь
наглядно!
не хватает
размеров.

Using hardware registers to store the protection IDs has the advantage of speed but has the disadvantage of only having a limited capacity.

Since HP-UX uses PA's global addressing capability any process can attempt to access any global address, including ones that do not belong to it. The operating system therefore needs to protect against this sort of un-authorized access. This is where the protection IDs are used.

In this example a process needs to have access to both its own text and data areas, but might also need access to some of the shared memory areas. Since all these shared memory areas appear within its own normal virtual address space, each will have to have its access be individually controlled by a separate protection ID.

If our process needed to have access to 3 of the shared memory segments (plus its text and data) it would require access to 5 areas with their own protection IDs. On the 64bit processors running HP-UX 11.00 (or 10.20 with the

Slide: Access Control

appropriate patch) but on a 32it processor only 4 protection IDs can be store in the control registers. When the process accessed the fifth area then since the hardware can resolve the issue, then software must be used. Software is of course slower.

Running a program¹ that times access both 2 and 3 areas of memory, for both shared and ‘normal’ we can see this affect.

```
ken@kanga[tops] timex ./protid
3 sections of shared memory 2.272
2 sections of shared memory 0.245
3 sections of normal memory 0.694
2 sections of normal memory 0.243

real    3.49
user    3.45
sys     0.01

ken@kanga[tops]
```

Here a very substantial difference can be seen between the length of time taken to work on three areas of shared memory as opposed to normal memory, whereas for two areas there is no real difference.

Another interesting thing that can be seen from this example. Normally when the operating system code is being executed, commands like `timex` would report this as system time, but here that is not the case. Handling issue such as this is so performance critical that *wasting* time performing correct accounting is not done.

The kernel has two sets of routines used to handle protection id misses. There is a “normal” routine, that checks the full list of protection ids that a process is allowed to access, by reading the processes pregon list. As it does this it then builds up a cached list that can be used to handle subsequent misses. The fast version of the routine then attempts to use this list, if that doesn’t work then the normal routine is called. It is this fast version of the routine that does not perform the normal accounting checks.

The kernel has a variable `fast_resolvepid_on` that controls the building of the list, and therefore whether the fast routine can be used to handle this issue. Without it, the system time can be seen, but at the cost of greatly increase run times.

1. see the `protid.c` program in the labs.

Slide: Access Control

```
root@kanga[] adb -w /stand/vmunix /dev/kmem
fast_resolvepid_on/W0
fast_resolvepid_on:      1      =    0
root@kanga[]
```

Note: Modifying variables in a running kernel is not supported by Hewlett-Packard. Most values that can be changed in this way result in system failures.

Now rerunning the earlier test

```
ken@kanga[tops]
ken@kanga[tops] timex ./protid
3 sections of shared memory 64.096
2 sections of shared memory 0.243
3 sections of normal memory 0.704
2 sections of normal memory 0.243
```

```
real 1:05.32
user 12.45
sys 52.63
```

```
ken@kanga[tops]
```

Here apart from the overall slow down of the run timex is now able to report the system mode cpu time.

From this test program it can be seen that where a program uses a larger number of areas of shared memory (or shared memory mapped files¹) than there are available protection ID registers then there can be² performance issues.

1. For memory mapped files then a protection id of 0 can be requested, this allows public access, so that no protection ID checks are performed see the section on shared libraries below.
2. Whether there will be depends upon the access pattern, if firstly one area is then accessed, then later another and so on, little or no problem should be seen. If however like the example program all the areas are accessed in rapid succession then the problem can be apparent.

Slide: Access Control

Protection IDs and Shared Libraries

As with shared memory shared memory shared libraries are mapped into the globally accessible S1 and S2 quadrants. Here however the protection IDs are normally handled differently.

In HP-UX shared libraries are implemented as memory mapped files. The mmap system call is used to attach files to the address space of processes. If the flag MAP_SHLIB is set when the file is mapped and the kernel parameter public_shlibs is set and the file permissions include `-r-xr-xr-x`¹ then the pages are marked public, setting their protection ID to 0 so that it does not need to be checked against a protection register..

If this behavior is not considered acceptable (as for instance it could be considered to violate the directory permissions checking) then the kernel parameter can be used to disable it. This can however then lead to performance problems associated with the contention over the protection registers, as with shared memory.

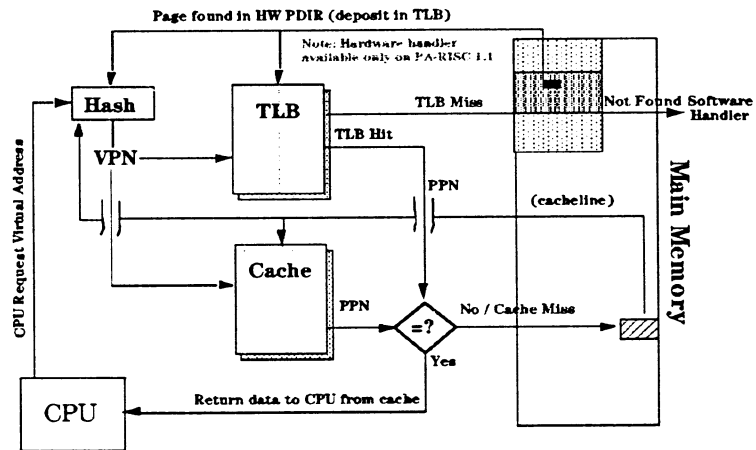
Алгоритм. Век SD нечет
G app. неоспариве.
PR. ID = 0

1. Older versions the permissions needed to be exactly `r-xr-xr-x`, This is documented in the online help screens for generating shared libraries.

Left blank intentionally

Slide: TLB/Cache Summary

TLB/Cache Summary



a69654

Notes:

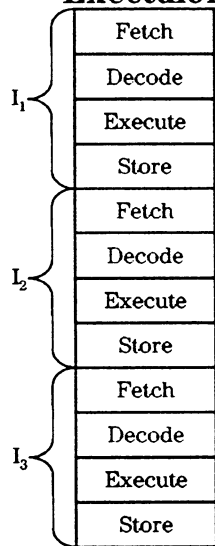
Slide: TLB/Cache Summary

This slide shows the entire process of satisfying a request for a virtual address by the CPU. All of the individual steps have been discussed in detail and are shown together here.

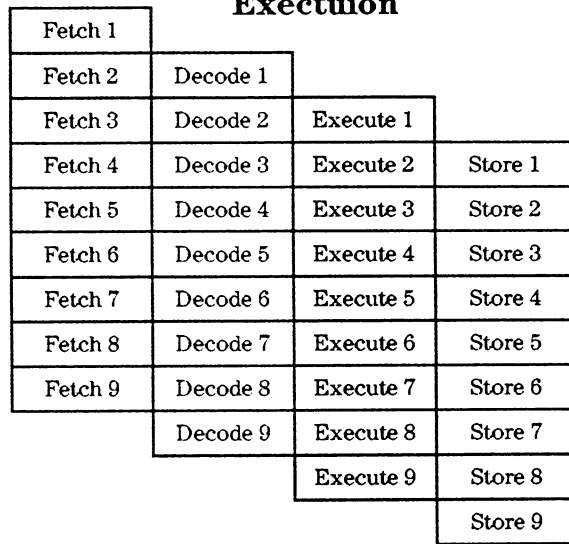
Slide: Instruction Pipelining

Instruction Pipelining

Non-Pipelined Execution



Pipelined Execution



a69655

Notes:

Slide: Instruction Pipelining

At this point we have looked at how PA-RISC addresses memory through virtual addresses and how these addresses are translated to physical addresses through the TLB and cache. Now we want to look at the remaining architecture issues of instruction pipelining and interrupt handling.

A major feature of the PA-RISC architecture that enhances performance is the presence of **instruction pipelining**.

Instruction execution is divided up into several stages. A simple example of this is to divide instruction execution into four stages: fetch, decode, execute and store. In a non-pipelined environment these stages would execute sequentially, one instruction at a time. The fetch of a new instruction could not begin until all four stages of the previous instruction were complete. If each stage took one clock cycle to complete, we would only be executing an instruction once every four clock cycles.

The logic behind dividing instruction execution into stages is that we assume the CPU is able to carry out one instance of each stage in parallel. So while it is fetching one instruction, it may be decoding or executing another. If we are able to keep the pipeline full, we now are able to complete execution of one instruction every clock cycle.

The number of stages in the pipeline, or the number of operations that can be performed in parallel, is referred to as the **depth of the pipeline**. In the example discussed above the depth of the pipeline is four. The longest period of time it takes to complete any one stage is the **width of the pipeline**. In our simple example, the width is 1 clock cycle.

Three features of PA-RISC make it possible to pipeline instructions.

- The first is **multistage instructions**, of which we have already shown an example. The number of stages is dependent on the processor.
- **Prefetch of instructions** into the instruction cache also assists in pipelining. The architecture allows for prefetch of up to seven instructions into the instruction cache along with the current instruction.
- **Instruction Address queues** also enhance the pipelining by allowing the next instruction to be brought into an execution register while the current instruction is executing.

The next slide illustrates superscalar pipelining employed by many newer processors.

Problems with pipelining

Whilst pipelining the execution of instructions can increase performance, by allowing one instruction to be completed every clock cycle rather than every four. It does introduce some problems.

1. Branches, where one instruction tells the processor to go to a new part of the program, what is to be done about the remaining instructions in the pipeline.
2. Register interlock, successful pipelining presumes that each of the stages can be done in parallel since they do not need to share hardware. Where one instruction needs to access some data from a register and the preceding instruction is still storing the result away there is contention over the CPU register.

Slide: Instruction Pipelining

For branches the problem is that the following instruction has already been decode and is ready to execute, but the current instruction is saying do something else. Well under these circumstances there are two possible courses of action. The next instruction can be discarded, and the time can be wasted. Alternatively the next instruction can be executed.

This section option seems to go against the wishes of the programmer, the CPU is executing an instruction after a branch. If the programmer knows that this will take place though it can be taken into account.

Consider the following example of pseudo code (PA assembly does not look anything like this).

high level	assembly language
a=3	LDI 3 => GR5
b=4	LDI 4 => GR6
c=a+b	ADD GR5,GR6 => GR7
goto 10	BR \$10
	NOP

When the branch instruction is executed, the value of 'c' has not been read, so it is not yet need, in fact the line 'c=a+b' does not need to be executed until either the registers GR5 or GR6 are needed for something else or the 'c' is used. If the add instruction were to be positioned after the branch (in what is known as the delay slot) it could being worked out whilst the CPU fetched the new instruction at \$10.

Also in this example 4 stage pipeline were storing away the result of an instruction happen on the clock cycle after the execute, moving this addition would have a further benefit. The second instruction would still be trying to write the value 4 into GR6 at the same time as the add instruction were trying to read the value out age. This can not happen, and so the CPU would stall for one cycle to allow the store to complete before starting the execution.

Moving the add down to the position after the branch not only allows the delay branch slot to be filled it would also give the load instruction time to complete. Moving this one instruction could therefore save 2 cycles as the above example would take 6 CPU cycles and a reorder example: -

Slide: Instruction Pipelining

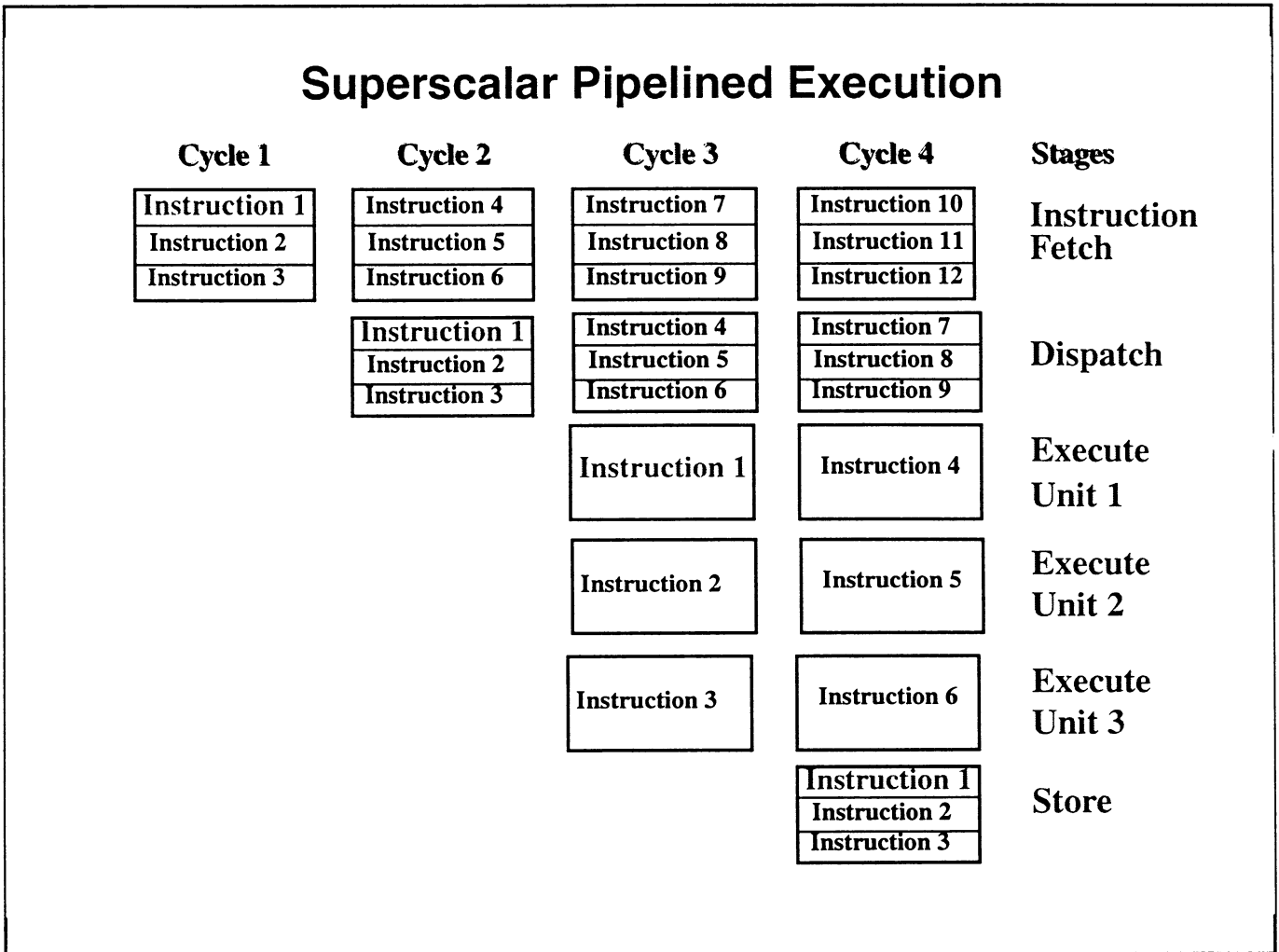
high level	assembly language
a=3	LDI 3 => GR5
b=4	LDI 4 => GR6
c=a+b	BR \$10
goto 10	ADD GR5,GR6 => GR7

would only take 4.

This sort of reorganisation is typically carried out by the optimising compilers.

Pipelining works because different circuitry is used to process the different stages of an instruction. Different types of instructions can also use different circuits, so it could be possible to use these in parallel as well. Many modern CPU do this and it is known as SuperScalar execution and will be discussed next.

Slide: Superscalar Pipelined Execution



Notes: В ааеае 4 онеаруаа 3а тааа.
В срааааа 2,4

Slide: Superscalar Pipelined Execution

Machines that issue multiple independent instructions per clock cycle have been called superscalar machines. In a superscalar machine, the hardware can issue a small number (say 2 to 4) of independent instructions in a single clock cycle.

The slide shows a generalized and very simplified superscalar pipeline.

Note that if the instructions in the instruction stream are dependent or do not meet certain criteria, only the first instruction in the sequence is issued.

Slide: Interrupts During Instruction Execution

Interrupts and Exceptions

- Interrupts are events external to the execution of the current instruction.
- Exceptions are events resulting from current instruction

Classes of Interruptions

- Fault
- Trap
- Interrupt
- Check

Notes:

Slide: Interrupts During Instruction Execution

The architecture defines the flow of execution to be sequential from one instruction to the next. There are three conditions that can affect this flow of control

- **Branch Instructions** whose effect will be discussed in Module 3
- **Nullification of Instructions** which is an instruction that is skipped over. These usually exist in conjunction with a branch instruction and will also be discussed in Module 3.
- **Interrupts and Exceptions**

Here we want to discuss interruptions during instruction execution that alter the flow of control. When these interrupts occur, the state of the processor must be saved so that the flow may resume as normal once the interrupt is handled.

There are four different classes of interrupts that may affect the flow of execution

Fault — *memory garbage*

Faults are normally non-fatal interruptions that signal a condition where the current operation cannot be completed until further action is taken by hardware or software. An example of this is a page fault. Instructions that generate faults re-execute once the fault condition is addressed.

Trap — *kernel*

There are several types of traps, some of which signal an error type of condition and others that are simply used to signal attention desired from the hardware.

Arithmetic traps such as underflow, overflow, and divide by zero are examples of traps that indicate an error condition. Instructions that generate these traps are usually not re-executed although it would be possible for a software handler to address the situation and allow execution to continue.

The second type of trap might be debugging support in which a trap generates a particular action when defined conditions occur. Recall the PSW has several trap bits such as the Taken Branch Trap Enable bit which is an example of this type of trap.

Interrupt

An interrupt occurs when an external member requires attention. An I/O device or power supply are examples of entities that may send interrupts to the CPU. An I/O device would interrupt to signal requests to send/receive data and a power supply would interrupt for power failure.

Check

These are conditions detected as malfunctions in hardware and are often fatal to the flow of execution.

All types of interrupts are processed in the same way but the action taken for each interrupt depends on the type and priority of the interrupt.

Slide: Interrupt Groups

Group 1	Highest Priority Immediate Processing
Group 2	Interrupts
Group 3	Signal During Execution Complete Instruction After Handling Interrupt
Group 4	Signal After Execution

Notes:

Slide: Interrupt Groups

Interrupts are categorized into four groups based on their priority and severity. The groups and types of interrupts within each are listed below.

Each interrupt is assigned an interrupt number which serves as an index into a table that indicates which interrupt service routine to execute. The group numbers determine when an interrupt will be serviced in regards to the current instruction execution. Within each group the individual interrupts are serviced in the order listed in the tables below. Refer to the PSW diagram on page 2-31 for details on the bits that affect each of these interrupts.

Group 1

1	<p>High-Priority Machine Check A hardware error has been detected that must be handled before processing can continue</p>
----------	---

Group 2

2	<p>Power Failure Interrupt The machine is about to lose power.</p>
3	<p>Recovery Counter Trap Bit 0 of the recovery counter is 1 and the PSW R-bit is 1.</p>
4	<p>External Interrupt A module writes to the processor's IO_EIR or to the broadcast IO_EIR register, or the interval timer compares equal to its associated comparator register.</p>
5	<p>Low-Priority Machine Check <i>— ECC ERROR</i> A hardware error has been detected which is recoverable and does not require immediate handling.</p>
29	<p>Performance Monitor Interrupt An implementation-dependent event related to the performance monitor coprocessor requires software intervention.</p>

Group 3

6	<p>Instruction TLB Miss Fault / Instruction Page Fault The instruction TLB entry needed by instruction fetch is absent, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table.</p>
----------	--

Slide: Interrupt Groups

7	<p>Instruction Memory Protection Trap Instruction address translation is enabled and the access rights check fails for an instruction fetch or instruction address translation is enabled, the PSW P-bit is 1, and the protection identifier check fails for an instruction fetch.</p>
8	<p>Illegal Instruction Trap An attempt is being made to execute an illegal instruction or to execute a GATEWAY instruction with the PSW B-bit equal to 1.</p>
9	<p>BREAK Instruction Trap An attempt is made to execute a BREAK instruction.</p>
10	<p>Privileged Operation Trap An attempt is being made to execute a privileged instruction without being at the highest privileged level (0).</p>
11	<p>Privileged Register Trap An attempt is being made to write to a privileged space register or access a privileged control register without being at the highest privileged level (0).</p>
12	<p>Overflow Trap A signed overflow is detected in an instruction which traps on overflow.</p>
13	<p>Conditional Trap The condition succeeds in an instruction which traps on condition.</p>
14	<p>Assist Exception Trap A coprocessor or special function unit has detected an exceptional condition or operation. An exceptional operation may include unimplemented operations or operands.</p>
15	<p>Data TLB Miss Fault / Data Page Fault Data TLB entry needed by operand access of a load, store, or semaphore instruction is absent, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table.</p>
16	<p>Non-Access Instruction TLB miss fault The instruction TLB entry needed for the target of a FLUSH INSTRUCTION CACHE instruction is absent, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table.</p>

Slide: Interrupt Groups

17	Non-Access Data TLB Miss Fault / Non-Access Data Page Fault The data TLB entry needed by an instruction is not present, and if TLB misses are handled by hardware, the hardware miss handler could not find the translation in the Page Table.
26	Data Memory Access Rights Trap Data address translation is enabled, and an access rights check fails on an operand reference for a load, store or semaphore instruction, or a cache purge operation.
27	Data Memory Protection ID Trap Data address translation is enabled, the PSW P-bit is 1, and a protection identifier check fails on an operand reference for load, store and semaphore instructions, and cache purge operations.
28	Unaligned Data Reference Trap Data address translation is enabled, and a load or store instruction is attempted to an unaligned address (unaligned absolute access is undefined)
18	Data Memory Protection Trap / Unaligned Data Reference Trap Data address translation is enabled, and an access rights check or a protection identifier check fails on an operand reference for a load, store, or semaphore instruction, or a cache purge operation; a load or store instruction is attempted to an unaligned address with virtual address translation enabled.
19	Data Memory Break Trap Load, store instruction or cache purge operations to a page with the TLB B-bit set to 1 in the data TLB entry.
20	TLB Dirty Bit Trap Load, store instruction access to a page with the D-bit set to 0 in the data TLB entry.
21	Page Reference Trap Load, Store instruction to a page with the T-bit set to 1 in its data TLB entry.
22	Assist Emulation Trap An attempt is being made to execute an assist instruction for an unimplemented assist processor or to execute a coprocessor instruction for a coprocessor whose corresponding bit in the Coprocessor Configuration Register (CR10) is 0.

*B CISC
MONTRO,
39PC6
HET.*

Group 4

Slide: Interrupt Groups

23	Higher-Privilege Transfer Trap An instruction is about to be executed at a higher privilege level than the instruction just completed and the PSW H-bit is 1.
24	Lower-Privilege Transfer Trap An instruction is about to be executed at a lower privilege level than the instruction just completed and the PSW L-bit is 1.
25	Taken Branch Trap An unconditional branch or a taken conditional branch was executed, and the PSW T-bit is 1.

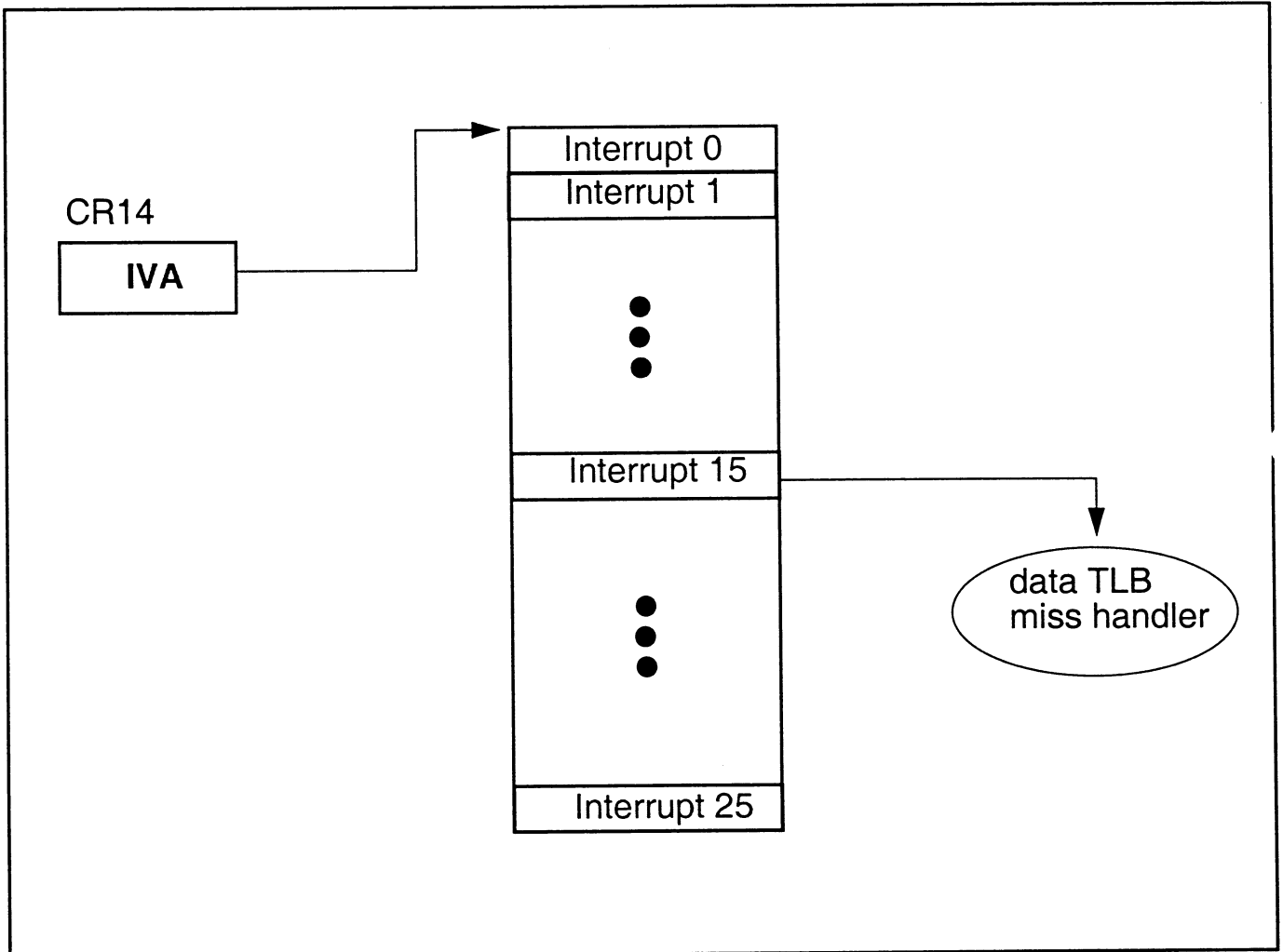
Further details of each interrupt type is documented in Chapter 5 of the *PA-RISC 2.0 Architecture* document. Included is information about the contents of the interrupt registers and notes related to the conditions that cause the trap. This is useful in debugging since the trap type reported by HP-UX is normally the interrupt type defined by the hardware.

For example, on a Data Memory Protection Trap, HP-UX reports trap type 18 and the contents of the interrupt registers are

- ISR** Space Identifier of the virtual address
- IOR** Offset of the virtual address
- IIR** The instruction causing the trap

Left blank intentionally

Slide: Interrupt Vector Table



Notes:

Slide: Interrupt Vector Table

Т.е. ке адрес, а обработка
смысла го 32 бита, в т.ч.
и с перекрестом

In order to service an interrupt, the hardware must be able to locate an interrupt service routine. The **Interrupt Vector Table (IVT)** contains these addresses. The base address of the IVT, the **Interrupt Vector Address (IVA)** is kept in Control Register 14 (CR14). The IVT contains one entry for each type of interrupt and is indexed by interrupt number. Each entry is 32-bytes (or 8 words) in length, and contains the actual instructions. Using the address in CR14 and the interrupt number, the following calculation can then be used to locate IVT entry of a particular service routine

$$\text{IVA} + (0x20 * \text{Interrupt_Number})$$

The IVA can also be found in the mpinfo structure (to be discuss later).

The following is an example of looking at Interrupts 17 (*Non-access Data TLB Miss Fault / Non-Access Data Page Fault*) and 15 (*Data TLB Miss Fault / Data Page Fault*) using adb:

```
#adb -k /stand/vmunix /dev/mem
```

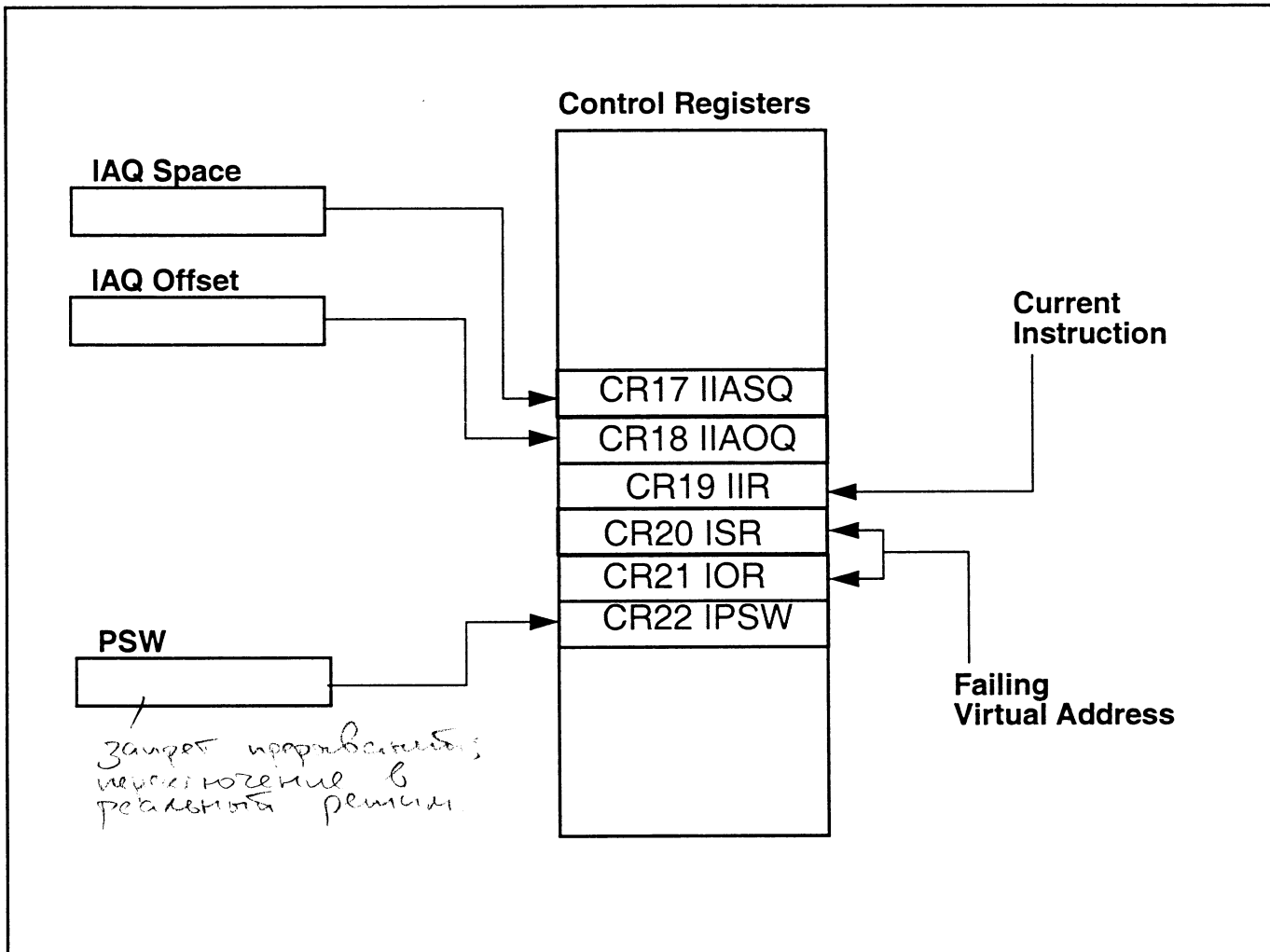
```
0x2b000+(0x20*0d17),8?ia
```

```
$i_nadtlb_miss_2_0:
$i_nadtlb_miss_2_0:           MFCTL           cr21,r9
$i_nadtlb_miss_2_0+4:       EXTRDU         r9,51,20,r1
$i_nadtlb_miss_2_0+8:       MFCTL           cr20,r8
$i_nadtlb_miss_2_0+0xC:     COPY           r8,r24
$i_nadtlb_miss_2_0+10:      EXTRDU         r8,53,54,r17
$i_nadtlb_miss_2_0+14:      XOR            r1,r17,r17
dnatlb_bl_patch_2_0:
dnatlb_bl_patch_2_0:         B              dnatlbmss_PCXU
dnatlb_bl_patch_2_0+4:      EXTRDU         r8,31,32,r25
```

```
0x2b000+(0x20*0d15),8?ia
```

```
$i_dtlb_miss_2_0:
$i_dtlb_miss_2_0:           MFCTL           cr21,r9
$i_dtlb_miss_2_0+4:       EXTRDU         r9,51,20,r1
$i_dtlb_miss_2_0+8:       MFCTL           cr20,r8
$i_dtlb_miss_2_0+0xC:     COPY           r8,r24
$i_dtlb_miss_2_0+10:      EXTRDU         r8,53,54,r17
$i_dtlb_miss_2_0+14:      XOR            r1,r17,r17
dtlb_bl_patch_2_0:
dtlb_bl_patch_2_0:         B              dtlbmss_PCXU
dtlb_bl_patch_2_0+4:      EXTRDU         r8,31,32,r25
```

Slide: Interrupt State Save



Notes:

Slide: Interrupt State Save

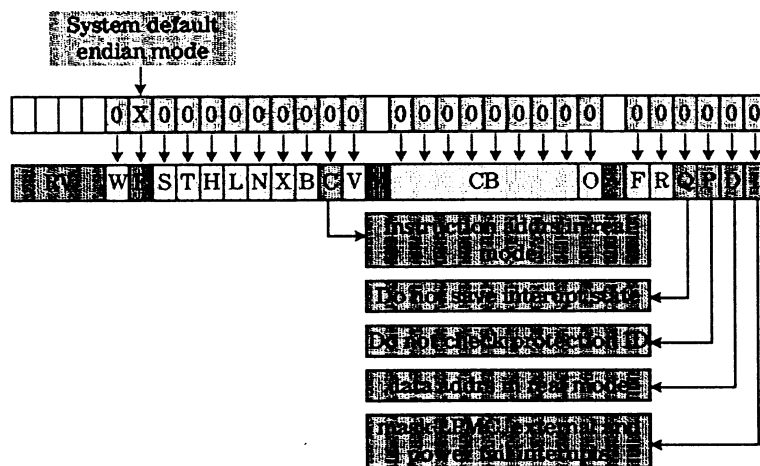
The process described in the prior slide is the software view of the interrupt handling. From an architecture standpoint we are interested in what the hardware does to handle the interrupt. The hardware implements the interrupt as a context switch that takes the following actions:

1. The PSW in effect at the time of the interrupt is saved in the IPSW (CR22). For group 2 and 3 interrupts, the IPSW will contain the value at the beginning of execution. For group 4 the value will reflect the value after execution is complete.¹
2. Defined bits in the PSW are set to 0 to disable lower priority interrupts.
3. The instruction queue is saved in IASQ and IAOQ registers (CR17 and CR18, respectively) to allow execution to resume at the point of the interrupt.
4. The privilege level is set to the highest possible level (zero).
5. Since the instruction causing the interrupt may be useful, information about the instruction executing at the time of the interrupt is saved in IIR (CR19).
6. General Registers 1, 8, 9, 16, 17, 24, and 25 are copied to the shadow registers.
7. Execution begins at the address obtained from the IVA.

1. This information about when the PSW is saved is important for dump readers. In failures that are trap related it is important to know which instruction caused the trap.

Slide: The Processor Status Word and Interrupt Handling

Processor Status Word and Interrupt Handling with interrupts



a60661

Notes:

Slide: The Processor Status Word and Interrupt Handling

When the PSW is clear as part of the handling the interrupt a number of the bits are of particular significance for interrupt handling.

Bits	meaning	effect
C	Code address translation	Clearing these two bits has the effect of putting the processor into “real” mode, using physical addresses directly.
D	Data Address translation	
Q	Interrupt State save	Since there are only 1 set of control and shadow registers to hold the pre-interrupt data, it is necessary to control what happens if a second interrupt occurs. If these registers get over-written then it might not be possible to return to the original piece of code.
P	protection ID checking.	not that this happens when in real mode.
I	external interrupt disable	Block the delivery of the interrupts from Powerfails, LPMCs and external interrupts.

Since there are only 1 set of control and shadow registers to hold the pre-interrupt data, it is necessary to control what happens if a second interrupt occurs. If these registers get overwritten then it might not be possible to return to the original piece of code.

However without saving this information handling the interrupt might not be possible.

Therefore during the early stages of handling an interrupt it is important not to get a second one. Clearing the I bit disables all the asynchronous interrupts except HPMCs, which are special case, since they are always fatal it does not matter whether it is possible to come back from them.

Clearing C,D and P and going into real mode avoid generating all the TLB related interrupts.

It is then the programmers responsibility to make sure that no other trap/fault occur whilst in the early stages of the interrupt service routine.

If it is desirable to re-enable interrupts then the first level interrupt service routine needs to save the interrupt registers to a stack, it can then re-enable the state saving via the Q bit, and allow external

Slide: The Processor Status Word and Interrupt Handling

interrupts, via the I bit. However the causes of external interrupts might need to be prioritized, and this can be achieved through the EIEM (CR15), this allows individual interrupt bits to be blocked, for later processing. This programming of the EIEM is normally achieved using some pre-defined configurations known as the SPL (system priority levels).

Left blank intentionally

Slide: Architecture Summary

Architecture Summary

- Memory and I/O access through virtual addresses.
- Register based CPU operations
- Virtual address translation through TLB and Page Directory
- Memory access through cache
- Interrupts allow exceptions to normal instruction flow.

Notes:

Slide: Architecture Summary

In summary, some of the things we have seen in this module are

- Most access to memory or I/O on a PA-RISC system is done through virtual addresses which are mapped to physical address.
- The CPU on a PA-RISC system makes heavy use of a large register context. There are sets of general registers, control registers, floating point registers, an instruction queue, and processor status word.
- The TLB is used to translate virtual address to physical addresses. The TLB holds the most recently accessed translations and the Page Directory holds all translations for main memory.
- All data and instructions are accessed through cache.
- The normal instruction flow on a PA-RISC system is sequential; interrupts allow for changes to this normal flow.

Module 2 — System Architecture

Lab: Exploring Cache & TLB Behavior

Using the programs found in the directory `/home/tops/sysarch` examine the behavior of the cache and the tlb on the system.

Firstly run the `./config` script. This will tune the test programs for your system and compile them.

1) The program `cache`, times memory accesses where a pair of pointers are setup at roughly the size of the cache apart. The program prints out a series of times at different separations. Run the program and explain the reason for the different times as the separation changes.

If your system uses a PA7200 processor the assist cache will mean that you do not see any interesting effects other than the fact it doesn't slow down like other processors.

If your system uses a PA7300LC or PA8500 then try using `cache2` instead.

2) The program `tlb` accesses a range of pages, using this program try and find the size of the tlb fitted to the lab system

If your system uses a PA1.1 processor the effect may be very small, and you might not see it. If so try running it a few times.

pages, create HA 113

3) Using the `protid2` program see how the system performance changes as the number of shared memory segments used is changed.

Try changing the value of `fast_resolvpid_on` in the live kernel to 0 and seeing the effect again.

DO NOT FORGET TO CHANGE IT BACK AFTERWARDS!

4) `Protid3` & `protid4` are the same program as `protid2` but compiled with different options, what is difference observed in their behavior as opposed to `protid2`.

Depending on the patches on your system these may run identically to `protid2`.

ген 8-мисе комментов сам мигрени

5) Using the program `useShlibs` see what difference the file permissions on the shared libraries in the directory make.

Firstly run `./useShlibs pause &`. Then run a second copy and time it `timex ./useShlibs`.

Then kill your first copy of `useShlibs`.

Now change the file permissions on the shared libraries to 500 and repeat the test.

Время выполнения паков стало меньше

Lab: Exploring Cache & TLB Behavior

6) Using `adb` try to change the first instruction in the routine `rtprio` in the running kernel in memory. The memory can be accessed through the device files `/dev/mem` and `/dev/kmem`.

The `kmem` device file gives access to the kernel's virtual memory.

The `mem` device file gives access to physical memory. `Adb` can perform virtual to physical address translation by using the `-k` option.

By the way `adb`'s error messages are not always very helpful.

```
adb -w /stand/vmunix /dev/kmem
```

```
rtprio/i
```

```
rtprio/X
```

```
rtprio/W0
```

go into `adb` allowing writes

print the first instruction in `rtprio`

now print it in Hex, you will need this to set it back later.

write a zero into the address, does this work?

Now try the same test using

```
adb -k -w /stand/vmunix /dev/mem
```

```
rtprio/i
```

```
rtprio/X
```

```
rtprio/W0
```

print the first instruction in `rtprio`

now print it in Hex, you will need this to set it back later.

write a zero into the address, does this work?

Why is there a difference.

Slide: Architecture Summary

Module 2 — System Architecture

System Specifications

The following tables show a comparison of several current systems in terms of their TLB and cache configurations.

	Max Processors	Mhz	Chip Arch	TLB	Cache
Gecko (712) PCX-L	1	100	PA7x00 PA 1.1	Unified 64+8 blk HW Walker	
Raven T' (C100, C110) PCX-T'	1	100 120	PA7200 PA 1.1	Unified 120+16blk HW Walker	256K Instr 256K Data
Raven U (C160, C180) PCX-U	1	160 180	PA8000 PA 2.0	Unified 96+No blk No Walker	512K x 512K 1024Kx1024K
Raven U+ (C200, C240) PCX-U		200 236	PA8000 PA 2.0		
Raven L2 (C160L) PCX-L2	1 2	132 165	PA7300 PA 1.1		64K x 64k 1MB Sec cache
Merlin L2 (B132L, B160L) PCX-L2	1	132 165	PA7300 PA1.1		1MB Sec cache
Skyhawk (J200, J210, J210XC) PCX-T'	2	100 120	PA7200 PA 1.1	Unified 120+16 blk HW Walker	256K Instr 256K Data XC: 1MB x MB
SkyHawk/U (J280, J282) PCX-U	2	180	PA8000 PA 2.0	Unified 96+No blk No Walker	1MB x 1MB

Module 2 — System Architecture

	Max Processors	Mhz	CPU Arch	TLB	Cache
Ultralight (Dx50, Dx60) PCX-T'	2	100 120	PA7200 PA 1.1	Unified 120+16 blk HW Walker	256K Instr 256K Data
Ultralight (D200, Dx10) PCX-L	1	75 100	PA7100 PA 1.1	Unified 64+8 blk HW Walker	256K Unified
Ultralight (Dx20, Dx30) PCX-L2	1	132 160	PA7300 PA 1.1		64K Instr 64K Data
Ultralight (Dx70, Dx80) PCX-U	2	160 180	PA8000 PA 2.0	Unified 96+ No blk No Walker	512K/512K 1M/1M
Ultralight (D390) PCX-U+	2	240	PA8200 PA 2.0		2MB/2MB
KittyHawk (Kx00, Kx20) PCX-T'	4	100 120	PA7200 PA 1.1	Unified 120+16blk HW Walker	256K Instr 256K Data
Mohawk (K260, K460) PCX-U	4	180	PA8000 PA 2.0	Unified 96+No blk No Walker	1Mb Instr 1Mb Data
Bravehawk (K370, K570) PCX-U+	6	200	PA8200 PA 2.0		2MB/2MB
Dragonhawk (K380, K580) PCX-U+	6	240	PA8200 PA 2.0		2MB/2MB

	Max Processors	Mhz	CPU Arch	TLB	Cache
TNT (T500) PCX-T	12	90	PA7x00 PA 1.1		

Module 2 — System Architecture

Slide: Architecture Summary

	Max Processors	Mhz	CPU Arch	TLB	Cache
Nitro (T520) PCX-T'	14	120	PA7150 PA 1.1	Unified 120+16blk HW Walker	1Mb Instr 1Mb Data
Jade (T600) PCX-U	12	180	PA8000 PA 2.0	Unified 96+No blk No Walker	1Mb Instr 1Mb Data
Lancelot V2200 PCX-U+	16	200	PA8200 PA 2.0		2MB/2MB
Exemplar V2250 PCX-U+	16	240	PA8200 PA 2.0		2MB/2MB

Module 2 — System Architecture

Left blank intentionally

Module 3

Kernel Services

“This new learning amazes me, Sir Bedevere. Explain again how sheep’s bladders may be employed to prevent earthquakes.”

King Arthur,
Monty Python and the Holy Grail

Objectives:

- Understand how user code interfaces with the kernel through the system call interface.
- Understand how kernel events are scheduled through the callout table.
- Understand the allocation of kernel memory in kernel buckets.

Slide: Kernel Services Overview

Kernel Services Overview

- The system call interface
- Callout table
- Kernel Buckets — *managing kernel memory*

a69663

Notes:

Slide: Kernel Services Overview

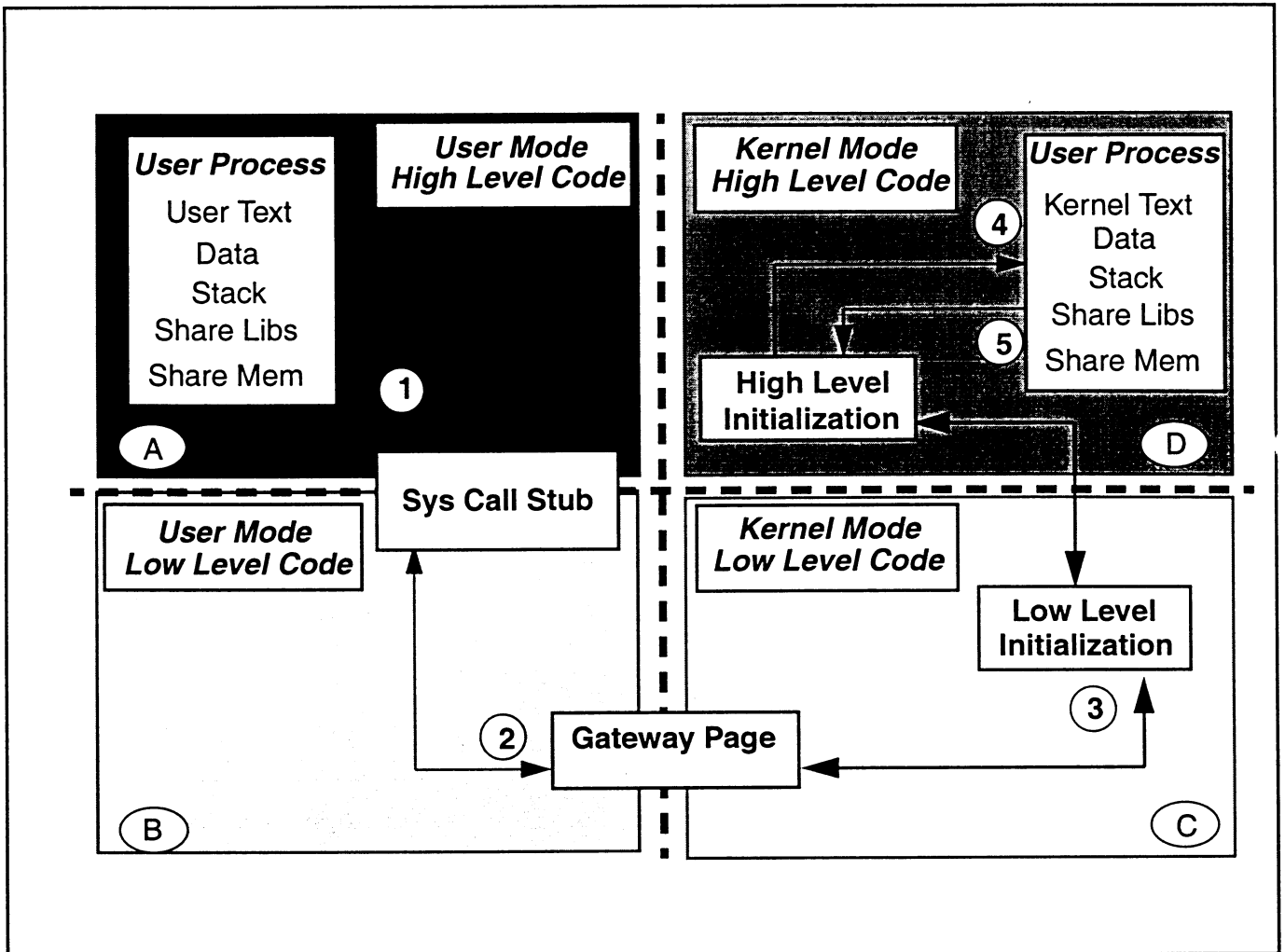
In the last two modules we have dealt with general issues of PA-RISC that are not specifically related to the HP-UX operating system. With this module we will begin looking at the specifics of HP-UX.

As discussed in the overview in Module 1, this course will focus primarily on several major subsystems. There are other specifics of HP-UX that are interesting to kernel developers and debuggers that do not relate to one of these specific subsystems. We want to deal with some of these topics early as they may relate to later discussions throughout.

In this module we will briefly cover three primary topics:

- Kernel entry through the **System Call Interface**
- Scheduling kernel events through the **Callout Table**
- Managing kernel memory with **Kernel Buckets**

Slide: The System Call Interface



Notes:

Slide: The System Call Interface

System Call Introduction

The System Call Interface is a collection of library functions that provide kernel facilities to user programs. These facilities include file operations, signal handling, and process operations.

When invoking a system call, a process running in user mode will change its execution mode to kernel. A system call number is passed that should match an entry in the system call table, called **sysent []**. The file `/usr/include/sys/scall_define.h` defines symbolic names for all the system calls; it is created automatically at kernel build-time from a master description of system calls. The operation is performed by the kernel on behalf of the user process and the results of the operation are returned.

System Call Flow

System calls are the mechanism which allow a user's program to use services provided by the operating system. The standard specification of HP-UX requires that system calls appear to be simple procedure calls to and from high-level language (C, etc.) code. In reality, the user's high-level language code makes calls to library routines which satisfy this requirement, and the library routines direct the flow of control into the operating system.

The System Call Interface consists of four key areas.

User Mode - High Level

User Mode - Low Level

Kernel Mode - Low Level

Kernel Mode - High Level

Slide: The System Call Interface

High Level Overview of the System Call Interface

Using the slide on the previous page, we will walk through the flow of the system call interface.

1. *User Mode - High Level*

When a user program performs a system call, execution passes into an assembly language stub located in the C library. The assembly language stubs generally define the system call entry point, and they include the branch to the system gateway pages with the system call number as an argument.

2. *User Mode - Low Level*

The standard gateway pages elevate the privilege level and branches into the kernel system call initialization routine.

3. *Kernel Mode - Low Level*

The low-level initialization code is responsible for initializing the kernel stack environment, saving the relevant user state, and setting up the appropriate kernel state.

4. *Kernel Mode - High Level*

Control then goes to the high-level system call initialization code, where the kernel routine for the specified system call is finally called.

5. *Kernel Mode - High Level ---> Kernel Mode - Low Level--> User Mode --> Low Level --> User Mode - High Level*

The return path proceeds back through the high and low level system call return code, and the library stub code is re-entered via a branch external instruction which sets the privilege level back to its original privilege level. The stub completes the call by branching back to the user's code.

64 bit System Call Interface

The System Call Interface has not changed with 11.0 when running in 32-bit mode. However, there are changes to the System Call Interface when running in 64-bit mode. We can have a 32-bit program or a 64-bit program running on a 64-bit kernel. The calling convention is extended, there is a new 64-bit gateway page, and an argument coercion stub at step (4) above has been added. These will be discussed later on.

Slide: The System Call Interface

Stub Entry

High-level user code invokes a system call via an ordinary function call, for example, by calling *write(2)*, a function in the C library. That function is called a “stub.” Stubs, usually generated from macro definitions processed by the preprocessor to the C compiler, define the entry point for each system call.

The stub itself is quite simple, and really only consists of a macro call to generate the appropriate call and return code for the specified system call. It expects the standard calling convention: arguments to the system call are passed in the normal argument registers. The stub and kernel agree on some additions to the calling convention: the system call number is passed in via the caller save register r22, #defined as **SYS_CN**,¹ and r31 is architected to be the stub’s BLE return pointer. The stub then branches to the kernel’s gateway page.

Gateway Page

For 32-bit applications², the gateway page, called **gateway_page**, is at a fixed, well-known address: 0xC0000004. Thus, the 32-bit stub simply branches to the hard code address 0xc0000004 and passes the system call number in register GR22 in the delay branch slot.

For 64-bit applications, there is a system vector page (**sysvec**), which contains pointers to a separate gateway page called **gateway64_page**. Both pages are mapped immediately after the 32-bit gateway page, and a pointer to the **sysvec** page is passed to the application at exec time, and saved by crt0/dld in a program global: **__scall_entry_table_addr**. Therefore, the 64-bit stub is slightly more complicated.

-
1. Per the 2.0 calling convention, arg4 is passed in r22. To preserve **SYS_CN** semantics and still pass in arg4, the 64-bit stub copies r22 into **SYS_ARG4** (r28) before setting r22 to the system call number. The kernel low-level code knows this, and restores arg4 from **SYS_ARG4**.
 2. “Applications,” not kernel. 32-bit applications run unchanged on a 64-bit kernel, so there is still a 32-bit gateway page on the 64-bit kernel.

Slide: The System Call Interface

Gateway Instruction -- Promotion into Kernel Mode

The 32- and 64-bit gateway pages are kernel pages set up at boot time via *pdapage()* with special access rights: **PDE_AR_GATE**. These special access rights, in conjunction with the gateway instruction, sets the privilege level to 0, thereby promoting the user thread into kernel mode.

The gateway instruction is a type of branch with the side effect of changing the current privilege level. Per the architecture, “If the **,GATE** completer is specified and the PSW-C bit is 1, the privilege level is changed to that given by the two rightmost bits of the type field in the TLB entry for the page from which the branch instruction is fetched if that results in a higher privilege.”¹ In other words, when the kernel sets the access rights on the gateway pages to **PDE_AR_GATE** (0x4C), with the two rightmost bits being 0, executing a gateway instruction on those pages will promote the user into privilege level 0.

Note that both the gateway instruction and the necessary page permissions are required. Since only the kernel can set TLB access rights, this does not present a security issue.

The initialization sequence for most system calls is essentially identical. The immediate effect of executing the gateway instruction with a target of `‘.+8’` is to branch two instructions forward, where an interspace branch (**BE**) sends the flow of control to the first quadrant of space zero where kernel text is executed with the appropriate privilege level.

The gateway page for 32-bit applications does a simple promotion to privilege level 0 using a gate instruction and branches into the system call handler, **syscallinit**.

The 64-bit gateway page is similar, except that it has no guarantee that `sr7` is 0, so register **SR2** is used to explicitly reference the kernel space.

1. Kane, *PA-RISC 2.0 Architecture*, p. 7-10. See the Bibliography.

Slide: The System Call Interface

Lightweight System Calls

A certain class of system calls do not require any of the overhead of “normal” system calls. They simply set or get fields from the user area or proc table; they do not block, sleep or have any multiprocessor issues. For performance reasons, such “lightweight” system calls vector off at the gateway page to simple assembly; as a result, system call counts and metrics are not collected. The list of lightweight system calls includes: *sigblock()*, *sigvector()*, *sigsetmask()*, *getpid()*, *getppid()*, *getuid()*, *geteuid()*, *getgid()*, *getegid()* and *umask()*.

The 32-bit gateway page actually looks like a jump table of 4-instruction blocks, indexed by the system call number, with most blocks branching to *syscallinit*. It starts like this:

```
root@kgcc[.root] ied -h /.root/.adb_hist adb -k /stand/vmunix /dev/mem
```

```
0xc0000004,10/ia
0xC0000004:      SUBI,>>=      253,r22,r0
0xC0000008:      B,N           ffffffff00000018
0xC000000C:      ZDEP         r22,30,30,r1
0xC0000010:      BLR,N        r1,r0
0xC0000014:      NOP

0xC0000018:      GATE         ffffffff00000020,r0
0xC000001C:      LDIL         L%0x45800,r1
0xC0000020:      BE           1672(sr7,r1)
0xC0000024:      DEPI         3,31,2,r31

0xC0000028:      GATE         ffffffff00000030,r0
0xC000002C:      LDIL         L%0x45800,r1
0xC0000030:      BE           1672(sr7,r1)
0xC0000034:      DEPI         3,31,2,r31

0xC0000038:      GATE         ffffffff00000040,r0
0xC000003C:      LDIL         L%0x45800,r1
0xC0000040:      BE           1672(sr7,r1)
0xC0000044:
l
```

The 64-bit system vector page allows all the “heavyweight” system calls to share a single 4-instruction block, leaving room to create new lightweight system calls in the future.

1. Since the gateway page is 4096 bytes, that’s 1024 instructions. Take away the jump table prologue and divide by four, that leaves the valid lightweight system call range as 0-253 (*LW_MAXNUM*).

Slide: The System Call Interface

`syscallinit()` - `machine/asm_scalls`

The assembly level initialization code continues in the first quadrant of system space by executing from the `syscallinit` label. The goal at this point is to get a correct stack frame set up to get out of assembly code and into C. The key concept in setting up the stack is that with the exception of registers dedicated to the calling convention, only registers which have already been preserved will be modified. Hence, upon entering `syscallinit`, only the caller save registers may be used. Since this initialization routine does not make use of any of the entry save partition registers, it does not save any of them. High-level code which uses any of these entry save registers will be responsible for saving them.

The first task is to find the pointer to the user area. Its space id is kept in the per-processor data structure (ppdp) in the field `uareasid`; the virtual address is a constant `UAREA` in the 32-bit kernel, and in the ppdp's `saved_uptr` in the 64-bit kernel. Long pointer operations must be used to access `uptr`; short pointer displacement would not be successful since `uptr` is in the second quadrant, which is tracked by `sr5`, and `sr5` has not yet been saved. Thus, `sr2` and `gr1` are used.

Once the `uptr` is found, long pointer operations save the system call arguments passed through registers into the array `u.u_arg[]`; in the 64-bit kernel, it's assumed that the caller is following the 64-bit calling convention, at least until the high level code is reached. The base of the kernel stack is loaded from in `u.u_pcb.pcb_ksp`, and a `save_state` structure is built on top of this, as well as a stack frame.

`sr5` is saved in the save state, then `uareasid` is copied into `sr5`, so additional `uptr` accesses can use short displacements. The floating point status and exception registers are also saved. If the process is being traced, the callee save registers are copied into the save state as well. Since this entry into the kernel was from another space, the external call mechanism registers must be saved in addition to the "normal" calling convention registers. The registers saved are:

- The user's stack pointer (`sp`)
- The stub return address (`r31`)
- The caller return address (`rp`)
- The user's data pointer (`dp`)
- The user's space registers (`sr4`, `sr6`, `sr7`)

The space registers, stack pointer, and data pointer are then set to the kernel's versions. The `u.u_pcb.pcb_ksp` is zeroed to indicate that execution is currently proceeding on the kernel stack. Finally, since the kernel stack frame is now constructed, it proceeds through its prologue to call the high level code `syscall()`, passing it the system call number and a pointer to the save state.

Slide: The System Call Interface

syscall() - machine/syscall.c

Upon entering C code, some metrics are gathered, the processor is marked as being in system mode, and the user structure is initialized with the proper environment for the call: the error indicator **u_error** is zeroed, **u_syscall** is set to the system call number, and the system call return value **u_rr_val1** is zeroed. The system call number is then used to reference into two arrays: **sysent []**, which contains a pointer to the system call routine and the count of arguments it expects, and **sysaux []**, which contains 64-bit information about the call, such as the 64-bit stub and the number of registers (versus arguments) used by the system call. If there are more arguments than were passed into the low-level initialization via the argument registers, then they are copied out of user space into the **u_arg[]** array in the user structure.¹

сохранить контекст

After the user structure has been set up, a *setjmp()* call is made to allow the process to return from an interrupted system call that was sleeping. If the *setjmp()* recovery path is taken, the **u_error** field in the user structure is checked to see whether an error had occurred before the interrupt. If not, then **u_error** is set to **EINTR**. The return then follows the normal system call return path.

В Linux базисная структура user_struct имеет поле u_arg[] для хранения аргументов системного вызова.

Argument coercion (64-bit only)

In the 64-bit kernel, the **u_arg[]** array is effectively an array of 64-bit values. However, all arguments to system calls are not 64 bits; in fact, they differ between the 32- and 64-bit environments. Thus, before invoking the system call, *argument coercion stubs* must convert the arguments in **u_arg[]** into quantities with the same numerical value, stored in the same place, but in a format ready for use by the kernel function that implements the system call. In other words, each argument must be cleaned up to match its corresponding uap structure field. The stub branches to, rather than calls, the function that actually does the system call. Coercion is implicit (i.e., no stubs are used) in the 32-bit kernel and there are no plans to change this. It is explicit in the 64-bit kernel.

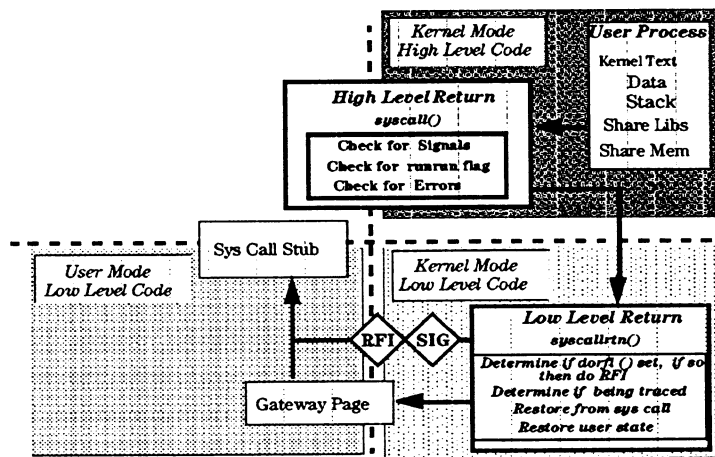
В рамках атомного вызова
вызывающего системный вызов не проверяется,
и внутри sleep — где горит
критическая секция.

sleep — это легкая sleep, а не упрямая

1. This is where the 32-bit versus 64-bit calling convention is honored. The process's runtime environment is checked, and the appropriate number of bytes are copied in from the user stack.

Slide: System Call Return Path

System Call Return Path



a69665

Notes:

Slide: System Call Return Path — 3 *наблюдать поведение ядра*

On return from a system call or trap, the system accomplishes the following tasks in the routine *syscall()*:

Signals

After the individual handler returns, the kernel checks for signals and sets up to return to the stub routine. The return path transitions from high-level return code through low level assembler into the stub to get back to the user. After the system call handling routine exits, a check is made to see whether any signals have been delivered to this thread. If a signal is waiting to be handled, the routine *psig()* will change the stack environment so that the return will be to the signal handler. For the purpose of understanding the system call code, this functionality is transparent.

Return Values

From a user perspective, a system call either returns a value, or it returns a negative number and sets the global variable **errno** to an error value. In the kernel, system calls return errors by setting **u_error** in the user structure, and return values are set in **u_r.r_val1** and **u_r.r_val2**.

Before leaving the high-level code, **u_error** is tested. If it contains a nonzero value, then an error has occurred. If this is the case, then the return status field (i.e., *r22*) of the save state structure is set to indicate that an error has occurred, and the save state return value (i.e., *ret0*) is set to **u_error**. If no error has occurred, and the system call has completed normally (as opposed to being interrupted by a signal), then the system call return value is copied from **u_r.r_val1** into *ret0* of the save state structure.

Context Switch

Before the process leaves the *syscall()* routine, it checks the **runrun** flag to see whether a higher priority thread is waiting to be scheduled. If so, *setrq()* is called to place the thread on the appropriate priority queue, and *swtch()* then performs a context switch.

*При последующей обработ. сигналов ядро
берет упр. сразу обработчика
(→ свободным образом заимствует свои ресурсы)*

Slide: System Call Return Path

The assembly code resumes after *syscall()* returns. The return point begins at the label *\$syscallrtn*. This code simply restores the appropriate user state before jumping back to the stub. First, the save state structure pointer is initialized along with the pointer to the user structure.

Signal Processing

The only minor issue arises from some signal processing that must occur during the low-level return. The save state flags are checked to see whether the return should proceed along the “normal” path into the stub or into the return from interrupt (RFI) path. Since this code path is used for all processes returning from a signal handler, it is possible that the original signal was the result of a trap. If this is the case, then execution continues from the *dorfi* label. As previously mentioned, it is also possible that a user stub is not being returned to, but rather a signal handler is being invoked. Although this is mostly transparent to this section of the kernel, the signal handler has a number of standard arguments, and these must be loaded into the argument registers for the handler. These arguments are restored from the *save state* structure where they were placed by the *sendsig()* routine.

Traced Process Return

The proc structure flags are checked at this time to see whether or not the process is being traced. If so, then all of the callee save registers must be restored from the save state structure. This is because they may have been modified by a *ptrace()* command during the course of the system call. *ptrace()* performs the modification of these registers as a result of a request from the parent process; such requests are handled when *syscall* checks for signals by calling *issig()*. The traced process then follows the same code as a normal process with one exception: if the traced process is being single stepped through a system call, it returns to the user by executing code at the label *sstepsys*. The code functions by placing the return space and offset values into the PC queue fields of the user structure, and then a break instruction is executed. The break handler recognizes the reason that the break occurred and handles actually loading the PC queue with the requested values.

Restoring User State

The return value registers, *ret0* and *ret1*, are loaded with their respective values from the save state structure. The status of the values returned can either be normal return values, or *ret0* can contain an error number to be set in *errno* for the process. This status is saved in *RS*; this is the same register that was previously used to pass the system call number into the kernel, *r22*.

Next, the code simply restores the registers that were originally saved on entry to the low-level code. The restoration of the state registers is basically straightforward; the saved values are retrieved from the save state structure. The floating point exception registers are restored in the same manner that they were saved. The stub return address, the caller’s return address, and the user’s global data pointer are reloaded into the appropriate registers. The user’s data space register *sr5*, is also restored, but as a result the kernel’s data space is only accessible via a long pointer displacement access. A long pointer store word is used to return the kernel stack pointer to *u_pcb.pcb_ksp*. The value stored in *u_pcb.pcb_ksp* is actually the base of the kernel stack, reflecting the popping of the save state structure.

Slide: System Call Return Path

Return to Stub

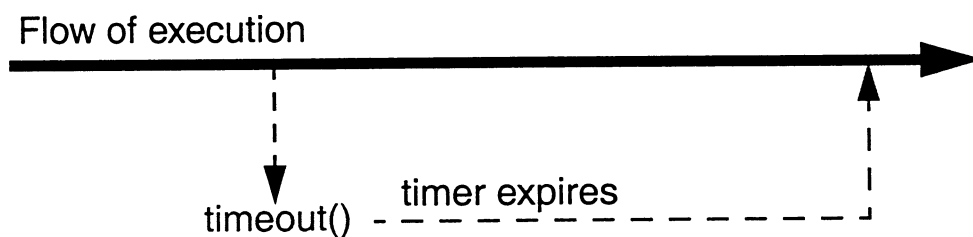
Finally, the return to the stub can be performed. Register 31 contains the return address of the stub, but since space register 0 was not saved at the time of the *ble* instruction, there is no record of which space to return to. Fortunately, sr4 tracks the program counter's space, and it has been restored to its value at the time of the original branch. Therefore, a *ldsid* (load space ID) instruction is used to load sr0 with the external return space. The user's stack pointer is restored in the delay slot of the **branch external** instruction. Note that by ensuring that the two least significant bits of r31 are set, the return "demotes" to privilege level 3.

Stub Exit

Control returns to the stub, and the return status register is checked to see whether or not an error occurred during the call. If an error did occur during the call, then the routine `_syscall_err` is called. This is just a short piece of assembly code which sets the global `errno` with the error number recorded during the call. There is only one instruction remaining to get back to the user: a *bv* (branch vector) back to the return pointer.

Kernel Timeout Services

- ❑ Schedule future events in kernel
- ❑ HP-UX interface is callout table
- ❑ Specified function executed when timer expires



Notes:

Slide: Kernel Timeout Services

The kernel needs to provide a way for users (e.g. drivers) to schedule events that will occur in the future. HP-UX provides this functionality through the callout structures and the routines that interface with them.

If a driver wishes to be notified of some event in the future it will call one of the **timeout routines** in the kernel (*timeout()*, *Ktimeout()*, or *mp_timeout()*) which creates an entry in the callout table. A time is specified for the callout in hundredths of seconds (“ticks”) from the current time.

The driver continues execution and when the timer expires, a routine specified at the time the callout was scheduled will be invoked.

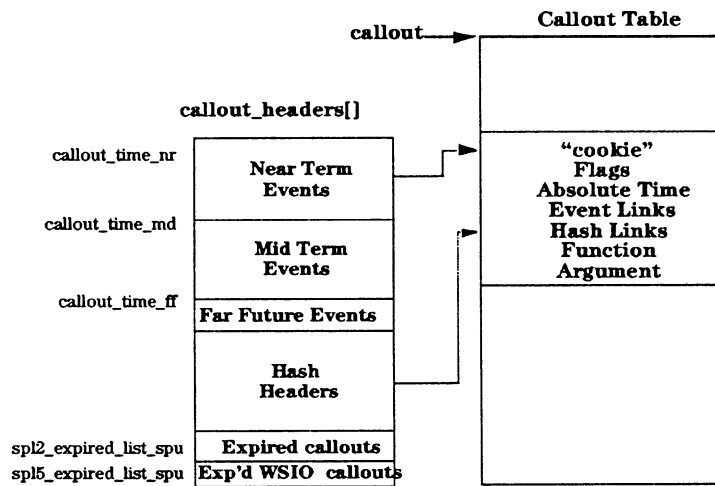
A common example of this is the SCSI driver. When initiating a request to the SCSI bus, the driver will schedule a timeout some number of ticks in the future, identifying *c700_bus_timeout()* as the routine to call. When the request on the bus completes, the timeout is removed from the callout table via the kernel function *untimeout()*. If the request does not complete before the timer expires, then *c700_bus_timeout()* will be called to report the timeout and abort the request.

That is the general overview of how the kernel timeouts are handled. In the next two slides we will look at the callout structures used to handle this and the process of scheduling a timeout.

Monitors are created as sub-managers to
the user processes.

Slide: The Callout Structures

The Callout Structures



a69667

Notes:

Slide: The Callout Structures

To manage the timeouts in the kernel, HP-UX maintains a callout table. The table of actual callout entries are in a structure pointed to by the kernel variable callout. Each entry is of type callout:

```
struct callout {
    u_char c_cookie;
    u_char c_pad;
    callout_flags_t c_flag;
    u_int c_abs_time_hi;
    u_int c_abs_time_lo;
    struct callout *c_time_next;
    struct callout *c_time_prev;
    struct callout *c_hash_next;
    struct callout *c_hash_prev;
    union {
        struct {
            void (*cc_func)();
            caddr_t cc_arg;
        } real_callout;
        struct {
            struct callout *tc_thdr_next;
        } time_header;
        struct {
            u_int hc_walkcount;
            u_int hc_walklength;
        } hash_header;
    } var;
    spu_t c_spu;
};
```

Callout Buckets

Each entry in the callout table belongs to one of the callout buckets. There are three different buckets available:

1. **Near Term Events:** Events in the near term bucket are scheduled to expire in the next few ticks. The header associated with an event will have exactly the same time as the event.
— npocto ne xbeturo neta & near
2. **Mid Term Events:** Events in the mid term bucket are not scheduled to occur for some longer period of time. In this bucket events are linked to a header that represents a range of times.
— ahanzuro & mid
3. **Far Future Events:** Events in the far future bucket are schedule to happen further in the future than the mid term events. There is only one header linking all of the far future events together.

The time ranges for these queue is dynamic. The decision to place an event in the mid term bucket is based on the fact that the time is greater than the greatest time in the near term bucket. As events expire and entries free in the near term bucket other buckets will be reorganized and moved into the near term bucket.

Slide: The Callout Structures

Callout Headers

Each event in the callout table has a header associated with it. Each header has one or more callout events associated with it. The headers are stored in the `callout_headers[]` array. Each entry in this array is actually a timeout entry itself and is therefore of type callout as shown previously.

At the beginning of the `callout_headers[]` array are the **headers for the near term events**. The first entry is pointed to by `callout_time_nr` and the entry following the last by `callout_time_nrNCB_NR`. There is one header for each clock tick; the number of headers is `ncb_nr` headers, which is computed as the tunable `ncallout + nproc/32`, rounded up to a power of 2. Each entry linked to a header in this area will have exactly the same expiration time as the header.

Following the near term headers are the **mid term headers**. The first entry in this area is pointed to by `callout_time_md` and the entry following the last by `callout_time_mdNCB_MD`. The header in this area records a low time and a high time representing a range of entries. All entries in the callout table linked to a header here will fall within that range. There are also `ncb_nr` mid term headers.

After the mid term headers is a single header entry, pointed to by `callout_time_ff`, which is the single header for all **far future events**.

After the three bucket header areas there are headers for the head of the hash list. In addition to being on one of the callout buckets each callout entry will also be on a hash list. This makes it fast and easy to locate an entry for removal from the callout table. The `cc_func` and `cc_arg` are used to generate the hash, since they are passed to `untimeout()`.

Two final entries in the `callout_headers[]` array are the `sp12_expired_list_cpu[]` and `sp15_expired_list_cpu[]` which hold the list of expired timeouts for each spu. Timeouts scheduled by classic s700 drivers with `Ktimeout()` are queued on the sp15 list when they expire; all others are queued on the sp12 list.

Free List

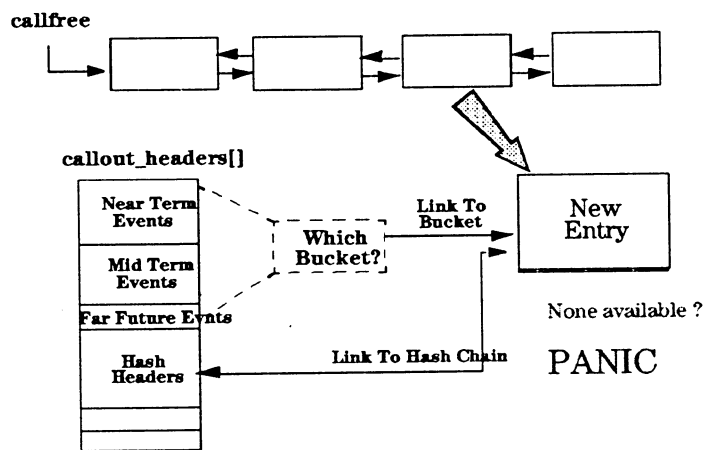
One additional kernel variable, `callfree`, is maintained to keep track of all the callout entries on the free list. If this pointer becomes null (indicating no free entries on the list) then a system panic (“callout table overflow”) will occur.

Module 3 — Kernel Services

Left blank intentionally

Slide: Scheduling a New Timeout

Scheduling a New Timeout



a69668

Notes:

Slide: Scheduling a New Timeout

Users schedule timeouts in the callout table through calls to one of the kernel timeout routines such as `timeout()`, `mp_timeout()`, or `Ktimeout()`. Each of these do some front end processing but it is the routine `settimeout_for_cpu()` which ultimately schedules the timeout.

`Settimeout_for_cpu()` accepts as arguments a pointer to a function to execute when the time expires, a one word argument to be passed to the function, the timeout value specified as the number of ticks from the current time, and some flags.

The `settimeout_for_cpu()` routine performs the following steps:

1. Make sure we are not called with a negative timer. If so print a message and return.
2. Acquire the `callout_lock` spinlock.¹
3. Obtain a callout entry from the free list.
4. Initialize the new callout entry with the function, argument, spu, and flags.
5. Set the time in the entry (`c_abs_time_*`) to equal `ticks_since_boot` and then add the timeout ticks to this value.
6. Call `insert_callout()` to insert the event into the proper bucket and onto the proper hash chain.
7. Release `callout_lock`.

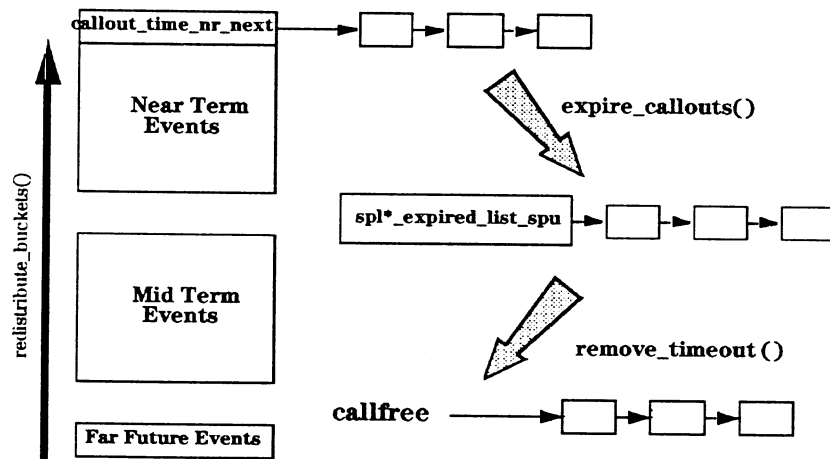
The decision of which callout bucket to put the event into is based on the current range of times in each bucket.

- If the scheduled time is less than the time in the `callout_time_nr_next` entry then place the new entry into the near term bucket.
- If the scheduled time is less than or equal to the time in the `callout_time_nr_futurist` entry then place the new entry into the mid term bucket.
- Otherwise place entry into the far future bucket.

1. Refer to Module 6, Multi Processing for information about spinlocks.

Slide: How Timeouts Expire

How Timeouts Expire



a68660

Notes:

Slide: How Timeouts Expire

Every clock tick (10 ms) the system gets a clock interrupt. The clock interrupt handler calls *hardclock()*, but only on the monarch processor. It calls *expire_callouts()*, which does the following:

1. Acquire the **callout_lock** (spinlock). — *Гарантуйце, што ні адзін працэсар не атрымае доступу да гэтага спіналока*
2. Call *redistribute_buckets()*. This function checks if the header in **callout_time_nr_next** has caught up with that in **callout_time_md_next**; if so, the current mid term bucket is distributed into the near term buckets, and the time in **callout_time_md_futurist** is incremented by **ncb_nr*ncb_nr**, and the **callout_time_md_next** is advanced to the next mid term header.

If the header in **callout_time_md_next** has caught up with **callout_time_ff**, the far future list is scanned, to see if any entries can be moved to the near term or mid term buckets. In addition, the time in **callout_time_md_futurist** is copied into **callout_time_ff**, which is then incremented by **ncb_nr**.

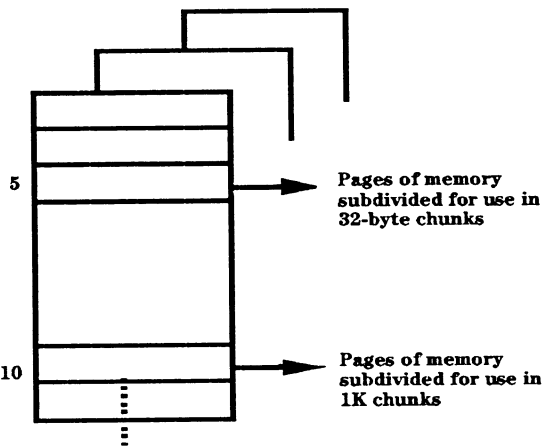
3. Move expired timeouts from the head of the near term header to either the **sp12_expired_list_cpu** or the **sp15_expire_list_cpu**, depending on whether the **DRIVER_TIMEOUT** flag is set (from *Ktimeout()*).
4. Increment the time in **callout_time_nr_futurist** by **ncb_nr**, and advance the **callout_time_nr_next** to the next header.
5. Release **callout_lock**.
6. For each spu, schedule a low-priority interrupt on EIRR bit 21 if any entries are on its sp15 list.
7. For each spu, schedule a low-priority interrupt on EIRR bit 24 if any entries are on its sp12 list.

When the low-priority interrupt is received on each spu, it sets the processor level as necessary, and then for each entry on its expired callout list, frees the entry (with the **callout_lock** locked) and calls the associated function with its argument.

Slide: Kernel Memory Allocation Buckets

Kernel Memory Allocation Buckets

bucket []
(per cpu)



a69670

Notes: 1. Be осторожен с памятью:
1. Визуализация структуры.
2. Визуализация структуры.

Slide: Kernel Memory Allocation Buckets

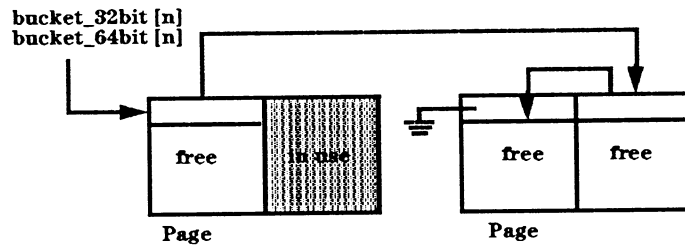
A user process allocates memory by increasing the size of its data area. Since the kernel doesn't have a data region, it has to allocate memory differently. The kernel uses a power-of-2 mechanism for the dynamic allocation of small amounts of memory. Since the allocation unit for the VM subsystem is the page, it takes a page and subdivides this into equally-sized units for that bucket. Unused units are linked in a free list: there is no designation of in-use units.

The 32-bit kernel uses an array called **bucket_32bit[]** and the 64-bit kernel uses an array called **bucket_64bit[]**. Requests for the dynamic allocation of memory are rounded up to the next power of 2 (e.g. if a kernel subsystem needs to store a structure 48 bytes long, this will be allocated from the 64-byte bucket). The smallest bucket contains 32-byte chunks of memory.

For memory allocations larger than a page, there are **page_buckets_32bit** (32-bit kernel) and **page_buckets_64bit** (64-bit kernel). They are used for allocation from 1 to 8 pages long.

Slide: Bucket Contents

Bucket Contents



a69671

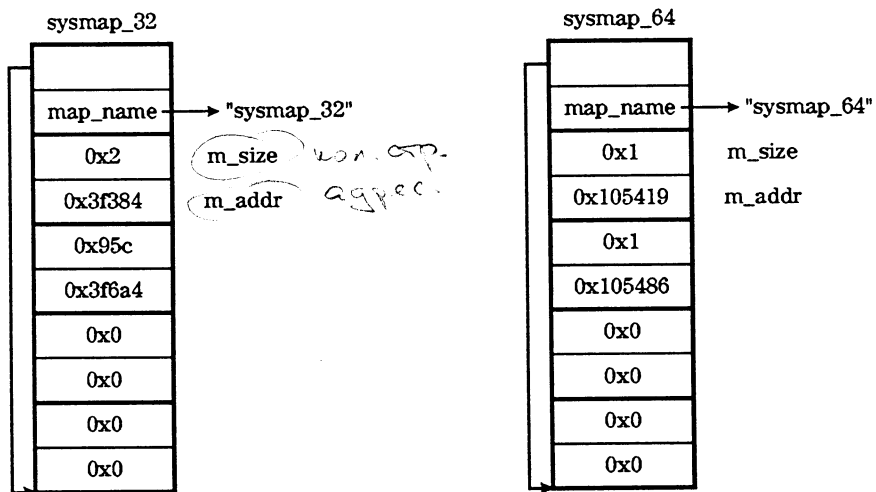
Notes:

Slide: Bucket Contents

The first word of an unused memory chunk points to the next unused chunk. When a chunk is freed up, the contents of the bucket element are put in its first word, and its address is placed in the bucket. When a chunk is needed, the system moves the first chunk's pointer into the bucket and returns the pointer that was in the bucket (which gives the requestor the first chunk in the chain). When physical memory runs low, the kernel wakes up vhand. One of the vhand's tasks is to clean up the malloc buckets, looking for pages with all chunks unused, which can be taken out of the malloc bucket and returned to the physical memory allocator; this is done by the function *kfree_unused()*.

Slide: When the Bucket is Empty: the sysmap

When the Bucket is Empty: the sysmap



a69672

Notes:

Адрес не может быть 0, поэтому всегда добави и вычитается 1 фиктивная страница.

Slide: When the Bucket is Empty: the sysmap

When a bucket is empty, the system needs to actually allocate a new page to fill the bucket. The kernel gets a physical page from the physical memory allocator by calling *kalloc()*. However, the kernel needs a way of assigning a unique virtual address to a newly acquired page.

It does this through the use of a *resource allocation map* called **sysmap**. This structure keeps track of all the virtual addresses available in the first quadrant.

The first entry in a resource map contains:

- pointer to the last entry
- pointer to the name of the map (e.g. “sysmap_32bit”)

Subsequent entries contain:

- **m_size**: number of free pages (0x2 in the example)
- **m_addr**: first page in this area (0x3f384)

Entries are maintained in ascending order of address. No holes are allowed in the map.

In the slide, note that “0” is the end of the sysmap table. **m_addr** should be page number, however “0” cannot be used as the page number since “0” is the end of the sysmap table. Instead “1” is added to all our page numbers before we put the value into **m_addr**. For example 0x105419 is really address 0x105418.

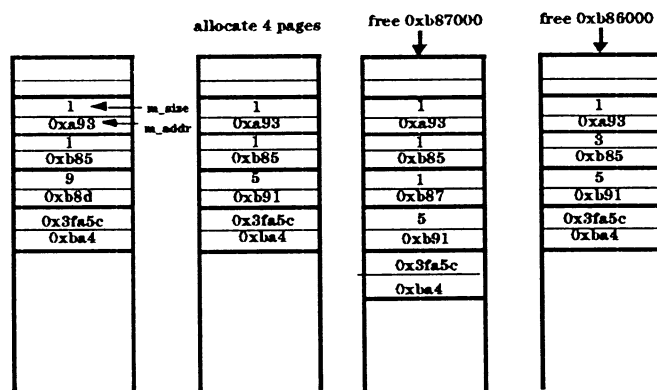
Other resources maps include

- **quad2map_32bit**
- **quad3map_32bit**
- **quad4map_32bit**
- **quad1map_64bit**
- **quad4map_64bit**

The **_32bit** structures are used in both 32-bit and 64-bit kernels. The **_64bit** structures are used only on 64 bit kernels.

Slide: A Populated sysmap

A Populated sysmap



a69673

Notes:

Slide: A Populated sysmap

The slide shows an example of a populated sysmap on a running system.

- One page free at address 0xa93000 (0xa93000-1 = 0xa902ff) <-- m_addr
- Virtual addresses 0xa94000 through 0xb84fff are being used
- One page free at 0xb85000 (0xb85000-1 = 0xb84fff) <-- m_addr
- Virtual addresses 0xb86000 through 0xb8cfff in use
- 9 pages free at 0xb8d000 (0xb8d000-1 = 0xb8cfff) <-- m_addr
- Virtual addresses 0xb96000 through 0xba3fff in use

When allocating virtual addresses from the sysmap, the first entry large enough to accommodate the request is used. If there are insufficient contiguous virtual addresses to satisfy the request, the system panics with: **kalloc: out of kernel virtual space**. Address ranges which are lost are not recovered.

The size of the map is determined by **nsysmap** (this is a tunable kernel parameter) and the amount of physical memory in the machine. When freeing virtual addresses (returning addresses to the map), contiguous space is coalesced on the list. If an entry must be added to the map and all of the entries are already in use a warning is reported: **sysmap: rmap ovflo, lost [3f600,3f6a3]**

Lab: System Calls

- 1) How long do read and write systems calls take?

Using dd a simple test can be made.

Firstly make sure that the file /stand/vmunix is in memory.

```
dd if=/stand/vmunxi of=/dev/null bs=64k
```

Then try timing it again.

```
timex dd if=/stand/vmunxi of=/dev/null bs=64k
```

```
timex dd if=/stand/vmunxi of=/dev/null bs=64
```

These two will run at very different speeds yet achieve the same amount of work, the time difference is down to the extra system calls.

- 2) Go into vi (you don't need a file)

```
vi
```

Suspend the session using ^Z

Find the process id

```
ps
```

Now using Q4 look at the system call information in the UAREA

```
ied -h // .q4_history q4 /stand/vmunix /dev/mem
```

```
q4> load struct proc from proc max nproc
```

```
loaded 276 struct proc's as an array (stopped by max count)
```

```
q4> keep p_stat && p_pid == YOUR_PID
```

```
kept 1 of 276 struct proc's, discarded 275
```

```
q4> load struct kthread from p_firstthreadp
```

```
loaded 1 struct kthread as an array (stopped by max count)
```

```
q4> load struct pregion from kt_upreg
```

```
loaded 1 struct pregion as an array (stopped by max count)
```

```
q4> load struct user from tospace(p_space) | p_vaddr
```

```
loaded 1 struct user as an array (stopped by max count)
```

```
q4> print -tx | more
```

The fields you are interested in are `u_arg[]`, `u_nargs` (if you're running HP-UX 11, you will need to work it out for yourself if you are running 10.20) and `u_syscall`.

The value of `u_syscall` is the system call number. This can be looked up in the header file

```
/usr/include/sys/syscall.h (HP-UX 10.20)
```

```
/usr/include/sys/scall_define.h (HP-UX 11.00)
```

Lookup this system call in the manual, and explain the arguments found in `u_arg[]`. The number of arguments that are relevant can be found from either `u_nargs` or the manual page. The other elements from `u_arg[]` are holding the values of earlier calls that happened to have more arguments.

Lab: Callout Structures

1. Invoke q4:

```
# ied -h $HOME/q4_history q4 -p /dev/mem /stand/vmunix
```

2. Get a listing of all the fields defined in a callout structure:

```
q4> fields struct callout
```

What is the size (in bytes) of the callout structure? _____

(Note: You will substitute this number in later commands wherever the “<SIZE>” token is found.)

3. Load all the callout structures from the callout table:

```
q4> load struct callout from callout max ncallout
```

How many structures were loaded? _____

4. List all the different flag fields in these structures:

```
q4> print c_flag | sort -u
```

What types of callout structures are there?

5. Eliminate all the non-PENDING_CALLOUTs:

```
q4> keep c_flag == PENDING_CALLOUT
```

How many structures were kept? _____

Module 3 — Kernel Services

Lab: Callout Structures

6. List all the different functions pointed to by these structures:

```
q4> print -x var.real_callout.cc_func | sort -u
```

How many are there? _____

7. In a separate window, start up the “adb” kernel debugger:

```
# adb -k /stand/vmunix /dev/mem
```

(Note: You will not get a prompt from “adb”.)

8. Select one or more addresses from the output of step 6 and enter it to “adb”:

```
0x<SELECTED_ADDRESS>/a
```

What is the name of this function? _____

Repeat for any other addresses you wish.

9. Display the first 10 instructions of this function:

```
0x<SELECTED_ADDRESS>,0xa?ia
```

What is the first instruction? _____

(Note: “adb” seems to be better at displaying symbol names and instructions than “q4”.)

10. Back in your “q4” window, load the “near term” callout headers:

```
q4> load struct callout from callout_headers max (callout_time_md - callout_time_nr) /  
<SIZE>
```

How many structures were loaded? _____

Lab: Callout Structures

11. List the different types from the flag fields:

```
q4> print c_flag | sort -u
```

What type of callout structures are these?

12. List the absolute time fields for these headers:

```
q4> print c_abs_time_hi c_abs_time_lo
```

Note the sequence of the numbers.

What is the numerical difference between adjacent time fields? _____

How much time does this difference represent? _____

13. Load the headers for “mid term” events:

```
q4> load struct callout from callout_time_md max (callout_time_ff -  
callout_time_md) / <SIZE>
```

How many structures were loaded? _____

12. List the absolute time fields for these headers:

```
q4> print c_abs_time_hi c_abs_time_lo
```

Note the sequence of the numbers.

What is the numerical difference between adjacent time fields? _____

How much time does this difference represent? _____

13. Load the callout header for all “far future” events:

```
q4> load struct callout from callout_time_ff
```

Module 3 — Kernel Services

Lab: Callout Structures

14. Display the contents of this structure and fill in the following values:

```
q4> print -tx
```

```
c_hash_next: _____
```

```
c_hash_prev: _____
```

```
var.real_callout.cc_func: _____
```

15. Load the linked list of callout structures attached to this one:

```
q4> load struct callout from c_time_next max ncallout next c_time_next
```

How many structures were loaded? _____

16. List out the flag fields:

```
q4> print c_flag
```

Where is the header structure in this list? _____

What new type shows up? _____

17. List out the absolute times for these structures:

```
q4> print c_abs_time_hi c_abs_time_lo
```

18. Load the hash headers:

```
q4> load struct callout from callout_time_ff + <SIZE> max  
(callout_time_ff - callout_time_md) / <SIZE>
```

What is the absolute time set to? _____

Lab: Callout Structures

19. Display the flags, time and links:

```
q4> print -x c_flag c_abs_time_lo c_time_next c_hash_next
```

What is the new type of header? _____

What is the absolute time set to? _____

Are there links to an event chain? _____

Are there links to a hash chain? _____

20. Load the “expired” headers:

```
q4> load struct callout from callout_time_ff + (callout_time_ff -  
callout_time_md + <SIZE>) max 2
```

21. Display all the fields:

```
q4> print -tx
```

What new header type are these? _____

How does c_time_next compare to addrof? _____

Why? _____

Module 3 — Kernel Services

Module 4

Process Management

The world is a complicated place, Hobbes.

Whenever it seems that way, I take a nap in a tree and wait for dinner.

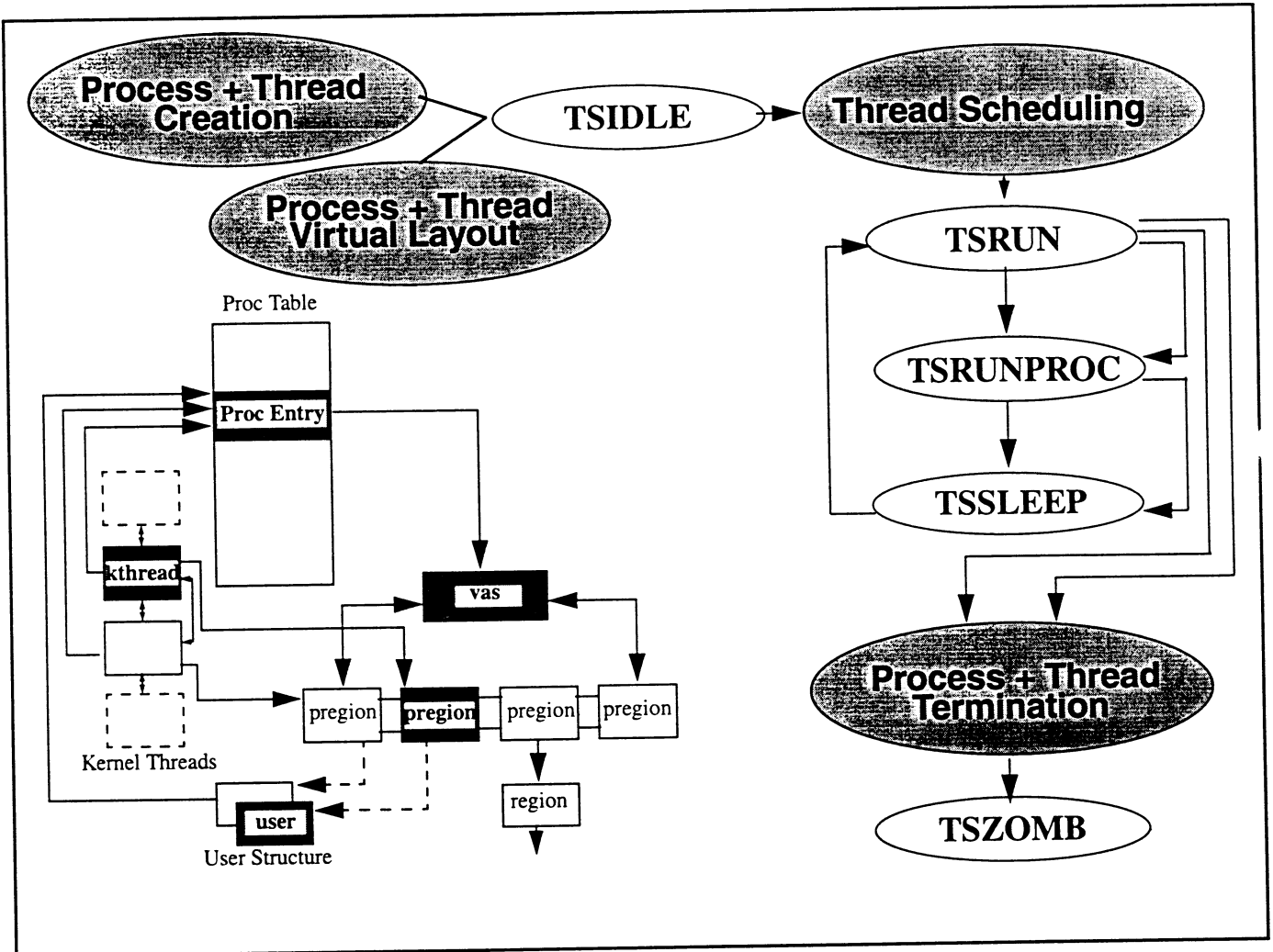
Sam Watterson

Objectives :

- Define how a process and the main thread are created.
- Define how a process and the main thread terminates.
- Define the process structure and all components that make up the virtual image.
- Define the process schedulers.
- Define how a thread's priority changes through time and the transitions it makes.

Module 4 — Process Management

Slide: Introduction



Notes:

Module 4 — Process Management

Slide: Introduction

The intent of this module is to provide a detailed overview of Process and Thread Management within HP-UX. This will consist of the following areas:

- Definition of a Process and Thread.
- How a Process and Thread are created using a *fork()* and *vfork()* syscall.
- How an address space is overlaid via a call to *exec()*.
- How a Process and Thread terminate.

Once we understand how and when a Process and Thread are created, the next step will be the review of the Virtual area that make-up the Process and Thread Management Subsystem. This area will consist of the following:

- How the virtual space is allocated for a Process and Thread Structure.
- The Components of the Process and a Thread.
- How the Process Table (Proc Table), Thread, Virtual Address Space (vas) and Pregions are structured.
- Review of the UAREA.

Thread Scheduling will contain a complete overview of the following:

- Overview of the 3 Thread Scheduling queues.
- How the Thread Run queues are setup and initialized.
- A walk through time with the scheduling routines and how we adjust a Thread's priority.
- Define and determine how we handle a timeslice.
- Discuss the routines involved with context switch and how this affects the *Process Control Block (pcb)*.

Slide: Program, Process and Thread Definitions

Program, Process and Thread Definitions

What is a Program

A file containing instructions to be executed
Can be either binary or interpreted (i.e. a script)

What is a Process

An instance of a running program

Processes no longer run — *они уже не существуют*

Processes are containers for threads

What is a Thread

An independently schedulable part of the program

Shares the virtual address space, and files of the whole process

There must always be at least one thread within the process.

a696271

Notes:

Slide: Program, Process and Thread Definitions

What is a Program?

A program is a file containing instructions to be executed. In Unix programs can either contain machine code instructions that can be executed by the processor directly, or they can be interpreted programs where some other program will need to be executed to handle the instructions found in the file. Interpreted programs are often referred to as scripts.

When a program is executed, using an `exec` system call, the kernel needs to know what type of program file is being requested. On some operating systems there is a strong concept of file typing, but in Unix a regular file is simply a *bucket* full of bytes, and it is the applications problem to work out its type. However in the case of executing programs, it is the job of the kernel itself to do this. The technique that is used is known as magic numbers - the programmer thinks of a number, and imbeds it at a chosen location in the file (often the beginning) and trying to identify a file, you just look at the location, and see whether your number comes up. Many of the magic numbers used on the system (by applications/commands as well as the kernel) are listed in the file `/etc/magic`, this file is the main place that the `file(1)` command looks in to identify files.

The various forms of binaries that HP-UX uses all have their own magic numbers. Scripts ideally start with `#!` and the name of the interpreter to use. }

отрабатывается непосредственно ядром
If no magic number can be found then the file is assumed by the kernel to be a Posix Shell script, and as such a Posix shell is `exec'd` and the file is passed to it.

What is a Process?

SOID где купитов не Binaries. где
суммировка UNIX, но HP-UX
это генерат.

The “*How HP-UX Works: Concepts for the System Administrator*” provides a good overview of a process in the 9.0X environment:

***“A process is a running program, managed by such systems components as the scheduler and the memory management subsystem. Although processes appear to the user to run simultaneously, in fact a single processor is executing only one process at any given moment.*”**

Described most simply, a process consists of text (the code that the process runs), data (used by the code), and stack (a “place” in the kernel where the parts of a program are stored as the process is running). Two stacks are associated with a process, kernel stack and user stack. The process uses the user stack when in user space and the kernel stack when in kernel space.”

This definition however no longer holds true from HP-UX 10.10, where the kernel schedules threads rather than processes. In this newer model, a process is a container for a group of threads. It provides the common environment for them to run in, such as the address space.

Slide: Program, Process and Thread Definitions

What is a thread?

With the release of 10.10 (Walnut Creek) the operating system contained a thread structure. The new structure had been put in place to support a multiple thread environment, but HP only implemented a single threaded approach for the 10.10 release (i.e., each process contains only one thread).

In the previous process based kernel, each process is represented in the kernel by two data structures, proc and user. The proc structure is non-swappable, and user is swappable. Each process also has a kernel stack, which is allocated with the user structure in the Uarea.

In the thread based kernel, a new thread structure is needed. For each process, there is still a proc structure. The proc structure is memory resident and it contains per-process data that is shared by all the kernel threads within the process. Each kernel thread is represented by a kthread structure and a user structure. There is also a separate kernel stack for each kernel thread. A Uarea consists of a user structure and a kernel stack. While the Uarea is swappable, the kthread structures are always memory resident.

Each kthread structure contains a pointer to its associated proc structure. It has a pointer to the next thread within the same process. The head and tail pointers to a process' thread list are included in the proc structure. All the active threads in the system are linked together on the active threads list.

The design and implementation of Kernel Threads in HP-UX will be separated into three different phases of work:

Phase I: Release 10.10

Moving from a process-based kernel to a thread-based kernel. This involves splitting the proc and user structures and providing a thread structure. It is basically doing the base data structure changes for threads across the entire kernel. At the end of the phase, one process will correspond to one thread and threads become the schedulable entities.

Phase II: Release 10.30/11.0

Added core support for multiple kernel threads per process. HP-UX 10.30 provides a POSIX 1003.1b compliant API for multi-threaded application development based on kernel threads. The current model is a "1x1" model, meaning each user thread is bound to a single kernel thread. A POSIX kernel threads library, libpthread, has to be linked in order to use these new interfaces in multi-threaded applications. HP-UX 11.0 adds more thread features to make it POSIX 1003.1c compliant.

Phase III: Future Release

Adding the support for the MxN model. Refer to the white paper for details on available models

Slide: Program, Process and Thread Definitions

used.

This new structure can also be defined from the “*Threads White Paper*” (KDB Doc id FGD9404221640) and serves as a good intro to understanding threads:

“Similar to a process, a thread contains a program counter, a stack, and a set of machine (or register) states for management purposes. These “management” facilities are used to maintain a thread’s “state” information throughout its entire life. State information monitors the condition of an entity (like a thread or process); it provides a snap-shot of an entity’s current condition. For example, when a thread context switch takes place, the newly scheduled thread’s register information tells the processor where the thread left off in its execution. More specifically, a thread’s program counter would contain the current instruction to be executed upon start up.”

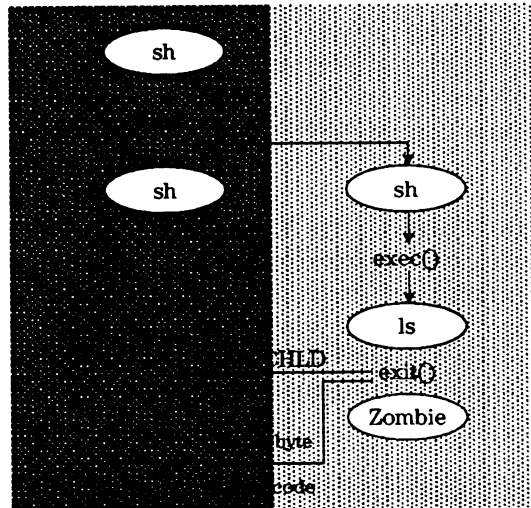
In the threads model, processes are still the “containers” for the set of instructions that carry out the overall task of the program. In the threads model, a programmer may divide this set of information into logical, executable segments called threads. Each thread shares its host process’ address space, giving it access to resources that are owned (or used) by the process (e.g. a process’ pointers into the file descriptor table). Each thread also has its own resources (i.e private data structures that maintain state information, and a unique counter) for management purposes.”

We will review in detail the layout of the process and the thread structures later in this module. First, let us begin by understanding the life cycle a process and thread. Understanding how a process is born and how it interacts with its thread will help clear the way to understanding their internal structures later in this module.

Slide: System Calls for a Process's Lifecycle

System Calls for a Process Lifecycle

Parent



Child

a696272

Notes:

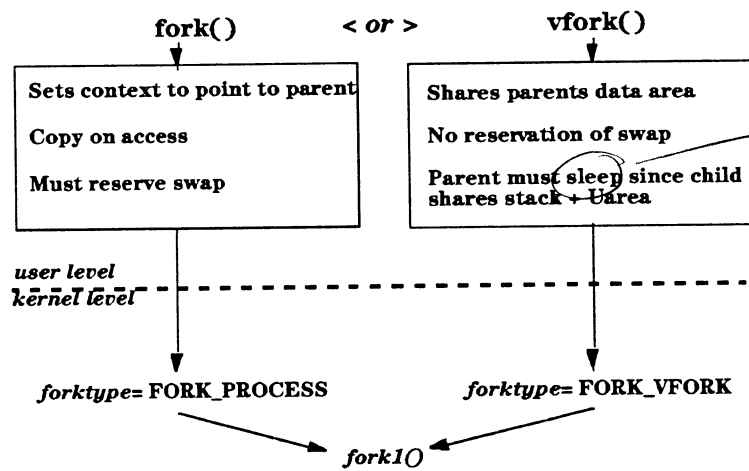
Slide: System Calls for a Process's Lifecycle

Now, based on the definition of a process and thread, let us take a look at how they are created, and their life cycle. The overview on the previous page shows some of the routines that make up the life cycle of a process and its thread. We will discuss these in detail. The information to follow will walk us through the creation of a child process, and its thread, via the *fork()* syscall. Then we will discuss what happens when a call to *vfork()* is made, and how the *exec()* system call changes the program that is being run.

The slide on the previous page shows an overview of the areas we will cover based on a user calling the *fork()*, *exec()*, *wait()* and *exit()* system calls.

Slide: Process Creation: fork() or vfork()

Process Creation: fork() or vfork()



a696273

Notes:

Slide: Process Creation: `fork()` or `vfork()`

How is a process created?

Before we jump into a detailed overview of how the process and thread structures are laid out, let's take a look at how a process is created and the routines involved:

Overview

Process 0 is created and initialized at system boot time but all other processes are created by a `fork()` or `vfork()` system call. A process is always created with a single thread. If a multi-threaded process calls `fork()`, the new process contains a replica of the calling thread and the process's entire address space.

The original `fork()` system call worked by copying the entire parent process for the child. Since typically the child executes an `exec()` shortly after being `fork()`'d, and this results in throwing away the newly created address space it can be wasteful in some circumstances. However it is this idea of making a copy of the original program that gets to run first that results in most to the capabilities that we value in the Unix environment. Features such as easy IO redirection and piping are a result of having a shell running as the child to setup the file descriptors appropriately invoking the new command.

Having a new copy of the original program is however not always needed, and under these circumstances the `fork()/exec()` model where the `fork()` did make a new copy is wasteful. So With the advent of Virtual Memory Unix at Berkeley a second type of `fork` was added, `vfork()` the virtual memory friendly `fork` was added. In the `vfork`, the new process does not acquire a new address space of its own, but rather runs in its parents one. Since it would not be tenable to have two processes sharing the same address space at the same time, the parent process is put to sleep until it's address space is no longer needed by the child since its has either performed an `exec()` and acquired a new address space to reflect the new program, or has issues and `_exit()` and terminated.

`Vfork` however violates a major feature of the `fork/exec` model. Where the child has it's own address space, it can not pollute it's parents one. With a `vfork` where the child runs in the parents address space then any changes made by the child will affect the parent. Therefore the children of `vforks` are not supposed to modify any data, in practice this can be difficult.

The ability of the `vfork` children to modify the address space of the parent was considered also to be a BUG. This has led it to have a varied history in HP-UX.

Slide: Process Creation: `fork()` or `vfork()`

History of `fork()` and `vfork()`

7.0X

`Fork()` copies parent data area

`vfork()` process and child shares parents data area, the register context is changed to point to the parent area

8.0X

Changed `fork` on 700s and 800s to do “copy on write” which saves time. This caused swap reservation problems on the Series 700s. A patch was created to provide old functionality.

9.0X

`fork()` and `vfork()` remained the same for the Series 800 and Series 700 utilized `vfork()`.

10.0X

In previous versions of HP-UX, `vfork()` on the Series 800s did not implement true `vfork()` semantics, but it was actually implemented as `fork()` with “copy-on-write” for the data segment of the child. On the Series 700, `vfork()` has not changed from the 9.0X release.

The 9.0 Series 700 internal code structure and performance characteristics are provided on 10.0 Series 800 computers. Note that `vfork()` exists on 9.0 Series 800 computers, but it is implemented as `fork()`. The programming interface has not been changed. For more information, refer to the `vfork(2)` manpage.

`fork()` - `kern_fork.c`

Per the man page for `fork(2)`:

The `fork()` system call causes the creation of a new process. The new child process is created with exactly one thread or lightweight process. The new child process contains a replica of the calling thread and its entire address space, possibly including the state of the mutexes and other resources.

If the calling process is multi-threaded, the child process may only execute async-signal safe functions until one of the `exec()` functions is called. An async-signal safe function is a function which may be safely invoked from a signal handler, and allows a thread to synchronously wait for an asynchronous signal which was sent to the process. Fork handlers may be installed via `pthread_atfork()` in order to maintain application constants across `fork()` calls (i.e., release resources such as mutexes in the child process).

Slide: Process Creation: fork() or vfork()

This means that the child process inherits the following attributes from the parent process:

- + Real, effective, and saved user IDs.
- + Real, effective, and saved group IDs.
- + List of supplementary group IDs (see *getgroups(2)*).
- + Process group ID.
- + Environment.
- + File descriptors.
- + Close-on-exec flags (see *exec(2)*).
- + Signal handling settings (*SIG_DFL*, *SIG_IGN*, address).
- + Signal mask (see *sigvector(2)*).
- + Profiling on/off status (see *profil(2)*).
- + Command name in the accounting record (see *acct(4)*).
- + Nice value (see *nice(2)*).
- + All attached shared memory segments (see *shmop(2)*).
- + Current working directory
- + Root directory (see *chroot(2)*).
- + File mode creation mask (see *umask(2)*).
- + File size limit (see *ulimit(2)*).
- + Real-time priority (see *rtprio(2)*).

as well

Each of the child's file descriptors share a common open file description with the corresponding file descriptor of the parent. This implies that changes to the file offset, file access mode, and file status flags of file descriptors in the parent also affect those in the child, and vice-versa.

The child process differs from the parent process in the following ways:

partial

- The child process has a unique process ID.
- The child process has a different parent process ID (which is the process ID of the parent process).
- The set of signals pending for the child process is initialized to the empty set.
- The trace flag (see the *ptrace(2)* *PT_SETTRC* request) is cleared in the child process.
- The *AFORK* flag in the *ac_flags* component of the accounting record is set in the child process.
- Process locks, text locks, and data locks are not inherited by the child (see *plock(2)*).
- All *semadj* values are cleared (see *semop(2)*).
- The child process's values for *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to zero.
- The time left until an alarm clock signal is reset to 0 (clearing any pending alarm), and all interval timers are set to 0 (disabled).

Slide: Process Creation: *fork()* or *vfork()*

vfork() - *kern_fork.c*

Per the man page for *vfork(2)*:

vfork() is a higher performance version of *fork()* that is provided on some systems where a performance advantage can be attained.

The newly created child process will only contain one thread - a copy of the thread calling *vfork()*.

vfork() differs from *fork()* only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if *vfork()* is misused.

The use of *vfork()* for any purpose except as a prelude to an immediate *exec()* or *exit()* is not supported. Any program that relies upon the differences between *fork()* and *vfork()* is not portable across HP-UX systems.

All HP-UX implementations must provide the entry *vfork()*, but it is permissible for them to treat it identically to *fork()*. On some implementations, the two are not distinguished because the *fork()* implementation is as efficient as possible. Other versions may do the same to avoid the overhead of supporting two similar calls.

vfork() can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to do an *exec()* (see *exec(2)*), the data space copied from the parent to the child by *fork()* is not used. This is particularly inefficient in a paged environment, making *vfork()* particularly useful. Depending upon the size of the parent's data space, *vfork()* can give a significant performance improvement over *fork()*.

vfork() differs from *fork()* in that the child borrows the parent's memory and thread of control until a call to *exec()* or an *exit()* (either by a call to *exit()* or abnormally (see *exec(2)* and *exit(2)*). The parent process is suspended while the child is using its resources.

vfork() returns 0 in the child's context and (later) the pid of the child in the parent's context.

vfork() can normally be used just like *fork()*. It does not work, however, to return while running in the child's context from the procedure which called *vfork()* since the eventual return from *vfork()* would then return to a no longer existent stack frame. Note, *_exit()* should be called instead of *exit()* if you cannot be sure that *stdio* buffers are not flushed at the time of the fork/vfork, as this will cause multiple flushes of the same data - once from the parent and again for the child.

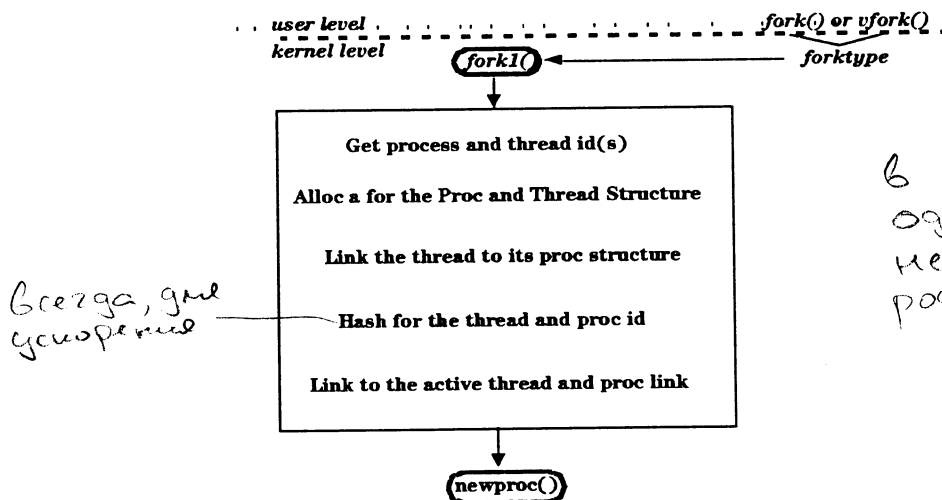
The [*vfork,exec*] window begins at the *vfork()* call and ends when the child completes its *exec()* call.

Module 4 — Process Management

Left blank intentionally

Slide: Process Creation: fork1()

Process Creation, fork1()



a696274

Notes:

Slide: Process Creation: fork1()

fork1() - *kern_fork.c*

Called by *fork()* to create the new process. *fork1()* will function in the following manner:

fork1() begins by passing the type of fork called (*forktype*). The *forktypes* are:

FORK_PROCESS - specified when the *fork()* system call is utilized
FORK_VFORK - specified when the *vfork()* system call is utilized

The following information is based on a call to **FORK_PROCESS** or *fork()*.

getnewpid() - *pm_getpid.c*

Is called by *fork1()* to setup the *pid* (process id) of the new process. *mpid* is used by *getnewpid()* to generate a unique *pid*. *mpid* must be initialized so that no process will use any of the pids 0 - 7 or "0 - **PID_MAXSYS**". These are reserved for system processes only.

The systems process pids are defined in the following manner:

PID_SWAPPER	= 0
PID_INIT	= 1
PID_PAGEOUT	= 2
PID_STAT	= 3
PID_UNHASH	= 4
PID_NETISR	= 5
PID_SOCKREGD	= 6
PID_COMMIT	= 7
PID_MAXSYS	= 7

getnewtid() - *pm_getpid.c*

Is called by *fork1()* to setup the *tid* (thread id) for the main thread in the new process. This is achieved by *mtid*. In the same manner *mpid* locates a unique pid for the process, *mtid* must be initialized so that no thread will use any of the tid 0 - 7 or "0 - **TID_MAXSYS**", which are reserved for system threads.

The system thread *tids* are defined in the following manner:

Slide: Process Creation: fork1()

(In HP-UX 10.10, there is a one to one relationship between a proc id and thread id. As of HP-UX 10.30 and 11.0, there is a one-to-many relationship between a proc id and thread id's.)

TID_SWAPPER	= 0
TID_INIT	= 1
TID_PAGEOUT	= 2
TID_STAT	= 3
TID_UNHASH	= 4
TID_NETISR	= 5
TID_SOCKREGD	= 6
TID_COMMIT	= 7
TID_MAXSYS	= 7

Before we allocate a *proc* and *thread* structure, we check and make sure the user process is not exceeding *nproc* (Number of *proc* table entries). We will discuss the layout of the *proc* table later in this module.

allocproc() - pm_proc.c

Is called to allocate a free *proc* structure entry and clear it out. We also allocate any memory needed for the process via a call to the kernel memory allocator **MALLOC**. At this point the allocated process table slot is removed from the free list. The entry is marked “*process creation in progress*”, which corresponds to a *p_flag* definition of **SIDL** (process creation state). We will discuss process flags and states later in this module. The process slot is not linked into the active process list at this point until *all* of the thread structures are allocated and initialized.

allocthread() - pm_proc.c

Is called by *fork1()* to setup the thread structure, and add it to the active thread list. It will be initialized to a *kthread* flag state of **TSIDL** (thread creation state).

link_thread_to_proc() - pm_proc.c

Will link the thread to the per-process threads list of the corresponding process.

thread_hash() - kern_fork.c

Will link the *kthread* structure to a hash chain for the kernel thread id.

Slide: Process Creation: fork1()

proc_hash() - *kern_fork.c*

Will hash the proc structure for our *uid* and *pid*.

link_thread_to_active() - *pm_proc.c*

Insert thread slot onto the active list after *thread[0]*. The *link_thread_to_active()* routine does a doubly linked list insert using indexes instead of pointers.

link_proc_to_active() - *pm_proc.c*

Insert proc slot onto the active list after *proc[0]*. The *link_proc_to_active()* does a doubly linked list insert using indexes instead of pointers:

switch(newproc)

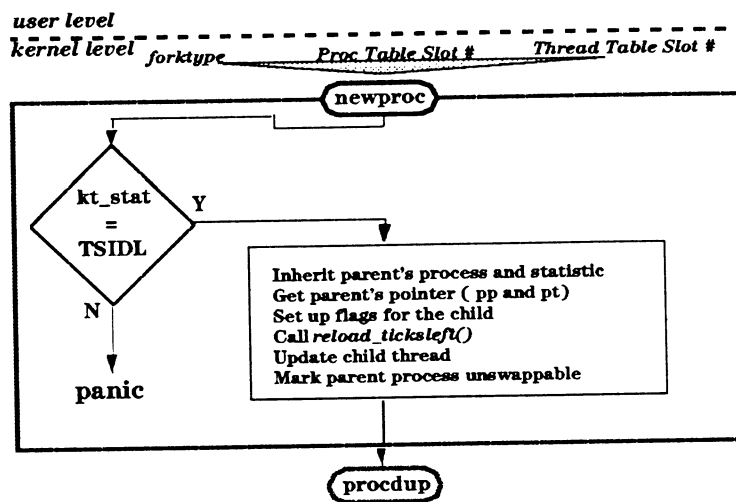
fork1() will call *newproc()* with the *forktype*, *proc table slot*, and *thread* to create a new process.

newproc() will return to *fork1()* and invoke a case statement based on the return value:

- 1 = Indicates this is a child process executing. The child's user area is filled in the information.
- 0 = Indicates the parent is executing, the child's pid is copied to the user are and the case statement is exit.

Slide: Process Creation: newproc()

Process Creation: newproc()



a696275

Notes:

Slide: Process Creation: `newproc()`

`newproc()` - `kern_fork.c`

`newproc()` is called with the *forktype*, the *proc table slot*, and the *thread table slot* by `fork1()`. The *forktype* passed will be “`FORK_PROCESS`”.

`newproc()` will complete the following steps before returning to `fork1()`:

- It verifies the *kt_stat* (thread state) is `TSIDL`, if not we will panic.
- Get a pointer to the parent process and thread.
- Call `vfork_buffer_init()` to allocate and initialize `vforkinfo` buffer in the case of `FORK_VFORK` fork type.
- If the process is a multithreaded process, we need to make sure no forks or exits are occurring at the same time:
 - + Call `process_write_sync()` to gain exclusive write access to the process.
 - + Call `process_wide_suspend()` to suspend all other threads in the process.
- If *forktype* is `FORK_VFORK`, we can now safely assign the `vforkinfo` buffer into the parent process, since no other competing `vforks` can get through in a multithreaded process.
- Call `procdup()` to create a child process/thread pair.

`newproc()` will call `fork_inherit()` to do all the simple assignments. This will include direct assignment from the parent's proc structure to the child's.

- In a Multiprocessor environment, if `vfork()` is called, we do not want the child to be picked up by another processor before the parent is fully switched out. So, we leave the `TSRUNPROC` bit on. The code that picks up a process to run (`find_process_my_spu()`) will ignore `TSRUNPROC` processes. When the parent has switched out completely, it will clear the `TSRUNPROC` bit for the child.
- Next `newproc()` calls `reload_ticksleft()` to update the child thread (*ct*) and mark the parent proc pointer (*pp*) unswappable by calling `make_unswappable()`. This is to prevent to parent from being swapped while we copy the environment for the child.
- `newproc()` issues a switch to `procdup()` and passes the *forktype*, *parent's proc pointer* (*pp*), *child's proc pointer* (*cp*), *parent's thread* (*pt*), and *child's thread* (*ct*).

Module 4 — Process Management

Slide: Process Creation: `newproc()`

- If we're in the parent thread's context:
 - + Call `process_wide_unsuspend()` to unsuspend all other threads in the process we suspended with `process_wide_suspend()` call above.
 - + Call `process_wrtie_sync_done()` to clear this thread from being a writer and see if there are any writers or readers waiting to wake them up.

`fork_inherit()` - *kern_fork.c*

Perform all of the direct assignments from the parent proc structure to the child's.

`reload_ticksleft()` - *pm_policy.c*

Fill in the `kt_ticksleft` corresponding to `kt_schedpolicy`

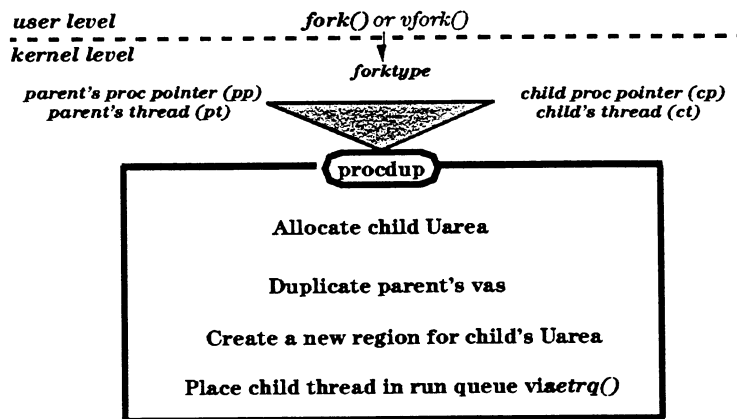
`make_unswappable()` - *kern_fork.c*

Partially simulate the environment of the new process so that when it is actually created (by copying) it will look right.

Left blank intentionally

Slide: Process Creation: procdup()

Process Creation: procdup()



a696276

Notes:

Slide: Process Creation: `procdup()`

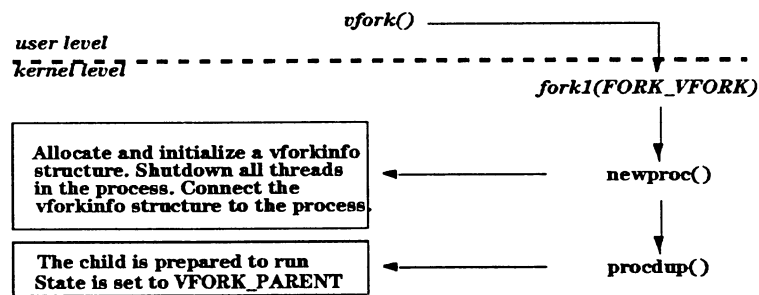
`procdup - pm_procdup.c`

`procdup()` is called from `newproc()` when most of the child's structure has been created. `newproc()` issues a switch to `procdup()` and passes the *forktype*, *parent's proc pointer* (`pp`), *child's proc pointer* (`cp`), *parent's thread* (`pt`), and *child's thread* (`ct`). `procdup()` will:

- Build a Uarea and address space for the child.
- Duplicate the parent's virtual address space.
- Create a new region for the child's Uarea.
- Attach the region: `PF_NOPAGE` keeps *vhand* from paging the Uarea.
- Mark the pregion to be owned by new child thread. The address space is owned by the process.
- Place the child thread on the run queue by calling `setrq()`.

Slide: Process Creation: vfork() Example

Process Creation: vfork() example



a696277

Notes:

Slide: Process Creation: `vfork()` Example

Before we jump into the other method of process creation (`exec()`), let us take a quick look at what happens when a child process is created via a `vfork()` call.

`vfork()` calls `newproc()` to create a new process.

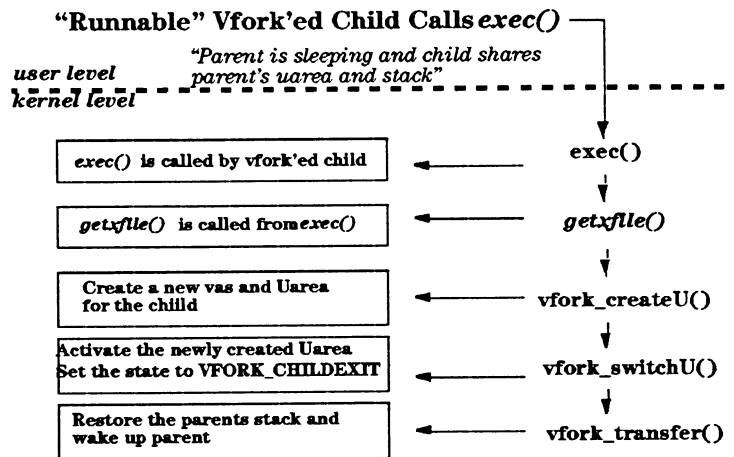
`newproc()` calls `vfork_buffer_init()` to allocate the `vforkinfo` structure:

```
struct vforkinfo {
    vfork_state_t vfork_state;
    struct kthread *ptheadp;
    struct kthread *cthreadp;
    pgcnt_t buffer_pages;
    unsigned_long u_and_stack_len;
    caddr_t saved_rp_ptr;
    long saved_rp;
    caddr_t u_and_stack_buf;
    struct vforkinfo *prev;
    struct pregion *p_upreg;
}
```

This structure is referenced through the `proc` structures of the parent/child `vfork` pair. It remains until the child does an `exec()` or `exit()`, at which point the structure is freed. The structure holds state information about the `vfork` and holds a copy of the parent's stack, while the child decides to `exit()` or `exec()`. Once the `vforkinfo` structure is allocated, the `vfork_state` is set to `VFORK_INIT`. When we get to `procdup()`, the child is made runnable, the state is set to `VFORK_PARENT`, and the parent sleeps. At this point, the parent and child share the same `UAREA` and `STACK`.

Slide: Process Creation: vfork() Calls exec() Example

Process Creation: vfork() Calls exec() Example



a696278

Notes:

Slide: Process Creation: `vfork()` Calls `exec()` Example

Based on the last page, the child process “runs” using the parents stack until it does an `exec()` or `exit()`. In the case of `exec()`, the routine `vfork_createU()` is called to create a new `vas` and `Uarea` for the child (it copies the current ones into these). We then call `vfork_switchU()` to activate the newly created `Uarea` and to set up the `space` and `pid` registers for the child. The state is then set to `VFORK_CHILDEXIT`. On an `exec()`, we call `vfork_transfer()` directly from `vfork_createU()` to restore the parents stack. We then wake up the parent.

`vfork_createU` - `pm_procdup.c`

Called from `getxfile()` when child does a `vfork()` followed by an `exec()`. This routine sets up a new `vas` and duplicates the `stack/U` from the parent (which it's been using up till now). It switches the child over to use this `stack/U`.

`vfork_switchU` - `asm_utl.s`

`vfork_switchU` switches the current process onto a new `U-area/stack`.

`vfork_transfer` - `kern_fork.c`

This is code that does all the work for `vforks`. When a process does a `vfork`, a `vforkinfo` struct is allocated and shared between parent & child. Both the parent and child's `proc` structure has a pointer to the `vforkinfo`.

Once allocated in `vfork_buffer_init()`, the `vfork_state` is set to `VFORK_INIT` until we get to `procdup()` where the child is made runnable, the state is set to `VFORK_PARENT` and the parent sleeps. At this point the parent and the child share the same `UAREA` and `STACK`.

When we go through `save` before starting the next process, we check the `UAREA` of the process whose state we just saved to see if it's doing a `vfork` and if it's the parent (`VFORK_PARENT` state). If so, we call `vfork_transfer` within the `VFORK_PARENT` state. The schedlock is held to prevent the child, or any process, from starting to run until we finish the `save()`.

In the `VFORK_PARENT` state, we calculate the size of the stack and `uarea` and copy it into the `vforkinfo` `u_and_stack_buf` area. This is so we can restore the parent later when the child does an `exec` or `exit`. We then set the state to `VFORK_CHILDRUN` and return.

In the case of an `exit`, the state is changed to `VFORK_CHILDEXIT` in `exit()` by the child, we then wake the parent and call `switch()`. When the parent goes through `resume`, the parent will restore its own stack.

Overview of the steps taken:

Slide: Process Creation: `vfork()` Calls `exec()` Example

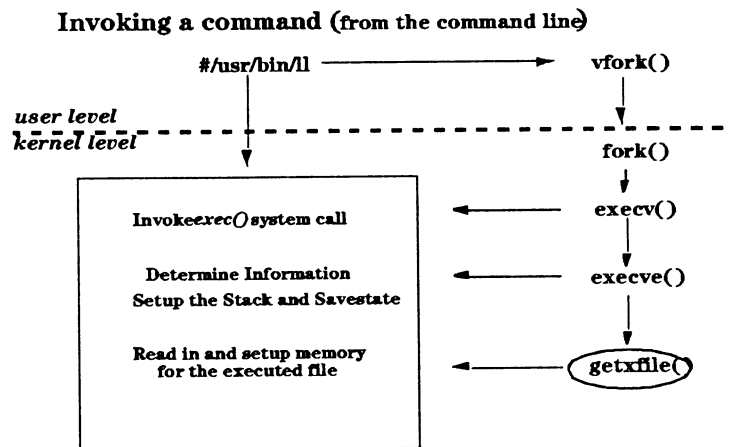
- 1] create a buffer *`vforkinfo()`*, parent and child share
- 2] set state to `VFORK_PARENT` and put parent to sleep
- 3] in save, parent gets its stack saved
- 4] child runs in parent's *`vas`* using parent's *`stack/Uarea`*
- 5] child does *`exec()`* and state goes to `VFORK_CHILDEXIT` and we fix up parents stack now, else child does *`exit`* and we just set `VFORK_CHILDEXIT`.
- 6] parent wakes up and restores its stack in resume if necessary
- 7] parent cleans up *`vforkinfo`* struct

Module 4 — Process Management

Left blank intentionally

Slide: Process Creation: exec()

Process Creation: exec()



a696279

Notes:

Если файл откр. на запись его нельзя
использовать и наоборот — иначе BUSY

Если файл исполнен —
PERMISSION DENIED

(уровень не тот)

— magic совпадает, но этого мало.

Огранич. размер параметров файла,
лимитно 10кб.Н, после сменяется
до 200к.

Slide: Process Creation: `exec()`

`exec()` - `kern_exec.c`

Now that we have discussed how a process is created with `fork()` and `vfork()`, let's review how a process would call `exec()`. When a program is invoked in HP-UX, it is invoked by a call to `fork()` or `vfork()` and then calls `exec()` to begin. An example of this would be running a command like “`/usr/bin/ll`” from your shell. The result of a call to `exec()` will cause the overlaying of an existing process's text with a new copy of an executable file. The entire user context (`text`, `data`, `bss`, `heap`, and `user stack`) are replaced. Only the arguments passed to `exec()` are passed from the old address space to the new address space.

`exec()`, in all its forms, loads a program from an ordinary, executable file onto the current process, replacing the current program. The path or file argument refers to either an executable object file or a file of data for an interpreter. In this case, the file of data is also called a script file.

An executable object file consists of a header (see `a.out(4)`), text segment, and data segment. The data segment contains an initialized portion and an uninitialized portion (`bss`). A successful call to `exec()` does not return because the new program overwrites the calling program.

`execv()` calls `execve()`. `execve()` sets the stage for `exec()` and performs the following steps:

1) Determine Information:

- Makes calls to the vnode-specific routines to extract information like the `uid` and `gid` for this executable.
- Check to see if the executable is a script.
- Copy the filename, argument, and environment pointers.
- `getxfile()` is invoked. We will discuss this in more detail on the next slide.

2) Setup the Stack and Savestate:

- The buffer containing the name of the executable and arguments are copied into a per process record (`pstat_cmd`). Move stack pointer up from `USERSTACK` to make room for argument, environment pointers and strings. Save `argc` and `argv` values in the `save_state` structure.

3) Cleanup:

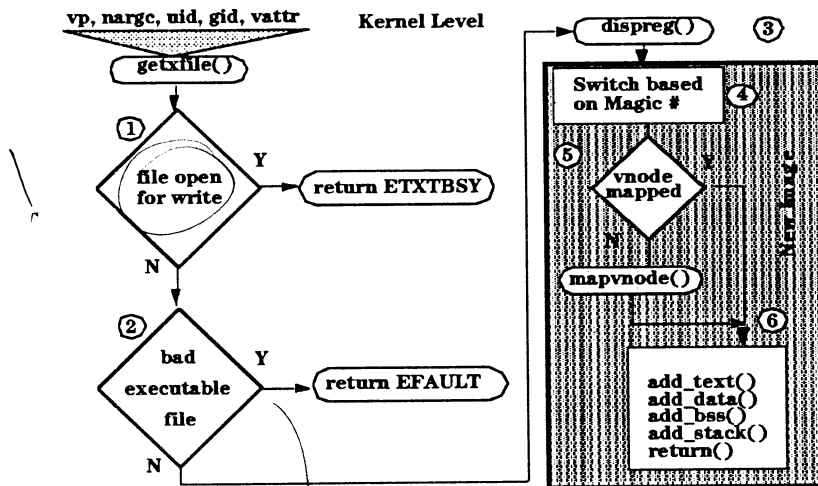
- Release any buffers held.
- Release file `vnode`

B SAM etc организация процесса - PM

Module 4 — Process Management

Slide: `execve: Call to getxfile()`

`execve: call to getxfile()`



Perm. Denied?

a696280

Notes:

Perm. Denied? because vno mapped, no EFAULT he was a good error.

Slide: `execve: Call to getxfile()`

`getxfile()` - `kern_exec.c`

The `execve()` function calls `getxfile()` to perform the real work. When called, `getxfile()` is passed in:

+A pointer to a vnode, representing the file to be executed	<code>vp</code>
+A count of arguments	<code>nargc</code>
+The uid and gid	<code>uid, gid</code>
+Header file information,	<code>ap</code>
+Vnode attributes	<code>vattr</code>

`getxfile()` performs the following steps:

- (1) See if someone is currently writing to this file
- (2) Check Permissions
- (3) Dispose of old preregions - at this point we are committed to the image
- (4) A switch statement is entered based on the type of magic number. We define 3 types for the case statement:

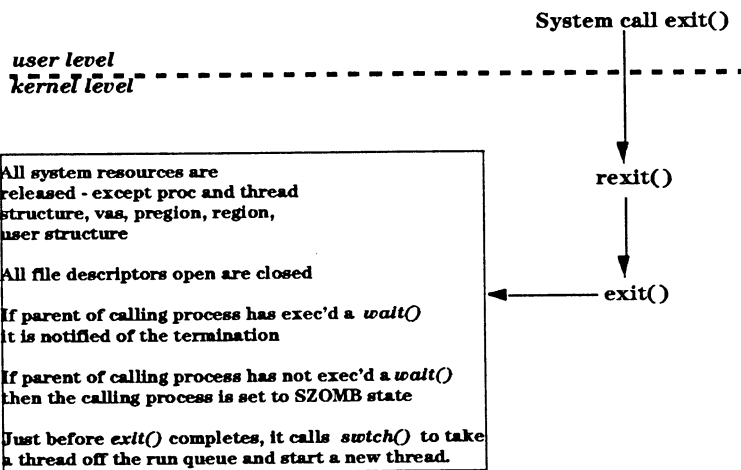
<i>u branch</i>	}	EXEC_MAGIC - Only a data object is created.	#407
<i>8 VP-VV</i>		SHARE_MAGIC - Create text and data (but do not assume file alignment).	#410
<i>memory stack preregion magic</i>		DEMAND_MAGIC - Assume both memory and file are aligned.	#413

For the case of **EXEC_MAGIC**, the function `create_execmagic()` is called to place the entire executable (`text` and `data`) into one region.

- (5) Determine if we plan to `execself()` or `!execself()`. If we do not plan to `execself()`, the count of processes referencing the text vnode is incremented, and `mapvnode()` is called to map the text region into the calling process. If the process does an `execself()`, the prior text and regions are retained. Next, `add_data()` is called to add the data region.
- (6) After the switch has completed, we setup the `bss` by calling `add_bss()`, and the stack by calling `add_stack()`.

Slide: Process Termination: exit()

Process Termination: exit()



a696281

Notes:

Зомби из shella.

```
[ #!/bin/sh  
sleep 30s  
exec sleep 30000s ]
```

Нока ќе соопште отед, едн отед оток то
ед зомби инт, он сиз гмуромат.

Slide: Process Termination: `exit()`

Стороа дугагого, запероче гандноб
`exit()` or `_exit()` — *немеженно!*

There are two entries in the standard C library for exit, `exit()` itself and also `_exit()`. With the introduction of the fast standard IO library in the early 80's it became desirable to be able to register functions to be called automatically when the process exited. The new version of the stdio library introduced buffers that need to be flushed in order for the last outputted data actually to be written. If the programmer called `exit` without first calling `flush` then this data would be lost. Since this would require all existing Unix programs to be rewritten, it was necessary to have a way to do this without expecting the program to perform the flush explicitly. Therefore a new function `atexit()` was introduced that allows the programmer to request that certain functions are called when the program exits.

When `exit()` is called these functions are then executed.

With `_exit()` they are not, the kernel is entered immediately and the real exit procedure is started.

*Ночне fork() —
— обрн и орау мурд
но exit() ода спуче
дугагоа г.е. 2 кону.*

It is vital that programs use the correct form of exiting, for instance if the child of a `vfork` were to call `exit()` then it would result in all the stdio buffers of the parent being flushed and closed, and the parent would probably core dump the next time any IO was performed.

`exit()` - *kern_exit.c*

`exit()` may be called by a process upon completion, or the kernel may have made the call on behalf of the process because of a problem. All file descriptors open in the calling process are closed. If the parent process of the calling process is executing a `wait()`, `wait3()`, or `waitpid()`, it is notified of the calling process's termination. If the parent of the calling process is not executing a `wait()`, `wait3()`, or `waitpid()`, and does not have `SIGCLD` (death of a child) signal set to `SIG_IGN` (ignore signal), the calling process is transformed into a zombie process; this is so the parent process can later wait for it. The parent process ID is set to 1 for all of the calling process's existing child processes and zombie processes. This means that process 1 (*init*) inherits each of the child processes.

`exit()` passes status to the system and terminates the calling process in the following manner. Included are excerpts of the `exit()` routine located in the *kern_exit.c* source code:

- Remove all of the threads from a multithreaded process so we can terminate normally. This must be the first thing done in `exit()` so subsequent tasks need not worry about 2 threads racing through `exit()`.
- Set the per-process `p_flag` to `SWEXIT` to indicate it is exiting.
- Release memory-mapped semaphores.
- Release virtual memory. If process resulted from a `vfork()`, give resources back to the parent.

Slide: Process Termination: `exit()`

- Destroy any adopted processes.
- Turn all child processes over to the *init* process. Notify *init* it has zombies waiting.
- Unlink the current process from the active list of processes.
- Unlink current thread from active list of threads.
- Proc enters **SZOMB** state. Thread enters **TSZOMB** state.
- Send **SIGCLD** to either the debugger (if the child has been traced), the parent, or *init*, and to tear down this process.
- If `vfork()` created the process, we need to reset `vfork_state` only if `exit()` is right after the `vfork()`. This will allow `resume()` to restore the parent.
- The parent is still sleeping waiting for the child to `exec()` or `exit()`. We can't wake up the parent now because the child is still on the parent's stack. We can't have the parent running on processor before the child is completely switched out.
- We must hold the "thread lock" across the context switch path. The signal path and the *init* process must wait until the last thread is non-**TSRUNPROC**.

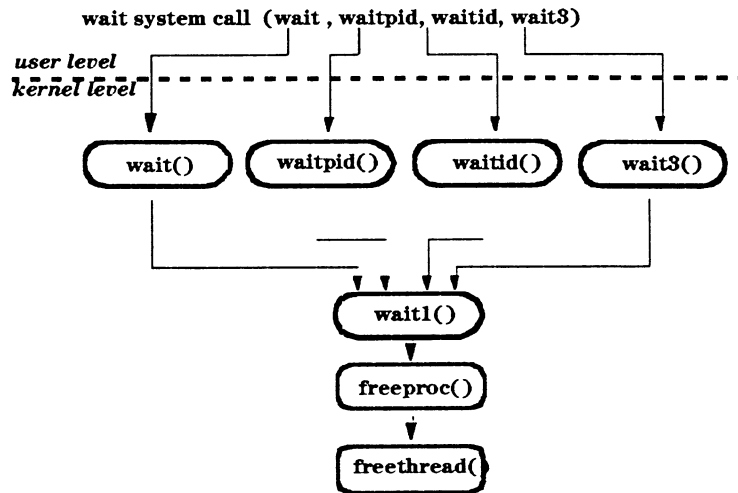
Module 4 — Process Management

Left blank intentionally

Slide: Process Termination: wait()

процесс завершил выполнение.

Process Termination: wait()



a696282

Notes:

Slide: Process Termination: wait()

wait()

The *wait()* function will suspend execution of the calling process until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If status information is available prior to the call to *wait()*, return will be immediate.

wait3()

The *wait3()* function allows the calling process to obtain status information for specified child processes. The options argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the header <sys/wait.h>:

```
# define WNOHANG      1      /* don't hang in wait */
# define WUNTRACED    2      /* tell about stopped, untraced children */
# define _WNOWAIT     4      /* don't register the wait */
```

WNOHANG causes the wait not to hang if there are no stopped or terminated processes, rather returning an error indication in this case (*pid*==0).

WUNTRACED indicates the caller should receive status about untraced children which stop due to signals. If children are stopped and a wait without this option is done, it is as though they were still running; nothing about them is returned.

WNOWAIT causes the wait to not be registered. This means the process that has been waited on can be waited on again with identical results, provided the status of the child does not change in the meantime.

waitid()

The *waitid()* function suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to *waitid()*, *waitid()* returns immediately.

id_type and *id* arguments are used to specify which children *waitid()* will wait for.

If *id_type* is **P_PID**, *waitid()* will wait for the child with a process ID equal to (*pid_t*)*pid*.

If *id_type* is **P_PGID**, *waitid()* will wait for any child with a process group ID equal to (*pid_t*)*pid*.

If *id_type* is **P_ALL**, *waitid()* will wait for any children and ID is ignored.

The *options* argument is used to specify which state changes *waitid()* will wait for. It is formed by OR-ing together one or more of the following flags:

Slide: Process Termination: wait()

WEXITED Wait for processes that have exited.

WSTOPPED Status will be returned for any child which stopped upon receipt of a signal.

WCONTINUED Status will be returned for any child which stopped and has been continued.

WNOHANG Return immediately if there are no children to wait for.

WNOWAIT Keep the process whose status is returned in *infop* in a writable state. This will not affect the state of the process; the process may be waited for again, after this call completes.

waitpid() *B.T.R. que fg, bg*

The *waitpid()* function will behave identically to *wait()*, if the **pid** argument is -1 and the options argument is 0. Otherwise, its behavior will be modified by the values of **pid** and **options** arguments.

The **pid** argument specifies a set of child processes for which a status is requested. The *waitpid()* function will only return the status of a child process from this set:

- + If **pid** is equal to -1, status is requested for any child process. In this respect, *waitpid()* is then equivalent to *wait()*.
- + If **pid** is greater than 0, it specifies the process ID of a single child process for which status is requested.
- + If **pid** is 0, status is requested for any child process whose group ID is equal to that of the calling process.
- + If **pid** is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of **pid**.

wait1()

Search for a terminated child (zombie), finally lay it to rest, and collect its status. Look also for stopped (traced) and continued children, and pass back status from them. The **pid** argument tells which processes are of interest; it is the same as *waitpid()* except the case of 0 has been translated to a specific process group ID.

Some of the steps performed by *wait1()*:

- Call *MTPROC_REAP_LOCK()* to protect the child process list. This prevents multiple threads from reaping simultaneously.

Slide: Process Termination: wait()

- Find and count our step-children who are not our children. If zombies, undebug them and signal their real parents. If stopped, signal the step-parent.

We need to search all processes here. Searching only the active process list is not good enough because zombie processes are no longer on the active process list.

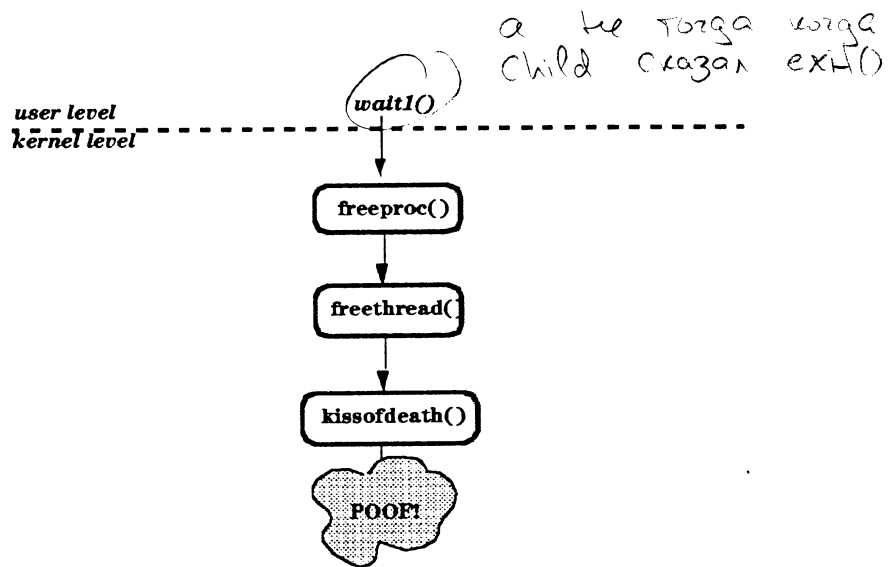
- Grab our own exit lock. We hold this lock while we walk our child list so a zombie can't remove itself from the list after we find it there (e.g., because we're ignoring SIGCLD).
- Search our own child process list.

If an **SZOMB** process is found, then the entry is removed from the proc hash table (*proc_unhash()*) and the thread is deallocated (*thread_free()*).

We then call *freeproc()* to release the process structure of the zombie process.

Slide: Process Termination: freeproc() and freethread()

Process Termination: freeproc() & freethread()



a606283

Notes:

Slide: Process Termination: `freeproc()` and `freethread()`

`freeproc()` - `pm_proc.c`

Is called from `wait1()` to release process structures that zombie processes are no longer using. `freeproc()` performs the following steps:

- Decrement the # of active proc table entries.
- Add the child's *rusage* (sum of stats of the reaped child) structure to the parent.
- Release the *rusage* data structure.
- Release the Process Timers data.
- Release the Virtual Address Space (*vas*).
- Return the proc entry to the free list

`freethread()` - `pm_proc.c`

Is called from `wait1()` to release the thread structure. `freethread()` performs the following steps:

- Decrement the # of active thread table entries.
- Reap the released threads
- Return thread entry to free list

`kissofdeath()` - `vm_vas.c`

Is called from `freethread()` and `freeproc()` to clear the kernel stack and uarea for the process.

Slide: Process Structure Layout

Process Structure Layout

Text - Machine code that is executed

Data - Initialized data; accompanies text

bss - Uninitialized data (Block Started by Symbol)

Heap - Undefined space, used by a process when needed

Private Memory Mapped Files (MMAP) applications can map file data into process virtual address space

User Stack - Store thread variables, subroutines, etc.

UAREA - Per-thread structure; can be paged out

Kernel Stack - Process access to system calls

Shared Libraries - Processes share data from shared library (Text, data, bss)

Shared Memory - Processes share address space

a696284

Notes:

Slide: Process Structure Layout

Now that we understand how a process and thread are created, let us take a look at what defines the process and thread structure. A process is created based on the execution of a program or script. The process and thread will be a image of an program in time. Based on this, a process will consist of the following components:

Text

Machine code that is executed

Data

Initialized data that accompanies the Text.

bss

Un-initialized data

Heap

This is basically undefined space that can be used by the process when needed. The process could utilize this area via a call to *malloc()*.

Private Memory Mapped Files (MMAP)

MMAP files allow applications to map file data into a process's virtual address space. This will be discussed in more detail within the Memory Management Module.

User Stack

User Stack is utilized while a thread is executing in User Mode to store variables, subroutines, etc.

UAREA

The user structure is a per-thread structure containing information that can be paged out. In a thread based kernel, each kernel thread has its own Uarea.

Kernel Stack

The Kernel Stack is used when the process attempts to access system calls, etc.

Slide: Process Structure Layout

Shared Libraries

Utilized by processes to reduce the amount of memory consumed. Processes will share data related to a function from a shared library

Shared Memory

Processes can also share address space. This may be useful for communicating between threads.

There are many ways that the address space of a process can be laid out. For HP-UX 11.00 there is the issue of address space size, either 32-bit, or 64-bit, processes. For 32-bit processes then due to the space limitations it is desirable to reorganize the available 4Gigabyte address range for different types of processes. The type of address map that a process will use is selected based upon it's magic number. There have been magic numbers to control the way that execs were performed in Unix since at least version 6 (before System V) where executables could be loaded read only and therefore shareable, or not. With the introduction of virtual memory then an additional option was added to allow programs to be demand loaded.

With HP-UX the different magic numbers for shared (SHARE_MAGIC) and demand loaded (DEMAND_MAGIC) executable have the same meaning. Then first on the 700 series with HP-UX 9 and then on the 800 series at 10 the other historic magic number (EXEC_MAGIC) was used to allow a new memory map that allowed for larger data areas. Then for 10.20 a patch introduced a fourth magic number (SHMEM_MAGIC) that allows the use of more shared memory was introduced.

For the 64bit environments, there is currently no need to have different memory maps available as the standard one allows up to 4TB for the program text, another 4TB for it's private data and a total of 8TB for shared areas. With current storage technology, it is unlikely that anyone would need to exceed these limits in the near future. Also since these limits are purely imposed by the way that HP-UX currently apportions the 64bit global virtual addresses between the user space and the space registers.

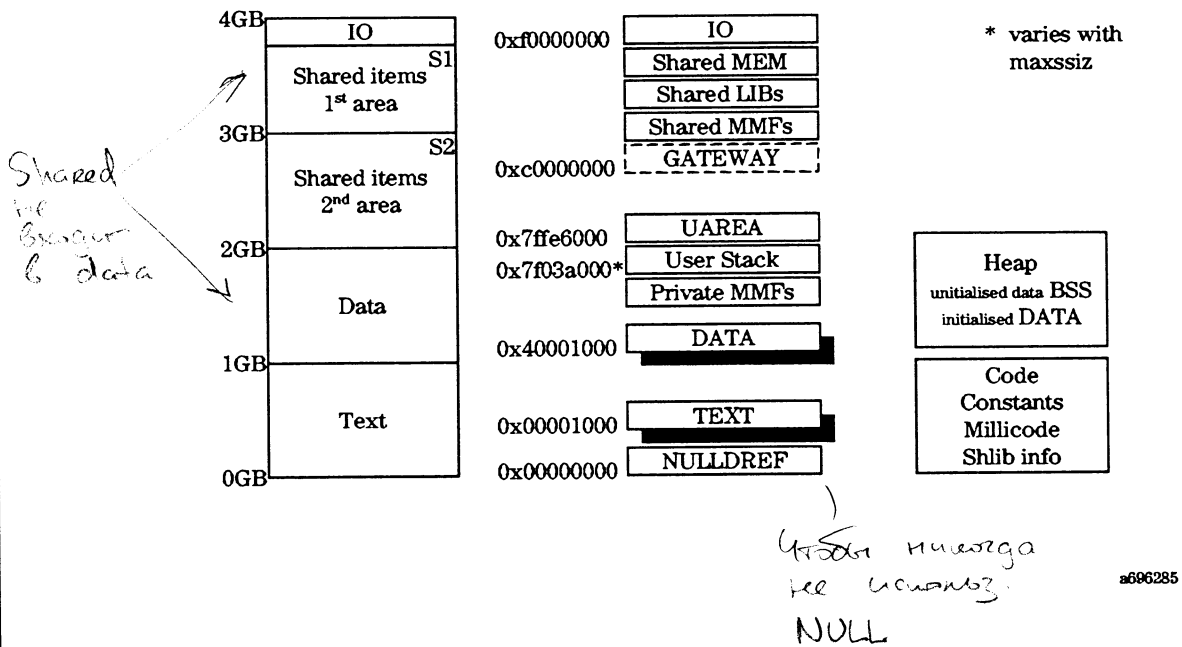
Left blank intentionally

11
12

13
14
15
16

Slide: 32-Bit User Mode Address Space Layout

32-Bit User Mode Address Space Layout



Notes:

No growth. no csex. thread
 64k + 1 safety page

Slide: 32-Bit User Mode Address Space Layout

Prior to 11.0, 32 bits of process address space were available to the user. The old 32-bit address space layout limits applications with only 1 GB of process text, 1 GB of process private data using SHARE_MAGIC or 1.9 GB using EXEC_MAGIC, and 1.75 GB of shared object space with an additional 0.25 GB for user IO. With 64-bits of address space available on PA2.0 systems, users will be able to access 4 TB of text, 4 TB of private data, and 8 TB of shared objects. User IO is 1/16 of 1 TB and is included within the 8 TB of shared object space.

PA RISC 2.0 can operate in two addressing modes: narrow and wide. These two modes are governed by the “W” bit which is located in the Processor Status Word (PSW). When operating in narrow mode, the process is limited to a 32-bit address space. The address space layout for a narrow process is represented by Figure 1:

The actual address used for the various part of the address space can be seen using various tools. Earlier versions of Glance showed the global virtual address space of each part of the processes virtual address space on the M (memory regions screen). With the introduction of 64 bit addresses however there are not sluffing space available on a standard 80 column terminal so as from version C.02.10 these details have not been provided, even for 32bit processes. The X-Windows version, gpm, through is not so constrained and so still displays this information.

Рачунарне
data some 1GB
No user stack
text userout 6
data
- не може да се види
- може да се види.

Module 4 — Process Management

Slide: 32-Bit User Mode Address Space Layout

Later in this module then we will look at the kernels data structures that describe the address space of processes and then using the dump analysis tool “q4” we will be able to look at the address space.

From this view we can see that the address space starts before the text area with an area known as the Null Dereferencing area, this is a page full of 0's starting at address 0 (NULL). This area is to get around a very common programming problem. Many programs make the assumption that if pointer is NULL, that what it points to must be zero. This Null Dereferencing area is optional programmer can turn it off when linking. In which case the first 4K (one pages worth) of address space will be a whole, and any attempt to access it will result in the program being killed with a SIGSEGV. Any program that is intended to be ported widely should do this to make sure that this assumption is not relied upon.

Following on from this, the next area is the TEXT area starting 1 page from the beginning of the virtual address space. This is held shared, and as can be seen, it (and the NULLDR) are in a separate space from the rest of the areas.

The text area is used to hold the machine code instructions for the program and for DEMAND_MAGIC or SHARE_MAGIC executables is held read only. Using the size -v command, the text area is subdivided into a number of sub areas.

The screenshot shows the output of the 'size -v' command for a process named 'ppp'. It displays memory regions and their virtual addresses. The regions are listed in a table format with columns for Type, File Name, P/P, Ref Count, RSS KD, VSS MD, and Virtual Address.

Type	File Name	P/P	Ref Count	RSS KD	VSS MD	Virtual Address
NULLDR	<nullhref>	Shared	87	4	4	0x0000044b.0x00000000
TEXT	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:16658>	Shared	7	77	100	0x0000044b.0x00001000
DATA	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:16658>	Priv	1	80	212	0x00000ba1.0x40001000
HEMWP	<map>	Priv	1	12	16	0x00000ba1.0x7b002000
HEMWP	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7368>	Priv	1	4	4	0x00000ba1.0x7b005000
HEMWP	<map>	Priv	1	4	16	0x00000ba1.0x7b007000
HEMWP	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7316>	Priv	1	24	44	0x00000ba1.0x7b00b000
HEMWP	<map>	Priv	1	40	132	0x00000ba1.0x7b015000
HEMWP	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7120>	Priv	1	12	12	0x00000ba1.0x7b037000
STACK	<stack>	Priv	1	16	20	0x00000ba1.0x7b03a000
UNRES	<uarea>	Priv	1	16	16	0x0000114a.0x7ff05000
LIBTX1	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7120>	Shared	111	56	60	0x00000000.0x00003000
LIBTX1	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7316>	Shared	98	924	1388	0x00000000.0xc0012000
LIBTX1	<reg.vxfs,/usr,/dev/vg00/lvol13,inode:7368>	Shared	114	4	4	0x00000000.0xc0150000

Starting with information about the bindings into the shared libraries. HP-UX also has a set of very low level functions that are implemented using simpler/faster call conventions. These are known as millicode

Slide: 32-Bit User Mode Address Space Layout

and are used to carry out functions that perhaps CISC processor might have built in instructions for (which would then typically be implemented in microcode on the processor. RISC CPUs do not have microcode).

The fact that the text area is held read only and is shared can also make it desirable to store some data in it. This is useful for literal constants (such as the “hello world\n” from the famous program), or declared constants within programs. Normally C’s const functionality is only a compile time feature and does not stop programs changing them through pointers at run time¹. So the text area includes a subarea call

`$LITS` — *константы неперемещаемые в текст (read-only)*

In the next quadrant, again starting 1 page in at 0x40001000, we find the data area. From the programs point of view this area is subdivided into other areas, such as data, BSS and the heap. Some of this can be seen from the size command.

The heap, though will not show up as this gets created dynamically as the program runs. It is frequently referred to as the malloc area.

After the data area the next set of area are memory mapped files. Programs can chose to map files into their address space using the mmap system call. This allows access to the files without the need to make system calls, for some applications this can yield massive performance gains. When mapping files the program can chose to map them either shared or private, as can be seen from the gpm output, these were mapped private. Private memory mapped files are added into the address space of the calling process starting just below the user stack and working downwards. One common feature of HP-UX that makes use of memory mapped files is shared libraries. In this example it can be seen that the program has attached to three shared libraries (the last three items in it address space), Now as well as shared libraries needing text space from the library functions themselves. They also need data space both initialized and uninitialized (BSS) space. The initialized data like the text will come from the shared library itself, but unlike the text area, needs to be held read write and therefore not shared, so it is mapped private. The uninitialized area does not need to come from the file, and so makes use of a feature of mmap that allows mapping anonymous (i.e. not to a file).

For most shared libraries you will find 3 memory mapped file sections, the lib text area, and then a private mapping to the same file for the initialized data area, and an anonymous mapping for the BSS space.

The next major area is the user stack, this will be used to hold the stack of the initial thread within the processes (the only one for normal single threaded programs). For multithreaded applications, the additional threads use private/anonymous memory mapped files for their stacks. Unlike the main stack these are not dynamically sized, rather the call to pthread_create specifies the size of the stack (default 64K) plus a safety space (default 4K).

All further reference to the stack refers to the stack of the initial thread of the process.

1. Fortran is even stranger in this respect, since all parameters are passed by reference (pointers) if a number is passed to a routine and into a variable, and that variable is then changed the original number will be changed, so that the constant 3 could end up with a value of say forty two.

Handwritten notes on the right margin:
- Heap
- malloc area
- shared libraries
- text space
- data space
- BSS space
- user stack
- pthread_create
- size of stack (default 64K) plus safety space (default 4K)

Module 4 — Process Management

Slide: 32-Bit User Mode Address Space Layout

The stack is the top area in the users data quadrant, this is where the local variables of the functions and the function calling information is stored. On HP-UX on the PA systems the stack grows upward (the old Motorola based 2/3/400 series systems grew their stacks downward).

The stack is the one part of the users address space that grows dynamically without the user needing to make system calls.

The start address of the stack will vary depending upon the setting of maxssiz.

Appearing above the users stack, but not actually with the same space¹, their appear the UAREAs. Each thread has it's own UAREA, as this contains information such as the PCB which hold the CPU context whilst the thread is not running and also the kernel stack for the thread used whilst running in system mode. The short pointer address for the all the UAREAs is the same, it is hardcoded at 0x7ffe6000. Where the process has multiple threads each UAREA will be in a separate space in the global memory map.

The top two quadrants are then used predominantly for shared object. The very top one starts with the gateway page and has the IO space at the end. The shared objects will be shared memory and shared memory mapped files. Shared libraries are implemented as memory mapped files.

```
ken@kgcc[ken] size -v /usr/contrib/bin/ppp
```

Subspace	Size	Physical Address	Virtual Address
\$SHLIB_INFOS	10214	36864	4096
\$MILLICODES	1360	47080	14312
\$ILTS	11976	48440	15672
\$CODES	68220	60416	27648
\$UNWIND_STARTS	6816	128640	95872
\$UNWIND_ENDS	11216	135456	102688
\$RECOVER_ENDS	4	136672	103904
\$PFA_COUNTERS	8	139264	1073745920
\$DATAS	4520	139272	1073745928
\$SHORTDATAS	64	143792	1073750448
\$PLTS	1560	143856	1073750512
\$DLTS	8	145416	1073752072
\$GLOBAL\$	16	145424	1073752080
\$SHORTBSS\$	220	0	1073752096
\$BSS\$	150640	0	1073752320
Total	256842		

```
ken@kgcc[ken]
ken@kgcc[ken]
ken@kgcc[ken]
```

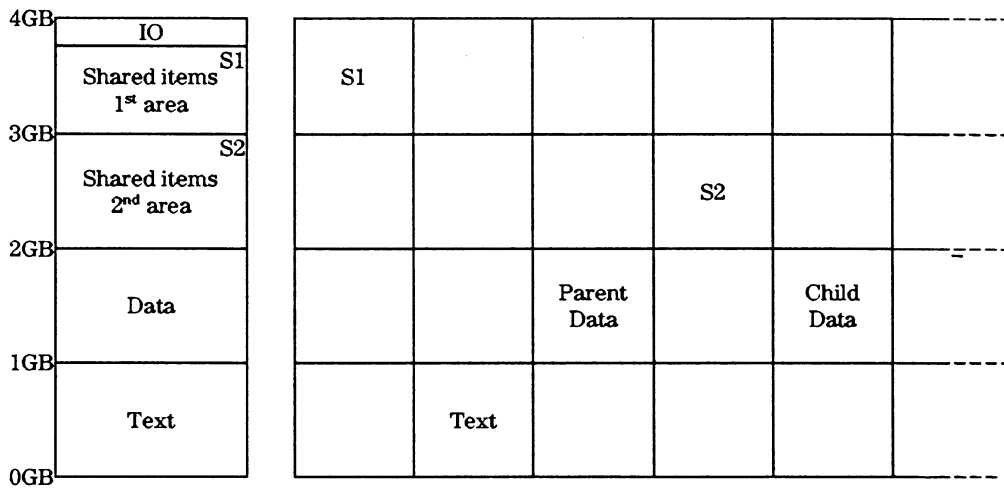
1. This changed with the introduction of thread structures into the kernel. With the earlier process based kernel (10.01 and earlier) then the UAREA was mapped into the same space as the users private data, and used access rights on the page to stop users accessing their own UAREAs.

Module 4 — Process Management

Left blank intentionally

Slide: **SHARE_MAGIC or DEMAND_MAGIC: The Global Perspective**

SHARE_MAGIC or DEMAND_MAGIC: The Global Perspective



a696286

Notes:

Slide: **SHARE_MAGIC or DEMAND_MAGIC: The Global Perspective**

The local address space of processes is mapped into the systems global address space by the space registers. In the global memory map we can see how different areas of address space are shared.

For the text area, the different processes that are running the same program use the same value for SR4, and so use the same space for their text areas.

On the other hand all processes need to have their own private data so that each process uses a separate space for its data quadrant. Whilst in kernel mode, then not only do the processes need their private data but so do the threads, and so UAREAs are now all in separate spaces as well.

The shared area S1 and S2 are then global, and so all processes have common values for SR6 and SR7.

Slide: Limitations of 32-bit Memory Map

Limitations of 32-Bit Memory Map

Soft limits

- maxssiz
- maxdsiz
- maxtsiz
- shmmax

Design limits

- One quadrant (1GB) for text
- Only one quadrant for data
- Small stack space
- Only two shared quadrants for the whole system

a696287

Notes:

Slide: Limitations of 32-bit Memory Map

This memory map whilst it is suitable for the vast majority of applications does impose some limitations that cause problems to some types of applications.

The limitations can be sub-divided into a those that are imposed by the system administrator, and those that are due to the architecture of the memory map.

The kernel parameters `maxtsiz`, `maxdsiz` and `maxssiz` limit the main areas of user programs.

- **maxtsiz**, limits the size of the text area. The default value of 64MB is large enough for all but the largest of applications, equating to approximately 50,000,000 lines of C. Typically applications that exceed this value are written in C++ and make use of templates to generate the code.
- **maxdsiz**, limits the size of the data area of the program, this includes the initialized, un-initialized and heap areas. Again `maxdsiz` defaults to 64MB. Here however it is not uncommon to find applications that need this value increased.
- **maxssiz**, limits the size of the user stack for the initial thread in the process. Here the default value is much lower at just 8MB. The stack is used to hold the local, automatic, variables of programs and the function calling information. Whilst many programmers would argue that 8MB should be sufficient, as large data items ought to be declared either globally or dynamically and hence be in the data area, this value is the most commonly needed to be changed, Fortran programs are perhaps most likely to need this value raising.
- **shmmax**, limits the size of individual areas of shared memory.

The limitations that the memory map gives us are:

- There is only one quadrant available for the program text.

There is room for nearly 1GB of text and currently there are probably not any single programs of that size, so this is not too much of a limitation. Programming around the limitation would not be too difficult either, since a large program would probably be built using many libraries and by using shared libraries that do not occupy this part of the memory map the limit would effectively be raised to about 2.75GB.

- There is only one quadrant available for the private data of the process.

This limitation is much more of a problem than the limitation on text space, and also possibly more difficult to program around. Therefore starting at HP-UX 9, on the 700¹ series and alternative memory map that allows larger data areas was introduced.

1. The 800 series has to wait until HP-UX 10. Most of the applications that use large amount of private data tend to be scientific and technical ones, commercial applications tend to have large amount of data in files or other shared areas.

Slide: Limitations of 32-bit Memory Map

- The stack is limited in size as it starts near the top of the data area.

There is not much that can be done about this, the maximum amount of stack space is only about 79MB. Programs that exceed this value need to be re-coded to place data into the global area so as to reduce stack space usage.

- The kernel stack is limited to 104K.

As with the above limitation on the user stack, only here the result is a system panic, rather than just having the offending process killed with a segmentation violation.

- There is only ~1.75GB of shared space available, for the whole system.

На 600 shared
ресурсов, т.е.
оно в VM занимает

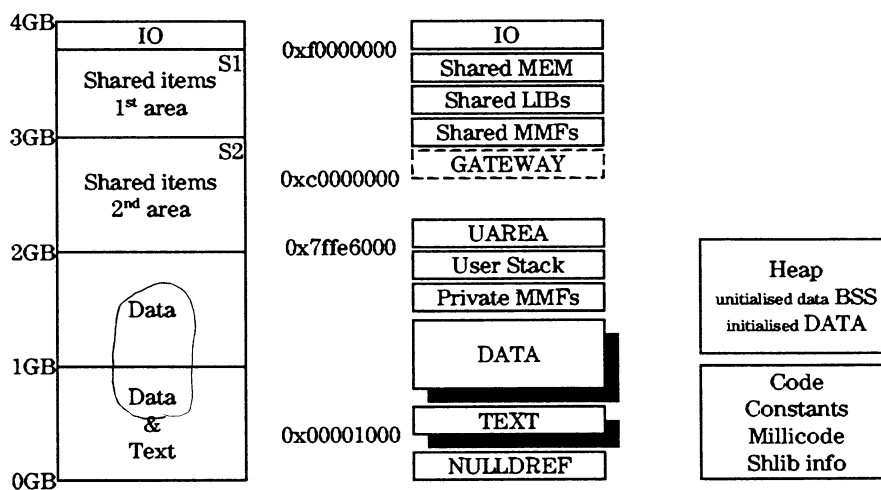
Prior to HP-UX 10, this situation was worse as the third quadrant was used for shared libraries and other memory mapped files, whilst the fourth quadrant was available for use with shared memory, but since this quadrant also houses the gateway page and the IO space, this limited shared memory to a page less than 768MB. In practice it tends to be shared memory that hits this limitation so in the move to HP-UX 10 the top two quadrants were re-organised such that both could carry out both functions. This allowed the limit of ~1.75GB of shared space.

For 10.01 this was not so much of a problem as the systems supported a maximum of 2GB of memory, and the main user of large shared memory areas (databases) wanted them to be memory resident. HP-UX 10.10 allowed memory configurations of up to 3.75GB (as we shall see in the IO module, 1/16th of the physical memory map is set aside for IO devices), and so the limitation on the shared areas became more of a problem. A patch for HP-UX 10.20 then introduced a new memory map to allow for larger amounts of shared memory, up to 2.75GB at this point we are starting to get close to the fundamental limits of 32 bit systems.

Left blank intentionally

Slide: Allowing Larger Data Areas

Allowing Larger Data Areas



a696288

Notes:

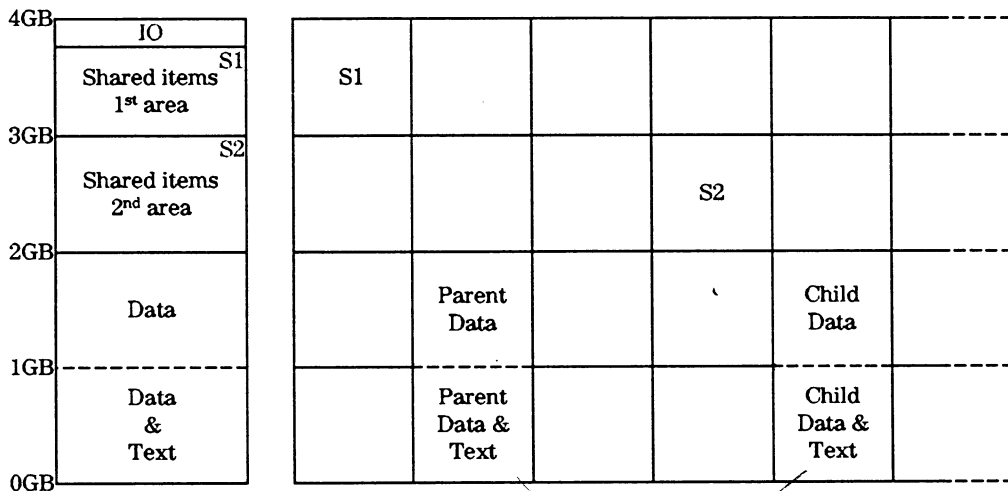
Slide: Allowing Larger Data Areas

In order to allow larger data areas programs can be linked using the `-N` options, to select a different memory map. The executable file is marked with a different magic number so that the kernel knows what type of program it is.

With this new magic number (`EXEC_MAGIC`) the memory map is re-laid out. Since the text area of programs does not change at run time, then it is safe to start the data area immediately on top of the text area. This then allows the data area to make use of the normally wasted virtual address space above the text area in the first quadrant. Given a small text area and no shared libraries the data space can reach about 1.9GB.

Slide: EXEC_MAGIC: The Global Perspective

EXEC_MAGIC: The Global Perspective



copy on write
CHANGES

a696289

Notes:

Slide: EXEC_MAGIC: The Global Perspective

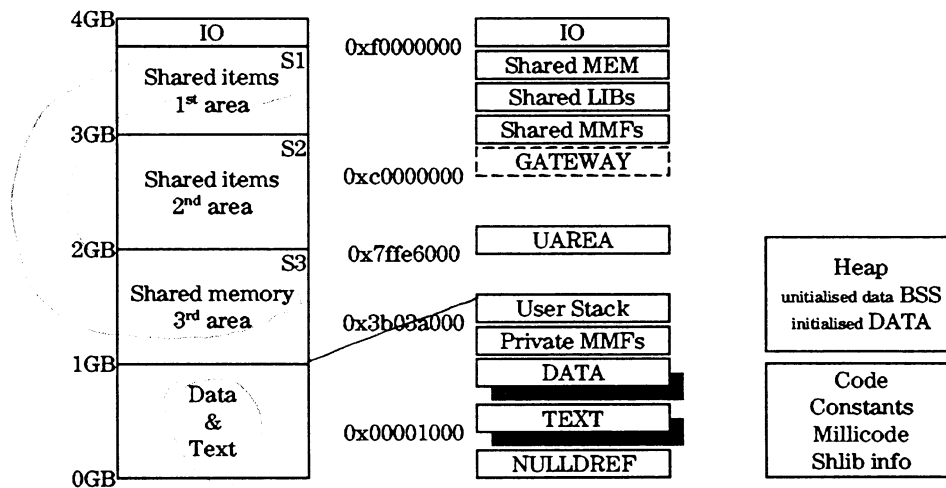
In the EXEC_MAGIC memory map the private data now needs to use the same global quadrant as the text area, and since HP-UX's normal sharing mechanism works on the quadrants¹ it is no longer possible to share the same virtual memory view of the text area between processes, and so the text area is no longer shared² and is not held read-only.

Not sharing the text area has the effect of increasing memory utilization. In an attempt to limit this, starting with HP-UX 10.10 address aliasing is used to allow multiple virtual pages to be mapped to a single physical page³. The use of aliasing is limited to parent and child processes. If a second copy of the program is exec'd then a second copy of the physical pages will be needed.

1. this is how the space registers map between the local and global views
2. this is also the historic meaning of exec_magic.
3. The use of virtually indexed caches tend to limit this, but as long as cache sizes do not exceed 4GB then the part of the virtual address used as the index will be the same for each of the virtual addresses sharing the physical address.

Slide: Allowing More Shared Memory

Allowing More Shared Memory



a696290

Notes:

Slide: Allowing More Shared Memory

Since the type of applications that use large amounts of shared memory rarely use large amounts of private memory they are likely to see large amounts of wasted virtual space in both of the first two quadrants. For some applications it would be desirable to be able to use this unused space to increase the amount of shared memory available.

Shortly after the release of HP-UX 10.20 a patch introduced a third optional memory mapped, that programs request using the SHMEM_MAGIC magic number.

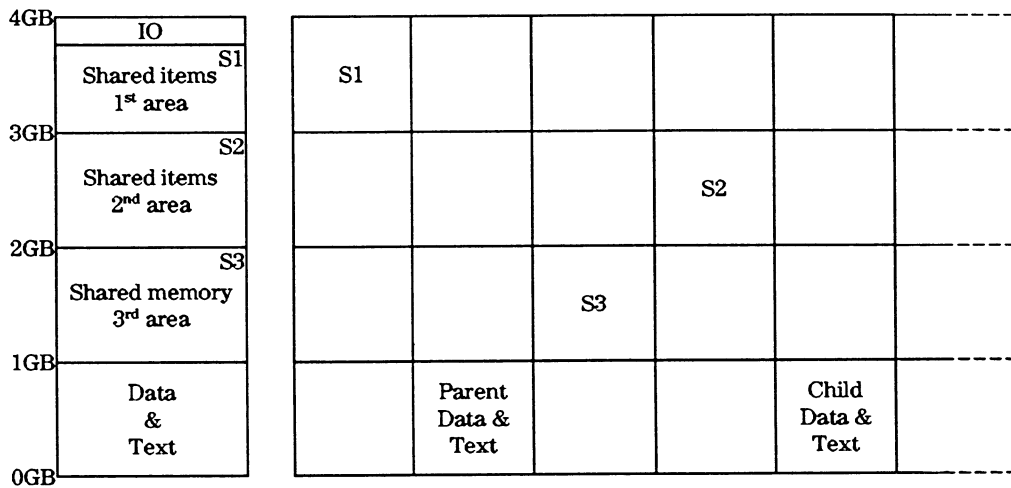
In this arrangement all of the programs own text and data, including the stack, are fitted into the first quadrant, leaving only the UAREA in the second quadrant. Since this is not in the same space as the users data and does not need to be accessible, the second quadrant can now be considered free to be used for other purposes.

This second quadrant then becomes a third area for holding shared information. Unlike the S1 and S2 quadrants S3 is only used to hold shared memory and is only available to programs compiled using the SHMEM_MAGIC option.

Using SHMEM_MAGIC it is then possible to get to ~2.75GB of shared memory.

Slide: SHMEM_MAGIC: The Global Perspective

SHMEM_MAGIC: The Global Perspective



a696291

Notes:

Slide: SHMEM_MAGIC: The Global Perspective

As with the EXEC_MAGIC case the text and data end up in the same quadrant, which precludes sharing the text. Aliasing is used to try to limit the overhead on the RAM utilization, but this again only helps in the parent/child situation. Where the program is exec'd then aliasing can not so easily be used, and so additional physical copies exist of the text area.

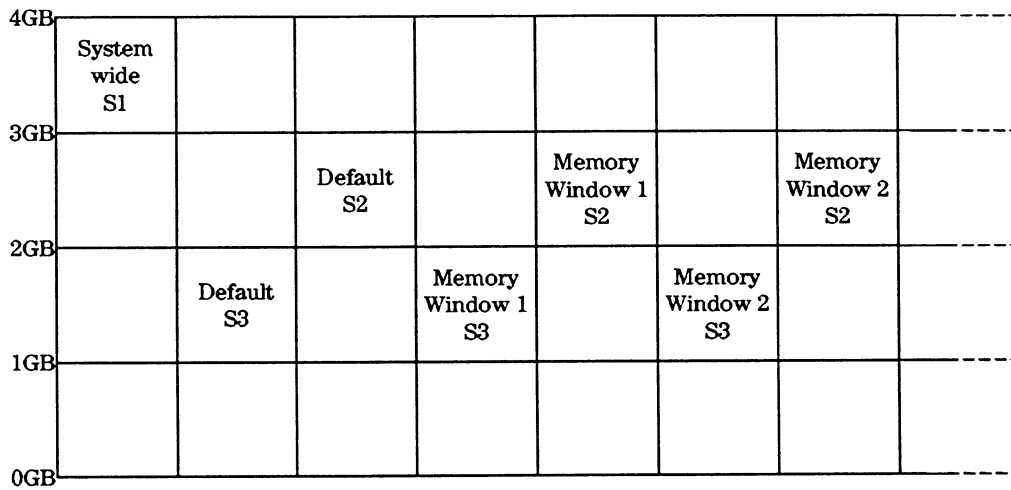
The early SHMEM_MAGIC patch documentation mentioned that since, even with aliasing, there is an increase in the number of virtual pages being used there could be a performance degradation due to an increase in the number of TLB misses. This difference is likely to be small (if measurable at all) however as the TLBs in PA1.1 and PA2.0 systems is small an unlikely to hold a significant of useful translations after a context switch.

This increase in memory usage with SHMEM_MAGIC applications is likely to be more of an issue than it was for EXEC_MAGIC applications. In the case where very large private data sets are required it is unlikely that the number of processes running the program will be high, and even if it is the performance will be limited by the paging performance of the swap disks. With SHMEM_MAGIC though it is very likely that the process count will be very high. Programmers can try to avoid this overhead by getting as much of the application into shared libraries (which are still shared) as possible.

Все что было
в
этом пространстве.

Slide: Shared Memory Windows

Shared Memory Windows



a606202

Notes:

Масок उपयोग

Slide: Shared Memory Windows

SHMEM_MAGIC limitation of 2.75GB was unlikely to be a major issue when the amount of RAM on systems was limited to 3.75GB, but with the introduction of 64bit HP-UX the amount of supportable RAM was increase to the physical capacity of the systems (32GB, although the processors could support up to 960GB). Ideally applications requiring large memory support should be moved into the 64bit world but there is still a considerable number of applications that run in 32bit mode only. Where customers would like to consolidate a number of 32bit applications on to a single large server it would be able to given each application (or group of processes) it's own set of separate shared memory areas.

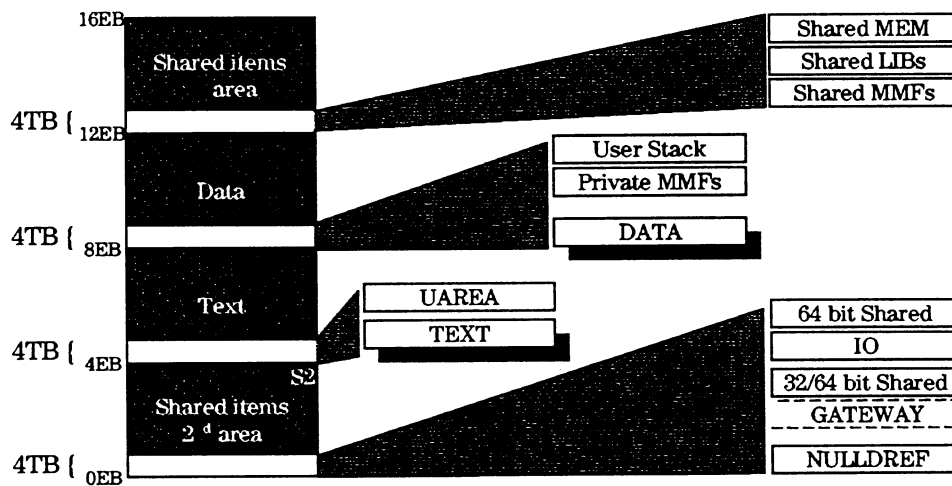
Patch PHKL_13810 introduced this ability as memory windows, using this technique the systems administrator can select the number of memory windows to support through the `max_mem_window` kernel parameter. Applications can then choose which memory window they wish to use using functionality such as `setmemwindow(1M)`, and for each of these memory windows a separate pair of S2 and S3 virtual quadrant will be setup.

The top quadrant, S1, remain global as it contains the gateway page and the IO area, but the other areas of shared memory can then separated for different groups of processes.

Whilst this does not increase the amount of shared memory that a single process can access it does allow for a greater amount to be used across the whole system.

Slide: 64-Bit Kernel Mode and User Mode Address Space Layout

64-Bit Kernel and User Mode Address Layout



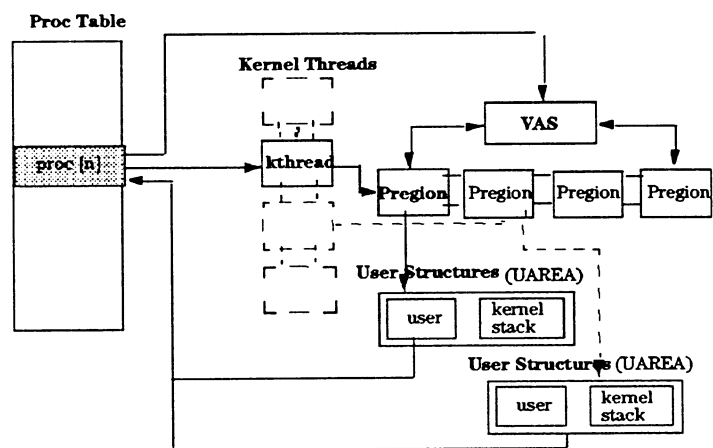
a606293

Notes:

Left blank intentionally

Slide: Process Structure: Virtual Layout Overview

Process Structure: Virtual Layout Overview



a696294

Notes:

Slide: Process Structure: Virtual Layout Overview

Process Table (Proc Table)

Every process has a process structure entry defined in the **Process Table**. This table is allocated at boot time and contains process-specific information necessary for scheduling, such as the process's state, signals, and size. The Process Table and its contents are memory resident (non-swappable). The process entry also contains per-process data that is shared by the kernel threads. As of the 10.30 and 11.00 releases, the kernel is "multi-thread aware": it schedules one or more threads for all processes.

Kernel Threads (Thread Table)

Each kernel thread is represented by a *kthread* structure and a *user* structure. The *kthread* structure is memory resident but the *user* structure is swappable. Both the *kthread* and the *user structure* contain data that corresponds to a given thread. The kernel thread contains the entry for scheduling and its attributes such as priority, states, and cpu usage.

VAS

Virtual Address Space or **VAS** contains all the information about a process's virtual space. It is dynamically allocated based on what is needed and is memory resident.

Pregion

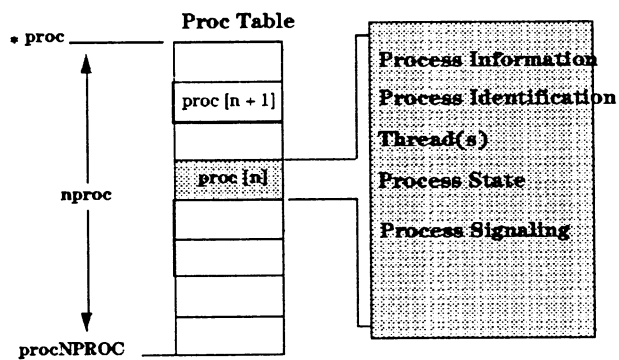
Contains information about the process and thread's address space based on *Text*, *Data*, *Stack*, and **SHMEM**. It provides the page count, protections, and starting address of each.

User Structure (UAREA)

The user structure contains per-thread data which is swappable.

Slide: Process Structure: The Process Table

Process Structure: — The Process Table



a696295

Notes:

Slide: Process Structure: The Process Table

The Process Table is made up of several components of many individual processes. Each process will have a Proc Table entry and within this entry will be Process **Identification**, Process **Thread**, Process **State**, Process **Priority** and Process **Signal** information. The table resides in memory and is not swapped. It must be accessible by the kernel at all times.

The following are some of the fields that make up the entries in the Process Table:

Process Identification

p_pid	unique process id
p_ppid	process id of parent
p_uid	real user id
p_suid	set (effective) user ID >
p_max	max number of open files allowed

Associated Thread Information

p_created_threads	0 for first thread. ++ for add-ons
p_firstthreadp	ptr to first thread in the process
p_lastthreadp	ptr to last thread in the process

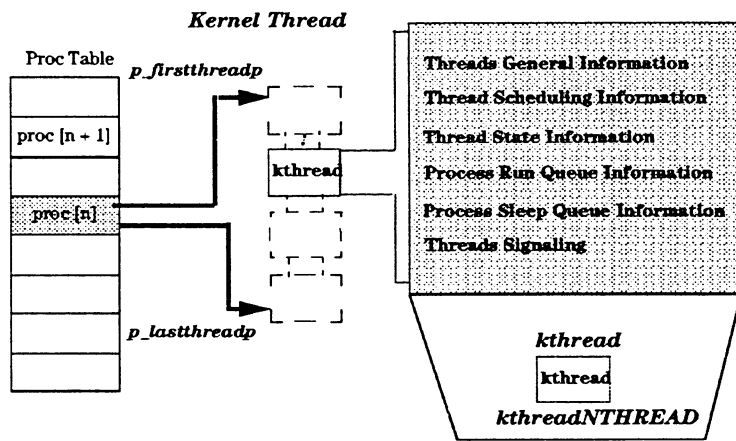
Process State

p_stat	current process state:	<i>состояние</i> <i>с 94</i>
SUNUSED	= 0	unused
SWAIT	= 1	abandoned state)
SIDL	= 2	intermediate state in process creation
SZOMB	= 3	intermediate state in process termination
SINUSE	= 5	proc entry in use AND process is not
p_flag	per process flags (in core, system process, being traced, etc)	
p_flag2	more of the same (see the enum's proc_flag and proc_flag2 in proc_private.h)	

Process Signaling

p_sigignore	signals being ignored
p_sigcatch	signals being caught by user
p_nsig	# of signals recognized by process

Process Management: Threads



a606206

Notes:

Module 4 — Process Management

Slide: Process Management: Threads

Threads

The kernel is now threads-based and contains additional structures to handle this new concept. Remember when we discussed the Process Structure, each process has an entry in the process table. The information stored in the Proc Table is now shared by all kernel threads within the process.

Kthreads

Each kernel thread, also sometimes referred-to as a *light weight process - LWP*, is represented by a *kthread* structure. Each *kthread* structure contains a pointer to the process structure with which it is associated, and in a multi-threaded environment the *kthread* structure has an additional pointer to other threads for the same process. The *kthread* structure will always be memory resident. This is mainly due to the fact *kthread* can allow access to information about a process even if the process has been swapped out. This information may consist of *process id*, *pointer to the process address space*, *file descriptors*, *current directory*, *uid*, and *gid*.

In the threads-based kernel the run and sleep queues consist of *kthreads*. Each *kthread* contains the forward and backward pointers for traversing these queues. Since this occurs, all scheduling related attributes such as priority and states are kept at the threads level.

The following is a brief look at the *kthreads* structures:

General Information

***kt_nextp** next thread in the same process
***kt_prevp** previous thread in the same process
kt_upreg pointer to pregon containing U-area

Scheduling

kt_link forward run/sleep queue link
kt_rlink backward run queue link
kt_procp pointer to proc structure
kt_sched scheduling policy for the thread POSIX, REALTIME

State and Flag Information

kt_flag per-thread flags (see enum *kthread_flag* in *kthread_private.h*)
kt_stat current thread state (running, sleeping... see *kthread_state* in *kthread_private.h*)
kt_cntxt_flags process context flags (is currently running on a processor or signallable)

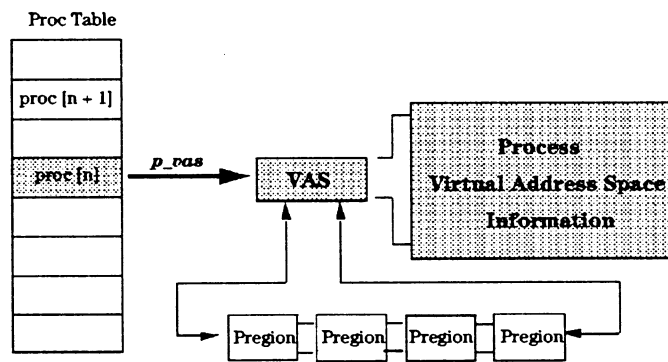
Signal Information

kt_sigsignals signals pending to this thread
kt_sigmask current signal mask

kernel mutex
mutex done
Mutex

Slide: Process Management: Virtual Address Space (VAS)

Process Management: Virtual Address Space (VAS)



a696297

Notes:

Slide: Process Management: Virtual Address Space (VAS)

A process will have a Proc Entry that contains information regarding a given process. Within the Proc Entry will be a pointer (*p_vas*) to the process's virtual address space. The **Virtual Address Space (vas)** maintains a doubly-linked list of preions that belong to a given process and its threads. The VAS is always memory resident and provides information based on the process's virtual address space.

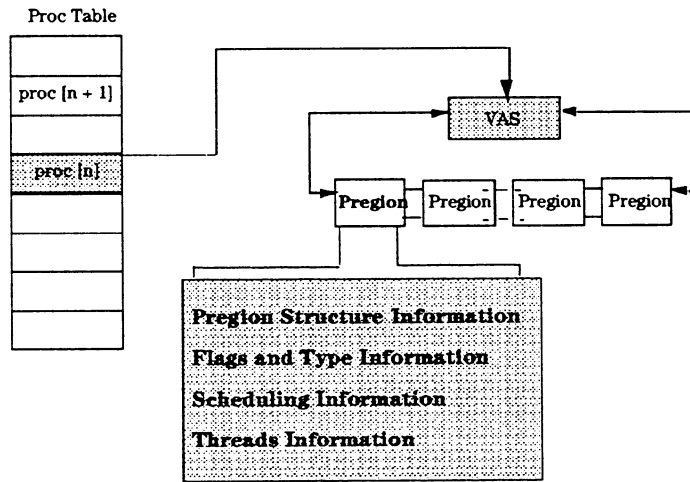
The following are some of the VAS structure's entries:

va_refcnt	Number of pointers to this vas
*va_proc	Pointer to process within the Proc Table
*va_fp	File table entry for a memory-mapped file's pseudo-vas
va_wcount	Count of writable MMFs sharing pseudo-vas
va_rss	Cached approx. of shared resident set size
va_prss	Cached approx. of private RSS (in mem)
va_dprss	Cached approx. of private RSS (on swap)

some mmap
u uoga
h p... ..

For more information see </usr/include/sys/vas.h>

Process Management — pregions



a696298

Notes:

proc() - 320000000000 & physical memory

Slide: Process Management: pregions

The **pregon** represents an active part of the process's **Virtual Address Space (VAS)**. A pregon is memory resident and dynamically allocated as needed. Each process has a number of **pregions** which describe the **regions** that are attached to the process. In this module we will only discuss to the **pregon** level. The Memory Management Module will provide more information regarding regions. **Pregions** are defined based on their type (recorded in the field *p_type*):

```
PT_UAREA      = 1,      /* U area */
PT_TEXT       = 2,      /* Text region. */
PT_DATA       = 3,      /* Data region. */
PT_STACK      = 4,      /* Stack region. */
PT_SHMEM      = 5,      /* Shared memory region. */
PT_NULLDREF   = 6,      /* Null pointer dereference page */
PT_LIBTXT     = 7,      /* shared library text region */
PT_LIBDAT     = 8,      /* shared library data region */
PT_SIGSTACK   = 9,      /* signal stack */
PT_IO         = 10,     /* I/O region */
PT_MMAP       = 11,     /* Memory mapped file */
PT_GRAFLOCKPG = 12,     /* Framebuffer lock page */
```

Most processes will have a “minimum” of 4 pregon defined: **PT_TEXT**, **PT_DATA**, **PT_STACK** and **PT_UAREA**. The following is an overview of the entries that make up a pregon:

Structure Information

p_next	Pointer to the next pregon
p_prev	Pointer to the previous pregon
*p_reg	Pointer to the region.
p_space	Virtual space for region
p_vaddr	Virtual offset for region
p_count	Number of pages mapped by pregon
*p_vas	Pointer to vas we're under

Flags and Type Information

p_flags	memory-locked, writable, text (see enum pregon_flags in pregon.h)
p_type	See list of types on the previous page

Scheduling Information

p_ageremain	Remaining number of pages to age
p_agescan	Index of next scan for vhand's age hand
p_stealscan	Index of next scan for vhand's steal hand
p_bestnice	Best nice value for all processes sharing the region used by this pregon
p_deactsleep	Sleep address for self deactivation

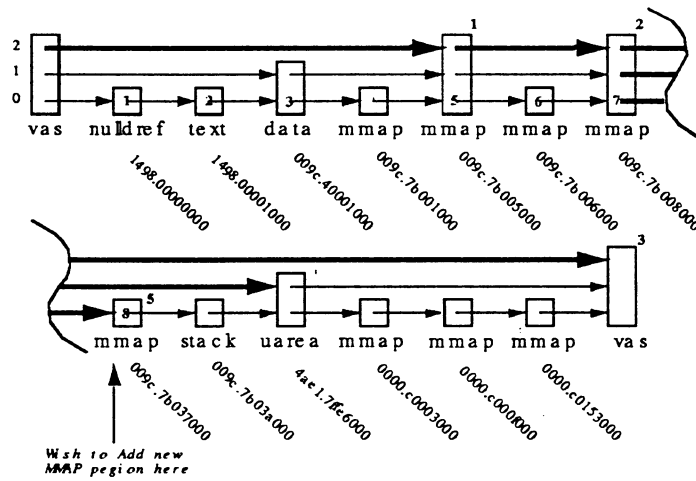
Thread Information

p_tid	Value to identify thread (for uarea pregon)
--------------	---

Slide: pregon Skip Lists

pregion Skip Lists

*gave explanation
now*



a606299

Notes:

*Ha spobna baze...
...*

Slide: pregion Skip Lists

Traversing pregion Skip List

pregion linked lists can get quite large if a process is using many discrete memory-mapped pregions. When this happens, the kernel can spend a lot of time walking the pregion list. We don't want to walk the list from front to back linearly, so we use *skip lists*¹ using four forward links. These are found in the beginning of the **vas** and pregion structures, in the *p_ll* element.

Skip list (struct p_ll): 0x14 bytes

```
struct p_ll {
    struct pregion *lle_next[4]; /* head/next in pregion skip list */
    struct pregion *lle_prev;   /* Last pregion in list */
};
```

Each time a **pregion** is linked into a **vas**' linked list of **pregions**, it is assigned a *level* from **0** to **3** based on a pseudo-random algorithm implemented in the routine. We are four times as likely to be assigned level 0 as level 1; we are four times as likely to be assigned level 1 as level 2; etc. So most of the **pregions** will have an assigned level of 0.

The highest level we're using is stored in the **vas** as *va_hilevel*. This tells us at what level to begin our search.

The slide shows a typical linked list of **pregions**, starting and ending at the **vas**. Let's suppose we need to attach a **PT_MMAP** pregion into the list at address **009c.7b016000**. Pregions are attached in virtual address offset order (the value of *p_space* is ignored for the purposes of ordering the **pregions**).

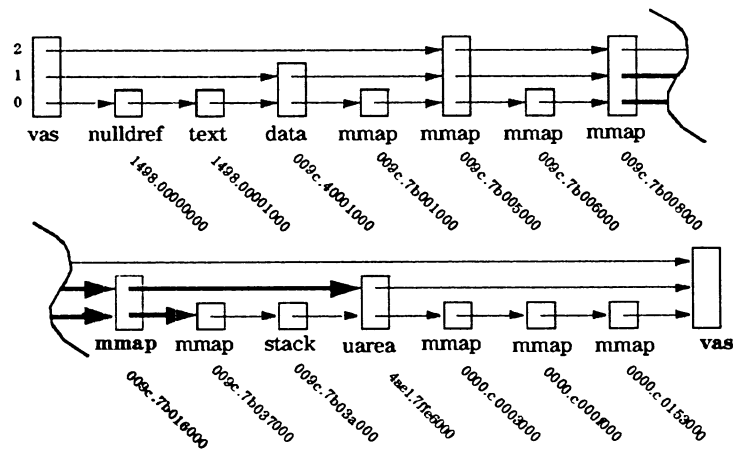
If we were searching linearly, it would take us eight tests to find that we need to insert before the **pregion** at **009c.7b037000**. Using the skip list, we start at our highest level (2 in this example) and search that linked list. When we find an address greater than ours or end at the **vas**, we drop down to the next level, and so on. In this example, we make three tests at level 2, one at level 1, and one at level 0 for a total of five tests.

We store the path taken (last link at each level which is before the position we want) in the **vas** array *va_pslpath[]*. We use this information when linking in a new **pregion**, and when starting a new search (as we frequently look for the same regions, this provides a caching mechanism for our starting point).

1. Skip lists were developed by William Pugh of the University of Maryland. An article he wrote for CACM can be found at <ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps.Z>.

Slide: Adding a New pregon

Adding a New pregon



a606300

Notes:

Slide: Adding a New pregon

Inserting into the Pregon Skip List

Now that we've found where the new **pregon** belongs in the list, we randomly choose a level for it (1 in this example) and simply update the links from that level down to level 0.

Examining skip lists of pregonis using debugging tools

For debugging purposes, we need only follow the level 0 links to find all **pregonis** of a process. Given q4 difficulty in the skiplist array syntax, however, it is often easier to traverse the skiplists in reverse:

```
q4> load proc_t from proc max nproc
loaded 276 proc_t's as an array (stopped by max count)

q4> keep p_pid == 5252
kept 1 of 276 proc_t's, discarded 275

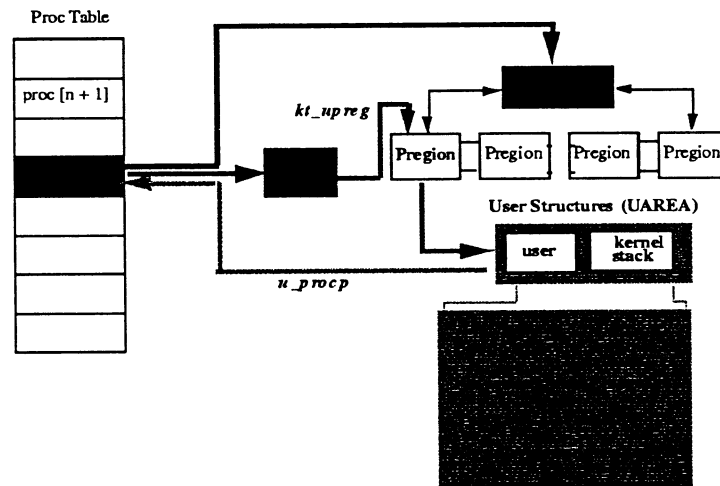
q4> load vas_t from p_vas
loaded 1 vas_t as an array (stopped by max count)

q4> load preg_t from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 17 preg_t's as a linked list (stopped by loop)

q4> print -x p_space p_vaddr p_type
  p_space          p_vaddr  p_type
0xe9a8c00 0x400003fffffce000 PT_UAREA
0xb3ed400 0xc0121000        PT_MMAP
0xb3ed400 0xc0013000        PT_MMAP
0xb3ed400 0xc0003000        PT_MMAP
0x7790800 0x7f7e6000        PT_STACK
0x7790800 0x7f7e3000        PT_MMAP
0x7790800 0x7f7e1000        PT_MMAP
0x7790800 0x7f7cc000        PT_MMAP
0x7790800 0x7f7c2000        PT_MMAP
0x7790800 0x7f7be000        PT_MMAP
0x7790800 0x7f7bd000        PT_MMAP
0x7790800 0x7f7b9000        PT_MMAP
0x7790800 0x7f7b8000        PT_MMAP
0x7790800 0x40001000        PT_DATA
0x3812000 0x1000            PT_TEXT
0x3812000 0                  PT_NULLDREF
0x1 0x1012d0900 0 (<- note that this is the vas at the end of the circular list, not a pregon)
```

Slide: Process Management: User Area (UAREA)

Process Management: User Area (UAREA)



a686301

Notes:

Slide: Process Management: User Area (UAREA)

User Structures (UAREA)

We have discussed the Process Table and how it points to a VAS entry that will point to a group of pre-gions. The pre-gion of a thread contains the pointer to the process's user structure. The user structure consists of the Uarea and Kernel Stack. Kernel threads contain the scheduling entries, so the user structure will maintain the *pcb* (process control block) for each thread. The *pcb* contains the thread's register state which is saved or restored by the kernel during a context switch. A context switch will be discussed later in this module but basically it is the switching of one thread's environment to another. The user structure will also contain the necessary information related to the execution of a system call by a thread. An example of this may be the system call number, savestate pointer, system call argument, system call error status, and the return value.

The kernel thread's UAREA is special in that it resides in the same address space as the process data, heap, private MMFs, and user stack. In a multi-thread environment each kernel thread will be given a separate space for its UAREA. The addressing of the Uarea is the same as the prior process based kernel structure. A kernel thread will reference its own Uarea through "*u*". The definition of "*u*" is (struct user).

You can not index directly into the user structure like you can the proc table. The only way into the Uarea is through the *kt_upreg* pointer in the Thread table.

Some of the significant fields in the struct user are:

User Structure Pointers

*u_procp	pointer to proc structure
*u_kthreadp	pointer to thread structure
*u_sstatep	pointer to a saved state
*u_pfaultssp	most recent savestate

Processes and Protection

The user credentials pointer (for uid, gid, etc) has been moved from the UAREA and is now accessed through the *p_cred()* accessor for the proc structure and the *kt_cred()* accessor for the kthread structure. See comments under the *kt_cred()* field in *kthread.h* for details governing usage.

System Call Fields

u_syscall	system call number (see sys/syscall.h)
u_arg[10]	arguments to current system call
*u_ap	pointer to arglist
u_error	return error code

Module 4 — Process Management

Slide: Process Management: User Area (UAREA)

r_val(n) syscall return values

Signal Management

u_sigonstack signals to take on sigstack
u_oldmask saved mask from before sigpause
u_code ``code'' to trap

UAREA Structure

The UAREA or user structure for the current HP-UX revision can be displayed with the *q4* “fields” command

```
q4> fields -c struct user
struct user {
    struct pcb u_pcb;
    struct proc *u_procp;
    struct kthread *u_kthreadp;
    save_state_t *u_sstatep;
    save_state_t *u_pfaultssp;
    :
    :
```

Module 4 — Process Management

Left blank intentionally

Module 4 — Process Management

LAB: Process Table Lab

These kernel data structures can be examined using the using the crash dump analyzer “q4”. Before q4 can be run on a kernel it must be prepared.

On your lab system

Start glance in one window¹ and then in another

```
# ls -l /stand/vmunix
# q4pxdb /stand/vmunix
# ls -l /stand/vmunix
```

1) What happens to the kernel file?

don't worry it is still a perfectly usable kernel.

Now we are ready to run q4, q4 assumes that you will use job control to suspend it, if you ever want to run commands back at the shell prompt so check that it is setup.

```
# stty susp ^Z
```

It is usually a good idea to run q4 under the control of ied, the input editor, which gives ksh style command history and editing capability to dumb unix program, you can also give ied a named file to use so that it can remember commands across sessions.

```
# ied -h ~/.q4_history q4 /stand/vmunix /dev/mem
```

To look at the process table in q4 firstly we need to load it. The process table is pointed to by a variable proc in the kernel and its size is controlled by the variable nproc.

```
q4> load struct proc from proc max nproc
```

This will then load the whole process table, but many of the slots in the table are currently unused, you can filter down to just the used ones with the command

```
q4> keep p_stat != 0          (or just keep p_stat which means the same thing)
```

The whole table can now be printed out with

```
q4> print -t | more
```

Or often more usefully just selected fields such as

1. if you do not have windows try using tsm on a dumb terminal, or working with your lab partner so that one of you runs glance and the other runs q4. Unfortunately newer version of glance get messed up if they are suspended so you can't easily toggle backwards and forwards between glance and q4 that way.

Module 4 — Process Management

Lab - continued

```
q4> print p_pid p_vas %#x p_flag p_comm (%#x just means print the previous field in hex)
```

Choose a single process, such as your glance process (the last command can be piped through grep) and notes its process id. Then keep just that one process

```
q4> keep p_pid == {your process id} (that is a C style '=' just in case the printers mess it)
```

Once only a single data structure is load fields in that structure can be used as variables in q4, so now you can load the processes vas structure using the field p_vas that points to it.

```
q4> load struct vas from p_vas
```

Using the print -t command the whole structure can be seen. Compare the information that you get from here with the data that glance gives looking at itself.

The vas structure acts as the head of the linked list of pregon, in order to implement the skip list the forward linked list pointers are contained in an array, and unfortunately q4 crashes when try to follow linked lists in arrays. The backwards pointers however are not held in an array so the list can be loaded using these.

```
q4> load struct pregon from va_ll.lle_prev next p_ll.lle_prev max 1000
      (the next option allows linked lists to be followed,
      if more than one structure is to be load, whether from
      a list or a table then a max needs to be given. For lists
      just use a big value, as q4 will know where to stop)
```

As well as printing out values from the data structures in the kernel q4 can also print out the actual structures as well, this is particularly useful for data structures like the process structure that are not described by header files on the customer system.

```
q4> fields -cv struct proc
```

The only problem with looking at structures in this way is that there are not comment to explain the fields. For most data structures there are header files in the directory /usr/include/sys, such as pregon.h for the pregon structures. Using these comments explain what information is begin printed by

```
q4> print -x p_type p_count %d p_space p_flag p_reg
```

Now look at the pregon using the “M” (memory regions) screen in glance, are there any difference between what glance and q4 tell you?

Module 4 — Process Management

The `p_reg` field is the pointer to the region structure, if you can remember from the introduction module the region structure is the system wide description of an area of virtual address space. If two processes are using the same areas they will point to the same region structure, and therefore have the same values for `p_reg`.

In the labs directory there is a small program `fork1.c`, and a shell script that compiles it to give three versions, `fork1_share_magic`, `fork1_exce_magic` and `fork1_shmem_magic`. Run two copies of each of these, they will each fork giving 12 processes in all. Note there process ids, and then using `q4` reload the process table, and follow each of them though to their preions and check out the address of the text, stack and data areas, and the address of the regions structures for the text areas.

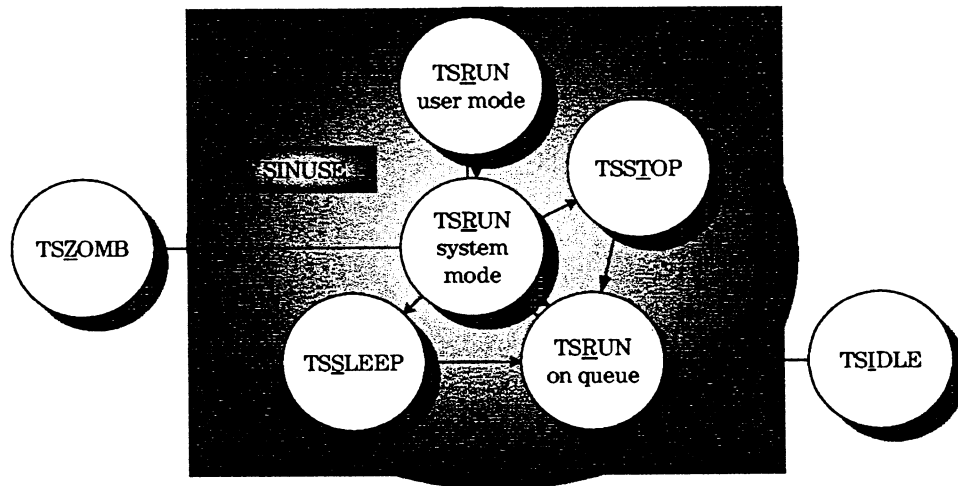
program	process	pid	text area p_space. p_vaddr	data area p_space. p_vaddr	stack area p_space. p_vaddr	text p_reg
share_magic	parent 1					
share_magic	child 1					
share_magic	parent 2					
share_magic	child 2					
exec_magic	parent 1					
exec_magic	child 1					
exec_magic	parent 2					
exec_magic	child 2					
shmem_magic	parent 1					
shmem_magic	child 1					
shmem_magic	parent 2					
shmem_magic	child 2					

Module 4 — Process Management

Left blank intentionally

Slide: Process and Thread States

Process and Thread States



a696302

Notes:

Slide: Process and Thread States

As well as viewing the processes lifecycle from the point of view of the systems calls, it is also useful to view in from the point of view of the different states the process and threads can be in.

Prior to HP-UX 10.10, when the kernel had no thread structures process could be in the states

- **SUNUSED** (p_stat has the value 0)
An empty slot in the process table.
- **SIDL**
A process being created, still inside the fork() system call
- **SRUN**
A process that is either running or runnable.
- **SSLEEP**
A process waiting for some event, such as IO
- **SSTOP**
A process stopped either through the action of a debugger, or through one of the job control signals SIGSTOP, SIGSUSP, SIGTTIN or SIGTTOU.
- **SZOMB**
A process that has completed an exit() system call and is now waiting for it's parent to issue one of the wait system calls.

With the advent of the thread structures into the kernel, the process states have been simplified as processes no longer run. Now the process states SRUN, SSLEEP and SSTOP have been replaced with the single process state of **SINUSE**.

The old states have then been transferred to the threads¹.

The command ps -l is able to display the states, but unfortunately does but know about threads, and tends to get confused when reporting on multithreaded processes, tending to report any process with multiple threads as being in the Run state. The underlined letter of the state in the slide is how ps -l reports the states.

1. An additional state TSSUSP was also added to separate out the two case where the stop state is used. It appears however that this new state is not used and that both threads being debugged and threads suspended through job control continue to use the TSSTOP state.

Slide: Process Flags (p_flag)

Process Flags (p_flag)

substates
(f b b-b ps)

SLOAD
SSYS
STRC
SDEACTSELF
SWEXIT
SPGRP_EXIT_ADJUSTED
SVFORK
SDEACT
SWAITIO

a696303

Notes:

Module 4 — Process Management

Slide: Process Flags (p_flag)

In addition to the states there are also a series of flags used to indicate other aspects of the current status of the process.

SLOAD	0x00000001	in core
SSYS	0x00000002	kernel process such as the swapper or pager
STRC	0x00000008	process is being traced
SWTED_PARENT	0x00000010	parent has waited on process
SDEACTSELF	0x00000020	deactivate process on its next fault
SWEXIT	0x00000080	working on exiting
SPGRP_EXIT_ADJUSTED	0x00000100	group count was adjusted on parent or self exiting
SVFORK	0x00000200	Vfork in process
SWANTS_ALLCPU	0x00200000	batch process hint
SSERIAL	0x00400000	run process serially
SDEACT	0x02000000	process marked for deactivation
SWAITIO	0x04000000	process waiting in syncpageio
SWTED_DEBUGGER	0x10000000	debugger has waited on process
SWCONT	0x20000000	process continued after being stopped

Slide: Thread Flags (kt_flag)

Thread Flags (kt_flag)

TSOMASK
TSOWEUPC
TSSEL
TSFIRSTTHREAD
TSDEACT
TSDEACTSELF
TSFAULTING

a696304

Notes:

Slide: Thread Flags (kt_flag)

For the threads there are several flag fields defined within the kthread structure, the major ones are

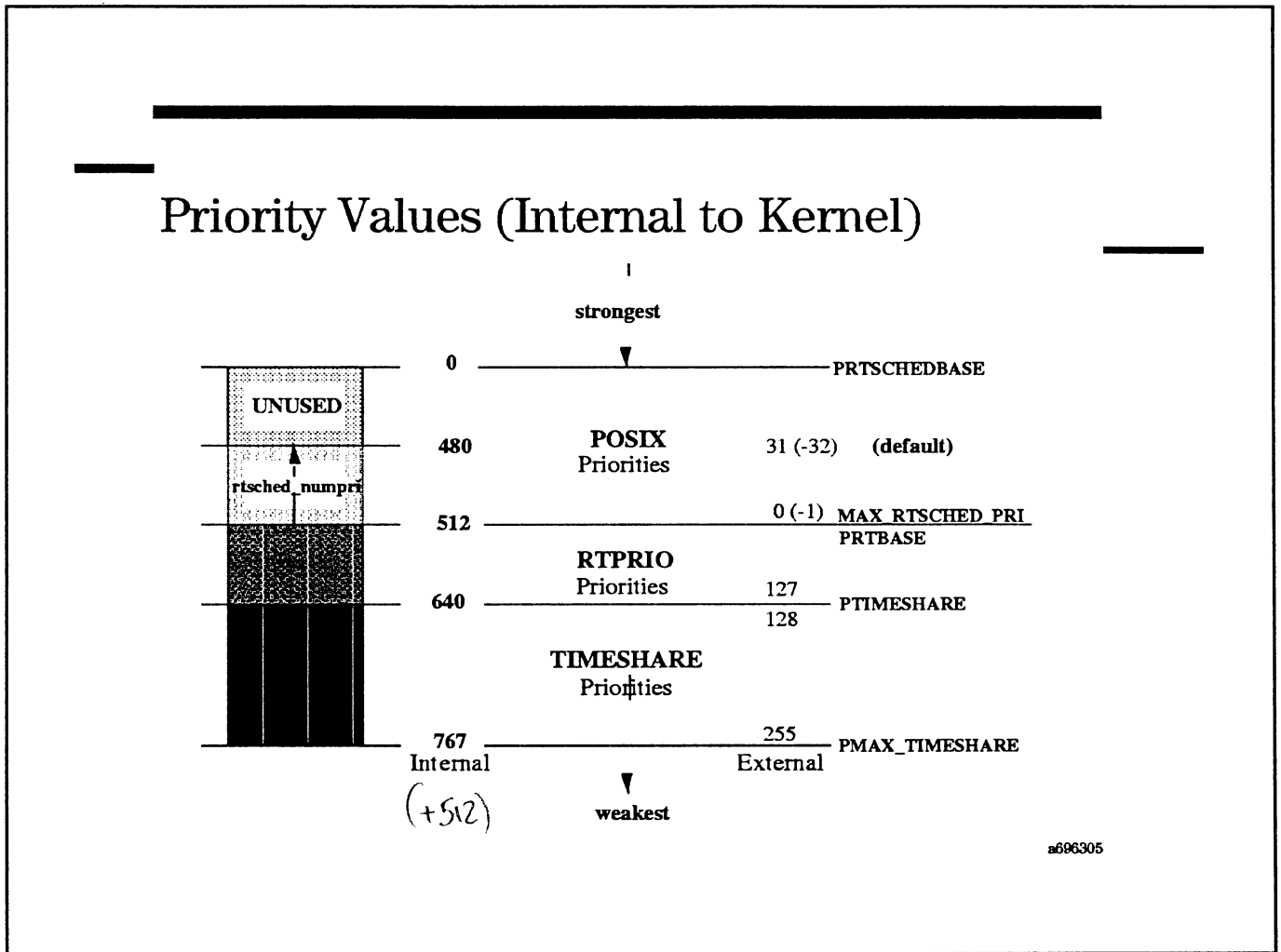
For the field **kt_flag**

TSOMASK	0x1	Restore old mask after taking signal Set by signals.
TSOWEUPC	0x2	Set by clock_int path. Only set in clock_int() when processor in USER_MODE
TSSSEL	0x4	Set by select path when a thread is performing a select call.
TSFIRSTTHREAD	0x8	The initial thread of the process, the one create by fork() rather than by a pthread_create(). This thread uses the main stack.
TSDEACT	0x10	Thread is deactivated. Set by sched() when it deactivates a thread.
TSDEACTSELF	0x20	Thread is to be self deactivated. Set by sched() when it wants to deactivate a thread but that thread is in such a state it cannot be disabled. The thread deactivates itself at the next fault
TSFAULTING	0x40	Thread is faulting. Set by faulting thread when it finds itself in self_deactivate().
TSDETACHED	0x80	Indicates that the thread has been detached using pthread_detach() so that its resources will be freed when it exits (using pthread_exit()).
TSSYS	0x100	A kernel thread, such as those belonging to VxFS.
TSINSIGWAIT	0x200	A thread currently waiting to receive a signal.
TSDEBUG	0x1000	Marks a special kernel thread used when debugging multithreaded programs.

In addition to the main **kt_flag** field, **kt_cntxt_flags** hold some more flags that will be relevant to the course.

TSRUNPROC	0x2	Indicates a thread that is actually running, TSRUN also applies to threads on the run queue.
TSMP_SEMA_BLOCK	0x20	Used when a thread is blocked on a kernel semaphore in multiprocessor systems.
TSSIGABL	0x100	The thread should be awoken in response to signals.

Slide: Priority Values (Internal to Kernel)



Notes:

*Only kernel - up to 767
 some garbage*

*SAM no longer - 2000-2005
 example.*

*Komangur RTPRIO - Realtime
 RTshred*

Module 4 — Process Management

Slide: Priority Values (Internal to Kernel)

The POSIX scheduler was added to HP-UX release 10.00. This scheduler co-exists with other HP-UX execution scheduling policies (timeshare, rtprio, and HP Process Resource Manager). This scheduler provides different execution scheduling policies.

The configurable parameter, *rtsched_numpri*, controls the number of scheduling priorities supported by the POSIX *rtsched* scheduler. The range of valid values is 32 to 512, with the default being 32.

Increasing *rtsched_numpri* provides more scheduling priorities at the cost of increased context switch time, and to a minor degree, increased memory consumption.

The POSIX scheduler provides execution scheduling at the highest priorities. A thread scheduled with it will be “stronger” than one scheduled with *rtprio* 0.

There are now five scheduling policies in HP-UX, all running simultaneously. We will discuss each in more detail but for now, per `/usr/include/sys/sched.h`, they are:

①	SCHED_FIFO	= 0	
	SCHED_RR	= 1	— <i>xbarra oguna wove</i>
	SCHED_HPUX	= 2	(also known as SCHED_TIMESHARE and SCHED_OTHER) <i>with AGE</i>
	SCHED_RR2	= 5	— <i>xbarra ga l uungoro uprop.</i>
	SCHED_RTPRIO	= 6	— <i>realtime</i>
	SCHED_NOAGE	= 8	(SR 5003360446 - Timeshare without priority aging)

There are now four sets of thread priorities: (Internal to External View)

POSIX Standard	512 to 480	0 to 31	(higher # = STRONGER priority per Posix 1003.1b)
Real-time	512 to 640	0 to 127	(lower # = stronger priority)
System, timeshare	640 to 689	128 to 177	
User, timeshare	690 to 767	178 to 255	

Note, threads which have been launched with POSIX realtime priorities will show as *negative priorities*. A ps listing displays “-32” for POSIX priority 31 (strongest by default). The priority of *ttisr*, for example:

```
# ps -ef|grep ttisr
```

```
F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD
1003 S root 12 0 0 -32 20 100454100 0 5e9908 Apr 20 ? 54:29 ttisr
```

Slide: Priority Values (External to Kernel)

Priority Values (External to Kernel)

-ve	Posix Realtime			
0	HP-UX Realtime			
127				
128	128	128	Fast sleep	
	System/ Sleeping	153		
	Timeshared	177	154	Slow sleep
			177	
255	178	Running		
	255			

Handwritten notes:
 - CurHAM for...
 - 128: User signals
 - 153: Fast sleep - because prop. reports to O...
 - 154: Slow sleep - CurHAM...
 - 177: go about syscall

a696306

Notes:

Slide: Priority Values (External to Kernel)

Whilst the threads structures and the other parts of the kernel working using these priority values offset by 512, the userland tools work using these priorities.

Negative priorities indicate Posix Realtime jobs.

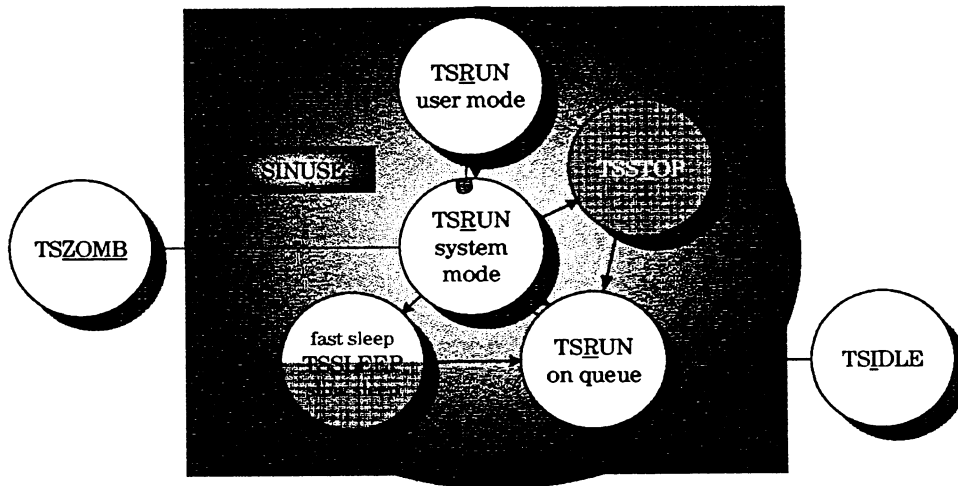
Priorities in the range 0-127 are the HP-UX Realtime priorities.

The range 128-255 is then used by the normal Time-shared threads. This range can be subdivided into user running priorities and sleeping or system priorities. When process are go to sleep the TSSIGABL flag from `kt_cntxt_flags` controls whether the process will wake in response to signals. This flag is normally set when the process sleeps with a priority in the range 154 to 177, and not for higher priorities (lower values).

This rule is not absolute, it can be affected by the nice values of less than 20.

Slide: Signal Handling

Signal Handling



a696307

Notes:

B approach sleep
Linux $\left\{ \begin{array}{l} 1. \text{ delay} \\ 2. \text{ wait} \rightarrow \text{longjump} \\ 3. \text{ catch} \rightarrow \text{na} \text{ reg. wake sleep} \end{array} \right.$

Slide: Signal Handling

The main discussion on signal handling will be in the IPC module, but it is useful to discuss some aspects of the way that signals are handled whilst looking at process management.

In the kernel services module we saw that when returning from kernel mode to user mode processes checked to see whether there were any pending signals and if so processed them. This is where signals get processed. Signals can also cause processes to wake up, so that they will attempt to return from kernel mode to user mode in order to process the newly arrived signal.

If a sleep is to be interrupted by the arrival of signals then the thread needs to release all kernel resources prior to sleeping; otherwise they could remain locked as the code to release after wake up would be bypassed on premature wakeup. The need to release these resources affects the performance of these routines so a decision needs to be made as to whether to allow signals to interrupt the sleep or not.

Typically where the sleep is expected to be very short, the sleep does not allow itself to be interrupted, thus allowing for higher performance. Where the sleep is expected to be more protracted, the TSSIGABL flag is set and the sleep can be interrupted.

There are some notable exceptions to the speed rule, however.

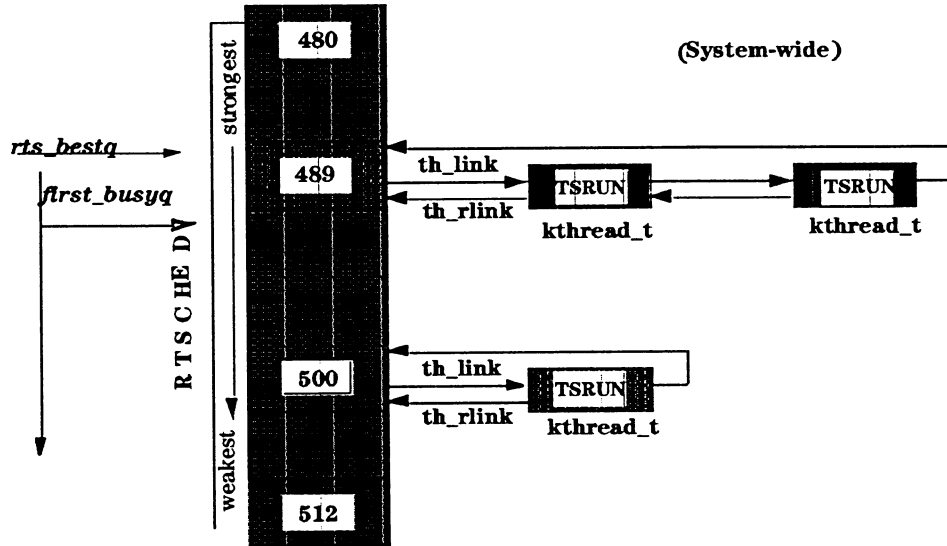
The parent of a vfork can not run as the child is using its address space, so it sleeps with a sufficiently high priority so as not to set TSSIGABL.

NFS, while usually quick enough to warrant the use of uninterruptible sleeps, can be problematic if the server does not respond.

Slide: RTSCHE D Run Queue

RTSCHE D Run Queue

rtsched_info.rts_qs[q]



a696308

Notes:

Slide: RTSCHEM Run Queue

POSIX real-time Scheduler (RTSCHEM)

rtsched provides a POSIX real-time priority scheme. All POSIX real-time priority threads are of greater scheduling importance than threads with HP-UX real-time or HP-UX timeshare priority. All HP-UX real-time priority threads are of greater scheduling importance than HP-UX timeshare priority threads, but are of lesser importance than POSIX real-time threads. Neither POSIX nor HP-UX real-time threads are subject to degradation.

rtsched(1) - execute process with real-time priority

```
rtsched -s scheduler -p priority command [arguments]
rtsched [ -s scheduler ] -p priority -P pid
```

rtsched executes command with POSIX or HP-UX real-time priority, or changes the real-time priority of currently executing process *pid*.

The following is a brief overview of the scheduling policies for the *POSIX* real-time scheduler:

SCHED_FIFO

First in-first out (FIFO) scheduling policy.

The priority range for this policy contains at least 32 priorities.

Threads scheduled under this policy are chosen from a thread list that is ordered by the time its threads have been in the list without being executed. Generally, the head of the list is the thread that has been in the list the longest time, and the tail is the thread that has been in the list the shortest time.

SCHED_RR

Round-robin scheduling policy, with a per-system time slice (time quantum). The priority range for this policy contains at least 32 priorities. This policy is identical to the **SCHED_FIFO** policy with the additional condition that when the implementation detects that a running process has been executing as a running thread for a time period of length returned by the function *sched_rr_get_interval()*, or longer, the thread becomes the tail of its thread list, and the head of that thread list is removed and made a running thread.

SCHED_RR2

Round-robin scheduling policy, with a per-priority time slice (time quantum). The priority range for this policy contains at least 32 priorities.

Slide: RTSCHEM Run Queue

This policy is identical to the **SCHED_RR** policy except that the round-robin time slice interval returned by *sched_rr_get_interval()* depends upon the priority of the specified thread.

There is currently no supported mechanism to change the timeslice value from the system default. This policy is functionally equivalent to **SCHED_RR**.

Slide: RTSCHEDED Run Queue

RTSCHEDED Data Structure

There is a single **RTSCHEDED** run queue for the system.

The **RTSCHEDED** global structure is defined within the *rtsched_info* structure. The following is an overview of this structure:

Some of the important fields within the *rtsched_info* structure:

```
int     rts_nready;      /* total number of threads on queues */
int     rts_bestq;      /* hint of which queue to find threads */
int     rts_numpri;     /* number of RTSCHEDED priorities*/
int     rts_rr_timeslice; /* global timeslice for SCHED_RR threads */
int     *rts_timeslice; /* round-robin timeslices for each pri
                        * (used by SCHED_RR2 threads */
```

The number of priorities are defined via tunable kernel parameter called “*rtsched_numpri*”. The minimum value supported for “*rtsched_numpri*” is 32. This is maintained as the *#define* **RTSCHEDED_NUMPRI_FLOOR** [32] within *pm_rtsched.h*.

If the tunable *rtsched_numpri* is less than **RTSCHEDED_NUMPRI_FLOOR**, a warning will be printed and *rtsched.rts_numpri* will be set to **RTSCHEDED_NUMPRI_FLOOR**. The maximum value supported is **RTSCHEDED_NUMPRI_CEILING** [512]. If the tunable *rtsched_numpri* is greater than **RTSCHEDED_NUMPRI_CEILING**, a warning will be printed and *rtsched.rts_numpri* will be set to **RTSCHEDED_NUMPRI_CEILING**.

Internal to the kernel, smallest (weakest) **RTSCHEDED** priority:

```
#define PRTSCHEDEDBASE      0
```

Internal to the kernel, maximum on (gen-able) number of **RTSCHEDED** priorities:

```
#define MAX_RTSCHEDED_PRI  512
```

Scheduling Policies

The scheduling policies described are defined in terms of a conceptual model, which contains a set of thread lists. There is one thread list for each priority. Any runnable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (*th_link*) as one end of its order and a tail (*th_rlink*) as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving

Slide: RTSCHEM Run Queue

threads between and within lists).

Each thread will be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *sched_setscheduler()* or *sched_setparam()* functions.

Associated with each policy is a priority range. The priority ranges for each policy can (but need not) overlap the priority ranges of other policies.

When a thread is to be selected to run, the thread that is at the head of the highest priority nonempty thread list is chosen.

RTSCHEM Queue Details

We search the global **RTSCHEM** run queues for the strongest (most deserving) thread to run. It returns the best candidate (as a *kthread_t* *). There is one thread list for each priority. Any runnable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (*th_link*) as one end of its order and a tail (*th_rlink*) as the other. *rtsched_info.rts_qp* points to the strongest **RTSCHEM** queue, and *rtsched_info.rts_bestq* points to the queue to begin the search.

Using the slide on the previous page, we search from *rts_bestq* downwards looking for non-empty run queues. When we find the first non-empty queue, we note its index in the local *first_busyq*. For that queue, we check all the threads, from head to tail. If there is a runnable thread, we update the *rts_bestq* value to the present queue and return a pointer to the thread found to our caller. If *first_busyq* was set, then we update the *rts_bestq* value to it and return the thread at the head of that queue to our caller. If *first_busyq* did not get set in the loop, then we panic. Why?

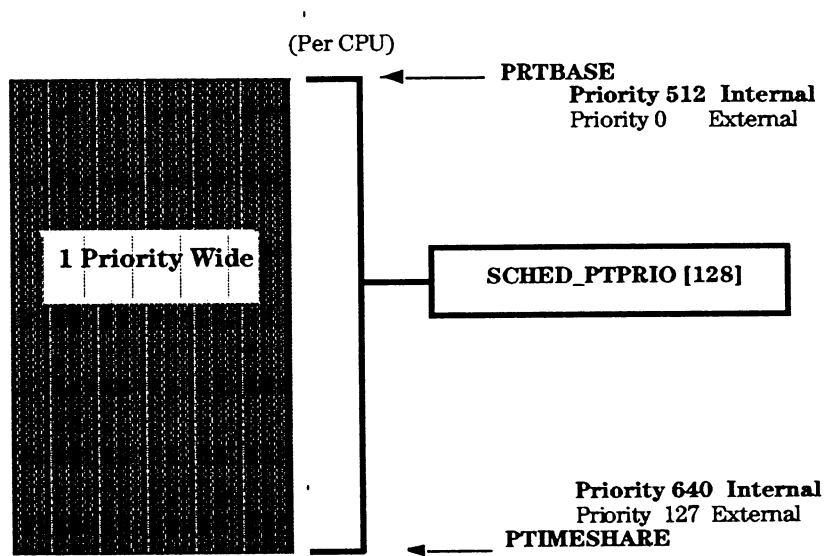
Because, we should only be called if *rtsched_info.rts_nready* is non-zero.

Module 4 — Process Management

Left blank intentionally

Slide: SCHED_RTPRIO Queue

SCHED_RTPRIO Queue



a696309

Notes:

Slide: SCHED_RTPRIO Queue

SCHED_RTPRIO (REAL TIME):

Priorities 0 (*highest realtime priority*) through 127 (*least realtime priority*) are reserved for real-time threads. The real time priority thread will run until it sleeps, exits, or is preempted by a higher priority real time thread. Equal priority threads will be run in a round robin fashion.

The *rtprio(1)* command may be used to give a thread a real-time priority. To use the *rtprio(1)* command a user must belong in the **PRIV_RTPRIO** privilege group or be superuser (root). The priorities of real-time processes are never modified by the system unless explicitly requested by a user (via a command or system call). Also, a real time process will “always” run before a timeshare thread.

The following a few key points regarding a real-time thread:

- Priorities are not adjusted by the kernel
- Priorities may be adjusted by a system call
- Real-Time priority is set in *kt_pri*
- The *p_nice* value has no effect

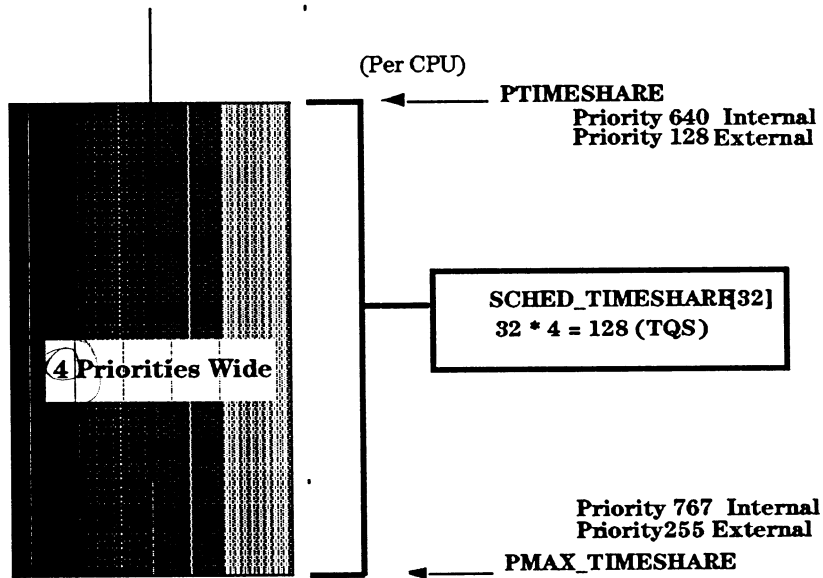
Scheduling Policies: SCHED_RTPRIO

Realtime scheduling policy with nondecaying priorities (like **SCHED_FIFO** and **SCHED_RR**) with a priority range between the POSIX real-time policies and the HP-UX policies.

For threads executing under this policy, the implementation must use only priorities within the range returned by the functions *sched_get_priority_max()* and *sched_get_priority_min()*, when **SCHED_RTPRIO** is provided as the parameter. Note, for the **SCHED_RTPRIO** scheduling policy, smaller numbers represent higher (*stronger*) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering. However, it is guaranteed that the priority range for the **SCHED_OTHER** scheduling policy is properly disjoint from the priority ranges of all of the other scheduling policies described and the strongest priority in the priority range for **SCHED_RTPRIO** is weaker than the weakest priority in the priority ranges for any of the POSIX policies, **SCHED_FIFO**, **SCHED_RR**, and **SCHED_RR2**.

Slide: SCHED_TIMESHARE Queue

SCHED_TIMESHARE Queue



a696310

Notes:

В рамках одного из 4х зрчм
возможно параллельное исполнение

Slide: SCHED_TIMESHARE Queue

SCHED_TIMESHARE

Timeshare threads are grouped into system and user priorities. Priorities 128 through 177 are reserved for system threads and priorities 178 through 255 are reserved user threads. These queues are 4 priorities wide. Thread deactivation will be discussed in more detail in the Memory Management Module. For the timeshare threads, the system picks the highest priority ready thread, and lets it run for a specific period of time (*timeslice*). As the thread is running its priority is adjusted to a lower one. At the end of the time slice, again the highest priority is chosen. Waiting threads gain priority and running threads lose priority in order to favor threads which do I/O and disfavor compute-bound threads.

The following are a few key points regarding a timeshare thread:

- Adjustable priorities

The following shows how the SCHED_TIMESHARE queue groups priorities:

- Grouping of **Time-share priority** thread: range 128-255
- Grouping of **System-level priority** thread: range 128-177
- Grouping of **User-level priority** thread: range 178-255
- Grouping of priority queues that are 1 priority wide
- Grouping of priority queues that are 4 priorities wide
- The process deactivation

Scheduling Policies: SCHED_OTHER (SCHED_HPUX, SCHED_TIMESHARE)

The SCHED_OTHER policy, also known as SCHED_HPUX and SCHED_TIMESHARE, provides a way for applications to indicate, in a portable way, that they no longer need a real-time scheduling policy.

For threads executing under this policy, the implementation can use only priorities within the range returned by the functions *sched_get_priority_max()* and *sched_get_priority_min()* when SCHED_OTHER is provided as the parameter. Note that for the SCHED_OTHER scheduling policy, like SCHED_RTPRIO, smaller numbers represent higher (stronger) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering. However, it is guaranteed that the priority range for the SCHED_OTHER scheduling policy is properly disjoint from the priority ranges of all of the other

Slide: SCHED_TIMESHARE Queue

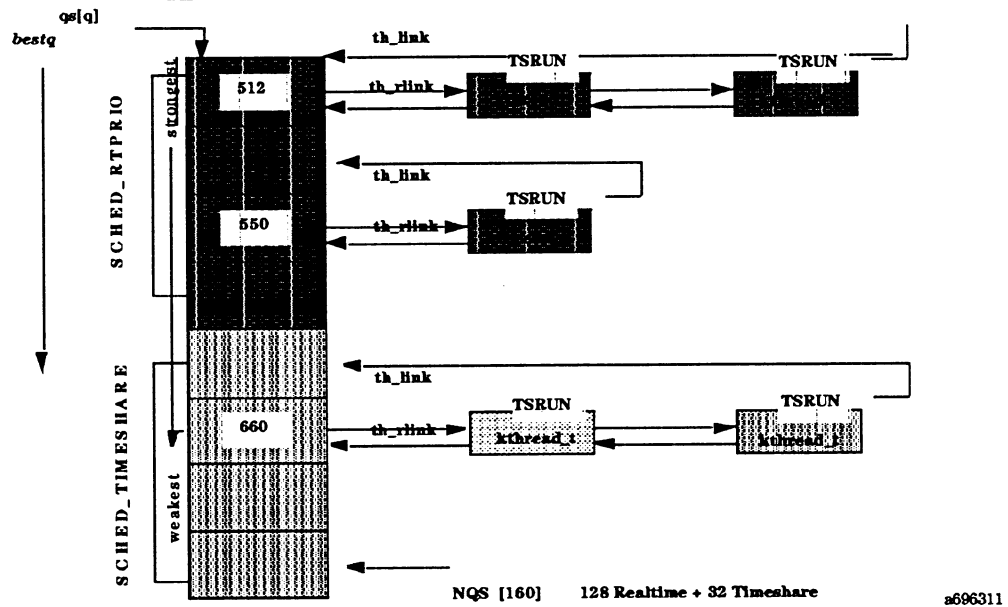
scheduling policies described and the strongest priority in the priority range for **SCHED_OTHER** is weaker than the weakest priority in the priority ranges for any of the other policies, **SCHED_FIFO**, **SCHED_RR**, and **SCHED_RR2**.

{ Note: Why is the **SCHED_TIMESHARE** queue 4 priorities wide? It's a legacy of the VAX-11, which had instructions to manipulate 32-bit fields. $128 \text{ priorities} / 32\text{-bits} \Rightarrow 4 \text{ priorities} / \text{bit}$.

Left blank intentionally

Slide: SCHED_RTPRIO and SCHED_TIMESHARE Queue Details

SCHED_RTPRIO and SCHED_TIMESHARE Queue Details



Notes:

Slide: SCHED_RTPRIO and SCHED_TIMESHARE Queue Details

SCHED_RTPRIO and SCHED_TIMESHARE Queue Details

The **SCHED_RTPRIO** and **SCHED_TIMESHARE** queue is searched with the same technique as the **RTSCHED** queue. For the current spu, we will find the most deserving thread to run. The search starts at *bestq*, which is an index into the table of run queues. There is one thread list for each priority. Any runnable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (*th_link*) as one end of its order and a tail (*th_rlink*) as the other.

The *mp_rq* structure constructs the run queues by linking threads together. The structure *qs* is an array of pointer pairs that act as a doubly linked list of threads. Each entry in *qs[]* represents different priority queue. It's sized by **NQS**, which is 160:

```
(128 SCHED_RTPRIO priorities)      / (1 wide for RTPRIO)    128
+(128 SCHED_TIMESHARE priorities) / (4 wide for TIMESHARE)  32
-----
160
```

The *qs[.th_link* pointer points to the first thread in the queue and the *qs[.th_rlink* pointer points to the tail.

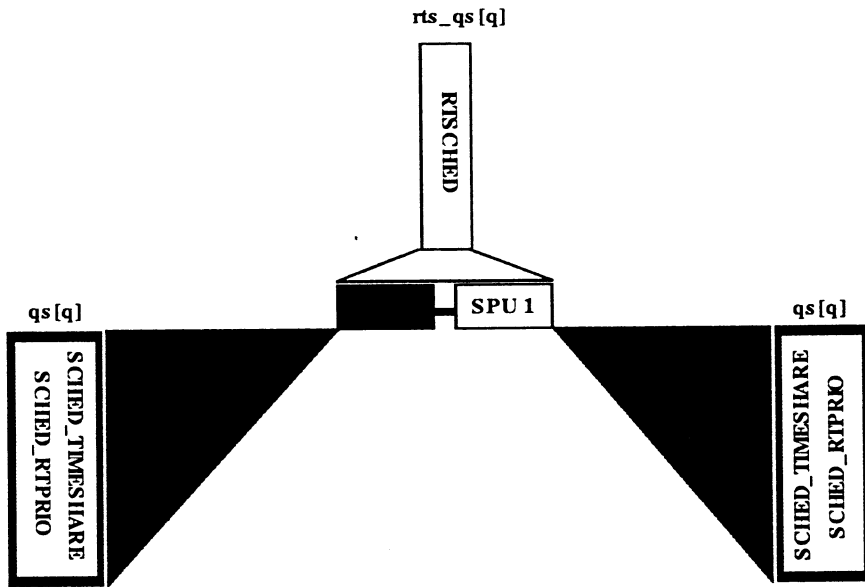
Run Queue Information (*mp_rq*):

```
int      bestq          /* Index into table of run queues */
int      neavg_on_rq    /* 256X avg of neff_on_rq */
int      nready_free    /* active, not locked to any spu */
int      nready_free_alpha /* active, not locked, needs alpha */
int      nready_locked  /* active, locked to this spu */
int      nready_locked_alpha /* active, locked, needs alpha */
int      asema_ignored  /* alpha related. see pm_policy.c */
struct   _horse *nexthorse /* next horse in the carousel (FSS) */
lock_t   *run_queue_lock /* lock for this processor's run Qs */
struct   mp_threadhd qs[NQS]
         struct kthread th_link/* linked list of running threads */
         struct kthread th_rlink
```

Note: The individual priority queues are held in FIFO order, even for those greater than 1 priority wide. This can result in a thread at priority 663 being scheduled before one at priority 660. Exception: If a thread is preempted, it is inserted at the head of the list.

Slide: Run Queue Initialization: rqinit()

Run Queue Initialization — rqinit()



a696312

Notes:

Slide: Run Queue Initialization: `rqinit()`

The queue for given thread is based on how the scheduling policy is defined. Before we jump into how the scheduling process handles threads, let's first review how the run queues are utilized and what they are for the system.

`rqinit()` - `pm_swtdh.c`

`rqinit()` - `pm_swtdh.c` is called from `init_main.c` on system start-up to initialize the (doubly-linked) run queues to be empty. We perform the following steps.

For all potential spus' mpinfo entries, get the `mp_rq` and `mp_threadhd` pointers.
Clear the run queue data including

`bestq`, an index into the array of run queue pointers(`qs`) that points to the highest priority non-empty queue.

`newavg_on_rq`, the run queue average for the processor.

`nready_locked` and `nready_free`, whose sums provided the total threads in the processor's run queues.

Set the current itimer value for all run queues. Link the queue header as the only element.

Now set up the queues such that both the `th_link` and `th_rlink` pointers point to the queues themselves.

- Next the **RTSCHED** Global run queue data structures are initialized. This involves the global structure `rtsched_info`:

```
typedef struct rtsched_info {
    int      rts_nready;          /* total number of threads on queues */
    int      rts_bestq;          /* hint of which queue to find threads*/
    int      rts_numpri;         /* number of RTSCHED priorities*/
    int      rts_rr_timeslice;   /* global timeslice for SCHED_RR threads
    int      *rts_timeslice;     /* round-robin timeslices for each pri
                                * (used by SCHED_RR2 threads*/
    struct mp_threadhd *rts_qp;  /* pointer to run queues */
    struct lock *rts_lock;      /* spinlock for the run queues */
} rtsched_info_t;
```

Slide: Run Queue Initialization: `rqinit()`

- The tunable parameter `rtsched_numpri` determines how many `rtsched` run queues exist. The minimum value supported is 32. This is maintained as:

```
#define RTSCHED_NUMPRI_FLOOR.  
  
#define          RTSCHED_NUMPRI_FLOOR    32  
#define          RTSCHED_NUMPRI_CEILING MAX_RTSCHED_PRI  
  
The weakest RTSCHED priority is MAX_RTSCHED_PRI - 1  
The strongest RTSCHED priority is MAX_RTSCHED_PRI -  
rtsched_info.rts_numpri
```

- `malloc()` is called to allocate space for the `RTSCHED` run queues. (`rtsched_numpri` * sizeof (struct `mp_threadhd()`) bytes are required).
- The resulting pointer is stored in `rtsched_info.rts_qp`.
- `rtsched_info.rts_nready` is set to 0.
- `rtsched_info.rts_numpri` is set to `rtsched_numpri`.
- `rtsched_info.rts_bestq` is set to 0.
- Timeslice is checked. Valid values are -1 (!= “no timeslicing”) and positive integers. If it is invalid, it is set to the default, HZ/10. `rtsched_info.rts_rr_timeslice` is set to timeslice, which round-robins with that many clock ticks. For each of the `rtsched_numpri` run queues, the struct `mp_threadhd` header block is linked circularly to itself.

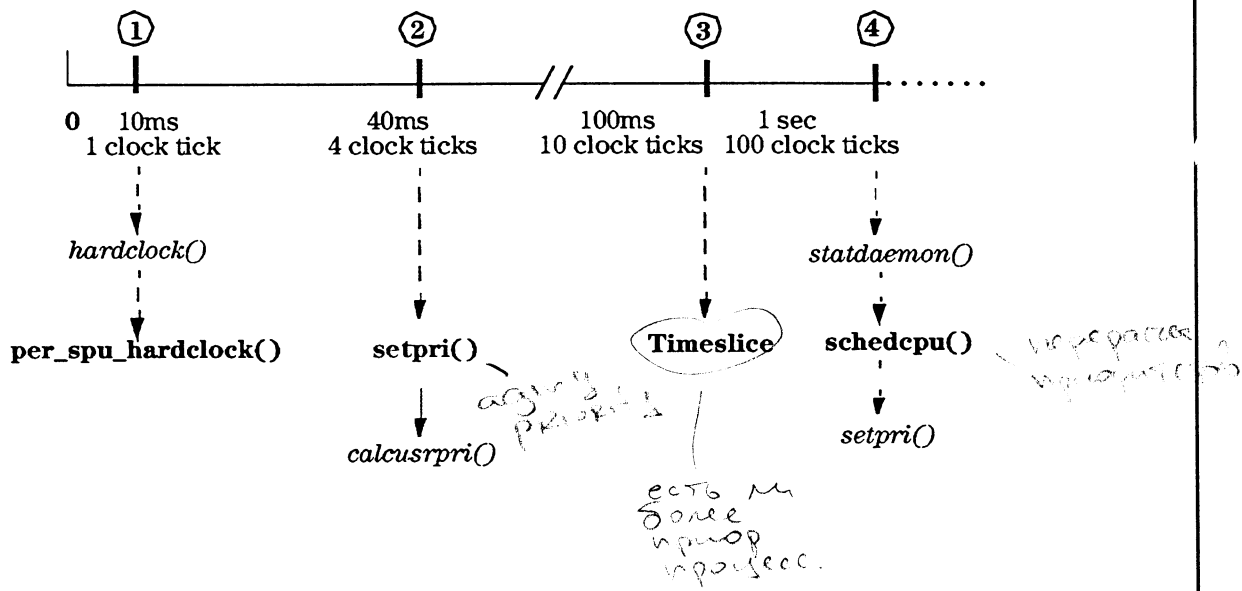
The run queues are balanced between processors on a multiprocessor system by `mp_spu_balance()`, which is called from `schedcpu()` every second.

Module 4 — Process Management

Left blank intentionally

Slide: Thread Scheduling: Timeline

Thread Scheduling — Timeline



a606313

Notes:

Slide: Thread Scheduling: Timeline

We have reviewed how a process and thread is created, how its virtual address space is allocated, and what type of scheduling policy it can run under. Now, let's take a look at how a process and its thread are affected as it passes through time. A thread's priority will be adjusted based on 3 key points in time:

(1) 10 milliseconds

The routine *clock_int()* adjusts the time interval (on the monarch) every clock tick (HZ). The monarch processor will call *hardclock()* to perform general maintenance. *per_spu_hardclock()* will be called for each processor from *hardclock()* to charge the running thread with CPU time accumulated (*kt_cpu*).

(2) 40 milliseconds (4 * 10ms)

EVERY 10ms, *per_spu_hardclock()* bumps *kt_cpu* for the thread it "caught running", and every fourth value ($\% 4 == 0$), it calls *setpri()*. *setpri()* calls *calculuspri()* to adjust the running thread's user priority (*kt_usrpri*).

(3) 100 milliseconds

Timeslice is a configurable kernel parameter that defines the scheduling interval. The *timeslice* interval is the amount of time one thread is allowed to run before the CPU is given to the next thread. Once a timeslice interval has expired a call to *swtch()* will be made. The *swtch()* routine is responsible for enacting a context switch and will be discussed in more detail later in this module. The value of timeslice is specified in units of clock ticks (10 milliseconds).

Timeslice acceptable Values:

Minimum: -1 - Disable round-robin scheduling completely.
Normal: 0 - Use the system default value (currently 10 ticks, or 100 milliseconds).

Maximum: 2 147 483 648 (0x7fffffff) (approximately 8 months)
Default: HZ/10

(4) 1 second

stataemon() loops on the thread list and every 1 second calls *schedcpu()* to recalculate "all"

Slide: Thread Scheduling: Timeline

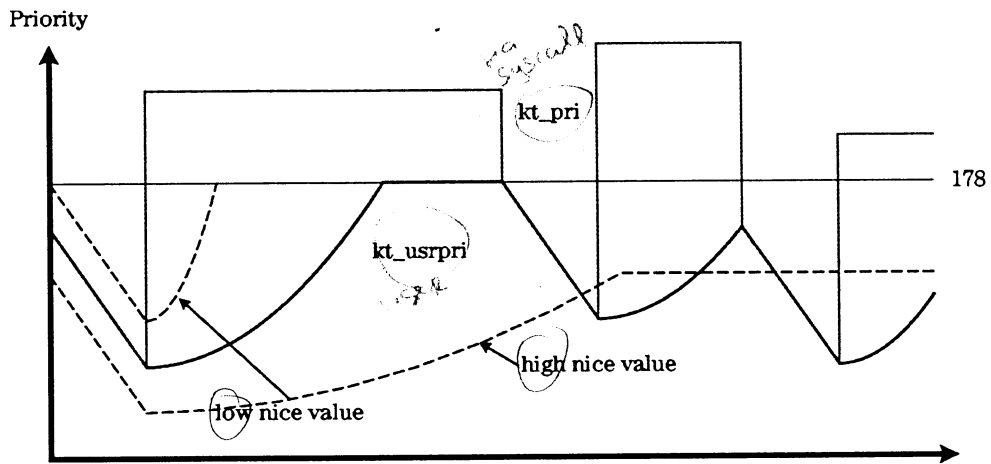
thread priorities and returns the user priority for all threads. The *kt_usrpri* priority will be given to the thread on the next context switch, if in user mode *kt_usrpri* is given right away.

Module 4 — Process Management

Left blank intentionally

Slide: Time-shared Thread Priority Behavior

Time-shared Thread Priority Behavior



a696314

Notes:

Slide: Time-shared Thread Priority Behavior

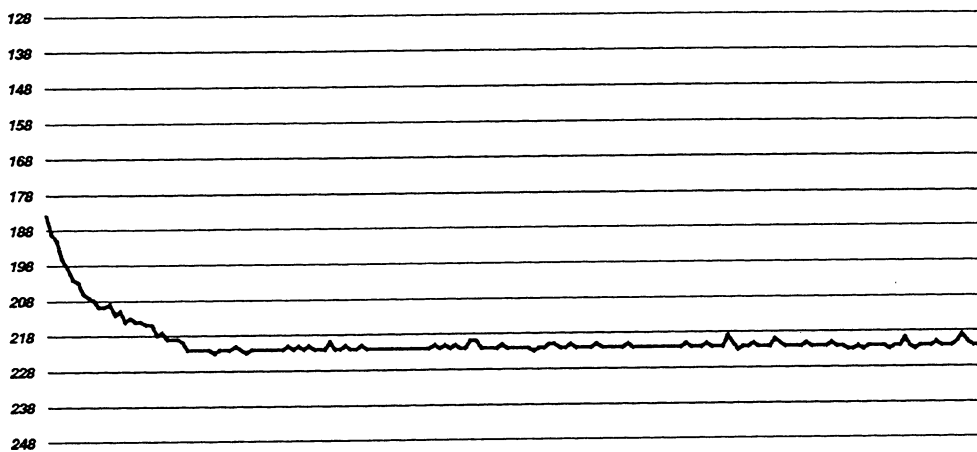
For time-shared processes as they run their priority is degraded, and as the scheduler selects the highest priority runnable process, they are less likely to get scheduled. Once a second statdaemon runs and raises their priority.

The thread structures has two fields for priority `kt_pri` and `kt_usrpri`, as we will see whilst the thread is running they are equal, but when the thread goes to sleep, it is assigned a sleeping priority which is stored in `kt_pri`, and it is the value of `kt_usrpri` that is being re-calculated, on wakeup the value of `kt_usrpri` is copied to `kt_pri`.

This behavior of lowering the priorities of running processes and raise the priority of sleeping processes means that the scheduler tend to favor processes that spend much of their time asleep, i.e. interactive processes, at the expense of CPU intensive programs.

Slide: Priority Degradation for CPU Intensive Applications

Priority Degradation for CPU Intensive Applications



a696315

Notes:

Slide: Priority Degradation for CPU Intensive Applications

In this example a CPU intensive program has been run on a heavily loaded system, its priority degrades over time, over the next few pages we shall look at how this managed.

This example program was very simple: -

```
#!/usr/bin/sh
```

```
while true
```

```
do
```

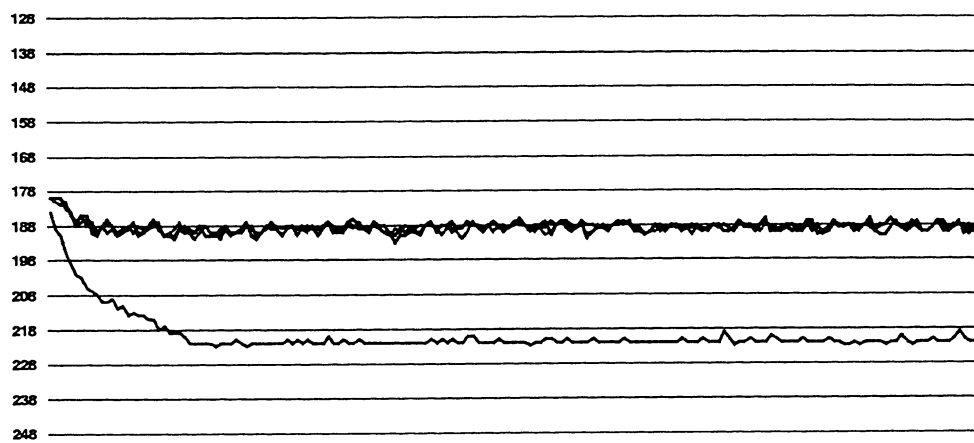
```
  :
```

```
done
```

Now adding some simulated interactive programs to the system we can look at their **kt_usrpri** values.

Slide: Priority Degradation for Interactive Programs

Priority Degradation for Interactive Programs



a696316

Notes:

Slide: Priority Degradation for Interactive Programs

Here we can see that the values for `kt_usrpri` for interaction type programs remains high compared with the priority of the cpu-intensive programs. When the interactive program wakes up it do so with the priority copied from `kt_usrpri`, and since this is that much greater than that of the intensive loop program, it will be scheduled at the next possible context switch.

The simulated interactive program was a modification of the previous example

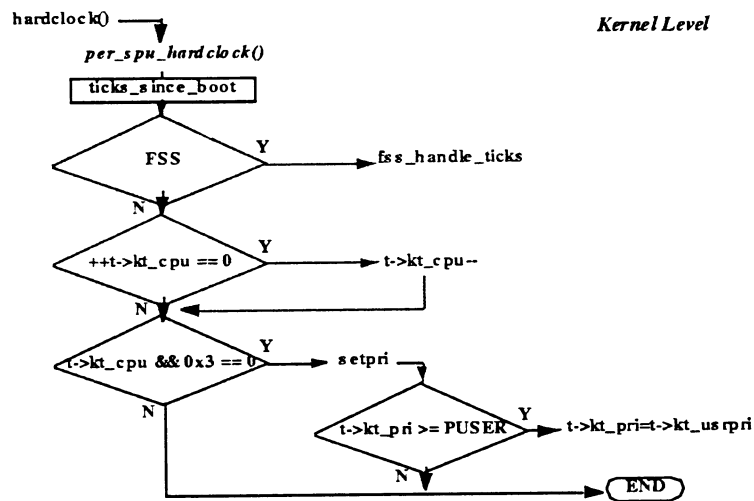
```
#!/usr/bin/sh

integer flag=1
trap "flag=1" 16
trap "flag=0" 17
trap "echo done; exit 0" 2

while true
do
    integer i=0
    while (( flag ))
    do
        i=i+1
    done
    sleep 5
done
```

This shell script can then be alternatively sent SIGUSR1 (16) and SIGUSR2 (17) signals to control whether it runs or sleeps. The program `priplot` in the examples directory can then be used to run any number of these in different patterns. See the later lab.

Thread Scheduling: hardclock()



a606317

Notes:

Slide: Thread Scheduling: `hardclock()`

`hardclock()` - *pm_clockint.c*

Is the monarch-only clock tick handler. It only runs on the monarch processor. Pre-processor `hardclock` activities are called from *per_spu_hardclock()*

`per_spu_hardclock()` - *pm_clockint.c*

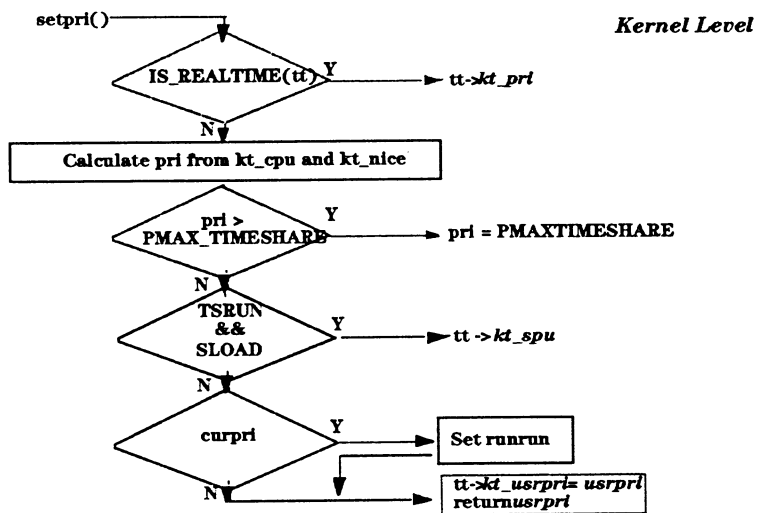
per_spu_hardclock() runs every 10 milliseconds and adjusts the cpu usage estimator (*kt_cpu*) of the current running thread. The priority of a thread gets worse (toward bigger numbers) as it accumulates CPU time. The cpu usage estimator (*kt_cpu*) will be adjusted by 1 on each 10 millisecond pass. After the 4th pass or 4 * 10 milliseconds, *hardclock()* will call *setpri()* to adjust the threads priority.

The following steps correspond to the facing page and review the flow of the *per_spu_hardclock()* routine of *pm_clockint.c* source file:

- We need to determine how many ticks have elapsed since the last time we were here. We look at the global *ticks_since_boot*, which *clock_int()* (and only *clock_int()*) updates. We subtract the last value we saw and that's the value we want.
- Convert ticks to microseconds.
- Increment the clock ticks for the thread. If we wrap around to zero then we will decrement *kt_cpu* by one. This insures we do not give a thread a better priority.
- Check to see if *kt_cpu* has reached a multiple of 4. If we have reached this, we have accumulated 40 milliseconds of time and we call *setpri()* to adjust the user thread priority (*kt_usrpri*).
- After *setpri()* returns we will update the current priority if it is weaker than **PUSER** (178 external, 690 internal).

Slide: Thread Scheduling: setpri()

Thread Scheduling: setpri ()



a696318

Notes:

Slide: Thread Scheduling: `setpri()`

`setpri()` - *pm_policy.c*

`setpri()` is called with a thread as its argument (`tt`) and returns a user priority for that thread. `setpri()` calls `calculuspri()` to get the new user priority. If the new priority calculated is stronger than that of the currently running thread on `tt`'s processor, `setpri()` generates an **MPSCHED** interrupt on that processor. `setpri()` then stores the new user priority in `kt_usrpri` and returns it to its caller.

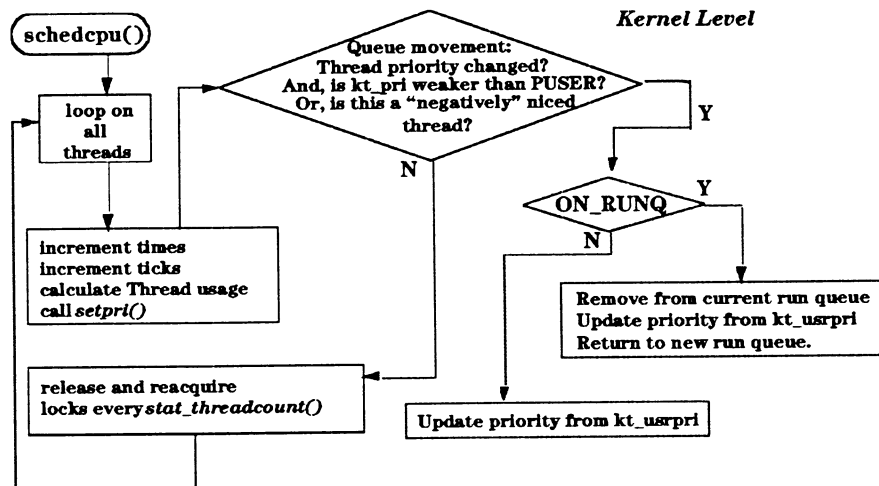
`calculuspri()` - *pm_policy.c*

This is the user priority (`kt_usrpri`) portion of `setpri()`. `calculuspri()` uses the `kt_cpu` and `p_nice(proc)` fields of the thread, `tt`, to decide what its user priority should be. It returns that value. It does not change any fields in `*tt`.

If `tt` is a **RTPRIO** or **RTSCHED** thread, then `kt_usrpri` should be the current value of `kt_pri`.

Slide: Thread Scheduling: schedcpu()

Thread Scheduling: schedcpu ()



a696319

Notes:

Slide: Thread Scheduling: schedcpu()

schedcpu() - pm_policy.c

Runs once a second (from the `statdemon`) to update a thread's scheduling priority. Perform various housekeeping duties on a periodic basis. `schedcpu()` will loop on the thread list.

Algorithm Overview :

- Clear the counts of runnable and paged threads seen this pass.
- If we have more than one active spu(s), and FSS (Fair Shared Scheduler) is on, we call `fss_balance`. Otherwise a call to `mp_spu_balance()` is made.
- We will drop and reacquire our spinlocks every `stat_threadcount`'th iteration.
- We calculate the load scaling factor.
- Acquire the sched and activethread locks.
- Walk the active thread chain, we'll make note if at the end.
- Update the current thread usage information: `kt_prevrecentcycles` and `kt_fractioncpu`
- Update `sqlen` for `TSRUN` thread and acquire the thread lock.
- Calculate a new `kt_cpu` for the current thread.
- `setpri()` will use this data to set the `kt_usrpri`
- If we are starving a thread (he hasn't run lately and we are making him weaker), relent and make his `kt_cpu` a little better than it was.
- Call `setpri()` to update the thread priority and go to `spl7` in the UP case UniProcessor
- If we are to change the thread priority, acquire the run queue lock for the thread. If he is on a run queue, `remrq()` him, update his `kt_pri` and release the run queue lock and `setrq()` him.
- Otherwise, release the run queue lock and just update his `kt_pri` then release the thread lock.
- Restore the `spl` level in the UP case.
- If we are running on an MP system and we have looped threadcount times, temporarily relinquish our spinlocks.

Slide: Thread Scheduling: schedcpu()

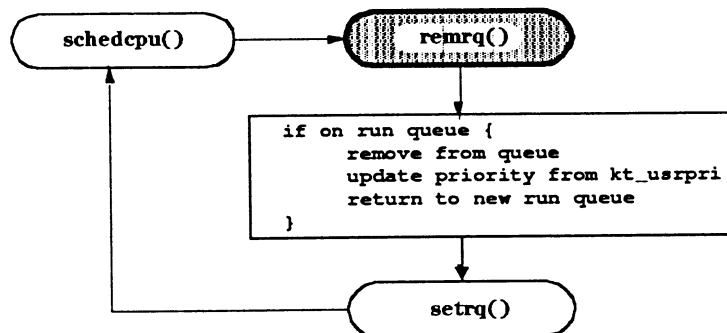
- Now we're done with the active threads.
- Release the activethread and sched_locks in the MP case.
- Update the statistics of runnable threads on run queues and swapped out.
- Call *reset_fss_counters()* to start FSS state afresh for the next second.
- Awaken the swapper into looking around.

Module 4 — Process Management

Left blank intentionally

Slide: Remove a Thread From a Queue: remrq()

Remove A Thread from a Queue: remrq()



a696320

Notes:

Slide: Remove a Thread From a Queue: `remrq()`

`remrq - pm_swch.c`

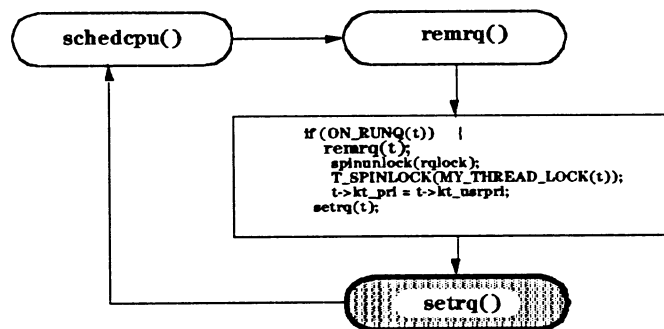
`schedcpu()` will drop to `spl7` and determine if the running threads priority has changed. If so, we will acquire the run queue lock for the thread. At that time we will remove the thread from the run queue via `remrq()`. `remrq()` will update the threads `kt_pri` and release the run queue lock. Then we will call `setrq()`.

`remrq () - pm_swch.c`

Selects the thread to remove from its run queue. Check that we have a valid `kt_link`. Go to `spl7` in the UP case. Find the thread `spu`. Drop the count of threads on run queues. Update `ndeactivated`, `nready_free`, `nready_locked`, in the `mpinfo()` struct, remove the thread from its run queue. Restore the old `spl` level. New changes: if it's **RTSCHED**, update **RTSCHED** counts.

Slide: Add a Thread to a Queue: setrq()

Add a Thread to a Queue: setrq()



a696321

Notes:

Slide: Add a Thread to a Queue: `setrq()`

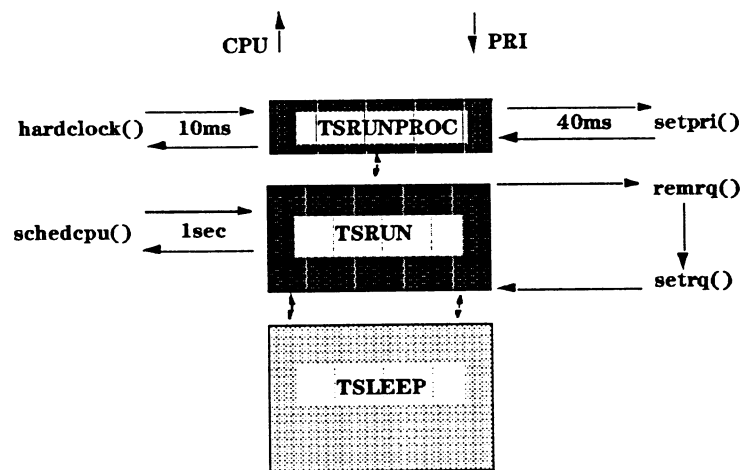
`setrq()` - *pm_swch.c*

`setrq()` is the routine used to put threads onto the run queues. There are per-spu run queues for normal (`SCHED_TIMESHARE`, `SCHED_RTPRIO`) threads and one global set of run queues for **RTSCHED** (`SCHED_FIFO`, `SCHED_RR`, `SCHED_RR2`) threads. The following is a brief overview of the steps taken by `setrq()`

- Get protection (*spl7* in UP case, thread lock in MP case).
- For the debug kernel check, determine if the queue the thread is already on is a run queue and if it has a valid priority.
- Switch on the scheduling policy of the thread to be stuck on a run queue. Do policy-specific setup.
- For timeshared and rtprio scheduled threads, update the processor specific run queue information from the `mpinfo` structure.
- For Posix realtime threads, the run queues are in the system-wide `rtsched_info` structure.

Slide: Adjusting a Thread Priority - Review

Adjusting a Thread Priority — Review



a696322

Notes:

Slide: Adjusting a Thread Priority - Review

Let's review how a thread priority is adjusted based on the information we have covered.

“Thread is executing “

Every 10 msecs, the routine *hardclock()* is called with SPL5 to disable IO modules, software interrupts and kernel preemption. This routine calls the per-processor hardclock routine *per_spu_hardclock()*, which looks for threads with a higher priority which are ready to run (the searching of the processor run queues depends on the scheduling policy). If a thread is found, the **MPSCHED_INT_BIT** in the processor **EIRR** (*External Interrupt Request Register*) is set. This causes the system to trap as soon as the processor level drops to **SPLPREEMPTOK**.

When the system receives an **MPSCHED_INT** interrupt while running a thread in USER mode, the trap handler puts the thread on a run queue and does a context switch to bring in the high priority thread (unless the thread has been running for less than *swtch_delay_cycles* - this is set to 5 milliseconds in the kernel and is not a tunable parameter).

SPLPREEMPTOK() *user mode*

Allow all interrupts. Enables LPMC, powerfail and all EIRR bits, except HPMC_INT, which is reserved for crash dump use.

SPLNOPREEMPT() *kernel mode*

Disable kernel preemption. Same as **SPLPREEMPTOK()**, except 'trap type' interrupts are disabled. 'trap type' interrupts cause the kernel to be preempted.

Currently there are 2 'trap type' interrupts:

MPSCHED

In an MP system, one cpu tells another cpu to check the *runq* by sending this interrupt type.

KPREEMPT

When the kernel is executing and a higher priority process should run (e.g. clock tick recomputed priorities or external interrupt caused a *wakeup()*), the kernel sends itself a **KPREEMPT** interrupt to cause a preemption when the spl level is **SPLPREEMPTOK**.

```
#define SPLNOPREEMPT    0xffffffff0
#define SPLPREEMPTOK    0xffffffffe
```

Slide: Adjusting a Thread Priority - Review

If the current executing thread is the thread with the highest priority then it will be charged with 10ms for running. *hardclock()* calls *setpri()* every 40ms to review the thread's working priority (*kt_pri*). *Setpri()* adjusts the user priority (*kt_usrpri*) of a timeshare thread process, based on cpu usage and nice values. While a time-share thread is running, *kt_cpu* time goes up and its priority (*kt_pri*) gets worse. **RTSCHED** or **RTPRIO** thread priorities are not adjusted.

“Thread is on the run queue”

Every 1 second, *schedcpu()* is decrementing the *kt_cpu* value for each thread on the run queue. *setpri()* is called to calculate a new priority of the current thread being examined in the *schedcpu()* loop. *remrq()* is called to remove that thread from the run queue and then *setrq()* places the thread back into the run queue according to its new priority.

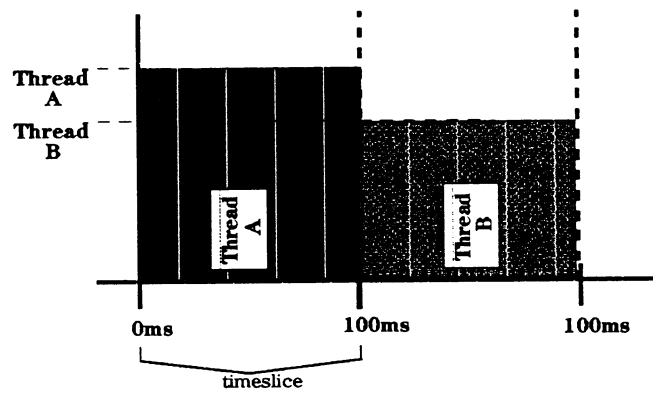
“Process is sleeping or on a swap device (not on the run queue)”

The user priority (*kt_usrpri*) is adjusted in *setpri()* and *kt_pri* is set in *schedcpu()*.

Left blank intentionally

Slide: Context Switching: Review

Context Switching — Review



a696323

Notes:

Slide: Context Switching: Review

Context Switching

In a thread-based kernel, the kernel manages context switches between the kernel threads rather than processes. Context switching occurs when the kernel switches from executing one thread to executing another. The kernel saves the context of the currently running thread and resumes the context of the next thread that is scheduled to run. When the kernel *preempts* a thread, its context is saved. Once the preempted thread is scheduled to run again, its context is restored and it continues as if it had never stopped.

The kernel will allow context switch to occur based on one of the following circumstances:

- Thread exits
- Thread's timeslice has expired and a *trap* is generated.
- Thread puts itself to sleep, while waiting on a resource.
- Thread puts itself into a debug or stop state
- Thread returns to user mode from a system call or trap
- When a higher thread becomes ready to run — give realtime - aδcomorrtlee
npnopas etur

Preemption

HP-UX does not allow the preemption of the kernel. Only user threads can be preempted. A kernel thread will be preempted only when it is about to return from kernel mode to user mode. If thread is at a priority higher than the running thread, it can preempt the current running thread. This will occur if the thread is awakened and receives the resource it has requested.

Who Gets Charged For The Time?

THREAD/PROCESSOR INTERVAL TIMING - *pm_cycles.h*

These timing intervals are used to measure user, system, and interrupt times for threads and idle time for processors. We take and record these measurements in machine cycles. This gives us maximum resolution. We do it atomically so no time falls between the cracks and is lost.

Algorithm Overview:

Here are all of the state transitions. In each diagram, time passes from left-to-right. A thread comes out of *resume()*, either from another thread or the idle loop:

In *resume_cleanup()* (under the protection of whatever lock is held there) we snapshot the time, attribute the interval to the previous thread if there was one or our own idle time if not, mark the new interval as

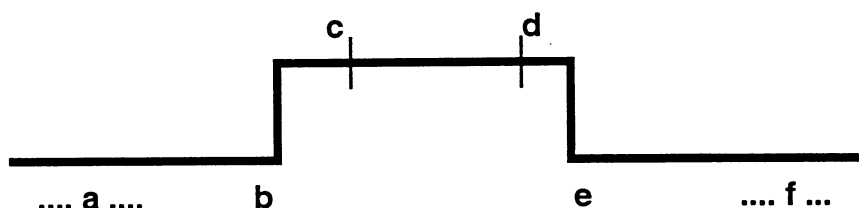
Slide: Context Switching: Review

starting at the time we just got, and change the current state to `SPUSTATE_SYSTEM`.

The processor is going idle:

In `switch()` (under the protection of whatever lock is held there) we snapshot the time, attribute the interval to the previous thread, mark the new interval as starting at the time we just got, and change the current state to `SPUSTATE_IDLE`.

A user thread makes a system call:



The time-line goes like this: The thread is running in user-mode at (a). It makes a system call at (b). It returns from the system call at (e) and it back to running in user-mode again at (f). It is in system-mode between (b) and (e). Toward the beginning of `syscall()`, at (c), we start a new (system-mode) interval attributing the previous interval to the thread as user time. Toward the end of `syscall()`, at (d), we start a new (user-mode) interval attributing the previous interval to the thread as system time.

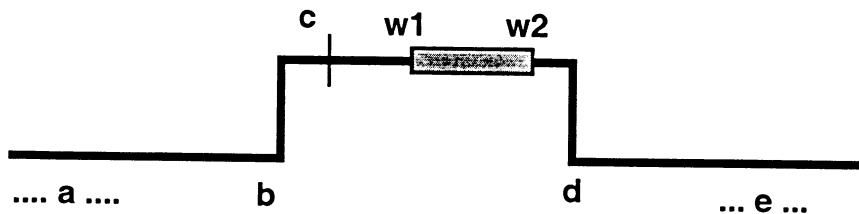
A thread (in user- or kernel-mode) traps:

(Consult the diagram for system calls above.) For timing purposes, traps are handled the same way system calls are handled, with the following exceptions:

1. (c) and (d) are located in `trap()`, not `syscall()`, and
2. whether or not (d) starts a user- or system-mode interval depends on the state of the thread at the time of the trap.

Slide: Context Switching: Review

An interrupt occurs:



This is where things get interesting. Interrupts are handled like traps except that any wakeups that occur while on the interrupt stack, such as (**w1**) and (**w2**) in the diagram above, start new intervals and attribute their time to the thread being awakened rather than the previous thread as recorded in *mpi->prevthreadp*.

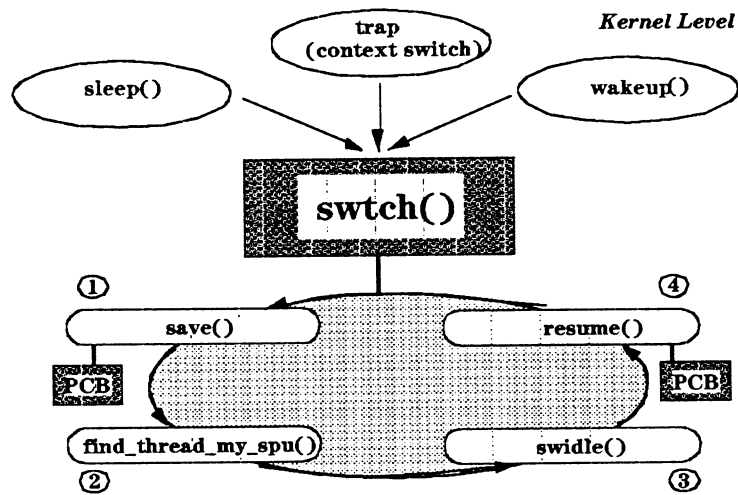
Interrupt time attributed to threads is stored in the *kt_krusq->time_usq.krt_itime* field of the thread structure. Concurrent writes to this field are prevented because wakeup is the only place (other than *allocthread()*) that writes to the field, and it only does this under the protection of a spinlock. Reads are performed (by *pstat()* and others) without locking. Note, these reads are not atomic.

The idea here is that the work being done is really on behalf of the thread being awakened, not the thread that is running right now.

This is an improvement over the old scheme, which gave the entire time from (**b**) through (**d**) to the current thread.

Slide: Thread Scheduling: swtch()

Thread Scheduling — swtch()



a696324

Notes:

Slide: Thread Scheduling: `swtch()`

`swtch ()` - `pm_swtch.c`

Finds the most deserving runnable thread, takes it off the run queue, and starts running it. The slide on the previous page shows the routines utilized by `swtch()`:

- (1) `swidle()` - loops awaiting to take action
- (2) `save()` - saves the thread's process control block (`pcb`) marker.
- (3) `find_thread_my_spu` - searches for new thread to switch in and removes the old.
- (4) `resume()` - resumes a removed thread by restoring the register context and transfers control to the thread.

The following reviews the `swtch()` routine and contains excerpts from the `pm_swtch.c` source file:

- If we are a uniprocessor machine and do not have a thread lock thread, go to `sp17`.
- Update time for the current KI state and push the `KT_CSW_CLOCK` on to (the `sp` value *on entry* to the signal handler).
- Generate a `ki_swtch` trace.
- Call `save(u.u_pcb)` to mark a return point when we come back via resume.
- If our `spu` is to be disabled, call `swidle()` directly.
- Otherwise, if there are any threads on this `spu` to be run, acquire the run queue lock for this processor.
- Call `find_thread_my_spu()` to get one. If we get one:

- Set up the new thread to run.
- Mark the interval timer in the `spu`'s `mpinfo`.
- Mark the `spu` as `MPSYS`
- Update the `fss` state.
- Remove the thread from its' run queue.
- Verify that it really is runnable.
- Set the `EIRR` to `MPSCHED_INT_ENABLE`
- Mark the thread as a running thread (`TSRUNPROC`)

- Else, run idle, accumulate time to clock and call `swidle()`. We will not return directly here
- Release the run queue lock.

Slide: Thread Scheduling: `switch()`

- Mark our spu as having handled the scheduling wakeup.
- Call `resume()` to return the new guy to his previous place
- If we went to spl7 as a thread_unlocked uniprocessor, restore the old spl level from interrupt save and accumulate time to the `KT_CSW_CLOCK` on tos and pop it.
- Return.

`find_thread_my_spu()` - `pm_policy.c`

For the current spu, find the most deserving thread to run. The search starts at `bestq`, which is an index into the table of run queues.

`swidle()` - `asm_util.s`

Performs “SWitch to the IDLE stack and branch to (not call) `idle()`”. Checks that we have a valid `kt_link`. Go to spl7 in the UP case. Find the thread’s spu. Drop the count of threads on run queues. Update `ndeactivated`, `nready_free`, `nready_locked` in the `mpinfo()` structure remove the thread from its run queue. Restore the old spl level. New changes: if it’s `RTSCHED`, update `RTSCHED` counts.

`resume()` - `resume.s`

Restores the register context and transfers execution to the new thread.

`save()` - `resume.s`

Routine called to save states

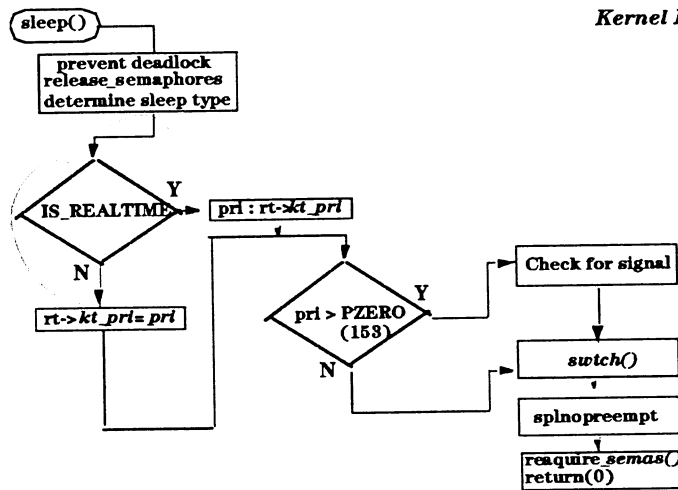
Module 4 — Process Management

Left blank intentionally

Slide: Thread Scheduling: sleep()

Thread Scheduling — sleep()

Monitors mu
rt->kt_pri



a696325

Notes:

Slide: Thread Scheduling: `sleep()`

`sleep()` - `pm_swch.c`

Gives up the processor until a wakeup occurs on channel. During the `sleep()` the thread enters the scheduling queue at priority (`pri`). When (`pri`) is \leq **PZERO** (153), a signal cannot disturb the sleep but if (`pri`) $>$ **PZERO** we will process the signal request.

In the case of **RTPRIO**, a signal will only be able to be disturbed if **SSIGABL** is set. The setting of **SSIGABL** is dependent on the value of (`pri`).

`sleep()` calls `real_sleep()` with the sleep type set to **ST_WAKEUP_ALL**.

`real_sleep()` - `pm_swch.c`

Is passed the following parameters from `sleep()`

```
caddr_t    chan;           what to sleep on
int        disp;          priority to sleep at and sleep flags
caddr_t    caller;       who called sleep()

type = ST_WAKEUP_ALL
```

Since `real_sleep()` performs the work, let's take a look at the steps performed:

- `real_sleep()` determines appropriate sleep queue from the type passed in.
- We determine if we must lock the sleep queue lock, if so we will release on the `swch()`.
- The semaphores that were held are released and we keep track of these in `sv_sema`.
- Acquire the thread lock.
- Set our `kt_wchan`.
- Set `kt_sleep_type`.
- Determine our priority while we sleep.
- If we are **REALTIME**, make our priority the stronger of the requested value and our `kt_pri`.
- Otherwise, we use the requested priority.

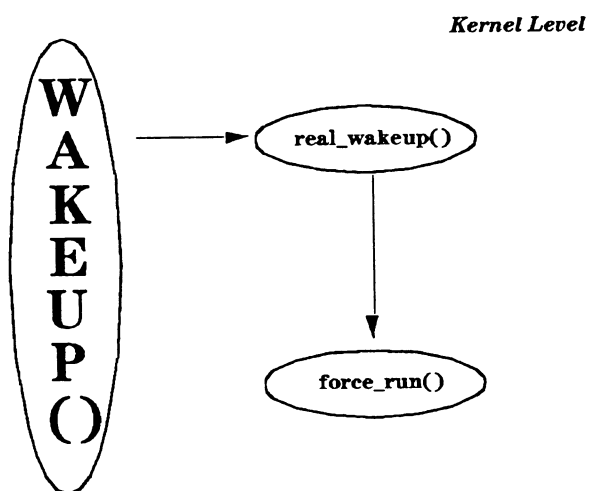
Slide: Thread Scheduling: `sleep()`

- Place the thread on the appropriate (selected by `hash()`) sleep queue.
- Unlock the sleep queue lock.
- If we are sleeping at an interruptible priority ($pri > \text{PZERO}$), mark ourselves as `TSSIGABL`.
- Check to see if we have received a signal while starting to nap with `issig()`. If so and if we have set our `kt_wchan`, call `unsleep()` to unhook things. Then make ourselves `TSRUN` and release the thread lock. Force the `spl` level to `splnopreempt()` and bail out to `psig()`.
- If we have no `kt_wchan` then quietly unlock the thread lock and leave via the out label. Mark our state as `SSLEEP` and bump the number of voluntary context switches for the thread. Call `swtch()` to block us.
- After time passes we wake up and check again with `issig()` to determine if we have received a signal. If so, we go to `psig()` to dispatch it. Otherwise, we are sleeping at an uninterruptible priority and mark our thread `TSSLEEP` and `!TSSIGABL`. The voluntary context switches for the thread are increased and `swtch()` is called again.
- Time passes and we awake to call `reacquire_semas()` to reacquire the semaphores that we put aside above.
- If priority was weak ($< \text{PZERO}$) we will be awakened by a signal if `PCATCH` is set. We will reacquire our semaphores and return 1 to denote a signal. Otherwise, we `longjmp()` back to `syscall`, leaving all semaphores and spinlocks unacquired.

Left blank intentionally

Slide: Thread Scheduling: wakeup()

Thread Scheduling — wakeup()



a696326

Notes:

Slide: Thread Scheduling: wakeup()

wakeup() - *pm_swch.c*

Is the counterpart to *sleep()*. If a thread goes to sleep by calling *sleep()*, it must be awakened by calling *wakeup()*.

When *wakeup()* is called, **ALL** threads that are sleeping on the wakeup channel will be awakened. This can result in one of the other threads getting the resource before us. This will be normal but to help avoid this we will reevaluate whether the resource is really available via a loop. An example of this would be:

```
acquire for buffer lock.
{
    while (buffer is marked busy)
        acquire sleep lock.
        sleep(buffer address);
        acquire the buffer lock.
}
mark the buffer as busy.
release the buffer lock.
```

wakeup() will call *real_wakeup()* to perform the actual work.

real_wakeup() - *pm_swch.c*

Is passed the channel to talk to (*chan*) and the type of **ST_WAKEUP_ALL**. Its purpose is the awake one or more threads sleeping on channel "*chan*" depending on the type passed.

The following is a brief overview of the actions taken by *real_wakeup()*:

- Determine appropriate *slpque* data structure based on the type of wakeup passed in.
- For an MP system, acquire the sleep queue lock if needed. For a UP system, set spl level to spl6.
- For all threads on the appropriate sleep queue, acquire thread lock.
- If the *kt_wchan* matches the argument *chan*, remove them from the sleep queue. Next update the sleep tail array, if needed.
- Clear their *kt_wchan* and clear their sleeping time.
- If they were **TSSLEEP** and not for a beta semaphore, assume they were not on a run queue and call *force_run()* to get the **TSRUN**.
- Otherwise, if they were swapped out (**TSRUN && !SLOAD**), take steps to get them swapped in.

Slide: Thread Scheduling: wakeup()

- If we're on the ICS, we want to attribute this time to the thread being awakened. Start a new timing interval attributing the previous one to the thread we're waking.
- Restore the spl level, in the UP case
- Release the sleep queue lock as needed, in the MP case

force_run() - *pm_swch.c*

Called when a thread is to be placed in the **TSRUN** state. The following is a review of the steps taken by *force_run()* and contains excerpts from the *pm_swch.c* source code:

- We mark the thread **TSRUN**.
- Assert that the thread is in memory (**SLOAD**), put it on a runq with *setrq()*
- If its priority is stronger than the one currently running on that processor, force a context switch.
- Set the processor's wakeup flag and notify the thread's processor (*kt_spu*) with *mpsched_set()* routine.

Module 4 — Process Management

Left blank intentionally

Module 4 — Process Management

Lab: Process Scheduling

- 1) Write a small CPU intensive program that does no IO and never sleeps. It can be written as either a shell script or a compiled program, it doesn't matter.

If you are running under X Windows, run `xload` to plot the system load average.

- 2) Run twenty copies of your program in the background and observe their priorities and `kt_cpu` values (this is the C column in the `ps -l` output). What happens?

To get an interactive feel of the rest of the system, ask yourself: Do other commands run OK or are they be greatly affected by the CPU load?

- 3) Run another copy of your program but use `nice -10` and as root. Then run again, this time using a negative amount of niceness `nice --10`.

What values for nice does `ps -l` show? Is this what you expected?

Compare the amount of CPU time for the two new processes against the original 20. How long does it take to catch up using the `nice --10`?

Does this effect the feel of the system?

- 4) Run one last copy of the program, only this time assign it a realtime priority with

```
rtprio 100 {yourprog}
```

What effect does this have on the interactivensess of the system?

Now TOC your system (we are going to read this crash dump later).

1. In a new window, start a process that will then sleep - waiting for input from the keyboard:

```
# more <BIGFILE>
```

then iconify this window.

2. In a second window, find the PID of this process:

```
# ps -el | grep more
```

Note the number in the fourth column: _____

Left blank intentionally

Lab: Process Structures

3. Change to the "/usr/contrib/lib" directory and invoke the q4 debugger on the running kernel:

```
# cd /usr/contrib/lib
# ied -h $HOME/.q4_history q4 -p /dev/mem /stand/vmunix
```

4. Load all the proc structures from the proc table:

```
q4> load struct proc from proc max nproc
```

5. Select the process structure belonging to our process using the PID obtained in step 2:

```
q4> keep p_pid == <PID-FROM-STEP-2>
```

6. View the fields of the process structure and note the following values:

```
q4> print -tx | more
```

(common to all structures displayed by q4)

spaceof: _____ (Virtual Address Space ID where structure exists)

addrof: _____ (Virtual Address Offset where structure exists)

physaddrof: _____ (Real Address where structure exists)

(specific to the process structure)

p_flag: _____ (Process flags set)

p_stat: _____ (Process state)

p_nice: _____ (Process NICE value)

p_uid: _____ (UID of process owner)

p_pid: _____ (PID of this process)

Module 4 — Process Management

Lab: Process Structures

p_ppid: _____ (PID of parent process)

Scan the other fields for interesting ones. Don't get bogged down!

7. Load the kthread structure for this process:

```
q4> load struct kthread from p_firstthreadp
```

8. Display the contents of the "kthread" structure and note the following:

```
q4> print -tx | more
```

kt_flag: _____ (The kthread flags set)

kt_cntxt_flags: _____ (The kthread context flags set)

kt_wchan: _____ (The Wait Channel the thread
is sleeping on)

kt_upreg: _____ (The pregon for this thread's
U-area)

kt_usrpri: _____ (The user level priority of this
thread)

kt_pri: _____ (The current priority of this
thread)

kt_stat: _____ (The state of this thread)

kt_tid: _____ (The Thread ID of this thread)

Scan the other fields for interesting ones.

9. Follow the "procp" pointer and the "vas" pointer to the "vas" structure:

```
q4> load struct proc from kt_procp
```

Module 4 — Process Management

Lab: Process Structures

```
q4> load struct vas from p_vas
```

10. Display the contents of the "vas" structure and note the following:

```
q4> print -tx | more
```

```
va_ll.le_next[0]: 0x_____ (These are the pointers into  
va_ll.le_next[1]: 0x_____ the skip list of preions)  
va_ll.le_next[2]: 0x_____ the skip list of preions)  
va_ll.le_next[3]: 0x_____ the skip list of preions)
```

```
va_ll.le_prev: 0x_____ (The backward link to all  
preion structures)
```

```
va_refcnt: 0x_____ (Number of structures pointing to this vas - not  
including preions)
```

Scan the other fields for interesting ones.

(Note: See the header file - /usr/include/sys/vas.h for more information)

11. Load the linked list of "preion" structures: (We will load it backwards using the "prev" pointer.)

```
q4> load preg_t from va_ll.le_prev max 100 next p_ll.le_prev
```

(Note: Some structures have accompanying typedefs of the form <STRUCTTYPE>_t. You do not need to use the keyword "struct" if you use the typedef.)

12. Display the following fields of the "preion" structures and note their values:

(See "/usr/include/sys/preion.h" for more information.)

```
q4> print -x p_type p_space p_vaddr p_off p_ll.le_next[0]
```

(Note: The p_ll.le_next[0] on each line is the address of the PREVIOUS line's preion structure. The last line is the vas structure itself.)

What is the address of the U-AREA preion? _____

How many preions are there? _____

How many different types are there? _____

Module 4 — Process Management

Lab: Process Structures

In what order are they linked together? _____

13. Load the pregon structure for the U-area, using the information from step 12:

```
q4> load struct pregon from 0x<ADDR-OF-UAREA-PREGION>
```

14. Find the VAS Space ID and the VAS Offset for this region:

```
q4> print -tx p_space p_vaddr
```

Note the value of "p_space": 0x_____

Note the value of "p_vaddr": 0x_____

15. Using the values from step 14, load the user structure from the U-AREA region:

```
q4> load struct user from 0x<p_space>.0x<p_vaddr>
```

16. Display the contents of the "user" structure and note the following:

```
q4> print -tx | more
```

spaceof: _____ (You'll use these in the memory lab

addrf: _____ exercise to verify your work.)

physaddrf: _____

PCB:

Which "r" registers are saved? ___ to ___

Which "sr" registers are saved? ___ to ___

Which "cr" registers are saved? ___ and ___ to ___

Locate the save locations for: pc, psw, ipsw

Which "fr" registers are saved? ___ to ___

Module 4 — Process Management

Lab: Process Structures

U-area:

Note the pointers back to the Process and Thread structures and to the Save State structure.

How many elements are there in the "arg[]" array? _____

Locate the save locations for: ap, rp, sp

(Note: See the header file "/usr/include/sys/user.h" for more information.)

For the lab exercise on Memory Structures, leave the "more" process there and keep this lab exercise handy.

Extra Credit:

18. Map out all levels of the pregion skip list links:

Module 5

Multiprocessor Systems

"If you do everything, you'll win."

-- Lyndon Baines Johnson

Objectives :

- Understand the basic concepts of MP systems.
- Gain familiarity with MP data structures.
- Understand locking strategies available in kernel for multiprocessing.
- Understand the kernel interface to control MP events.

Slide: Symmetric Multiprocessor Definition

Symmetric Multiprocessor Definition

Two or more processors

Shared main memory

Shared I/O

Single operating system control

a606120

Notes:

Slide: Symmetric Multiprocessor Definition

A multiprocessor is a system with two or more processing units that act in a controlled parallel manner to carry out system activity.

Our multiprocessor implementation has the following characteristics:

- **Two or more processors**

Our multiprocessor implementation is **symmetric**. Each processor has equal capabilities and any kernel task may execute on any processor in the system. In fact, a thread will often execute on more than one processor during its lifetime. Threads are scheduled in a parallel fashion but this aspect is transparent to users.

- **Shared main memory**

All processors have access to all of main memory in a shared fashion. Since each processor has its own cache and TLB, issues of **cache consistency** become a concern. This concern will be discussed later in this module.

- **Shared I/O**

Access to any I/O device is allowed from any processor.

- **Single Integrated Operating System**

There is a single operating system (HP-UX) in control of all hardware and software. The alternative would be to have each processor being controlled by its own independent operating system. This would of course negate some of the benefits of parallelism.

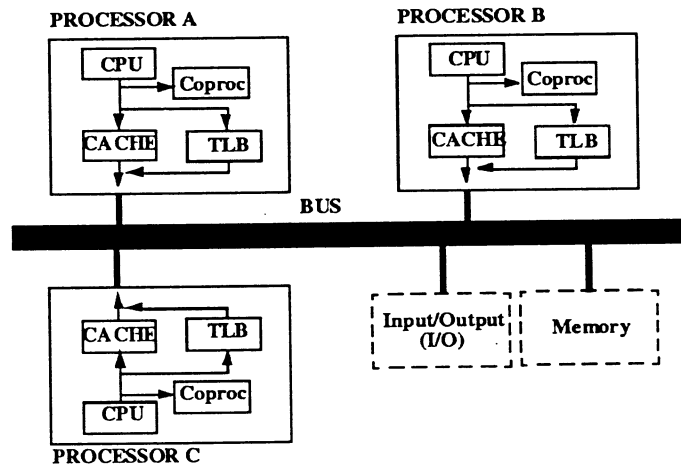
The fact that all processors have access to all memory and I/O resources classifies our implementation as **tightly coupled**. An implementation where each processor has its own private memory and I/O is known as a **loosely coupled** implementation.

NOTE: A full study on multiprocessor internals could easily fill several days of a class and is beyond the scope of this class. While there are many issues of interest such as interrupt handling, powerfail handling, etc. our study here will focus on the multiprocessor data structures and the locking strategies that guarantee consistency across parallel processors.

NOTE: Throughout the remainder of this module multiprocessor systems will be referred to as MP systems. Individual processors will be referred to as both “processors” and “SPUs” (System Processing Unit).

Slide: Hardware Overview

Hardware Overview



a606121

Notes:

Slide: Hardware Overview

The slide at the left shows the basic hardware diagram of an MP system with three processors. Currently, up to sixteen processors are supported.

When the system is powered on, one processor is selected to be the **monarch processor**. The monarch is responsible for all the initial system loader activity and the kernel boot. The selection of the monarch processor is based primarily on the physical slot location. Details of this selection process will be covered in the System Initialization module.

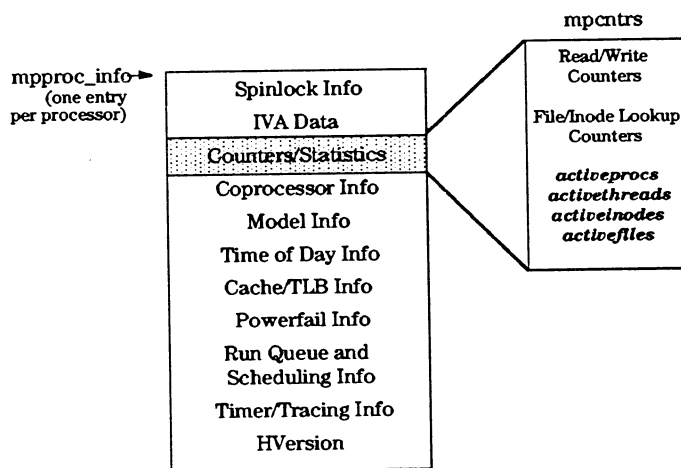
Each processor has its own **cache** and **TLB**. Many parts of the kernel know about the multiple cache and TLBs but its handling of each presents a view as if there were a single cache and TLB. When a processor needs to modify a **cache line**, which is the unit of data passed between cache and memory, it must ensure that the same cache line is not in use by another processor. This is done by sending a **cache coherency check** transaction out on the bus which notifies all processors that a particular cache line is needed. If another processor has the cache line and has modified it, it must flush it to main memory. If the cache line has not been modified, it must be marked invalid so that the next access will force a refresh of the cache line from memory. This is not necessary for the **instruction cache** since it is read-only.

The following example shows how the **cache coherency** issue might arise in a three processor environment:

- Processor A wants to modify a data cache line.
- Processor B has the cache line in its cache and has modified the data.
- Processor B must flush the cache line to memory and invalidate the cache line in its cache.
- Processor C has the same cache line unmodified in its cache.
- Processor C must also invalidate the cache line in its cache.

Slide: MP Data Structures

Multiprocessor Data Structures (mpinfo)



a096122

Notes:

Есть нег. агу. управления процессором и архив в формате списка за заданиями

Slide: MP Data Structures

The kernel maintains an MP data structure containing some necessary information for each spu on the system. The kernel variable `mpproc_info` points to an array of `mpinfo_t`, indexed by spu number. Another kernel variable `mpproc_infoNPROCESSOR_COUNT` points to the end of the array. The variable `nmpinfo` contains the number of `mpinfo` structures (and thus the number of processors).

The general content of each `mpinfo` entry is:

- Spinlock Information
- Interrupt Vector data
- MP Counters and Statistics
- Coprocessor Configuration
- SPU Model info
- Time of Day and Interval Timer data
- Cache/TLB Info (not fully implemented)
- Powerfail Information
- Run Queue and Scheduling Information
- Timer and Tracing Information
- Hversion data

The slide shows the `mpproc_info` per processor data structure which is made up of many different structures. We want to look at the overall structure and then each of the component structures in this and the following slides.

Per Processor Data Structure (`mpinfo_t`):

The `mpinfo_t` structure is a very large structure. Some of the more important fields are explained below. The entire structure can be displayed in q4 as follows:

```
q4> fields -c struct mpinfo
struct mpinfo {
    struct processorhpa *prochpa; /* Pointer to processor HPA */
    caddr_t iva; /* Interrupt Vector Address */
    struct mp_rq mp_rq; /* Processor Run queues */
    :
    :
```

definition and field name	meaning
prochpa	pointer to processor HPA (Hard Physical Address)
iva	Interrupt Vector Address (also stored in CR14)
topics	Top of Interrupt Stack
ibase	Base of Interrupt Stack
ics	ICS in use

Module 5 — Multiprocessor Systems

Slide: MP Data Structures

definition and field name	meaning
idlestack_ptr	Idle Stack Pointer
procindex	Processor Number
cpustate	Valid values are MPBLOCK, MPIDLE, MPUSER, MPSYS, MPSWAIT
spinlock_depth	Number of spinlocks held
entry_spl_level	If spinlock depth > 0, this is the spl level this processor was at when the first of the currently held spinlocks was taken.
threadp	Pointer to current thread structure
uareasid	Space ID of thread's UAREA
curstate	Valid values are SPUSTATE_NONE, SPUSTATE_IDLE, SPUSTATE_USER, SPUSTATE_SYSTEM, SPUSTATE_UNKNOWN, SPUSTATE_NOCHANGE
prevthreadp	Pointer to previous thread switched to
mpcntrs	Per-Process Counters
coproc_info	Coprocessor Configuration masks
model_info	Processor Model Information
tod_info	Time of day Information
cache_tlb_info	Cache/TLB configuration
pf_info	Powerfail state information
mp_rq	Processor Run queues

Within each mpinfo array entry is the **mpcntrs** structure which holds some interesting per-processor counters. The layout of this area is shown below.

Per Processor Counters (mpinfo.mpcntrs):

```
q4> fields -c struct mpcntrs
struct mpcntrs {
    u_long fsreads;      Number of reads to Filesystem Blocks
    u_long fswrites;    Number of writes to Filesystem Blocks
    u_long nfsreads;    Number of NFS reads requested
    u_long nfswrites;   Number of NFS writes requested
    u_long bnfsread;    Number of bytes read via NFS
    u_long bnfswrite;   Number of bytes written via NFS
    u_long phread;      Number of physical reads requested
    u_long phwrite;     Number of physical writes requested
    u_long runocc;      Number of times run queue has been non-empty
                        Checked and incremented by schedcpu().
    u_long runque;      Cumulative length of run queue since boot
    u_long sysexec;     Number of exec()'s since boot
    u_long sysread;     Number of read()/readv()'s since boot
    u_long syswrite;    Number of write()/writev()'s since boot
    u_long sysnami;     Number of filename lookups since boot
    u_long sysiget;     Number of inode fetches since boot
    u_long sysselect;   Number of select calls since boot
    u_long dirblk;      Number of directory disk blocks read
    u_long semacnt;     Number of SystemV semaphore operations since boot
    u_long msgcnt;      Number of SystemV message operations since boot
}
```


Slide: MP Data Structures

```
    u_long muxincnt;    Number of mux input transfers since boot
    u_long muxoutcnt;   Number of mux output transfers since boot
    u_long ttyrawcnt;   Number of raw characters read since boot
    u_long ttycanoncnt; Number of canonical characters processed
                        since boot
    u_long ttyoutcnt;   Number of characters output since boot
    long activeprocs;   Number of proc table entries allocated by this spu
    long activethreads; Number of thread table entries allocated by this
                        spu
    long activeinodes;  Number of inode table entries allocated by this spu
    long activefiles;   Number of file table entries allocated by this
                        spu
};
```

The information in the `mpcntrs` structure can be beneficial in comparing the activities of different processors. From this you may be able to determine which processor is handling the majority of NFS traffic or other specific filesystem type activity.

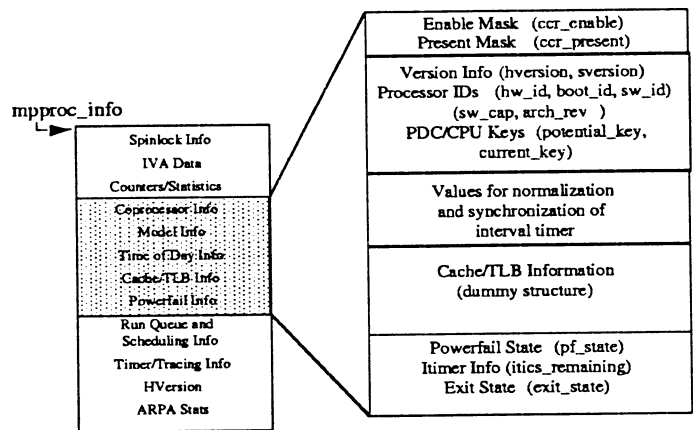
Perhaps the most interesting counters in this structure are the counts for active processes, threads, inodes, and files.

activeprocs	This is the count of the number of processes created by this spu (number of proc table entries). This count is incremented in <code>allocproc()</code> and decremented in <code>freeproc()</code> .
activethreads	Same as activeprocs except that this is the count of threads instead of processes (number of thread table entries). This count is incremented in <code>allocthread()</code> and decremented in <code>freethread()</code> .
activeinodes	This is the count of how many inodes have been allocated by the spu (number of inode table entries). The count is incremented whenever an inode is removed from the free list by routines such as <code>ieget()</code> and <code>vx_inoalloc()</code> .
activefiles	This is the count of how many file table entries have been allocated by this spu. The count is incremented in <code>falloc()</code> and decremented whenever a filetable entry is freed by a call to <code>FPENTRYFREE()</code> .

These counters track the number of active (in-use) entries for each of the respective kernel tables. These counters must be summed across all running processors to obtain the total number of active entries for each table. The decision of which processor's `mpinfo` structure to increment/decrement the counter in is based on which processor is the current processor. So if a process is created on spu A but later terminates while running on spu B, then the **activeprocs** counter will be incremented on spu A but decremented on spu B.

Slide: Per Processor Data Structures

Per Processor Data Structures (CPU Information)



a606123

Notes:

Slide: Per Processor Data Structures

Each `mpproc_info` entry also holds spu configuration information for coprocessors, model info, interval timers, cache and TLB, and powerfail operation.

Coprocessor Information (`mpinfo.coproc_info`):

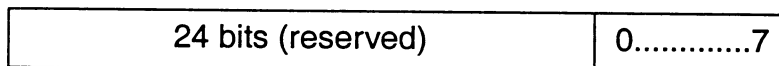
The purpose of the `coproc_info` structure is to maintain configuration information related to coprocessors present on the system.

This structure has only two elements:

```
q4> fields -c struct coproc_info
struct coproc_info {
    u_int ccr_enable;    coprocessor enable mask
    u_int ccr_present;  coprocessor present mask
};
```

The `ccr_enable` and `ccr_present` items have the same format as the *Coprocessor Configuration Register* (CCR / CR10). Each is treated as an 8-bit value which indicates the presence and state of coprocessors. The bits in `ccr_present` indicate a coprocessor is present while the bits in `ccr_enable` indicate that it is present and has passed selftest.

Bit positions in `ccr_enable/ccr_present` are numbered 0 to 7 with bit 7 corresponding to general register bit 31



Bit 0 and 1 in the `coproc_info` indicate the state of the floating point coprocessor. If the Floating Point Coprocessor is present and enabled, the value of both elements will be `0xC0`.

Model Information (`mpinfo.model_info`):

The model information in the `mpproc_info` structure contains the version numbers, identifiers, and capabilities of the processor.

```
q4> fields -c struct model_info
struct model_info {
    u_int hversion;    Hardware Version
    u_int sversion;    Software Version
    u_int hw_id;       Hardware ID
    u_int boot_id;     Boot ID
    u_int sw_id;       Software ID
    u_int sw_cap;      Software Capability
    u_int arch_rev;    Architecture Revision
    u_int potential_key; Potential Key
    u_int current_key; Current Key
};
```

Slide: Per Processor Data Structures

The hardware version (**hversion**) is an integer value that identifies the spu type of the processor. This value is more precise than the model string returned by the *'uname'* command because it distinguishes the speed of the spu as well. For example, a model 715/50 has an hversion of 0x3100 while a 715/33 has an hversion of 0x3110. Both systems would return simply "715" for the model string but the hversion allows for more distinction. A full mapping of hversion to spu type can be obtained from the Service CD-ROM.

The architecture revision (**arch_rev**) identifies the PA-RISC level of the spu. This is an integer value that is mapped as follows:

0	PA-RISC 1.0
4	PA-RISC 1.1
8	PA-RISC 2.0

Time of Day Information (mpinfo.tod_info):

The time of day structure in mpproc_info contains frequency information for the processor's time of day clock as well as synchronization information to keep the interval timers for all processors in sync:

```
q4> fields -c struct tod_info
struct tod_info {
    double freq_ratio;
    u_long offset_correction;
    u_long initial_sync_done;
    u_long itmr_sync_value;
    u_long monarch_sync_value;
    u_long ticks_since_sync;
    u_long ticks_per_clk_sync;
    double max_freq_ratio;
    double min_freq_ratio;
    double itmr_freq;
    u_long tod_acc;
    u_long itmr_acc;
};
```

Cache/TLB Information (mpinfo.cache_tlb_info):

Currently, the Cache/TLB information in the mpinfo_t structure is just a placeholder. As long as we have homogenous MP systems, there is no need to keep per-processor Cache and TLB statistics. The ability exists to maintain per-processor Instruction and Data Cache/TLB information in this location should we ever move away from strictly homogeneous multiprocessing.

Slide: Per Processor Data Structures

Powerfail Information (mpinfo.pf_info):

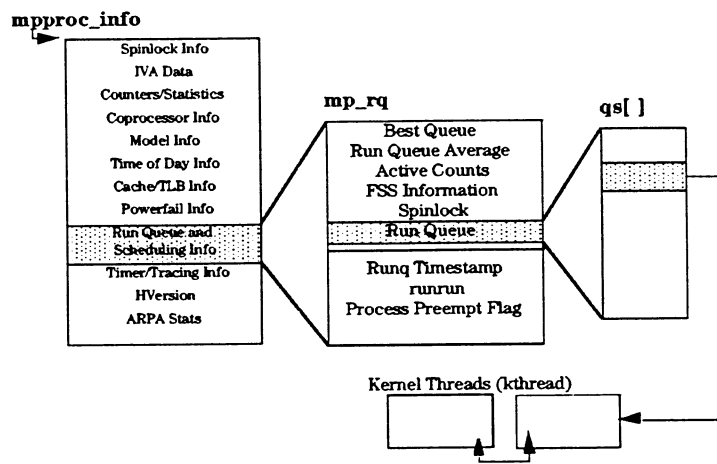
```
q4> fields -c struct pf_info
struct pf_info {
    u_int pf_state;           indicator of the current powerfail state
    u_int itics_remaining;
    u_int exit_state;
};
```

The pf_state field can have the following values:

```
#define PF_SAVESTATE          0
#define PF_PFR_SAVESTATE     1
#define PF_LOCAL_PD           3
#define PF_REMOTE_PD         4
#define PF_RECOVERY           6
```

Потенциально в SMP
можно разные типы
процессоров.

Multiprocessor Run Queues



a606124

Notes:

Slide: MP Run Queues

Each processor has its own set of run queues. Information about these run queues is in the `mpproc_info` structure with the format `mp_rq`

Run Queue Information (`mpinfo.mp_rq`):

```
q4> fields -c -x struct mp_rq
struct mp_rq {
    int bestq;
    int neavg_on_rq;
    int nready_free;
    int nready_free_alpha;
    int nready_locked;
    int nready_locked_alpha;
    int asema_ignored;
    u_int ticks_last_migration;
    ulong_t cycles_last_migration;
    struct _horse *nexthorse;
    lock_t *run_queue_lock;
    struct mp_threadhd {
        struct kthread *th_link;
        struct kthread *th_rlink;
    } qs[160];
    int spares[10];
};
```

This structure is able to construct the run queues by linking threads together. The structure `qs` is an array of pointer pairs that acts as a doubly linked list of threads. Each entry in `qs[]` represents a different timeshare priority queue. It is sized by `NQS` which is currently 160. The `qs[.th_link]` pointer points to the first thread in the queue and `qs[.th_rlink]` points to the tail. The threads in the queue are linked together through `kthread->th_link` and `kthread->th_rlink`.

The `bestq` is an index into the array of run queue pointers (`qs`) and points to the highest priority non-empty queue. If all queues are empty, it points to lowest priority queue. The `neavg_on_rq` is basically the average run queue length for this processor. Simply reading the value from the structure will not give you an understandable number because it is treated as a fixed point number with 8 bits of fraction. The calculation is done with integer arithmetic (shifts and adds) but the general algorithm is

$$\text{neavg_on_rq} = ((255 * \text{neavg_on_rq} - \text{neavg_on_rq}) + \text{current_value}) / 256$$

The current value in this calculation is the total current run queue length. The average is recalculated every 1/100th of a second. This run queue average value is used during spu load balancing which will be discussed on page 6-37.

The sum of `nready_free` and `nready_locked` is the total number of threads in the processor's run queues. The total number of threads locked to this spu is indicated by `nready_locked` and those free to run on any spu are counted in `nready_free`.

The `*nexthorse` pointer is used by Process Resource Manager (PRM) and `*run_queue_lock` is the

Slide: MP Run Queues

spinlock for locking the run queues.

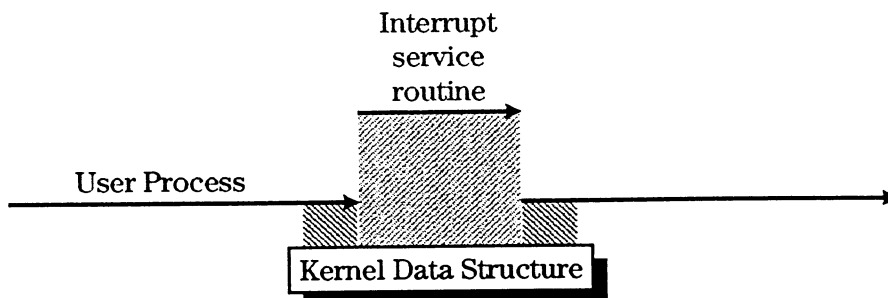
Also in this area of the `mpinfo_t` structure (but not part of `mp_rq`) are the timestamp for the last pass through the run queues (`rqi_cr_it`) and the runrun scheduling flag (`runrun`).

Module 5 — Multiprocessor Systems

Left blank intentionally

Slide: Uniprocessor Data Contention

Uniprocessor Data Contention



a606125

Notes:

Slide: Uniprocessor Data Contention

Where global data structures are being updated by a routine in the kernel, it is not possible for the whole update to be made as a single atomic operation. So during the update it is important that no other part of the kernel attempts to use the data, as it is probably in an inconsistent state.

Since one process may not preempt another whilst it is in the kernel, for single processor machines there is not risk of one process interrupting another during such an update. Here the only potential source of conflict is between a process and an interrupt service routine.

The kernel protects against the possibility of this occurring by masking the delivery of interrupts that could attempt access to the structure being updated.

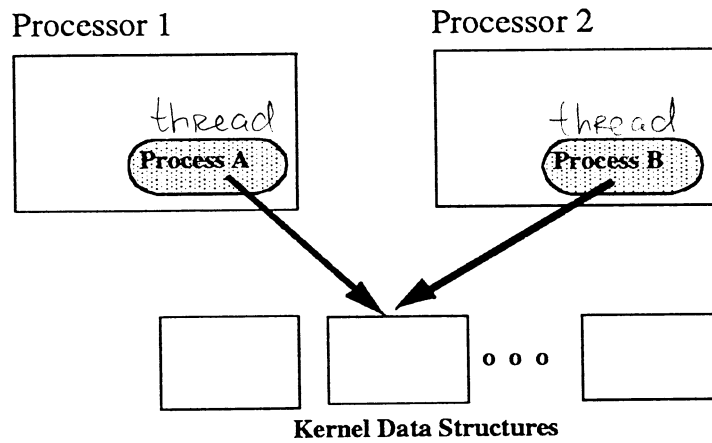
The masking of interrupts is achieved by programming the EIEM (external interrupt enable mask) control register. This register is loaded with sets of pre-configured masks known as the SPLs (system priority levels) where SPL7 is the highest, which disables all external interrupts down to SPL0 where they are all allowed.

Before attempting to update a data structure that is likely to be needed by a service routine, the SPL will be raised to a level that will not allow the interrupt to be processed. Once the update is complete then the SPL can be dropped back to its original level.

For Multiprocessor systems then there is much greater opportunity for contention, here not only can there be contention between processes and interrupt service routines, there can also be contention between processes running on different CPUs. Here a system of locking will be needed.

Slide: Process Synchronization

thread
Process Synchronization



a606126

Notes:

Slide: Process Synchronization

There are three basic forms of contention in the kernel which present the need for some sort of protection

- **Contention between two interrupt service routines**
- **Contention between an interrupt service routine and a process**
- **Contention between two processes**

As discussed in the previous slide in a uniprocessor environment these contentions are easily dealt with. Contention involving an interrupt service routine is handled through raising the **spl** level in a critical code path so that lower interrupt routines or processes cannot run. To prevent process contention we simply do not allow any process to be preempted while running in the kernel mode.

These protection mechanisms are unsuitable for a multiprocessor environment. Waiting for the current process to reach a safe point, sleep, or until it exits the kernel does not give the desired parallelism that we need.

In an MP environment, we abandon the old method and protect kernel data with **software semaphores**, **locks**, and **synchronization primitives**. Kernel data structures are divided into sets with a semaphore or lock guarding each set.

Slide: Locking Strategies

Locking Strategies

Semaphores and Spinlocks

Spinlocks synchronize processes through a busy wait

Semaphores control data access through blocking strategy

a606127

Notes:

Slide: Locking Strategies

In a multiprocessor system it is important to provide a mechanism for protecting global data structures while allowing multiple processors to execute code concurrently in the system. HP-UX provides for this concurrence through locking strategies of **spinlocks** and **semaphores**.

Spinlocks are used to implement a busy wait condition for a resource. If a processor attempts to obtain a spinlock that is held by another processor, it will busy wait until the lock becomes available.

Semaphores control access through blocking strategies. With blocking semaphores, a processor attempting to acquire a semaphore already held by another processor will put its current thread to sleep and context switch to another task.

Spinlocks are used to synchronize access to data between multiple processors while semaphores are used to synchronize access between multiple processes or threads, regardless of how many processors there are. Spinlocks therefore have little value in a uniprocessor system. Within the kernel the *MP_SPINLOCK()* macro checks the uniprocessor flag and simply returns if not an MP system.

In an MP system the decision of whether to use spinlocks or blocking semaphores is a performance issue based on the expected time to busy wait versus the overhead of a process context switch. Additionally, if the lock must be taken while on the Interrupt Control Stack (ICS), then the process cannot block and must use a spinlock.

Slide: Spinlocks

Spinlocks

Used to guarantee access to global data structures by a single thread of execution

Acquired prior to section of code that accesses global data structures

When lock is not available, spinlock busy waits until lock is free

Interrupts are disabled and process is not allowed to sleep while holding spinlock

a696128

Notes:

Slide: Spinlocks

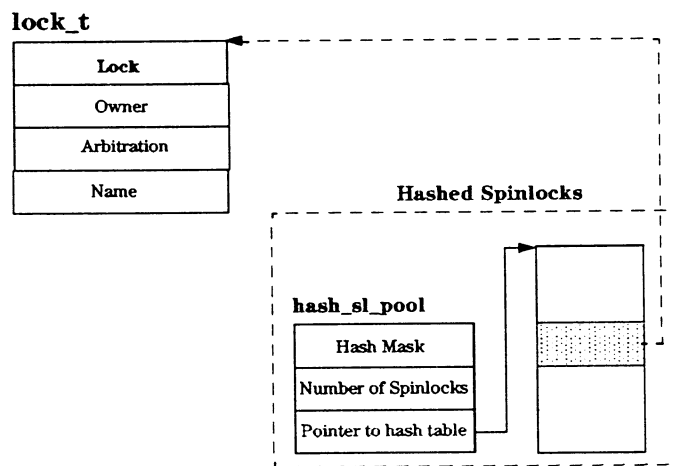
Spinlocks are at the heart of controlling concurrence within an MP system. Their primary purpose is to protect global data structures by controlling access to critical data. When entering an area of code that modifies a global data structure, the kernel will acquire an associated spinlock and then release it when complete.

When a processor attempts to acquire a spinlock that is already owned by another processor, it will busy wait until the lock is free.

While a processor owns a spinlock almost all interrupts are disabled and the current thread is not allowed to sleep. This is done similar to the way that uniprocessor systems do using the *spl()* calls to put themselves at a particular spl level.

Slide: Spinlock Data Structures

Spinlock Data Structures



af06120

Notes:

Slide: Spinlock Data Structures

There are two types of data structures for our spinlock implementation. There is the *lock_t* structure which represents a single spinlock. There are also hashed spinlocks for locating a spinlock within a pool. First we need to understand the single spinlock structure:

Spinlock Data Structure (*lock_t*):

```
q4> fields -c struct lock

struct lock {
    u_int sl_lock;           0 indicates locked spinlock; else last PPDP
    u_int sl_owner;         holds owner's PPDP(Per Processor Data Ptr)
    u_int sl_flag;          1 indicates a waiter exists
    u_int sl_next_cpu;      index into the sl_arb array
                           start search for next owner at this index
    u_int sl_indirect:1;    together with sl_name_ptr, points to
    u_int sl_name_ptr:23;   the name of the spinlock
    u_int sl_pad[3];        padding to 32 bytes
};
```

Notice that the lock structure does not contain a link to the owner's thread or process structure. The *sl_owner* field contains the PPDP or *mpinfo_t* pointer of the owning processor, and we assume that the current running thread holds the lock. It is easy to see that if a thread exits the kernel without releasing the spinlock, there would be no footprint of who had done this.

The *sl_lock* field contains a 0 when the spinlock is locked. When this spinlock is not locked, *sl_lock* contains the PPDP of the last processor to own the lock.

The *sl_flag* field contains a 1 when one or more processors are waiting for the lock, otherwise this field contains a 0.

Hashed Spinlocks

A single spinlock works well for a global data structure that has a single instance or is accessed in a synchronous fashion. For a data structure that has multiple instances, like a vnode structure, it is desirable to have a spinlock associated with individual entries or a group of entries. Using a single spinlock for all vnodes would lead to a lot of contention but using a spinlock for each vnode would be overcompensating for the contention issue.

Our solution is to allocate a pool of spinlocks which are accessed via a hash function. These spinlocks are appropriately called **hashed spinlocks**.

A pool of hashed spinlocks is allocated using *alloc_h_spinlock_pool()* which the kernel calls from *init_hashed_spinlocks()*. A spinlock for a particular instance is then accessed by hashing on the address or some other unique attribute of that instance.

Slide: Spinlock Data Structures

Again using Q4, we can see the hashed spinlock pool of type *hash_sl_pool* is laid out as follows:

```
q4> fields -c struct hash_sl_pool
struct hash_sl_pool {
    u_int    hash_sl_mask;
    int      n_hash_spinlocks;
    int      padding[5];
    lock_t   **hash_sl_table;
};
```

In this structure the **hash_sl_table** points to an array of size **n_hash_spinlocks** which contains pointers to **lock_t** structures. The hash functions return an index into this array of pointers.

Once we have the spinlock data structure, whether directly or through the hash table, the acquisition details to be discussed next are the same.

Module 5 — Multiprocessor Systems

Left blank intentionally

Slide: Load and Clear Word Instruction

LOAD and CLEAR WORD Instruction

*Атомарнае загрузка
і выдаленне. Семафоры*

Mnemonic:

LDCW, cmplt, cc xld (s,b) ,t

Purpose:

Load a value from GR b + offset specified by the Index Register x or Displacement d into GR t, then clear the word (set to 0) all in one atomic operation.

Example:

ldcws,co (arg0) ,ret0

a006130

Notes:

Slide: Load and Clear Word Instruction

The PA instruction set includes an instruction, LOAD AND CLEAR WORD, that loads data from a memory locations and writes a 0 back as a single atomic operation, this is intended to allow efficient implementation of locking. However in early PA implementations the instruction was very slow, up to 180 cpu cycles, and so locking was implemented purely in software. HP-UX 11 though, no longer supports the PA1.0 systems and so this problem has gone away and LDCW can be used. HP-UX 10.20, although it does not “support” PA1.0 systems, is still able to be booted on them, and so still uses the old scheme. This module is based on the HP-UX 11 view.

The **LOAD AND CLEAR WORD (LDCW)** instruction is used to read and lock a word semaphore in main memory. In this context, the “semaphore” is a locking primitive used by the spinlock code. This should not be confused with kernel semaphore structures to be discussed later in this module. The semaphore is simply a 4-byte value, which is set to 0 to indicate the semaphore is locked, and set to 1 if the semaphore is available.

The spinlock code uses this semaphore to indicate whether the spinlock is available or locked. The `sl_lock` field in the `lock_t` structure is the semaphore primitive.

The LDCW instruction has two main purposes:

- Load a value from main memory into a general register.
- Clear the value in main memory by setting it to zero.

This is done in one atomic operation and in one clock cycle. By saying the operation is atomic, we mean the portion of the instruction that does the Load and the portion that does the Clear is indivisible and uninterruptable, even if two processors execute a LDCW instruction on the same memory location at the same time.

There are only two results of a LDCW instruction:

- If the value in memory is 0 (i.e. locked), it is loaded into GR t and then cleared, which means it's unchanged. The spinlock code will see that GR t is 0 and recognize the spinlock is owned by another processor.
- If the value in memory is 1 (i.e. unlocked), it is loaded into GR t, and then cleared (changed from 1 to 0). The spinlock code will see that GR t was 1 and realize that it now owns the spinlock.

Slide: Semaphores

Semaphores

sleep
(wakeup)

Guard kernel data structures

Waiting thread relinquishes spu

Spinlocks protect semaphore data structure

Three Types

Alpha	released on sleep
Beta	retained during sleeps
Synchronization	used for event signaling rather than protecting data structures.

a606131

Notes:

Slide: Semaphores

Like spinlocks, semaphores also guard kernel data structures by controlling access to regions of code that are associated with the set of data structures. The difference is that unlike spinlocks, the waiting thread relinquishes the CPU while waiting on the lock. Depending on the type of semaphore, the process either sleeps or simply does a *switch()* to allow another thread to run.

There are three types of semaphores available in the kernel:

- **Alpha semaphores** which must be released when a process sleeps — не унеем урбага суагб с нунд
- **Beta semaphores** which a process may hold while sleeping — мунд
- **Synchronization semaphores** which are used for event signalling rather than for protecting data structures.

The reason why an **alpha semaphore** cannot be held during sleep is that it is used to protect data structures that must be consistent at the time of context switch. An example of this is the fields in the process/thread structure that describe the process state.

A **beta semaphore** can be held while sleeping because the data structures they protect need not be consistent at the time of context switch. An example of this is the page table during a page fault. The resource must remain locked during the resolution of the fault but the process should yield the processor while its page is brought in from memory.

Alpha and Beta semaphores are also known as **Mutual Exclusion semaphores**.

The table below shows some of the semaphores the kernel creates at initialization time from *init_semaphores()* and *realmain()*. They are listed only by type and name. You can look at the kernel source to observe how each are used.

Alpha Semaphores	filesys_sema mdisc2_sema
Beta Semaphores	msem_betasem iomap_bsema bsem_nstbl, bsem_pinfo, profil_sema, pid_sema, pcred_sema, superpage_text_lock
Synchronization Semaphores	runin runout

Slide: Processor Scheduling

Processor Scheduling

New process is set to start on same spu as parent

One set of run queues per spu

Run queue selection is based on value in
`kthread.kt_spu_wanted`

Each iteration of `schedcpu()` calls
`mp_spu_balance()` to attempt to balance spu utilization

Idle spu can steal threads

a606132

Notes:

Slide: Processor Scheduling

We have completed our discussion of the MP data structures and the locking facilities available in the kernel to synchronize access. Perhaps the biggest challenge in a multiprocessing environment is to evenly distribute the work across available processors. When a process is created on an HP-UX system, it is initially set to run on the same spu as the parent. This decision is made based on the fact that a forked process will likely use some of the same context as the parent. Thus by launching on the same processor we can take advantage of previously cached data and avoid cache coherency issues.

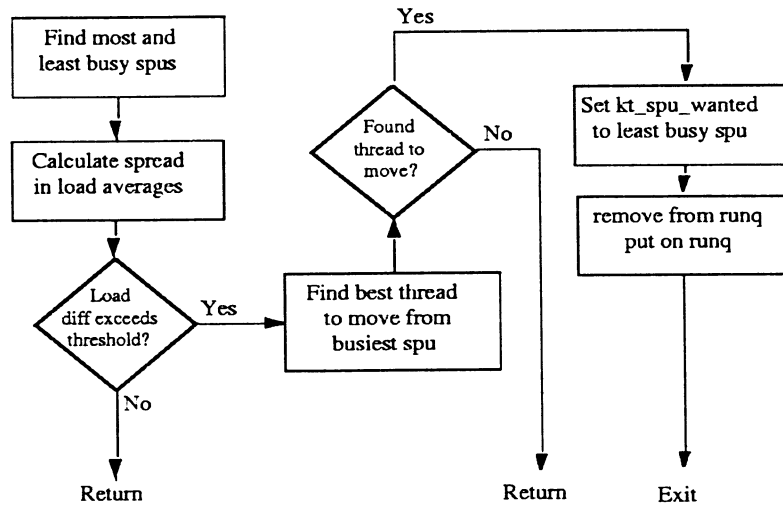
In the Process Management module you have already studied scheduling policies and run queues. In a multiprocessor environment, each spu has a separate run queue. Once a thread is put on a run queue for a certain processor it remains in that queue until removed with *remrq()*.

When a thread is ready to run and put on the run queue with *setrq()*, the processor it is scheduled on is based on the thread's **kt_spu_wanted** field.

One of our concerns with this setup is to keep the relative load balanced across processors. To do this each iteration of *schedcpu()* calls the routine *mp_spu_balance()* which attempts to balance processor utilization by moving threads to different processors' run queues. Additionally, any spu in an idle state may attempt to steal threads from other processors.

Slide: SPU Load Balancing

SPU Load Balancing



a696133

Notes:

Slide: SPU Load Balancing

Mp_spu_balance() is called on each iteration of *schedcpu()* to move threads between available spus in an attempt to balance the load.

The first thing *mp_spu_balance()* does is calculate the spread in instantaneous load averages on spus. To do this it first finds the least and most busy spu by reading through the *mpproc_info* structure to find the highest and lowest values for *mpproc_info[].mp_rq->neavg_on_rq*.

Once we have read through all the *mpproc_info* entries the high and low are compared. If the difference is less than 0.2 then we exit with no balancing to be done.

If the difference in load averages is greater than our threshold, then we need to find the best thread to move from the busiest spu. To do this we read through the threads in the busy processor's run queues computing an arbitrary score for each thread. Prior to computing the score we check that

- The thread has not run in the last second
- The thread is not locked to a particular spu
- The thread has not been moved between processors "recently" (currently 7 ticks)

If any of these conditions are not satisfied then this thread is not a candidate to be moved. Knowing that a thread satisfies these conditions we compute the score based on the thread id:

This algorithm is not completely accurate in that it simply attempts to identify the oldest thread based on the thread id. A more reliable algorithm might be to look at the timestamp on the thread (*kt_start*) but the current algorithm only has a problem if the thread ids wrap back to zero. The assumption being made in this algorithm is that any threads which would be hurt by an unfair move would be eliminated by the checks of *kt_lastrun_time* and *kt_spu_group*.

Once we identify the thread to be moved (the one with the lowest score) its *kt_spu_wanted* field is set to the spu id of the least utilized spu. Calls to *remrq()* and *setrq()* remove the thread from the current spu's run queue and insert it into the run queue for the new spu.

The process explained might be considered a reactive form of load balancing. There is also a proactive form in which an idle spu will "steal" a thread from another spu. The spu chosen to steal from is the one with the longest run queue that is not idle.

Slide: Processor Control

Processor Control

MP control system call (mpctl)

Supported as of 10.20

Available operations

MPC_GETNUMSPUS	Return number of processors
MPC_GETFIRSTSPU	Return index of first real spu
MPC_GETNEXTSPU	Return index of first real spu after arg 1
MPC_GETCURRENTSPU	Return index of current spu
MPC_SETPROCESS	Assign process to a specific SPU

a606134

Notes:

Module 5 — Multiprocessor Systems

Slide: Processor Control

When a thread is created, the kernel attempts to keep it on one processor for its entire life. In the circumstances mentioned previously there is the possibility of a thread moving between processors during its lifetime.

Because this movement is not always desirable some application developers desire the ability to specify **processor affinity**, which is the ability to lock execution of a thread to a specific processor.

This type of processor control, along with other operations, is available through the *mpctl(2)* system call interface. The operations available through *mpctl(2)* are:

MPC_GETNUMSPUS	Returns the number of processors on the system
MPC_GETFIRSTSPU	Returns an index to the first real spu This should always return 0 for the monarch
MPC_GETNEXTSPU	Returns the index of the spu after the spu specified in arg1
MPC_GETCURRENTSPU	Returns the index of the current spu for the process
MPC_SETPROCESS	Binds the specified process id to a specified spu (advisory only; scheduling policy will still take precedence over this call)
MPC_SETPROCESS_FORCE	Same as above, except binding overrides scheduling policy
MPC_SETLWP	Binds a lightweight process (thread) to a specified spu (advisory only; scheduling policy will still take precedence over this call)
MPC_SETLWP_FORCE	Same as above, except binding overrides scheduling policy
MPC_GETPROCESS_BINDINGTYPE	Returns MPC_ADVISORY or MPC_MANDATORY to indicate the current binding type of a specified process
MPC_GETLWP_BINDINGTYPE	Same as above, for a lightweight process (thread)

Module 5 — Multiprocessor Systems

Slide: Processor Control

The most commonly used operation is the one to set processor affinity, `MP_SETPROCESS`. The following C program demonstrates the use of this call:

```
#include <stdio.h>
#include <sys/mpctl.h>

main(argc, argv)
int    argc;
char   *argv[];
{
    spu_t  spu;
    pid_t  mypid;
    int    numspu;

    if (argc < 2) {
        printf("setprocessor spunum\n");
        exit(1);
    }

    mypid=getpid();
    numspu = mpctl(MPC_GETNUMSPUS);
    spu = atoi(argv[1]);

    printf("Number of processors = %d\n",numspu);

    spu = mpctl(MPC_SETPROCESS, spu, mypid);
    printf ("spu = %8d pid = %8d\n", spu, mypid);
}
```


Slide: Processor Control

We should avoid the mindset that processor affinity will always benefit the performance of a thread. Some cases where processor affinity may prove beneficial are:

- A thread builds a large cache context, where changing processors would result in the cache lines needing to be flushed and the data reloaded into the new processor's cache.
- Two threads that communicate with each other via an IPC facility such as message queues or shared memory. Since the IPC structures themselves are memory resident, two threads communicating from different processors will cause the cache related to the IPC structure to be continually flushed and reread to guarantee consistency.

Any decision to assign processor affinity needs to be balanced with the impact of single threading applications within the assigned processor.

Slide: MP Problems

MP Problems

MP Race

Cache Coherence

Deadlock/Lock order

a606135

Notes:

Lab: Multiprocessor Data Structure

Since the lab machines are all uni-processor systems it is difficult to examine most of the issues associated with the MP environment. However, since HP-UX 9, the kernel has been the same for both single processor systems and multiprocessor systems so all the issues can still be looked at.

Looking at mpinfo structure.

There are several ways to access the mp_info structures. The “Beginner Guide to Q4” shows how to load the whole mp_info table, for instance. The way that software running on a processor accesses it is by using one of the “temporary” control registers. CR_24 is a pointer to the processor’s own MP info structure. Since this structure changes constantly it is easier to read one from a crash dump, so use the dump generated in the lab for module 4.

Transfer of control - kernel

When a system crashes either with a Panic, HPMC or TOC, a crash dump is saved. But before this happens the system attempts to record key information about the crash into a pair of tables: the crash_event_table and the crash_processor_table. From the crash_event_table entry a pointer references a restart parameter block (rpb) and this contains a dump of the CPU registers, including CR_24, and so this can be used to find the mp_info structure for the processor.

```
root@hert106[crash.0] ied q4
@(#) q4 $Revision: 1.79a $ $Date: 97/09/08 12:00:22 $ 0
Reading kernel symbols ...
Reading kernel data types ...
Initialized PA-RISC 1.1 (no buddies) address translator ...
Initializing stack tracer ...
Get the latest Q4 news by typing "news".
q4> load crash_event_t from &crash_event_table
loaded 1 crash_event_t as an array (stopped by max count)
q4> print -tx
      indexof  0
      mapped   0x1
      spaceof  0
      addrof   0x21000
      physaddrof 0x21000
      realmode 0
      cet_reserved 0          # see /usr/include/machine/crash.h
      cet_hpa   0xffffa0000   # identifies which CPU crashed
      cet_event 0x2          # crash event was a TOC
      cet_savestate 0x6528e8  # pointer to RPB
      cet_start_itmr 0
      cet_cur_itmr 0
q4> load struct rpb from cet_savestate
loaded 1 struct rpb as an array (stopped by max count)
```

Under HP-UX 11.0, a single rpb structure type exists for both PA1.1 and PA2.0 systems. Now the structure contains either 32-bit registers or 64-bit registers. For 32-bit processors, read the value of the registers from pim.narrow. For 64-bit processors, read from pim.wide.

Slide: MP Problems

There are various problems that may be encountered in a multiprocessor environment. Each of these problems affect the kernel in the areas of performance and/or data consistency.

One of these problems is referred to as an **MP Race** condition. This is a situation where there is improper synchronization between processors such that a thread running on one processor adversely affects the execution of a thread on a different processor. Race conditions can always be avoided through proper use of spinlocks and semaphores.

Perhaps the biggest issue in terms of performance in a multiprocessor environment is **cache coherency**. We have already discussed cache coherency in the hardware overview at the beginning of this module. In the processor control discussion we touched on the performance impacts as a result of these issues.

One area that we have not dealt with is **deadlock detection** and **lock ordering**. In a semaphored environment such as this, there is an obvious potential for getting into a deadlock situation, especially if threads on two processors are performing similar operations. To help prevent this, all semaphores have a lock order (**sa_order**) associated with them. The guidelines setup are simply that semaphores with the lowest lock order should be locked first. This way we can guarantee that multiple semaphores are locked in the same order by all threads, reducing the opportunity for deadlock. The kernel has assertions to enforce this lock ordering.¹

1. These assertions are only available in the debug kernel, however.

Module 5 — Multiprocessor Systems

Slide: Lab: Multiprocessor Data Structure

```
q4> print -tx | grep head
    pim.narrow.rp_pcsq_head  0xab3
    pim.narrow.rp_pcoq_head  0xfad3 ..... The processor was in
    pim.wide.rp_pcsq_head_hi  0          user mode when the
    pim.wide.rp_pcsq_head_low  0          system was toc'd
    pim.wide.rp_pcoq_head_hi  0
    pim.wide.rp_pcoq_head_low  0
```

```
q4> print -tx | grep cr24
    pim.narrow.rp_cr24  0x63e288
    pim.wide.rp_cr24_hi  0
    pim.wide.rp_cr24_low  0
```

```
q4>
q4> load struct mpinfo from pim.narrow.rp_cr24
loaded 1 struct mpinfo as an array (stopped by max count)
q4>
q4> print -tx | wc
2110 4220 110745          # The mpinfo structure is very big.
q4>
```

What is the value of

```
prochpa          _____
    does it agree with the hpa from the crash event structure ____

iva_curpri       _____      Current threads priority, is this what
                                     you would expect ?

threadp          _____      Pointer to the current running thread

curstate         _____

coproc_info.ccr_enable _____

coproc_info.ccr_present _____      What does this mean ?

model_info.arch_rev _____      Is this a PA1.0, PA1.1 or PA2.0 system

mp_rq.bestq      _____      Is there anything on that queue
                                     ( Empty queues both the link and rlinks
                                     point to the queue not a thread )
```

What is the lowest priority queue that is occupied ?

```
runrun          _____      There is more than one flag
```

Module 5 — Multiprocessor Systems

Slide: Lab: Multiprocessor Data Structure

Module 6

InterProcess Communication

“We’re supposed to meddle with things we don’t understand. If we hung around waitin’ till we understood everything, we’d never get anything done.”

Granny Weatherwax

“Interesting Times”, Terry Pratchett

Objectives :

- Define the methods used for InterProcess Communication.
- Define the components of the “InterProcess Communication Facilities.”
- Define the purpose of Semaphores.
- Define the purpose of Message Queues.
- Define the purpose of Shared Memory.
- Describe how processes communicate via signals.

Slide: IPC Introduction

InterProcess Communication Introduction

- Files
- Pipes
- Sockets
- Shared Memory
- Message Queues
- Semaphores
- Signals

a696136

Notes:

Module 6 — InterProcess Communication

Slide: IPC Introduction

Unix has traditionally provided a rich set of InterProcess Communication (IPC) facilities. These facilities can provide for the exchange of data between processes and the synchronization of processes.

Table 1: IPC facilities

Facility	data	sync
Files	✓	✗
File locks	✗	✓
Pipes	✓	✓
Messages queues	✓	✓
Semaphores	✗	✓
Shared Memory	✓	✗
Sockets	✓	✓
Signals	✗	Async

Files are the most common variety of IPC facility, and they will be covered in the filesystems module. Pipes allow some of the semantics of files to be used to pass data between processes, with pipe however there is a built in synchronization mechanism. If an attempt to read from a pipe is made when there is no data available then the read will block (sleep).

With the release of SYSV Unix, new IPC facilities were added, message queues, semaphores and shared memory. These have become collectively the SYSV IPC facilities. This module starts off looking at these. Message queues allow data to be passed between processes, and synchronization is provided. Semaphores provide a general purpose synchronization tool. Shared Memory allows multiple processes to see data at the same location, this can give the highest possible performance since data interchange does not involve system calls, and need not even involve any data copying, Shared memory though does not provide a built in synchronization capability and so it needs to be used with some other technique.

At a similar time Berkeley were also looking at IPC, but in their case they wished to be able to communicate between processes on different systems as easily as between local processes, so techniques

Module 6 — InterProcess Communication

Slide: IPC Introduction

like shared memory were non starters. Their solution was sockets. Sockets allow data to be passed between processes, and as with pipes synchronizes the communication.

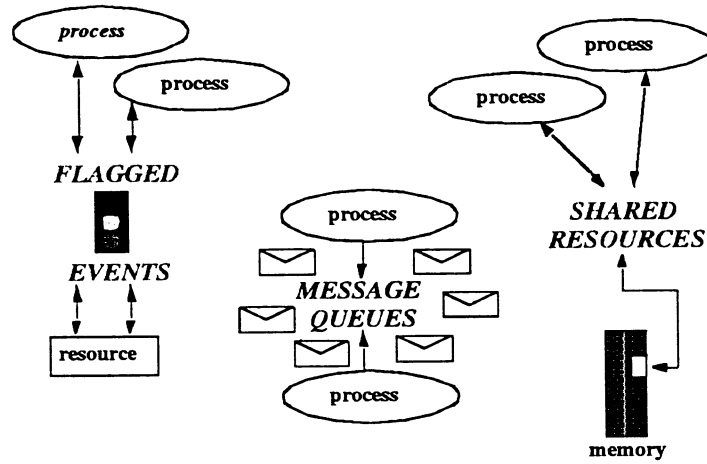
Signals can also be a form of IPC, whilst they do not communicate much in the way of data (just the signal type) they allow synchronization of processes. Unlike the other facilities that provide synchronization there is no need for the receiving process to sleep awaiting data, the signal can arrive asynchronously, and be dealt with by a handler routine.

Module 6 — InterProcess Communication

Left blank intentionally

Slide: IPC Facilities

IPC Facilities



a696137

Notes:

Slide: IPC Facilities

InterProcess Communication (IPC) is a term that defines methods of how a process may communicate with a program or another processes. Processes need a method of exchanging data or synchronizing execution to complete a given task.

Note: although the HP-UX 11.0 kernel is oriented around a threads-based model, the IPC methods still deal with communication between **processes**.

The methods provided by HP-UX may be divided into:

Flagged Events

Based on an action taking place. Tells a process something has occurred or if a flag is set, an event has taken place.

- semaphores
- signals

Communication Queues

A general means of communicating via a queue scheme, utilizing a unique identifier.

- message queues
- pipes
- FIFOs
- streams

Shared Resources

Process may share access with another by accessing the same areas of memory.

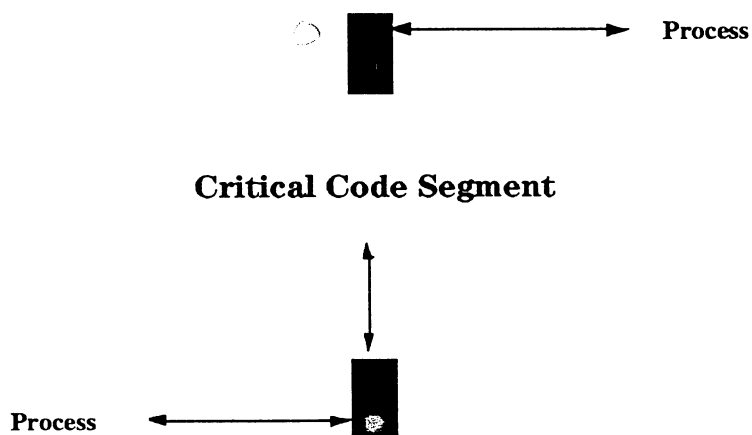
- shared memory

The purpose of this module will be to review the described methods, focussing on the components which are most frequently used; semaphores, message queues, shared memory, and signals¹.

1.

Slide: Process Level Semaphores

Process Level Semaphores



a696138

Notes:

Семафоры в рамках ядра не взаимодействуют —
— защита переключения контекста
накладной

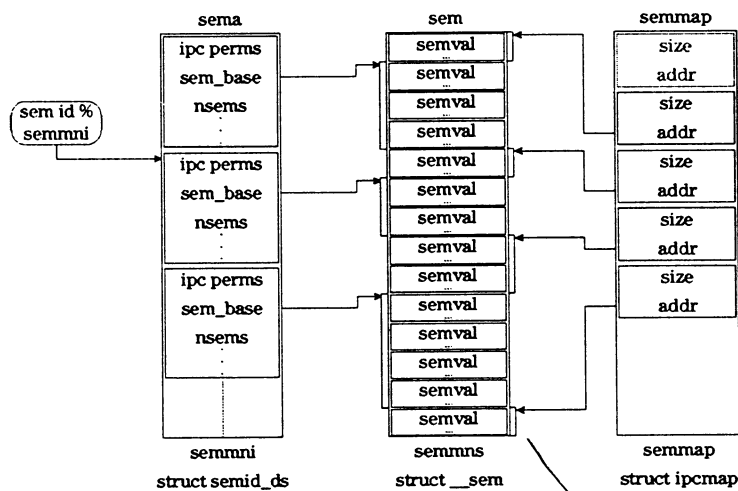
Slide: Process Level Semaphores

Overview

System V IPC semaphores are used mainly to keep processes properly synchronized to prevent collisions when accessing shared data structures. For an example, processes can lock out each other during critical code segments such as updates to shared data segments or databases. Semaphores achieve this control by utilizing event flags to signal availability of a given resource. They may be used as “binary” or “counting” semaphores.

Slide: System V Semaphores: Data Structures

System V Semaphores: Data Structures



*номер semid
пойти до
места semid до
в массиве ipcmap!*

Notes:

Определение на группах семифоров.

Slide: System V Semaphores: Data Structures

Semaphores are allocated and accessed as a **semaphore set** containing one or more semaphores, all associated with the same **id**. Header information for the semaphore ID is held in a **struct *semid_ds***, allocated from the system-wide ***sema[]*** array, sized from the tunable parameter **SEMMNI**. The ***semid_ds*** structure contains:

- operation permissions structure, which includes:
 - creator's user id
 - creator's group id
 - user id
 - group id
 - mode (permissions)
- pointer to the first semaphore in the set
- last time a **semop()** system call was performed

The actual semaphores are represented by a **struct *__sem***. These entries are allocated from a system-wide ***sem[]*** array whose first entry is pointed to by ***sem_base*** and is sized by the tunable parameter **SEMMNS**: contiguous entries are used for all semaphores in the same set. The ***__sem*** structure contains:

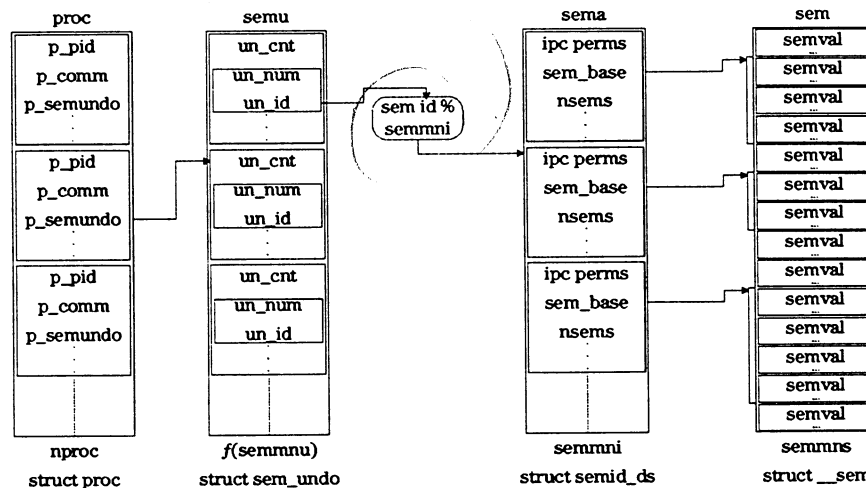
- the value of the semaphore
- process id of the process which performed the last operation on this semaphore
- number of processes waiting for the semaphore's value (***semval***) to become greater (***cval***)
- number of processes waiting for the semaphore value (***semval***) to equal 0

A resource map ***semmap[]***, comprising entries of type **struct *ipcmap*** and sized by the tunable parameter **SEMMPMAP**, holds information on free entries in the ***sem*** array.

The actual definitions of these tables can be seen in the file **/usr/conf/space.h.d/core-hpux.h**

Slide: System V Semaphores: Undo Structures

System V Semaphores: Undo Structures



a696140

Notes:

Проц завершил. процесс. автоматический
бросит. семфор

ID набора данных равен PID

Кто? — उपयोग (Informix)

— к тому-то процессу, имени файла
file (name, dir)

Мониторинг IPC: ipc

Slide: System V Semaphores: Undo Structures

When a process aborts or is killed, it may be desirable to reverse the action of semaphore operations which it has performed: this is referred to as “undoing” the operation. If this was not done, an aborting program could cause an application to hang by failing to indicate that a controlled resource in use by that program at the time of the abort is now accessible by other processes.

Applications indicate that they wish to use this functionality by specifying the **SEM_UNDO** flag in the system call. The first time a process performs a semaphore operation with this flag set, a *sem_undo* structure containing multiple *undo* structures is allocated from the system-wide array *semu[]* and pointed to by the *p_semundo* field in the proc structure. The *undo* structure is completed with the semaphore identifier, the semaphore number, and the shadow value. When the process terminates, the kernel routine *exit()* calls *semexit()* which processes the *undo* entries for this process and handles the associated *undo* operation(s). The *sem_undo* structure is then linked back into the freelist.

Since the value of any semaphore when such conditions occur is unpredictable, the system enforces a limit on how much the value of a semaphore can be changed by an *undo* operation: this limit is defined by the tunable parameter **SEMAEM**.

The *sem_undo* structure contains:

- semaphore number
- semaphore set id
- pointer to the next active undo structure
- adjustment counter

The tunable parameter **SEMUME** specifies the maximum number of semaphores on which an individual process can have pending undo operations.

Slide: System V Semaphores: Kernel Parameter Limits

System V Semaphores: Kernel Parameter Limits

- Sema include SYS V semaphores
- semmni Maximum Number of Identifiers (sets)
- semmns Maximum Number of Semaphores
- semmap Size of free space map (semmni+2)
- semume undos per processes
- semmnu processes per semaphore with undos
- semvmx maximum allowed value of a semaphore
- semaem maximum allowed undo size

a696141

Notes:

Slide: System V Semaphores: Kernel Parameter Limits

Semaphore Limits

Configurable kernel parameters are available to limit the number of sets of semaphores that can exist simultaneously on the system and the total number of individual semaphores available to users that can exist simultaneously on the system. In addition to the tunable parameters listed below, there are also two non-configurable parameters concerning semaphores whose values may not be modified:

semmsl is a limit on the number of semaphores that can exist in a set: its value is fixed at 2048.
semop is the maximum number of semaphores which can be changed by a single system call: it is defined as **SEMOPM** and its value is fixed at 500.

The following tunable kernel parameters control System V IPC semaphore operating limits:

- sema** Enable or disable System V IPC semaphores support in kernel
Acceptable Values:
minimum: 0 (exclude System V IPC semaphore code from kernel)
maximum: 1 (include System V IPC semaphore code in kernel)
default: 1
- semnmi** Defines the maximum number of sets (identifiers) of IPC semaphores which can exist on the system at any given time: used to size the *sema[]* array of *semid_ds* structures. If *semget()* is called when there are no free entries in this array, **ENOSPC** is returned.
Acceptable Values:
minimum: 2
maximum: memory limited
default: 64
- semmap** Specifies the size of the resource map used to represent available semaphore structures.
Acceptable Values:
minimum: 4
maximum: $\leq 32,767$
default: *semnmi*+2
- Each set of semaphores allocated per identifier occupies one or more contiguous slots in the *sem[]* array: as semaphores are allocated and deallocated, this array can become fragmented. If insufficient contiguous semaphore structures are available to satisfy a *semget()* request, **ENOSPC** will be returned even though the number of free semaphores may appear to be adequate. The *semmap[]* is a resource map which shows free spaces in the *sem[]* array. Each entry in the map points to a set of contiguous unallocated slots. If fragmentation occurs and there are insufficient entries in the *semmap[]* to represent all of the free slots in the semaphore array, an error is reported:
- danger: mfree map overflow**
- Note: fragmentation of the *sem[]* array is reduced if all semaphore identifiers have the same number of semaphores; if this is the case, *semmap* can be somewhat smaller.

Slide: System V Semaphores: Kernel Parameter Limits

- semms** Defines the system-wide maximum number of individual IPC semaphores that can be allocated for users.
Acceptable Values:
minimum: 2
maximum: $\leq 32,767$
default: 128
Used to size the *sema[]* array.
- semmnu** Defines the maximum number of processes that can have undo operations pending on the same IPC semaphore.
Acceptable Values:
minimum: 1
maximum: $nproc - 4$
default: 30
An *undo* is an optional, flag in a semaphore operation which causes that operation to be undone if the process which invoked it terminates.
semmnu specifies the maximum number of processes that can have *undo* operations pending on a given semaphore: it determines the size of the *sem_undo* structure.
A *semop()* system call using the **SEM_UNDO** flag will return **ENOSPC** if this limit is exceeded.
- semume** Maximum number of IPC semaphores on which a given process can have undo operations pending.
Acceptable Values:
minimum: 1
maximum: *semms*
default: 10
- semvmx** Maximum value any given IPC semaphore is allowed to reach (prevents undetected overflow conditions).
Acceptable Values:
minimum: 1
maximum: 65,535
default: 32,767
semvmx specifies the maximum value a semaphore can have. This limit must not exceed the largest number that can be stored in a 16-bit unsigned integer (65,535) or undetectable semaphore overflows can occur.
Any *semop()* system call that tries to increment a semaphore value to greater than *semvmx* will return **ERANGE**. If *semvmx* is greater than 65,535, semaphore values can overflow without being detected.

Slide: System V Semaphores: Kernel Parameter Limits

semaem Defines the maximum amount a semaphore value can be changed by a semaphore “undo” operation.

Acceptable Values:

minimum: 0

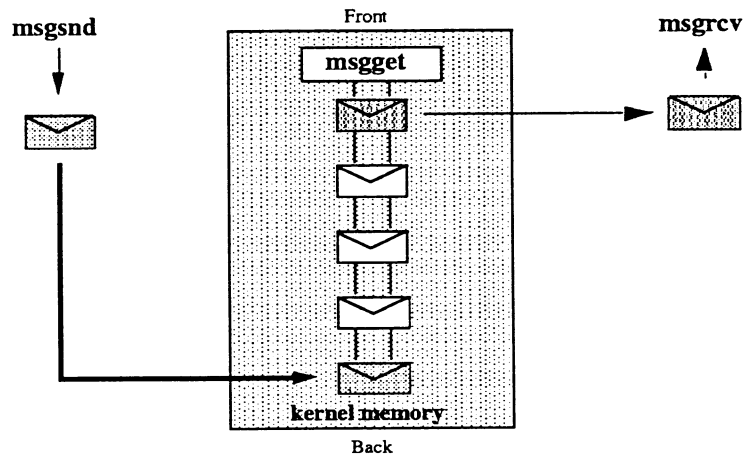
maximum: *semvmx* or 32,767, whichever is smaller

default: 16,384

The *undo* value is cumulative per process. If a process has more than one undo operation on a semaphore, the values of each undo operation are added together and the sum is stored in a variable named *semadj*. *semadj* then contains the number by which the semaphore will be incremented or decremented if the process terminates. Any *semop()* call that attempts to set the absolute value of *semadj* to a value greater than *semaem* will return **ERANGE**.

Slide: Message Queues

Message Queues



a596142

Notes:

Slide: Message Queues

Messages

Messages are small collections of data (400 bytes, for example) that can be passed between cooperating programs through a message queue. These are identified by a message queue identifier (*msgid*). Messages within a queue can be of different types, and any process with proper permissions can receive the messages. A receiving process can retrieve the first message, the first message of a given type, or the first message of a group of types. All messages that pass between processes are linked into a queue.

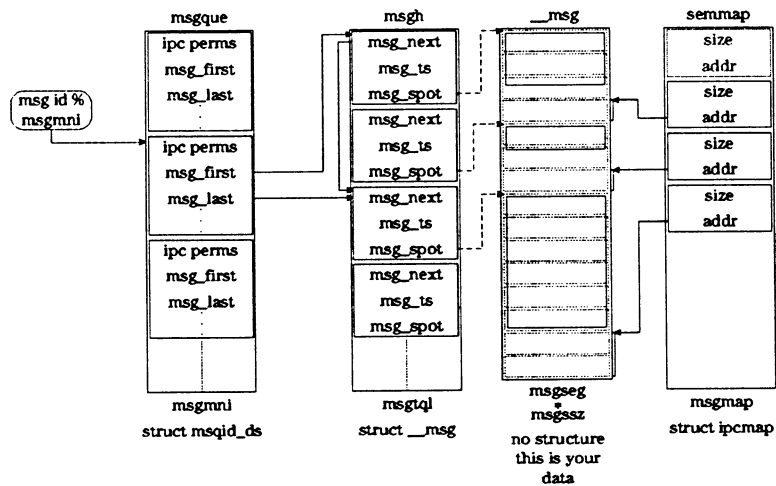
Message Queues

Message queues are implemented as linked lists of data stored in kernel memory. When a message is sent, it is linked to the back of the queue, and when a message is read, it is unlinked from the front of the queue. The message queue itself contains a series of data structures, one for each message, each of which identifies the address, type, and size of the message plus a pointer to the next message in the queue.

To allocate a queue, a program uses the *msgget()* system call. Messages are placed in the queue by *msgsnd()* system calls and retrieved by *msgrcv()*. Other operations related to managing a given message queue are performed by the *msgctl()* system call. We will discuss these in more detail later in the module.

Slide: Message Queues: Data Structures

Messages Queues: Data Structures



a696143

Notes:

Мессежы ўзг. апрацэсаванне ўзгадняецца апрацэсаваннем

Slide: Message Queues: Data Structures

The message allocation map contains a linked list of message queue structures. The data structure is defined as *msqid_ds[]* and contains the message queue id structure. There is one msg queue id structure for each queue in the system.

The *msqid_ds* data structure contains:

- Operation permission structure
- Total # of messages in the queue
- Maximum # of bytes in the queue
- PID of last message sent
- PID of last message received
- Pointer to the first message on the queue
- Pointer to the last message on the queue
- Time of last message sent
- Time of last message received
- Time of last change made

The *msqid_ds* structure consist of a pointer to a linked list of message header array called *msg[]*. The message header array contains one *_msg* structure for each message on the system.

The *_msg* structure contains the following:

- Pointer to next message on the queue
- Message type
- Message text size
- Message text address

A resource map, the *msgmap[]*, is used to maintain a record of free entries in the *__msg* array.

Again as with SYSV semaphores, these tables are declared in *space.h*.

Slide: Message Queues: Allocation of Space

Message Queues: Allocation of Space

System-wide:	msgmap	- number of message queue ids
	msgtql	- maximum number of messages
	msgmap	- size of the resource map
Queues:	msgmax	- maximum size for a single message
	msgssz	- size of a message segment
	msgseg	- maximum number of message segments in a single queue
	msgmnb	- maximum combined size of all messages in a single queue

a696144

Notes:

Slide: Message Queues: Allocation of Space

Configurable IPC Message Parameter

The following kernel parameters control allocation of space for messages and message queues:

- mesg** enable or disable messages at system boot time.
Acceptable Values:
minimum: 0 (exclude System V IPC message parameters from kernel)
maximum: 1 (include System V IPC message parameters in kernel)
default: 1
mesg specifies whether the code for System V IPC message parameters is to be enabled in the kernel at system boot time.
Series 800 systems: IPC messages are always enabled in the kernel.
Series 700 systems: if *mesg* is set to zero, all other IPC message parameters are ignored.
- msgmap** size of free-space resource map for allocating shared memory space for messages.
Acceptable Values:
minimum: 3
maximum: msgseg+2 or msgtql+2 whichever is lower
default: msgtql+2
Message queues are implemented as linked lists in shared memory, each message consisting of one or more contiguous slots in the message queue. As messages are allocated and deallocated, the shared memory area reserved for messages may become fragmented.
msgmap specifies the size of a resource map used for allocating space for new messages. This map shows the free holes in the shared memory message space used by all message queues. Each entry in the map contains a pointer to a corresponding set of contiguous unallocated slots, and includes a pointer to the set plus the size of (number of segments in) the set. Free-space fragmentation increases as message size variation increases. Since the resource map requires an entry for each fragment of free space, excessive fragmentation can cause the free-space map array to fill up and overflow. If an overflow occurs when the kernel requests space for a new message or releases space used by a received message, the system issues the message
DANGER: mfree map overflow
If this error message occurs, the kernel should be regenerated with a larger value for *msgmap*.
- msgmax** system-wide maximum size (in bytes) for individual messages.
Acceptable Values:
minimum: 0
maximum: (msgssz*msgseg), msgmnb or 65535 bytes whichever is lower
default: 8192 bytes
msgmax defines the maximum allowable size, in bytes, of individual messages in a queue. Its value should only be increased if applications being used on the system require larger

Slide: Message Queues: Allocation of Space

messages. This parameter prevents malicious or poorly written programs from consuming excessive message buffer space.

Any *msgsnd()* system call that attempts to send a message larger than *msgmax* bytes will return **EINVAL**.

msgmnb maximum combined size (in bytes) of all messages that can be queued simultaneously in a message queue.

Acceptable Values:

minimum: 0

maximum: (*msgssz***msgseg*), or 65535 bytes whichever is lower

default: 16,384 bytes

msgmnb specifies the maximum total combined size, in bytes, of all messages queued in a given message queue at any one time.

Any *msgsnd()* system call that attempts to exceed this limit will return:

EAGAIN if **IPC_NOWAIT** is set

EINTR if **IPC_NOWAIT** is not set

msgmni maximum number of message queues allowed on the system at any given time.

Acceptable Values:

minimum: 1

maximum: Memory limited

default: 50

msgmni defines the maximum number of message queue identifiers. allowed on the system at any given time. One message queue identifier is needed for each message queue created on the system.

When a New Queue is not Available - The *msgget()* system call returns **ENOSPC** on attempts to allocate a new message queue when *msgmni* message queues already exist.

Orphaned Message Queues - If a process allocates a message queue then fails to deallocate it when the process stops, the message queue remains on the system.

Abandoned message queues may be removed with the *ipcrm* command.

Processes that allocate message queues using the *msgget()* system call should deallocate messages using the **IPC_RMID** command to the *msgctl()* system call.

Message Queue StatusT - The *ipcs* command may be used to determine the status of active message queues.

msgseg number of message segments in a message queue.

Acceptable Values:

minimum: 1

maximum: 32,767

default: 2,048

msgseg defines the number of “message segments” that are available in the queue.

(*msgseg* * *msgssz*) defines the total amount of shared memory space that can be consumed by a single message queue (excluding message header space).

Slide: Message Queues: Allocation of Space

msgssz message segment size in bytes.

Acceptable Values:

minimum: 1
maximum: Memory limited
default: 8 bytes

msgssz specifies the size in bytes of the segments of memory space to be allocated for storing IPC messages. Space for new messages is created by allocating one or more message segments of *msgssz* bytes, as required to hold the entire message.

Queue Size and Fragmentation - Total space available for messages in a queue is defined by the product of *msgseg* \times *msgssz*. Changing the ratio of these two values changes how message space is fragmented for any given messaging usage pattern.

msgtql maximum number of messages that can exist on the system at any given time.

Acceptable Values:

minimum: 1
maximum: Memory limited
default: 40

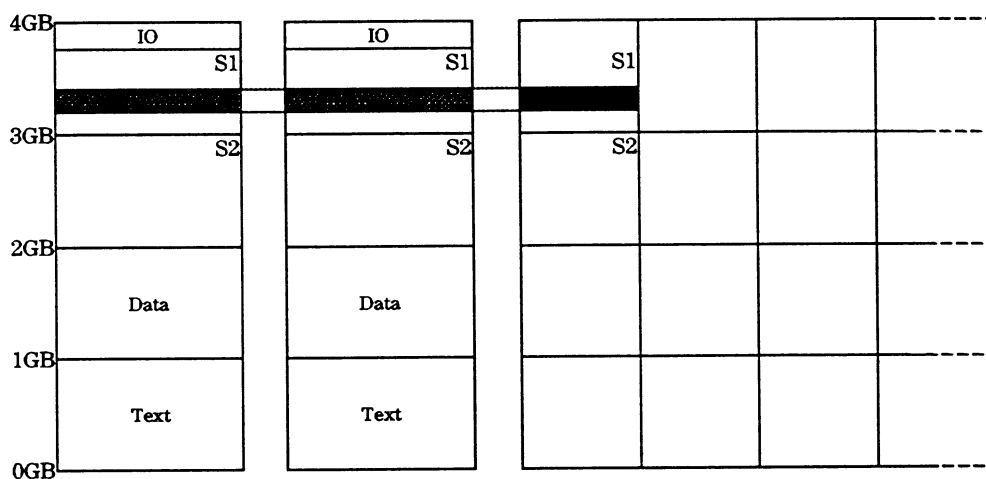
msgtql dimensions an area for message header storage: one message header is created for each message queued in the system. Thus, the size of the message header space defines the maximum total number of messages that can be queued system-wide at any given time.

Message headers are stored in shared (swappable) memory. If a *msgsnd()* system call attempts to exceed the limit imposed by *msgtql*, it:

- blocks waiting for a free header if the **IPC_NOWAIT** flag is not set,
- returns **EAGAIN** if **IPC_NOWAIT** is set.

Slide: Shared Memory and Global Virtual Address Space

Shared Memory and Global Virtual Address space



a696145

Notes:

В HP-UX абстрактно не
ограничен адрес VM

В гpusux — специально устроено
то не гарантируется

Slide: Shared Memory and the global memory map.

Overview

Shared Memory

Shared memory is reserved memory space for storing data structures and data being shared between or among cooperating processes. Sharing a common memory space eliminates the need for copying or moving data to a separate location before it can be used by other processes, reducing processor time and overhead as well as memory consumption.

Shared Memory Access and Use

Shared memory management is similar in many respects to messages and semaphores. A process requests a shared memory segment allocation by means of the *shmget()* system call, specifying the segment size in one of the function parameters. One or more processes can then attach to the allocated segment by using the *shmat()* system call and detach when finished by the *shmdt()* system call. The *shmctl()* system call is used to obtain information about the segment, and to remove the segment when it is no longer needed.

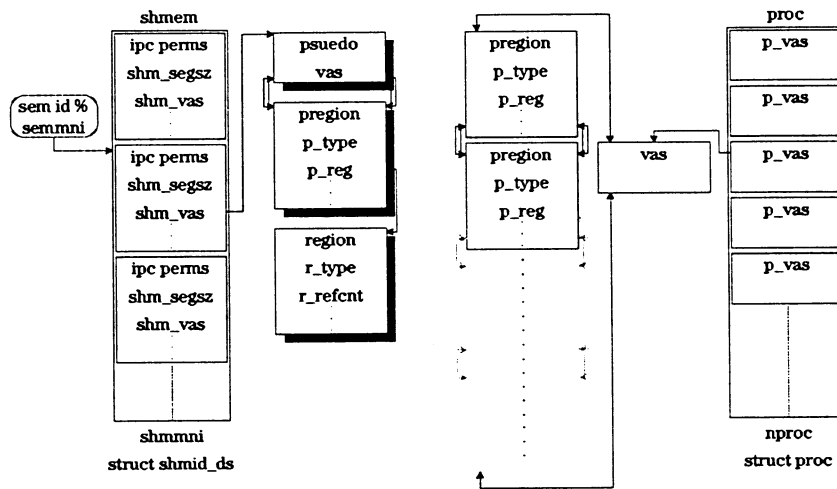
Semaphores can be used to prevent shared memory read/write access collisions, but the more common method is for one process to populate (write to) a given shared memory segment and other processes read from the segment (write once, read many). In such arrangements, when a subsequent write operation is to be performed, the writing process allocates a new segment, and cooperating processes attach to that new segment when ready to use it.

Shared Memory and the global memory map.

With HP-UX's use of the global memory map, the way that the addresses of shared memory areas are handled differs from many Unix implementations. On HP-UX when the shared memory area is created with a *shmget()* system call the area of global virtual address space is allocated. Since a process's local address space is just a view of four quadrants from this global map, when a process attaches to the shared memory segment using *shmat()*, its address is already defined. Also with this arrangement all processes that attach to the same segment will attach it at the same address. Many other Unix variants can not make this guarantee, instead when calling *shmat*, a process can attempt to request a location where it would like to have the area bound.

Slide: Shared Memory Management Data

Shared Memory Management



a696146

Notes:

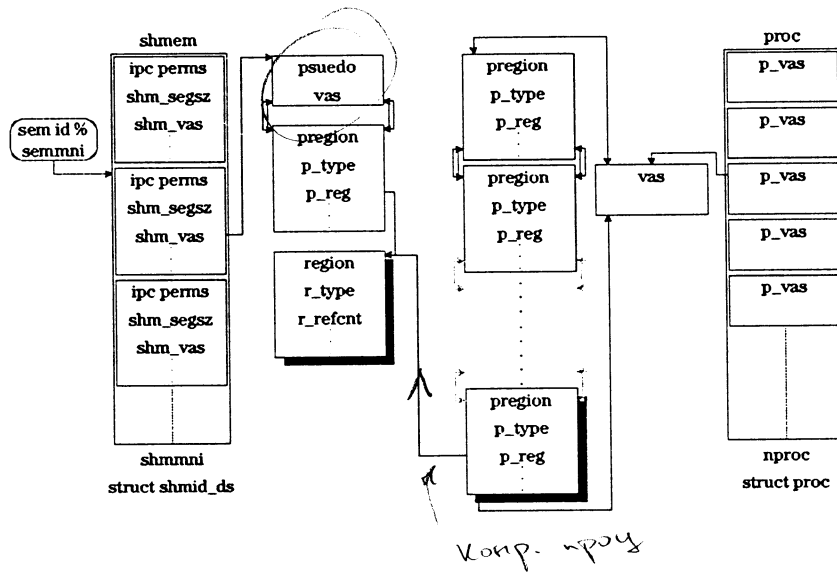
Slide: Shared Memory Management Data

The management of shared memory centers around the `shm` table. The `shmid_ds` data structures map the area of virtual address space for the segment using the same technique as other areas of the kernel, i.e. a VAS structure. When a VAS structure is not attached to a process it is known as a PSEUDO VAS.

When the shared memory segment is created via `shmget()`, the data structures are set up from this VAS, but no change as yet happens to the processes virtual address space.

Slide: Data Structures When Using Shared Memory

Data Structures When Using Shared Memory



a696147

Notes:

Slide: Data Structures When Using Shared Memory

A process attaches to a shared memory segment using `shmat()`. At this stage a new pregon is added into the process list, and this new pregon points to the region data structure of the shared memory segment.

This new pregon will then provide another protection ID to the list, which is used to allow access to memory. Since it is possible to attach to a shared memory segment as read only, this protection ID needs to have some way of also controlling write access. The least significant bit of the protection ID is the write enable bit.

Slide: 32-bit and 64-bit Shared Memory Coexistence

32-Bit/64-Bit Shared Memory Coexistence

IPC_SHARE32

Data size /
alignment

must be specified in 64-bit applications which
are to share data with 32-bit applications

must be accounted for when passing data
between 32-bit and 64-bit applications

a696148

Notes:

He zedobasib upo priznepu
gan nix 64/32 u bingabrubarus

Slide: 32-bit and 64-bit Shared Memory Coexistence

When writing 64-bit applications that need to share data with 32-bit applications, the **IPC_SHARE32** flag must be specified in the call to `shmget()`. This allows 32-bit and 64-bit applications to use the same shared memory segments, by ensuring that the memory is placed in the address space accessible to 32-bit applications. The default behavior is for shared memory areas requested by 64-bit processes to be allocated for use by 64-bit processes only.

Slide: Controlling Space Allocation for Shared Memory

Controlling Space Allocation for Shared Memory

System-wide: `shmem` - enable/disable shared memory

`shmuni` - maximum number of shared memory segments (ids) which can exist simultaneously

`shmmax` - maximum size of a single shared memory segment

Per-process: `shmseg` - maximum number of shared memory segments to which a single process can be attached at the same time

a696149

Notes:

Slide: Controlling Space Allocation for Shared Memory

Configurable IPC Shared Memory Parameters

- shmem** - enable or disable shared memory at system boot time. (Series 700 Only).
Acceptable Values:
minimum: 0 (exclude System V IPC shared memory code from the kernel)
maximum: 1 (include System V IPC shared memory code in kernel)
default: 1
shmem determines whether the code for System V IPC shared memory is to be included in the kernel at system boot time.
Series 800 systems: IPC shared memory is always enabled in the kernel.
Series 700 systems: If *shmem* is set to zero, all other IPC shared memory parameters are ignored.
When to Disable Shared Memory
Some subsystems such as Starbase graphics require shared memory. Others such as X Windows use shared memory (often in large amounts) for server- client communication if it is available, or sockets if it is not. If memory space is at a premium and such applications can operate, albeit slower, without shared memory, it may be preferable to run without shared memory enabled.
- shmmax** - maximum allowable shared memory segment size (in bytes).
Acceptable Values:
minimum: 2 Kbytes
maximum: 1TB (in 64-bit mode)
1Gb (in 32-bit mode)
default: 0x04000000 (64 Mbytes)
shmmax defines the system-wide maximum allowable shared memory segment size in bytes. Any *shmget()* system call that requests a segment larger than this limit returns an error.
Setting this value to 0x7fffffff has the effect of disabling this limit, allowing the operating system to impose its own hard limit for shared memory.
Note: although a total of 1.75Gb shared memory is possible in 32-bit mode (2.75Gb with SHMEM_MAGIC), it is not possible for a single shared memory segment to cross a quadrant boundary and hence the maximum segment size is still 1Gb.
- shmmni** - maximum number of shared memory segments allowed on the system at any given time.
Acceptable Values:
minimum: 3
maximum: 1024
default: 200 identifiers
shmmni specifies the maximum number of shared memory segments allowed to exist simultaneously, system-wide. Any *shmget()* system call requesting a new segment when *shmmni* segments already exist will return ENOSPC.

Slide: Controlling Space Allocation for Shared Memory

Setting *shmmni* to an arbitrarily large number wastes memory and can degrade system performance. Setting the value too high on systems with small memory configuration may consume enough memory space that the system cannot boot. A value that is as close to actual system requirements as possible should be selected for optimum memory usage: a value not exceeding 1024 is recommended unless system requirements dictate otherwise.

shmseg - maximum number of shared memory segments to which a single process can be attached simultaneously

Acceptable Values:

minimum: 1

maximum: shmmni

default: 120

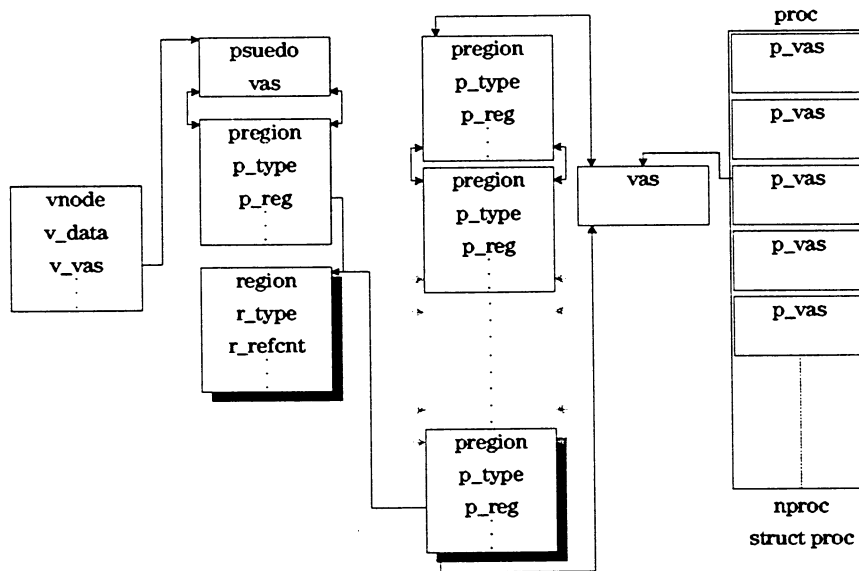
shmseg specifies the maximum number of shared memory segments to which a process can be attached at any given time. Any calls to *shmat()* that would exceed this limit return **EMFILE**.

Module 6 — InterProcess Communication

Left blank intentionally

Slide: Memory Mapped Files

Memory Mapped Files



a696150

Notes:

Точность реализации HP-UX :
 kernel груз. файл по параметру
 с помощью sysctl sysctl sysctl

Slide: Memory Mapped Files

A similar concept in many ways to shared memory is memory mapped files.

Normally to access a file requires the use of system calls for each access with memory mapping, files maybe attached to the address space of a process and subsequently accessed as any other data; that is, directly as a memory address. This can make certain types of access tens of thousands of times faster.

When mapping a file, it can be mapped either shared or private.

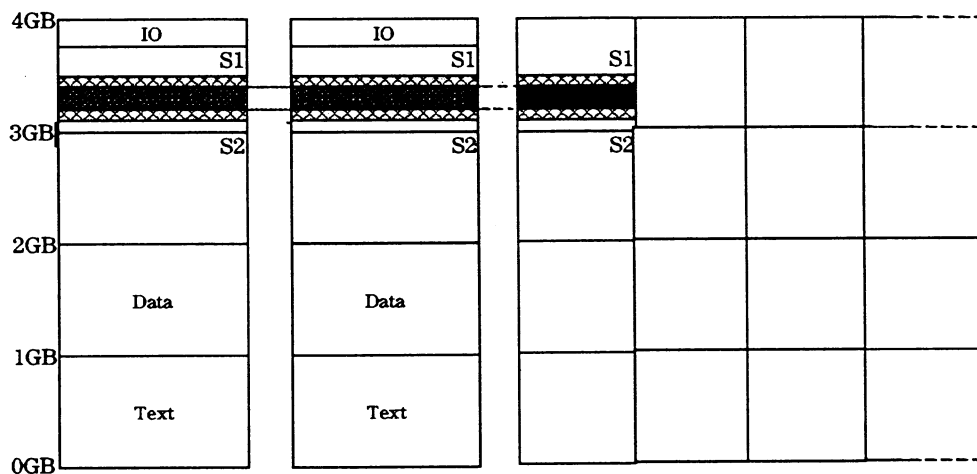
- Private memory mapping, reads the data from the file on the first access (by paging) but any changes that are made in memory are not written back to the file. Private mapping is performed to the private data quadrant of the process, typically quadrant 2 for 32-bit applications.
- Shared mappings both read and write data to the file. Here the section of file is mapped into the shared quadrants of the process (S1 or S2).

It is also possible to use the mmap system call to map an area of address space anonymously (not to a file).

All kernel activity related to files centers around vnode structures. For memory mapped files a pseudo VAS is built upon the v_vas pointer of the files vnode.

Slide: Memory Mapped Files and Global Virtual Address Space

Memory Mapped Files and Global Virtual Address Space



a696151

Notes:

Slide: Memory Mapped Files and Global Virtual Address Space

As with shared memory, when memory mapped files are attached into the address space, they first are attached into the global virtual address space. This means that all processes attaching to shared MMFs do so at the same address.

Although this allows the direct sharing of pointers, there is one draw back. When mapping a file it is not required to map the whole file. A process can subsequently decide to map a large area, and so can re-map the file to access a larger area. Only one process using the area does not cause problems, but if other processes are also sharing this part of the file it may not be possible.

If the area of virtual address space allocated to the initial mapping is surrounded by other segments, it would be necessary to take away the old mapping and then add a new larger mapping at a new location. If however, the current mapping is in use by another process this will not be possible, as all processes must see the area at the same location.

Slide: Signals

Signals

Signals can be generated based on:

- Process termination
- Process induced exceptions
- Unrecoverable error condition during a system call
- User initiated signals
- Kill signal
- Terminal signals
- Signals used to trace a process

a696152

Notes:

Slide: Signals

Signals are a software equivalent to a hardware interrupt. They are used by the kernel for communicating asynchronous events to a process thread. They may also be used for communication between multiple processes using the *kill()* system call and between multiple threads in the same process with the *pthread_kill(3T)* function and its associated kernel routine *__pthread_kill()*¹.

The following are some examples of signal event types:

- Process Termination
- Process induced exceptions
- Unrecoverable error condition during a system call
- User initiated signals
- Kill signal
- Terminal signals
- Signals used to trace a process

If the process is single-threaded, a signal handler can be set up to ignore, hold the signal pending, take the default action or invoke a user signal handler. With a multi-threaded process, it is possible to decide which thread(s) may handle the signals, thus not disturbing the other running threads.

Note:

The following information is based on the Signal Mechanism Internal Maintenance Specification and Kernel Threads Phase II Signals.

1. pthreads/signal.c for the implementation of *__pthread_kill()*

Slide: Signal Delivery Methods

Signal Delivery Methods

kill() system call
interrupt sent from a keyboard

Illegal Instruction
Memory Violations
Floating Point Exceptions

a696153

Notes:

Slide: Signal Delivery Methods

Signals may be divided into two types; asynchronously generated and synchronously generated:

Asynchronously Generated Signals

A signal which is not attributable to a specific thread. Examples are: signals sent via *kill()*, signals sent from the keyboard, and signals delivered to process groups.

Being asynchronous is a property of how the signal was generated and not a property of the signal number. All signals may be generated asynchronously.

← genuine
← generated!

Synchronously Generated Signals

A signal which is attributable to a specific thread. For example, a thread executing an illegal instructions or touching an invalid memory address, causes a synchronously generated signal.

Being synchronous is a property of how the signal was generated and not a property of the signal number.

Signals may be directed at a specific process or group of processes via the *kill()* system call, or via some asynchronous event such as terminal activity.

The POSIX standard requires that signals directed at a process be delivered to exactly one thread which has not blocked delivery of the signal. Which thread actually processes the signal is not specified: if all threads block the signal, the signal should remain pending until a thread unblocks the signal, sets the actions to ignore the signal or issues a call to one of the *sigwait()* functions for that signal.

The POSIX standard also requires that the *sigpending()* functions return signals that are pending for the process and the calling thread; the means it must be possible to distinguish between signals directed at a process and signal directed at a thread.

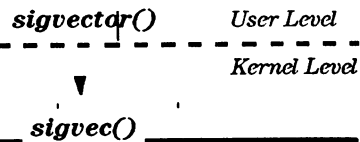
With the introduction of the *sigwait()* function, there is one more factor to be considered when determining the thread to handle a signal which has been directed at a process. Threads in *sigwait()* system calls are given preference over other threads in the process in this case.

HP-UX has extended this functionality to threads waiting in *pause()*, *sigpause()* and *sigsuspend()*.

В некоторых UNIX системах при использовании SIG-IGN приводит к тому, что сигнал не обрабатывается никак с обработкой

Slide: Signal Anticipation: sigvec()

Signal Vector: sigvec()



Check that `signo` is valid.

Read in the new `sigvec` into the kernel local 'vec'.

Set hi-order word of signal-handler mask to that of current mask

Check that the user is not trying to handle SIGKILL or SIGSTOP.

If the user wants the old `sigvec`,

Load up the kernel local 'ovec' with the appropriate fields from `u.u_procp->p_signal, u_procp->p_sighandmask`.

Set `ovec.sv_flags`.

Copyout() `ovec` to the user's address space.

If the user wants to change the `sigvec` for `signo`,

Check that there are no `sig_flags` bits

Translate `SV_BSDSIG` -> `!SA_NOCLDSTOP`.

Call `setsigvec()` to do the work of setting the signal vector

a696154

Notes:

SIGKILL — *нельзя вернуть* *госагора*
SIGSTOP — *нельзя вернуть* *госагора*

Slide: Signal Anticipation: sigvec()

Modifying a Signal Vector

sigvec() - sys/kern_sig.c

A process decides in advance of receiving a signal how it will deal with it. The process may choose to specify a user routine to handle the signal, or it may specify that the signal be blocked or ignored. The default action for the signal may also be explicitly requested. This type action would depend on the signal type. The process tells the kernel how it expects to deal with a particular signal via the *sigvector()* system call.

The *sigvector()* system call enters the system through a stub to the kernel routine *sigvec()*.

Upon entry to *sigvec()*, the kernel verifies that the signal is indeed a valid signal that may have its handler changed. The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored. Attempting to do so will result in **EINVAL**. The range of signals that is valid to change handlers for includes signals numbered 1 through **NSIG- 1**. **NSIG** is one greater than the highest numbered signal that a user may receive.

The *sigvec()* routine copies the new vector information from user space into the kernel before copying the current signal information into the old vector structure. The copying is done in this order because the old and new structure may be one in the same, and this prevents the new information from being overwritten. If the signal being processed is **SIGCLD** and the process flag **SV_BSDSIG** (Berkeley semantics) is currently set for the process, then the **SV_BSDSIG** flag is set in the returned vector flags word.

Before changing the vector, one more test is made for valid vector to modify. This time *sigvec()* checks to see if the user is trying to set the **SIGCONT** handler to **SIG_IGN**.

В kernel routine sigvec() user specifies
signal number and new handler }
in arg (signature in STREAMS)

Slide: Signal Handler: *setsigvec()*

Signal Handler: *setsigvec()*

“Installs the signal handler”

The handler address is copied into the `p_signal[]` array,
and then `p_sighandmask[]` is set according to the user's
request

If the signal handler is for the SIGCLD signal, check to
determine which semantics should be used Berkeley or
AT+T.

If an alternate stack has been requested, a bit within the
`u_sigonstack` field is set.

a696155

Notes:

Slide: Signal Handler: setsigvec()

Installing the Handler

sigsetvec() - sys/kern_sig.c

The signal vector must be changed atomically, so the *sched_lock* spinlock is obtained on entry into *setsigvec()*. The requested signal handler and mask entries for the requested signal are copied from the *sigvec* structure into the *p_signal[]* and *p_sighandmask[]* entries in the process structure. An important note is that the signal numbers begin with the number one, so a value of 1 must be subtracted from the signal number when referencing the arrays, which begin with zero.

If the signal vector being changed corresponds to the **SIGCLD** signal, then a test is made to see whether or not to use the Berkeley semantics in handling the signal. Additionally the process is sent a **SIGCLD** signal if it has any zombie children. This is done because *sigvector()* system call semantics say that such a signal will be sent if it is caught. The bit tested to check for Berkeley semantics comes from the *sv_flag* field of the *sigvec[]* structure. The same word is also used to set a bit indicating whether or not a signal should be handled on the alternate signal stack.

The example program below uses the *sigvector()* system call to associate the **SIGINT** signal with the user handler routine *sig_handler()*. The *sigvector* system call enters the system through a stub call to the kernel routine *sigvec()*.

```
#include /usr/include/sys/signal.h

sig_handler(sig,code,scp)
init sig, code;
struct sigcontext *scp
{
    if (scp>sc_syscall == SYS_NOTSYSCALL)
        printf("Received signal number %d \n", sig);
    else
        printf("Interrupt a system call \n");
}

main()
{
    struct sigvec vec;
    vec.sv_handler = sig_handler; /* specify the desired handler */
    vec.sv_mask = 0;
    vec.sv_onstack = 0;
    sigvector(SIGINT,&vec,0); /* set up the handler for SIGINT */
    kill(getpid(),SIGINT); /* send ourself a SIGINT */
}
```

— can be manually
overriden

In this example, *setsigvec()* causes the *p_signal[sig-1]* field of the proc structure to be set with the address of *sig_handler()* and the *p_sighandmask[sig-1]* field is set to zero. The appropriate bit in the *u_sigonstack* field is cleared, since the program does not intend a special signal stack to be used for handling this signal.

Slide: Signal Anticipation: Proc and Thread Adjustment

Signal Anticipation: Proc and Thread Adjustment

The proc structure fields `p_sigcatch` and `p_sigignore` are adjusted meaning a signal handler has been installed, or ignored..
The kthread structure `kt_sig` contains signal's bit vector.

`kt_sig`

`p_sigignore`

`p_sigcatch`

a606156

Notes:

Slide: Signal Anticipation: Proc and Thread Adjustment

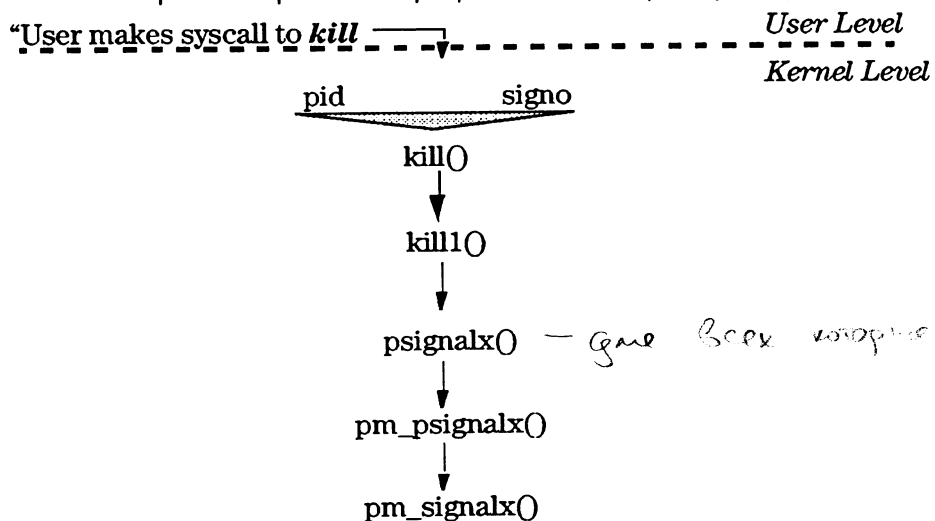
Adjusting the Proc and Thread Structure

There are three fields of special interest in the thread and proc structure when a signal's vector is changed. The first of the three is the *kt_sig*; the signal's bit vector is set in this word when a signal is set. If the handler is being set to ignore this signal then the appropriate bit in this word is set to zero to clear the pending signals of this type. The *p_sigignore* field indicates whether or not a particular signal is being ignored, and it is generally used as a mask to clear out all "uninteresting" signals. The *p_sigcatch* field is almost the opposite of *p_sigignore*. It is set when a signal is being caught by an installed signal handler. It is not exactly the opposite because the signal can be using the default actions, indicated by **SIG_DFL** in which case it is neither ignored nor caught. These fields are modified by *setsigvec()* and are used frequently during the signal causation and detection.

Referring to the example program, bit position *sig-1* of the *p_sigcatch* field is set, and the corresponding field is cleared in *p_sigignore*, by the call to *sigvector()*. There is no change to the *kt_sig* field until the *kill()* call.

Slide: Signal Causation: kill()

Signal Causation: kill()



a696157

Notes:

Slide: Signal Causation: kill()

Signal Causation

A user process “causes” a signal to be sent to itself or another process by executing the **kill()** system call.

kill() accepts two arguments:

- Pid of the process to receive the signal
- Signal number

kill() checks that pid 1 (init) isn't being sent sigkill or sigstop. It also checks that to see if **KILL_ALL_OTHERS** is being sent and if so, sets a flag.

kill1() looks at the parameter flags to decide whether to send the signal to a single specified process, the other processes in the same process group, or all non-system processes.

In the ordinary case, the process id specifies a single process to be sent a signal. The routine calls **psignalx()** to do the signal delivery.

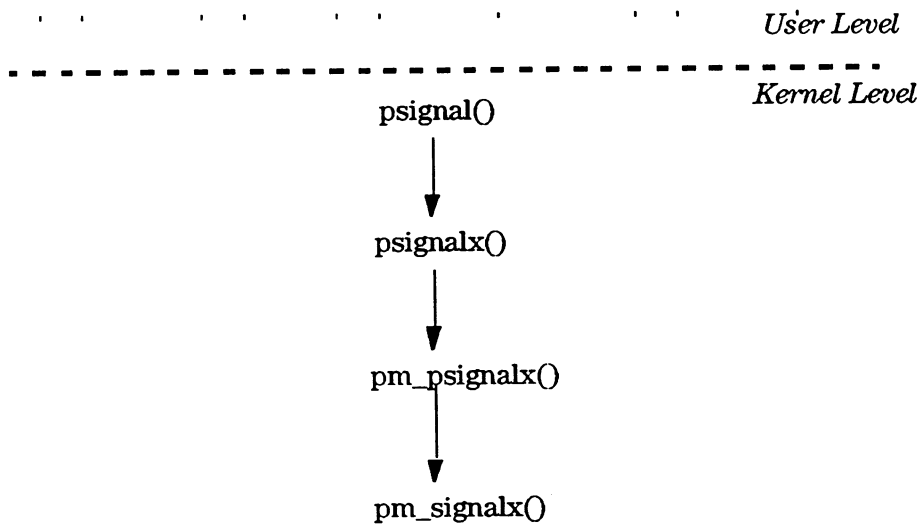
Another possibility is that the signal is supposed to be delivered to the user's entire process group. This is accomplished by following the process group hash chain to its end looking for all of the processes that have the same group id. When one is found, **pm_psignalx()** is called to actually signal the process.

The other case is that the entire process table of active entries must be scanned to locate the processes to signal. This happens when the process requests that either all processes or all processes with the same user id has the caller be signaled. Once again both of these instances are subject to having the appropriate user ids. If the **KILL_ALL_OTHERS** flag is set for these cases, then the calling process will not be signaled.

In the example program, **kill()** is called with the simplest set of arguments (the calling process' pid), which causes the signal **SIGINT** to be sent to that process. **kill1()** then calls the workhorse routine **psignal()** to handle delivery of the signal.

Slide: Signal Delivery: psignal()

Signal Delivery: psignal()



a696158

Notes:

Slide: Signal Delivery: `psignal()`

Now that we have threads to consider, the lower level signal routines have all been changed.

Although signal delivery can now be thread based, the old process based signal interfaces must still be supported. This was achieved by making the high level process based signal routines simply by wrappers for thread based signal routines. The process based wrapper routines can pass a pointer to the first thread in the process to allow low level thread based signal routine.

For example, the routine `ps_psignalx` is now a wrapper routine, which performs some locking operations then passes `p->pfirsthreadp` to the workhorse routine `pm_signalp()`.

Signal Delivery

The routine `psignal()` is called to actually deliver a signal to the process itself. The receiving process will not respond directly to the delivery, but it may have its state changed to allow the signal to be correctly dealt with. However, `psignal()` and all of the higher level routines just perform error checking and locking. No actual work is done, until it gets into `pm_signalp()`.

The `pm_signalx()` routine has become very complicated due to threads. This routine handles posting the signal to a process and to a thread.

Based on the signal action for the signal, it will ignore the signal, process the signal or post the signal to the pending signal mask and either return (for blocked signals) or set a thread running so it can be handled by `issig()` and/or `psig()`.

Also, before any action is taken the process' `p_flag` is checked to see if the parent process is `ptrace`'ing. If so, the receiving process will be setup to perform the default action of the signal so that the process will not be able to ignore it. This will allow the parent process to trace its child and tell it how to deal with the signal.

The action that a process/thread will take after detecting the signal is defined to be one of four choices :-

SIG_IGN	Ignore the signal. If it is determined that the process would ignore this signal, then return without any action.
SIG_HOLD	Hold the signal.
SIG_CATCH	Catch the signal.
SIG_DFL	Performs the default action associated with the signal, irrespective of the state of the process or thread.

Signal Delivery

The `pm_signalx()` function has an extra parameter, `is_proc_sig`: this indicates if the signal was originally sent to the process, rather than a specific thread.

Slide: Signal Delivery: psignal()

If a signal is a **SIGKILL** or **SIGSTOP** and the signal was directed at a thread, post the signal to *p_sig* rather than *kt_sig*. Additionally, if the signal is **SIGCONT** and was directed at a thread which doesn't block the signal, the signal is posted to *p_sig*.

The function then calls *find_sigwaiters()* to check to see if any thread is waiting for a signal in *sigwait()*, *pause()*, *sigpause()* or *sigsuspend()*. If a thread is found to be waiting for this signal in *sigwait()*, it is removed from the *p_sigwaiter* list and woken up. Otherwise, if a thread is found waiting for one of the other routines for this signal, the signal is dispatched to that thread; the thread is not removed from the *p_sigwaiter* list in this case until it is certain that the signal will cause the thread to abort its sleep (i.e. signals that are ignored won't cause the thread to abort its sleep).

If the signal was posted to a process, *psignal_find_thread_for_sig()* is called to try to find a thread to handle the signal: this returns **NULL** if no suitable thread is found to handle the signal (e.g. all the threads block the signal or a process wide operation was in progress), or if a running thread in the process was found which will handle the signal. If **NULL** is returned and the signal is not ignored, it is posted to *p_sig* and we bail out. If there were running threads in the process then one of them will either process the signal or dispatch it to another thread in the process via *issig()*. If *psignal_find_thread_for_sig()* finds a suitable thread to handle the signal (e.g. sleeping and interruptible and doesn't block the signal) it returns a pointer to the thread which it wants to handle the signal.

The following is a general overview of signal processing:

Process thread is niced to zero to assure prompt handling:

SIGTERM
SIGKILL

Zero the signal bits in the *kt_sig* vector that stops the process thread:

SIGCONT

Turn off the **SIGCONT** signal bit:

SIGSTOP
SIGTSTP
SIGTTIN
SIGTTOU

In the example program, a signal handler is set up to catch the **SIGINT** signal and no general processing is performed.

Additional processing is dependent on the process state. If the action associated with the signal is other than **SIG_DFL**, then the process thread is made runnable. In cases where the default action is specified for a signal and the process thread is sleeping, then the following signal processing occurs:

Slide: Signal Delivery: `psignal()`

If the process is in a `vfork()`, no additional processing is performed, otherwise the current signal (`kt_cursig`) is set and the process thread is stopped:

SIGSTOP
SIGTSTP
SIGTTIN
SIGTTOU

The signal is thrown away:

SIGIO
SIGIURG
SIGCLD
SIGPWR
SIGWINDOW

The final case is if the process thread is stopped. The following processing is done regardless of the signal action:

Make the process thread runnable
SIGKILL

If using Berkeley semantics, the parent is sent **SIGCLD**. If the process thread is waiting in an event, it is set to a sleeping state. Otherwise it is made runnable.
SIGCONT

Ignore these signals
SIGSTOP
SIGTSTP
SIGTTIN
SIGTTOU

Process thread remains in the stopped state, but it is unstuck if sleeping. This allows the process thread to run immediately when it is continued.

Slide: Signal Recognition: *issig()*

Signal Recognition: *issig()*

We check for signals:

In the high level system call code after the call has been performed

After handling a trap

issig() loops until:

There are no more signals to handle

An attempt is made to handle a signal for a system process

A signal is of the SIG_CATCH type

a696150

Notes:

Slide: Signal Recognition: `issig()`

Signal Reception

`issig()` - `sys/pm_signal.c`

`issig()` returns 0, if there are no signals, a positive number denotes which signal and a negative number denotes that this thread is exiting so pass the word back up.

The function `issig()` is called in both of the cases mentioned above to test whether or not a process has any signals pending. It also provides some very preliminary processing for signals, which have the default action set.

The operation of `issig()` is simply a loop which scans through the current threads' `kt_sig` vector looking for pending signals. If a signal handler has been set for a pending signal, then `issig()` returns the signal number in `kt_cursig`. No processing is done for ignored signals, with the exception of `SIGCLD`. An ignored `SIGCLD` signal results in any zombie child processes being freed.

For some signals being handled by default, the complete processing is actually handled here, rather than in `psig()`. Signals that have a default action to be ignored are discarded at this point. All other signal numbers are returned by `issig()` for further processing by `psig()`.

In the example program, `issig()` is called on the return path from the `kill()` system call. It results in `kt_cursig` being set to `SIGINIT` for the process' thread.

Slide: Signal Handling: *psig()*

Signal Handling: *psig()*

Consistency check to make sure signal number is positive. If not
panic

Determine action of signal

Call *sendsig()* routine to set up user's signal handler (if needed).

a696160

Notes:

Slide: Signal Handling: `psig()`

Signal Handling

`psig()` - `sys/pm_signal.c`

Running `psig()` would trivially be done:

```
if ( issig() )
    psig()
```

In practice, `issig()` is invoked using the macros `ISSIG()`, `ISSIG_SLEEP()` and `ISSIG_EXIT()`: only `ISSIG_EXIT()` actually calls `psig()`.

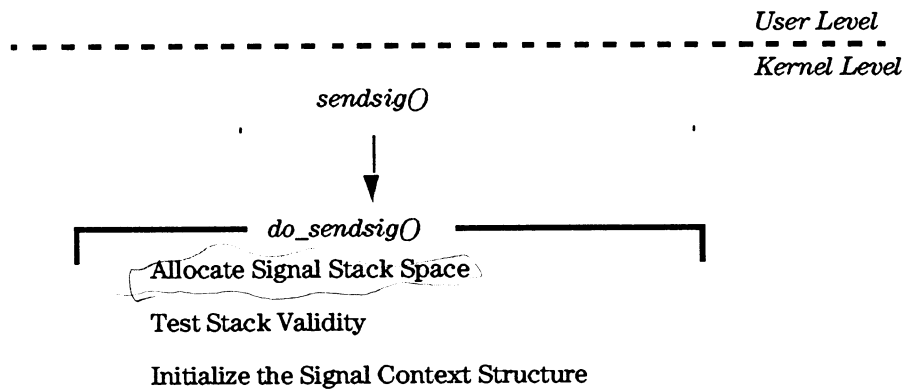
`psig()` starts out with a consistency check which verifies that the signal being handled is a positive number. If it's negative, then the system panics.

If the action to be taken for the signal is `SIG_DFL`, `exit()` will be called, causing the process to be terminated. The default action for the `SIGKILL`, `SIGIOT`, `SIGBUS`, `SIGQUIT`, `SIGTRAP`, `SIGEMT`, `SIGFPE`, `SIGSEGV`, `SIGSYS` signals is for `core()` to be invoked to dump a core file of the process.

The majority of `psig()` is designed to handle the specific case that the user has set up their own signal handler, the non-default case. This code begins with another consistency test. However instead of panicing as was the case in earlier releases of HP-UX, it is now ignored.

Slide: Signal Handler Commencement: *send_sig()*

Signal Handler Commencement:
send_sig()



a696161

Notes:

После обработки сигнала,
не сразу в процесс, а раньше
обработ. сигнала
(longjmp генерируется там signals
sigcleaning?)

Slide: Signal Handler Commencement: `sendsig()`

`sendsig()` - *mach.800/pm_sendsig.c*

`sendsig()` launches a user signal handler by building a user stack so that the previous user state can be preserved. The function `do_sendsig()` performs all the work. Preserving the context is accomplished in several steps:

- **Allocate Signal Stack Space**

A test is made to see whether signals are being handled on top of the user's normal stack or a special signal stack. If the regular stack is being used, then `grow()` is called to extend the stack if it is not large enough to hold the context information. A flag is set to record that a special signal stack is being used if this is the case.

- **Test Stack Validity**

If the user process has thrashed their stack by setting its stack pointer to an erroneous value that it does not have permissions to access, then the process is sent an illegal instruction signal to terminate. Precautions are taken to prevent the user from catching or ignoring the signal.

- **Initialize the Signal Context Structure**

The `sigcontext` struct is built off the stack pointer. It gets initialized with the on stack indication, signal mask, save state, and system call recovery action when appropriate. The user state is copied from the base of the kernel stack to the local frame of the `sigcontext` structure. In the case of a system call, many of the copied registers may not have been stored there, so they would not have to be copied. Currently the code does not make any such distinction. Also if the signal was discovered during the course of a system call, the system call restart info and number are saved in the `sigcontext` struct. If the signal is not recorded as being discovered in either the trap or system call code, then a panic occurs.

A procedure call to the user signal handler is constructed in the save state structure. This will cause the system call or trap code to effectively make the call to the handler.

The save state information on the stack is modified by:

- Clearing the floating point state
- Passing Arguments to the Handler

The `SS_ARGSVVALID` flag gets set indicating that the argument register needs to be restored from the save state structure. Then the argument register fields are filled in with the signal number, code, and

Slide: Signal Handler Commencement: `send_sig()`

sigcontext pointer.

- **Initialized Save State Registers**

The stack pointer is set to reflect the user's current top of stack, and the return pointer is set to `p_sigreturn` which was previously set during the `sigvector()` call.

- **The PC is coerced to the user handler address.**

For a signal discovered during a trap, this requires modifying the saved PC queue. A system call simply sets the link address in the saved register 31 to the handler value.

Module 6 — InterProcess Communication

Left blank intentionally

Slide: Signal Handler Return: *sigcleanup()*

Signal Handler Return: `sigcleanup ()`

- Probe Save State Structure Accessibility
- Copy Save State Structure From User Space into the Kernel
- Restore the Previous on Stack Indicator
- Restore the Signal Mask
- Restore the Trap Service State
- Restore the System Call State

a696162

Notes:

Slide: Signal Handler Return: sigcleanup()

sigcleanup() - *mach.800/pm_sendsig.c*

Handler Return

As discussed in the Signal Handler Commencement on the exit of the signal handler, the user context must be restored after the signal handler is exited. Furthermore, the appropriate signal mask and on the stack indicator must be set, so this requires reentering the system. In the case of signals which were discovered during the process of a trap, an **RFI** must be executed to correctly restore the state.

Signal Handler Exit

Upon exit from the signal handler, the address returned to is the one which has been set in the return pointer field of the signal context structure. Since **sendsig()** set this to the value stored in the *p_sigreturn* field of the proc structure, it will go to the library signal return routine. This field was the one filled in during the first *sigvector* call.

The library signal handler return routine is *_sigreturn* which is simply a stub system call to the *sigcleanup()* kernel routine. The high level system call initialization routine, *syscall()*, intercepts this particular call before most of the system call processing.

The *sigcleanup()* functions by doing the following:

- **Probe Save State Structure Accessibility**

The accessibility of the signal context structure in the user space is probed to make sure that the stack pointer really points to an appropriate address

- **Copy the Save State Structure From User Space into the Kernel**

- **Restore the Previous on Stack Indicator**

The on stack indicator is copied from the signal context struct in the *u_sigonstack* user struct field.

- **Restore the Signal Mask**

The previous signal mask is copied out of the signal context structure. The **SIGKILL**, **SIGCONT**, and **SIGSTOP** signals are all masked off to ensure

Slide: Signal Handler Return: sigcleanup()

the user cannot change mask settings on these while the user signal handler is executing.

- **Restore Trap Service State (when appropriate)**

If the signal context flags indicate that the signal was discovered during a trap, then a few updates to the kernel save state structure are performed. The user privilege bits are masked off in the PC queue to prevent the user from attempting to elevate their privilege level. The save state flag which indicates that the process is in a trap is cleared, and the flag to cause the process thread to exit through the system call RFI path is turned on. This will cause the execution to proceed through the *dorfi* label when existing through the system call code.

- **Restore the System Call State (when appropriate)**

If the signal was not originally discovered during execution of a trap, then it must have been during a system call. In this case, the state must be fixed to allow the process to correctly deal with a possibly uncompleted system call. First, the user privilege bits are ORed into the external link register to prevent the user from attempting to elevate their privilege level during return from the system call. Then the signal context structure is examined to find out how the user wishes the system call to be completed. If the signal context structure field *sc_eosys* is set to **EOSYS_INTERRUPTED**, then the system call was one that was actually interrupted during its execution. If the user has set the *sc_syscall_action* field of the signal context structure to anything except **SIG_RETURN**, the system call will be reentered. The arguments are recopied into the user structure from the user context, and the **EOSYS_RESTART** flag is set in the user structure, and this will cause the high level system call initialization code to attempt the system call again.

Otherwise it is a system call that should not be restarted. An interrupted system call that is not being restarted causes the system call to return an error indication with *errno* being set to **EINTR**. A non-interrupted system call simply has its return values restored from the context structure into the user structure. From here execution proceeds back through the normal system call routine.

Module 6 — InterProcess Communication

Left blank intentionally

Lab: Signals

The lab files are in the directory.

The program `vfork`, is a very simple example it just comprises of

```
main()
{
    vfork();
    pause();
}
```

The call to `vfork` creates a new child process which will then use its parent virtual address space. Since the parent's address space is now in use by the child the parent can not run, and consequently is put to sleep. An un-interruptible sleep is used to ensure that the parent really does not try to run. This allows us to look at the process and thread structures as signals are sent.

Run `vfork` in the background and note its process id.

Then using Q4 load the process table and keep just the parent process.

```
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
q4> keep p_stat && p_pid == <<YOUR PID>>
kept 1 of 276 struct proc's, discarded 275
q4>
```

Then load the thread structure.

```
q4> load struct kthread from p_firstthreadp
loaded 1 struct kthread as an array (stopped by max count)
```

and print the signal fields

```
q4> print -tx | grep sig
```

What are the value of the `kt_sig` fields?

Now kill this parent process and reload the thread structure.

```
q4> forget 0
q4> load struct kthread from p_firstthreadp
loaded 1 struct kthread as an array (stopped by max count)
```

Now what values are in `kt_sig`?

Is this what you would expect?

Were any additional signals delivered to the process and if so which ones?

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

The lab files are in `/home/tops/labs/mod6`.

`mserver` creates a small buffer of shared memory, and a set of two semaphores, which it uses as a binary exclusive lock (write lock) and a counter (read lock). `mserver` then prints out the ids of these two and exits, that was all it's job in life, the real work is done in `mclient`.

`mlook` reads the area of shared memory without worrying about locks and things. It also reports the current state of the two semaphores.

`mclient` prompts the user as to whether they would like a read or write lock. For read it then attempt to acquire the read lock and display the message in the buffer. For writes it attempts to acquire the write lock and then prompt for a message. In both cases `mclient` then holds the lock until you hit return.

Compile the programs with
`make mserver mlook mclient`

(There is no need for a makefile in such a simple case)

Run `mserver` and set up a buffer and the semaphores, and then create at least 4 windows to run the other programs, use one to run `mlook`, and the others to run `mclient`. Have a play with the locking and get to a situation where two processes hold read locks and one is waiting for a write lock.

Then using Q4 look at the IPC data structures.

Firstly look at the shared memory areas. Using the data structures find the process IDs of the processes using out shared memory segment. The `mserver` program will have given the ID's for the shared memory segment and semaphore set, using `ipcs` just check this out.

```
root@tiger[mod6] ipcs
IPC status from /dev/kmem as of Mon Jan 25 15:15:37 1999
T      ID      KEY          MODE          OWNER      GROUP
.
.
.
Shared Memory:
.
.
.
m      7008 0x6d140296 --rw-rw----   ken      users
Semaphores:
.
.
.
s      17 0x73140296 --ra-ra----   ken      users
```

Lab: Shared Memory and Semaphores()

In order to read the shared memory tables you need to know the shared memory kernel parameters. Unlike parameters like `nproc`, the shared memory ones are not directly variables in the kernel, but instead are held in a data structure called `shminfo`. This can be seen from `/usr/conf/space.h.d/core-hpux.h`

```
q4> load struct shminfo from &shminfo
loaded 1 struct shminfo as an array (stopped by max count)
```

What is the value of `shmmni` on your system ?

```
q4> print -t
indexof  0
mapped   1
spaceof  0
addr     7016000
physaddr 7016000
realmode  0
shmmax   67108864
shmmin   1
shmmni   200
shmseg   120
q4>
```

Since this is a single structure its members can be reference by `q4` whilst it is on the top of the stack, so the `shmid` table can now be loaded.

```
q4> load struct shmid_ds from &shmem max shmmni
```

However since we only want to read the entry for our shared memory segment it can be loaded directly.

```
q4> load struct shmid_ds from &shmem skip <<your shmid>>%shmmni
loaded 1 struct shmid_ds as an array (stopped by max count)
q4>
```

Lab: Shared Memory and Semaphores

```
q4> print -t
  indexof 0
  mapped 1
  spaceof 0
  addrof 8204264
  physaddrof 8204264
  realmode 0
shm_perm.uid 101
shm_perm.gid 20
shm_perm.cuid 101
shm_perm.cgid 20
shm_perm.mode 33200
shm_perm.seq 35
shm_perm.key 1830027926
shm_perm.ndx 0
shm_perm.wait 0
shm_perm.pad ""
shm_segsz 1008
shm_vas 4301158400
shm_lpid 4091
shm_cpid 3914
shm_nattch 3
shm_cnattch 1
shm_atime 917303130
shm_dtime 0
shm_ctime 917277074
shm_pad ""
q4>
```

How many process are attached to your segment, shm_nattch, gives this figure ?

From the shmids_ds structure the shared memory segments virtual address space can be loaded.

```
q4> load struct vas from shm_vas
loaded 1 struct vas as an array (stopped by max count)
```

Why is this VAS called a pseudo, what field hints at this ?

What was type of magic number did the creating program use ?

Is this VAS for a 32bit or 64bit environment?

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

```
q4> print -tx
      indexof 0
      mapped 0x1
      sizeof 0
      addrof 0x1005e7800
      physaddrof 0xa29f800
      realmode 0
va_ll.lle_next[0] 0x1008f6c00
va_ll.lle_next[1] 0xffffffffffffffff
va_ll.lle_next[2] 0xffffffffffffffff
va_ll.lle_next[3] 0xffffffffffffffff
va_ll.lle_prev 0x1002f6c00
va_hilevel 0
va_pslpath[0] 0x1005e7800
va_pslpath[1] 0x1005e7800
va_pslpath[2] 0x1005e7800
va_pslpath[3] 0x1005e7800
va_refcnt 0x1
va_rss 0
va_prss 0
va_dprss 0
va_proc 0
      va flags VA PSEUDO
      va_fp 0
      va_wcount 0
va_vaslock.interlock 0x16399c0
va_vaslock.delay 0x126a0
va_vaslock.read_count 0
va_vaslock.want_write 0
va_vaslock.want_upgrade 0
va_vaslock.waiting 0
va_vaslock.no_swap 0x1
va_cred 0
va_hdl.hdl_textsid 0
va_hdl.hdl_textpid 0
va_hdl.hdl_datasid 0
va_hdl.hdl_datapid 0
va_hdl.v_hdlflags 0
va_ki_vss 0
va_ki_flag 0
va_ucount 0x1
va_runenv.r_machine R_PARISC
va_runenv.r_arch R_PARISC_2_0
va_runenv.r_os R_HPUX_OS
      va_runenv.r asmodel R 64BIT
      va_runenv.r magic R SHARE MAGIC
va_lgpg_env.lgpg_data_size 0
va_lgpg_env.lgpg_text_size 0
va_lgpg_env.lgpg_next_brk_inc 0
va_lgpg_env.lgpg_user_brk_cnt 0
va_lgpg_env.lgpg_next_stk_inc 0
      va_winId 0xffffffff
      va_winMap 0xfeb880

q4>
```


Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

The VAS for a shared memory segment will only have one pregon, load this

```
q4> load struct pregon from va_11.11e_prev
loaded 1 struct pregon as an array (stopped by max count)
q4>
```

From this pregon

What flags are set ?

What is the space and offset?

What address is this pregon stored at?

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores)

```
q4> print -tx
      indexof  0
      mapped   0x1
      spaceof  0
      addrrof  0x1008f6c00
      physaddrf 0x3af6c00
      realmode  0
p_ll.lle_next[0] 0x1005e7800
p_ll.lle_next[1] 0xffffffffffffffff
p_ll.lle_next[2] 0xffffffffffffffff
p_ll.lle_next[3] 0xffffffffffffffff
p_ll.lle_prev  0x1005e7800
      p flags  PF SHARE32|PF PSEUDO|PF ACTIVE|PF ALLOC
      p_type   PT_SHMEM
      p_reg    0x100f9ac00
      p space  0xc0eb000
      p vaddr  0x81673000
      p_off    0
      p_count  0x1
      p_ageremain 0
      p_agescan 0
      p_stealscan 0
      p_vas    0x1005e7800
      p_tid    0x1
      p_forw   0x100bac000
      p_back   0x100224900
      p_regsknode 0x100ea7440
p_nestediomaps  0
      p_pagein 0x8
      p_bestnice 0x27
      p_deactsleep 0
      p_locality 0
      p_strength 0
      p_prot    0xb
      p_nextfault 0xffffffffffffffff
      p_last_rhead 0xffffffffffffffff
      p_mq_refcnt 0
      p_lchain  0
      p_lactions 0
p_hdl.hdlflags  0x1
      p_hdl.hdlar 0x3f
      p_hdl.hdlprot 0x41be
      p_hdl.p_spreg 0
q4>
```

Lab: Shared Memory and Semaphores

The pregion points to the system wide region structure, and from there, there is a list of all the pregions sharing the same region, (in our case this will include the pregions from the processes using the shared memory segment).

Under the original HP-UX 10.20 release this was a simple set of linked list pointers, starting with the field r_pregs in the region and then walking the list using the fields p_prpNext p_prpprev in the pregions. Unfortunately although that is what is described in the header files and even in the output of Q4 it is not what actually happens a patch (PHKL_9075) changes the kernels implementation to use skip lists, but the header files and the Q4 debug information have not been updated to match this.

These data structures are correctly defined for HP-UX 11

The the region structure.

```
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4>
```

What is the reference count on this region?

What is the value of r_psklh.l_header.n_next[0] ? and are there any other skip-list levels in use ?

What is the value of r_psklh.l_tail ?

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

```
q4> print -tx
      indexof 0
      mapped 0x1
      spaceof 0
      addrof 0x100f9ac00
      physaddrof 0xb134c00
      realmode 0
      r_flags RF_ALLOC
      r_type RT_SHARED
      r_pgsz 0x1
      r_nvalid 0x1
      r_dnvalid 0
      r_swalloc 0x1
      r_swapmem 0
      r_vfd_swapmem 0
      r_lockmem 0
      r_pswapf 0
      r_pswapb 0


---


      r_refcnt 0x4
      r_zomb 0
      r_off 0
      r_incore 0x4
      r_dbd 0
      r_scan 0
      r_fstore 0
      r_bstore 0x1001c1b00
      r_forw 0x1001f7a00
      r_back 0x100c26600
      r_hchain 0
      r_byte 0
      r_bytelen 0
      r_lock.interlock 0x1639180
      r_lock.delay 0
      r_lock.read_count 0
      r_lock.want_write 0
      r_lock.want_upgrade 0
      r_lock.waiting 0
      r_lock.no_swap 0x1
      r_mlock.b_lock 0
      r_mlock.order 0x5a
      r_mlock.owner 0
      r_poip 0
      r_root 0x101257e80
      r_key 0
      r_chunk 0x100fdb00
      r_next 0x100f9ac00
      r_prev 0x100f9ac00
      r_preg_un.r_un_pregskl 0x100f9ace8
      r_preg_un.r_un_pregion 0x100f9ace8


---


r_psklh.l_header.n_next[0] 0x101354000
      r_psklh.l_header.n_next[1] 0
      r_psklh.l_header.n_next[2] 0
```

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores)

```
r_psklh.l_header.n_next[3] 0
  r_psklh.l_header.n_prev 0
    r_psklh.l_header.n_key 0
  r_psklh.l_header.n_value 0
  r_psklh.l_header.n_flags 0
  r_psklh.l_header.n_cookie 0
  r_psklh.l_tail 0x100ea7440
  r_psklh.l_cache 0
  r_psklh.l_cmpf 0
  r_psklh.l_level 0
  r_psklh.l_cookie 0
    r_excproc 0
    r_lchain 0
    r_mlockswap 0
    r_pgszhint 0x1
  r_hdl.r_space 0
  r_hdl.r_prot 0x41be
  r_hdl.r_vaddr 0x81673000
  r_hdl.r_hdlflags 0x10
```

q4>

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

As you may remember from when looking at the skip lists used with the pregrions there is a problem with Q4 following linked list pointers in arrays, so as with the pregrion lists this one needs to be walked backward, using the tail field.

The skip lists is built using a structure typedef'd as sknode_t, which header file is this defined in ? What field in this structure is the pointer to the previous entry in the linked list ?

Now load the linked list of sknode_t's

```
q4>
q4> load sknode_t from r_psklh.l_tail next n_prev max 100
loaded 5 sknode_t's as a linked list (stopped by null pointer)
```

The field n_value holds the pointers to the pregrions in this example. Unfortunately the a set of data structures can not be loaded by following a set of pointers in Q4 without resorting to programming it in Perl. So to load the pregrions that share this region each of them needs to be loaded separately.

```
q4> print -x n_prev n_next[0] n_flags n_value
      n_prev n_next[0] n_flags      n_value
0x101354300      0      0x3 0x1008f6c00
0x101354080 0x100ea7440 0x3 0x101356000
0x101354000 0x101354300 0x3 0x101352c00
0x100f9ace8 0x101354080 0x3 0x101358700
0           0x101354000      0           0
```

At the end of this list is the skip list header (remember we loaded it backwards)

```
q4>
q4> load struct pregrion from 0x1008f6c00
loaded 1 struct pregrion as an array (stopped by max count)
q4> load struct pregrion from 0x101356000
loaded 1 struct pregrion as an array (stopped by max count)
q4> load struct pregrion from 0x101352c00
loaded 1 struct pregrion as an array (stopped by max count)
q4> load struct pregrion from 0x101358700
loaded 1 struct pregrion as an array (stopped by max count)
q4>
q4>
```

Once the pregrions have been loaded then they can be combined together into a single Q4 pile to make them a little more manageable.

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

```
q4> history
.
.
.
30 <none> array      1 struct shminfo  stopped by max count
31 <none> array      1 struct shmid_ds  stopped by max count
32 <none> array      1 struct vas       stopped by max count
33 <none> array      1 struct pregion   stopped by max count
34 <none> array      1 struct region    stopped by max count
35 <none> list       5 sknode_t        stopped by null pointer
36 <none> array      1 struct pregion   stopped by max count
37 <none> array      1 struct pregion   stopped by max count
38 <none> array      1 struct pregion   stopped by max count
39 <none> array      1 struct pregion   stopped by max count
q4>
```

```
q4> merge 38
merged 1 item with 1 item making 2 items
q4> merge 37
merged 1 item with 2 items making 3 items
q4> merge 36
merged 1 item with 3 items making 4 items
```

The list of pregions can now be viewed.

Looking at the pregions what is interesting about the p_space, p_vaddr and p_reg fields ?

Is this what you would expect?

By looking at flags which of the pregions belongs to the VAS on the shared memory segment and which belong to the processes?

```
q4> print -x p_type p_space p_vaddr p_count %d p_reg p_flags
p_type p_space p_vaddr p_count p_reg p_flags
PT_SHMEM 0xc0eb000 0x81673000 1 0x100f9ac00 PF_SHARE32|PF_PSEUDO|PF_ACTIVE|PF_ALLOC SHMEM
PT_SHMEM 0xc0eb000 0x81673000 1 0x100f9ac00 PF_NOPAGE|PF_ALLOC Process
PT_SHMEM 0xc0eb000 0x81673000 1 0x100f9ac00 PF_NOPAGE|PF_ALLOC Process
PT_SHMEM 0xc0eb000 0x81673000 1 0x100f9ac00 PF_NOPAGE|PF_ALLOC Process
q4>
```

Load the vas structures for the process related pregions, again there is no easy way to do this, they need to be loaded one by one and then combined if needed.

Lab: Shared Memory and Semaphores)

```
q4> print -x p_vas p_flags
      p_vas p_flags
0x1005e7800 PF_SHARE32|PF_PSEUDO|PF_ACTIVE|PF_ALLOC
0x101356c00 PF_NOPAGE|PF_ALLOC
0x101348800 PF_NOPAGE|PF_ALLOC
0x101352600 PF_NOPAGE|PF_ALLOC
q4>

q4>
q4> load struct vas from 0x101356c00
loaded 1 struct vas as an array (stopped by max count)
q4> load struct vas from 0x101348800
loaded 1 struct vas as an array (stopped by max count)
q4> load struct vas from 0x101352600
loaded 1 struct vas as an array (stopped by max count)
q4>

q4> history
.
.
.  42 <none> mixed?      4 struct pregion  merge of 36 and 41
   43 <none> array      1 struct vas     stopped by max count
   44 <none> array      1 struct vas     stopped by max count
   45 <none> array      1 struct vas     stopped by max count
q4>
q4>
q4>
q4> merge 44
merged 1 item with 1 item making 2 items
q4> merge 43
merged 1 item with 2 items making 3 items
q4>
```

Once the VAS structures are loaded then they have a reference to their processes table entries. So the proc structures can then be loaded.

```
q4> print -x va_proc
      va_proc
0xed8380
0xed8740
0xed8b00
q4>
q4> load struct proc from 0xed8380
loaded 1 struct proc as an array (stopped by max count)
q4> load struct proc from 0xed8740
loaded 1 struct proc as an array (stopped by max count)
q4> load struct proc from 0xed8b00
loaded 1 struct proc as an array (stopped by max count)
q4>
```


Lab: Shared Memory and Semaphores

```
q4> history
.
.
.
 47 <none> mixed?      3 struct vas      merge of 43 and 46
 48 <none>  array      1 struct proc     stopped by max count
 49 <none>  array      1 struct proc     stopped by max count
 50 <none>  array      1 struct proc     stopped by max count
q4>
q4> merge 49
merged 1 item with 1 item making 2 items
q4> merge 48
merged 1 item with 2 items making 3 items
q4>
```

Finally once the process table entries have been loaded then the processes can be identified.

```
q4>
q4> print p_pid p_comm p_uid
p_pid  p_comm p_uid
 4088  "mclient"  101
 4089  "mclient"  101
 4090  "mclient"  101
q4>
```

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores)

Semaphore structures.

The mserver, mclient programs as well as using shared memory, use a semaphore set to control access to shared data. So next lets look at the data structures for this.

From the /usr/conf/space.h.d/core-hpux.h file we can see that the semaphore kernel parameter are held in the seminfo structure.

```
q4> load struct seminfo from &seminfo
loaded 1 struct seminfo as an array (stopped by max count)
q4>
```

How many semaphore sets are there configured (not actual in use)?
How many processes can have undo operations registered?
How many semaphore can one process have undoes registered against?

```
q4>
q4> print -t
  indexof  0
   mapped  1
  spaceof  0
   addrof  7015928
physaddrof 7015928
  realmode  0
   semmap  66
  semnmi  64
   semnms  128
  semnmu  30
   semmsl  2048
   semopm  500
  semume  10
   semusz  104
   semvmx  32767
   semaem  16384
q4>
```

Using the semaphore set ID from either ipcs or the mserver load just the semid_ds structure for our set.

```
q4> load struct semid_ds from &sema skip 17%semnmi
loaded 1 struct semid_ds as an array (stopped by max count)
```

Semnmi can only be used as the seminfo structure is currently on top of the stack in Q4.

Lab: Shared Memory and Semaphores

How many semaphores are in the set?

What is the address of the first one?

What time was the last semaphore operation performed?

```
q4> print -t
      indexof  0
      mapped   1
      spaceof  0
      addrof   7951376
      physaddrof 7951376
      realmode  0
      sem_perm.uid  101
      sem_perm.gid  20
      sem_perm.cuid 101
      sem_perm.cgid 20
      sem_perm.mode 33200
      sem_perm.seq  0
      sem_perm.key  1930691222
      sem_perm.ndx  0
      sem_perm.wait 0
      sem_perm.pad  ""
      sem_base 8077912
      sem_otime 917303156
      sem_ctime  917277074
      sem_nsems 2
      sem_spare  0
      sem_pad    ""
```

Now load the set of semaphores

```
q4> load struct __sem from sem_base max sem_nsems
loaded 2 struct __sem's as an array (stopped by max count)
```

What are the current values of the semaphores?

What is the PID of the last process to operate on the semaphore?

How many processes are waiting for the value of one of the semaphores to increase?

How many processes are waiting for the value of one of the semaphore to be zero?

Lab: Shared Memory and Semaphores

```
q4> print -t
  indexof  0
  mapped   1
  spaceof  0
  addrof   8077912
physaddrof 8077912
  realmode 0
  semval  2
  sempid 4090
  semmcnt 0
  semzcnt 1

  indexof  1
  mapped   1
  spaceof  0
  addrof   8077920
physaddrof 8077920
  realmode 0
  semval  1
  sempid 4090
  semmcnt 0
  semzcnt 0
q4>
```

There is no definite way to know which process locked a semaphore! but most uses of semaphores make use of undos and it is possible to find out which processes have undos registered for a semaphore.

The semu table consists of semmnu entry of struct sem_undo's BUT instead of the array being of size 1 like it is declared... it is of size semume. This technique is fairly commonly used in the kernel and in 'C' in general (its called encapsulation).

Whilst this technique is commonly used Q4 does not make it easy to then read the whole structures in a table. Q4 will only load the structures as they are defined.

The real size of the entries in the semu table is given by the semusz from seminfo.

Using Q4 the sem_undo structures can be loaded singly and then the array of undo structures can be loaded from there.

There is an alternative way of loading the sem_undo structures though. The first field of struct sem_undo, un_np, forms a linked list of the active sem_undo structures. The first element in this link list is referenced by the kernel variable **semunp**. Using this the useful sem_undos can be loaded. This would still mean that the arrays of undo structures would need to load singly but it is a good start.

Lab: Shared Memory and Semaphores

```
q4> load struct sem_undo from semunp next un_np max 100
loaded 3 struct sem_undo's as a linked list (stopped by null pointer)
q4>
```

Once the linked list of semaphore structures has been loaded then the first undo structure for each process can be read. The field `un_cnt` says how many undos are used for each `sem_undo`.

In the case of the `mclient` program it is incredibly unlikely that these structures would be loaded at a time when there was more than one undo needed, so in this case (and probably many others) this linked list is sufficient.

```
q4> print -tx
      indexof  0
      mapped   0x1
      spaceof  0
      addrof   0x7a02d0
physaddrof 0x7a02d0
      realmode 0
      un_np    0x7a0338
      un_cnt   0x1
un_ent[0].un_aoe 0xffffffffffffffff
un_ent[0].un_num 0
un_ent[0].un_id 0x11

      indexof  0x1
      mapped   0x1
      spaceof  0
      addrof   0x7a0338
physaddrof 0x7a0338
      realmode 0
      un_np    0x7a0198
      un_cnt   0x1
un_ent[0].un_aoe 0xffffffffffffffff
un_ent[0].un_num 0
un_ent[0].un_id 0x11

      indexof  0x2
      mapped   0x1
      spaceof  0
      addrof   0x7a0198
physaddrof 0x7a0198
      realmode 0
      un_np    0
      un_cnt   0x1
un_ent[0].un_aoe 0xffffffffffffffff
un_ent[0].un_num 0
un_ent[0].un_id 0x50
q4>
```

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

q4>

Printing just the key fields makes this more readable.

```
q4> print un_np %#x un_cnt un_ent[0].un_aoe un_ent[0].un_num un_ent[0].un_id
      un_np un_cnt un_ent[0].un_aoe un_ent[0].un_num un_ent[0].un_id
0x7a0338      1          -1          0          17
0x7a0198      1          -1          0          17
0              1          -1          0          80
q4>
```

In this example the semaphore set was ID 17, from here it can be seen that the first two entries have a pending undo of -1 registered against semaphore 0 in the set.

The process table entries point to the point to the sem_undo structures so if we print out the addresses of these sem_undo's then we can search the process table for the entries pointing to them.

```
q4> print -x addrof
      addrof
0x7a02d0
0x7a0338
0x7a0198

q4> load struct proc from proc max nproc
loaded 276 items
q4> keep p_stat
kept 96 of 276 struct proc's, discarded 180
q4> keep p_semundo == 0x7a02d0 || p_semundo == 0x7a0338
kept 2 of 96 struct proc's, discarded 94
```

Now having isolated the processes structures they can be identified.

```
q4> print p_pid p_comm
p_pid  p_comm
4089  "mclient"
4090  "mclient"
q4>
```

Lab: Shared Memory and Semaphores

As a last look at semaphores, lets identify why a process is asleep and what it is waiting for.

The third mclient process that should be executing should be sleeping attempting to get the write lock on the buffer.

So lets take a look at what it is doing.

Get back the list of all active processes, either by reloading it or by recalling the earlier example.

```
q4> recall -1
copied 96 items
q4>
```

Find the mclient processes, and extract the one we haven't looked at yet.

```
q4> print p_pid p_comm | grep mclient
4088 "mclient"
4089 "mclient"
4090 "mclient"
q4> keep p_pid == 4088
kept 1 of 96 struct proc's, discarded 95
```

The state information is now largely in the thread structure so load that. In this case mclient is single threaded so there is no need to follow the linked list.

```
q4> load struct kthread from p_firstthreadp
loaded 1 struct kthread as an array (stopped by max count)
```

The kt_wchan field is used to know when to wakeup a thread, it normally points to a kernel data structure associated with the reason the thread is asleep.

```
q4> print -x kt_wchan
kt_wchan
0x7b425e
```

When the data structure is in the 'statically' mapped part of the kernel then Q4 (and adb) can usually interpret addresses using the 'p' format

```
q4> print kt_pri kt_wchan %p
kt_pri  kt_wchan
 667  __sem+0xc6
q4>
```

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

The `kt_pri` value can also be used to find out information about a sleeping process. For the `wchan` value `__sem+0xc6` we can see that the sleep appears to be associated with semaphores. Looking in the `sem.h` for priority information could help to interpret it further.

Firstly convert the priority into the external format we normally see (just subtract 512).

```
q4> kt_pri-512
0233    155    0x9b
q4>
```

Then `grep` for `pri`, priorities are typically defined relative to either `PTIMESHARE(128)` or `PZERO(153)`

```
root@tiger[mod6] grep -i pri /usr/include/sys/sem.h
# define PSEMN (PZERO + 3)    /* sleep priority waiting for greater value */
# define PSEMZ (PZERO + 2)    /* sleep priority waiting for zero */
root@tiger[mod6]
```

`PZERO` is 153 (`PTIMESHARE+25`) the break point between hi and lo pri sleeps.

This thread is sleeping at `PSEMZ` so it's `kt_wchan` points to the `semzcnt` field of the `__sem` structure. In order to be able to load the structure we need to know the offset of this field.

```
q4> fields -cv struct __sem
struct __sem {
    u_short semval; /* off 0 bytes, len 2 bytes */
    u_short sempid; /* off 2 bytes, len 2 bytes */
    u_short semncnt; /* off 4 bytes, len 2 bytes */
    u_short semzcnt; /* off 6 bytes, len 2 bytes */
};
q4>
```

This field has an offset of 6 inside the structure.

```
q4> load struct __sem from kt_wchan-6
loaded 1 struct __sem as an array (stopped by max count)
q4> print -tx
indexof 0
mapped 0x1
spaceof 0
addrof 0x7b4258
physaddrof 0x7b4258
realmode 0
semval 0x2
sempid 0xffa
semncnt 0
semzcnt 0x1
```

This shows that there is one process sleeping waiting for the semaphore to have a zero value.

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

```
q4> print semval sempid semncnt semzcnt
semval sempid semncnt semzcnt
      2  4090      0      1
q4>
```

Whilst this is the semaphore it does not tell us which semaphore in a set it is nor which semaphore in the set. To find that we will have to search the semid table to see which set includes this `__sem` structure.

To load the table we need the seminfo data again, either reload it or recall an earlier copy.

```
q4> history | grep seminfo
 53 <none> array      1 struct seminfo  stopped by max count
 57 <none> array      1 struct seminfo  copy of 53
 63 <none> array      1 struct seminfo  copy of 53
 68 <none> array      1 struct seminfo  copy of 53
 81 <none> array      1 struct seminfo  copy of 53
q4> recall 81
copied 1 item
```

Then load the table

```
q4>
q4> load struct semid_ds from &sema max semmni
loaded 64 struct semid_ds's as an array (stopped by max count)
```

this loads the whole table, to get just the active element filter of a field like `sem_base`.

```
q4>
q4> keep sem_base
kept 18 of 64 struct semid_ds's, discarded 46
```

Then using the `addrOf` of the semaphore structure the `wchan` pointed to filter out the `semid_ds` that contains it.

```
q4> keep ( sem_base <= 0x7b4258 ) && ( ( sem_base + 8*sem_nsems - 1 ) >= 0x7b4258
kept 1 of 18 struct semid_ds's, discarded 17
q4>
```

Having obtain the `semid_ds` then we know which semaphore in which set the third `mclient` is sleeping on.

Lab: Shared Memory and Semaphores

In the lab directory there are three programs `message_wrt`, `message_rd` and `message_db`.

`Message_wrt` creates a message queue and then writes messages to it. It prompts for a message type and then sends an appropriate message. For type 1 it asks for the message text, for other types it generates its own.

`Message_rd` then allows you to select the type of message to read.

`Message_db` is a modified version that only works on 32bit kernels. `Message_db` as well as using the supported system call interface to read messages also opens `/dev/kmem` and digs out the content of the message queue. As with any digging around in `/dev/kmem`, this is not really supportable and would need recoding to cope with any changes in the kernel. Hence the problem with 64bit kernels.

Run `message_wrt` and send a number of messages. Whether you chose to run the other two is up to you. They should give a good understanding of how the message types can be used to selectively fetch the message from the queues. Leave some messages to look at with `Q4`,

As with the other SYSV IPC tools the kernel parameters are held in an info structure that can be seen in the `space.h` header files. So load that first

```
q4> load struct msginfo from &msginfo
loaded 1 struct msginfo as an array (stopped by max count)
```

How many message queues can be set up with this kernel configuration ?

```
q4> print -t
indexof 0
mapped 1
spaceof 0
addr of 7015968
physaddr of 7015968
realmode 0
msgmap 42
msgmax 8192
msgmnb 16384
msgmni 50
msgssz 8
msgtql 40
msgseg 2048
q4>
```

Now the `msqid_ds` table can be loaded.

```
q4> load struct msqid_ds from &msgque max msgmni
loaded 50 struct msqid_ds's as an array (stopped by max count)
```

Lab: Shared Memory and Semaphores

Filter just the in use entries using a field such as `msg_perm.key`

```
q4> keep msg_perm.key
kept 3 of 50 struct msqid_ds's, discarded 47
q4>
```

Compare what you have loaded with the output of `ipcs -q`.

From `ipcs -q` get the message queue id and key of your queue.

```
root@tiger[sys] ipcs -q
IPC status from /dev/kmem as of Tue Jan 26 12:59:57 1999
T      ID      KEY      MODE      OWNER      GROUP
Message Queues:
q      0 0x3c1c0465 -Rrw--w--w-   root      root
q      1 0x3e1c0465 --rw-r--r--   root      root
q      2 0x6b140271 --rw-r--r--   ken       users
root@tiger[sys]
```

Now keep just that one

```
q4> keep msg_perm.key == 0x6b140271
kept 1 of 3 struct msqid_ds's, discarded 2
q4>
```

```
q4> print msg_first %#x msg_last %#x msg_qnum msg_lspid msg_rtime msg_cbytes
msg_first msg_last msg_qnum msg_lspid msg_rtime msg_cbytes
0x78ad78 0x78aeb0      14      4257      0      1288
q4>
```

From here the list of message headers can be loaded. The field `msg_first` points to the start of the list which can be followed using the `msg_next` field

```
q4> load struct __msg from msg_first next msg_next max msg_qnum
loaded 14 struct __msg's as a linked list (stopped by max count)
```

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

Print out the message types and their positions in the message pool.

```
q4> print msg_type msg_ts msg_spot
msg_type msg_ts msg_spot
   1      92      0
   2      92     12
   3      92     24
   4      92     36
   5      92     48
   7      92     60
   9      92     72
  11      92     84
  13      92     96
  21      92    108
  21      92    120
  12      92    132
   1      92    144
   1      92    156
```

q4>

The message pool is made out of msgseg segments of msgssz in size, from the earlier msginfo output get the segment size (8 in my example).

Now the messages themselves can be read. The message format is up to the application and so can not be loaded and printed in Q4, instead use the examine command.

The message format that is written by message_wrt is:-

a serial number, for the message	4 bytes
a date	4 bytes (unless message_wrt is compiled 64bit)
a text message	a string of up to 80 bytes.

```
q4> examine __msg + 144*8 using 2Xs
0xc 0x36adb28b another message
q4>
```

```
q4> examine __msg + 72*8 using 2Xs
0x6 0x36adb277 type 9 message
q4>
```

If you have a 32-bit kernel try using the format DY, where D is a decimal, Y is a date, and then the string. For the 64-bit kernel this doesn't work unless message_wrt has been compiled 64-bit, and as it doesn't strictly adhere to the correct data types it might not then work.

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

1. In one window, display all the existing message queues:

```
# ipcs -qbo
```

How many message queues are there? 2

How many bytes are in the first queue? 0

How many messages are in the first queue? 0

2. In another window, invoke q4:

```
# ied -h $HOME/.q4_history q4 -p /dev/mem /stand/vmunix
```

3. Display the msgmap array:

```
q4> examine &msgmap for 42 using x | more
```

Look at the first short integer. It is the total number of available entries in the map. How many are there? _____ The next number is zero. Skip it.

The next two numbers give the location of the first free message segment and the number of contiguous segments available until the next used segment (or the end of the message buffer). The location is the offset into the message buffer. From the start of the message buffer, where is the first set of free segments located?

How many segments are in this set? _____

4. In the first window, run the program "msg_setup".

```
# /usr/local/bin/msg_setup
```

Note: This program creates a message queue, inserts 3 messages, then reads the second message. This will (probably) cause a small fragmentation of the message buffer, which will be reflected in msgmap.

Lab: Shared Memory and Semaphores

5. Display all the existing queues again:

```
# ipcs -qbo
```

How many queues are there? _____

How many messages are in the last queue? _____

What is the "id" number of the new queue? _____

6. Back in the second window, re-examine the msgmap array:

```
q4> examine &msgmap for 42 using x | more
```

The last two non-zero numbers give the location and size of the last set of free message segments. Where is the last set of free segments located? _____

How many segments are in this set? _____

Compare this output to the output in step 3. What happened to the prior entry?

7. Display the msgque array:

```
q4> load struct msqid_ds from &msgque max 50
```

```
q4> print -tx | more
```

Scan down through all the entries and relate them to the output lines of ipcs.

8. List all the pointers to the first message header in each queue:

```
q4> print -x msg_first | more
```

Each message segment takes 8 bytes. Relate these pointers to the values seen in the msgmap in step 6.

Lab: Shared Memory and Semaphores

9. Extract the entry for the message queue created in step 4:

```
q4> keep msg_qnum == 2
```

How many structures were kept? _____

(If there is more than one, we'll be working with only the last one. Load it alone by its virtual address. q4> load struct msgid_ds from 0x<ADDR>)

10. Display that structure:

```
q4> print -tx
```

What is the value of msg_first? _____

This is a pointer to the first message header in this queue.

11. Display the message headers in this queue:

```
q4> load struct __msg from msgfirst max 3  
q4> print -tx | more
```

What message header does message header #1 point to? _____

How many segments is each message? _____

Lab: Shared Memory and Semaphores

1. In one window, display all the existing semaphores:

```
# ipcs -sb
```

How many sets of semaphores are there? _____

How many semaphores are in the first set? _____

How many semaphores total are in all these sets? _____

2. In another window, invoke q4:

```
# ied -h $HOME/.q4_history q4 -p /dev/mem /stand/vmunix
```

3. Display the semmap array:

```
q4> examine &semmap for 66 using x | more
```

Look at the first short integer. It is the total number of available entries in the map. How many are there? _____ The next number is zero. Skip it.

The next two numbers give the location of the first free semaphore structure and the number of contiguous structures available until the next used semaphore structure (or the end of the array). The location is the offset into the array of the semaphore structure. From the start of the array, where is the first set of free semaphore structures located? _____

How many semaphore structures are in this set? _____

4. In the first window, run the program "sem_setup".

```
# /usr/local/bin/sem_setup
```

Note: This program creates a semaphore set with 3 semaphores, then a set with 5 semaphores, and finally removes the set with 3 semaphores. This will (probably) cause a small fragmentation of the semaphore array, which will be reflected in semmap.

Lab: Shared Memory and Semaphores

5. Display all the existing semaphores again:

ipcs -sb

How many sets of semaphores are there? _____

How many semaphores are in the last set? _____

What is the "id" number of the new set? _____

6. Back in the second window, re-examine the semmap array:

q4> examine &semmap for 66 using x | more

The last two non-zero numbers give the location and size of the last set of free semaphore structures. Where is the last set of free semaphore structures located?

How many semaphore structures are in this set? _____

Compare this output to the output in step 3. What happened to the prior entry?

7. Display the sema array:

q4> load struct semid_ds from &sema max 64
q4> print tx | more

Scan down through all the entries and relate them to the output lines of "ipcs".
Note: Some entries in the array will represent removed semaphore sets and won't have an output line that corresponds to them.

8. List all the pointers to the first semaphore structure in each set:

q4> print -x sem_base

Each semaphore structure takes 8 bytes. Relate these pointers to the values seen in the semmap in step 6.

Module 6 — InterProcess Communication

Lab: Shared Memory and Semaphores

9. Extract the entry for the semaphore set created in step 4:

```
q4> keep sem_nsems == 5
```

How many structures were kept? _____

(If there is more than one, we'll be working with only the last one. Load it alone by its virtual address. q4> load struct semid_ds from 0x<ADDR>)

10. Display that structure:

```
q4> print -tx
```

What is the value of sem_base? _____

This is a pointer to the first semaphore structure in this set.

11. Display the semaphore structures in this set:

```
q4> load struct __sema from sem_base max 5
```

```
q4> print -tx | more
```

What is the value that each of these semaphores is initially set to? _____

Module 7

Memory Management

The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better.

— Edsger Dijkstra

You don't really want to know what goes in to making pepperoni.

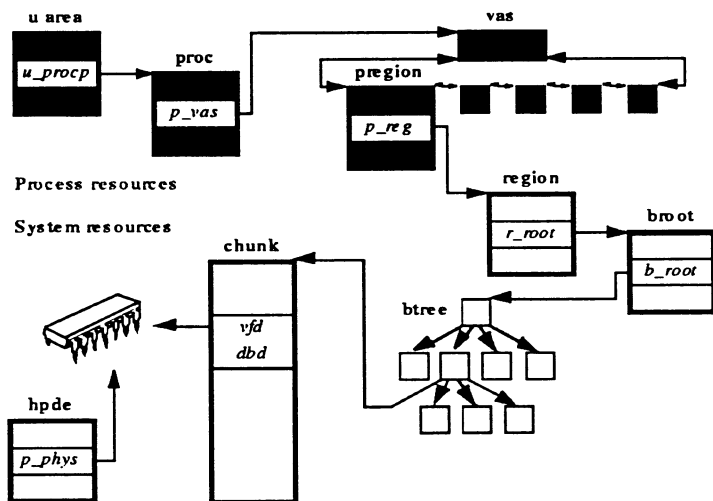
— Ed Springer

Objectives :

- Understand the purpose of the different structures associated with virtual memory.
- Understand how and why memory pages are allocated, freed up, and recovered.

Slide: Data Structure Overview

Data Structure Overview



a606163

Notes:

Если 210-50 сейчас не нужен,
 может он и не устанавливается -
 - отключить.

Module 7 — Memory Management

Slide: Data Structure Overview

In the Process Management module, we looked at structures down to the pregions, which describe each region of the process (such as text, data, and stack). All of these structures are *process resources*, because each process has its own unique structures and they aren't shared among multiple processes.

Once we get below the pregion level, we are looking at *system resources*. These structures can be shared among multiple processes (although they are not required to be shared).

In addition to eventually mapping pregions to physical memory through the region, btree, and vfd structures (which are processor independent), we need to provide support for the processor's ability to translate virtual addresses to physical memory (which is processor dependent). PA-RISC uses *page directory entries*, or *pdes*, for this purpose.

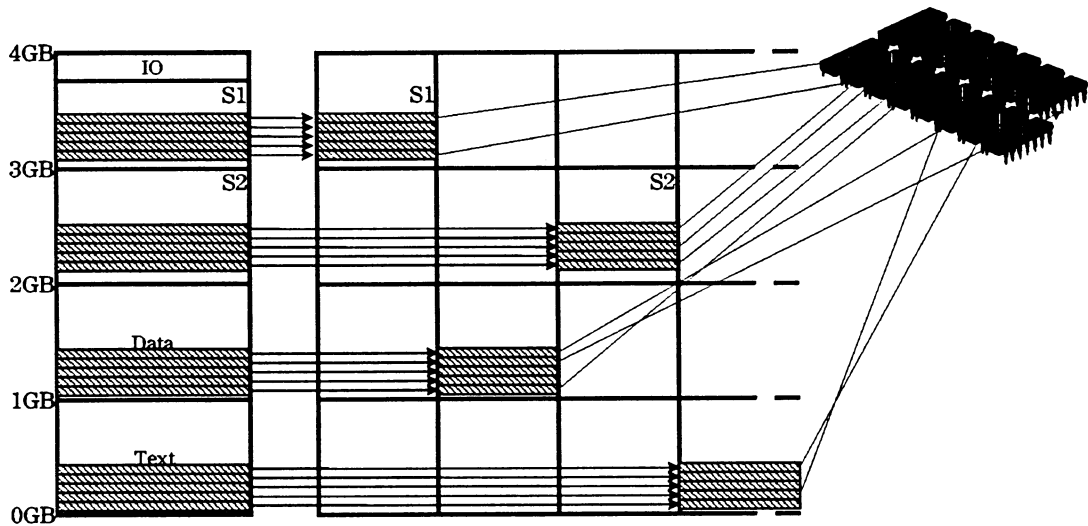
This module will begin by building from the physical memory up to the region. Refer back to this slide frequently to keep from getting disoriented.

Also note that this module will cover both 32 and 64-bit kernel implementations, and PA 1.1 and 2.0 versions, including the issues of dealing with 32-bit processes running on a 64-bit kernel. As much of the higher-level structures in both implementations have the same use, although wider in the 64-bit implementation, they will be discussed in parallel, and their differences pointed out where significant. Take note of the use of structures with slightly different names (such as *htbl* and *htbl2_0*), which are chosen at run time depending on the underlying hardware.

After the system structures, we will look at the virtual memory view of *fork()* and *exec()*. We will finish the module by exploring how and why pages are moved between disk and memory.

Slide: Physical RAM Underlying Virtual Memory

Physical RAM Underlying Virtual Memory



a606164

Notes:

Slide: Physical RAM Underlying Virtual Memory

Although programs which run on HP-UX are not greatly concerned with the underlying hardware on a given machine, everything the system runs has to be in RAM or *physical memory*, the chips where program instructions and data are stored.

A machine has a finite amount of RAM available to it, but the view to each process in HP-UX is that it has either a 4Gb address space (4 1-Gigabyte quadrants for a 32-bit process) or a 16Eb address space (4 4-Exabyte quadrants for a 64-bit process). To accomplish this slight of hand, HP-UX implements *virtual memory*.

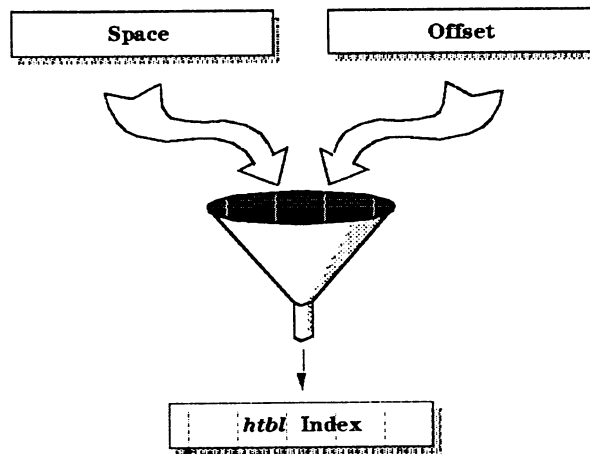
In the *virtual memory* model, a process has its own view of how its memory is laid out, independent of the underlying physical allocation of memory. HP-UX creates a mapping between each process' view of memory, and the actual allocation of physical memory to those processes. This mapping is managed by HP-UX, and implemented in the underlying hardware. When a process references an address in its view of memory, the system combines the address referenced with the space ID appropriate for that process and the type of access, and finds the corresponding location in physical memory, which is then used.

As mentioned in the architecture section, the PA-RISC hardware attempts to convert a virtual address to a physical address with the *TLB* or the *Block TLB*. If it can't resolve the address, it generates a page fault (interrupt type 6 for an instruction TLB miss fault; interrupt type 15 for a data TLB miss fault). It is then up to the kernel to handle this fault.

Originally, HP-UX used an inverse page table called *PDIR* (Page Directory) which mapped physical pages to virtual addresses. Now it uses a hashed table called *HTBL* (Hashed Page Table), which allows for a broad range of addresses to be searched quickly.

Slide: Calculating an Index into the HTBL

Calculating an Index into the HTBL



a696165

Notes:

Slide: Calculating an Index into the HTBL

Once the hardware has failed to find a virtual to physical mapping for an address, the first step in converting a virtual address to a physical address in software is to calculate a *hash index* into *htbl* (or *htbl2_0* for PA2.0 machines). This *hash index* is then used as an index into the array of *hpde* (or *hpde2_0*) structures to begin the search of the page directory. HP-UX uses simple algorithms to create the index, so that they can be computed in very few instructions (2 or 3), as this is a very common operation, and has been revised over the years based on empirical performance tests:

(PA1.1) $\text{index} = ((\text{space} \ll 5) \text{ xor } (\text{offset} \gg 12)) \& (\text{nhtbl} - 1)$

(PA2.0) $\text{index} = ((\text{space} \gg 10) \text{ xor } (\text{offset} \gg 13)) \& (\text{nhtbl} - 1)$ } *X example*

where *nhtbl* is the number of entries in HTBL. *nhtbl* is always a power of two, so *nhtbl - 1* becomes a mask limiting the index to a valid value.

As an example, suppose 32-bit PA1.1 HP-UX is translating the address **0xd158.0x7ffe6000**.

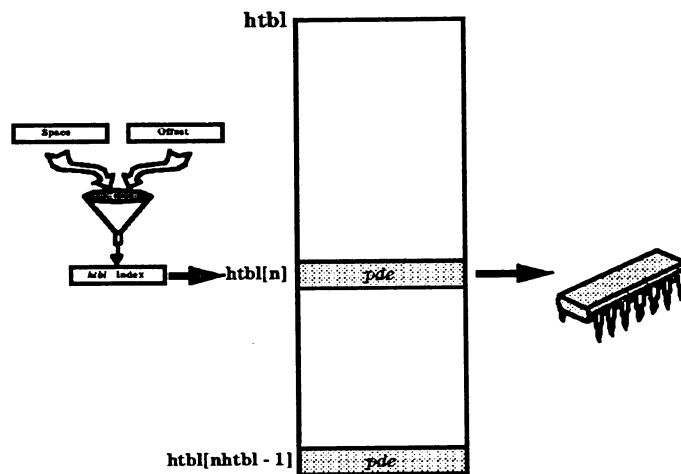
- First, shift the space left five bits, giving **0x1a2b00**.
- Next, shift the offset right 12bits, giving **0x7ffe6**.
- Then perform an exclusive-or of the two numbers: **0x1a2b00 xor 0x7ffe6 = 0x1dd4e6**.
- Finally, mask this with *nhtbl - 1*. For this example, suppose *nhtbl* is **0x8000**. *nhtbl - 1* is then **0x7fff**, and the index is masked to **0x54e6**.

The size of the hash table is based on the size of physical memory, as well as other virtual-address-consuming tasks, then adjusted to fit power-of-two requirements to make the algorithms more efficient.

- Each graphics driver estimates how many entries it will need for I/O mapping. This number is stored in *niopdir*.
- The kernel first approximates *nhtbl* as the sum of *niopdir* and the number of pages of RAM (*phys_mem_pages*).
- *nhtbl* is now adjusted to a power of two. If the first approximation is 25% larger than a power-of-two, or less, it is rounded down. Otherwise it is rounded up to the next power of two.

Slide: Mapping a Virtual Address to a Physical Address

Mapping a Virtual Address to a Physical Address



a696166

Notes:

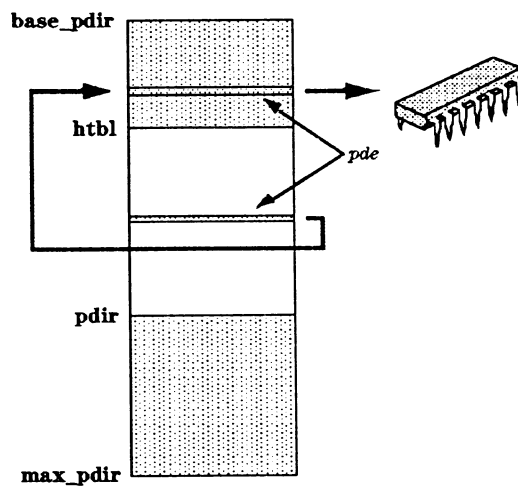
Slide: Mapping a Virtual Address to a Physical Address

The index generated by the hashing algorithm is now used as an index into HTBL. Each entry in the table is referred to as a *pde* (*page directory entry*), and is of type `struct hpde` (or `hpde2_0`).

The virtual space and offset are compared with information in the *pde* to verify that we have the correct entry. If we do, we retrieve the physical address from the *pde*. The translation from virtual address to physical address is complete.

Slide: When Multiple Addresses Hash to the Same HTBL Entry

When Multiple Addresses Hash to the Same HTBL Entry



a696167

Notes:

Slide: When Multiple Addresses Hash to the Same HTBL Entry

As with any hash algorithm, multiple addresses can map to the same HTBL index. For example, `0x0958.0x7ffe6000` hashes to the same index as the previous hashing example, `0xd158.0x7ffe6000`. The entry in HTBL is actually the starting point for a linked list of pdes. Each entry has a *pde_next* pointer which points to another pde, or contains NULL if it is the last item of the linked list.

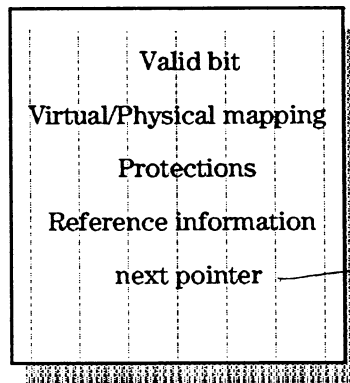
HTBL is surrounded by two other collections of pdes to which each HTBL entry can point. One area is from *base_pdir* to *htbl*, and the other area is from *pdir* (which is also the end of HTBL) to *max_pdir*. This can be thought of as one large collection of pdes stretching from *base_pdir* to *max_pdir*. The group of entries used as the linked-list starting points (that is, HTBL) is located within this collection. HTBL and the surrounding pdes are referred to collectively as the *sparse PDIR*. HTBL is always aligned to begin at an address that is a multiple of its size (that is, a multiple of `nhtbl * sizeof(struct hpde)`).

In practice, there are enough entries in HTBL itself that the linked lists seldom grow beyond three links. *pdir_free_list* points to a linked list of sparse PDIR entries that are not being used and are available for use. *pdir_free_list_tail* points to the last pde on that linked list.

Slide: The hpde Structure

The hpde Structure

hpde contents



HA элемент
с тем же
значением
след.

a696168

Notes:

Module 7 — Memory Management

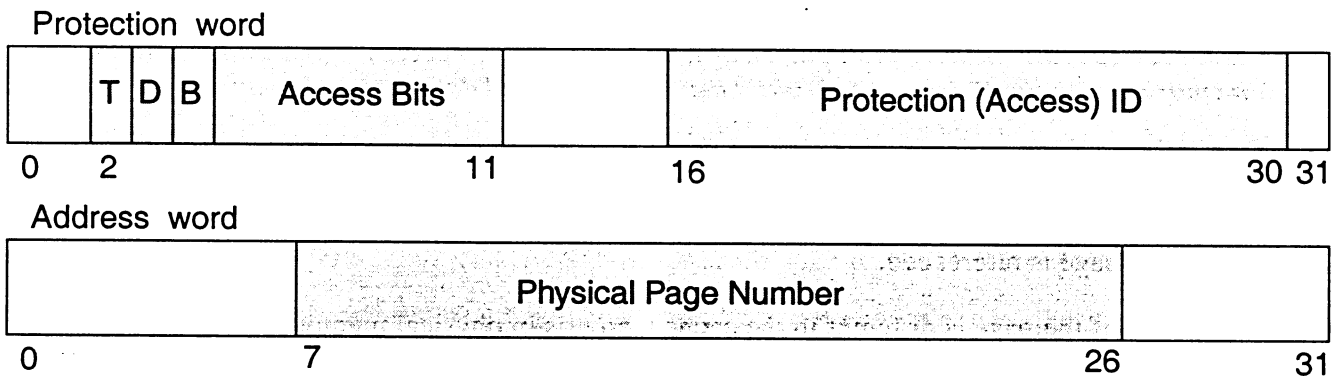
Slide: The hpde Structure

Each pde contains information on the virtual-to-physical address translation, along with other information necessary for the management of each page of virtual memory.

Each pde contains a copy of the data that are in the hardware TLB. The operating system is responsible for updating the TLB entry flags when the status of a page changes, so this is where we store the data which is then passed to assembly statements that update the TLBs. Specifically:

- *idtlba* (INSERT DATA TLB ADDRESS) and *iitlba* (INSERT INSTRUCTION TLB ADDRESS) and *idtlbp* (INSERT DATA TLB PROTECTION) and *iitlbp* (INSERT INSTRUCTION TLB PROTECTION) are used to insert page translation and protections on a PA1.1 system.
- *idtlbt* (INSERT DATA TLB TRANSLATION) and *iitlbt* (INSERT INSTRUCTION TLB TRANSLATION) are used to insert the page translations and protections on a PA2.0 system.

This figure shows graphically which bits are required by the PA-RISC1.1 hardware. Conforming to a particular hpde format allows the OS to invoke a *hardware table walker*, hardware which implements the same TLB miss handler functionality described here in software, but faster:



All other bits in the structure may be used at the discretion of HP-UX.¹

¹ Processors with a hardware table walker also require additional information: the valid bit, virtual space and offset, and reference flag.

Slide: The hpde Structure

Page information in each hpde

pde_valid: The entry is valid only when this flag is set. Before making a change to the pde, the kernel unsets this flag. It then makes the change and sets the flag.

pde_vpage: Contains the high 15 bits (20 bits in 2.0) of the virtual offset.

pde_space: Contains the complete 16-bit (32 bits in 2.0) virtual space.

pde_ref: Used by *vhand()* (discussed later in this module, beginning on page 11-73) to tell if a page has been used recently. This flag is set by the low-level trap handling code whenever a page reference generates a trap or fault. Specifically, the kernel sets the flag whenever it receives an interrupt 6 (*instruction TLB miss fault*), 7 (*instruction memory protection trap*), 15 (*data TLB miss fault*), 18 (*data memory protection trap*), 20 (*TLB dirty bit trap*), 21 (*page reference trap*), 26 (*data memory access rights trap*), 27 (*data memory protection ID trap*), or 28 (*data memory unaligned data reference trap*).

pde_accessed: Used by the *stingy cache flush* algorithm. This flag is set if the page is referenced as data. When the kernel flushes the page from the processor's cache, both the data cache flush and the instruction cache flush routines are called. If this flag isn't set, the data cache flush routine knows the page isn't in the data cache. Specifically, the kernel sets the flag whenever it receives an interrupt 18 (*data memory protection trap*), 26 (*data memory access rights trap*), 27 (*data memory protection ID trap*), or 28 (*data memory unaligned data reference trap*).

pde_rtrap: When set, any access to this page causes a *page reference trap interruption* (interrupt 21). When handling an interrupt 16 (*non-access instruction TLB miss fault*) or 17 (*non-access data TLB miss fault*), the kernel sets this bit if *pde_ref* is not set, so it will be notified and can set *pde_ref* when the page is referenced.

pde_dirty: Marks if the page is different in the cache than it is in physical memory. If a page isn't dirty and is flushed from the cache, the processor doesn't have to write it back to physical memory. Whenever a *Store*, *LOAD AND CLEAR WORD INDEXED*, or *LOAD AND CLEAR WORD SHORT* instruction is executed and the D bit in the TLB is not set, the processor issues an interrupt 20 (*TLB dirty bit trap*). HP-UX handles this trap by setting the D bit in the pde and then updating the hardware TLB.

pde_dbrk: Data breakpoint enable: not used by HP-UX, but present for the TLB.

Slide: The hpde Structure

Когерентна структура
управління пам'яттю

pde_ar: Access rights used by the TLB.¹ Following are the defined access rights used in HP-UX.² Values set in the *p_prot* field of a process' pregion translate to access rights as noted.

PDE_AR_KR (0x00): Kernel read-only page. May only read in kernel mode; no write or execute allowed. Used for kernel page copies when the source page is not in memory and has to be paged in, or when emulating a NULLDREF page with an older executable.³ Maps from pregion's PROT_KERNEL | PROT_READ.

PDE_AR_KRW (0x10): Kernel read-write page. May only read or write in kernel mode; no execute allowed. Widely used for data pages which are available to the kernel but not to user processes. Maps from PROT_KERNEL | PROT_WRITE and PROT_KERNEL | PROT_READ | PROT_WRITE.

PDE_AR_KRX, *PDE_AR_KXR* (0x20): Kernel read-execute page. May only read or execute in kernel mode; no write allowed. Used for kernel text. Maps from pregion's PROT_KERNEL | PROT_EXECUTE and PROT_KERNEL | PROT_READ | PROT_EXECUTE.

PDE_AR_KRWX, *PDE_AR_KXRW* (0x30): Kernel read-write-execute page. May read, write, and execute in kernel mode. Used for interrupt stacks and the MP communications area page (*mp_comm_area*). Maps from pregion's PROT_KERNEL | PROT_WRITE | PROT_EXECUTE and PROT_KERNEL | PROT_READ | PROT_WRITE | PROT_EXECUTE.

PDE_AR_UR (0x0f): User read-only page. May read in either user mode or kernel mode; no write or execute allowed. Used for memory-mapped pages when PROT_WRITE and PROT_EXEC aren't specified. Maps from pregion's PROT_USER | PROT_READ.

PDE_AR_UX (0x7f): User execute-only page. May only execute in user mode; read and write are not allowed. Used if we are emulating a NULLDREF page with an older executable.³

¹ For a good example of how a program can manipulate this field, see the *mmap(2)* and *mprotect(2)* manpages.

² *Kernel mode* is equivalent to privilege level 0, and *user mode* is equivalent to privilege level 3. For more detail on access rights than is supplied here, refer to Chapter 3 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, especially Table 3-1.

³ Binaries compiled on S800 machines at HP-UX 7.0 and earlier set the text start-point half-way into Page 0, at 0x800 instead of 0x1000. To maintain backwards compatibility, HP-UX needs to generate a SIGSEGV when the lower half of the page is accessed. It sets the normal access rights of the page to *PDE_AR_UX* so the text can execute in user mode. On a data protection fault it determines if the read is from an address in the range 0x800 to 0xffff. If so, it changes the rights on the page to *PDE_AR_KR*, copies the requested byte(s) to the requested register, and then resets the rights. Otherwise it generates the segmentation violation signal.

If a text pregion requires this emulation, its *p_hdl.hdlflags* will include *PFL_NONULLREFPG*. If a physical page requires this emulation, its *pfdat* entry's *pf_hdl.hdlpf_flags* will include *HDLPF_ONULLPG*.

Slide: The hpde Structure

PDE_AR_URW (0x1f): User read-write page. May read or write in either user mode or kernel mode; no execute allowed. Used for memory-mapped pages when *PROT_WRITE* is specified and *PROT_EXEC* is not; also used for pages mapped through *iomap()*. Maps from pregon's *PROT_USER | PROT_WRITE* and *PROT_USER | PROT_READ | PROT_WRITE*.

PDE_AR_URX, *PDE_AR_URXKR*, *PDE_AR_CW* (0x2f): User read-execute page. May read in either user mode or kernel mode; may only execute in user mode; no write allowed. Used for memory-mapped pages when *PROT_EXEC* is specified and *PROT_WRITE* is not. Also used for copy-on-write pages so they can be read and executed, but cause a fault when a write is attempted. Also used on the *break page* used by *ptrace()* to single-step through user programs. Maps from pregon's *PROT_USER | PROT_EXECUTE* and *PROT_USER | PROT_READ | PROT_EXECUTE*.

PDE_AR_URWX (0x3f): User read-write-execute page. May read or write in either user mode or kernel mode; may only execute in user mode. Used for memory-mapped pages when both *PROT_WRITE* and *PROT_EXEC* are specified; Maps from pregon's *PROT_USER | PROT_WRITE | PROT_EXECUTE* and *PROT_USER | PROT_READ | PROT_WRITE | PROT_EXECUTE*.

PDE_AR_GATE (0x4c): On execute, promote to kernel mode. No read or write allowed; may execute in either user mode or kernel model. Used for the gateway page to promote the privilege from user mode to kernel mode.

PDE_AR_NOACC (0x73): No access allowed to the page. Maps from pregon's *PROT_USER | NONE* and *PROT_KERNEL | NONE*.

PDE_AR_INVALID (0xff): Used to fill saved access rights fields in various structures; never used in the pde or sent to the TLB.

pde_uncache: Not used.

pde_protid: Protection ID used by the TLB. Although this field is eighteen bits long, HP-UX and the TLB only use the lower fifteen bits.

HP-UX generates protection IDs using an algorithm called *hdl_alloc_id()* (also used to generate space IDs). The algorithm creates a pseudo-random number within the bounds of the protection ID (0 - 32767). It then checks a table called *protid_map* which has one bit per possible protection ID. If the pseudo-random ID is already being used (that is, the *protid_map* bit is clear), it searches the map (first in the neighborhood of the chosen ID, then progressively farther away) until it finds an unused ID. In the unlikely event all 32768 IDs are being used, the system has no alternative but to panic (**panic: hdl_alloc_spaceid: spacemap exhausted**).

pde_accessed: Also used by the stingy cache flush algorithm. This flag is set if the page is referenced as text. When the kernel flushes the page from the processor's cache, both the data cache flush and the instruction cache flush routines are called. If this flag isn't set, the instruction cache flush routine knows the page isn't in the instruction cache. Specifically, the kernel sets the flag whenever it receives an interrupt 6 (*instruction TLB miss fault*) or 7 (*instruction memory protection trap*).

Slide: The hpde Structure

pde_uip: This is a lock flag used by the trap handling code, which can't afford to take the time to use spinlocks or semaphores when it changes a translation (that is, virtual space and offset) in a pde. The assembly code must first own the global flag *ll_pde_upd_lock* to make a translation change. *ll_pde_upd_lock* is normally non-zero. There is a special PA-RISC instruction, *LOAD AND CLEAR WORD INDEXED with coherent operation hint (LDCWX,CO)*, designed to atomically obtain a lock. Once the trap code owns this lock, it sets *pde_uip*, changes the translation, clears *pde_uip*, and releases *ll_pde_upd_lock*.

Before any translation changes can be made to the pde from the upper-level virtual memory code, the kernel must first obtain a pde lock with the *PD_LOCK* macro (which uses a spinlock) and must then wait for *pde_uip* to clear (indicating that no trap code is modifying the entry).

This flag is useful in a multiprocessor environment.

pde_phys: The physical page number. This is the physical memory address divided by the page size, 4096 bytes.

pde_modified: This tells the upper-level virtual memory routines if the page has been modified since the last time it was written to a swap device. If it hasn't changed when the system chooses to reclaim the page, there is no need to rewrite the page to the swap device.

pde_ref_trickle (these two bits are in the *other entry information* word): Used in conjunction with *pde_ref* on machines with a hardware table walker (that is, the hardware is able to search HTBL directly, and only generates a TLB miss when it can't find the page in HTBL). This will be discussed along with the reference flag later in the module.

Other entry information

pde_block_mapped: This particular page is mapped by the Block TLB. We cannot alias these pages.

pde_alias: If *PDE_ALIAS* is set, this pde has been allocated from elsewhere in kernel memory rather than being a member of the sparse PDIR. More on this beginning on page 11-21

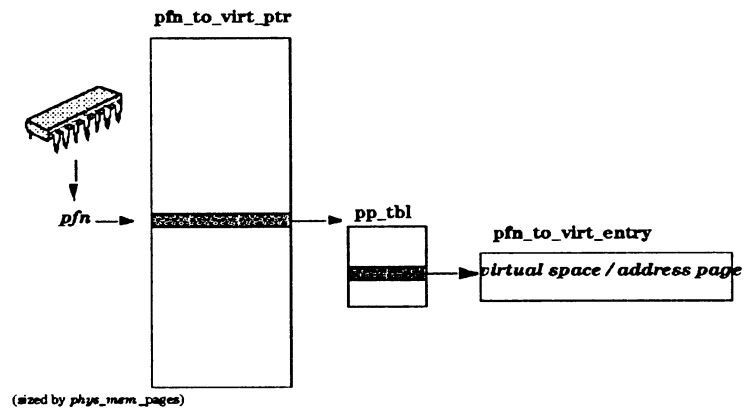
var_page: Indicates how many 4k subpages are mapped into this superpage entry. This is not a count, but a base-2 log operation, such that the count of subpages == $1 \ll (2 * var_page)$. Note that this only has relevance for PA2.0 systems.

Next entry pointer

pde_next: The pointer to the next pde in the linked list, or NULL if this is the end of the list.

Slide: Physical to Virtual Address Translation

Physical to Virtual Address Translation



a696160

Notes:

Slide: Physical to Virtual Address Translation

HP-UX stores physical-to-virtual information in an array called *pfn_to_virt_ptr*,¹ of type `struct pfn_to_virt_ptr`. Each of these structures, in turn, points to a short list of `pfn_to_virt_entry` structures. Overall, there is one `struct pfn_to_virt_entry` per page of physical memory.

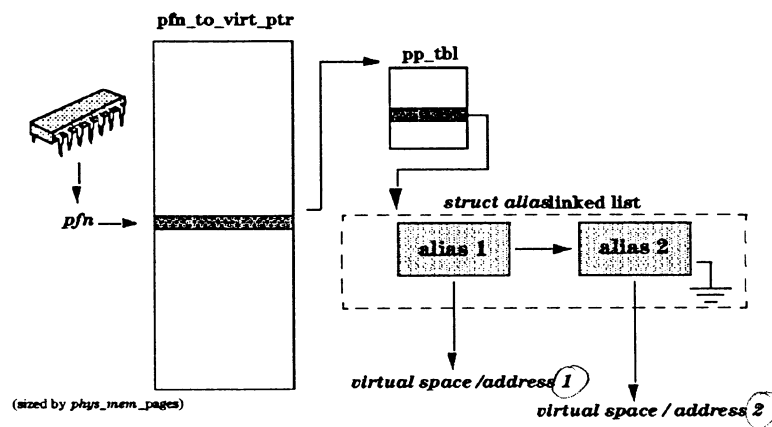
When the kernel first maps a virtual page to a physical page, it also completes the corresponding `pfn_to_virt_entry` structure, inserting into this the virtual page and space ID which correspond.

The kernel uses a spinlock every time it needs to modify a virtual to physical mapping, to avoid conflicts with other processors. Currently there are 32 (`NPDIR_LOCKS`) locks implemented for this, with a hashing algorithm used to choose a lock based on virtual address (see `GET_PD_LOCK_PTR` in `vm_pdir1_1.c` and `vm_pdir2_0.c`).

¹ This array was called *pgtopde_table* prior to HP-UX 10.10, and did not have the alias capability -- It was simply an array of `xpdes`. Other schemes used in 10.10 and 10.20 to implement aliasing include *pfn_topde_info_table* and the *pfn_to_virt_table*

Slide: Mapping Multiple Virtual Addresses to One Physical Address

Mapping Multiple Virtual Addresses to One Physical Address



a696170

Notes:

Slide: Mapping Multiple Virtual Addresses to One Physical Address

We have covered the case where a single virtual address maps to a physical page. As we will discover when we talk about *fork()* and *copy-on-write* beginning on page 11-49, it is quite useful to be able to map multiple virtual addresses to one physical address. `pfn_to_virt_ptr[]` and `pfn_to_virt_entry` structure is where we keep track of these multiple mappings.¹

As we have seen, the first mapping of a virtual address to a physical address will result in the `pfn_to_virt_entry` containing a space and virtual page number. If we then map a second, different virtual address to the same physical address, this condition will be detected when updating `pfn_to_virt_entry`. When the space listed for that pfn is `INVALID_SPACE_ENTRY` (0xFFFFFFFF), but the `offset_page` is non-zero, the system knows to use the `alias_or_offset` element as an alias structure pointer rather than a virtual page number.

Creating a new alias for an existing page involves inserting the new translation into the `pdir` as well as creating a `pfn_to_virt_entry` for the new alias, and linking it to the page's list. The global variable `max_aapdir` contains the total number of alias pdes on the system. Once a page is allocated for use as alias pdes, it is not returned, so the value of `max_aapdir` may grow over time but will never shrink.²

The number of available alias pdes is stored in `aa_pdircnt`. When an alias pde is used or reserved (we reserve one if we include an HTBL pde in an alias linked list, in case we have to move it later), `aa_pdircnt` is decremented. When an alias pde is returned to `aa_pdirfreelist` or unreserved, `aa_pdircnt` is incremented.

The number of available alias structures is kept in `aa_entcnt`. Once a page is allocated for use as a group of alias structures, it is not returned. We don't keep track of the total number of alias structures on the system, just the number of available structures

The end of the hardware dependent structures

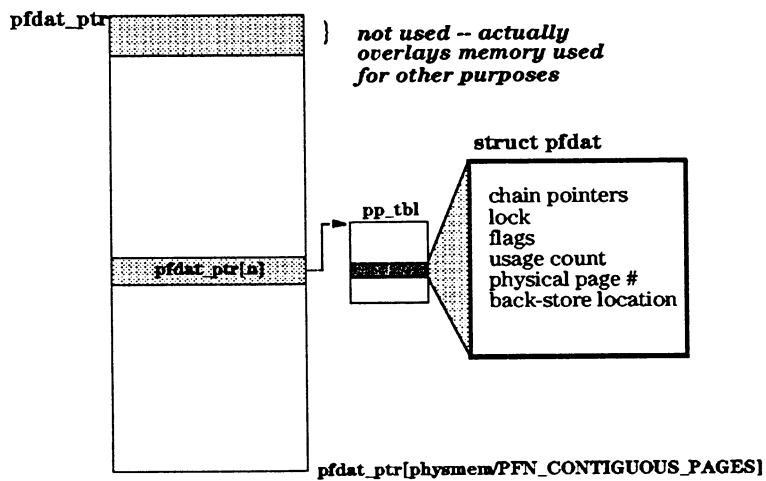
What we have discussed so far exists in the *hardware dependent layer (HDL)*. HP-UX tries to do as much as possible in a hardware-independent manner. Next we'll look at the structures in the *hardware independent layer (HIL)*.

¹ This is completely independent of the address aliasing discussed in Chapter 3 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. HP-UX does not use the hardware address aliasing feature discussed there.

² Each time `unhashdaemon` is woken up, it checks `aa_pdircnt`. If it has dropped below `min_alias_pdirs` (256), the kernel allocates another page and carves it into new alias pdes. Likewise, if `aa_entcnt` has dropped below `min_alias_entries`, the kernel allocates a page and carves it into new alias structures.

Slide: Hardware Independent Page Information Table (pfdat)

Hardware Independent Page Information Table (pfdat)



a606171

Notes:

Slide: Hardware Independent Page Information Table (pfdat)

The hardware independent layer, or HIL, of the virtual memory subsystem is not at all concerned with TLB misses. Its purpose is to manage which pages are in memory and which pages have been written to the swap devices, and how to move one to the other. The act of moving data from physical memory to a swap device, or moving data from a swap device to physical memory, is called *paging*.

The most basic HIL component is the *pfdat*, or page frame data table.¹ There is one *pfdat* entry for each page frame that can be written to a swap device. HP-UX never pages kernel memory (the pages which contain the kernel's text, and data), so *pfdat* manages only a subset of physical memory. In order to avoid wasting too much space managing physical memory which never gets paged, and to allow more flexibility of table management the table is allocated in two tiers. One spot is allocated in the *pfdat_ptr* array to manage groups of `PFN_CONTIGUOUS_PAGES` (4096 pages or 16MB) of memory. Each entry in this array points to a table of `PFN_CONTIGUOUS_PAGES` *pfdat* structures. Where the memory is not *paged* (such as the case with kernel memory), the table is not allocated, and the pointer is left as `NULL`².

Once initialized, the table pieces are linked together into *views* (currently only a `PA_VIEW`) of *ponds* of *page-pools* (e.g. `NON_EQUIV_MAPPED_POOL`) of different allocation *colors* (for keeping cached areas together). For details on the page-frame allocation policies, see the allocation and deallocation routines in *vm_pgalloc.c*. Fields in each *pfdat* entry include:

Linked list information

pf_hchain: Hash chain list. This is a singly-linked list. The pointer to the first hash chain entry is in an array called *phash* (which will be discussed later).

pf_next, *pf_prev*: Pointers to a doubly-linked list of available (free) pages. There is a *struct pfdat* array, indexed by allocation *color*, named *phead* which uses only its *pf_next* and *pf_prev* elements to point to the head and tail of the free page linked list.

Hashing information

pf_devvp, *pf_data*: The algorithm to find which hash chain to use to find a page is based on the *vnode* pointer of the swap device and the disk block number on that device where the page has been written. More on this later.

Lock information

pf_lock: This is a beta semaphore structure used to lock the page while modifying the *pde* (that is, the physical-to-virtual translation, access rights, or protection ID).

pf_cache_waiting: This is incremented prior to requesting *pf_lock* (remember, the process can go to sleep waiting for it) and is decremented when we the page lock).

¹ At some point in the history of AT&T's UNIX development, it was decided to refer to a particular page of memory as a *page frame*. You will see this terminology throughout the hardware independent layer.

² Earlier revisions of HP-UX would allocate a single table for all pageable physical memory, with the non-pageable parts of memory unallocated at the beginning of the table.

Slide: Hardware Independent Page Information Table (pfdat)

Other information

pf_pfn: The page frame number. We can figure this out by where we are in the table and dividing by the size of *struct pfdat*, but having the number here saves time.

pf_use: This shows how many regions (covered later) are using this page. When this goes to zero, we can be placed on the free linked list.

pf_flags: Flags which show the status of the page. The values used are:

P_QUEUE (0x001): This page is on the free queue (head of queue is *phead*).

P_BAD (0x002): This page has been marked as bad by the memory deallocation subsystem.

P_HASH (0x004): This page is on a hash queue (head of queue is in *phash*).

P_ALLOCATING (0x008): This page is in the process of being allocated. This flag keeps another process from taking this page while it is being remapped.

P_SYS (0x010): This page is being used by the kernel rather than by a user process. Pages marked with this flag (dynamic buffer cache pages, btree pages (covered later), and the results of kernel memory allocation) are being used in addition to the kernel static pages that weren't included in *pfdat*.

P_HDL (0x020): Not used.

P_DMEN (0x040): This page is locked by the memory deallocation subsystem. This is set and cleared with an *ioctl()* call to the *dmem* driver.

P_LCOW (0x080): This page is being remapped by the copy avoidance subsystem (covered later).

P_UAREA (0x100): This page is used by a pregon of type *PT_UAREA*.

Slide: Hardware Independent Page Information Table (pfdat)

pf_hdl: Hardware dependent layer structure for pfdat, which contains:

Hardware dependent

hdlpf_flags: Flags which show the HDL status of the page. The values used are:

HDLPF_TRANS (0x01): A virtual address translation exists for this page. Used in conjunction with HDLPF_MOD and HDLPF_REF.

HDLPF_NOTIFY (0x02): Not used.

HDLPF_PROTECT (0x04): Page is protected from user access. We do this by saving the page's access rights and protection ID, and then changing the actual rights and ID such that user processes cannot access the page. This flag indicates that the saved values are valid.

HDLPF_STEAL (0x08): Virtual translation should be removed when pending I/O is complete.

HDLPF_ONULLPG (0x10): See footnote 3 on page 11-15.

HDLPF_MOD (0x20): Setting this flag is an inexpensive upper-level alternative to changing the *pde_modified_e* or *pde_modified_o* flag in the pde. When the HDL routine *hdl_getbits()* checks for a modified flag on a particular page frame, it first checks HDLPF_MOD. If HDLPF_MOD is not set and HDLPF_TRANS is set, it calls *pdmodget()* to retrieve the value of *pde_modified_e* or *pde_modified_o*.

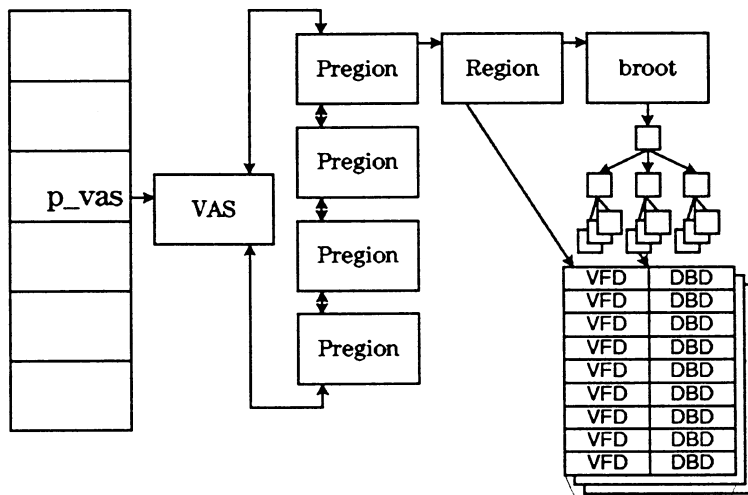
HDLPF_REF (0x40): Setting this flag is an inexpensive upper-level alternative to changing the *pde_ref_e* or *pde_ref_o* flag in the pde. When the HDL routine *hdl_getbits()* checks for a referenced flag on a particular page frame, it first checks HDLPF_REF. If HDLPF_REF is not set and HDLPF_TRANS is set, it calls *pdrefget()* to retrieve the appropriate value of *pde_ref_e*, *pde_ref_trickle_e*, *pde_ref_o*, or *pde_ref_trickle_o*.

HDLPF_READA (0x80): Read-ahead page in transit. This indicates to the *hdl_pfault()* routine that it should start the next I/O request before waiting for the current I/O request to complete.

hdlpf_savear, *hdlpf_saveprot*: When a page is aliased, we have a place to store the access rights if we need to change them temporarily. This gives us a place to store them if the page is not aliased.

Slide: How Processes Reference Pages

How Processes Reference Pages



*Double down
no wrapping,
no spacing*

a696172

Notes:

Slide: How Processes Reference Pages

As we saw in the process management module, each process has its own virtual address space. This space is sparse and comprises of series of regions, each of which is described by a preion structure.

The preions are private to the address space they belong to, which allows the sharing of areas of address space such as the text areas or shared libraries a futher structure. A region, is then used to give a system wide view of this area of virtual address space.

Ultimately the region of address space needs to be described at the page level, since this is the level that memory is managed. Since pages are small (4K) and regions can be large (even a 32-bit application can have a 1.9GB data area, which requires ~500,000 pages) the way that the pages are described need to carefully planned.

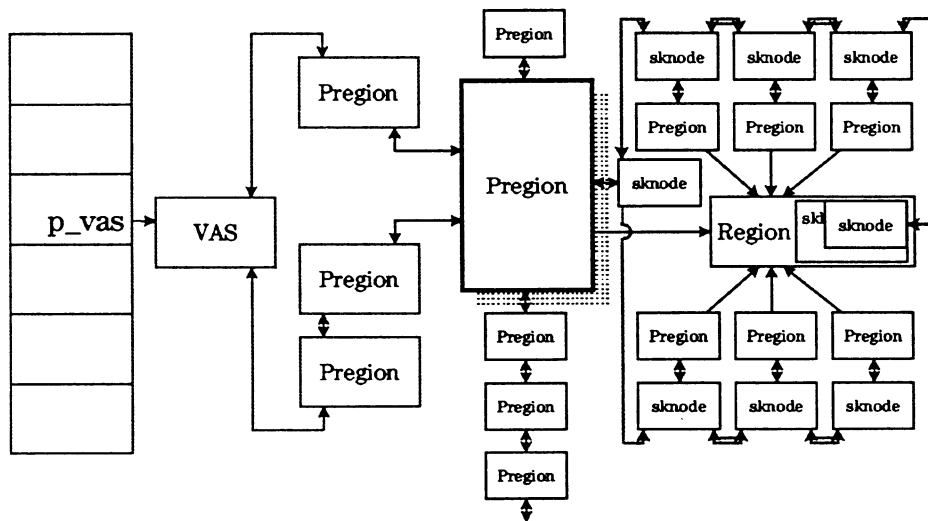
First, if there is the potential for having to describe a large number of pages, the descriptions need to be small. Making the descriptions small then makes them harder to manage themselves. The structures used to describe the pages are virtual frame descriptors (VFDs) and disk block descriptors (DBDs). Each page will have one of each of these structures, and they are jsut 32 bits in size. If an attempt to manage them using a linked list the pointer used to maintain the list would end up double or tripling there size. So vfd/dbd pairs are not managed singly rather they are grouped into chunks.

The region itself can be sparse, not all pages in a region might need to be described.

The method of locating the vfd/dbd pairs need to be able to cope efficiently with potentially very large numbers, and needs to be able to handle sparse sets. The system used is a btree.

Slide: Pregion Structures and Linked Lists

Pregion Structures and Linked Lists



a606173

Notes:

*Вру откарне справуны гаворкасе
се багачы*

Slide: Pregion Structures and Linked Lists

Since processes view their memory from within their own virtual view, the structures closest to the region deal with the virtual mappings of the regions. The per-process structure corresponding to a region is called a pregon (for pseudo-region) structure, and it contains the virtual memory mapping of each region of memory it is interested in. Note that with shared-memory regions, each process will have its own pregon, but all of those pregon structures will point to the same region.

Each per-process pregon structure contains the following elements:

p_ll and *p_next/p_prev*: Pointers linking together all the pregon structures for a given virtual-memory view into a single, doubly-linked list. The *p_next* and *p_prev* pointers are really macros for the link list structures: *p_ll.ll_e_next[0]* and *p_ll.ll_e_prev*. The multiple *ll_e_next* pointers actually implement a *skip list* for faster searching of the otherwise linear list.

p_reg: Points to the region attached by the pregon.

p_space, p_vaddr: The beginning virtual address of the pregon.

p_off: Offset into the region, in pages, of the address *p_space.p_vaddr*. The difference between the pregon's *p_off* and the region's *r_off* is that the pages before *p_off* have *vfddb* structures associated with them, and pages before *r_off* do not. An example of a non-zero *p_off* element is a shared library pregon/region; an example of a non-zero *r_off* element is a data pregon/region.

p_ageremain, p_agescan, p_stealscan, p_bestnice: Used in the *vhand* algorithm, to be discussed later in this module, to determine which memory hasn't been used recently and can be pushed-out to backing store to make room for others

p_forw, p_back: The doubly-linked list of active pregon structures. The pregon structures on this list are the ones that are scanned by the page daemon as candidates to write to a swap device. Where multiple pregon structures share the same region only one will appear on this global linked list. The pregon structure that does appear on the list will be the one referenced by the PSEUDO VAS structure.

Incidentally the **slide is wrong** in this respect, the pregon shown as being on the global list is not the one that would have been selected. The slide was drawn this way just to highlight that a pregon can appear on several lists.

p_deactsleep: The address at which a deactivated process is sleeping.

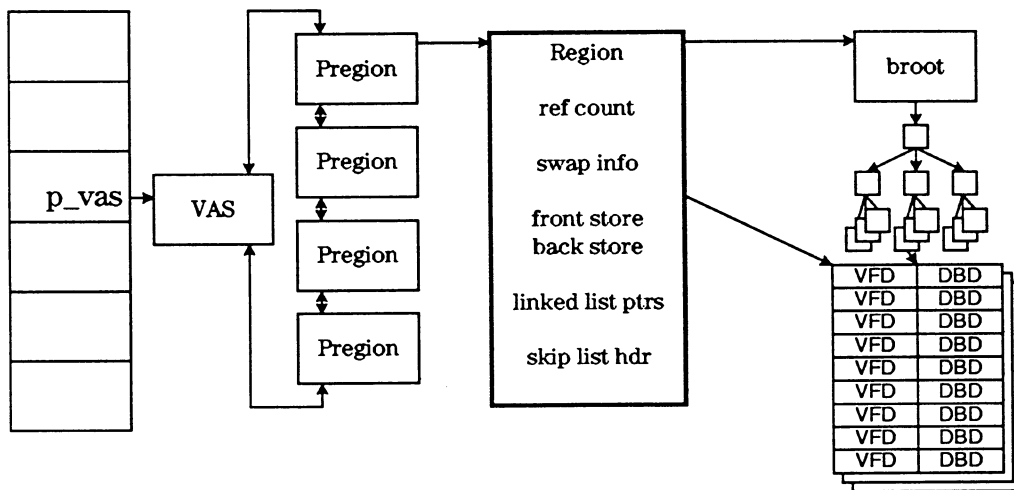
p_pagein: Size of an I/O to schedule when moving data into memory.

p_strength, p_nextfault: Used to track the ratio between sequential and random faults; used to adjust *p_pagein*.

As well as there being linked lists of all pregon structures on the virtual address space and the global pregon list all pregon structures on a region are also listed. With the original 10.20 release and all earlier ones there was a simple linked list. However patch PHKL_9075 changed this to a system of skip lists was introduced. Now the region structure contains a `struct skl` to act as the header for this linked list. The list itself is made up of a series of `struct sknode` structures. The `n_value` field of which then points to its corresponding pregon. The pregon structures point back using the field `p_regsknode`.

Slide: Region Structures

Region Structures



a696174

Notes:

Slide: Region Structures

The *region* is a kernel data structure that associates a group of virtual pages together for a common purpose. It can have one of two types, private (used by a single process) or shared (able to be used by more than one process). The space for the region is allocated as needed. The region structure is never written to a swap device, although its btree may be.

The region is the highest level system-wide memory resource for processes in HP-UX. Regions are pointed to by *pregions* (discussed in the next section), which are a per-process resource.

region (struct region):

Chain pointers

r_forw, *r_back*: Linked list of all active regions. The head of the linked list is *regactive*, a dummy region that only uses *r_forw* and *r_back*.

r_next, *r_prev*: Linked list of all regions associated with the *r_fstore* vnode. This is a circularly linked list.

r_pswapf, *r_pswapb*: Linked list of all regions using pseudo-swap. The head of the linked list is *pswaplist*, a pointer to the first region on the list.

Old a.out support

Most executables start the text and data on a (4 Kbyte) page boundary. HP-UX can treat these as memory-mapped files, because a page in the file maps directly to a page in memory.

Some executables which were compiled on old revisions of HP-UX do not start the text on a page boundary.¹ These executables need some extra care and feeding. The regions that reference these old executables are identified with the *RF_UNALIGNED* flag in *r_flags*.

r_byte, *r_bytelen*: Contain the offset into the a.out file and the length of the text in the a.out file.

r_hchain: Unaligned regions are kept on a hashed list so they can be easily found and shared. The vnode pointer and the byte offset are hashed into one of 32 (*TEXTSHSZ*) values with the hash algorithm *TEXTHASH(VP, DATA)*. This creates an index into *texts[]*, an array of 32 region pointers. HP-UX then follows the hash chain created by *r_hchain* in each region on the chain until it finds a (*r_fstore*, *r_byte*) pair that matches what it's looking for.

If the *RF_HASHED* flag is set in *r_flags*, *r_hchain* contains valid data.

pregion information

r_refcnt: Number of pregions sharing this region.

r_incore: Number of pregions sharing this region that are *in core* (that is, the process they are associated with has the *SLOAD* flag set).

¹ See footnote 3 on page 11-15.

Slide: Region Structures

Virtual memory information

r_key, r_chunk: If we are referencing only one chunk (CHUNKENT pages) or fewer, we don't bother using the btree; instead we use this key and chunk. If *r_key* is UNUSED_IDX (0x7fffffff), we haven't yet used either the btree or *r_chunk*. If *r_key* is DONTUSE_IDX (0x7ffffffe), we are using the btree pointed to by *r_root*. If *r_key* is any other value, it is the key associated with *r_chunk*.

r_root: Pointer to the root of the btree which maps all the pages we're using if they consist of more than a chunk (that is, *r_key* is DONTUSE_IDX).

r_dbd: If the btree pages have been written to a swap device, the location of the first page in the linked list of pages is stored here. When the pages are brought in from the swap device, subsequent pages are pointed to by a pointer in the last chunk of the previous page.

r_fstore, r_bstore: Vnode pointers to where the data for this region comes from initially, and where it should be paged to if necessary. This data depends on the type of preregion above the region (see the discussion on preregions in the following section). In most cases, *r_bstore* is set to the paging system vnode, a global called *swapdev_vp* which is initialized at system startup, and the *r_fstore* is set to the original source of the information such as the executable file (or NULL) for uninitialized memory areas.

r_nvalid: Number of valid pages in the region. This equals the number of valid vfds in the btree or *b_chunk*.

r_dnvalid: If the system decides to swap this entire process, we copy the value from *r_nvalid* here so we can calculate how many pages the process will need when it is allowed to fault back in. This information is used in deciding which process to reactivate when the time comes.

r_swalloc: If RF_SWLAZY¹ is set in *r_flags*, this is the number of pages actually allocated for this region on the swap device. Otherwise, this is the total number of pages reserved and allocated for this region on the swap device

r_swapmem: The number of pages reserved and allocated (there's really no difference) in pseudo-swap. This isn't used until there is no more device swap space available.

r_poip: Number of page I/Os in progress. When we schedule a page to be transferred between memory and a swap device, we increment this counter. When the I/O is complete, we decrement this counter. If we need to free this region, we wait until this counter goes to zero before proceeding, so we don't get any I/Os to or from nonexistent or (worse) remapped pages.

¹ A flag which can be turned-on by enabling the lazy-swap feature in the executable file -- see *chattr(1)*.

Slide: Region Structures

Other information

r_flags: Various indicators of the state of the region. Highlights include

RF_ALLOC (0x00000004): Since we allocate and free regions on demand in HP-UX, there is no free list, so this flag is always set.

RF_UNALIGNED (0x00000020): Old version of an executable where the text doesn't start on a page boundary.¹ If this is the case, we set this flag, read the text in through the buffer cache to align it, and point the vlds at the buffer cache pages.

RF_WANTLOCK (0x00000080): Another stream wants to lock this region, but found it already locked and went to sleep. This ensures that we call wakeup() after unlocking the region so the waiting stream(s) can proceed.

RF_HASHED (0x00000100): The text is unaligned (*RF_UNALIGNED*) and thus is on a hash chain. The region is hashed with *r_fstore* and *r_byte*; the head of each hash chain is in *texts[]*. The *RF_UNALIGNED* flag may be set without the *RF_HASHED* flag (if the system tries to get the hashed region but it is locked, the system will create a private one), but the *RF_HASHED* flag will never be set without the *RF_UNALIGNED* flag.

RF_EVERSWP (0x00000200): Set if the btree has ever been written to a swap device. The only purpose of this flag is for debugging.

RF_NOWSWP (0x00000400): Set if the btree is now written to a swap device. The only purpose of this flag is for debugging (we can tell this independently by checking for *r_dbd* != 0).

RF_IOMAP (0x00002000): This region was created with an *iomap()* system call. *iomapped* regions need special handling when calling *exit()*.

RF_LOCAL (0x00004000): If the a.out file is accessed with NFS and either its sticky bit is set or the global parameter *page_text_to_local* is non-zero, this flag is set and *r_bstore* points to *swapdev_vp* instead of the a.out vnode.

RF_EXCLUSIVE (0x00008000): If a user memory-mapped file is mapped into the data quadrant as *MAP_EXCLUSIVE*, the mapping process is allowed exclusive access to the region. This flag is set, and *r_excproc* is set to the proc table pointer.

RF_SUPERPAGE_TEXT (0x00200000), *RF_PG_SIZE_FIXED* (0x02000000): Used to indicate if variable-sized pages are (or are not) being used on systems which support them.

RF_MPROTECTED (0x01000000): Indicates that the *mprotect(2)* call has been used to change the protections on these virtual pages.

¹ See footnote 3 on page 11-15.

Slide: Region Structures

r_type: The type of region this is.

RT_PRIVATE (0x1): Region cannot be shared among multiple processes, such as process-private data or stack regions.

RT_SHARED (0x2): Region can be shared among multiple processes, such as shared memory or process-executable text.

r_pgsz: The size of the region in pages. This is how big the region would be if all the pages were in memory.

r_off: Offset into the page-aligned vnode, in pages. This is valid only if *RF_UNALIGNED* isn't set. Page *r_off* of the vnode is referenced by the first entry of the first chunk of the region's btree.

r_lock: Lock structure used to get read or read/write locks to modify the region structure.

r_zomb: Set if an executing a.out file on a remote system has changed. The pages are flushed from the processor's cache so the next access will result in a fault. The fault handler finds that *r_zomb* is non-zero, prints the message **Pid %d killed due to text modification or page I/O error** and sends the process a SIGKILL.

r_excproc: The pointer to the proc table entry of the process with exclusive access if *RF_EXCLUSIVE* is set in *r_flags*. Not valid if *RF_EXCLUSIVE* is not set.

Slide: Region Structures

r_mlock, r_lockmem, r_mlockcnt: Used to lock a region's pages in memory. The algorithm used to lock a region can be found in `mlock_region()` in `vm_memlock.c`.

r_hdl: The hardware-dependent layer elements.

HDL region entry (struct `hdlregion`):

r_space, r_prot, r_vaddr: Hints used when attaching the region to a pregon. *r_vaddr* is used to remember the virtual address used for shared memory; the other two are universal.

r_hdlflags: Flags to tell which hints are valid.

RHDL_SID_ALLOC (0x04): *r_space* is valid.

RHDL_PROTID_ALLOC (0x10): *r_prot* is valid.

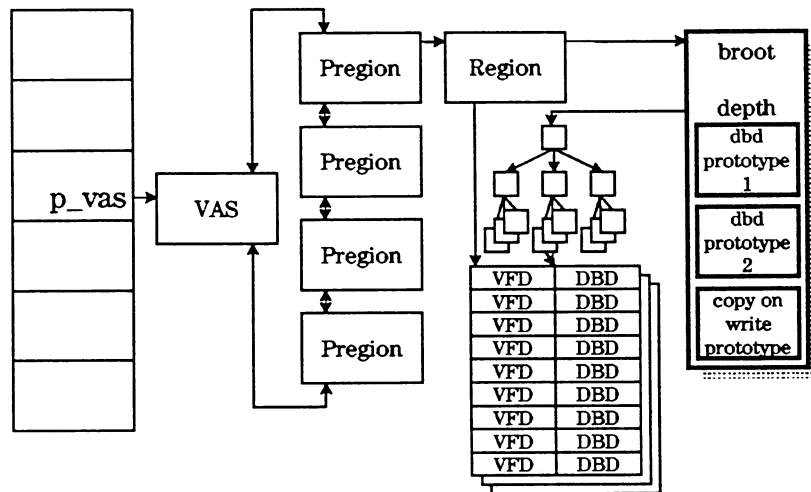
RHDL_MMAP_ATTACHED (0x01): Region has been memory-mapped to user space.

RHDL_MMAP_PUBLIC (0x02): Normally when a memory-mapped region is attached for the first time, it uses the second quadrant's space ID as the protection ID. If another process then attaches the region to its pregon, the kernel must generate a unique protection ID and change the ID on all the pages in the region.

If the memory-mapped file is mapped `MAP_SHARED` and read-only, and the permissions on the file are at least read and execute for user, group, and other, the kernel anticipates that it will be shared with other processes. It sets the protection ID to 0 (any process is allowed access) and sets `RHDL_MMAP_PUBLIC`.

Slide: The Root of the btree: broot Structures

The Root of the btree: broot Structures



*text
 mono
 7pat6 uz
 p...
 n...
 a he
 101600
 swap*

a606175

Notes:

Slide: The Root of the btree: broot Structures

In an unfortunate duplication of semantics, we refer to two different structures as the *root node* of a btree. On the one hand, this is what we call the bnode at depth 1 that is the start of the tree itself. On the other hand, this is what we call the structure that *points* to that root bnode. So to keep things clear, in this material we'll refer to the bnode at depth 1 as the *root bnode*, and the structure that points to it as the *root of the btree*.

Root of the btree (struct broot):

btree information

b_root: Points to the root bnode, the starting point of the btree.

b_depth: Tells how deep the btree is.

b_rpages: An upper bound on the number of pages required for a btree to represent the number of real pages to be managed by this btree. This number is calculated by the kernel routine *vfdpgs()*. The kernel reserves this many swap pages for the btree.

b_rp: Pointer to the region structure using this btree.

b_npages: How many pages are being used to construct this btree. This counts the pages used for both chunks and bnodes.

b_list: A pool of memory from which we can get new bnodes or chunks for this btree. This is actually a pointer to the beginning page of a linked list of all the pages used to construct this btree. We use the last chunk of each page as a link pointer to the next page of chunks. (Using an entire chunk for a 4-byte pointer is wasteful, but makes our bookkeeping easy.)

b_nfrag: Number of unused chunk-sized fragments in *b_list*. When we allocate a page for the btree, we use the highest chunk on the new page as a pointer and set *b_nfrag* to one less than the number of chunks on a page, $(\text{NBPG}/\text{sizeof}(\text{chunk_t}) - 1)$. We then use the fragments from highest to lowest, decrementing *b_nfrag* each time.

vfd/dbd prototypes

b_proto1, *b_proto2*, *b_protoidx*: Because of the cost, both in time and memory usage, associated with btrees, we delay actually allocating space for chunks until the last possible moment. We can do this by storing the value of the default dbd in this root structure. To handle the case where one btree stores information with two possible default dbd's, we have two prototype dbds. If we have a region with a common default value for dbd, we store it in *b_proto1*, set *b_protoidx* to `UNUSED_IDX (0x7fffffff)`, and leave *b_proto2* unused. If we have two default dbd values, we store the lower-addresses dbd in *b_proto1*, the higher-addresses dbd in *b_proto2*, and the page offset in the region where the switch takes place in *b_protoidx*.

Slide: The Root of the btree: broot Structures

b_vproto: An array where we can store information about what vfds should have the copy-on-write bit set. This can often occur on pages before the page is in memory. If the entry isn't used, *b_vproto.v_start* is set to -1. Otherwise, *v_start* is the page index of the beginning of the copy-on-write range, and *v_end* is the page index of the last page of the copy-on-write range.

If all five entries are used, we must allocate all the chunks for the pages in the range so we can set the copy-on-write bit in the vfds explicitly. This is much less efficient than using *b_vproto*, but the prototype area is sized such that we rarely use all the entries.

Caching information

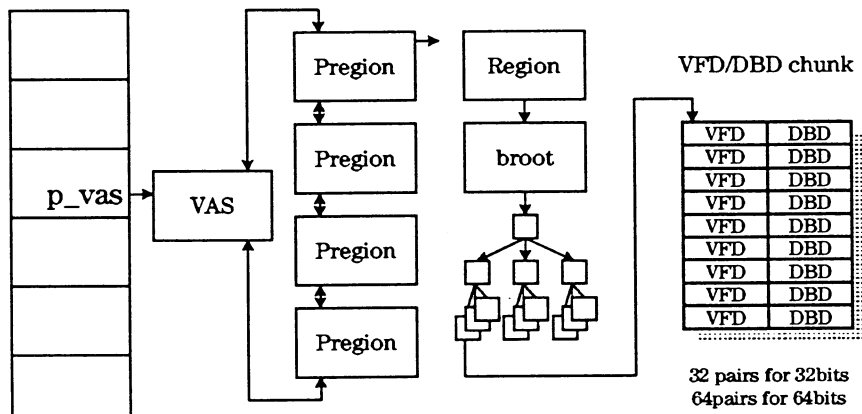
b_key_cache, *b_val_cache*: The virtual memory subsystem frequently asks for the same chunk over and over. We take advantage of this by storing the most recently accessed key/chunk pair in *b_key_cache[0]* and *b_val_cache[0]*, and storing the second most recently accessed key/chunk pair in *b_key_cache[1]* and *b_val_cache[1]*. When we need to search the btree, we first look at these two cached pairs to see if what we need is there. If not, we perform the tree search.

Module 7 — Memory Management

Left blank intentionally

Slide: Managing VFD/DBD Pairs

Managing VFD/DBD Parts



a696176

Notes:

Slide: Managing VFD/DBD Pairs

We need a one-to-one correspondence between vfd's and dbd's. If the page data is in memory, we find out where it is by looking in the vfd. If the page data is not in memory, we find out where it is by looking in the dbd. By definition, if the vfd's *pg_v* bit is set we use the vfd, and if it is not set we use the dbd. We get the one-to-one correspondence with the *vfd/dbd* structure. This structure simply contains one vfd (*c_vfd*) and one dbd (*c_dbd*).

Since we usually need information on groups of pages rather than individual pages, we cluster groups of vfd/dbd's together in *chunks*. There are 32 vfd/dbd's in a chunk on a 32-bit system, 64 on a 64-bit system, defined as *CHUNKENT* in *vfd.h*. Groups of virtual memory pages, such as those allocated to a particular process for a particular use, are organized and sequentially-numbered within *chunks*: so the vfd/dbd for the first page (page 0) within a particular region is the first vfd/dbd in the chunk, and so on.

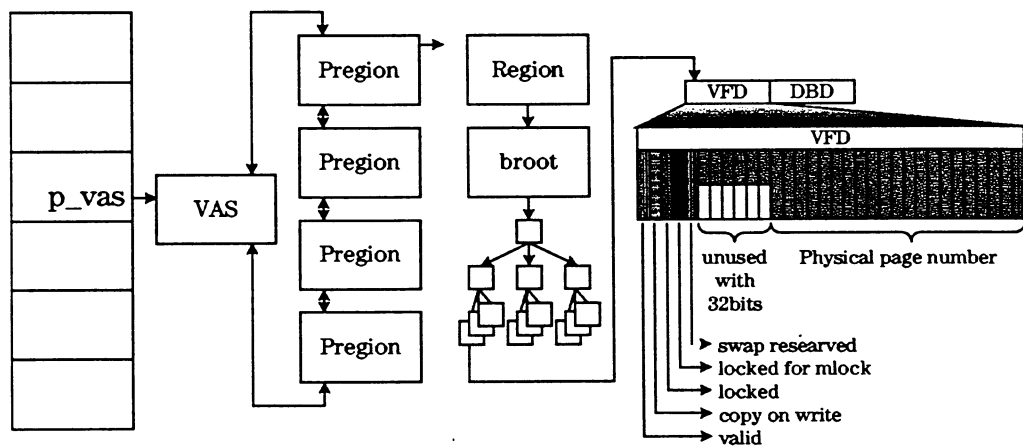
When up to *CHUNKENT* pages are referenced, only a single chunk is needed, and it is referenced directly. When more entries are needed, the chunks are organized as a database, structured as a btree¹. The btree structure allows for quick searches and a reasonably efficient storage of sparse data.

As with most databases, a btree is searched with a *key*, and yields a *value*. In the btrees we are concerned with here, the *key* is the page number in the *region* (such as the third virtual page) divided by the number of vfd/dbd's in a chunk, *CHUNKENT*, and the *value* we get back is the chunk containing the desired vfd/dbd. So if we're searching for the vfd/dbd containing the third page of the *region*, we'll search with a key of 0, and will use the vfd/dbd pair at offset 3 in the resulting chunk. On the other hand, if we're looking for the 500th page, a 32-bit kernel will search for a key of 15 (that is, the integer part of $500 / 32$), and will use the vfd/dbd pair at offset 20 in the resulting chunk (that is, $500 \text{ modulo } 32$). The size of the chunks was chosen to match the size of the nodes in the btree. Since this consists largely of pointers it is twice the size in the 64-bit kernel, as in the 32-bit kernel; hence the larger chunk size. Matching the size of the chunks and bnodes makes managing the memory easier.

¹ From the terms binary and balanced tree. The type of search is rooted in binary search trees, but the btree algorithm allows us to more easily keep the tree balanced.

Slide: Virtual Frame Descriptors

Virtual Frame Descriptors



a696177

Notes:

Slide: Virtual Frame Descriptors

The next building block we need is a way to refer to the page of memory we describe in `pfdat`. We could just use the page frame number, but since it's at most 20 bits long, we can use the extra bits as flags. We call this one-word structure a *virtual frame descriptor*, or *vfd*.

Virtual Frame Descriptor (struct `vfd`):

`pg_v`: Don't get this confused with `pde_valid`. If this flag is set, `pg_pfnnum` is valid and the page's data is in memory. If it is not set, the page's data is on a swap device.

`pg_cw`: If this flag is set, the page's permissions are such that a write will cause a data protection fault, at which time the system will make another copy of the page (we'll see why this is useful when we discuss `fork()`).

`pg_lock`: If this flag is set, raw I/O is occurring on this page. Either the data is being transferred between the page and the disk, or data is being transferred between two memory pages. The kernel will sleep waiting for the I/O to finish before launching further raw I/O to or from this page. To keep things simple, a side effect is that nothing can read the page while it is being written to disk.

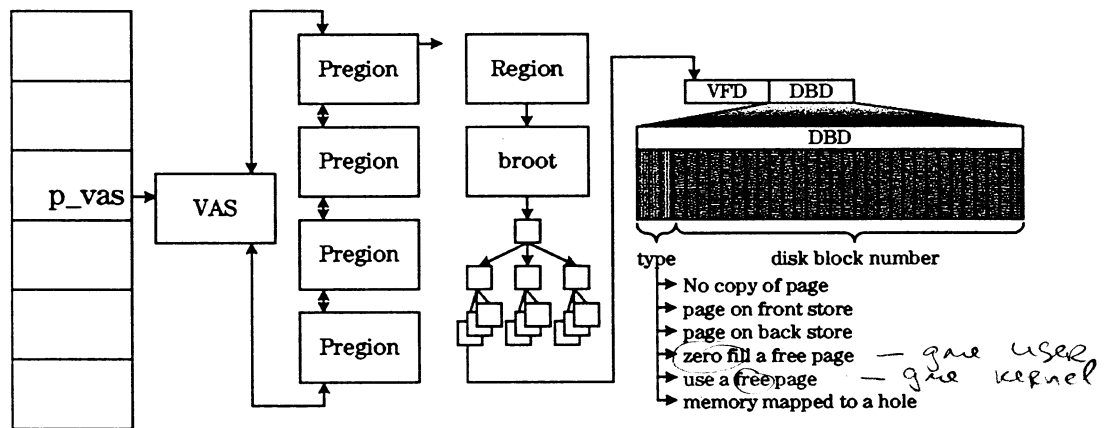
`pg_mlock`: If this flag is set, the page has been locked by the process.

`pg_swresv`: swap space has been reserved as backing store for this page.

`pg_pfnnum` (used in the kernel source much more frequently as its defined alias, `pg_pfn`): This is our old friend, the page frame number. From this, we can access the correct `pfdat` entry for this page. We can also use it as an index into `pfn_to_virt_ptr` and `pp_tbl` arrays to get to the virtual address information. Note that on 64-bit HP-UX, all 27 remaining bits in the `vfd` are used for the page frame number, but in 32-bit only 21 bits are used, as only smaller memory systems are supported.

Slide: Disk Block Descriptors

Disk Block Descriptors



a606178

Notes:

Slide: Disk Block Descriptors

When the `pg_v` bit in a `vfd` is not set, the page of data is not in memory. To find it, we examine a structure called the *disk block descriptor*, or *dbd*. This, like the `vfd` structure, is one word long.

Disk Block Descriptor (struct `dbd`):

dbd_type: There are five types of `dbds`:

DBD_NONE (0x0): There is no copy of this data on disk.

DBD_FSTORE (0x1), *DBD_BSTORE* (0x2): This page can be found on the associated region's front store device or the back store device, respectively. As we'll soon see, the region contains pointers to a front store `vnode` and a back store `vnode`. If the `dbd` points to front store, the system will page in from the *r_fstore* `vnode` in the region; if it points to back store, the system will page in from the *r_bstore* `vnode`.

DBD_DZERO (0x3): This is a *demand zero* page. It doesn't exist. When the page is requested, allocate a page and initialize it to all zeros.

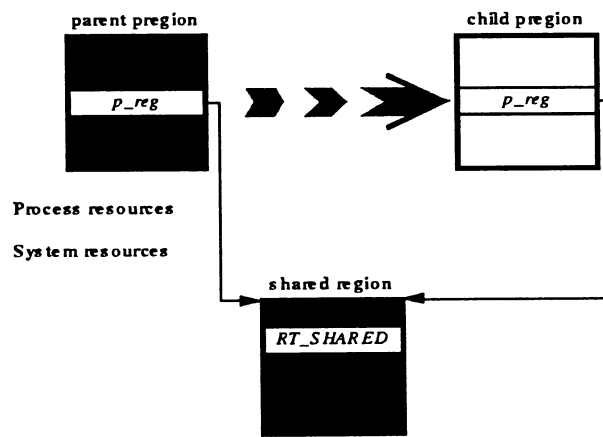
DBD_DFILL (0x4): This is a *demand fill* page. It doesn't exist. When the page is requested, allocate a page but don't initialize it. Because this is a security risk (somebody could write a program and read data from the pages it allocates), this is only used for a new U-area, where we are going to initialize the data immediately.

DBD_HOLE (0x5): This is used for a hole in a sparse memory-mapped file. When we read from it, we get zeros (just as when we read from a sparse file). When we write to it, we allocate a page, initialize it to all zeros, and insert the data; the `dbd` will then be changed to be type `DBD_NONE`.

dbd_data: We'll study the value stored here later. For now, think of it as an abstract pointer that somehow points to the data in a file pointed to by a `vnode`.

Slide: fork() — Duplicating pregonions with Shared Regions

fork () — Duplicating pregonions with Shared Regions



a006170

Notes:

Slide: `fork()` — Duplicating preions with Shared Regions

Duplicating preions during `fork()`

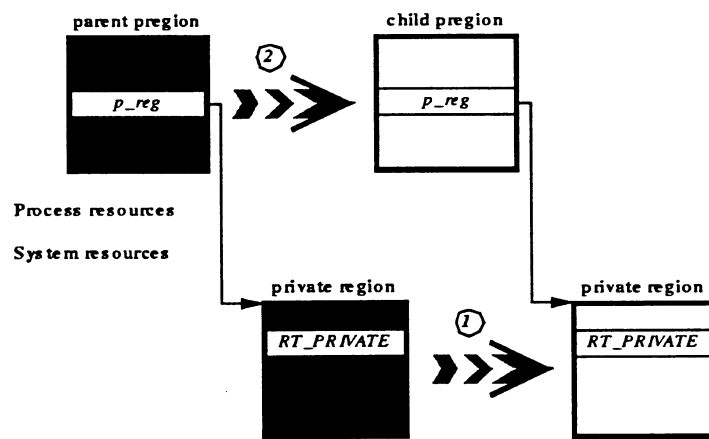
Under the kernel *procdup()* routine (discussed in Process Management section), the system walks the preion list of the parent process, duplicating each preion for the child process. The manner in which this is done is dictated by the region type. If the region is type `RT_SHARED`, a new preion is created which attaches to the parent's region. If the region is type `RT_PRIVATE`, the region is duplicated first, and then a new preion is created and attached to the new region. Because we can share an `RT_SHARED` region between parent and child, we only have to create a new preion and attach it to the shared region.

First, we allocate a new preion and copy fields from the parent preion to the child preion. We initialize the *vhand* parameters *p_agescan*, *p_ageremain*, and *p_stealscan* (discussed later in this module) to 0 and add the child preion to the active preion chain just before the *stealhand* so it won't be stolen for awhile (also discussed later). Finally, we increment the region elements *r_incore* (the number of in-core preions accessing the region) and *r_refcnt* (the number of preions, in-core or paged, accessing the region).

We see that there is very little to be done with respect to virtual memory when forking a shared region. The procedure gets much more complex when we copy an `RT_PRIVATE` region.

Slide: fork() — Duplicating pregon with Private Regions

fork () — Duplicating pregon with Private Regions



a006180

Notes:

Slide: `fork()` — Duplicating preregions with Private Regions

Unlike the `RT_SHARED` case where we first allocate a new *preregion*, in this case we first allocate a new *region*. We set the child region's forward store pointer (`r_fstore`) to the same value as the parent's (incrementing the `vnode`'s `v_count` in the process), and set the child's back store pointer (`r_bstore`) to the kernel global `swapdev_vp` (incrementing its `v_count` also).

We attach this new region to the end of the global region linked list headed by `regactive`, and reserve swap. If there is not enough swap space at this point, `fork()` will fail and the error will be `ENOMEM`. We initialize the child region's `btree` structures, and reserve swap space for the number of pages the `btree` will consume if it were to be completely filled. We copy the parent's `vfd` and `dbd` proto values to the child's `btree` root.

Next, we set the `vfd` proto in both the parent region and the child region such that all pages of the region are *copy-on-write*. Whenever a new `vfd/dbd` pair is added to the `btree`, `b_vproto` now indicates that the copy-on-write flag (`pg_cw`) is to be set in the `vfd`.

Copy-on-write means that we won't copy the pages in the parent's region until we have to. Both parent and child can read the pages and neither needs to know that they're sharing the same page. But when either the parent or child writes to the page, we need to make another copy so that the other process still sees the original view of the page.

Before the alias feature was added in 10.10, the best HP-UX could do was copy-on-write for the parent, and *copy-on-access* for the child. Since each physical page could only have one virtual translation, the child process would have to copy the page whether the process was reading or writing.¹

For each chunk of `vfd/dbds` in the parent's `btree`, we now create a chunk for the child's `btree` and fill it with the proto values for `vfd` and `dbd`. Because we already set the `vfd` proto to copy-on-write for the entire region, each of the default `vfds` in the new chunk will have the `pg_cw` bit set.

We now traverse the chunk of `vfd/dbds` entry by entry. If the `vfd` is not valid (that is, `pg_v` is not set), we simply set `pg_cw` in the parent's `vfd`.

¹ The default behavior for HP-UX 11.00 is to utilize copy-on-write only for `EXEC_MAGIC` and `SHMEM_MAGIC` text pages, and to utilize copy-on-access for child processes in all other private region cases. This behavior can be changed by modifying the global kernel parameter `cow` to have a value of 1 instead of 0. When `cow = 1`, all copy-on-write pages marked for copy-on-write (except those that have yet to be read in from `r_fstore`) will use true copy-on-write instead of copy-on-access. Since this is the direction HP-UX seems to be taking, this module discusses the kernel's behavior when `cow = 1`.

Slide: `fork()` — Duplicating preions with Private Regions

Setting copy-on-write when the vfd is valid

The real work comes when the vfd is valid. First, we increment `r_nvalid` (the number of valid pages) in the child region. The vfd contains a `pf_n` (page frame number), which we use as an index into `pfdat[]`. We increment the `pfdat` entry's `pf_use` count (number of regions using this page).

If the parent vfd's copy-on-write bit isn't set, we need to set the `pde` such that translations to the page behave as copy-on-write. We translate from physical address to virtual address. If the page is aliased, we need to convert each of the aliased `pdes` in turn. If the page is not aliased, we only need to convert one `pde`. The information in the `pdir` is updated, and the TLB purged, so that the old data will not remain in hardware. Because any access of the virtual addresses will now cause a TLB miss fault, we can wait to write the new access rights information to the TLBs at the time of the fault. We have now finished dealing with the low-level page structures.

Finally in this case where `pg_cw` wasn't set in the parent vfd, we set it and copy the parent vfd to the child. If `pg_lock` (locking the page in memory) is set in the parent, we unset it in the child (locks aren't inherited).

Finishing up

Now we have dealt with all the chunks in the parent's `btree` and are to the point where we've done all we need to do to set up copy-on-write. We set the child's `r_swalloc` to the number of pages we've reserved for the region pages and the `btree` pages, and we use `r_prev` and `r_next` to link the child region to the parent region.

From here, we do the same thing that `shared_copy()` does, except that the kernel chooses a new space for the preion rather than copying it from the parent preion. So now we have two different ranges of virtual addresses (different space, same offset) translating to the same range of physical addresses.

- Each parent process access of any of its virtual addresses will cause a TLB miss fault because we have purged the addresses from the TLB.
- Each child process access of any of its virtual addresses will also cause a TLB miss fault because these addresses had not previously existed in the TLB, and don't even exist in HTBL.

Special consideration when duplicating the uarea

We save the `uarea` for last when we fork, because we're going to set the child process up to resume after we're done. When we're doing a `vfork`, we save the `uarea` creation for `exec()` and continue to use the parent's `uarea`.

When we are doing a `FORK_PROCESS fork()`, we allocate a temporary space to make a working copy of the parent's `uarea` that we can modify into the child's `uarea`; we'll free the temporary space after we copy it to the new region later. `fork()` updates the `savestate` in the parent `uarea`'s `u_pcb` just before copying the data (`vfork()` doesn't do this because it's creating the `uarea` during the `exec()`, and the `savestate` will

Slide: `fork()` — Duplicating preions with Private Regions

change immediately).

We allocate a region for this new uarea, initialize its data structure, set its back store to the swap device, and add it to the list of active regions. The uarea has no `r_fstore` value, since it comes with ready-made data.

Next we allocate space for the preion and initialize it. Each uarea gets a unique space ID, because when we have multiple threads per process each thread has its own uarea. We mark the new preion with the `PF_NOPAGE` flag. Uarea preions are not added to the list of active preions, and thus don't have to worry about the `agehand` and `stealhand`. The only time a uarea's pages are written to a swap device is when the entire process is swapped.

We then attach the preion in its rightful place in the linked list of preions connected to the `vas`, store its pointer in `r_pregs`, and sets `p_prpNext` to `NULL`. We also set `r_incore` and `r_refcnt` to one.

We reserve swap space for our uarea pages and `btree` pages and set our default `dbd` to `DBD_DFILL`.¹ Then we actually allocate `UPAGES` pages (defined as enough space for the `user` structure and the kernel stack - 3 for 32-bit, 6 for 64-bit). For each of these pages we take a `pfdat` entry from `phead` (sleeping if we can't get one immediately), store the `pfid` in the `vfd`, set `pg_v`, increment `r_nvalid`, create entries in the virtual-to-physical and physical-to-virtual tables, set the `pfdat` entry's `P_UAREA` and `HDLPF_TRANS` flags, and set `dbd` to `DBD_NONE`. We don't update the TLBs — we'll wait for a TLB miss fault to do that. Now we make our copied area look like it really belongs to the child process, by pointing `u_procp` to the child process and `u_kthreadp` to the child thread.

We're now at the point where the child could conceivably run successfully. So we save our current state in the copied uarea with a `setjmp()` call and point to it with `pcb_sswap`. When the child first 'resumes', the resume code detects that `pcb_sswap` is non-zero and does a `longjmp()` to get back here. The child will then return from `procdup()` with the value `FORKRTN_CHILD`.

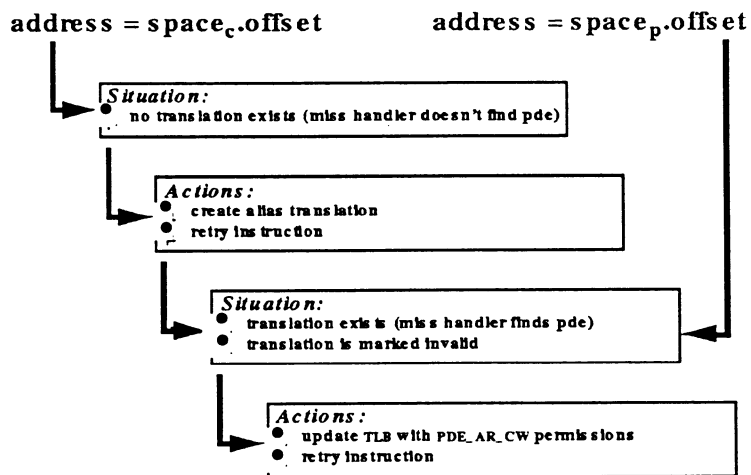
We copy the parent's open file table to the child and then copy the copied uarea into the actual preion. It is during this copy that we will get TLB miss faults which will cause the preion's `pdes` to get written to the TLB, associating the uarea's virtual address with the physical pages we set up.

Now we're done and return from `procdup()` with the return value `FORKRTN_PARENT`.

¹ The uarea is the only region that uses `DBD_DFILL`, which allocates memory but doesn't initialize it, because we know we're going to write over it immediately. If any other region (such as data) were allowed to use `DBD_DFILL`, crafty users could read pages that were flushed by other users, which would be a large security hole.

Slide: fork() — Copy-On-Write First Read Mechanics

fork () — Copy-On-Write First Read Mechanics



a606181

Notes:

Slide: `fork()` — Copy-On-Write First Read Mechanics

Reading from the parent's copy-on-write page

Let's see what happens when the parent region accesses one of its `RT_PRIVATE` pages for read. The processor generates a TLB miss fault, which the kernel deals with as an interrupt. The TLB miss fault handler finds the pde and inserts the information (including the new access rights) into the processor's TLB. On return from the interrupt, the processor retries the read and is successful, since `PDE_AR_CW` allows user-mode read and execute access.

Reading from the child's copy-on-write page

Now let's see what happens when the child region accesses one of its pages for read. Unlike the parent region case, the TLB miss handler won't find a pde for the virtual address, because we haven't set one up yet. All we did initially was set the virtual address in the pregion structure. Now that we actually need the page, we go about setting up the aliased translation.

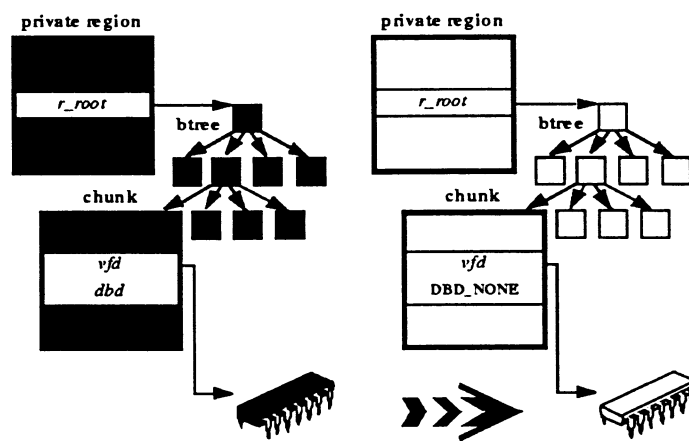
The first thing we do is create a save state in case we run into any problems. We then grab the `vas` pointer (`u.u_procp->p_vas`) and search the skip list to find the pregion containing the page with this address.

If the HTBL pde for this new translation isn't being used, we'll use it for our alias. Otherwise, we get an alias pde from `aa_pdirfreelist` or `aa_pdirfreelist2_0` (which contains allocated memory), rather than from `pdir_free_list` (which contains free pdes in the sparse PDIR). We initialize the pde, replace the single `pfn_to_virt_entry` with an alias list in the `pfn_to_virt_ptr` table, and return from the interrupt.

Now we'll retry the read and fail again with a TLB miss. But this time the miss handler will find a translation and load it into the TLB. We return from interrupt and try the read the third time, and are finally successful.

Slide: fork() — Copy-On-Write First Write Mechanics

fork () — Copy-On-Write First Write Mechanics



a696182

Notes:

Slide: `fork()` — Copy-On-Write First Write Mechanics

Writing to a copy-on-write page that has a virtual translation

If we write to the parent's copy-on-write page or to the child's copy-on-write page when it exists in the TLB¹, we will get a memory protection fault because the page has read/execute-only permissions. Again, we create a save state, get the vas pointer, and find the pregion that contains this virtual address. If the page reference count in the pfdat entry (`pf_use`) is 1, we know that the other process already copied the page, so all we have to do is change the access rights on the page.

If the reference count is greater than one, we then delete this virtual-to-physical translation since we're going to create another. This consists of purging the TLBs, setting the page's access rights to `PDE_AR_NOACC`, flushing the caches, purging the TLBs again, and marking the pde invalid. If both the pde is now unused and this is the HTBL pde (that is, this is the first pde on the hash chain), we try to find another pde in the chain that we can copy to the unused area. If we are unsuccessful, we will simply end up with an invalid first hash entry with a valid `pde_next` pointer. If this *isn't* the HTBL pde, we return the pde to the appropriate free list (`pdir_free_list` if it's from the sparse PDIR, or `aa_pdirfreelist` if it's from the alias pde pool) and remove it from the hash chain. We also remove the alias entry,² and break any connection the page may have had with a swap device.

Next, we get a free pfdat entry from `phead` which gets us a free page to copy to. We add the new virtual-to-physical translation to the sparse PDIR. We set the access rights to `PDE_AR_KRW` (that is, no user permissions) to keep anybody from accessing the page as user data until we've copied the data to it. We also add a new entry to `pfntopde_info_table` to correspond to our new page frame number.

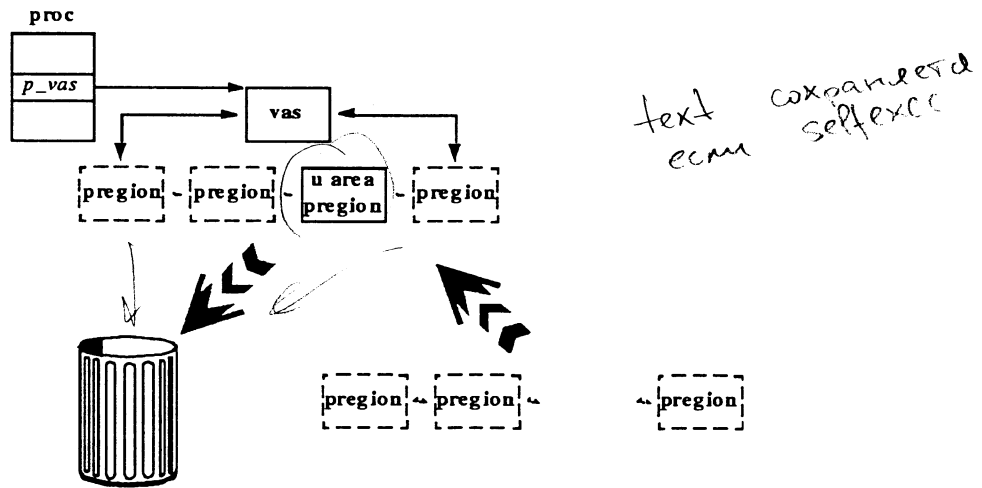
We now copy the data from the original page to the new one. When we're done, we fix the access rights (they were cached in the pregion's `p_hdl.hdlar`). We then purge the TLB (which has this translation because of the copy) so the next access will cause a TLB miss fault and update the access rights.

¹ We will cover the case where an accessed page must be faulted into memory later in the module.

² We don't ever convert the `pfnto_virt_entry` from an alias type back to a non-alias type when there's only one entry left. It simply becomes an alias list that contains one member.

Slide: Virtual Memory and exec()

Virtual Memory and exec()



a606183

Notes:

Slide: Virtual Memory and exec()

When the system performs an `exec()`, the virtual memory system concerns itself with cleaning up old preions/regions and setting up new ones.

Cleaning up from a `vfork()`

Cleanup in the `vfork()` case is simple. Since we were executing in the child process but borrowing everything from the parent process, we simply create our own uarea and return the parent's resources. Then we're ready to add text, data, and so on.

First we get a new `vas` and attach it to our process (`p_vas`). Then we copy the uarea and stack of the parent process and create the uarea preion/region just as we did in `fork()` for a `FORK_PROCESS` fork type. We copy the uarea copy into our new uarea region, point to this now-complete uarea from our thread, and change from using the parent's kernel stack to our new child kernel stack.

Disposing of the old preions: `dispreg()`

If we are calling `exec()` after a `FORK_PROCESS` fork, we have a few regions to dispose of first. Normally, we dispose of all preions except for the `PT_UAREA` preion, which we still need. If the file is calling `exec()` on itself, we save a little processing and keep the `PT_TEXT` and `PT_NULLDREF` regions, too. First we deactivate the preion we're disposing of by removing it from the active preion list. If necessary, we adjust the pageout hands (discussed later in this section), to prevent further paging activity here.

If the region is type `RT_PRIVATE` or we're the last preion attached to it, we need to free up its resources. We wait for any pending I/O to the region to complete (that is, `r_poip = 0`, so no I/O request returns to modify a page and the page now has a different purpose). Then we traverse the region's `btree`, deleting all the virtual address translations.

If the `pde` is not the `HTBL` `pde`, we move the `pde` from its hash list to its free list. If it's from the `HTBL` we try to fill it with a translation down its linked list, and then free the copied `pde`. We remove the physical-to-virtual translation from the page's `pfn_to_virt_entry`. If it was the last virtual translation for this physical page, we clear `HDLPF_TRANS` in the `pfdat` entry. We now remove the preion pointer from the process' virtual address space list and free the memory used by the preion (that is, return it to its kernel memory bucket). Finally, we decrement `r_incore` and `r_refcnt`. If `r_refcnt` goes to zero, we go about freeing the region, too.

Slide: Virtual Memory and exec()

Before we free the region, we again wait for `r_poip` to go to zero so we won't get any unexpected I/O to the pages we're freeing. We then walk the btree again. For each valid page we find, we decrement `r_nvalid` and decrement `pf_use` in the `pfdat` entry. If the particular physical page is not aliased, its `pf_use` will now be 0 and we can free the page for other uses. We set its `P_QUEUE` flag and put it in the `pfdat` free list (`phead`) and increment the kernel global `freemem`. If any other processes are waiting for memory, we wake them all up so that the first one here can have the page (the losers of the race will go to sleep again). If `r_bstore` is `swapdev_vp`, we release our reserved swap pages (`r_swalloc`). We also release the swap pages we had reserved for the btree structure (`r_root->b_rpages`) and free the pages themselves (invalidate their `pdes`, purge the TLBs, flush the caches, move the non-HTBL `pdes` from the hash list to the free list, link the `pfdat` entry into `phead`). Since `r_root` and `r_chunk` (if it's still around) were obtained from memory allocation buckets, so they are moved back to the buckets rather than being freed in the true sense of the word.

Finally, we decrement `activerregions`, remove this region from the `r_forw / r_back` region chain, and return the region memory to the memory allocation bucket from whence it came.

Building the new process

If our's is the first process to use the `a.out` as an executable, the `a.out` `vnode`'s `v_vas` will be `NULL`, and we create the `pseudo-vas`, `pseudo-pregion`, and `region`. Otherwise, we just update the `pseudo-vas`' reference count.

In the non-`EXEC_MAGIC` case, we simply attach a `PT_TEXT` `pregion` to the `pseudo-vas`' `region`. In the `EXEC_MAGIC` case, we set `VA_WRTEXT` in the process `vas`, duplicate the `pseudo-vas`' `region` as a type `RT_PRIVATE` `region` (performing all the steps we discussed for an `RT_PRIVATE` `region` on page 11-49), set `RF_SWLAZY` in the new `region` so we don't reserve swap ahead of time, and attach a `PT_TEXT` `pregion` to it. In both cases, we assign a new space to the `pregion`'s virtual address.

We then attach a `PT_NULLDREF` `pregion` to the global `region` (`globalnullrp`), using the same space as `PT_TEXT`.

We duplicate the `pseudo-vas`' `region` as a type `RT_PRIVATE` `region` (again performing the steps from page 11-49) using `r_off` to point to the beginning of the data portion of the `a.out` file, and attach a `PT_DATA` `pregion` to it. If this is an `EXEC_MAGIC` executable, we use the `PT_TEXT` `pregion`'s space ID, otherwise we assign a new space ID.

Then we grow the `PT_DATA` `pregion` by the size of `bss` (uninitialized data area), using `dbd` type `DBD_DZERO`. This sets `b_protoidx` to the end of the initialized data area and sets `b_proto2` to `DBD_ZERO`. We also reserve more swap.

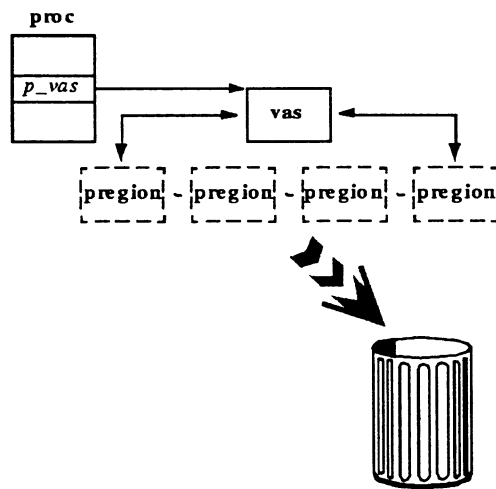
Slide: Virtual Memory and exec()

For the user stack, we create a private region of 3 pages ($SSIZE + 1$), set the `dbd` proto value to `DBD_DZERO`, and attach a `PT_STACK` pregon to it at `USRSTACK_ADDR` (`0x7b03a000` on 32-bit and `0x800003ff'c0000000` on 64-bit). We use the `PT_DATA` pregon's space.

When a shared library is linked to the process, we typically create two `PT_MMAP` pregon regions — an `RT_SHARED` pregon containing text mapped into a shared quadrant with a space of 0 (`KERNELSPACE`), and an `RT_PRIVATE` pregon containing associated data (such as library global variables) with the `PT_DATA` pregon's space. If `VA_WRTEXT` is set, the data pregon takes the first available address above 0 (in the first or second quadrant); otherwise it is assigned the first available address in the processes data quadrant.

Slide: Virtual Memory and exit()

Virtual Memory and exit()



freed memory

a696184

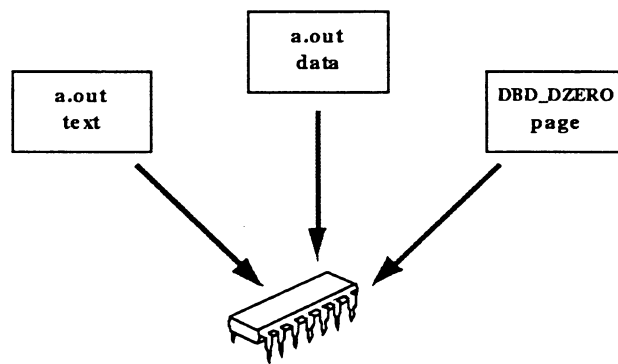
Notes:

Slide: Virtual Memory and exit()

From the virtual memory perspective, an `exit()` is the same as the first part of an `exec()`. We dispose of our virtual memory resources, but we don't allocate new ones. Thus, if we exit from a `vfork` child before the child has performed an `exec()`, we have nothing to clean up from a virtual memory perspective except to return resources to the parent process; if we exit from a non-`vfork` child, we dispose of our virtual memory resources by calling `dispreg()`

Slide: Bringing in a Page the First Time (Demand Paging)

Bringing in a Page the First Time (Demand Paging)



*Справка
адресов
в памяти
адресов
загрузка
адресов*

Notes:

*First time - when user requests user
page*

Slide: Bringing in a Page the First Time (Demand Paging)

Traditional UNIX systems used to require bringing in the entire image of an executable before executing even a single instruction. This could be quite wasteful and time consuming, as well as placing a sometimes heavy demand on free memory. Modern versions of UNIX (4.2 BSD, System V.2.2) started using demand loaded programs. By default, all HP-UX processes are *load-on-demand*.

In general, a demand paged process does not preload a program before it is executed. The process code and data are stored on disk and loaded into physical memory on demand in page increments. Programs often contain routines and code that are rarely accessed. For example, error handling routines might constitute a large percentage of a program and yet may never be accessed. In HP-UX, pages are loaded only when they are actually needed; only then are the pages allocated physical memory and loaded from disk or otherwise initialized. We refer to this as *faulting in* a page.

Recall that when we initialize our regions, we set the dbd proto value to DBD_FSTORE for text and initialized data, and to DBD_DZERO for stack and uninitialized data. In all cases, we set the data part of the dbd to DBD_DINVAL (0xffffffff).

When a process first accesses a page, it will trigger a TLB miss fault because the physical page (and therefore its translation in the sparse PDIR) doesn't yet exist. The fault handler is responsible for bringing in the page and restarting the instruction that faulted. In determining whether the page is valid or not, the fault handler has determined which pregon in the faulting process contains the faulting address. The fault code eventually calls *virtual_fault()*. The arguments passed to this routine are the virtual address causing the fault, the pregon containing that virtual address, and a read/write flag.

We search the btree for our page's vfd and dbd. If the valid bit in the vfd flag is set, we're done — another process beat us to it. If the `r_zomb` flag is set in the region,¹ we print the **pid %d killed due to text modification or page I/O** error message and return SIGKILL (which the handler will send to the process).

Faulting In A DBD_DZERO Page (uninitialized data, stack)

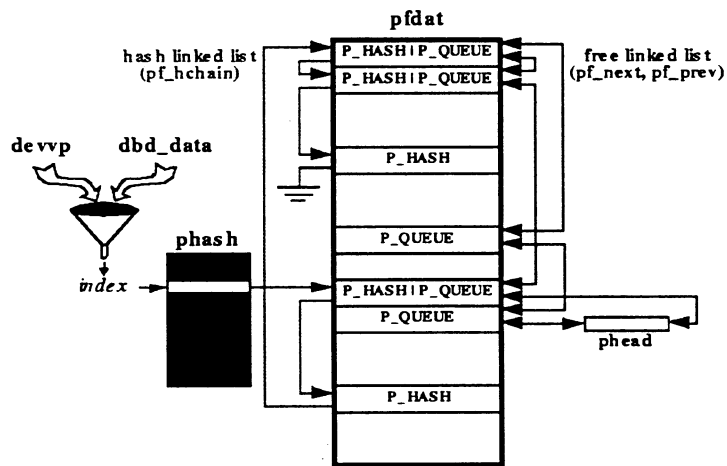
We obtain a free pfdat entry from phead, place its pfn (pf_pfn) in the vfd, set the vfd's valid bit, and increment the region's `r_nvalid` counter (number of valid pages). Then we add a virtual-to-physical translation to the sparse PDIR and zero the page. We change the dbd contents to DBD_NONE and 0xffff0c.² Finally, we add a virtual-to-physical translation for the page to the sparse PDIR.

¹ See page 11-48.

² When the dbd is marked as type DBD_NONE, the kernel uses various unique values in the dbd_type field so that on a DBD_NONE panic the analyzing engineer can tell what code path resulted in the DBD_NONE type.

Slide: Reclaiming Pages from the Free List

Reclaiming Pages from the Free List



a696186

Notes:

Arrows pointing to the P_HASH and P_QUEUE entries in the pfdat stack.

Slide: Reclaiming Pages from the Free List

We call the `r_fstore` vnode's `v_op->vn_pagein()` routine, which is set to `ufs_pagein()`, `nfs_pagein()`, `cdfs_pagein()`, `vx_pagein()`, or any other file system we happen to add. Each of these is responsible for recovering the correct page from the free list or reading the correct page from the disk.

Each of these routines calls `vm_no_io_required()`, which checks for the faulted page on the hash list:

- We get the device vnode pointer (`devvp`) by calling `r_fstore`'s `v_op->vn_bmap()`. This vnode pointer points to the actual disk device (such as `/dev/vg00/lvol5`) rather than to the file referenced by `r_fstore`.
- If the `dbd` data field is `DBD_DINVAL`, we call `r_fstore`'s `v_op->vn_mapdbd()` to get the actual location of the disk block on the disk device and store this value in the `dbd` data field. This will be true the first time the page is retrieved from front store. Subsequent retrievals will already have a valid data field because we set it the first time.
- We call `pageincache()` with the device vnode pointer and the `dbd` data to determine if the faulted page is on the hash list.

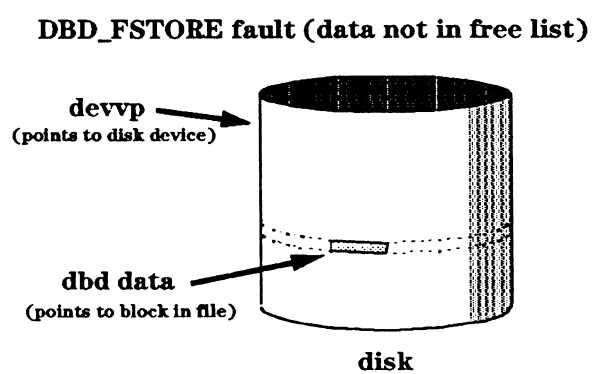
The `pageincache()` routine hashes on the vnode pointer and data to choose one of the `pfdat` pointers in `phash[]`. The hash algorithm is `((data >> 5) + data) xor (vp >> 6) & phashmask`. It walks the `pf_hchain` chain of `pfdat` entries looking for a matching vnode pointer (`pf_devvp`) and data value (`pf_data`). If it finds a match, it removes it from the free list.

- If `pageincache()` returns a `pfdat` entry, we increment the region's valid page count (`r_nvalid`), update the `vfd` with the `pfn` (`pf_pfn`), and add a virtual-to-physical translation for the page to the sparse `PDIR` if it had been removed.

On successfully finding the page in the free list, `vm_no_io_required()` returns a 1. No I/O is required to retrieve the page. This is called a *soft page fault*.

Slide: Retrieving Pages from Disk

Retrieving Pages from Disk



a606187

Notes:

Slide: Retrieving Pages from Disk

If the page is not found in the free list, the *vn_pagein()* routines know which page to fetch because it was stored in the dbd by *vm_no_io_required()*. The routines also schedule read-ahead pages for I/O. The number of read-ahead pages is based on the value of *p_pagein* in the pregon. This value is adjusted based on whether the file is being accessed randomly or sequentially. If it is being accessed **randomly**, we want to **minimize** the number of read-ahead pages; if it is being accessed **sequentially**, we want to **maximize** the number of read-ahead pages. However, we won't read past the end of this pregon's pages.

- Each time we schedule I/O for this pregon, we set *p_nextfault* in the pregon structure to the page we would expect to read next if we were to read ahead further.
- If our next page fault matches *p_nextfault*, we are accessing the file sequentially. We multiply *p_pagein* by two (but to no more than *maxpagein_size*, a global set to 64). If *p_strength* is less than 100 (defined as PURELY_SEQUENTIAL) we increment *p_strength*.
- If our next fault doesn't match *p_nextfault*, we are accessing the randomly. We divide *p_pagein* by two (to no less than *minpagein_size*, a global set to 1). If *p_strength* is greater than -100 (defined as PURELY_RANDOM) we decrement *p_strength*.

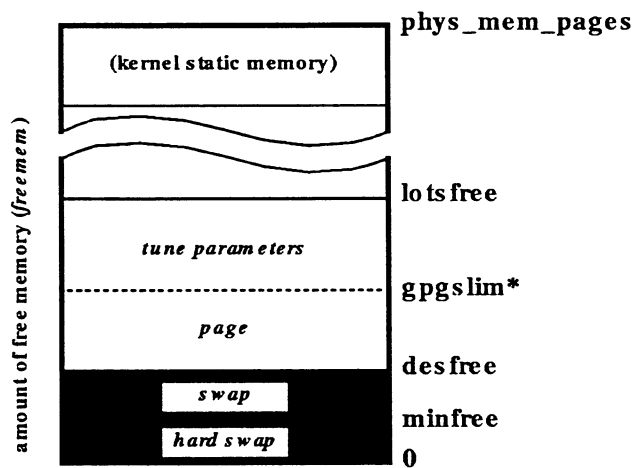
So we can see that *p_strength* will vary between -100 and 100, and *p_pagein* will vary by powers of two between 1 and 64.

We allocate a page of memory from phead, add a virtual-to-physical translation to the sparse PDIR, schedule the I/O from the disk to the page, and sleep waiting for the non-read-ahead I/O to complete (we don't wait for the read-ahead I/O to complete). The vfd is marked valid, and the dbd is left with *dbd_type* set to DBD_FSTORE and *dbd_data* set to the block address on the disk.

Regardless of whether the page data is retrieved from zero-fill, free list, or disk, the pde has been touched. The instruction is retried and gets a TLB miss fault; the miss handler writes the modified pde data into the TLB; the instruction is retried again and succeeds.

Slide: When to Push Pages Out

Push Pages Out



* floats between desfree and lotsfree

a606188

Notes:

Slide: When to Push Pages Out

The memory management system monitors the amount of available memory, also called free memory, in the system and when memory falls below a certain threshold, attempts to free up more memory. This is described in detail in the upcoming section, but first an overview of the circumstances under which we invoke the various memory managers.

A number of kernel global variables, called memory management parameters, are used to monitor the amount of free memory in the system and control when and what method (paging or swapping) is used to free additional memory.

Memory Management Load Measures

freemem: Number of free pages in the phead linked list

parolem: Number of pages expected to be free soon (that is, they are “on parole”) because they have been scheduled for I/O.

lotsfree (**tunable**): Plenty of free memory, specified in pages. When the number of free pages falls below *lotsfree*, *vhand()* tunes its parameters each time it is awakened. On systems with 32 MBytes of free memory or less after kernel initialization (referred to here as *small-memory systems*), *lotsfree* is set to 1/8 of non-kernel memory, not to exceed 256 pages (1 MByte). On systems with more than 32 MBytes of free memory after kernel initialization (referred to here as *large-memory systems*), *lotsfree* is set to 1/16 of non-kernel memory, not to exceed 8192 pages (32 MBytes). If non-kernel memory exceeds 2 GBytes, *lotsfree* is set to 16384 pages (64 MBytes). These default values may be overridden by explicitly setting *lotsfree*.

desfree (**tunable**): Amount of memory desired free, specified in pages. When the number of free pages falls below *desfree*, the swapper begins looking for entire processes to swap out of memory. On small-memory systems, *desfree* is set to 1/16 of non-kernel memory, not to exceed 60 pages (240 Kbytes). On large-memory systems, *desfree* is set to the 1/64 of non-kernel memory, not to exceed 1024 pages (4 MBytes). If non-kernel memory exceeds 2 GBytes, *lotsfree* is set to 3072 pages (12 MBytes). These default values may be overridden by explicitly setting *desfree*, but *desfree* must be less than *lotsfree*. swap

minfree (**tunable**): Minimal amount of free memory which is tolerable, specified in pages. When the number of free pages falls below *minfree*, the swapper recognizes that the system is desperate for memory and attempts to swap out a process whether it is running / runnable or not (referred to as *hard swap*). On small-memory systems, *minfree* is set to 1/2 of *desfree*, not to exceed 25 pages (100 KBytes). On large-memory systems, *minfree* is set to 1/4 of *desfree*, not to exceed 256 pages (1 MByte). If non-kernel memory exceeds 2 GBytes, *lotsfree* is set to 1280 pages (5 MBytes). These defaults may be overridden by explicitly setting *minfree*, but *minfree* must be less than *desfree*.

gpgslim: When the system boots, this is set to 1/4 the distance between *lotsfree* and *desfree* ($(desfree + (lotsfree - desfree)/4)$). As the system runs, it moves this value between *desfree* and *lotsfree* as needed. When the sum of free memory and paroled memory drops below *gpgslim*, *vhand()*

Module 7 — Memory Management

Slide: When to Push Pages Out

begins aging and stealing little-used pages in an attempt to increase the available memory above this threshold.

vhandrunrate (**tunable**): Number of times per second `vhand()` is run when the sum of free memory and paroled memory (`freemem + parolem`) is less than `lotsfree`. The default value is 8.

coalescerate: How often `vhand()` should attempt to reclaim unused memory from the kernel allocation buckets. This value starts at 128 — every 128th time `vhand` runs it attempts to return bucket memory to the system. If it is unsuccessful, it multiplies `coalescerate` by two (checks half as often) up to every 512th time. If it is successful, it resets `coalescerate` to every 128th time. (If `coalescerate` is set to zero via *adb*, the system will never try to return bucket memory to the system.)

memzeroperiod (**tunable**): Minimum time period between events of `freemem` reaching zero. This parameter determines how often `gpgslim` is adjusted when `vhand()` is running. Every $2 * \text{memzeroperiod}$ seconds, or when `freemem` has reached zero twice (whichever condition is satisfied first), `gpgslim` is adjusted. It is moved up towards `lotsfree` if `freemem` did not reach zero twice within `memzeroperiod`; it is moved down towards `desfree` if `freemem` did reach zero twice within `memzeroperiod`. The default value for `memzeroperiod` (in tenths of a second) is 30, or 3 seconds. Due to the algorithm used to adjust `gpgslim` ($[3 * \text{gpgslim} + (\text{either } \text{lotsfree} \text{ or } \text{desfree})] / 4$), it may range slightly below `desfree` or slightly above `lotsfree`.

hitzerofreemem: Number of times in the last `memzeroperiod` `freemem` has reached zero.

targetcpu (**tunable**): Maximum percentage of CPU cycles we want `vhand()` to take. The default value (in tenths of a percent) is 100, or 10.0%.

maxqueuetime (**tunable**): Affects how many pending pageouts we allow per second. The system will try to keep `parolem` below the number of pageouts per second (a moving number) times `maxqueuetime`. The default value (in tenths of a second) is 10, or 1 second; this means the system will try not to schedule more pageouts than the system can handle in one second.

pageoutcnt: Recent count of pageouts finished.

pageoutrate: Current pageout rate, calculated from `pageoutcnt`.

maxpendpageouts: The number calculated from `maxqueuetime`: $\text{pageoutrate} * (\text{maxqueuetime} / 10)$.

max_pgrate: Maximum `pageoutrate` seen in the last `pgrateupdate` period. Used to determine if the system is thrashing. Thrashing is defined as low CPU usage with a high paging rate when either several processes are running, or several processes are waiting for I/O to complete, or there are active processes which have been marked for serialization. It indicates that the system is spending too much time paging in and out, and not enough time doing real work.

curr_pgrate: Maximum `pageoutrate` seen in the current period.

pgrateupdate (**tunable**): How often in seconds to copy `curr_pgrate` to `max_pgrate`. Default value is 60.

Slide: When to Push Pages Out

agehand, stealhand: The hands of `vhand()`, of type `pregion` pointer. These hands walk the active `pregion` list, created by `p_forw` and `p_back` in each active `pregion`. There is also a dummy `pregion`, `bufcache_preg`; when a hand points to it, pages in the buffer cache are aged or stolen.

agerate, stealrate: How many pages the age hand and steal hand should visit per second, respectively. These parameters are continually adapting to system load. These are defined in the kernel as static variables, meaning they don't appear in the symbol table.

handlaps, targetlaps: The actual and desired distance in laps, respectively, between the age hand and the steal hand. `targetlaps` is defined in the kernel as a static variable, meaning it doesn't appear in the symbol table.

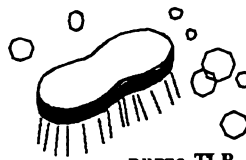
vhandinfoticks (tunable): This is a very useful global variable for debugging paging problems. When set to a nonzero value with `adb`, then every `vhandinfoticks` ticks (a tick is 10 msec) `vhand` (when running) will print information to the console. On a system that is so swamped that normal utilities such as `monitor` and `vmstat` can't run, you can still get timely information by setting this parameter (for example, setting it to 500 results in a line of information every five seconds). It prints the values of `handlaps`, `targetlaps`, `agerate`, `gpgslim`, `parolem`, `stealrate`, and `freemem`.

Slide: vhand — the Page Daemon

vhand — the Page Daemon

Aging

pde
pde_ref < 0



purge TLB

Accessing



TLB miss



pde
pde_ref < 1



write TLB

a696189

Notes:

Slide: vhand — the Page Daemon

Each *vhandrunrate* (default 8) times per second, a routine called `schedpaging()` runs. If the supply of free memory is less than `lotsfree`, it wakes up `vhand()`. The `vhand()` routine then schedules itself on the callout list with a `timeout()` call to run again `hz / vhandrate` ticks later.

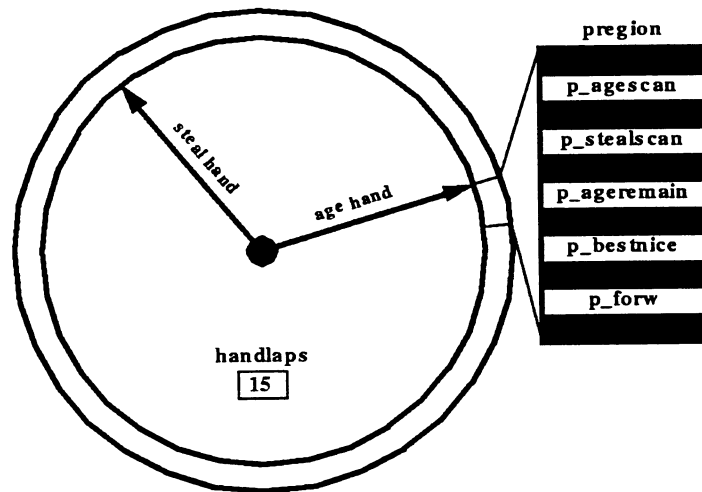
The page daemon can also be awakened by `allocpfd()`, a routine that allocates a page of memory. If all the pages on the free memory list (`phead`) are locked, or the routine has been called while using the interrupt control stack (ICS) and all pages on the free list are also in the page cache (`phash`), `allocpfd()` is not able to get any pages.¹ If on the ICS without any available pages, `allocpfd()` wakes the page daemon. Regardless of which stack the system is running on, `allocpfd()` then wakes up `unhashdaemon`, who's job it is to remove a number of pages from the page cache. If on the ICS, `allocpfd()` will then return `NULL`; if not on the ICS, `allocpfd()` will sleep waiting for a page to become available, and will then retry.

When free memory falls below `gpgslim`, the page daemon attempts to free enough pages to bring the supply of memory back up to `gpgslim`. Between `gpgslim` and `lotsfree`, the page daemon continues to age pages (that is, clear their reference bits) but no longer steals pages. The purpose of the `vhand()` daemon, then, is to clear the reference bits in the `pde` for each page, purge the TLB entry to force the reference through the `pdir`, and then check these bits at a later time to see if they have been referenced. Old pages are then pushed-out to the swap/paging device.

¹ The system is not allowed to call `sleep()` from the ICS, and removing pages from the page cache requires calling `vn_rele()`, which can sleep.

Slide: The Two-Handed Clock Algorithm

The Two-Handed Clock Algorithm



Способ выбора
запрос TLB на
оспаривании. а006100
и смотреть — было ли.

Notes:

Slide: The Two-Handed Clock Algorithm

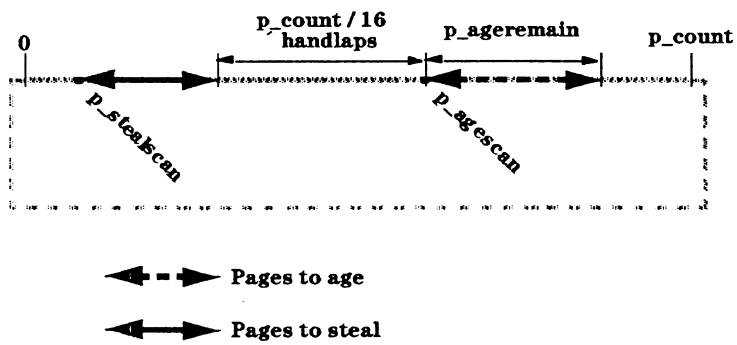
The algorithm used to select pages to page out is called a *two-hand clock algorithm*. The general notion of the clock algorithm is to scan physical memory looking for memory pages that have not been referenced recently and move them to secondary storage — to the swap space. The algorithm uses two “clock hands” walking memory, with the first hand (the *age hand*) clearing the reference bits on pages and the second hand (the *steal hand*) following along sampling the reference bits. Pages that have not been referenced from the time the age hand passes to the time the steal hand passes are pushed out of memory. The hands rotate at a variable rate determined by the demand for memory.

A doubly linked list of preregions, called the *active preregion list*, is used by the page daemon to scan memory. Conceptually speaking, the preregions can be viewed as being wrapped around in a big circle. In the middle of the circle are two hands that resemble clock hands. Walking the *active preregion list* is meant to simulate scanning physical memory.

- When `vhand()` wakes up, we see if it is time to retrieve memory from the kernel memory allocation buckets (see *coalescerate* on page 11-70). We increment *coalescent* and compare its value to *coalescerate*. If it is higher, we try to remove pages from the buckets until we get `freemem` above `lotsfree` and reset *coalescent* to zero.
- Next we update the value of `gpgslim` (see *memzeroperiod* on page 11-70).
- We update *pageoutrate* to approach our actual page out rate, using *pageoutcnt* (number of recent pages successfully written out).
- We update *targetlaps*, the number of laps we want between the age hand and the steal hand. During normal operation, we want the two hands to be as far apart as possible, because it gives processes more time to reset the cleared reference bit if it is really using a page. But this takes more CPU cycles, because the steal hand has to look at more pages to find enough to steal. So if we are taking fewer CPU cycles than *targetcpu* says to take, we try to increase *targetlaps* (but only up to 15); if we are taking more CPU cycles than *targetcpu* says to take, we decrease *targetlaps*.
- We update *agerate*, the number of pages to age per second. If *handlaps* is at the target, *targetlaps*, we converge *agerate* to *stealrate*; otherwise we use the *stealrate* plus 1/4 of the pages needed to catch up with *targetlaps*, or the *stealrate* minus 1/4 of the pages needed to slow down to *targetlaps*.
- If *vhandinfoticks* is non-zero (see page 11-71), we print diagnostic information to the console.

Slide: How Much of the pregion to Age and Steal

How Much of the pregion to Age and Steal



a866101

Notes:

Slide: How Much of the pregon to Age and Steal

Now we are ready to move the hands. Logically, we think of the age hand moving first and then the steal hand moving. But actually the steal hand moves first (for reasons that should become obvious).

- If the steal hand is pointing to `bufcache_preg`, we steal buffers from the buffer cache with `stealbuffers()`. The global parameter `dbc_steal_factor` determines how much more aggressively we steal buffer cache pages than pregon pages. A value of 16 says to treat buffer cache pages no differently than pregon pages; the default value of 48 says to steal buffer cache pages three times as aggressively as pregon pages.

We steal pages even if we're using a "fixed" buffer cache. Its name is misleading. A fixed buffer cache will be sized anywhere from zero pages to `bufpages` pages.

- If the steal hand points to a pregon/region with no valid pages (`r_nvalid == 0`), we can push its btree out to the swap device.
- If none of the processes using this region are loaded in memory (`r_incore == 0`), we can push the entire region out to the swap device, not just a sixteenth of it.
- Otherwise, we may steal all pages between `p_stealhand` and `(p_agescan - p_count/16 * handlaps)`, as shown in the slide. If our steal quota (calculated from `stealrate`) is met without stealing all the pages, we'll stop short.
- Finally, we update `p_stealscan` to be the page number in the pregon after the last page we have stolen.

If we haven't stolen enough pages (calculated from `stealrate`), we move on to the next pregon and repeat the process until we have satisfied the system's demands.¹

Next we move the age hand to clear the reference bit from a selected number of pages.

- If the age hand is pointing to `bufcache_preg`, we age one sixteenth of the pages in the buffer cache with `agebuffers()`.
- We find the best nice value (that is, lowest number) of all the pregon regions using this region. For each page in the region, if the nice value is greater than a randomly generated number, we don't age this page.
- Otherwise, we age all pages between `p_agehand` and `(p_agehand + p_ageremain)` by clearing the `pde_ref` bit and purging the TLB.
- Finally, we update `p_agehand` to be the page number in the pregon after the last page we have aged.

If we haven't aged enough pages (calculated from `agerate`), we move on to the next pregon and repeat the process. Note that we moved the steal hand first to keep it behind the age hand. If we moved the age hand first, we could possibly age and steal a page in the same cycle.

¹ The mechanism of choosing where on the swap devices to write the page will be discussed next.

Slide: Reserving Swap Space

Most UNIX systems and UNIX-like systems allocate swap space at the time it is needed. At first glance this sounds like a straightforward idea, but if the system ever runs out of swap space and needs to write a process's page(s) to a swap device, it has no alternative but to kill the process.

To alleviate this problem, HP-UX uses the concept of *swap reservation*. When a new process is forked or executed, if there is not enough unreserved swap space to handle the entire process, it is not allowed to start.

At system startup, *swapspc_cnt* and *swapmem_cnt* are initialized to the total amount of swap space and pseudo-swap available. Any time the *swapon()* call is made, *swapspc_cnt* is adjusted by the amount of swap added.

A process “reserves” swap simply by decrementing these two counters. The kernel doesn't actually assign disk blocks until they are needed. Swap space is used up first (until *swapspc_cnt* = 0). When a process actually has to allocate swap, it knows it has enough room to do so.

If there is no device/file system swap space, the system uses pseudo-swap as a last resort. It decrements *swapmem_cnt* and locks the pages into memory. Pseudo swap is either free or allocated — it is never reserved.

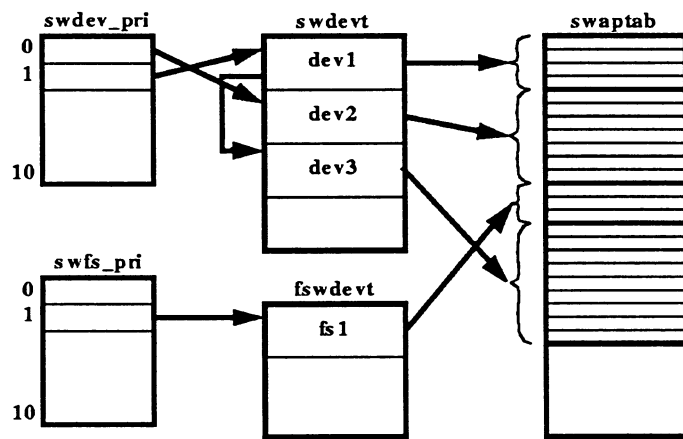
Here is how we calculate the various swap space values:

- Total swap available on the system is *swapspc_max* (for device swap and file system swap) + *swapmem_max* (for pseudo-swap).
- Allocated swap is *swapspc_max* - [*sum*(*swdevt*[*n*].*sw_nfpgs*) + *sum*(*fswdevt*[*n*].*fsw_nfpgs*)] (for device swap and file system swap) + (*swapmem_max* - *swapmem_cnt*) (for pseudo-swap).
- Reserved swap is *swapspc_cnt* - allocated swap (for device swap and file system swap).

In HP-UX, the only reasons a process will die for lack of swap space are because of data area growth (using *sbrk()*) or stack growth.

Slide: Choosing a Swap Location

Choosing a Swap Location



a006103

Notes:

Больше заправ. только если нет свободных

Выбор устройства по приоритетам из /etc/fstab

При добавлении на хост файла и при запуске системы будет на будет, только текущее

Slide: Choosing a Swap Location

All swap devices and file systems enabled for swap have an associated *priority* ranging from 0 to 10. The priority is specified in the priority parameter to the *swapon(1M)* command. The priority indicates the order in which the swap space from this device or file system is used. Space is taken from (numerically) lower priority systems first (that is, starting at priority 0). A data structure called *swdev_pri[]* is used to link together swap devices with the same priority. That is, the entry in *swdev_pri[n]* is the head of a list of swap devices having priority *n*. The first field in *swdev_pri[]* structure is the head of the list; the *sw_next* field in the *swdevt[]* structure links each device into the appropriate priority list. The *swfs_pri[]* data structure serves the same purpose for file system swap.

The first rule of swap space allocation is to start at the lowest priority swap device or file system. In this slide, the lowest priority associated with a swap location is priority 0; two devices have priority 0. The second rule of swap space allocation states that if multiple devices have the same priority, swap space is allocated from the devices in a round robin fashion. Thus, to interleave swap requests between a number of devices, the devices should be assigned the same priority. Similarly, if multiple file systems have the same priority, requests for swap are interleaved between the file systems. In the slide, swap requests are initially interleaved between the two swap devices at priority 0.

Now suppose that both of the devices at priority 0 are full. The system represented by the slide also has a device at priority 1 and a file system at priority 1. The third rule of swap space allocation states that if a device and a file system have the same priority, all the space is allocated from the device before any space is allocated from the file system. Thus, the device at priority 1 will be filled before swap is allocated from the file system at priority 1.

Slide: Choosing a Swap Location

Device swap table (struct swdevt):

sw_dev: The major (upper 8 bits) and minor (lower 24 bits) numbers of the swap device.

sw_start: Beginning of swap area on disk, in Kbytes.

sw_nblksavail: Size of the swap area, in Kbytes.

sw_nblkseabled: This has to be a multiple of *swchunk*, which defaults to 2 Mbytes. So *sw_nblkseabled* may be smaller than *sw_nblksavail*.

sw_nfpgs: Number of free pages on this swap device. This is updated any time a page is used or freed.

sw_priority: Priority of this swap device (0 - 10). This can also be determined by which *swdev_pri[]* linked list we're on.

sw_head, *sw_tail*: Index into *swaptab[]* of the first and last entries associated with this swap device. The *swaptab* element *sw_next* creates a linked list of *swaptab* entries for this swap device.

sw_next: Pointer to the next *swdevt* entry at this priority. This is a circular list, so if this is the only device on this priority it will point to itself. We use this to update the pointer in *swdev_pri* to implement the round-robin use of all devices at a particular priority.

Slide: Choosing a Swap Location

File system swap table (struct fswdevt):

fsw_next: Pointer to next fswdevt entry at this priority. As with sw_next in swdevt, this is a circular list.

fsw_nfpgs: Number of free pages in this file system swap. This is updated any time a page is used or freed.

fsw_allocated: Number of swchunks (default 2 MBytes) actually allocated on this file system swap. Each chunk is a file in the file system swap directory, and has a name constructed from the system name and the swptab index (such as becky.6 for swptab[6] on a system named becky).

fsw_min: Minimum swchunks to be preallocated when this file system swap is enabled.

fsw_limit: Maximum swchunks allowed on this file system. If set to zero, there is no limit.

fsw_reserve: Minimum blocks (of size fsw_bsize) reserved for non-swap use on this file system.

fsw_priority: Priority of this swap device (0 - 10). This can also be determined by which swfs_pri[] linked list we're on.

fsw_vnode: Vnode of the file system swap directory under which the swap files are created. This directory is named `paging` and resides under the file system's root directory.

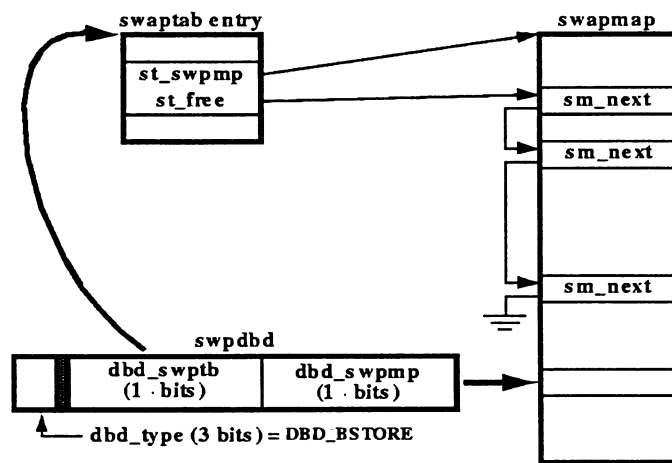
fsw_bsize: Block size used on this file system. This is used to determine how much space on the file system *fsw_reserve* is reserving.

fsw_head, *fsw_tail*: Index into *swptab[]* of the first and last entries associated with this file system swap. The *swptab* element *sw_next* creates a linked list of *swptab* entries for this file system swap.

fsw_mntpoint: The character representation of *fsw_vnode*. This is used for utilities (such as *swapinfo(1M)*) and error messages.

Slide: The swaptab/swapmap Structure

The swaptab/swapmap Structure



a696194

Notes:

Slide: The swaptab/swapmap Structure

Each swaptab[] entry has a pointer to a unique swapmap, called *st_swmp*. There is one entry in the swapmap for each page represented by the swaptab entry (default 2 MBytes, or 512 pages); in other words, it is sized by swchunk.

We maintain a linked list of free swap pages beginning at the swaptab entry's *st_free* and using each free swapmap entry's *sm_next*. When we need a page of swap, we walk the structures.

- Beginning with the lowest priority, we begin by examining *swdev_pri[]*.curr, which points to a swdevt entry.
- If *sw_nfpgs* is zero (no free pages), we follow the pointer *sw_next* to get the next swdevt entry at this priority.
- If none of these have free pages, we move on to *swfs_pri[]*.curr, the file system swap at this priority, checking *fsw_nfpgs* for free pages.
- If we are still unsuccessful, we move to the next priority and try again.
- Once we find a swdevt or fswdevt with free pages, we walk that device's swaptab list, starting with *sw_head* or *fsw_head*, and using *st_next* in each swaptab entry, until we find a swaptab entry with non-zero *st_nfpgs*.
- *st_free* points to the first free swapmap entry (and thus first free page) in this swaptab chunk.
- We create a dbd using 14 bits of *dbd_data* for the swaptab index and 14 bits for the swapmap index. We set *r_bstore* in the region to the disk device vnode or the file system directory vnode, and mark the dbd *DBD_BSTORE*. Now we have stored all the information we need to be able to retrieve the page from swap.

Slide: The swaptab/swapmap Structure

Swap table entry (struct swaptab):

st_free: First free page in this chunk (since each swapmap entry maps to a 4 Kbyte page of swap, we can consider this to either be an index to the free page or an index to the free swapmap entry).

st_next: Next swaptab entry for the same device or filesystem swap. If we are at the end of the list, *st_next* will be -1.

st_flags: Various flags:

ST_INDEL (0x01): In the case of file system swap, this chunk is being deleted. Don't allocate any pages from it. Since this is only set via a routine called *realswapoff()*, which is never called, this flag shouldn't be set.

ST_FREE (0x02): In the case of file system swap, this chunk may be deleted, because none of its pages are in use. In the case of remote swap, we don't want to delete the chunk immediately, so we set *st_free_time* to current time plus 30 minutes (1800 seconds) at the time we set this flag. Once that time has passed, we can free the chunk. If the chunk is needed again in the interim, we can clear this flag.

This mechanism (*chunk_release()*) is called from *lsync()*.

ST_INUSE (0x04): Swaptab entry is being changed.

st_dev, *st_fsp*: One or the other of these is used to point back to the *swdevt* or *fswdevt* entry which references this swaptab entry.

st_nfpgs: Number of free pages in this swchunk.

st_swmp: Pointer to swapmap array that defines this swchunk of swap pages.

st_free_time: See explanation for *ST_FREE* flag.

Slide: The swaptab/swapmap Structure

Swap map entry (struct swapmap):

sm_ucnt: Number of threads using this page. When this goes to zero, the swap page is free and we update the free pages linked list.

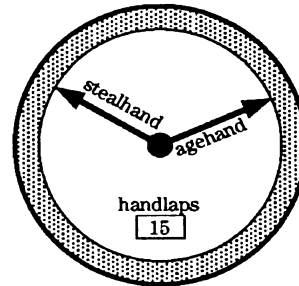
sm_next: Next free page. This is only valid if *sm_ucnt* is zero; that means that this swapmap entry is included in the linked list beginning with swaptab's *st_free*.

When faulting in from swap, we follow the same process as we did when faulting in from the file system on page 11-65 — we hash *r_bstore* and *dbd_data* together, check for a soft fault, then call *devswap_pagein()*. This routine knows to use the *dbd_data* as a 14-bit swaptab index and a 14-bit swapmap index; from these, it can determine the location of the page on disk.

Slide: Swapping Using the Pager

Swapping Using the Pager

Deactivate



a696195

Notes:

More frame to chase
— 5-10 min, no 370
Exception

Slide: Swapping Using the Pager

Since the pager is tuned to be nice regarding I/O usage and CPU usage, the swapper allows the pager to fault out swapped processes. The swapper marks the process to be swapped for *deactivation*, which takes it off the run queue. Since it can't run, once its pages are aged they can't be referenced again. When the steal hand comes around, it can steal all the pages in the region.

When memory pressure is high, the swapper intelligently picks a process to swap using the routine *choose_deactivate()*. This routine is biased to choose non-interactive processes over interactive ones; to choose sleeping processes over running ones; and to choose processes that have been running longest over newer processes.

In the process that is chosen, we:

- set the process' SDEACT flag and set its threads' TSDEACT flags;
- remove its threads from the run queue;¹
- set the proc's *p_deactime* to the current time so we will know how long this process has been deactivated;
- move the process in the active pregion chain so that it is directly in front of the steal hand (the age hand shouldn't be far behind);
- add the uarea pregion to the list of active regions so it can get paged out, too;
- increment the global counter *deactive_cnt*.

A process that has been inactive long enough that all its pages have been aged and stolen is almost swapped already. A list of these inactive processes is kept with the head pointed to by the global *deactprocs*, with the chain running through the pregion element *p_nextdeact*. If the average number of free pages drops below *lotsfree*, we go ahead and swap these (which essentially consists of paging out the *uarea*).

When memory pressure eases, it is time to reactivate a deactivated process. This time we call *choose_reactivate()*, which is biased to choose interactive processes over non-interactive ones; to choose runnable processes over sleeping ones; and to choose processes that have been deactivated longest over those that have been recently deactivated.

Besides clearing the deactivation flags in the *proc* table entry and in the threads, all we have to do is fault in the *uarea*. All other pages are allowed to fault in as needed.

¹ If we can't deactivate the process because it is waiting for I/O, we set its SDEACTSELF flag and the threads' TSDEACTSELF flags. The process is then deactivated in the paging routines when I/O is complete.

Module 7 — Memory Management

Lab: Forking and Address Spaces

The files for this lab are in the directory `/home/tops/labs/mod7`. Just type `make` in this directory to build the example programs for your system.

Fork and virtual address spaces

Running the program `fork1.share_magic`, use `q4` to print out the lists of **pregions** for both parent and child. Use this information to fill in the following table, Also, note the addresses of the two text **pregions** for use later in the lab.

Table 1: pregon information

	parent				child			
	space	vaddr	size	p_reg	space	vaddr	size	p_reg
U_Area								
Stack								
Data								
Text								

Parent text pregon address (addrof pseudo field) _____

Child text pregon address (addrof pseudo field) _____

```
ken@tiger[mod7] make
cc -Ae +DA1.1 -Wl,-a,archive +ESlit -O -o fork1.share_magic fork1.c
cc -Ae +DA1.1 -Wl,-a,archive +ESlit -O -Wl,-N -o fork1.exec_magic fork1.c
ken@tiger[mod7]

ken@tiger[mod7] ./fork1.share_magic
plock failed: Not owner      ### program attempt to use plock and I'm not root
      buffer      buffer1      buffer2      buffer3      buffer4
buffer addresses 0x4000f560 0x40010000 0x40012000 0x40014000 0x40016000
buffer page #    0xe      0xf      0x11      0x13      0x15
kill -USR1 2135 ; will cause the parent to write buffer1
kill -USR1 2136 ; will cause the child to write buffer2
kill -USR2 2135 ; will cause the parent to read buffer3
kill -USR2 2136 ; will cause the child to read buffer4
kill -INT 2135 ; will cause the parent to write cbuffer
kill -INT 2136 ; will cause the child to write cbuffer
cbuffer buffer is in the text area so you will die if
try this test on the share_magic example.
```

Module 7 — Memory Management

Lab: Forking and Address Spaces

```
q4>
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
q4> keep p_stat
kept 77 of 276 struct proc's, discarded 199
q4> name it all
so named
q4>

q4>
q4> keep p_pid == 2135
kept 1 of 77 struct proc's, discarded 76
q4> name it parent
so named
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4> load struct pregion from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 7 struct pregion's as a linked list (stopped by loop)
q4> name it p_pregs
so named
q4> print -x p_type p_space p_vaddr p_count %d p_reg
  p_type  p_space          p_vaddr p_count  p_reg
PT_UAREA 0x47ad800 0x400003fffffce000    7 0x100215a00
PT_STACK 0x18b5800          0x7f7e6000    6 0x1002e8c00
PT_MMAP   0x18b5800          0x7f7e5000    1 0x100e90e00
PT_DATA   0x18b5800          0x40001000   27 0x1001fca00
PT_TEXT   0x782a400          0x1000       30 0x10021f800
PT_NULLDREF 0x782a400          0           1 0x100530a00
0 0x1 0x100c04e00 12602880 0x100e9a900 --- garbage as vas is in linked list
q4>

q4> recall all
copied 77 items
q4> keep p_pid == 2136
kept 1 of 77 struct proc's, discarded 76
q4>
q4> name it child
so named
q4>
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4> load struct pregion from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 7 struct pregion's as a linked list (stopped by loop)
q4> name it c_pregs
so named
q4> print -x p_type p_space p_vaddr p_count %d p_reg
  p_type  p_space          p_vaddr p_count  p_reg
PT_UAREA 0x1636000 0x400003fffffce000    7 0x100224800
PT_STACK 0x2d93800          0x7f7e6000    6 0x100224c00
PT_MMAP   0x2d93800          0x7f7e5000    1 0x100e48c00
PT_DATA   0x2d93800          0x40001000   27 0x100ae5a00
PT_TEXT   0x782a400          0x1000       30 0x10021f800
PT_NULLDREF 0x782a400          0           1 0x100530a00
0 0x1 0x100d5be00 14007808 0x100ab5600
q4>
```

Lab: Forking and Address Spaces

Note that the two text areas are the same.

The data, stack and **mmap** areas are different.

This system has a 64-bit kernel so the `u_area`, which is kernel data, and is the way out in 64-bit address land.

Module 7 — Memory Management

Lab: Looking at the Text Area

This is an advanced lab. If you don't have time, skip to the lab on the data areas.

The two processes share the same region. All of the **pregions** using the same region can be found by looking at the skip list in the region structure.

Load the region structure from one of the two text pregrions and fill in the following information:

Type _____
Size in pages (r_pgsz) _____ does this match the size reported by the pregrion ?
Reference count (r_refcnt) _____
Front store and back store _____ & _____

```
q4> keep p_type == PT_TEXT
kept 1 of 7 struct pregrion's, discarded 6
q4>
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4>
q4>
q4> print -tx
```

```
    indexof  0
    mapped   0x1
    spaceof  0
    addrof   0x10021f800
physaddrof  0x1875800
    realmode 0
    r_flags  RF_ALLOC
    r_type   RT_SHARED
    r_pgsz  0x2f
    r_nvalid 0x1d
    r_dnvalid 0
    r_swalloc 0
    r_swapmem 0
r_vfd_swapmem 0
    r_lockmem 0
    r_pswapf  0
    r_pswapb  0
    r_refcnt 0x3
    r_zcomb  0
    r_off    0
    r_incore 0x3
    r_dbd    0
    r_scan   0
    r_fstore 0x1008d0c00
    r_bstore 0x1008d0c00
    r_forw   0x1001fca00
    r_back   0x100215a00
    r_hchain 0
    r_byte   0
    r_bytelen 0
```

Module 7 — Memory Management

Lab: Looking at the Text Area

```
    r_lock.interlock 0x1638d80
      r_lock.delay 0
    r_lock.read_count 0
    r_lock.want_write 0
    r_lock.want_upgrade 0
      r_lock.waiting 0
      r_lock.no_swap 0x1
    r_mlock.b_lock 0
      r_mlock.order 0x5a
      r_mlock.owner 0
        r_poip 0
        r_root 0x10068df00
          r_key 0
          r_chunk 0x100a6c800
          r_next 0x1001fca00
          r_prev 0x100ae5a00
    r_preg_un.r_un_pregskl 0x10021f8e8
    r_preg_un.r_un_pregion 0x10021f8e8
    r_psklh.l_header.n_next[0] 0x100a79e40
    r_psklh.l_header.n_next[1] 0
    r_psklh.l_header.n_next[2] 0
    r_psklh.l_header.n_next[3] 0
      r_psklh.l_header.n_prev 0
      r_psklh.l_header.n_key 0
      r_psklh.l_header.n_value 0
      r_psklh.l_header.n_flags 0
      r_psklh.l_header.n_cookie 0
        r_psklh.l_tail 0x100a40940
        r_psklh.l_cache 0x100a79e40
        r_psklh.l_cmpf 0
        r_psklh.l_level 0
        r_psklh.l_cookie 0
          r_excproc 0
          r_lchain 0
          r_mlockswap 0
          r_pgszhint 0x1
          r_hdl.r_space 0x782a400
          r_hdl.r_prot 0x6210
          r_hdl.r_vaddr 0
          r_hdl.r_hdlflags 0x14
```

The size of the pregon and the region are different, The program text is not at the start of the executable file. There is a header. The pregon has an offset inside the region. The field **p_off** in the pregon would have given this.

The field **r_psklh.l_tail** points to the list of **sknode_t**'s.

Load this list and print out the **n_value** fields, these are the addresses of the pregon, compare this against the pregon addres of the parent and child. Only the ones loaded should have a value of zero, keep the other one that does not match.

Module 7 — Memory Management

Lab: Looking at the Text Area

```
q4>
q4> load sknode_t from r_psklh.l_tail next n_prev max 100
q4>
q4> print -x addrrof n_next[0] n_prev n_value n_flags
      addrrof  n_next[0]      n_prev      n_value n_flags
0x100a40940      0 0x100ab7540 0x100e98000      0x3 --- PSEUDO VAS
0x100ab7540 0x100a40940 0x100a79e40 0x100e9a900      0x3 --- parent text
0x100a79e40 0x100ab7540 0x10021f8e8 0x100ab5600      0x3 --- child text
0x10021f8e8 0x100a79e40      0      0      0
q4>

q4> recall p_pregs
copied 7 items
q4> keep p_type == PT_TEXT
kept 1 of 7 struct pregion's, discarded 6

q4> print -x addrrof
      addrrof
0x100e9a900
q4>

q4> forget 0
q4> forget 0
q4>
q4> recall c_pregs
copied 7 items
q4> keep p_type == PT_TEXT
kept 1 of 7 struct pregion's, discarded 6
q4> print -x addrrof
      addrrof
0x100ab5600
q4>
q4> forget 0
q4> forget 0

q4> keep indexof == 0
kept 1 of 4 struct sknode's, discarded 3
```

This sknode should now point to the pregion belonging to the pseudo vas of the executable. Load this pregion and fill in ..

The flags	_____
Type	_____
Space	_____
Vaddr	_____
Offset within the region	_____ does this account for the difference seen earlier?
p_vas	_____

Is this pregion on the global linked list of pregions (p_forw & p_back)? Where the ones for the text areas?

Module 7 — Memory Management

Lab: Looking at the Text Area

```
q4> load struct pregion from n_value
loaded 1 struct pregion as an array (stopped by max count)
q4> print -tx
    indexof  0
    mapped   0x1
    spaceof  0
    addrof   0x100e98000
    physaddrof 0x9798000
    realmode  0
p_ll.lle_next[0] 0x100c7dc00
p_ll.lle_next[1] 0xffffffffffffffff
p_ll.lle_next[2] 0xffffffffffffffff
p_ll.lle_next[3] 0xffffffffffffffff
p_ll.lle_prev  0x100c7dc00
  p_flags  PF_PSEUDO|PF_ACTIVE|PF_EXACT|PF_ALLOC
  p_type   PT_MMAP
  p_reg    0x10021f800
  p_space  0x782a400
  p_vaddr  0xd000
  p_off    0xd
  p_count  0x22
p_ageremain  0
p_agescan    0
p_stealscan  0
  p_vas      0x100c7dc00
  p_tid      0x1
  p_forw     0x100e9a600
  p_back     0x100aebc00
p_regsknode  0x100a40940
p_nestediomap 0
  p_pagein  0x8
  p_bestnice 0x27
p_deactsleep 0
p_locality   0
p_strength   0
  p_prot     0xb
  p_nextfault 0xffffffffffffffff
  p_last_rhead 0xffffffffffffffff
  p_mq_refcnt 0
  p_lchain   0
  p_lactions 0
p_hdl.hdlflags 0x1
  p_hdl.hdlar 0x10
  p_hdl.hdlprot 0x6210
  p_hdl.p_spreg 0
q4>
```

Now go back to the region structure for the text.

The front store points to the vnode for the executable file (vnode are the kernels main file management structure). The vnode has a field `v_vas`. Does this match the `p_vas` field from the above question?

Module 7 — Memory Management

Lab: Looking at the Text Area

q4> history

HIST	NAME	LAYOUT	COUNT	TYPE	COMMENTS
1	<none>	array	276	struct proc	stopped by max count
2	all	mixed?	77	struct proc	subset of 1
3	parent	mixed?	1	struct proc	subset of "all"
4	<none>	array	1	struct vas	stopped by max count
5	p_pregs	list	7	struct pregion	stopped by loop
6	<none>	mixed?	77	struct proc	copy of "all"
7	child	mixed?	1	struct proc	subset of 6
8	<none>	array	1	struct vas	stopped by max count
9	c_pregs	list	7	struct pregion	stopped by loop
12	<none>	mixed?	1	struct pregion	subset of "c_pregs"
13	<none>	array	1	struct region	stopped by max count
14	<none>	list	4	struct sknode	stopped by null pointer
19	<none>	mixed?	1	struct sknode	subset of 14
20	<none>	array	1	struct pregion	stopped by max count
21	<none>	array	1	struct vas	stopped by max count

q4> recall 13

copied 1 item

q4>

q4>

q4> load struct vnode from r_fstore

loaded 1 struct vnode as an array (stopped by max count)

q4> print -tx

```
indexof 0
mapped 0x1
spaceof 0
addrof 0x1008d0c00
physaddrof 0x3ad0c00
realmode 0
v_flag 0x810a
v_shlockc 0
v_exlockc 0
v_tcount 0x2
v_count 0x7
v_vfsmountedhere 0
v_op 0x740c78
v_socket 0
v_stream 0
v_vfsp 0x100743000
v_type VREG
v_rdev 0
v_data 0x1008d0cb0
v_fstype VVXFS
v_vas 0x100c7dc00
v_lock.b_lock 0
v_lock.order 0x5a
v_lock.owner 0
v_cleanblkhd 0x100eb8248
v_dirtyblkhd 0
v_writecount 0
v_locklist 0
v_scount 0x22
```

Module 7 — Memory Management

Lab: Looking at the Text Area

```
v_nodeid 0x28a
v_ncachedhd 0
v_ncachevhd 0xe8c480
v_pfdathd 0x1442500
v_last_fsync 0
```

q4>

Load the vas structure and check it's type (v_flag). Does the next and previous pointers point to the earlier region ?

```
q4> load struct vas from v_vas
loaded 1 struct vas as an array (stopped by max count)
q4>
q4> print -tx
```

```
          indexof 0
          mapped 0x1
          spaceof 0
          addrof 0x100c7dc00
          physaddrof 0x647dc00
          realmode 0
va 11.11e next[0] 0x100e98000
va_11.11e_next[1] 0xffffffffffffffff
va_11.11e_next[2] 0xffffffffffffffff
va_11.11e_next[3] 0xffffffffffffffff
va 11.11e prev 0x100e98000
          va_hilevel 0
          va_pslpath[0] 0x100c7dc00
          va_pslpath[1] 0x100c7dc00
          va_pslpath[2] 0x100c7dc00
          va_pslpath[3] 0x100c7dc00
          va_refcnt 0x4
          va_rss 0
          va_prss 0
          va_dprss 0
          va_proc 0
          va flags VA PSEUDO
          va_fp 0
          va_wcount 0
          va_vaslock.interlock 0x1639620
          va_vaslock.delay 0x186a0
          va_vaslock.read_count 0
          va_vaslock.want_write 0
          va_vaslock.want_upgrade 0
          va_vaslock.waiting 0
          va_vaslock.no_swap 0x1
          va_cred 0
          va_hdl.hdl_textsid 0
          va_hdl.hdl_textpid 0
          va_hdl.hdl_datasid 0
          va_hdl.hdl_datapid 0
          va_hdl.v_hdlflags 0
```

Module 7 — Memory Management

```
        va_ki_vss 0
        va_ki_flag 0
        va_ucount 0x1e
va_runenv.r_machine R_PARISC
        va_runenv.r_arch R_PARISC_2_0
        va_runenv.r_os R_HPUX_OS
va_runenv.r_asmodel R_64BIT
        va_runenv.r_magic R_SHARE_MAGIC
va_lgpg_env.lgpg_data_size 0
va_lgpg_env.lgpg_text_size 0
va_lgpg_env.lgpg_next_brk_inc 0
va_lgpg_env.lgpg_user_brk_cnt 0
va_lgpg_env.lgpg_next_stk_inc 0
        va_winId 0xffffffff
        va_winMap 0
```

q4>

Module 7 — Memory Management

Lab: Looking at the Data Areas

Parent first

Go back to the list of preions for the parent, and keep just the data preion. Then load the region structure.

What is its type? _____

The number of pages _____

How many are valid? _____

```
q4> recall p_pregs
copied 7 items
q4> keep p_type == PT_DATA
kept 1 of 7 struct preion's, discarded 6
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4>

q4> print r_type r_pgsz r_nvalid
      r_type r_pgsz r_nvalid
RT_PRIVATE    27    26
q4>
```

The page count should be small enough not to need to use more than one chunk of vfddbds (a structure holding a pair of vfds and dbds). If there is only one chunk then there is no need for a tree. The `r_chunk` field points to the one chunk of vfddbds.

Looking in the broot structure we can confirm whether there is a tree by checking the depth of the tree. Load the broot and check the depth.

```
q4> load struct broot from r_root
loaded 1 struct broot as an array (stopped by max count)

q4> print b_depth
b_depth
0      so there is no tree
q4>
```

Go back to the region structure and then load the vfddbds for the pages in the region. The field `r_chunk` points to them. The field `r_pgsz` gives the number of vfddbds. Having loaded these structures print out the fields `c_vfd.pgm.pg_v`, `c_vfd.pgm.pg_cw` and `c_vfd.pgm.pg_pfnm` these are the valid bit, the copy on write flag and the physical page number.

Module 7 — Memory Management

Lab: Looking at the Data Areas

```
q4> forget 0
q4> load struct vfddb from r_chunk max r_pgsz
loaded 27 struct vfddb's as an array (stopped by max count)
q4>

q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x \
      c_vfd.pgm.pg_pfnnum %#10x

#   V   CoW      PPN
0  0x1   0      0xa67f
1  0x1  0x1     0xb0e0
2  0x1  0x1     0xb0e1
3  0x1  0x1     0xb0e3
4  0x1  0x1     0xafce
5  0x1   0      0xa67e
6   0   0x1      0
7  0x1   0      0xb10c
8  0x1   0      0xb10d
9  0x1   0      0xb10e
10 0x1   0      0xb10f
11 0x1  0x1     0xae9c
12 0x1  0x1     0xae9d
13 0x1  0x1     0xae9e
14 0x1  0x1     0xae9f
15 0x1  0x1     0xa784 --- buffer1
16 0x1  0x1     0xa785 --- buffer1
17 0x1  0x1     0xa786 --- buffer2
18 0x1  0x1     0xa787 --- buffer2
19 0x1  0x1     0xad50 --- buffer3
20 0x1  0x1     0xad61 --- buffer3
21 0x1  0x1     0xad52 --- buffer4
22 0x1  0x1     0xad63 --- buffer4
23 0x1  0x1     0xacc0
24 0x1  0x1     0xacc1
25 0x1  0x1     0xacc2
26 0x1  0x1     0xacc3

q4>
```

The `forh1.share_magic` program gave the page numbers of the different buffers. Highlight these in your table of ppns.

Module 7 — Memory Management

Lab: Connecting Physical Page Numbers (PPN) to Virtual Addresses

This is an advanced activity. If you are short of time, you may skip it.

Let's try and convert the pfn back to a virtual address.

The physical memory map of the V-Class is not contiguous and so HP-UX 11 does not assume a contiguous memory model. We therefore need to know a few things before we start.

The tables that deal with physical page numbers work in sections of the minimum sized piece of contiguous memory which is defined in <machine/vmparams.h>

```
#define PFN_CONTIGUOUS_PAGES    0x1000 /* 16Mb for 4K pages */
```

We first need to find where a page is in, and then where the header file also defines a pair of macros.

```
#define PFN_BASE(PFN)          ((PFN) / PFN_CONTIGUOUS_PAGES)
#define PFN_OFFSET(PFN)       ((PFN) % PFN_CONTIGUOUS_PAGES)
```

(Programmers bonus question: Can you write efficient versions of these macros?)

```
#define PFN_CONTIGUOUS_PAGES    0x1000 /* 16Mb for 4K pages */
#define PFN_CONTIGUOUS_PAGES_SHIFT 12
#define PFN_CONTIGUOUS_PAGES_MASK 0x0fff

#define PFN_BASE(PFN)          ((PFN) >> PFN_CONTIGUOUS_PAGES_SHIFT)
#define PFN_OFFSET(PFN)       ((PFN) & PFN_CONTIGUOUS_PAGES_MASK)
```

Would be much faster...

The structures for converting physical address to virtual addresses are in the file <machine/pde.h> it then defines

```
#define PFN_TO_VIRT(PFN)       (&pfn_to_virt_ptr[PFN_BASE(PFN)].pp_tbl[PFN_OFFSET(PFN)])
```

So to convert physical pages to struct pfn_to_virt_ptr's

```
q4> 0xa784/0x1000
012    10    0xa
q4>
q4> 0xa784%0x1000
03604    1924    0x784
q4>
```

The physical to virtual information can then be loaded as

```
q4> load struct pfn_to_virt_ptr from pfn_to_virt_ptr skip 10 (10 from the earlier sum)
```


Module 7 — Memory Management

Lab: Connecting Physical Page Numbers (PPN) to Virtual Addresses

```
loaded 1 struct pfn_to_virt_ptr as an array (stopped by max count)
q4> load pfn_to_virt_entry_t from pp_tbl skip 0x784 (the remainder from the calculation)
loaded 1 pfn_to_virt_entry_t as an array (stopped by max count)
q4> print -tx | more
        indexof  0
        mapped   0x1
        spaceof  0
        addrof   0xdc0840
        physaddrof 0xdc0840
        realmode 0
        space 0x18b5800
alias or offset.offset page 0x40010
        alias_or_offset.alias 0x40010
q4>
```

The offset of buffer1 was 0x40010000 and from the pregon we know it's space to be 0x18b5800

0x40010 is the page number, multiplying by 4K (0x1000) gives 0x40010000 so it matches.

Module 7 — Memory Management

Lab: Looking at the Child

Recall the child's pregion list and follow the same procedure to get the list of valid and copy-on-write flags and the physical page number for the child.

```
q4> recall c_pregs
copied 7 items
q4>
q4> keep p_type == PT_DATA
kept 1 of 7 struct pregion's, discarded 6
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4> print r_type r_pgsz r_nvalid
      r_type r_pgsz r_nvalid
RT_PRIVATE      27      26
q4> load struct vfddb from r_chunk max r_pgsz
loaded 27 struct vfddb's as an array (stopped by max count)
q4>

q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x\
          c_vfd.pgm.pg_pfnnum %#10x
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnu
#   V   CoW      PPN
0  0x1  0x1      0x6391
1  0x1  0x1      0xb0e0
2  0x1  0x1      0xb0e1
3  0x1   0      0xa67c
4  0x1  0x1      0xafce
5  0x1  0x1      0xb687
6   0   0x1         0
7  0x1   0      0xa678
8  0x1   0      0xa679
9  0x1   0      0xa67a
10 0x1   0      0xa67b
11 0x1  0x1      0xae9c
12 0x1  0x1      0xae9d
13 0x1  0x1      0xae9e
14 0x1  0x1      0xae9f
15 0x1  0x1      0xa784 --- buffer1
16 0x1  0x1      0xa785 --- buffer1
17 0x1  0x1      0xa786 --- buffer2
18 0x1  0x1      0xa787 --- buffer2
19 0x1  0x1      0xad60 --- buffer3
20 0x1  0x1      0xad61 --- buffer3
21 0x1  0x1      0xad62 --- buffer4
22 0x1  0x1      0xad63 --- buffer4
23 0x1  0x1      0xacc0
24 0x1  0x1      0xacc1
25 0x1  0x1      0xacc2
26 0x1  0x1      0xacc3
q4>
```

Lab: Looking at the Child

Are the pages that correspond to the buffers the same as the parents?

(If this is advanced and you are short of time, skip and look at the data access cases.)

Since both the parent and child are pointing to the same physical pages this will show up in the usage count in the pfdat table. So let's take a look at that using the physical page for the first of buffer's pages (0xa784).

Again we need the PFN_CONTIGUOUS_PAGES converted values so we get the two parts 0xa as the base and 0x784 as the offset. The base is used as the element number in the pfdat_ptr table. The offset can then be used in there.

```
q4> load struct pfdat_ptr from pfdat_ptr skip 0xa
loaded 1 struct pfdat_ptr as an array (stopped by max count)
q4> load struct pfdat from pp_tbl skip 0x784
loaded 1 struct pfdat as an array (stopped by max count)
q4>

q4> print -tx | more
      indexof  0
      mapped   0x1
      spaceof  0
      addrof   0x13e4440
physaddr     0x13e4440
      realmode 0
      pf_hchain 0
      pf_devvp 0
      pf_next  0x14076c0
      pf_prev  0x1032c28
      pf_vnext 0
      pf_vprev 0
      pf_lock.b_lock 0
      pf_lock.order 0x8c
      pf_lock.owner 0
      pf_flags  0
      pf_pfn   0xa784  ----- check that this agrees.
      pf_fill  0
      pf_use   0x2     ----- the reference count = 2 parent and child
      pf_cache_waiting 0
      pf_data  0
      pf_sizeidx 0x20
      pf_size  0x2
      pf_size_fill 0
      pf_group_id 0
      pf_fs_priv_data 0
      pf_pad   ""
      pf_hdl.hdlpf_flags 0x61
      pf_hdl.hdlpf_savear 0xff
      pf_hdl.hdlpf_saveprot 0x5a20
q4>
```

Module 7 — Memory Management

Lab: Reading Translations from the Page Directory

This is an advanced lab

Using `<machine/pde.h>` we can find the hash function used by the kernel to take a space and offset and find the linked list of pde's.

The kernel uses different tables and structures for 32bit and 64bit hardware.

Table 2:

	32bit	64bit
start of table	htbl	htbl2_0
size of table	nhtbl	nhtbl2_0
data type	struct pde	struct pde2_0

The header file defines the macros as

```
#define pdirhash1_1(sid, sof) ((sid << 5) ^ ((u_long) (sof) >> (PGSHIFT)))
#define pdirhash2_0(sid, sof) (((u_long) (sid)) >> GVA_OFFSET_BITS) ^ \
    (((u_long) (sof)) >> PGSHIFT))
```

The output of this is then masked against `(nhtbl[2_0] - 1)` to make it reference an entry in the `htbl[2_0]`.

`GVA_OFFSET_BITS` & `PGSHIFT` are defined in `<machine/param.h>` to be 10 and 12

The space was `0x18b5800` and the offset `0x40010000`.

This system was a 64bit system.

```
q4> 0x18b5800 >> 10 ^ 0x40010000 >> 12
01061306      287430  0x462c6
q4>
```

```
q4> 0x462c6 & (nhtbl2_0-1)
061306 25286  0x62c6
q4>
```

```
q4> load struct hpde2_0 from htbl2_0 skip 0x62c6
loaded 1 struct hpde2_0 as an array (stopped by max count)
q4>
```

This will then load the first possible translation, but since hashing does not produce unique results the space and offset need to be checked

```
q4> print -x pde_space pde_vpage pde_next
pde_space pde_vpage pde_next
```

Module 7 — Memory Management

Lab: Reading Translations from the Page Directory

```
0x9885c00 0x400d1 0xce1200
q4>
```

This space number is wrong, so try the linked list.

```
q4> load struct hpde2_0 from pde_next next pde_next max 100
loaded 3 struct hpde2_0's as a linked list (stopped by null pointer)
```

Unfortunately q4 does not seem to like to load the whole list in one go, so we needed to load the first element and then walk the list

```
q4> print -x pde_space pde_vpage pde_next
pde_space pde_vpage pde_next
0xdc4ec00 0xc13fd 0x8c6680
0x18b5800 0x40010 0xce1d80 --- buffer1
0xdc4ec00 0x813fd 0
q4> keep pde_space == 0x18b5800
kept 1 of 3 struct hpde2_0's, discarded 2
q4> print -x pde_space pde_vpage pde_ar
pde_space pde_vpage pde_ar
0x18b5800 0x40010 0xf
q4>
```

from <machine/pde.h> we can see the access rights definition.

```
#define PDE_AR_UR      0x0f
#define PDE_AR_URW    0x1f
```

so the page has been marked read only.

Module 7 — Memory Management

Lab: Data Access, Parent Reads First

Fork one will respond to signals by either reading or write various data areas.

```
ken@tiger[machine] kill -USR2 2135 # The parent
```

results in

```
entering myHandler signal 17 pid 2135 parent
leaving myHandler signal 17 pid 2135 parent
```

This should have resulted in the parent reading buffer3 and so no changes.

Check list by re-reading the list of vfd's for both parent and child.

```
q4> recall p_pregs
copied 7 items
q4>
q4> keep p_type == PT_DATA
kept 1 of 7 struct pregion's, discarded 6
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4> name it p_reg
so named
q4>
q4> load struct vfd dbd from r_chunk max r_pgsz
loaded 27 struct vfd dbd's as an array (stopped by max count)
q4>

q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnum
 0 0x1 0 0xa67f
 1 0x1 0x1 0xb0e0
 2 0x1 0x1 0xb0e1
 3 0x1 0x1 0xb0e3
 4 0x1 0x1 0xafce
 5 0x1 0 0xa67e
 6 0 0x1 0
 7 0x1 0 0xb10c
 8 0x1 0 0xb10d
 9 0x1 0 0xb10e
10 0x1 0 0xb10f
11 0x1 0x1 0xae9c
12 0x1 0x1 0xae9d
13 0x1 0x1 0xae9e
14 0x1 0x1 0xae9f
15 0x1 0x1 0xa784 --- buffer1
16 0x1 0x1 0xa785 --- buffer1
17 0x1 0x1 0xa786 --- buffer2
18 0x1 0x1 0xa787 --- buffer2
19 0x1 0x1 0xad60 --- buffer3 still marked copy on write with same pgn
20 0x1 0x1 0xad61 --- buffer3
21 0x1 0x1 0xad62 --- buffer4
22 0x1 0x1 0xad63 --- buffer4
23 0x1 0 0xb6c4
```

Module 7 — Memory Management

Lab: Data Access, Parent Reads First

```
24 0x1 0 0xb6c5
25 0x1 0 0xb6c6
26 0x1 0 0xb6c7
q4>
```

now check the child

```
q4> recall c_pregs
copied 7 items
q4> keep p_type == PT_DATA
kept 1 of 7 struct pregion's, discarded 6
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4> name it c_reg
so named
q4>
q4> load struct vfddb from r_chunk max r_pgsz
loaded 27 struct vfddb's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnun
  0          0x1          0x1          0x6391
  1 0x1 0x1 0xb0e0
  2 0x1 0x1 0xb0e1
  3 0x1 0 0xa67c
  4 0x1 0x1 0xafce
  5 0x1 0x1 0xb687
  6 0 0x1 0
  7 0x1 0 0xa678
  8 0x1 0 0xa679
  9 0x1 0 0xa67a
 10 0x1 0 0xa67b
 11 0x1 0x1 0xae9c
 12 0x1 0x1 0xae9d
 13 0x1 0x1 0xae9e
 14 0x1 0x1 0xae9f
 15 0x1 0x1 0xa784 --- buffer1
 16 0x1 0x1 0xa785 --- buffer1
 17 0x1 0x1 0xa786 --- buffer2
 18 0x1 0x1 0xa787 --- buffer2
 19 0x1 0x1 0xad60 --- buffer3 still marked copy on write with same ppn
 20 0x1 0x1 0xad61 --- buffer3
 21 0x1 0x1 0xad62 --- buffer4
 22 0x1 0x1 0xad63 --- buffer4
 23 0x1 0x1 0xacc0
 24 0x1 0x1 0xacc1
 25 0x1 0x1 0xacc2
 26 0x1 0x1 0xacc3
q4>
```

Module 7 — Memory Management

Lab: Child Accesses the Data

A kill -USR2 on the child will cause a read of buffer4.

Re-check the vfds and see if there are any changes.

Did either parent or child change? (Neither, Both, Parent or Child)

This machine supports variable pages sizes. The kernel had selected to use 16K "super pages" so the normal 4K pages are being managed in blocks of 4 in this example. You might have noticed the contiguous physical page numbers.

```
ken@tiger[machine] kill -USR2 2136
```

```
entering myHandler signal 17 pid 2136 child
leaving myHandler signal 17 pid 2136 child
```

```
q4> recall p_reg
copied 1 item
q4> load struct vfdv from r_chunk max r_pgsize
loaded 27 struct vfdv's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfn
0 0x1 0 0xa67f
1 0x1 0x1 0xb0e0
2 0x1 0x1 0xb0e1
3 0x1 0x1 0xb0e3
4 0x1 0x1 0xafce
5 0x1 0 0xa67e
6 0 0x1 0
7 0x1 0 0xb10c
8 0x1 0 0xb10d
9 0x1 0 0xb10e
10 0x1 0 0xb10f
11 0x1 0x1 0xae9c
12 0x1 0x1 0xae9d
13 0x1 0x1 0xae9e
14 0x1 0x1 0xae9f
15 0x1 0x1 0xa784 --- buffer1
16 0x1 0x1 0xa785 --- buffer1
17 0x1 0x1 0xa786 --- buffer2
18 0x1 0x1 0xa787 --- buffer2
19 0x1 0x1 0xad60 --- buffer3
20 0x1 0x1 0xad61 --- buffer3
21 0x1 0x1 0xad62 --- buffer4 still marked copy on write with same pfn
22 0x1 0x1 0xad63 --- buffer4
23 0x1 0 0xb6c4
24 0x1 0 0xb6c5
25 0x1 0 0xb6c6
26 0x1 0 0xb6c7
q4>
```


Module 7 — Memory Management

Lab: Child Accesses the Data

```
q4> recall c_reg
copied 1 item
q4> load struct vfddb from r_chunk max r_pgsc
loaded 27 struct vfddb's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnnum
    0          0x1          0          0x6391
  1 0x1      0      0xafc7
  2 0x1      0      0x9b4b
  3 0x1      0      0xa67c
  4 0x1 0x1      0xafce
  5 0x1      0      0xb687
  6  0 0x1          0
  7 0x1      0      0xa678
  8 0x1      0      0xa679
  9 0x1      0      0xa67a
 10 0x1      0      0xa67b
 11 0x1 0x1      0xae9c
 12 0x1 0x1      0xae9d
 13 0x1 0x1      0xae9e
 14 0x1 0x1      0xae9f
 15 0x1 0x1      0xa784 --- buffer1
 16 0x1 0x1      0xa785 --- buffer1
 17 0x1 0x1      0xa786 --- buffer2
 18 0x1 0x1      0xa787 --- buffer2
 19 0x1      0      0xad74 --- buffer3 pages have been copied, new ppn's
 20 0x1      0      0xad75 --- buffer3
 21 0x1      0      0xad76 --- buffer4
 22 0x1      0      0xad77 --- buffer4
 23 0x1      0      0xacc0
 24 0x1      0      0xacc1
 25 0x1      0      0xacc2
 26 0x1      0      0xacc3
q4>
```

Module 7 — Memory Management

Lab: Data Access, Parent Writes First

Using `kill -USR1` on the parent a write to `buffer1` can be induced.

Re-read the vfd's and see what (if anything happens this time).

```
ken@tigger[machine] kill -USR1 2135

entering myHandler signal 16 pid 2135 parent
leaving myHandler signal 16 pid 2135 parent

q4> recall p_reg
copied 1 item
q4>
q4> load struct vfddb from r_chunk max r_pgsz
loaded 27 struct vfddb's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnnum
 0 0x1 0 0xa67f
 1 0x1 0x1 0xb0e0
 2 0x1 0x1 0xb0e1
 3 0x1 0x1 0xb0e3
 4 0x1 0x1 0xafce
 5 0x1 0 0xa67e
 6 0 0x1 0
 7 0x1 0 0xb10c
 8 0x1 0 0xb10d
 9 0x1 0 0xb10e
10 0x1 0 0xb10f
11 0x1 0x1 0xae9c
12 0x1 0x1 0xae9d
13 0x1 0x1 0xae9e
14 0x1 0x1 0xae9f
15 0x1 0 0x9b00 --- and now the parent has written
16 0x1 0 0x9b01
17 0x1 0 0x9b02
18 0x1 0 0x9b03
19 0x1 0x1 0xad60
20 0x1 0x1 0xad61
21 0x1 0x1 0xad62
22 0x1 0x1 0xad63
23 0x1 0 0xb6c4
24 0x1 0 0xb6c5
25 0x1 0 0xb6c6
26 0x1 0 0xb6c7
q4>
```

This example was run on a C200+ running a 64-bit HP-UX 11.00 (9808) kernel, which has support for variable page sizes. The system has selected 16K pages for TLB/pdir access for much of the data area. This causes ppns to be allocated in contiguous blocks. It also means that accessing one part of these "super pages" means the whole one needs to be copied.

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

Most of the address space on HP-UX uses copy on write only for the parent and copy-on-access for the child. By default, only the text areas of exec_magic and shmem_magic applications use the full copy on write implementation. So let us now look at an example with such a text area.

Run fork1.exec_magic and look at the text preions for both the parent and child

Fill in the following table:

Table 3:

	p_type	p_space	p_vaddr	p_count	p_reg	p_off
Parent						
Child						

```
ken@tiger[mod7] ./fork1.exec_magic
plock failed, but don't worry.: Not owner
      buffer      buffer1      buffer2      buffer3      buffer4
buffer addresses  0x2d560    0x2e000    0x30000    0x32000    0x34000
buffer page #    0xe       0xf       0x11      0x13       0x15
cbuffer is at address 0x1aa78
kill -USR1 2372 ; will cause the parent to write buffer1
kill -USR1 2373 ; will cause the child to write buffer2
kill -USR2 2372 ; will cause the parent to read buffer3
kill -USR2 2373 ; will cause the child to read buffer4
kill -INT 2372 ; will cause the parent to write cbuffer
kill -INT 2373 ; will cause the child to write cbuffer
cbuffer buffer is in the text area so you will die if
try this test on the share_magic example.
```

```
q4>
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
q4> keep p_stat
kept 79 of 276 struct proc's, discarded 197
q4> name it all
so named
q4> keep p_pid == 2372
kept 1 of 79 struct proc's, discarded 78
q4> name it parent
so named
```

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
q4>
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4>
q4> load struct pregion from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 7 struct pregion's as a linked list (stopped by loop)
q4> keep p_type == PT_TEXT
kept 1 of 7 struct pregion's, discarded 6
q4>
q4> name it p_preg
so named
q4>
q4> recall all
copied 79 items
q4> keep p_pid == 2373
kept 1 of 79 struct proc's, discarded 78
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4> load struct pregion from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 7 struct pregion's as a linked list (stopped by loop)
q4> keep p_type == PT_TEXT
kept 1 of 7 struct pregion's, discarded 6
q4> name it c_preg
so named
q4> recall p_preg
copied 1 item
q4> merge c_preg
merged 1 item with 1 item making 2 items
q4>
q4> print -x p_type p_space p_vaddr p_count %d p_reg p_off %d
  p_type  p_space p_vaddr p_count      p_reg p_off
PT_TEXT 0xdcdb000 0x1000      30 0x100e8ae00  13
PT_TEXT 0xe9e5000 0x1000      30 0x100e84a00  13
q4>
```

Now load the vfds for the two regions. Note that in the case of the text area the pregion had an offset, given by p_off. The first p_off vfds should be invalid. We will need to discount them from later arithmetics. (Also remember that the text area starts at page 1 and not at page 0, so another arithmetic tweak will also be needed.)

How many of the page number are the same between parent and child?

```
q4> recall p_preg
copied 1 item
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4> name it p_reg
so named
q4>
```

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
q4> print r_type r_pgsz %d r_off
      r_type r_pgsz r_off
RT_PRIVATE    43      0
q4>
```

```
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnnum
```

#	V	CoW	ppn	
0	0	0x1	0	
1	0	0x1	0	
2	0	0x1	0	
3	0	0x1	0	
4	0	0x1	0	
5	0	0x1	0	
6	0	0x1	0	
7	0	0x1	0	
8	0	0x1	0	
9	0	0x1	0	
10	0	0x1	0	
11	0	0x1	0	
12	0	0x1	0	
13	0x1	0x1	0xb27c	0
14	0x1	0x1	0xb548	1
15	0x1	0x1	0xb549	2
16	0x1	0x1	0xa794	3
17	0x1	0x1	0xa795	4
18	0x1	0x1	0xb126	5
19	0x1	0x1	0xb127	6
20	0x1	0x1	0xb060	7
21	0x1	0x1	0xb061	8
22	0x1	0x1	0xb062	9
23	0x1	0x1	0xb063	10
24	0x1	0x1	0xb0d5	11
25	0x1	0x1	0xb47e	12
26	0x1	0x1	0xb47f	13
27	0x1	0x1	0xa6b2	14
28	0x1	0x1	0xa6b3	15
29	0x1	0x1	0xb0f0	16
30	0x1	0x1	0xb0f1	17
31	0x1	0x1	0xb270	18
32	0x1	0x1	0xb271	19
33	0x1	0x1	0xb272	20
34	0x1	0x1	0xb273	21
35	0x1	0x1	0xb538	22
36	0x1	0x1	0xb539	23
37	0x1	0x1	0xb53a	24
38	0x1	0x1	0xb53b	25 cbuffer is in here, fork1 printed it's address so you can
39	0x1	0x1	0xb110	26 work it out, don't forget the p_off and the fact that the
40	0x1	0x1	0xb111	27 pregion starts at 1 and not zero.
41	0x1	0x1	0xb112	28
42	0x1	0x1	0xb113	29

```
q4>
```

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
q4>
q4> recall c_preg
copied 1 item
q4> load struct region from p_reg
loaded 1 struct region as an array (stopped by max count)
q4> name it c_reg
so named
q4> print r_type r_pgsz %d r_off
      r_type r_pgsz r_off
RT_PRIVATE      43      0
q4> load struct vfd_bdb from r_chunk max r_pgsz
loaded 43 struct vfd_bdb's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnnum
      0          0          0x1          0
 1  0  0x1      0
 2  0  0x1      0
 3  0  0x1      0
 4  0  0x1      0
 5  0  0x1      0
 6  0  0x1      0
 7  0  0x1      0
 8  0  0x1      0
 9  0  0x1      0
10  0  0x1      0
11  0  0x1      0
12  0  0x1      0
13  0x1 0x1    0xb27c
14  0x1 0x1    0xb548
15  0x1 0x1    0xb549
16  0x1 0x1    0xa794
17  0x1 0x1    0xa795
18  0x1 0x1    0xb126
19  0x1 0x1    0xb127
20  0x1 0x1    0xb060
21  0x1 0x1    0xb061
22  0x1 0x1    0xb062
23  0x1 0x1    0xb063
24  0x1 0x1    0xb0d5
25  0x1 0x1    0xb47e
26  0x1 0x1    0xb47f
27  0x1 0x1    0xa6b2
28  0x1 0x1    0xa6b3
29  0x1 0x1    0xb0f0
30  0x1 0x1    0xb0f1
31  0x1 0x1    0xb270
32  0x1 0x1    0xb271
33  0x1 0x1    0xb272
34  0x1 0x1    0xb273
35  0x1 0x1    0xb538
36  0x1 0x1    0xb539
37  0x1 0x1    0xb53a
38  0x1 0x1    0xb53b
39  0x1 0x1    0xb110
40  0x1 0x1    0xb111
```

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
41 0x1 0x1 0xb112
42 0x1 0x1 0xb113
q4>
```

Advanced students: skip to writing to the text area if short of time.

Now read the physical to virtual mapping for the pages 38, 39 and 40 of the vfd list to convert back to the virtual addresses.

Are these pages aliased?

What are the offset page numbers?

Is the space number that of the parent or the child?

```
q4> load struct pfn_to_virt_ptr from pfn_to_virt_ptr skip 0xb
loaded 1 struct pfn_to_virt_ptr as an array (stopped by max count)
q4>
q4> load pfn_to_virt_entry_t from pp_tbl skip 0x53b
loaded 1 pfn_to_virt_entry_t as an array (stopped by max count)

q4> recall 23      ### this just gets the pfn_to_virt_ptr back.
copied 1 item
q4> load pfn_to_virt_entry_t from pp_tbl skip 0x110 max 2
loaded 2 pfn_to_virt_entry_t's as an array (stopped by max count)
q4> merge 24      ### just glues all three data structures together.
merged 1 item with 2 items making 3 items

q4>
q4> print -x space alias_or_offset.offset_page
space alias_or_offset.offset_page
0xe9e5000 0x1a ----- parent space, page offset 26, page 25 in pregon
0xe9e5000 0x1b ----- parent space, page offset 27, page 26 in pregon
0xe9e5000 0x1c ----- parent space, page offset 28, page 27 in pregon
q4>
```

The pages don't yet have their alias structures set up as the child has not accessed them. The alias structures would be set up on a page fault from the child.

Reading the page directory entry for cbuffer (fork1 gave us the address, 0x1aa78 in this example).

```
q4>
q4> 0xe9e5000 >> 10 ^ 0x1aa78 >> 12
0723616 239502 0x3a78e
q4> 0x3a78e & (nhtbl2_0-1)
0123616 42894 0xa78e
q4>
q4> load -r struct hpde2_0 from htbl2_0 skip 0xa78e
loaded 1 struct hpde2_0 as an array (stopped by max count)
q4>

q4> print -x pde_vpage pde_space pde_ar pde_phys
```

Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
pde_vpage pde_space pde_ar pde_phys
0x1a 0xe9e5000 0x2f 0xb53b
q4>
```

```
#define PDE_AR_UR      0x0f
#define PDE_AR_URW    0x1f
#define PDE_AR_URX    0x2f
#define PDE_AR_URWX   0x3f
#define PDE_AR_UX     0x7f
```

So the page is marked Read and Execute only. Normally exec_magic pages are read/write/execute

A kill -INT on the data access parent will cause it to write to the cbuffer. Then re-check the the vfd's for both parent and child.

Which one changed?

```
ken@tigger[machine]
ken@tigger[machine] kill -INT 2372
ken@tigger[machine]
```

```
entering myHandler signal 2 pid 2372 parent
leaving myHandler signal 2 pid 2372 parent
```

Parent has now written to cbuffer at address 0x1aa78

```
q4> recall p_reg
copied 1 item
q4>
q4> load struct vfddb from r_chunk max r_pgsz
loaded 43 struct vfddb's as an array (stopped by max count)
q4>
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfn
0 0 0x1 0
1 0 0x1 0
2 0 0x1 0
3 0 0x1 0
4 0 0x1 0
5 0 0x1 0
6 0 0x1 0
7 0 0x1 0
8 0 0x1 0
9 0 0x1 0
10 0 0x1 0
11 0 0x1 0
12 0 0x1 0
13 0x1 0x1 0xb27c
14 0x1 0x1 0xb548
```


Module 7 — Memory Management

Lab: True Copy-On-Write, with exec_magic Text Areas

```
15 0x1 0x1 0xb549
16 0x1 0x1 0xa794
17 0x1 0x1 0xa795
18 0x1 0x1 0xb126
19 0x1 0x1 0xb127
20 0x1 0x1 0xb060
21 0x1 0x1 0xb061
22 0x1 0x1 0xb062
23 0x1 0x1 0xb063
24 0x1 0x1 0xb0d5
25 0x1 0x1 0xb47e
26 0x1 0x1 0xb47f
27 0x1 0x1 0xa6b2
28 0x1 0x1 0xa6b3
29 0x1 0x1 0xb0f0
30 0x1 0x1 0xb0f1
31 0x1 0x1 0xb270
32 0x1 0x1 0xb271
33 0x1 0x1 0xb272
34 0x1 0x1 0xb273
35 0x1 0x1 0xb538
36 0x1 0x1 0xb539
37 0x1 0x1 0xb53a
38 0x1 0 0xa769 << new page
39 0x1 0x1 0xb110
40 0x1 0x1 0xb111
41 0x1 0x1 0xb112
42 0x1 0x1 0xb113
q4>
```

The parent got a new page, and cleared the copy of write flag.

(Advanced)

Now check the page directory entry again, what are the new access rights?

```
q4> 0xe9e5000 >> 10 ^ 0x1aa78 >> 12
0723616 239502 0x3a78e
q4> 0x3a78e & (nhtbl2_0-1)
0123616 42894 0xa78e
q4> load -r struct hpde2_0 from htbl2_0 skip 0xa78e
loaded 1 struct hpde2_0 as an array (stopped by max count)
q4> print -x pde_vpage pde_space pde_ar pde_phys
pde_vpage pde_space pde_ar pde_phys
0x1a 0xe9e5000 0x3f 0xa769
q4>
```

New page is now read write.

Module 7 — Memory Management

Lab: Data Access, Child Writes

New repeat the test for the child, did the child keep the original page?

```
ken@tiger[machine]
ken@tiger[machine] kill -INT 2373
ken@tiger[machine]
```

```
entering myHandler signal 2 pid 2373 child
leaving myHandler signal 2 pid 2373 child
```

```
q4> recall c_reg
copied 1 item
q4> load struct vfd dbd from r_chunk max r_pgsz
loaded 43 struct vfd dbd's as an array (stopped by max count)
q4> print -x indexof %3d c_vfd.pgm.pg_v %#4x c_vfd.pgm.pg_cw %#4x c_vfd.pgm.p>
indexof c_vfd.pgm.pg_v c_vfd.pgm.pg_cw c_vfd.pgm.pg_pfnun
      0      0      0x1      0
 1  0  0x1      0
 2  0  0x1      0
 3  0  0x1      0
 4  0  0x1      0
 5  0  0x1      0
 6  0  0x1      0
 7  0  0x1      0
 8  0  0x1      0
 9  0  0x1      0
10  0  0x1      0
11  0  0x1      0
12  0  0x1      0
13 0x1 0x1 0xb27c
14 0x1 0x1 0xb548
15 0x1 0x1 0xb549
16 0x1 0x1 0xa794
17 0x1 0x1 0xa795
18 0x1 0x1 0xb126
19 0x1 0x1 0xb127
20 0x1 0x1 0xb060
21 0x1 0x1 0xb061
22 0x1 0x1 0xb062
23 0x1 0x1 0xb063
24 0x1 0x1 0xb0d5
25 0x1 0x1 0xb47e
26 0x1 0x1 0xb47f
27 0x1 0x1 0xa6b2
28 0x1 0x1 0xa6b3
29 0x1 0x1 0xb0f0
30 0x1 0x1 0xb0f1
31 0x1 0x1 0xb270
32 0x1 0x1 0xb271
33 0x1 0x1 0xb272
```

Module 7 — Memory Management

Lab: Data Access, Child Writes

```
34 0x1 0x1 0xb273
35 0x1 0x1 0xb538
36 0x1 0x1 0xb539
37 0x1 0x1 0xb53a
38 0x1 0 0x9f24 << child also now has new copy, leaves old clean one
39 0x1 0x1 0xb110 on the freelist / page cache.
40 0x1 0x1 0xb111
41 0x1 0x1 0xb112
42 0x1 0x1 0xb113
q4>
```

If you want more labs, try changing `fork1` so that the child will read `cbuffer`, and then repeat the test reading the physical to virtual mappings and the aliases linked list.

Lab: Memory Structures

1. In this exercise, you will continue to look at structures from the “more” process that you started in the lab exercise on process structures. You may have to “nudge” the process back into memory if you were on a busy system, and the physical addresses from the proc structure lab won’t match anymore.
2. Using the correct “p_ll.lle_next[0]” value from the process structures lab exercise, step 12, load the pregon of the U-area:

```
q4> load struct pregon from 0XXXXXXXXX
```

Instead of simply using the “p_space” and “p_vaddr” values to go directly to the U-AREA (as you did in step 15), you’ll follow the memory structures to the same place.

3. Load the associated region structure:

```
q4> load struct region from p_reg
```

4. Display the contents of the region structure and note the following:

```
q4> print -tx | more
```

```
r_type: _____ (region type)
r_pgsz: _____ (size of region in pages)
r_refcnt: _____ (How many pregon are pointing to this region)
r_incore: _____ (memory resident)
r_fstore: _____ (frontstore location)
r_bstore: _____ (backstore location)
r_root: _____ (pointer to vfddb root structure)
r_key: _____ (top-level search key for chunk)
r_chunk: _____ (pointer to only vfddb chunk)
```

(Note: See the header file “/usr/include/sys/region.h” for more information.)

5. To get to the associated pages in memory for a region, it is necessary to follow either the r_root pointer or the r_chunk pointer - depending on whether the region has more than 32 pages in it. The U-AREA region has only three or four pages so you’ll use the r_chunk pointer. Also the r_key field will contain a 0x0 if the r_chunk pointer is used.
6. Load in four vfd/dbd structures from the r_chunk location:

```
q4> load struct vfd/dbd from r_chunk max 4
```

Module 7 — Memory Management

Lab: Memory Structures

7. Display the contents of these four structures and note the following values in each structure:

```
q4> print -tx c_vfd.pgi.pg_vfd
```

index	c_vfd.pgi.pg_vfd
_____	_____
_____	_____
_____	_____
_____	_____

8. Take the “c_vfd.pgi.pg_vfd” field of index 0, drop the leading digit, and add three zeroes to the end to “multiply by 4096” (size of a page).
9. Compare the number you got in step 8 to the value “physaddrof” you noted in the Process Structures Lab, step 16. They should be the same. Thus you should see that the U-AREA region can be reached through the memory structures. By extension, any page (currently resident in memory) can be located through the memory structures.
10. EXTRA CREDIT: Things get more complex when the size of the region exceeds 32 pages. Go to the region structure of a “large” region* and follow the “r_root” pointer through the “broot” structure to the “bnode” structures and find the first memory-resident page. Good hunting!

*Note: I suggest you explore the region for “libc”. You can use GLANCE to find which region that is on a given process.

11. Now let’s see if we can locate the page directory entry for this page in the PDIR. Using the pfn number obtained in step 8, let’s load and search the PDIR for the correct pde.

```
q4> load struct hpde from base_pdir max max_pdir until max_pdir
```

Note: This is going to take several seconds (20-30), so be patient.

```
q4> keep pde_phys == <c_vfd.pgi.pg_vfd>  
      (FOR INDEX 0 FROM STEP 7  
      WITH THE FIRST DIGIT REMOVED)
```

Lab: Memory Structures

12. Print out the following fields and compare them to the results of step 16 of the Proc Structures lab.

```
q4> print -x pde_space pde_vpage
```

pde_space _____ (should match spaceof)

pde_vpage _____ (should match first 15 bits of addrof shifted by one bit)

13. Print out the entire pde and answer the questions.

```
q4> print -tx | more
```

pde_valid _____

pde_ref _____

pde_ar _____

pde_protid _____

pde_phys _____

Is there another pde linked to this one? _____

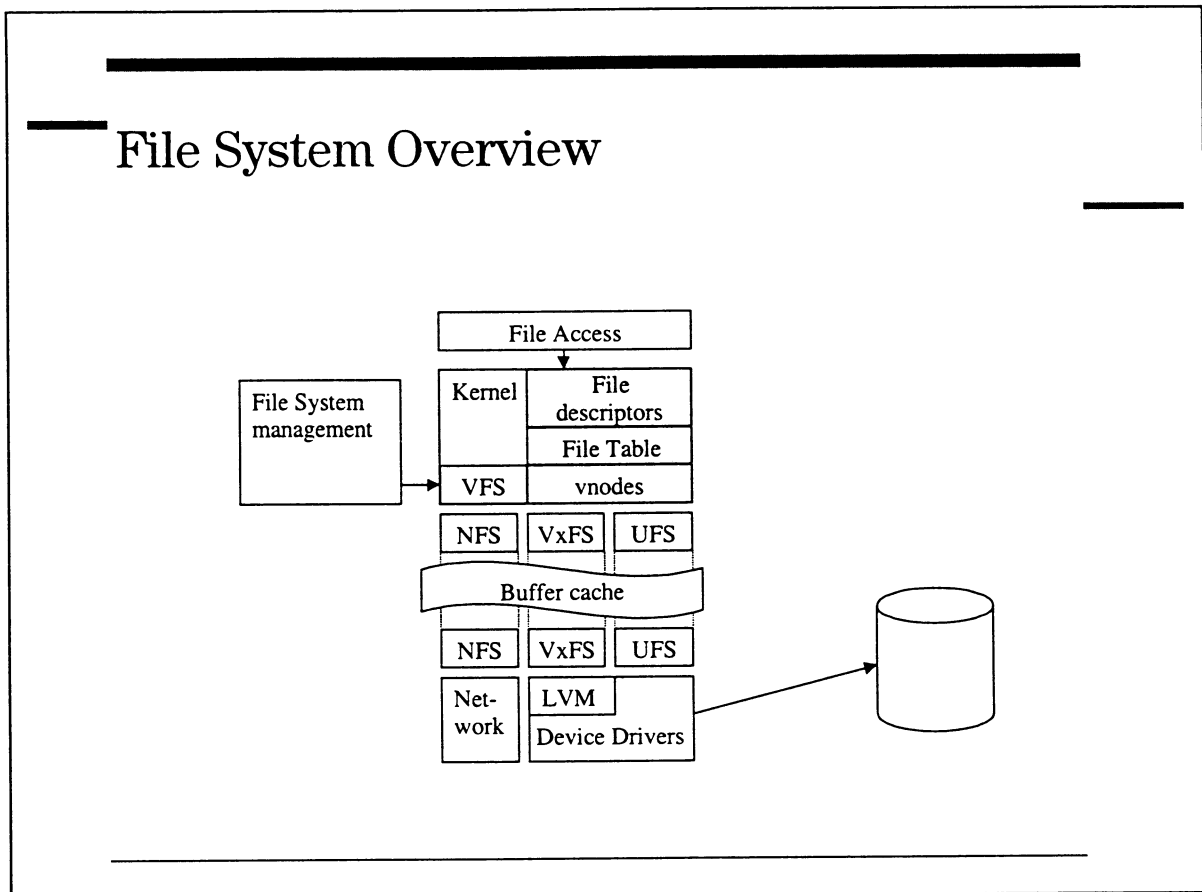
Module 8 - File Systems

Objectives

Upon completion of this module, you will be able to do the following:

- Understand the disk-based data structures for both the HFS and JFS file systems.
- Be able to display these data structures using tools such as **fsdb**.
- Understand the various design tradeoffs of the different file systems
- Understand the way in which the HP-UX kernel is able to support different types of file system designs.
- Understand the kernel data structures for file system management.
- Understand the kernel data structures for accessing individual files.
- Be able to navigate these data structures using the **q4** kernel debugger.
- Have an appreciation of the buffer cache.

8-1. SLIDE: File System Overview



Student Notes

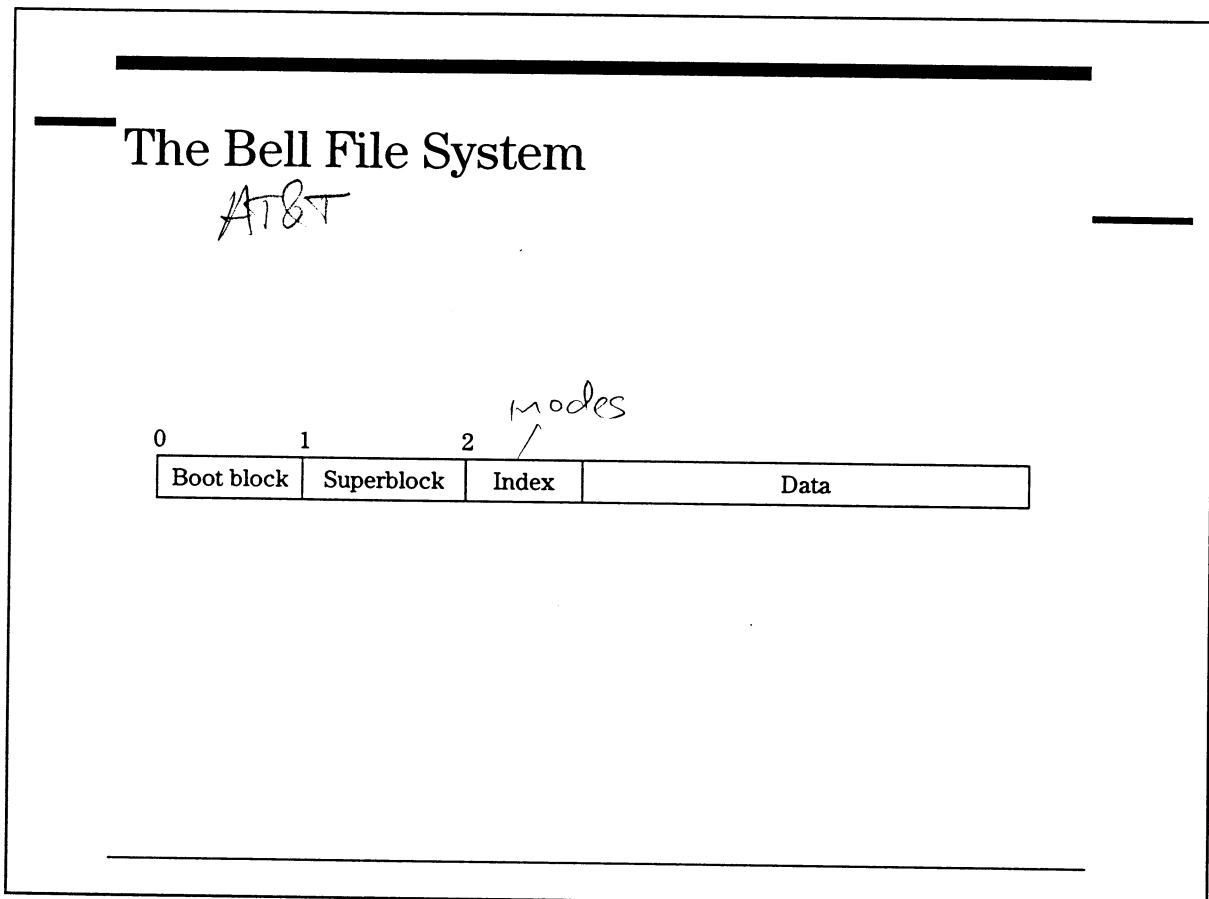
In terms of the internals of the file system, we have a number of different areas to look at. The data is stored on disk, so we need to take a look at the formats used there. Since disks are so slow, the data is typically cached through the buffer cache. The kernel needs to keep track of the individual files that are being used and also the file systems that are currently mounted.

The HP-UX kernel allows the use of several different types of file system without higher levels of code having to worry about the differences. The kernel needs to implement some form of switching mechanism that allows the correct file system specific functions to be called, based upon which file is being accessed.

For file access, this switching is performed using a data structure called a virtual node or **vnode**. For the file systems a virtual file system structure or **vfs** is used. Both of these data structures contain pointers to sets of functions that work on this type of file system.

Higher level code such as a close systems call can then access the **vnode** structure for the file and it will say which close function needs to be called to close this type of file.

8-2. SLIDE: The Bell File System



Student Notes

Although the Bell file system has not been used on HP-UX systems for many years (since before System V), it serves as a good starting point for a discussion of file systems.

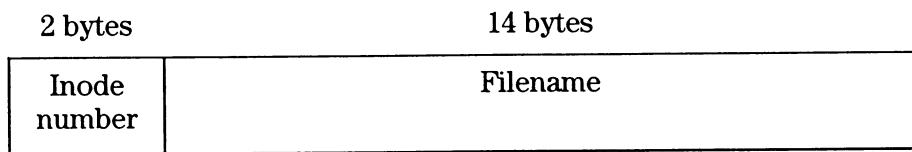
At the start of the disk in block zero is an area that technically does not belong to the file system — it can be used for other purposes. Typically it was used as a boot block, where the disk was configured as the systems boot device.

After this comes the *superblock*, which contains the fundamental description of the file system as a whole. In the Bell superblock, the free maps were also held.

After the superblock was the index for the disk. Since we are more often interested in the entries or nodes in the index than the whole table, the name for this has become turned around and is now known as the inode table. Since the name inode is used both on disk and also in the kernel, the notes will refer to them as either disk or in-core inodes where possible. Each file on the disk then has an inode and this describes everything about it, except its name.

Module 8
File Systems

These then were the data structures that defined the disk layout. The other data structure on disk that is relevant is the directory. Bell directories used a fixed length record structure where each entry was 16 bytes long, and consisted of the inode number and the filename.



The filename was fairly unusual in terms of C programs, as it holds a character string but does not need to be null terminated.

8-3. SLIDE: Problems with the Bell File System

Problems with the Bell File System

- There is only one copy of the superblock.
- Inodes are held in a table.
- The block size controls both the size of disk transfers and the smallest sized area that can be allocated.
- All the control data is at the start, the data is at the end.
- The file system was designed for simplicity and not performance.

Student Notes

The Bell file system, while it has the advantage of being extremely simple, suffers from many shortcomings. These include:

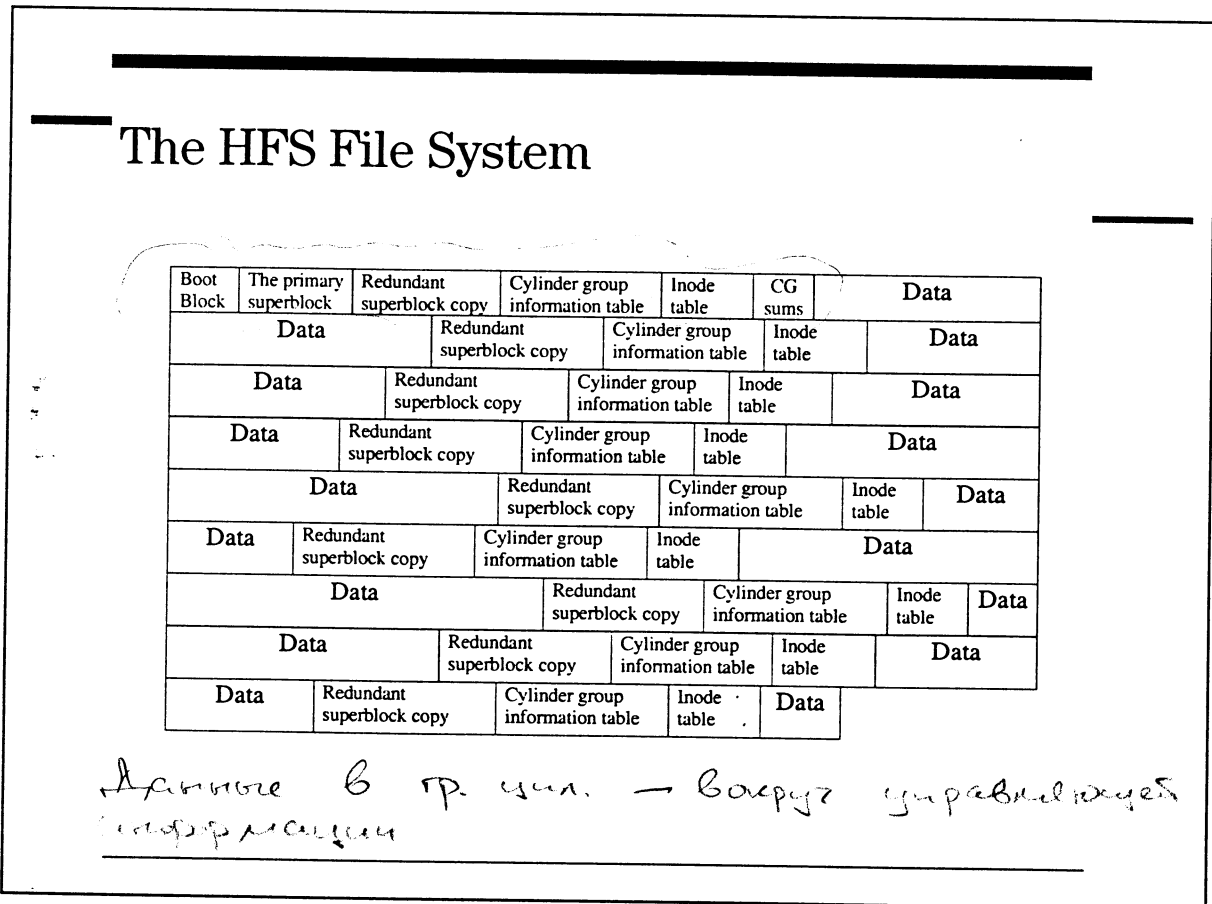
- There is only a single copy of the superblock. This is stored very close to the start of the disk, and is therefore likely to be overwritten if anyone accidentally writes to the disk. Also since it contains dynamic summary data that is frequently being written to, a system failure during a superblock write could leave it corrupt.
- The inodes are held in a table. While tables have very good performance characteristics, once they have been created they are fixed in size. This leaves open the possibility of running out of inodes before running out of disk space. Fear of doing this tends to lead to over creating inodes, and so wasting disk space.
- The block size was used both to control the size of disk transfers and also the minimum area of disk space that ever gets allocated. This is a problem as these two functions have different and conflicting interests.

For the case of the minimum storage unit, if this value is large then space will be wasted when storing small files.

For the case of transfer sizes, then a larger size can lead to needing to make fewer transfers. Each transfer will typically involve a movement of the disk head and then waiting for the data to spin around to be under the disk head as well as the time taken for the transfer itself. Making fewer transfers, while it won't actually reduce the time spent transferring data, will reduce the time spent in the other phases. Of course this argument only holds true if we actually need the data, but frequently we will.

- The control data, the superblock with its free list, and the inodes are at the beginning of the disk. The data is then stored after that. This can lead to a large physical separation between user data and the control data that goes with it. When extending a file it is necessary not only to write the new data, but also to update the inode to know where the new data is and to update the free list to show that additional blocks are now in use. Where the data and controls are widely separated, large head movements are needed to move between them. On older disk drives this tends to be slow and also to cause mechanical wear and tear to the drive head assembly.
- The Bell file system was beautifully simple but its design was not really aimed at getting the best out of the disk drives. Disk drives are mechanical systems. Their components move and take time to do so. Placing data in the easiest location for the programmer does not necessarily make life easy on the disk drive. Having the file system understand the physical layout and mechanics of the disk drive can yield huge performance increases in real systems, at the cost of increased program complexity.
- Additionally the Bell file system tended to hardcode many aspects of its design, and was therefore not very tunable. Originally the block size was fixed at 512 bytes (many file system commands such as `ls`, `du`, and `df` still use this value as the block size). Later versions allowed the block size to be changed, but there remained little else that could be tuned.

8-4. SLIDE: The HFS File System



Student Notes

As part of the development of the 4.2BSD version of UNIX a redesign of the file system was undertaken by one of the developers, Kirk McKusick. Minor changes were made to this design for 4.3BSD and it is this design that has until recently been the main file system used in HP-UX.

HP-UX refers to this file system as HFS, but it has many names:

- The High Performance File System (HFS in HP-UX).
- The Berkeley file system.
- The Fast File System (FFS in SYS V.4).
- The McKusick file system.
- The UNIX File System (UFS).

Berkeley simply refers to the file system as the UFS, the UNIX file system, and since the code to handle it comes from them, the code refers to it as UFS. This is the name that will be used in the rest of this module.

UFS sets out to tackle many of the issues that we earlier identified with the original Bell file system.

The overall file system is subdivided into smaller pieces known as cylinder groups.

The control structures are then spread across these cylinder groups. This has the effect of localizing the control structures with the data and thus reducing the size of most of the head movements, thus increasing the performance.

- Superblocks */etc/sbtab*

The cylinder groups provide a convenient place to store the redundant copies of the superblock. Since loss of the superblock data can make the recovery of data from a file system extremely difficult, it is useful to have spare copies of it on the disk for use by **fsck** or **fsdb** in an emergency.

When the file system is created, **mkfs** attempts to spread these superblocks around the disk, aiming to get copies onto different disk surfaces and into different radial positions. This improves the chance of being able to successfully locate an intact superblock copy after a mechanical head crash. These days it is rare to attempt recovery under such circumstances, but there are companies who specialize in this sort of operation.

The redundant superblock copies are not accessed or updated during normal operation. Although they contain a full copy of the superblock, the dynamic fields are not kept up to date. These blocks are only updated when the size of the file system is changed with the **extendfs** command and when the **-A** option is used with **tunefs**.

- Blocks and Fragments

Another change that McKusick made was to break up the relationship whereby the block size governs both the minimum unit of disk space that can be allocated and the size used for data transfers.

For the UFS, a larger block size is used and this can be divided into smaller pieces to allow small files to be stored more efficiently. Unfortunately these smaller pieces were given the name *fragments*.

If the block size of a Bell file system were to be raised to 8K (the default size on HP-UX) then it could lead to the wasting of large amounts of disk space when handling small files.

For example take 3 files

Name	Size
f1	200 bytes
f2	1K
f3	3K
total space used	24K

Since UNIX tends to have large numbers of small files this sort of arrangement is going to be highly wasteful of disk space.

On the other hand a larger block size allows more data to be transferred between the disk and the system for a given number of disk seeks.

The UFS solution allows the blocks to be broken into 1,2,4 or 8 fragments of 1,2,4 or 8K. Small files are then able to utilize these fragments with the following restrictions

- The fragments are only used at the end of a file.
- All the fragments are contiguous.
- All the fragments come from the same block.
- Only a small file, one that only uses direct pointers (see the section on UFS disk inodes) can use fragments.

В inode хранятся
не только
размер файла

В кратких ссылках генерируется
перечень фрагментов (HP-UX)

Since increasing the block size is now no longer wasteful of disk space, the block size can be increased to allow for more efficient transfers of larger amounts of data. Of course if the size of the pieces of data to be transferred is small then this too could be wasteful. The block size needs to be chosen to match the application. Generally speaking, where large or sequential transfers are to be made, the larger the block size the higher the transfer speed will be.

- Localizing the control structures and the data

By holding the freemap for the cylinder group within the groups itself, and also a portion of the inode table, file system updates can often happen without the need for large head movements between the data and its controlling structures.

The reduction in the number of these large and frequent head movements is good for performance. Unfortunately not all of these movements can be eliminated, as we shall see, but UFS does a good job in reducing them in most circumstances.

- Understanding the mechanical nature of disk drives.

Another area where dramatic improvements in performance were possible was in the layout of data. The Bell file system tended to layout its data in the simplest of fashions. The UFS on the other hand likes to calculate which blocks would be the fastest to use in any given situation. To this end lots of details about the disk need to be programmed into the file system at creation time.

New file systems are created by the command **mkfs**, which takes as its arguments many characteristics of the disk device.

```
/usr/sbin/mkfs size nsect ntrack blksize fragsize \  
ncpg minfree rps nbpi
```

size The size of the file system was needed to correctly set-up the superblock and the rest of the file system.

Nsect The number of sectors per track. Actually these are the number of DEV_BSIZE units. All IO to block style devices such as a disk is performed in DEV_BSIZE units.

Module 8
File Systems

- Ntrack** The number of tracks per cylinder, this is the number of data surfaces in the disk drive.
- Blksize** The block size chosen for this file system.
- Fragsize** The size of the fragments to split blocks into.
- Ncpg** The number of cylinders per group, typically 16 but there are several limiting factors that can affect this.
- Minfree** A tuning parameter used to limit the fragmentation of the file system.
- Rps** In order to be able to choose the optimum location for data the system needs to know how fast the disk spins. This value is used in conjunction with another tuning parameter **rot_delay**, which cannot be set explicitly from the **mkfs** command, and needs to be set afterwards using **tunefs**.
- Nbpi** The size of the inode table is set as a ratio of bytes to inodes. For HP-UX 10.x and 11.00 this defaults to a value of 6144. Again there are other limiting factors that will affect the actual number of inodes created.

Since **mkfs** requires so many arguments, most of which are dependent on the model of disk drive in use, a friendly front end, **newfs**, was provided. This defaulted some parameters and originally read the others from the file **/etc/disktab**.

Many of these parameters however do not work with modern disk drives or LVM.

- Size** The size of a logical volume is not related to the type of disk drives it resides on. It may in fact be spread over many models of disk drives.
- Nsect** The number of sectors per track is no longer constant. On earlier disk drives, each track has the same number of sectors. This makes it easy for the file system to calculate, for instance, the sector number above or below a particular sector.

Constant-geometry disk drives however are limited in capacity. The major limiting factor on the capacity of a disk drive is density that data can be stored. Since the inner-most track is shorter than the outer-most one, then its sectors will be shorter and the density will be higher.

Modern disk drives try to keep the sector length constant allowing them to use the highest usable density right across the disk. This gives a much higher capacity.

It also gives an interesting performance characteristic since the transfer speed is affected by the number of sectors per track and the outer-most track will have more sectors and therefore a higher transfer speed.

- Rps** The rotational speed of the disk drive was multiplied by another value, the rotation delay, when working out the optimum placement for adjacent blocks of a file.

The rotational delay was the expected time between IO requests. At HP-UX 10.x an additional optimisation was added to the device drivers so that adjacent IO requests are merged together. Since the requests now go out as one single operation there is no gap to have a time delay in.

As the rotation speed is multiplied with a value that is now zero, its own value is now unimportant.

These changes have meant that there is little benefit in using a **disktab** file so it is no longer used by **newfs** by default.

These changes to the layout of the file system give UFS a dramatic performance advantage over the Bell file system. When McKusick initially tested the new design against the original layout he was able to show a ten-fold improvement in performance. As disk sizes increase this advantage increases.

8-5. SLIDE: HFS Disk-resident Data Structures

HFS Disk-resident Data Structures

- superblock
- cylinder groups
- inodes
- directories

Student Notes

The HFS file system uses four data structures to describe the layout of the disk.

superblock	The superblock gives an overall description of the disk and includes the physical characteristics of the device. Also included is current summary information such as the amount of free space.
cylinder groups	Each cylinder group contains an area of control structures in addition to user data. This includes the redundant superblock copy, the freemaps for both fragments and inodes, and local summary information.
inodes	The inode table for UFS is split up so that each cylinder group contains a part of the overall table. This way the inode for a file can be stored close to its data.

directories

The view of the UNIX file system that users see, with its hierarchical structure, is formed by storing filenames in special files called directories.

The directory files hold the filenames and the inode numbers.

8-6. SLIDE: The Superblock

The Superblock

Static data

- files system type
- file system size
- cylinder group size
- block size
- fragment size
- tracks per cylinder
- sectors per track
- inodes per cylinder groups
- rotational delay } устан.
- rotation speed } устан.

Dynamic data

- the clean flag
- last mount point ! (last mounted on.)
- free inode count
- free block count
- free fragment count
- directory count

это много из пакета сформировано
там же LVM - не знаю
это много из пакета сформировано

Student Notes

The superblock contains the overall description of the file system and its current status. Within the superblock there is a mixture of static data and dynamic data fields. In addition to the status information within the superblock itself, there is summary information for each cylinder group held in a separate structure. Since the number of cylinder groups is not fixed, this information could not be held within the superblock, as it is a fixed sized structure.

Most of the static fields are initialized when the file system is created and do not change. With the advent of LVM it became possible to change the size of the device that the file system was built upon, and so the **extendfs** command was introduced. This modifies those parameters that are concerned with the size of the file system. A few of the other static values can be changed using the **tunefs** command.

The dynamic fields within the superblock describe the current status of the file system and provide a few useful bits of information to aid troubleshooting.

The internal structure of the UFS superblock has these two types of data mixed.

The superblock can be displayed using the commands **tuneefs -v** or **dumpfs** on HP-UX.

```
root@tiger[new] tuneefs -v /dev/vg00/lvol1
super block last mounted on: /stand
magic 95014 clean FS_OK time Tue Jul 28 08:44:49 1998
sblkno 16 cblkno 24 iblkno 32 dblkno 80
sbsize 2048 cgsiz 2048 cgoffset 24 cgmask 0xffffffff8
ncg 29 size 69632 blocks 67733
bsize 8192 bshift 13 bmask 0xfffffe000
fsiz 1024 fshift 10 fmask 0xfffffc00
frag 8 fragshift 3 fsbtodb 0
minfree 10% maxbpg 77
maxcontig 1 rotdelay 0ms rps 60
csaddr 80 cssiz 27648 csshift 9 csmask 0xfffffe00
ntrak 7 nsect 22 spc 154 ncyl 453
cpg 16 bpg 308 fp 2464 ipg 384
nindir 2048 inopb 64 nspf 1
nbfree 5563 ndir 12 nifree 11103 nffree 86
cgrtor 7 fmod 0 ronly 0
fname
featurebits 0x1 id 0x0,0x0
optimize FS_OPTTIME

cylinders in last group 5
blocks in last group 96

root@tiger[new]
```

Module 8
File Systems

The actual structure is defined in the header file /usr/include/sys/fs.h

```
struct fs
{
    int32_t fs_unused[2];           /* unused */
    daddr_t fs_sblkno;             /* addr of super-block in filesystem */
    daddr_t fs_cblkno;             /* offset of cyl-block in filesystem */
    daddr_t fs_iblkno;             /* offset of inode-blocks in filesystem */
    daddr_t fs_dblkno;             /* offset of first data after cg */
    int32_t fs_cgoffset;           /* cylinder group offset in cylinder */
    int32_t fs_cgmask;             /* used to calc mod fs_ntrak */
    int32_t fs_time;               /* last time written */
    int32_t fs_size;               /* number of blocks in fs */
    int32_t fs_dsize;              /* number of data blocks in fs */
    int32_t fs_ncg;                /* number of cylinder groups */
    int32_t fs_bsize;              /* size of basic blocks in fs */
    int32_t fs_fsize;              /* size of frag blocks in fs */
    int32_t fs_frag;               /* number of frags in a block in fs */
    /* these are configuration parameters */
    int32_t fs_minfree;            /* minimum percentage of free blocks */
    int32_t fs_rotdelay;           /* num of ms for optimal next block */
    int32_t fs_rps;                /* disk revolutions per second */
    /*
    */
    /* these fields can be computed from the others */
    int32_t fs_bmask;              /* ``blkoff'' calc of blk offsets */
    int32_t fs_fmask;              /* ``fragoff'' calc of frag offsets */
    int32_t fs_bshift;             /* ``lblkno'' calc of logical blkno */
    int32_t fs_fshift;             /* ``numfrags'' calc number of frags */
    /* these are configuration parameters */
    int32_t fs_maxcontig;          /* max number of contiguous blks */
    int32_t fs_maxbpg;             /* max number of blks per cyl group */
    /* these fields can be computed from the others */
    int32_t fs_fragshift;          /* block to frag shift */
    int32_t fs_fsbtodb;            /* fsbtodb and dbtofsb shift constant */
    int32_t fs_sbsize;             /* actual size of super block */
    int32_t fs_csmask;             /* csum block offset */
    int32_t fs_csshift;            /* csum block number */
    int32_t fs_nindir;             /* value of NINDIR */
    int32_t fs_inopb;              /* value of INOPB */
    int32_t fs_nspf;               /* value of NSPF */
    int32_t fs_id[2];              /* file system id */
    struct mirinfo fs_mirror;       /* mirror states of root/swap */
    int32_t fs_featurebits;         /* feature bit flags */
    int32_t fs_optim;              /* optimization preference - see below */
    /*
    */
    /* sizes determined by number of cylinder groups and their sizes */
    daddr_t fs_csaddr;             /* blk addr of cyl grp summary area */
    int32_t fs_cssize;             /* size of cyl grp summary area */
    int32_t fs_cgsize;             /* cylinder group size */
    /* these fields should be derived from the hardware */
    int32_t fs_ntrak;              /* tracks per cylinder */
    int32_t fs_nsect;              /* sectors per track */
    int32_t fs_spc;                /* sectors per cylinder */
    /* this comes from the disk driver partitioning */
    int32_t fs_ncyl;               /* cylinders in file system */
    /* these fields can be computed from the others */
    int32_t fs_cpg;                /* cylinders per group */
    int32_t fs_ipg;                /* inodes per group */
    int32_t fs_fpg;                /* blocks per group * fs_frag */

```

```

/* this data must be re-computed after crashes */
    struct csum fs_cstotal;          /* cylinder summary information */
/* these fields are cleared at mount time */
    char        fs_fmod;            /* super block modified flag */
    char        fs_clean;          /* file system is clean flag */
    char        fs_ronly;          /* mounted read-only flag */
    char        fs_flags;          /* currently unused flag */
    char        fs_fsmnt[MAXMNTLEN]; /* name mounted on */
/* these fields retain the current block allocation info */
    int32_t     fs_cgrotor;        /* last cg searched */
#ifdef __LP64__
    struct csum *fs_csp;           /* incore fs_cs info buffer */
#else /* not __LP64__ */
    struct csum *fs_csp;           /* incore fs_cs info buffer */
    int32_t     fs_csp_pad;
#endif /* not __LP64__ */
    int32_t     fs_unused2[MAXCSBUFS-2]; /* unused */
    int32_t     fs_cpc;            /* cyl per cycle in postbl */
    short       fs_postbl[MAXCPG][NRPOS]; /* head of blocks for each rotation */
    int32_t     fs_magic;         /* magic number */
    char        fs_fname[6];      /* file system name */
    char        fs_fpack[6];     /* file system pack name */
    u_char      fs_rotbl[1];     /* list of blocks for each rotation */
/* actually longer */
};

```

The superblock structure is stored on disk, but it is also copied into the kernel when the file system is mounted. Some of the fields that are defined in the **fs** structure are only relevant to the copy in the kernel.

The following fields are of particular interest.

fs_sblkno fs_cblkno fs_iblkno fs_dblkno fs_cgoffset fs_cgmask	<p>These fields are used to work out where about the parts of the cylinder group information is stored.</p> <p>The /usr/include/sys/fs.h header file contains a series of <u>macros</u> that are used to calculate the positions of these structures in any given cylinder group.</p>
fs_time	The time the superblock was last updated
fs_size	<p>The size of the file system measured in fragments.</p> <p>The comments in the UFS related header files frequently mix up the terms block and fragment. This can make reading these files more difficult, but I'm afraid that is just the way they are.</p>
fs_dsize	<p>The number of fragments for storing data. The blocks used by things like the inode table, superblock and cylinder groups have been subtracted for the overall size.</p> <p>This is the size value that we see with bdf.</p>
fs_ncg	The number of cylinder groups within the file system.
fs_bsize fs_bmask fs_bshift	The block size and the shift and mask values to speed calculation on the block size.

Module 8
File Systems

fs_fsize fs_fmask fs_fshift	The fragment size and the shift and mask values to speed calculation on the fragment size.						
fs_frag fs_fragshift	The number of fragments per block						
fs_minfree	The minfree tuning parameter						
fs_rotdelay	The rotational delay tuning parameter. At HP-UX 10 and onwards this is usually 0 as adjacent IO are merged so that there is no delay between them.						
fs_rps	When a delay is expected between the transfer of successive blocks of a file this value is used to calculate how many blocks will be skipped during the delay. Since we now no longer expect a delay this value is of little relevance. Note this measures the revolutions per second, whereas the disktab file stores it in RPM.						
fs_maxcontig	The maximum contiguous blocks to transfer before a delay. See above						
fs_maxbpg	The maximum number of blocks a file can allocate from a single cylinder group before looking for space elsewhere. This value is used once the direct blocks have been used.						
fs_fsbtodb	The shift value used to convert from logical sector size, DEV_SIZE (1K) to the fragment size. Here the term block is being used for a third meaning, the DEV_SIZE unit that is used for all transfers.						
fs_sbsize	The size of the superblock on disk. The size of the fs structure rounded up to whole fragments.						
fs_cssize fs_csmask fs_csshift	The size of the cylinder group summary area.						
fs_csaddr	The address on the disk of the summary area. This is typically the first data block within the file system.						
fs_nindir	The number of indirection pointers that can be held within one block.						
fs_inopb	The number of inodes per disk block						
fs_nspf	The number of DEV_BSIZE sectors per fragment						
fs_id[2]	File system id, this does not appear to be used on HP-UX.						
fs_mirror	Status of mirroring of kernel devices						
fs_featurebits	Controls file system features such as accepting large files.						
fs_optim	Allows the allocation routines to select the optimum layout either for time or space. <table border="1" data-bbox="399 1585 1085 1657"> <tr> <td>FS_OPTTIME</td> <td>0</td> <td>Optimize for performance</td> </tr> <tr> <td>FS_OPTSPACE</td> <td>1</td> <td>Optimize for space utilization</td> </tr> </table> However HP-UX does not appear to use this UFS feature.	FS_OPTTIME	0	Optimize for performance	FS_OPTSPACE	1	Optimize for space utilization
FS_OPTTIME	0	Optimize for performance					
FS_OPTSPACE	1	Optimize for space utilization					
fs_cgsize	Size of the cylinder group data structure, this is affected by the number of fragments per cylinder group.						
fs_ntrak	The number of tracks per cylinder, the number of data surfaces on the disk drive.						
fs_nsect	The number of DEV_BSIZE sectors per track.						
fs_spc	The number of sectors per cylinder, fs_ntrak * fs_nsect						
fs_ncyl	The number of cylinders in the file system						
fs_cpg	The number of cylinders in a group. The maximum is 32						

fs_ipg	The number of inodes per cylinder group. The maximum is 2048	
fs_fpg	The number of fragments per cylinder group. The maximum is limited by the $(fs_bsize - 984) * 8$. 984 is the location of the start of the free fragment map within the cylinder group.	
fs_cstotal	The overall file system summary (struct csum)	
	cs_ndir	number of directories
	cs_nbfree	number of free blocks
	cs_nifree	number of free inodes
cs_nffree	number of free frags	
fs_fmod	A flag set whenever the superblock is changed.	
fs_clean	Indicates whether the file system was cleanly unmounted or whether fsck needs to be run before attempting to mount this file system	
fs_ronly	The file system has been mounted read only	
fs_flags	Not used	
fs_fsmnt	The name of the mount point we are/were mounted on.	
fs_cgrotor	The last cylinder group searched for disk space	
fs_postbl	The table of block numbers at the start of each rotational position of the cylinder group. Each track/cylinder is divided into NRPOS(8) positional areas for the allocation routines. This table gives the sector number to be found at the start of each of these areas. Since the number of sectors per track is not always a multiple of the block size a repeating pattern is used. The number of rows in this pattern is given by the parameter fs_cpc below.	
fs_cpc	See fs_postbl above.	
fs_magic	The file system magic number . On HP-UX the follow values are valid	
	0x011954	A short filename system (McKusick's birthday)
	0x095014	A long filename system
	0x195612	The type of file system is controlled by bits in the fs_featurebits field
	0x05231994	Supports large (>4GB) file systems & featurebits
	0x612195	B1 secure file system
fs_fname	Unused	
fs_fpack	Unused	
fs_rotbl	A table of offsets to the give the next block that starts within the same rotational portion of the cylinder. While this is declared as an array with only one element the size of this table varies according to the parameters given to mkfs when the file system was created. Size of fs_rotbl = $fs_cpc * fs_spc / block\ size$	

HP-UX 10.16 - B1 USA government
10.24 - коммер. вариант

Module 8
File Systems

The rotational field information is needed to deal with the problem that disk drive manufacturers do not design their product to fit with the UFS file system. If they did, all disks would have a number of sectors per track that was a multiple of the chosen block size. Since this is not the case, the file system needs some way of rapidly determining which blocks have similar rotational positions.

UFS divides each disk into 8 rotational areas. In the diagram the lines extending beyond the disk mark these areas.

The outer ring of numbers are the fragment numbers, and the inner numbers represent the block numbers. It is really these block numbers that we are interested in.

In this example there are 90 fragments in the cylinder. This will give 11 whole blocks and leave 2 remaining fragments. The twelfth block (number 11) will use these two and then need to cross to the next cylinder. At the end of the second cylinder block 22 will have 4 fragments before again moving onto the next cylinder. Likewise the third cylinder will end with block 33 using 6 fragments before moving onto the fourth cylinder. This fourth cylinder will end with block 44 finding the required 8 fragments without extending across a cylinder boundary. In this way a repeating pattern of 4 cylinders can be built up.

When the allocation routines are looking for a space in a certain position within the file system, they use the 8 rotational areas rather than having to calculate exact rotational positions, this eases the calculations.

The table **fs_postb1** within the superblock gives the block number of the first block within each cylinder with each of the rotational areas. The first number in each cell is from this table.

cylinder	position							
	0	1	2	3	4	5	6	7
0	0 7, 9	5	3	1 10	8	6	4	2 11
1	16 18	14	12 21	19	17	15	13 22	20
2	25 27	23 32	30	28	26	24 33	31	29
3	34 36, 43	41	39	37	35 44	42	40	38

Since the number of blocks per cylinder is not a function of the number of rotational positions, more than one block may start in some rotational positions. The **fs_rotb1** table on the end of the superblock provides the alternative block numbers by giving there offset from the last one tried.

0=7	1=9	2=9	3=0	4=0	5=0	6=0	7=2
8=0	9=0	10=0	11=0	12=9	13=9	14=0	15=0
16=2	17=0	18=0	19=0	20=0	21=0	22=0	23=9
24=9	25=2	26=0	27=0	28=0	29=0	30=0	31=0
32=0	33=0	34=2	35=9	36=7	37=0	38=0	39=0
40=0	41=0	42=0	43=0	44=0			

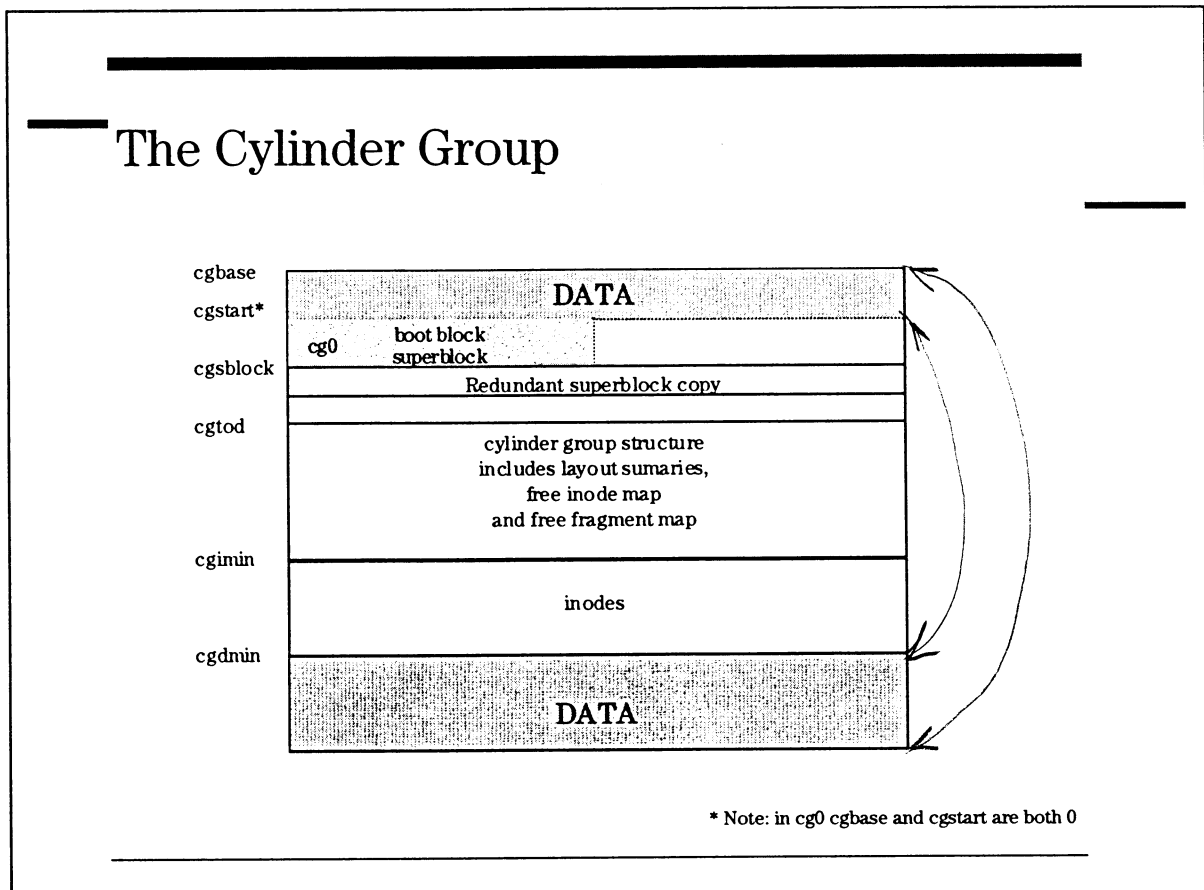
If block number 0 is unavailable, try adding 7. If block 7 is also in use, add 2 and see if block 9 is available. If block 9 is in use, then move on to the next cylinder.

Эвристичи размещение блоков - 5

лучше следующим

лучше в след секторе $1/8$ зарезан

8-7. SLIDE: The Cylinder Group



Student Notes

The cylinder groups provide a method of breaking the file system up into smaller pieces to allow data and control structures to be localized. Each cylinder group contains a user data and a section of file system structural data. The position of this structural data changes from one cylinder group to the next. It does this so that not all of these structures end up on a single platter. Therefore a hardware failure of part of the drive should not result in the loss of the information from all of the cylinder groups.

The offsets, **cgbase**, **cgstart** ... are defined as macros in the header file **fs.h**.

- Cgbase** The start of the overall cylinder group. It is found by multiplying the size of the cylinder group by the cylinder group number.
- Cgstart** Calculated using an offset from **cgbase**, but adding on approximately a track for each cylinder group, so as to move the area of control information. The remaining values are all calculated from here.
- Cgsblock** The start of the redundant superblock copy. For cylinder group zero, **cgstart** works out as 0. The start of the file system has space for a

bootblock (8K), and the primary superblock. The first redundant superblock starts on the next block boundary. So **cgsblock** will be **cgstart** plus 8K plus the superblock size rounded up to the next block boundary.

Cgtod The main cylinder group data structure of type **struct cg** then starts at the next block. It contains: -

cg_unused[2]	unused	
cg_time	The last update time	
cg_cgx	The index number of this cylinder group.	
cg_ncyl	The number of cylinder in this group. The last group will be short unless the total number of cylinders is a multiple of ncyl	
cg_niblk	Inodes in this cylinder group	
cg_ndblk	Data fragments in this cylinder group	
cg_cs	Summary information, struct csum	
	cs_ndir	The number of directories
	cs_nbfree	Number of free blocks
	cs_nifree	Number of free inodes
	cs_nffree	Number of free fragments
cg_rotor	Radial position of the last used block	
cg_frotor	Radial position of the last used fragment	
cg_itor	Radial position of the last used inode	
cg_frsum[MAXFRAG : 8]	A count of the number of 1,2,3,4... frag pieces	
cg_btot[MAXCPG : 32]	The free blocks per for each cylinder	
cg_b[MAXCPG][NRPOS : 8]	Free blocks per cylinder per radial position	
cg_iused[MAXIPG/NBBY] 2048/8	Free inode map for the cylinder group, there can be a maximum of 2048 inodes per group	
cg_magic	The magic number 0x090255	
cg_free[1] actually the remainder of the block	The free fragment map.	

Cgimin The location of the first inode in the cylinder group. Each inode is 128 bytes long and there can be a maximum of 2048 in the cylinder group.

Cgdmn The address of the first data block after the control area. There is also data before the control information.

This arrangement of the cylinder group structure imposes several limits on the way that file system got created.

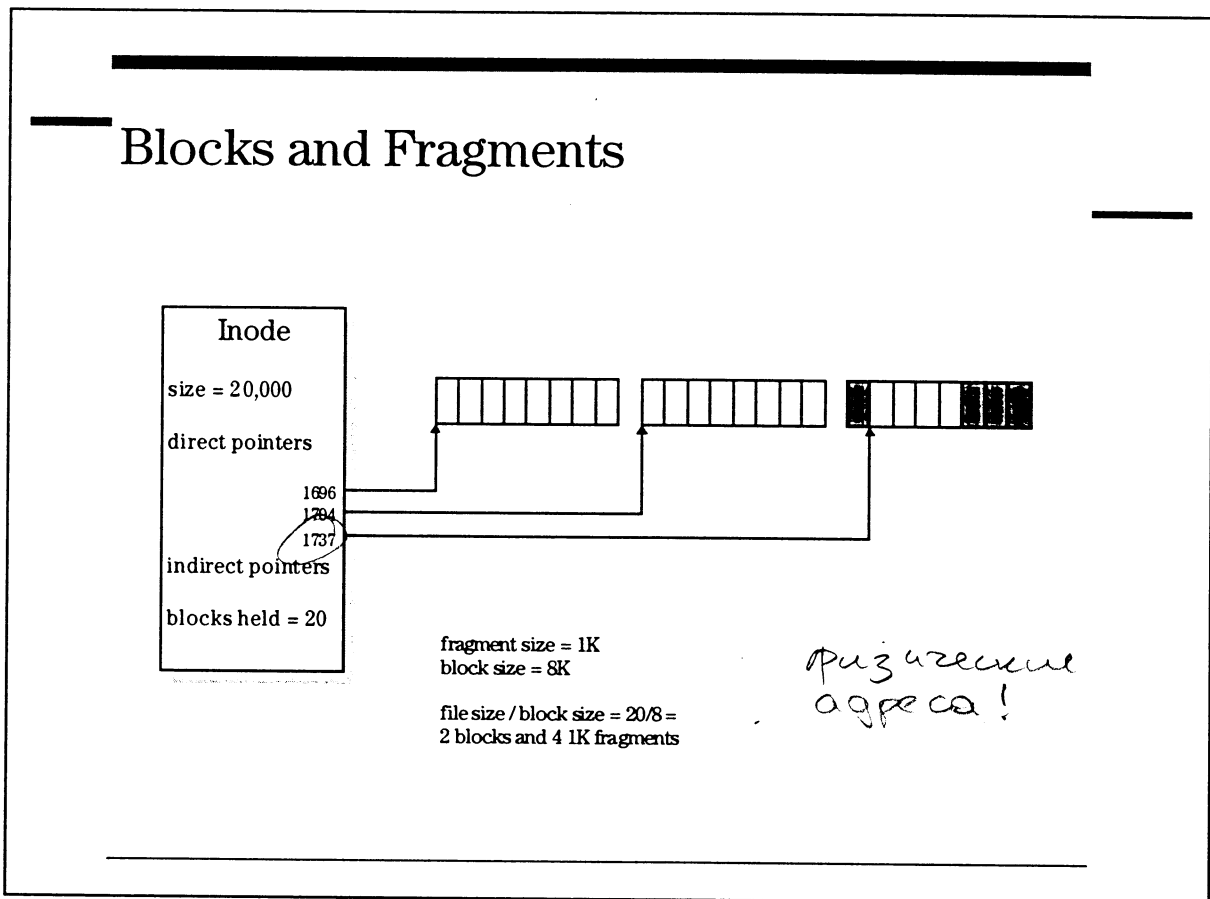
- The free inode map is a fixed-sized structure, so this imposes a limit on the number of inodes that can be created in a cylinder group.
- The fragment freemap starts at the end of the data structure, and the rest of the structure is 984 bytes long. Since the section of the inode table starts a block further into the file system, the **freemap** needs to fit into the gap. All fragments need an entry in the map. The maximum number of fragments in the cylinder group will be $(1\text{block}-984)*8$.

Module 8
File Systems

The rotor information is used by the allocation algorithms to select the best block to use.

The **freemaps** set the bit to 1 to indicate that the space is free, and 0 to indicate that it is in use.

8-8. SLIDE: Blocks and Fragments



Student Notes

One of the identified with the original Bell file system was the small block size leading to inefficient transfers.

The UFS tackles this issue by using a larger block size and sub-dividing these blocks into fragments.

Unfortunately the term blocks is frequently misapplied. It is often used when referring to fragments and even occasionally when referring to sectors.

Where a file is only using direct pointers, it is able to make use of fragments to avoid having to allocate a whole block, where not all of the space is currently required. There are restrictions on the use of fragments:

Module 8
File Systems

- Fragments are only used at the end of files.

As we can see from the slide, the overall size of the file is divided by the block size to calculate the number of whole blocks in the file.

Since there is no information stored with the pointer about how many fragments it is pointing to, it is necessary to ensure that the pointers point to a known size object.

- All fragments must be contiguous.

With the last pointer, the remainder of the earlier calculation is used to calculate how many fragments are needed to store the last part of the file.

- All the fragments used must come from the same block.

All the addresses used on the file system are fragment addresses (unless stated otherwise). The pointers used within the inode to reference the blocks of the file actually hold fragment addresses.

```
Disks - interactive disk editor [version 9.0 [NFS/HP-ACLS/4GB_FS]]
Command: i37
displaying inode 37 from fragment 36:

mode: 0100666  IFREG rw-rw-rw-
size : 20000  links : 1  uid : 0  gid : 3
time last accessed      : Tue Aug  4 16:17:10 1998
time last modified     : Tue Aug  4 16:17:10 1998
last time inode changed: Tue Aug  4 16:17:10 1998
data blocks: fragment addresses
      0: 1696      5:      0      10:      0
      1: 1704      6:      0      11:      0
      2: 1737      7:      0      si:      0
      3:      0      8:      0      di:      0
      4:      0      9:      0      ti:      0

blocks actually held 20
fversion : 0  ic_contin : 0  generation : 0
```

Orange fsdb

8-9. SLIDE: Inodes

Inodes

Describe

- type
- permissions
- number of names
- ownership
- size in terms of length
- size in terms of fragments held
- usage times
- one of the locations of data on the file system
- Major and Minor number
- link target name
- pipe information

Student Notes

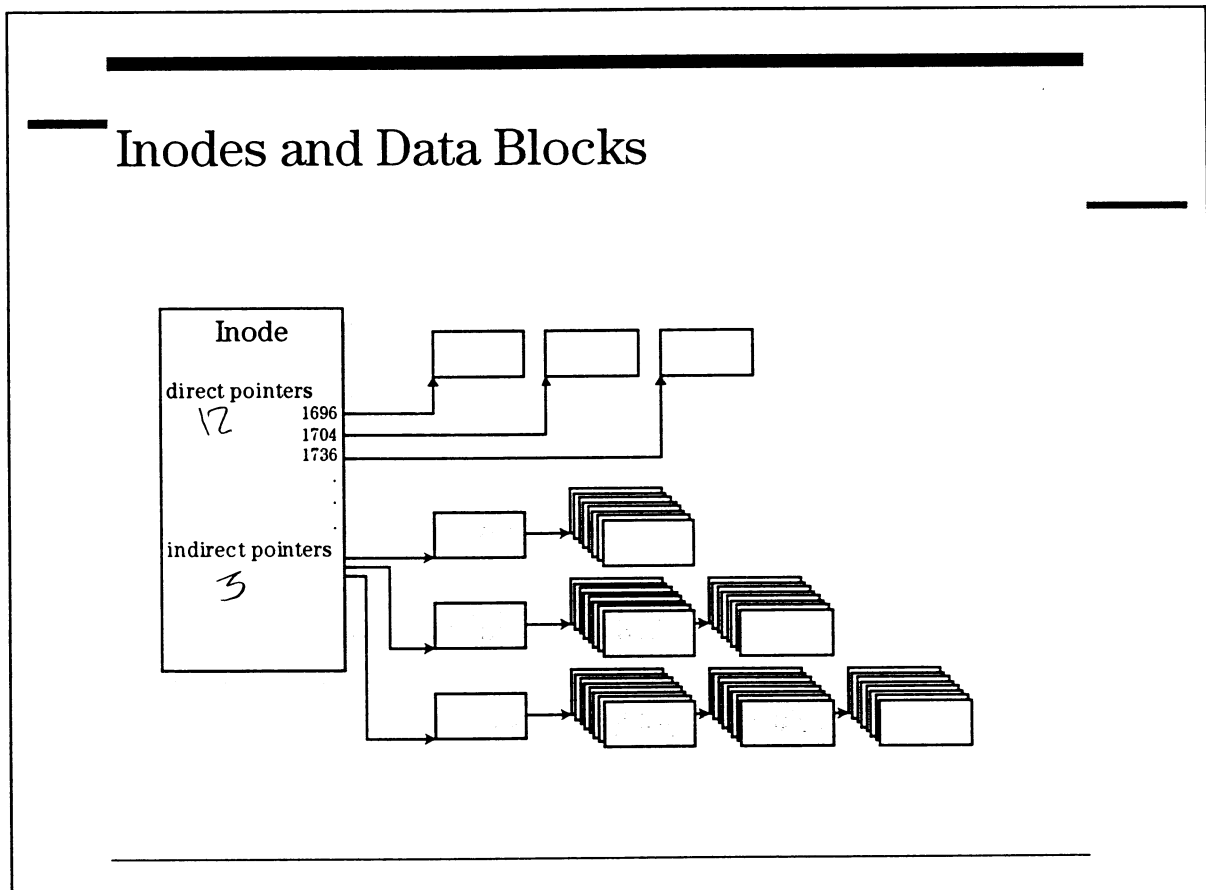
Each file on the file system is described by an inode, whether it is a regular file, a device file or a fast symbolic link.

The layout of an inode is defined as the structure `icommon` in the header file `/usr/include/sys/inode.h`. It is described as a common inode since the structure is used both within the kernel's in-core inode structure that we will see later.

ic_mode	The files type and permissions			
	Type	There are just four bits in a UFS inode to describe its type		
		0010000	IFIFO	Pipe
		0020000	IFCHR	Character special file
		0040000	IFDIR	Directory
		0060000	IFBLK	Block special file
		0070000	IFCONT	Continuation inode
		0100000	IFREG	Regular file
		0110000	IFNWK	Network special file (network special files were last supported under HP-UX 5.3 and only ever on the 300 series)
		0120000	IFLNK	Symbolic link, check the ic_flag field to find out the format
		0140000	IFSOCK	UNIX domain socket
	Permissions	There are 12 permission bits; the least significant 9 give the standard UNIX rw-rw-rw permissions. The top three give		
		SUID	For a program it sets the effective user id to the owner of the file	
		SGID	For a program in sets the effective group id to that of the file.	
For a data file is make files locks mandatory.				
Sticky or Text		For a directory it sets the group id of all files created in the directory to the group id of the directory.		
	For NFS based executables it causes the text to be paged to the local swap area.			
		For directories is only allows remove and rename operations on files within the directory if the user id matches either that of the file or the directory or if the user id is 0.		
ic_nlink	The link count of the file			
ic_uid_lsb	The least significant 16 bits of the user id. Prior to HP-UX 10.20 user and group ids were limited to only 16 bits and were stored here. With the expansion of UIDs to 32 bits then rather than expand this field and risk breaking backward compatibility with earlier code that read the inode information, the UID and GID fields were split, the most significant half of these values are stored towards the end of the inodes.			
ic_gid_lsb	The least significant 16 bits of the group id.			
ic_size	The offset of the last byte in the file. Generally the size of the file but sparse files confuse this.			
ic_atime	The time the file was last accessed (read), this is stored both as the number of seconds since the beginning of 1970 and the microsecond offset.			
ic_mtime	The time the file was last modified. As with ic_atime , this is stored both as the second and microsecond figure.			

ic_ctime	The time the inode was last updated for something other than a time stamp. This value can not be set via the utime system call.			
ic_un2	A union containing the different data depending upon the type of the file.			
	Device Files	The major and minor numbers are held in the first word.		
	Fast symbolic links	The name that the link reference is stored as a null terminated string, of length 60. (59 + the null)		
	Others	ic_db	The twelve direct pointers. These hold the fragment addresses of the start of the first 12 blocks. The last allocated one may point fragments	
		ic_un	Another union depending upon the type	
	fifos	ic_fifo	Status information about the pipe.	
	Others	ic_ib	The three indirect pointers.	
ic_flags	Currently the only flag used is IC_FASTLINK to indicate that the symbolic link is a fast one, i.e. the name is stored within the inode rather than on a separate fragment. <i>go 60 bytes - 3 inode (get SAM)</i>			
ic_blocks	The number of fragments held by the file, this is the value reported by du and ls -s			
ic_gen	The generation number of the inode. Each time an inode is re-allocated then the generation number is incremented. This allows programs, and particularly NFS to know whether this has happened. NFS accesses files using a file handle, which consists of various bits of information including the inode and the generation number. Should a file be deleted and a new one be issued the same inode, this generation field would should NFS that this is not the same file. Not all versions of HP-UX correctly manipulate this field.			
ic_uid_msb	The most significant part of the user id.			
ic_gid_msb	The most significant part of the group id.			
ic_spare	Some padding to make the length of the inode 128 bytes.			
ic_contin	The inode number of the continuation inode. Continuation inodes are how HP-UX implements access control lists for extended file permissions.			

8-10. SLIDE: Inodes and Data Blocks



Student notes

The UFS inode contains 12 direct pointers. If the file grows beyond this point, a system of indirection is used.

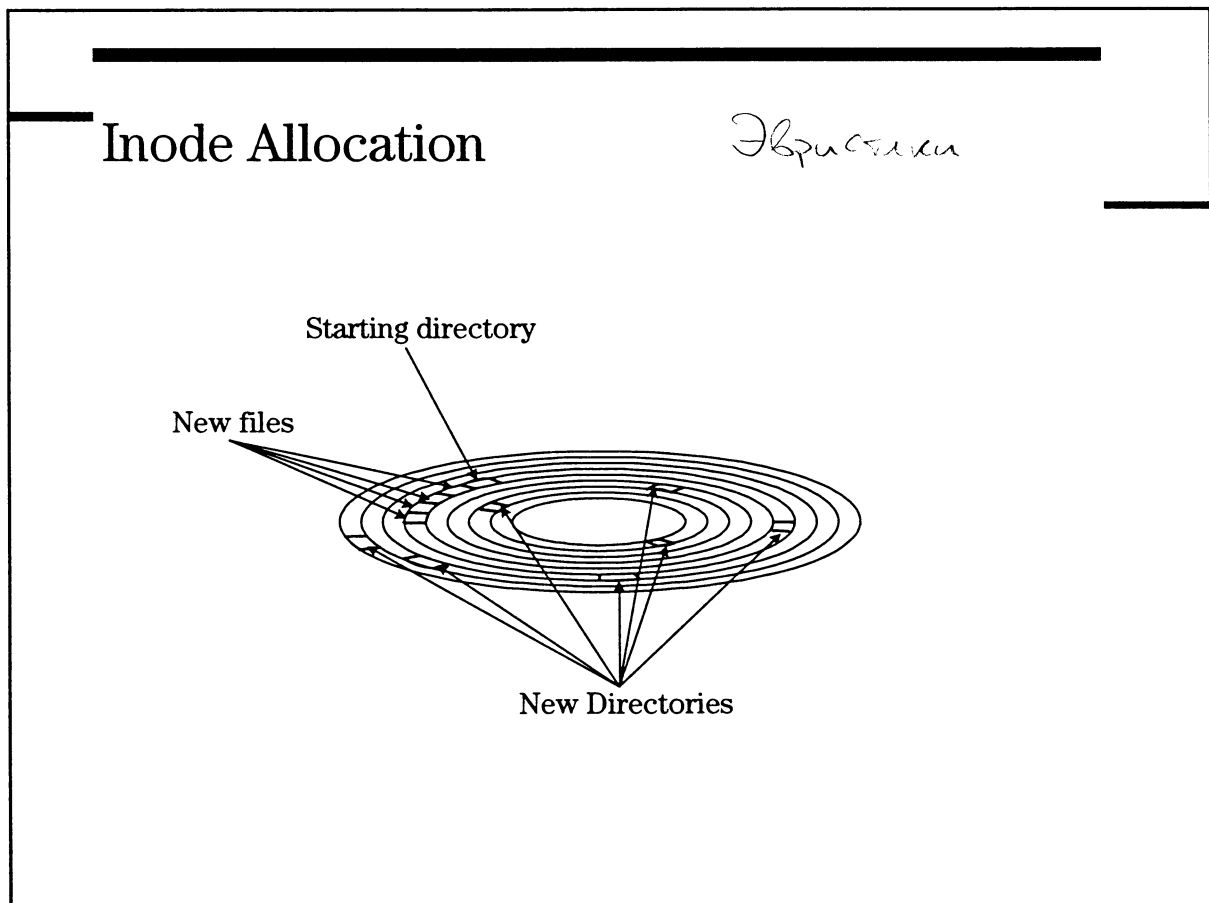
Once the file grows beyond just using the direct pointers, it is no longer able to use fragments to conserve space. Since the amount of wasted space would not exceed about 8%, this is not viewed as a problem.

The layout rules for disk blocks will position the remainder of the file outside the cylinder group where the inode resides. This helps other files to find space close to their inodes. From this point onwards the layout of the file will be controlled by the `maxbpg` tuning parameter.

The thirteenth pointer from the inode is for single indirection, using an 8K block size. This block could contain 2048 pointers — enough to map an additional 16Mbytes of file space.

The fourteenth pointer is for double indirection. Should more space be required, the last pointer provides for triple indirection. However with HP-UX this is only required when the file system is configured to support *largefiles* (over 2Gbytes, requiring 64bit system calls).

8-11. SLIDE: Inode Allocation



Student Notes

Now that we have an understanding of the physical layout of a UNIX File System, we want to look at how the kernel determines where to place inodes and data blocks within the file system.

When creating a file or a directory, the routine `direnter()` is called (from someplace like `ufs_create()`) to add a file/directory entry to a directory block. `direnter()` calls `dirmakeinode()` to create the inode for the file to be added.

The general algorithm of `dirmakeinode()` is as follows:

1. Determine the preference for the inode to be created.

The goals of the preferences are to:

- Allocate inodes for files in the same cylinder group as the parent.
- Allocate inodes for new directories in the cylinder group with a higher than average number of free inodes and the small number of directory entries.

When **dirmakeinode()** is called, it receives the parent's inode pointer as an input parameter. If type of file being created is not a directory, then parent's inode number is used as the preference. If the type is a directory, **dirpref()** is called to determine the best cylinder group for allocation.

This routine first determines the average number of free inodes in each cylinder group (**avgifree**) based on summary information in the superblock. Then, starting with cylinder group 0, **dirpref()** reads through each cylinder group checking the number of directories (**cs_ndir**) and the number of free inodes (**cs_nifree**). If the number of free inodes is greater than the average, and the number of directories is less than any cylinder group already checked, **mincg** is set to this cylinder group as the best candidate.

When done, **dirpref()** returns an inode pointer to the first inode in the located cylinder group.

2. Determine the specific inode to be allocated based on the preference.

At the heart of the inode allocation process is the **ialloc()** routine which takes the inode preference found in **dirmakeinode()** and finds the best inode for allocation. We will discuss the details of **ialloc()** on the next slide.

3. Allocate and initialize the inode.

After **ialloc()** is complete we will have the best free inode based on the preferences. **dirmakeinode()** will flag the inode as being changed (ICHG), set the link counts, **uids**, and mode before calling a final routine to complete the inode allocation.

If the inode being allocated is a directory, then **dirmakedirect()** is called to allocate an empty directory data block. If not a directory, then **iupdat()** is called to fill in the remaining fields in the inode, such as timestamps, and initialize all the data block pointers to zero.

Inode Allocation with **ialloc()**

Once a preference has been determined, **dirmakeinode()** calls **ialloc()** to allocate an inode based on the given preference. In reality, the inode preference is an optional argument to **ialloc()** and the general algorithm followed is based on the preference. That algorithm is to:

- Allocate the requested inode (if preference specified).
- Allocate an inode in the same cylinder group, or in cylinder group zero if no preference is specified.
- If no inodes are available in cylinder group, then a hash function is used to locate an available inode in another cylinder group.

All of this is driven from **ialloc()** which also calls the routines **hashalloc()** and **iallocg()** to perform the work. Let's look at the specifics of **ialloc()**.

1. Check the inode purge list and post the first entry.

The file system maintains a linked list of inodes that have been deallocated and are waiting to be posted to disk. This list is pointed to by the kernel variables **ipurgeh** and **ipurget** (head and tail). Entries are linked together with the **i_freef** and **i_freet** fields of the in-memory inode structure.

ialloc() calls the routine **check_purge_list()** to see if there are any entries on this free list. If there are, it calls **post_inactive()** to force the first entry on the list to be posted to disk. This is done because entries on the purge list are not actually available yet. It would be possible for it to appear that there are no inodes available while there are inactive ones on the purge list. By forcing the first entry to be posted we are sure there will be an inode available.

The **check_purge_list()** routine is only called if the number of free inodes recorded in the superblock (**fs_cstotal.cs_nifree**) is zero. If the number of free inodes is zero and no entries are found on the purge list then ENOSPC will be returned and message "out of inodes" will be reported.

2. Get cylinder group pointer for preferred inode and call **hashalloc()**.

hashalloc() is the routine that implements the policies of looking for the inode in the same cylinder group, then using the hash function, then using brute force to find an inode. When **hashalloc()** completes we will either return a pointer to a free inode or indication that no inodes exist.

3. **hashalloc()** calls **ialloccg()** to allocate an inode in the preferred cylinder group. **ialloccg()** will perform the following steps:

- Read cylinder group block for preferred block.
- Check cylinder group summary information to make see if there are free inodes in the cylinder group (**cs_nifree > 0**). If **cs_nifree** equals zero, no free inodes are available and control returns to **hashalloc()**.
- Convert the preferred inode number to an inode number relative to the start of the preferred cylinder group. If the inode is free use it, otherwise set the preference to be the last allocated inode (**cg_itor**).
- Starting with the preferred inode number, move sequentially through the inodes until one is found that is free
- If a free inode is found then we:
 - Set bit in **cg_iused** to indicate inode is in use.
 - Decrement number of free inodes in the cylinder group's summary information.
 - Decrement number of free inodes in superblock's summary information.
 - Determine if a directory increment directory counts in cylinder group and superblock.
 - Write the cylinder group block back to disk.

4. If no free inode was found in `ialloccg()` then `hashalloc()` performs a *quadratic rehash* to move through other cylinder groups.

This process starts with cylinder group 1 and increments the group number by a power of two on each iteration. On each iteration, `ialloccg()` is called to look for a free inode in the cylinder group

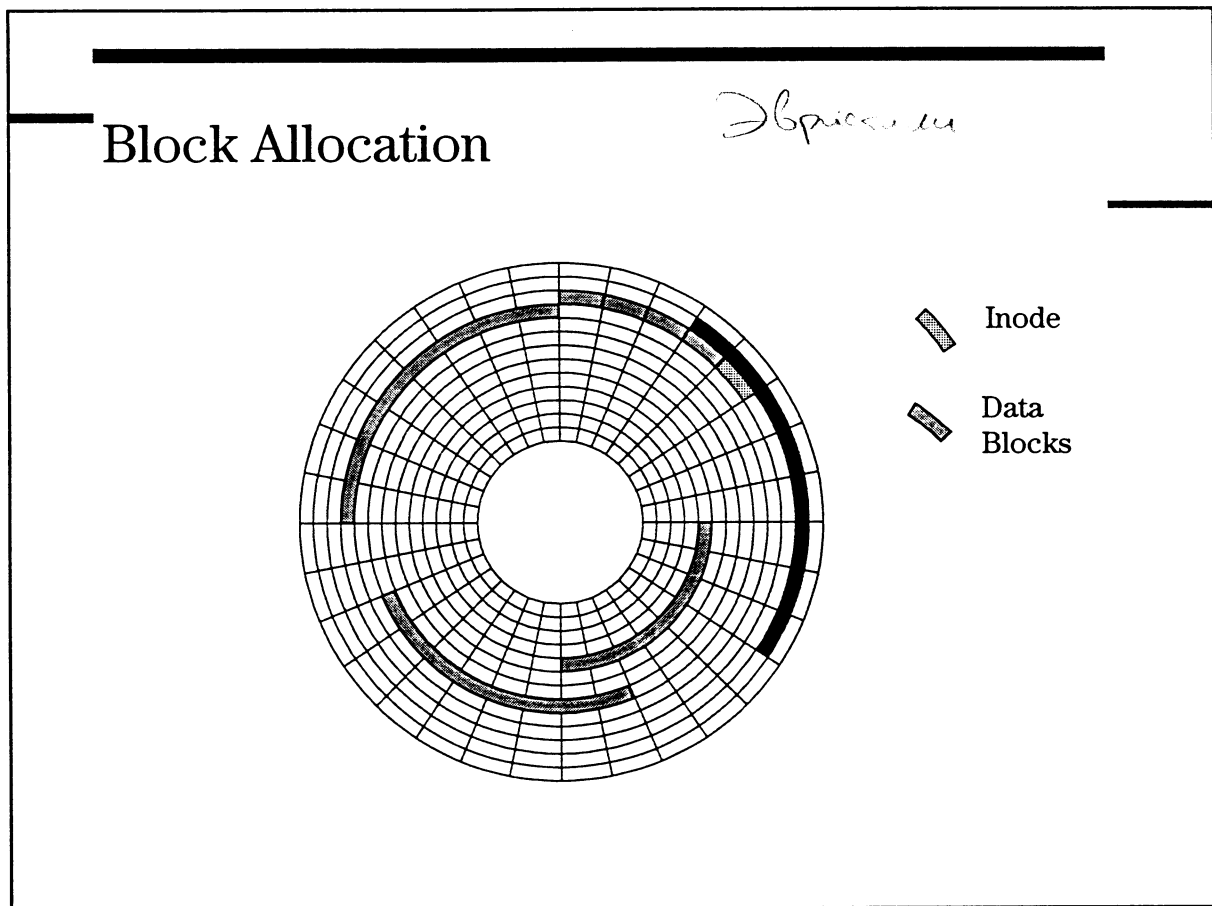
5. Perform a brute force search of the cylinder groups.

If a free inode was not found in the preferred cylinder group (step #3) or through the hash of cylinder groups (step #4) then we do an exhaustive search of each cylinder group, starting with cylinder group number 2.

6. We are now back in `ialloc()`. If `hashalloc()` found no free inode through any of the methods discussed, then we report ENOSPC as discussed in step #1.

7. If `hashalloc()` found an inode, then `ialloc()` calls `iget()` to read the inode from disk and then proceed with the steps in `dirmakeinode()`.

8-12. SLIDE: Block Allocation



Student Notes

The allocation of data blocks for files is also based on a set of preferences similar to those for inode allocation. The general preferences desired are:

- If we are allocating a direct block, allocate the block in the same cylinder group as the inode that describes the file.
- If block not available or allocating an indirect block, then allocate the block in a group with greater than average number of free blocks.

The routine that determines the block to be allocated based on these preferences is **blkpref()**. In this routine a file is logically divided into sections. The first section is made up of all the direct blocks and then all subsequent sections contain **fs_maxbpg** blocks.

blkpref() must make a determination of whether we are allocating a fragment in a previously allocated block and if we are requesting a block in a new section. If the desired

block has not been previously allocated and we are not starting a new **fs_maxcontig** section, then we perform the following steps:

1. If allocating a direct block then the next block is set to the first fragment of the same cylinder group.
2. If not allocating a direct block then compute the starting cylinder group and average number of free blocks.

The superblock field **fs_maxbpg** specifies the maximum number of blocks that can be allocated in any cylinder group. After exceeding this value, the next block is allocated in the next sequentially number cylinder group. It is assumed that if we have exceeded **fs_maxbpg** N times, then data blocks will have been allocated in the next N cylinder groups following the cylinder group of the inode.

The average number of free data blocks is computed from the number of free blocks and number of cylinder groups recorded in the superblock.

3. The final step is to search the cylinder groups.

The search starts at **startcgr** which was computed in steps #2 and continues until **fs_ncgr** (the number of cylinder groups in the file system). If the number of free blocks in any cylinder group is greater than the average (**avgbfree**) then the block next block is set to be the first block of this cylinder group and **fs_cgrotor** is set to this cylinder group number.

If a block is not found when **fs_ncgr** is reached, a similar search loop is executed starting at cylinder group 0 through **startcgr-1**.

A second possibility for **blkpref()** is that we are requesting allocation in a block that has already been allocated. In this case we take the following action:

1. Compute the **nextblk** address to be the second fragment of the currently allocated block.

We don't know for sure which fragments in the block are free but since we are only determining a preference we make a guess that it will be the second block. There is no guarantee that this will be right but it is a better guess than the first fragment because we know it has already been allocated.

2. Check to see if **fs_maxcontig** has been exceeded.

If we have not exceeded the maximum number of contiguous blocks (**fs_maxcontig**) then we return the block address computed in #1 as the preference. Otherwise we proceed with the search based on rotational delay.

3. Compute next block based on based on rotational delay.

If the maximum number of contiguous blocks has been exceeded, we request the next block in the cylinder group to be at the next rotational delayed position.

Once a block preference a has been determined through **blkpref()** we are ready to physically allocate a block based on these preferences. The routines involved in allocating

the block are `alloc()`, `hashalloc()`, `alloccg()`, and `alloccgblk()` and the general algorithm is to:

- Allocate the requested block .
- Allocate a rotational optimal block in the same cylinder group.
- Allocate any block in the same cylinder group.
- If no data blocks are available in the same cylinder group, then the same quadratic hash function (`hashalloc()`) used for inode allocation is used to rehash to another cylinder group.

Similar to `ialloc()` for inodes, `alloc()` acts as the **front-end** for all data block allocation. The `alloc()` routine performs some simply sanity checks such as free space and effective user id (for minfree allocation) and then calls `hashalloc()` to request the allocation. Once the allocation is complete, `alloc()` calls `getblk()` to allocate a file system buffer from the buffer cache and returns a pointer to this buffer to the requester.

The `hashalloc()` routine searches through the cylinder groups looking for a free block in the same manner as for inode allocation. For each cylinder group that `hashalloc()` determines should be searched, `alloccg()` is called to perform the following steps to see if a block of the needed size is available and to allocate the block :

4. Retrieve the cylinder group block.
5. If a full block is being requested, call `alloccgblk()` to allocate a full block.
6. If requesting a partial block and there are free blocks available, then set the allocation size to be a block.
7. Search for contiguous fragments of the size needed.

The `cg_frsum[]` array in the cylinder group block is used to search for a group of contiguous fragments of the size needed. We first search for a block of fragments of the size needed and work up toward a whole block until an allocation unit is found

If no partial block of necessary size is found, we will later return a null pointer to the requester, signalling an error. Otherwise, we return a pointer to the located block.

8. Once a full or partial block is located, `alloccgblk()` is called to allocate the block.
9. Set the free fragment bits in the cylinder group and update the file system and cylinder group summary totals.

Another possibility for data block allocation is the case where data occupies a partial block and needs to grow. If contiguous blocks are not available, the existing blocks will need to be moved to a whole free block or partial block large enough to hold the old and the new data. This is accomplished through the `realloccg()` routine. The fundamental principles during

reallocation are the same as those we have already discussed, so we will not go into further detail about this process.

8-13. LAB: HFS Structures

1. Execute the script "hfs_setup" in "/usr/local/bin" to create an HFS file system, mount it, and populate it with a variety of files. See the script on the last page for the steps to accomplish all this. It will help answer several of the questions in this exercise.

```
# /usr/local/bin/hfs_setup
```

2. Display the superblock information for this file system and note the following values:

```
# tune2fs -v /dev/vg00/hfsvol
```

ncg (number of cylinder groups)	_____
size (size of complete file system)	_____
blocks (size of data area)	_____
bsize (block size)	_____
fsize (fragment size)	_____
minfree (minimum user unusable space)	_____
maxbpg (maximum blocks per cylinder group)	_____
cpg (cylinders per cylinder group)	_____
bpg (blocks per cylinder group)	_____
ipg (inodes per cylinder group)	_____

3. Dump the contents of the directory:

```
# БТТ. u gne uctadwzob!  
xd -xc /hfs | more
```

Is there a file called "rmfile1"? _____

If version 10.X:

Is there a file called "rmfile2"? _____

If version 11.X:

Is there a file whose eight-character filename has been replaced by all null characters ("0")? _____

4. Display a long listing of the root directory with inode numbers and note the following:

```
# ll -i /hfs
```

Is there a file called "rmfile1"? _____

Is there a file called "rmfile2"? _____

Why didn't it show up in the "ll" listing? _____

What file is symbolically linked to "notempty"? _____

What file is hard-linked to "notempty"? _____

Are directories "adir1" and "adir2" in the same cylinder group? _____

5. In a separate window, invoke the "hfs" version of "fsdb" using the interactive editor "ied":

```
# ied -h $HOME/.fsdb_hfs_hist -p "fsdb_hfs>" fsdb -F hfs /dev/vg00/hfsvol
```

6. Print out the inode for the root directory:

```
fsdb_hfs> 2i
```

How many hard-links does it have? _____

How many fragments does this directory take? _____

What is the fragment number of the first fragment? _____

7. Display the entries in this directory:

```
fsdb_hfs> fd
```

How does the order of the entries displayed by fsdb differ from that displayed by "ll"?

Why? unsorted

Module 8
File Systems

8. Dump the contents of the first directory fragment:

```
fsdb_hfs> a0.fc
```

Note: There is no way in “fsdb” to control the output. Using “xd” at the command line is better.

9. Display the inode of the symbolic link:

```
fsdb_hfs> 9i
```

10. Dump the contents of the data block pointed to by a0:

```
fsdb_hfs> a0.fc
```

What is contained in the data block? _____

11. In a separate window, dump the contents of the file:

```
# xd -c /hfs/name3 | more
```

What appears to be contained in the data block now?

Why the difference? _____

12. Display the contents of the inode for the device file:

```
fsdb_hfs> 10i
```

How does this format differ from the other inodes? _____

13. Re-display the files in the root directory:

```
fsdb_hfs> 2i.fd
```

The file “newfile” was added last in this directory. Why does it appear before other files added earlier? _____

Script for hfs_setup:

```
#!/usr/bin/sh

lvcreate -L 12 -n hfsvol /dev/vg00

newfs -F hfs /dev/vg00/rhfsvol

mkdir /hfs

mount /dev/vg00/hfsvol /hfs

touch /hfs/afile /hfs/rmfile1 /hfs/rmfile2

chmod 7777 /hfs/afile

echo "This file is not empty" >> /hfs/notempty

mkdir /hfs/adir1 /hfs/adir2

echo "hello" >> /hfs/afile2

echo "some stuff" >> /hfs/adir1/file1

echo "some stuff" >> /hfs/adir2/file1

echo "some more stuff" >> /hfs/adir1/file2

echo "some more stuff" >> /hfs/adir2/file2

ln /hfs/notempty /hfs/name2

ln -s /hfs/notempty /hfs/name3

mknod /hfs/device c 64 0x010000

rm /hfs/rmfile1 /hfs/rmfile2

touch /hfs/newfile
```

8-14. SLIDE: Problems with UFS

Problems with UFS

- Long **fsck** time after an improper shutdown.
- Layout rules optimized for small- to medium-sized files.
- Loss of 10% min free area. (*minfree*)
- Modern disk drives do not have constant geometries.
- Inodes are still held in a table.

Student Notes

While UFS remains an extremely successful file system design it is not without its problems. The most apparent to most system administrators is the long time taken to check the consistency of the file system after an improper shutdown.

The layout rules used by the file system group the data for small files around the inode, deliberately scattering larger files to increase the probability of being able to do this. These rules have resulted in a file system that is highly resistant to the fragmentation issues that cause problems to so many file system designs when used with active file populations. When the file population is expected to remain static and the files are large, then these layout rules do not result in the optimum arrangement.

Part of the mechanism for avoiding fragmentation problems is ensuring that there is always an adequate supply of free space. To this end UFS has the **minfree** parameter, typically surrendering 10% of it's capacity to the goal of avoiding fragmentation. Again, where the file population is expected to remain static, fragmentation is unlikely to occur.

Modern disk drives do not have constant geometries. When the file system was designed, disk drives had a fixed number of sectors per track¹. Modern disk drives have a constant sector length, giving them much higher storage capacities, but obviating many of the layout optimizations of the UFS file system.

Lastly the number of inodes is fixed when the file system is created. More can only be created when the file system grows or when it is recreated. Even the case of growing the system is of only limited use and the number of new inodes will only be a function of percentage of new space allocated to the file system.

¹ See the discussion of the `nsect` file system creation parameter on a previous page.

8-15. SLIDE: The Veritas Extended File System

The Veritas Extended Filesystem

- Journalled updates to file system structures, for fast **fsck**.
- Extent-based allocation.
- Flexible inode allocation. (нес озпариваемости)
- Layout rules efficient for large pre-created files.
- Online administration. — без umount, без resize, backup, defrag
- Many mount options for tuning and control of behavior. (perf vs - integrity)
he также Mesogarithm — и поизбогат.

Online JFS — setext

Student Notes

The Veritas file system's main feature is the journal or intent log. The major problem with the UFS file system is that if it is not properly shutdown, the repair process using **fsck** is very time consuming. The reason for this is that there is no record of what was occurring at the time the system went down, so **fsck** has no knowledge of what repairs are needed. It therefore has no choice but to read every single inode on the file system and produce its own free lists, read every directory and compare them against the list of inodes now known to be in use. Then compare all of this back against the free lists in the cylinder groups and finally update the superblock to reflect the view of the file system we have now derived.

In contrast, the Veritas file system uses an intent log. Before modifying any file system data structure the proposed updates are recorded in the intent log. Once the update has occurred the intent log entry can be marked as completed.

This log can then be used to work out what was happening on the file system in the event of an improper shutdown. **fsck** needs just to read the intent log and see if:

Mauc. log 16ms
OS or 200 1ms

- 1) An update is incomplete within the log, and therefore has not yet started on the main disk, in which case no action is required.
- 2) If the log contains a whole transaction but no record of it having been completed on the file system, then the entry identifies what needs to be repaired and also how to repair it.
- 3) Lastly, if the log contains a completion record **fsck** knows that the structures have been successfully updated and that no further work is required.

This idea of using an intent log is not new. Database systems have been using this technique for many years.

The intent log is not the only feature of the Veritas file system. Veritas tackled the problem of blocks being both the minimum unit of storage and the unit of transfer differently from McKusick. Where as McKusick used a larger block and split it into fragments, Veritas uses a small block size, but organizes them into contiguous chunks called extents. Transfers can then be made of any number of blocks from within one extent. Also the organization of files is based on these extents rather than on the individual blocks. So where as any 100MB file on a UFS file system is likely to need 12800 8K data blocks (plus 8 indirection blocks) the Veritas file system might store the file using 4 or 5 extents. This has some obvious performance advantages.

One of the problems that were identified with the Bell file system, which McKusick did not tackle, was the problem of the inode table. Both of the earlier file systems create a table of inodes when the file system is built. This limits the number of files that can be stored in the file system. Since the running out of inodes often results in the need to back-up, rebuild and restore the file system, it is normal to over-create inodes.

The Veritas file system uses a flexible number of inodes. If more are needed they are simply added so long as there is disk space to hold them (or the data for the new files).

This need to have a flexible size to the inode table tends to complicate the design of the file system. The obvious choice of a container for a flexible-sized inode table is a file. It would not, however, be a good idea if the user could delete it, so it and the file that holds the flexible-sized inode free list need to be truly hidden from the users. The Veritas file system achieves this by having more than one set of files within the file system. Currently there are two filesets. The first is used to hold the attributes of the file system itself, while the second holds the user data.

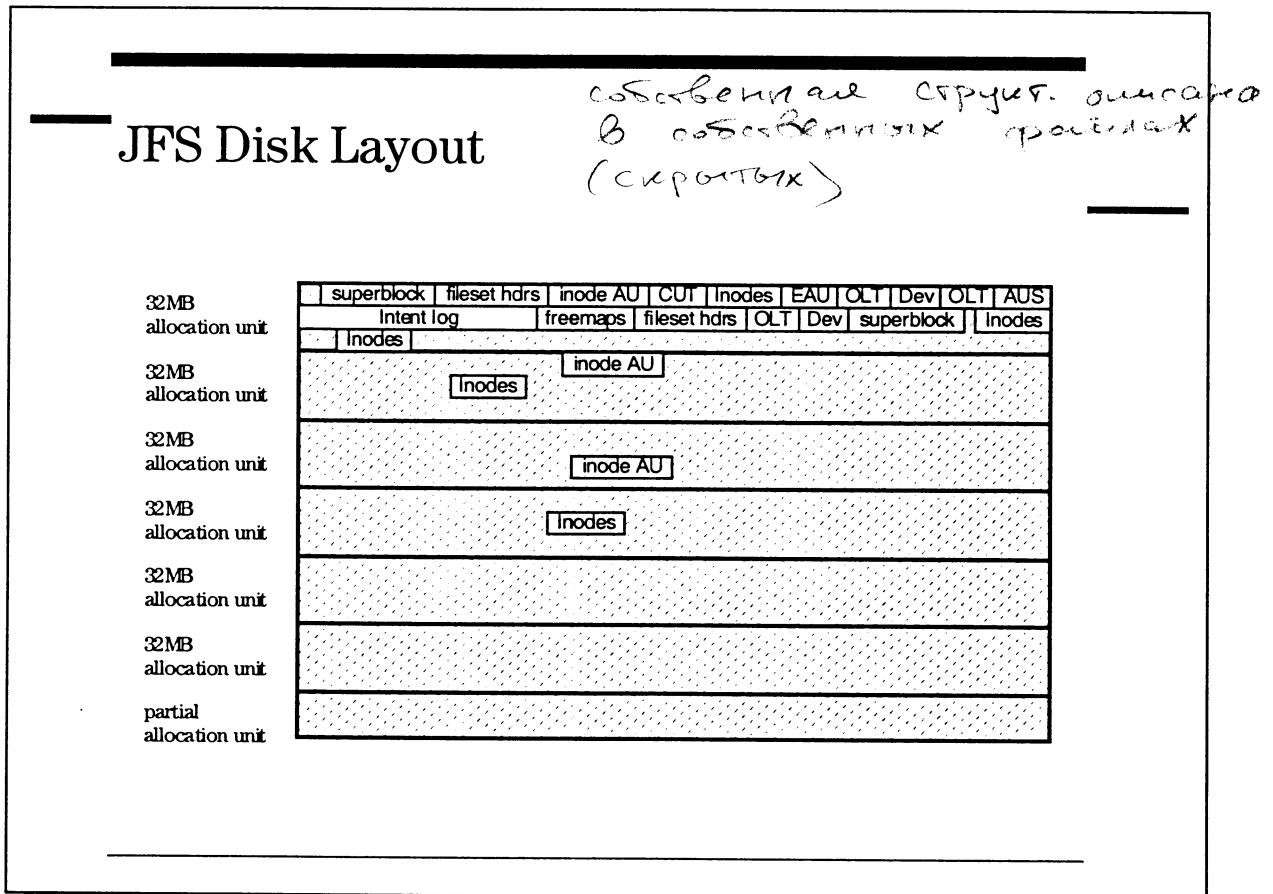
Lastly the Veritas file system uses a very different set of layout rules for its data from the McKusick file system. McKusick designed the UFS file system to cope well with the software development environment that was at the time viewed as the major use of UNIX systems. In such an environment there is constant creation, extension and deletion of files. Traditionally this has tended to badly fragment file systems causing either performance problems or the need to frequently reorganize the file system. McKusick designed the UFS file system to work in this sort of environment without becoming badly fragmented. To achieve this the layout rules for files and blocks tend to leave an even distribution of free space across the file system. Consequently whenever disk space is required the chances are that it will be available close to where it is required. This assumption tends to hold true while there is a reasonable amount of free space on the disk, and this leads to the **minfree** parameter.

Most modern file systems however have a very different usage pattern. With most database applications then the file system tends to hold a handful of large files. These files are then

Module 8
File Systems

stable. The data inside them changes frequently but the files themselves remain static. Since these environments don't tend to fragment the file system, the elaborate UFS rules are unneeded. Veritas file systems basically fill from the start packing the files in. So long as the file system does not become fragmented, this layout can yield higher performance.

8-16. SLIDE: JFS Disk Layout



Student Notes

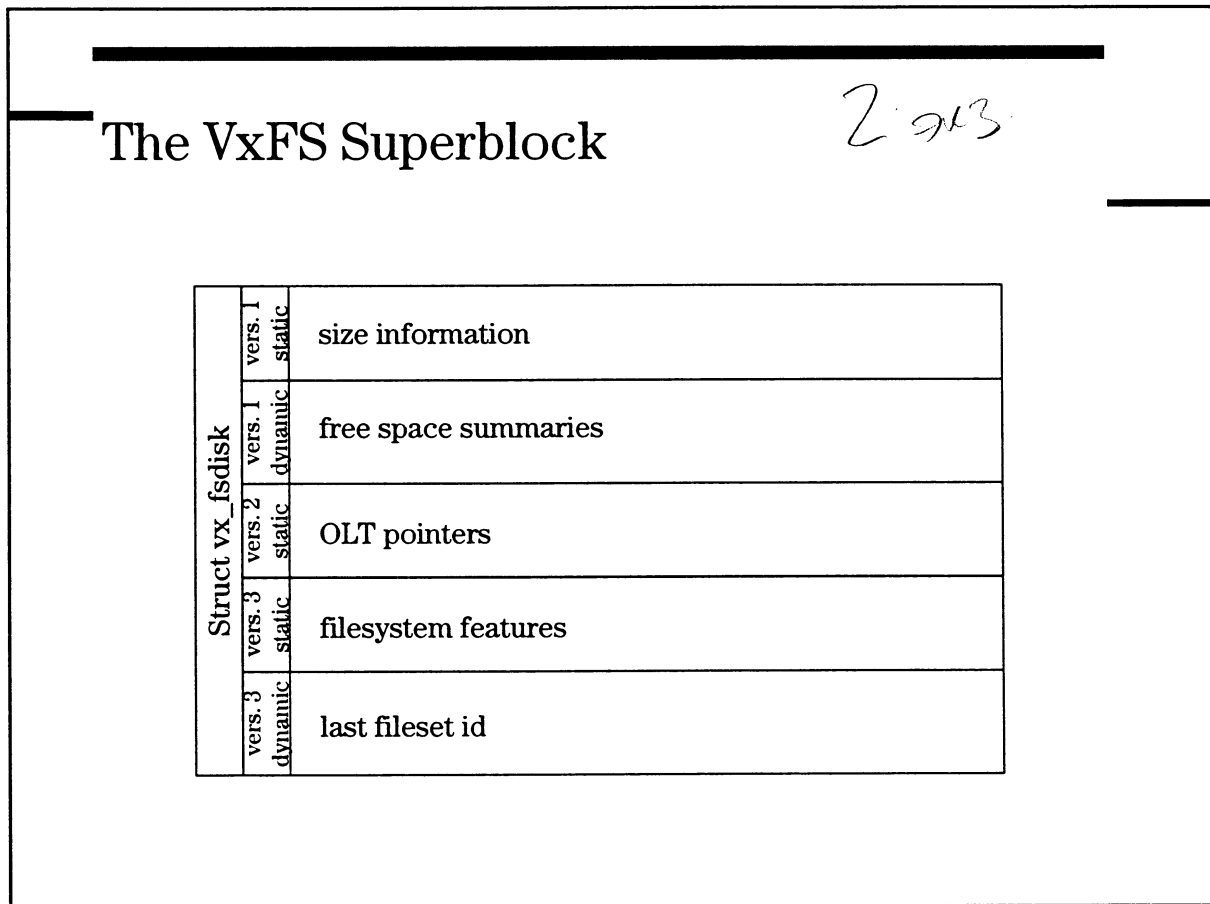
The Veritas file system shares many of the same concepts as the earlier UNIX file systems. It starts with a superblock, has inodes to describe the files and structures to describe free space. Also like UFS it divides the file system into smaller units to ease management. Here however no attempt is made to match these to the physical layout of the disk. Modern disk drives don't have constant geometry so the idea of the cylinder groups no longer works.

Additionally the Veritas file system has many new structures that are unique to it.

The actual structures of the superblock, inodes, and so forth, are different from the UFS case. On HP-UX the definitions of these structures can be found in the directory `/usr/include/sys/fs`, and the structures on the disk can be examined using `fsdb`.

This section makes heavy usage of `fsdb` to display the various structures on a VxFS disk.

8-17. SLIDE: VxFS Superblock



Student Notes

The role of the superblock is to provide an overall description of the file system. As with the UFS superblock it contains both static and dynamic data. Unlike the UFS example, these different types of information are held separately in sub-structures. The disk based superblock starts with the common read-only information, then has a separate structure for the read-write data. New versions of the file system (HP-UX 10.20 and 11.00 use VxFS version 3) then add new fields to the superblock by adding new sub structures, so there is now a static structure for both version 2 & 3 extensions and a new read-write area for version 3.

fsdb

(fsdb_uxfs, fsdb_hfs)

```
> 8b; p S          go to block 8 and print as a superblock
super-block at 00000008.0000
magic a501fcf5  version 3  ctime Fri May  1 15:54:36 1998  struct vx_fsdisk
log_version 6 logstart 0  logend 0
bsize 1024 size 102400 dsize 102400  ninode 0  nau 0
defiextsize 0  oilbsize 0  immedlen 96  ndaddr 10
aufirst 0  emap 0  imap 0  iexttop 0  istart 0
bstart 0  femap 0  fimap 0  fiexttop 0  fistart 0  fbstart 0
nindir 2048  aulen 32768  auimlen 0  auemlen 8
auiilen 0  aupad 0  aublocks 32768  maxtier 15
inopb 4  inopau 0  ndiripau 0  iaddrln 8  bshift 10
inoshift 2  bmask fffffc00  boffmask 3ff  checksum da55d376
free 82722  ifree 0
efree  2 10 1 1 2 1 1 1 0 1 0 0 2 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
flags 0 mod 0 clean 3c time Fri May  1 16:01:38 1998
oltext[0] 33  oltext[1] 1282  oltsize 1
iauiimlen 1  iausize 4  dinosize 256
dniaddr 3  checksum2 62c
features 0  checksum3 0
>
```

New versions often replace features of earlier ones, so many fields are no longer used. You can see that fields like **ninode** are 0, because from version 2 onwards inodes have been dynamically created. At version 3 the intent log is described by a file in **structural** fileset so **logstart** and **logend** are now unused.

Some key fields are: -

Version 3 This is a version 3 VxFS file system. If an attempt was made to mount this file system onto an earlier version of HP-UX it would know that this was a newer version of VxFS, that it could not handle it properly, and could return an appropriate error message.

Conversely new releases could see if this was an older version and would know how to talk to it.

Bsize 1024 This file system is using the default block size of 1KB.

Size 102400 The size of the file system is blocks.
Aulen 32768 The allocation unit size is 32MB.

Aublocks 32768 The number of data blocks in an allocation unit.

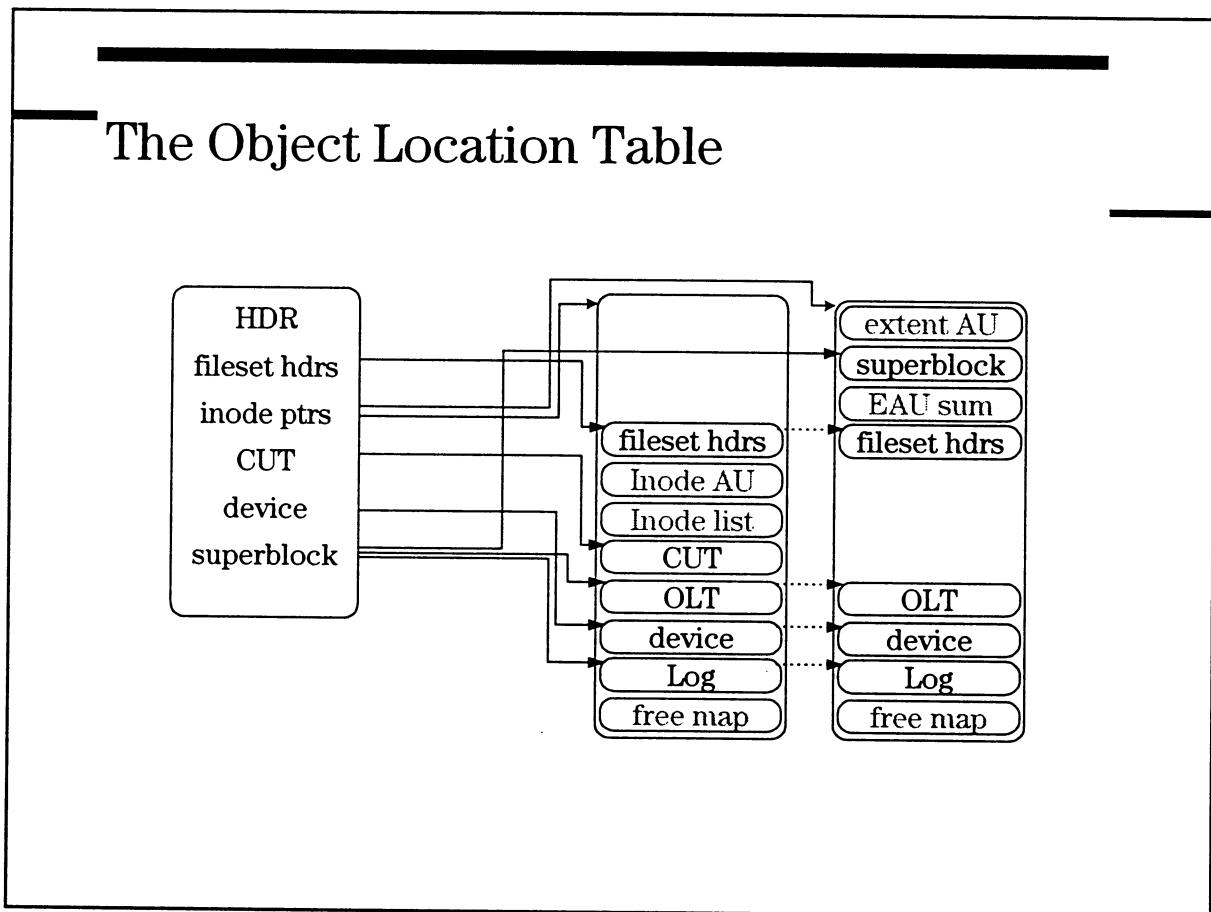
Efree The number of free extents on the file system as a whole.

The allocation policies for disk space within the Veritas file system work by trying to allocate large contiguous areas to files. In order to make this allocation easier, the superblock not only keeps track of the total amount of free space (**free**) but also the number of 1KB free extents, the number of 2K, 4K, 8K, and so on. This file system has a 32MB allocation unit size so the largest extent it keeps track of is limited to 32MB. This file system currently has 2 such extents free.

Module 8
File Systems

The `oltext` fields give the address of the *Object Location Table* (OLT). There are two of these, as the file system keeps redundant copies of key information. Loss of the OLT data would be as bad as losing the superblock.

8-18. SLIDE: The Object Location Table



Student Notes

The Object Location Table is used by the Veritas file system to locate most of its disk based data structures. The header uses a magic number to confirm that it is an OLT and a checksum to ensure its integrity. Also within the header there is a copy of the disk addresses for the two OLT copies.

Module 8
File Systems

```
> 33b; p olttext          from the superblock we see block 33 is the olt, or just
use the command olt
OLT at 0x00000021.0000
OLT head entry:          struct vx_olthead
    olt_magic 0xa504fcf5  olt_size 56  olt_totfree 872
    olt_time 894034476 378810  olt_checksum 0x354fbc44
    olt_esize 1  olt_extents[33 1282]
    olt_nsize 0  olt_next[0 0]
OLT fshead entry:       struct vx_oltfshead
    olt_type 2  olt_size 16  olt_fsino[3 35]
OLT initial iext entry: struct vx_oltulist
    olt_type 4  olt_size 16  olt_iext[16 1288]
OLT cut entry:         struct vx_oltcut
    olt_type 3  olt_size 16  olt_cutino 6
OLT device entry:     struct vx_oltdev
    olt_type 5  olt_size 16  olt_devino[8 40]
OLT super-block entry: struct vx_oltspb
    olt_type 6  olt_size 32  olt_sbino 33
    olt_logino[9 41]  olt_oltino[7 39]
OLT free entry:       struct vx_oltfree
    olt_type 1  olt_fsize 872
```

- The *OLT fshead entry* references the inode for the fileset headers within the *ATTRIBUTE* fileset; again there are two copies.
- The *OLT initial iext entry* points to the disk blocks that hold the initial portions of the *ATTRIBUTE* fileset. The rest of blocks can be found from the inode for file of inodes for this fileset. In order to find this information it would be necessary to use these disk blocks, and the inode numbers from the *fshead* entry above. Again there are two copies of this vital information. VxFS plays another trick to help in using the *ATTRIBUTE* fileset when there is damage to the file system. Most of the inodes are replicated and the second copy has an inode number 32 greater than the primary copies. Each of the two parts of this inode list referenced has 32 entries, so that the entry number of the primary copy is also the effective entry number of the second copy in the second part of the inode list.
- The *OLT cut entry* gives the inode number for the current usage table. The current usage table holds data about the fileset that changes regularly, like the number of blocks in use.
- The *OLT device entry* references the inodes for the device configuration record. This provides information about the size of the device (file system) and pointers to the free space summaries and maps.
- The *OLT superblock entry* keeps track of the superblock and its second copy (there are only 2!) by using an inode in the *ATTRIBUTE* fileset. This entry also keeps track of the log and the OLT itself.
- The *OLT free entry* describes the unused space in the OLT.

From these entries we can start to build a list of the inodes within the *ATTRIBUTE* fileset. Others can be found by following these starting points.

Inode #s	Type	Description
0	FREE	You can not use inode number 0
	32	IFEAU
		Extent Allocation Unit (eau state file)
1	FREE	
	33	IFLAB
		Device label file (looks very strange)
2	FREE	
	34	IFAUS
		Extent Allocation Unit Summary file
3	35	IFFSH
		Fileset headers
4		IFIAU
		Inode allocation unit for fileset 1 (see fileset header)
5	37	IFILT
		Inode list for fileset 1 (see fileset header)
6		IFCUT
		Current usage table
7		IFOLT
		OLT inode
8	40	IFDEV
		The device configuration inode
9	41	IFLOG
		Log inode
10	42	IFEMP
		Extent Map

fileset - map u name.

Module 8
File Systems

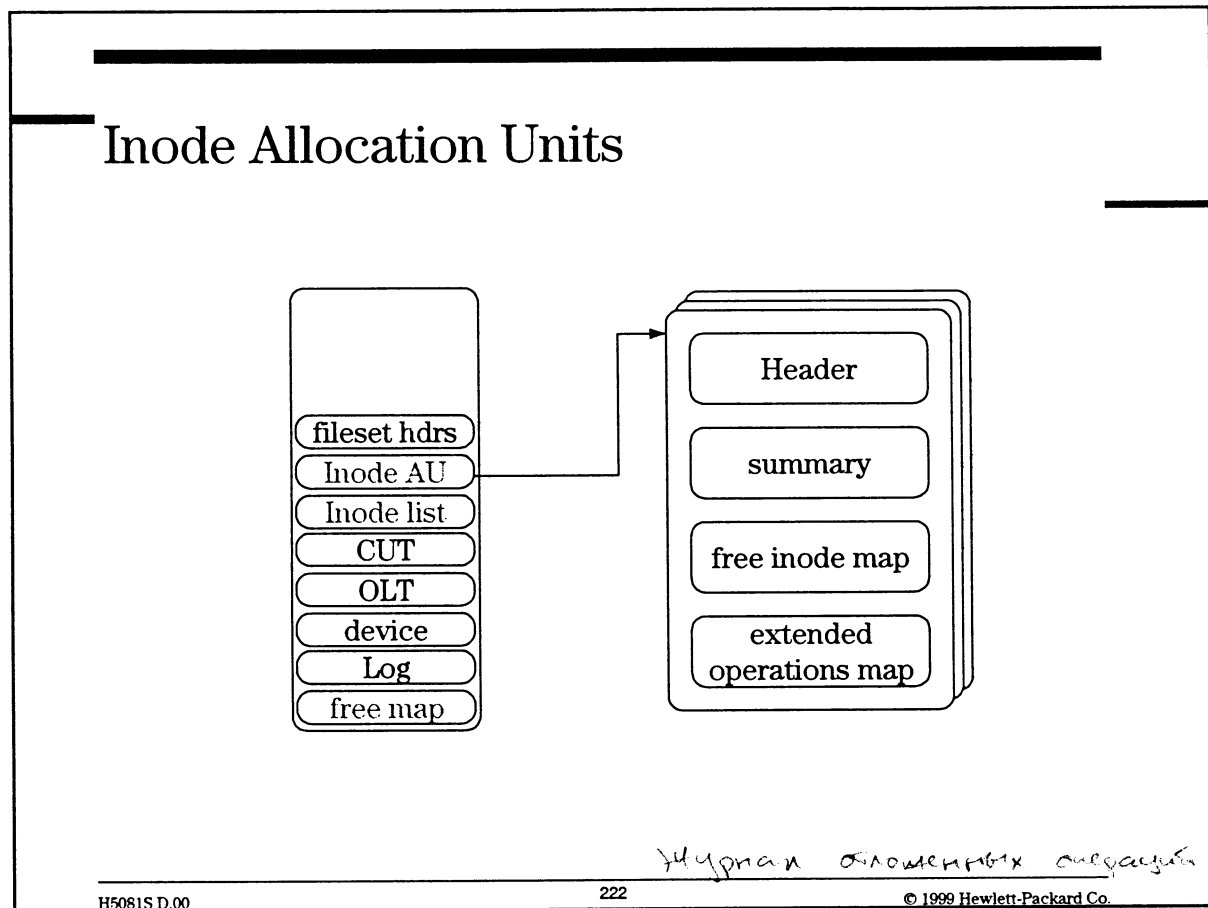
From the *OLT initial iext entry* the inodes from the *ATTRIBUTE* fileset can be read at block 16. The other OLT entries then describe the function of many of these inodes.

```
> 16b; p 6I      go to block 16 and print 6 inodes
inode structure at 0x00000010.0000
type FREE mode 0 nlink 0 uid 0 gid 0 size 0
atime 0 0 mtime 0 0 ctime 0 0
aflags 0 orgtype 0 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 0 version 0 0 iattrino 0 noverlay 0
inode structure at 0x00000010.0100 ①
type FREE mode 0 nlink 0 uid 0 gid 0 size 0
atime 0 0 mtime 0 0 ctime 0 0
aflags 0 orgtype 0 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 0 version 0 0 iattrino 0 noverlay 0
inode structure at 0x00000010.0200 ②
type FREE mode 0 nlink 0 uid 0 gid 0 size 0
atime 0 0 mtime 0 0 ctime 0 0
aflags 0 orgtype 0 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 0 version 0 0 iattrino 0 noverlay 0
inode structure at 0x00000010.0300 ③
type IFFSH mode 2000000777 nlink 1 uid 0 gid 0 size 2048
atime 894034476 378810 mtime 894034476 378810 ctime 894034476 378810
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 35
blocks 2 gen 0 version 0 0 iattrino 0 noverlay 0
de: 9 0 0 0 0 0 0 0 0 0
des: 2 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
inode structure at 0x00000011.0000 ④
type IFIAU mode 6000000777 nlink 1 uid 0 gid 0 size 4096
atime 894034476 378810 mtime 894034476 378810 ctime 894034476 378810
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 1 rdev/reserve/dotdot/matchino 0
blocks 4 gen 0 version 0 0 iattrino 0 noverlay 0
de: 11 0 0 0 0 0 0 0 0 0
des: 4 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
inode structure at 0x00000011.0100 ⑤
type IFILT mode 4000000777 nlink 1 uid 0 gid 0 size 32768
atime 894034476 378810 mtime 894034476 378810 ctime 894034476 378810
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 1 rdev/reserve/dotdot/matchino 37
blocks 32 gen 0 version 0 0 iattrino 0 noverlay 0
de: 16 1288 24 1296 0 0 0 0 0 0
des: 8 8 8 8 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
```

Inode 3 was the fileset headers inode we saw from the OLT. From this inode output we can see its type IFFSH and that it starts on block 9 of the disk and is 2 blocks long. Its one extent of length two, and the format of VxFS inodes are discussed later.

The two blocks are used to hold the two fileset headers.

8-19. SLIDE: Filesets and Headers



Student Notes

With the VxFS Version 2 disk layout, many structural elements of the file system are encapsulated in files to allow dynamic allocation of the file system structure. These files are called **structural** files. For a VxFS Version 3 disk layout, *all* structural elements are encapsulated in files. As the file system grows or is expanded, additional space can be allocated in the structural files as needed.

VxFS file systems are divided into two *file sets*, the **structural** fileset and the **unnamed** fileset. Each file set has a file set index number associated with it. The **structural** fileset is file set index #1, and the **unnamed** fileset is file set index #999.

The structural files are contained in the **structural** fileset. These files managed by the file system and are not viewable with standard file system commands. The **unnamed** fileset contains files that are visible and accessible by the user.

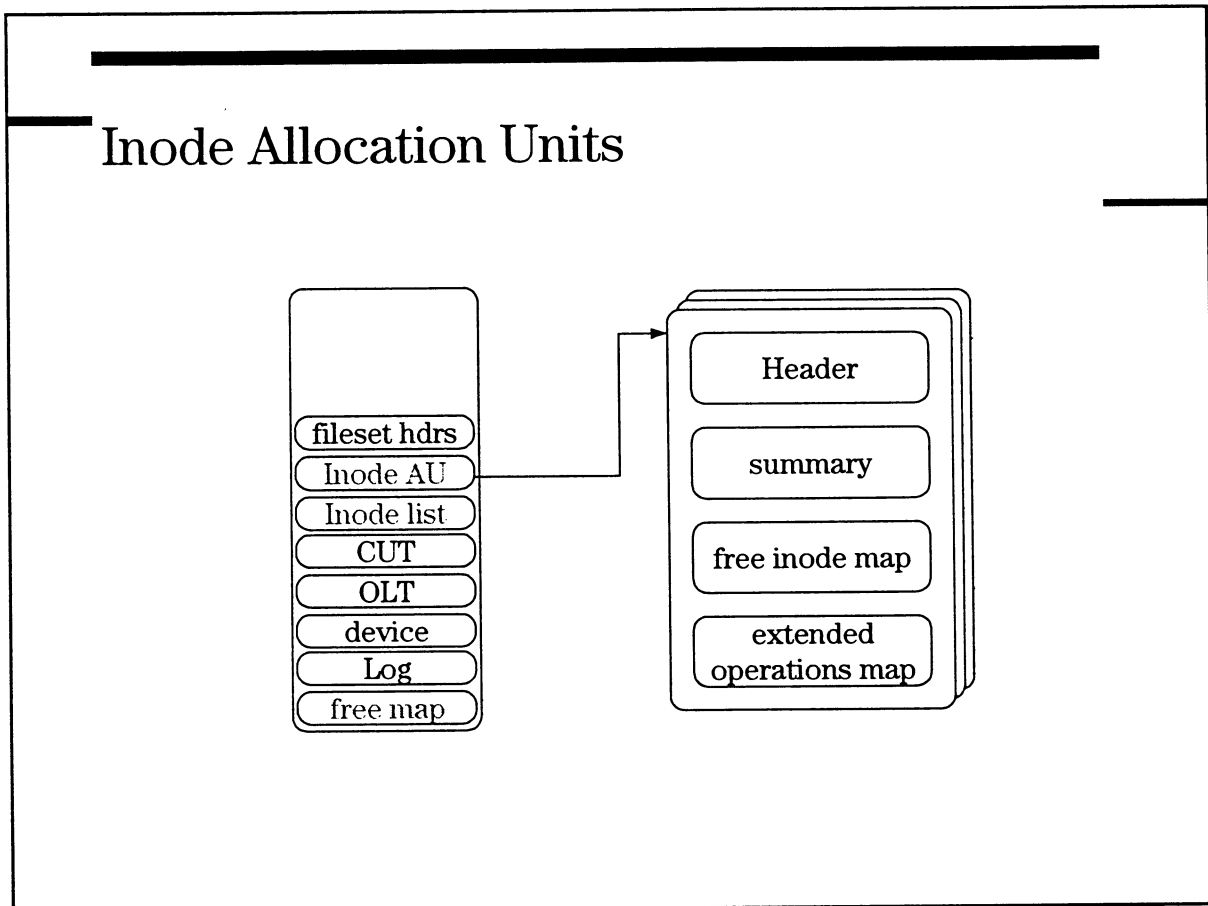

```
> 9b; p F          struct vx_fsethead
fset header structure at 0x00000009.0000
fsh_fsindex 1 fsh_volname "STRUCTURAL"
fsh_version 2 fsh_checksum 0x354faaa1 fsh_time Fri May 1 15:54:36 1998
fsh_ninode 128 fsh_nau 1 fsh_old_ilesize 0
fsh_fsextop 0x0 fsh_dflags 0xa fsh_quota 0 fsh_maxinode 4294967295
fsh_ilstino[5 37] fsh_iauino 4 fsh_lctino 0
fsh_uquotino 0 fsh_gquotino 0
fsh_attr_ninode 0 fsh_attr_nau 0
fsh_attr_ilstino[0 0] fsh_attr_iauino 0 fsh_attr_lctino 0
fsh_features 0 fsh_fsetid 0x0 00000000
clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000
fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000
fsh_backup 0x0 00000000 fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000
fsh_states 0x0 fsh_status ""
>
```

```
> 10b; p F        struct vx_fsethead
fset header structure at 0x0000000a.0000
fsh_fsindex 999 fsh_volname "UNNAMED"
fsh_version 2 fsh_checksum 0x35586acf fsh_time Fri May 1 15:59:13 1998
fsh_ninode 544 fsh_nau 1 fsh_old_ilesize 0
fsh_fsextop 0x0 fsh_dflags 0x1 fsh_quota 0 fsh_maxinode 4294967295
fsh_ilstino[65 97] fsh_iauino 64 fsh_lctino 0
fsh_uquotino 69 fsh_gquotino 0
fsh_attr_ninode 0 fsh_attr_nau 0
fsh_attr_ilstino[67 99] fsh_attr_iauino 66 fsh_attr_lctino 68
fsh_features 0 fsh_fsetid 0x0 00000001
clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000
fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000
fsh_backup 0x0 00000000 fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000
fsh_states 0x0 fsh_status ""
>
```

The fileset headers then give us the inodes that describe the files that hold the filesets own inodes (`fsh_ilstino`) and inode allocation unit (`fsh_iauino`).

Fileset #	Fileset name	Inode list (first)	Inode list (second)	Allocation unit
1	ATTRIBUTE	5	37	4
999	UNNAMED	65	97	64

8-20. SLIDE: Inode Allocation Units



Student Notes

Each fileset has an inode list and an inode allocation unit, which is its free list and extended operations map.

We will discuss the inode list for the fileset later, when we discuss the inodes themselves. VxFS has several formats of inodes available, as we shall see.

Having a variable sized inode list also forces the file system to use a variable sized free map and extended operations map for its inodes.

Having a circular log for recording the intended metadata updates, gives the file system an additional problem — how to handle operations that might require remaining pending over long periods of time. Details of why this should happen are given in the section on the intent log. Where it would be inappropriate to hold the inode update in the circular log, the update is performed on the inode and a map of inodes with these extended operations is retained. **fsck** can see from a flag in the inode allocation unit summary that there are extended inode operations pending in the fileset. There is no structure definition in the header files for this structure as it appears on the disk. Using the map, it can process the required inodes.

The inode list is held as a simple file, with space being allocated to it in much the same way as space would be allocated to any other file in the file system. The inode allocation unit however gets space allocated in fixed size pieces. The VxFS version 2 part of the superblock describes the size of these pieces, using the fields **iaumlen** and **iausize**. The pieces are typically 4 blocks (1K each).

The first block of each piece has a header structure **struct vx_iauhead**, followed by a summary block. There is no structure definition for how this appears on disk..

```

> 36b; p IA
> 64i
inode structure at 0x00000018.0000
type IFIAU mode 6000000777 nlink 1 uid 0 gid 0 size 4096
atime 894034476 378810 mtime 894034476 378810 ctime 900932787 320000
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 0
blocks 4 gen 0 version 0 20101 iattrino 0 noverlay 0
de:      36      0      0      0      0      0      0      0      0      0
des:      4      0      0      0      0      0      0      0      0      0
ie:       0      0      0
ies:      0
>

> 36b; p IA
iau header at 0x00000024.0000
magic a505fcf5 fsindx 999 aun 0
au_iextop 1 au_rifree 19 au_ribfree 2
>
from the header block
from the summary block
au_iextop is the flag to
indicate extend inode ops

```

These use a magic number for confirmation that this is the correct type of structure, and list the fileset number. If there is more than one inode allocation unit then the **aun** is the index number of this piece.

The summary information contains:

Au_iextop A flag that indicates that there are extended operations pending in the fileset.

Au_rifree The number of free inodes with respect to the number of allocated inodes specified in the filesets header field **fsn_inode**.

Au_ribfree Number of inode blocks free.

The third block is the free inode map. There is one bit for each allocated inode, a 0 indicating the inode is in use, 1 that it is free. Also for unallocated inodes a 0 is used.

Module 8
File Systems

```

> 38b ; p 32 xW                                     256 would be full block
00000026.0000: 00000000 00000000 00000000 00000000
00000026.0010: 00000000 00000000 00000000 00000000
00000026.0020: 00000000 00000000 00000000 00000000
00000026.0030: 00000000 00000000 00000000 00000000
00000026.0040: 0007ffff 00000000 00000000 00000000
00000026.0050: 00000000 00000000 00000000 00000000      unallocated inodes
00000026.0060: 00000000 00000000 00000000 00000000      fsh_ninode was 544
00000026.0070: 00000000 00000000 00000000 00000000
>

```

*OTAWA
one p.*

The last block in the piece is the extended inode operations map. It works like the freemap, only here a 1 indicates that this inode has some operations pending. This information is replicated in the inode itself.

```

> 39b ; p 32 xW
00000027.0000: 00000000 00000000 00000000 00000000
00000027.0010: 00000000 00000000 00000000 00000000
00000027.0020: 00000000 00000000 00000000 00000000
00000027.0030: 00000000 00000000 00000000 00000000
00000027.0040: 15500000 00000000 00000000 00000000
00000027.0050: 00000000 00000000 00000000 00000000
00000027.0060: 00000000 00000000 00000000 00000000
00000027.0070: 00000000 00000000 00000000 00000000
>

```

In this example

```

00000027.0040:      1      5      5      0
                   0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0
                   | | | |
inode             512 | | | |
                   513 | | | |
                   514 | | | |
                   515

```

Inodes 515, 517, 519, 521 & 523 have extended operations pending on them (they were unlinked while open).

One block of 1KB for the free map can map 8192 inodes. Once more than this number of inodes are allocated within the fileset a second piece will be added to the inode allocation unit.

```
> 10b; p F
fset header structure at 0x0000000a.0000
fsh_fsindex 999 fsh_volname "UNNAMED"
fsh_version 2 fsh_checksum 0x35b689c6 fsh_time Mon Jul 20 12:07:35 1998
fsh_ninode 10752 fsh_nau 2 fsh_old_ilesize 0
fsh_fsnext 0x0 fsh_dflags 0x1 fsh_quota 0 fsh_maxinode 4294967295
fsh_illistino[65 97] fsh_iauino 64 fsh_lctino 0
fsh_uquotino 69 fsh_gquotino 0
fsh_attr_ninode 0 fsh_attr_nau 0
fsh_attr_illistino[67 99] fsh_attr_iauino 66 fsh_attr_lctino 68
fsh_features 0 fsh_fsetid 0x0 00000001
clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000
fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000
fsh_backup 0x0 00000000 fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000
fsh_states 0x0 fsh_status ""
>

> 64i
inode structure at 0x00000018.0000
type IFIAU mode 6000000777 nlink 1 uid 0 gid 0 size 8192
atime 894034476 378810 mtime 894034476 378810 ctime 900932787 320000
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 0
blocks 8 gen 0 version 0 20101 iattrino 0 noverlay 0
de:    36 18276    0    0    0    0    0    0    0    0
des:   4 4    0    0    0    0    0    0    0    0
ie:    0 0    0
ies:   0
>
```

8-21. SLIDE: Free Space Management

Free Space Management

There are three parts to free space management

- the extent allocation unit state file
- the extent allocation unit summary file
- the extent allocation unit freemap file

Student Notes

Under version 2 VxFS, the start of the disk contained the unused boot block, the superblock, the primary copy of the OLT, the intent log and the redundant copy of the OLT. The remainder of the disk was then subdivided into allocation units. Each allocation unit had a 64KB header followed by a 32MB data area. This header kept the redundant superblock information and the extent free map for the allocation unit.

At Version 3 the layout has changed and now the whole disk is divided into allocation units, containing 32MB of data. The header information, rather than being in a fixed area at the start of the allocation unit, is now controlled via inodes in the *STRUCTURAL* fileset.

Since disk space is shared by the filesets within the file system, the free space maps are not a per-fileset resource, but a file system-wide resource.

There are three parts to the free space management structures. From the device configuration record referenced from the OLT we get:

```

> 1fset                                use fileset 1, this will display
the                                     the
fset header structure at 0x00000009.0000   fileset header, subsequent inode
references                                 references
fsh_fsindex 1  fsh_volname "STRUCTURAL"     will be from this fileset

-----8<-----

> 8i
inode structure at 0x00000012.0000
type IFDEV mode 2700000777 nlink 1 uid 0 gid 0 size 1024
atime 901023546 616661 mtime 901023546 616661 ctime 901023546 616661
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 40
blocks 1 gen 0 version 0 0 iattrino 0 noverlay 0
de:  34  0  0  0  0  0  0  0  0  0
des:  1  0  0  0  0  0  0  0  0  0
ie:   0  0  0
ies:  0

> 34b ; p DV
device configuration record at 0x00000022.0000
dc_magic 0xa514fcf5 dc_checksum 0xa5181d6e dc_flags 0x0
dc_devid 0 dc_len 204800
dc_stateino 32 dc_sumino 34 dc_mapino[10 42] dc_labeltype 2 dc_labelino 0
~

```

“dc_stateino”

The extent allocation unit (EAU) state file.

“dc_sumino”

The EAU summary file.

“dc_mapino”

The EAU freemap file. There are 2 inodes but they both reference the same disk blocks. Rebuilding the free map is a function that can be performed by **fsck**.

8-22. SLIDE: The Extent Allocation Unit State File

The Extent Allocation Unit State File

An allocation unit can be:

- all free
- all allocated in a single operation
- expanded, so that freemaps are needed
 - These freemaps can then be:
clean or dirty

Student Notes

The EAU state file has an entry for each allocation unit in the file system. The entries are just 2 bits each and have the **values**.

0	VX_EAU_EXPANDED	/* AU expanded, summary clean */
1	VX_EAU_DIRTY	/* AU expanded, summary dirty */
2	VX_EAU_ALLOCATED	/* AU not expanded, totally allocated */
3	VX_EAU_FREE	/* AU not expanded, totally free */

VX_EAU_EXPANDED indicates that the summary of this allocation unit has been set up and is up to date. VxFS does not set up all the structures for space that has not yet been needed.

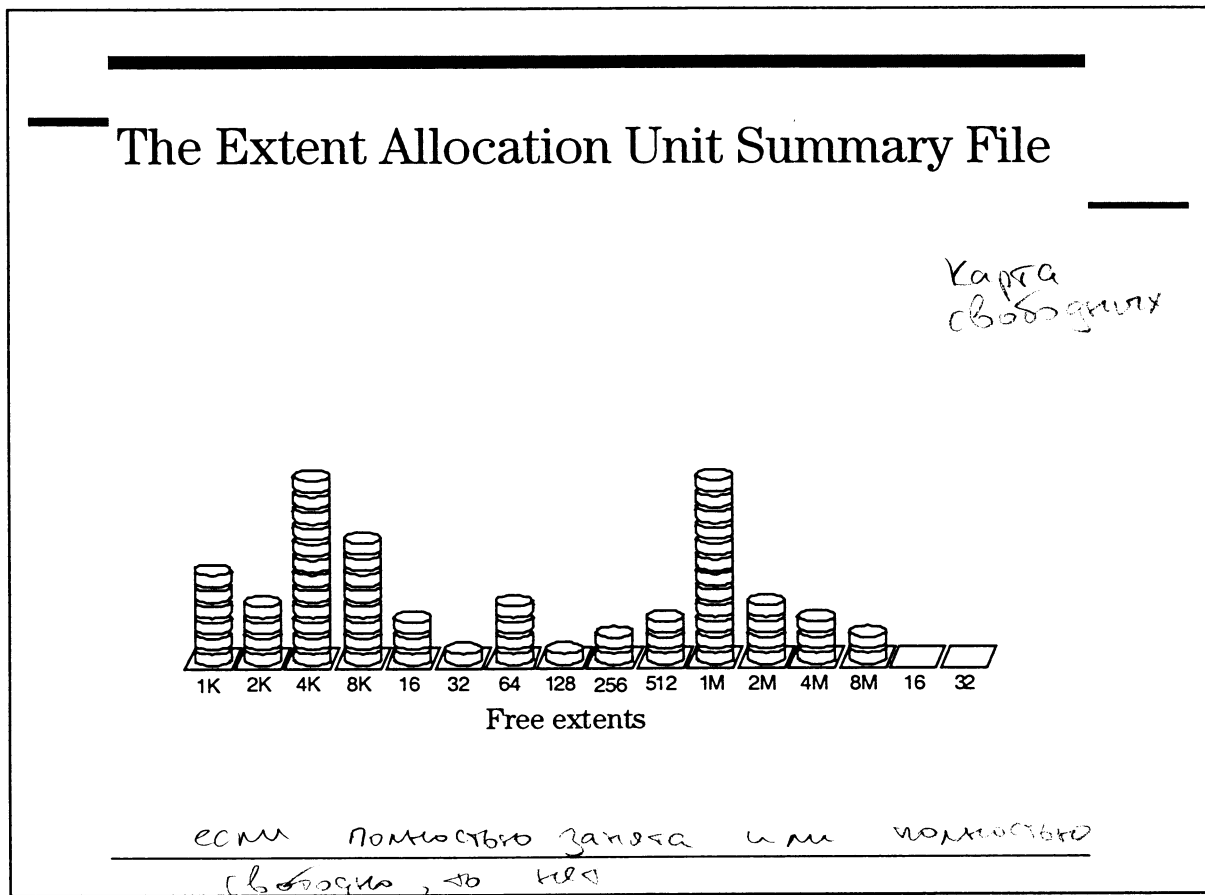
VX_EAU_DIRTY indicates that while the summary has been created it is currently out of date.

The VX_EAU_ALLOCATED value is used when the whole allocation unit has been allocated to a single file. Since the whole allocation unit is used as a single extent, no free space summary is required as no free space exists. While it is very common for large files to be given entire allocation units, this state value does not appear to be used unless **setext** is

used to reserve space for the file. Instead the `VX_EAU_EXPANDED` value is used and the summary shows that no space exists within the allocation unit.

`VX_EAU_FREE` indicates that no space has been apportioned from this allocation unit so the free space summary has not yet been built.

8-23. SLIDE: The Extent Allocation Unit Summary File



Student Notes

The EAU summary file again has one entry for each of the allocation units within the file system. Each allocation unit has one `vx_emapsum` structure — an array of 16 short integers that describe the number of free 1K, 2K, 4K, 8K ... 32M extents. The free space is always concatenated to give the largest piece possible. If a pair of 2K pieces are both free then they are considered to be 1 free 4K piece and not 2 free 2K pieces.

The EAU summary entries are only filled in if the state file entries indicate that the allocation unit has been expanded.

```

> 1 fset                                go into fileset one
fset header structure at 0x00000009.0000
fsh_fsindex 1  fsh_volname "STRUCTURAL"

-----8<-----                        not reproduced

> 32i
inode structure at 0x00000508.0000
type IFEAU mode 2500000777 nlink 1 uid 0 gid 0 size 1024
atime 901011000 678951 mtime 901011000 678951 ctime 901011000 678951
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 1 gen 0 version 0 0 iattrino 0 noverlay 0
de:  32 0 0 0 0 0 0 0 0 0
des:  1 0 0 0 0 0 0 0 0 0
ie:   0 0 0
ies:  0

> 32b ; p 8 xW
00000020.0000: 3c000000 00000000 00000000 00000000    0x3c == bin 00 11 11 00
00000020.0010: 00000000 00000000 00000000 00000000

> 34i
inode structure at 0x00000508.0200
type IFPAUS mode 2600000777 nlink 1 uid 0 gid 0 size 1024
atime 901011000 678951 mtime 901011000 678951 ctime 901011000 678951
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 1 gen 0 version 0 0 iattrino 0 noverlay 0
de:  35 0 0 0 0 0 0 0 0 0
des:  1 0 0 0 0 0 0 0 0 0
ie:   0 0 0
ies:  0

> 35b ; p 64 xH
          1K  2K  4K  8K  16K  32K  64K  128K
00000023.0000: 0001 0001 0001 0001 0000 0002 0001 0002    allocation unit 1
00000023.0010: 0000 0001 0000 0001 0001 0001 0001 0000
                256K 512K  1M  2M  4M  8M  16M  32M

00000023.0020: 0000 0000 0000 0000 0000 0000 0000 0000    allocation unit 2
00000023.0030: 0000 0000 0000 0000 0000 0000 0000 0000    (unexpanded)

00000023.0040: 0000 0000 0000 0000 0000 0000 0000 0000    allocation unit 3
00000023.0050: 0000 0000 0000 0000 0000 0000 0000 0000    (unexpanded)

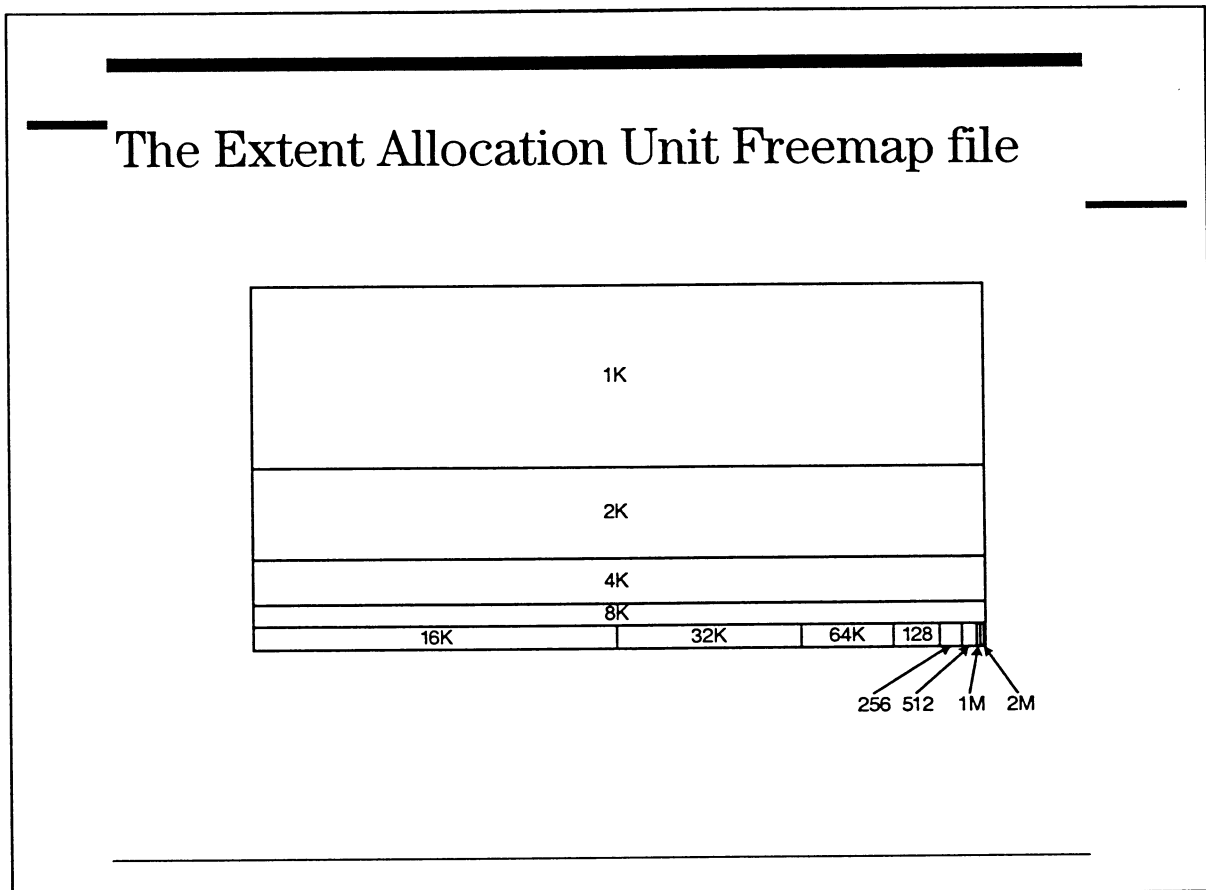
00000023.0060: 0000 0000 0000 0000 0000 0000 0000 0000    allocation unit 4
00000023.0070: 0000 0000 0000 0000 0001 0000 0000 0000

          1 2 5
        1 3 6 2 5 1          1 3
      1 2 4 8 6 2 6 8 6 2 1 2 4 8 6 2

efree 1 1 1 1 0 2 1 2 0 1 0 1 2 1 1 2    overall free space
                                           summary from the
                                           superblock

```

8-24. SLIDE: The Extent Allocation Unit Freemap File



Student Notes

The EAU freemap file divides the file system into 2MB pieces and has an entry for each of these. The individual pieces has a family of free maps for the single blocks and then for 2K, 4K 8K ... 2M extents. The freemaps, as with the summary files, concatenate free space but the 1K map is always used.

Each 2MB part of the disk has one free description structure which is 512 bytes long and is divided up like:

Range		Block	Bytes	Entries
Start	End	Size		(bits)
0x000	0x0ff	1K	256	2048
0x100	0x17f	2K	128	1024
0x180	0x1bf	4K	64	512
0x1c0	0x1df	8K	32	256
0x1e0	0x1ef	16K	16	128
0x1f0	0x1f7	32K	8	64
0x1f8	0x1fb	64K	4	32
0x1fc	0x1fd	128K	2	16
0x1fe	0x1fe	256K	1	8
0x1ff	Bits 0-3	512K	4bits	4
	4-5	1M	2bits	2
	6	2M	1bit	1
	7	Unused	1bit	1

* bits are counted in big endian order from most significant bit to the least significant.

The 1K block map is always used. The remaining maps have their space concatenated. If a pair of 2K extents are free, then neither will show as free in the 2K map. Instead an entry in the 4K map would show that the space was free.

Module 8
File Systems

```

> 1065b ; p 128 xW
00000429.0000: ffffffff ffffffff ffffffff ffffffff
00000429.0010: ffffffff ffffffff ffffffff ffffffff
00000429.0020: ffffffff ffffffff ffffffff ffffffff
00000429.0030: ffffffff ffffffff ffffffff ffffffff
00000429.0040: ffffffff ffffffff ffffffff ffffffff
00000429.0050: ffffffff ffffffff ffffffff ffffffff
00000429.0060: ffffffff ffffffff ffffffff ffffffff
00000429.0070: ffffffff ffffffff ffffffff ffffffff 1K blocks
00000429.0080: ffffffff ffffffff ffffffff ffffffff (all free)
00000429.0090: ffffffff ffffffff ffffffff ffffffff
00000429.00a0: ffffffff ffffffff ffffffff ffffffff
00000429.00b0: ffffffff ffffffff ffffffff ffffffff
00000429.00c0: ffffffff ffffffff ffffffff ffffffff
00000429.00d0: ffffffff ffffffff ffffffff ffffffff
00000429.00e0: ffffffff ffffffff ffffffff ffffffff
00000429.00f0: ffffffff ffffffff ffffffff ffffffff

00000429.0100: 00000000 00000000 00000000 00000000
00000429.0110: 00000000 00000000 00000000 00000000
00000429.0120: 00000000 00000000 00000000 00000000
00000429.0130: 00000000 00000000 00000000 00000000 2K extents
00000429.0140: 00000000 00000000 00000000 00000000 (none free)
00000429.0150: 00000000 00000000 00000000 00000000
00000429.0160: 00000000 00000000 00000000 00000000
00000429.0170: 00000000 00000000 00000000 00000000

00000429.0180: 00000000 00000000 00000000 00000000
00000429.0190: 00000000 00000000 00000000 00000000 4K extents
00000429.01a0: 00000000 00000000 00000000 00000000
00000429.01b0: 00000000 00000000 00000000 00000000

00000429.01c0: 00000000 00000000 00000000 00000000 8K extents
00000429.01d0: 00000000 00000000 00000000 00000000

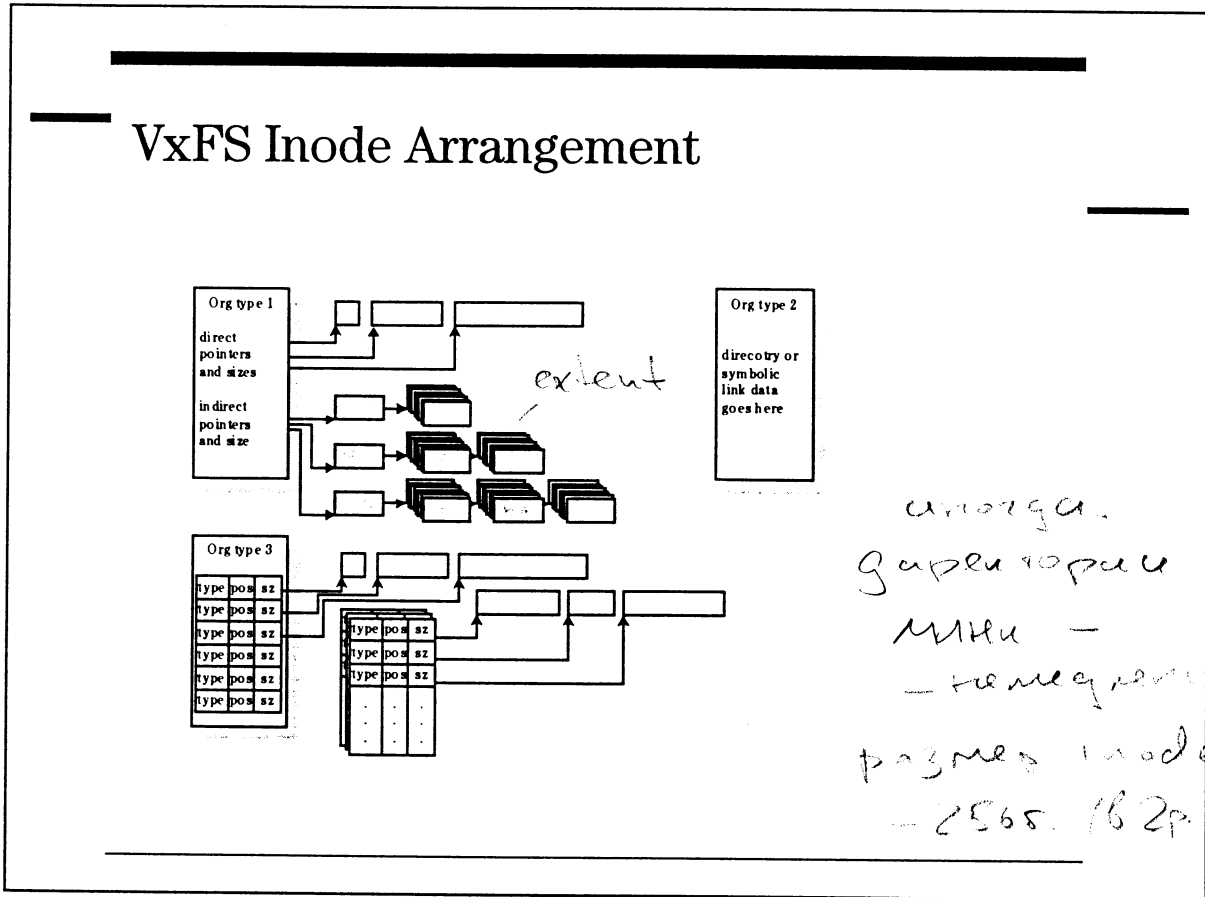
00000429.01e0: 00000000 00000000 00000000 00000000 16K extents

00000429.01f0: 00000000 00000000 00000000 00000002
                32K extents    64K      128K      ↓
                                256K
                                ↘
                                0000 00 1 0
                                512  1 2M

```

The address of the free map can be found from inode 10 or 42, as referenced in the device configuration entry.

8-25. SLIDE: VxFS Inode Arrangement



Student Notes

Each file, whether it is in the structural or unnamed fileset, has an inode associated with it. VxFS has several arrangements for its inodes.

Module 8
File Systems

The standard layout is similar to the UFS arrangement.

```
> 4i
inode structure at 0x00000461.0000
type IFREG mode 100666 nlink 1 uid 0 gid 3 size 104857600
atime 901023567 450001 mtime 901023579 900035 ctime 901023579 900035
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 102400 gen 0 version 0 12864 iattrino 0 noverlay 0
de: 1304 196608 32768 65536 0 0 0 0 0 0
des: 31464 8192 32768 29976 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
```

The inode has the file type, permissions, link count, ownership information and so on.

A key new field is the **orgtype** field. This is used to tell the kernel how the inode is arranged.

The **orgtype** of 1 indicates that this inode arrangement references data similar to the UFS arrangement, using direct and indirect pointers. There are 10 direct pointers but since these point to extents rather than single blocks, there is an extent-size value as well as the pointer to the data.

In this example the file is 100MB in size and is made out 4 extents,

The first extent starts on block 1304, and contains 31464 blocks.

The second extent starts at block 196608 and contains 8192 blocks.

The third extent starts at block 32768 and holds the entire 32MB second allocation unit

The last extent starts at block 65536 and holds 29976 blocks.

Should the file need more than 10 extents, then there are 3 indirect pointers. As with UFS these are for single, double and triple indirect pointers. All indirect extents are the same size and the **ies** field is the indirect extent size.

A second arrangement for inodes is *immediate*. There is room to store 96 bytes of information in an inode, so for small directories and symbolic links, rather than storing the information in a data block, it is stored directly inside the inode itself. This is the same way that fast symbolic links work for UFS.

The following is an example of a symbolic link to the directory `/usr/include/sys/fs`.

```

> 5i
inode structure at 0x00000461.0100
type IFLNK mode 120777 nlink 1 uid 0 gid 3 size 19
atime 901124121 720001 mtime 901124110 760001 ctime 901124110 760001
aflags 0 orgtype 2 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 5 version 0 30222 iattrino 0 noverlay 0
> im                move to the immediate part of the inode
00000461.0150: 00000000
> p 96c
00000461.0150: / u s r / i n c l u d e / s y s
00000461.0160: / f s 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000461.0170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000461.0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000461.0190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000461.01a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
>

```

orgtype 2 indicates an immediate inode arrangement and the *type* of IFLINK tells us there are up to 96 bytes worth of file named. Of course the size field tells us that in fact there are only 19 characters.

Module 8
File Systems

For a directory the usage of the inode is even more efficient.

```

> 6i
inode structure at 0x00000449.0200
type IFDIR mode 40777 nlink 2 uid 0 gid 3 size 96
atime 901124641 820000 mtime 894034722 910000 ctime 894034722 910000
aflags 0 orgtype 2 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 5
blocks 0 gen 0 version 0 40087 iattrino 0 noverlay 0
> im
00000449.0250: 01000000
> p db          print as a directory block
immediate directory block at 00000449.0250 - total free (d_tfree) 20
00000449.0254: d 0   d_ino 7 d_reclen 20 d_namlen 8
                d a t e . o u t
00000449.0268: d 1   d_ino 8 d_reclen 20 d_namlen 7
                o n e _ o n e
00000449.027c: d 2   d_ino 9 d_reclen 16 d_namlen 4
                l i s t
00000449.028c: d 3   d_ino 10 d_reclen 36 d_namlen 5
                f i l e s
>

```

VxFS directories do not store the entries “.” and “..” since it is known that all directories must have these. It is a waste to actual use disk space to store them.

“.” Points to itself, so any reference to “.” can be mapped to the current directory.
“..” Needs to know where the parent directory is, so the inode stores this information directory in the **dotdot** field.

The main directory entries themselves then store the

- Inode number.
- Record length.
- Filename length.
- The filename.
- Possibly some padding to bring it up to the next four byte boundary and to cope with deleted files leaving holes in the list of directory entries.

Inode number	Record length	Name length	Filename	Padding
4 bytes	2 bytes	2 bytes	Variable	Variable

This arrangement for directories is the same as that used by UFS.

The third organization for a VxFS inode is the *typed extent* arrangement.

```
> 10i
inode structure at 0x00000012.0200
type IFEMP mode 2400000777 nlink 1 uid 0 gid 0 size 57344
atime 901023546 616661 mtime 901023546 616661 ctime 901023546 616661
aflags 0 orgtype 3 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 42
blocks 56 gen 0 version 0 0 iattrino 0 noverlay 0
ext0: DATA boff: 0x00000000 bno: 1064 len: 8
ext1: DATA boff: 0x00000008 bno: 1072 len: 40
ext2: DATA boff: 0x00000030 bno: 1112 len: 8
ext3: NULL boff: 0x00000000 bno: 0 len: 0
ext4: NULL boff: 0x00000000 bno: 0 len: 0
ext5: NULL boff: 0x00000000 bno: 0 len: 0
>
```

one gap

Here only 6 extents can be described from the inode, but each entry has 4 fields

- The type.
- Position within the file of the start of this extent.
- The block number.
- The number of blocks within the extent.

This arrangement can be very useful to describe sparse files or situations where there are many extents and it is not desirable to use a fixed extent size after the tenth, which would be the limitation with the standard arrangement. Indirection is used to get around the problem of these *typed extent* inodes only having 6 entries.

Module 8
File Systems

```

> 4i
inode structure at 0x00000449.0000
type IFREG mode 100666 nlink 1 uid 0 gid 3 size 8232960
atime 894034841 730000 mtime 894034841 810028 ctime 894034841 810028
aflags 0 orgtype 3 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 8048 gen 1 version 0 6823 iattrino 0 noverlay 0
ext0:  INDIR  boff: 0x00000000 bno:    3880 len:     8
ext1:  NULL    boff: 0x00000000 bno:     0 len:     0
ext2:  NULL    boff: 0x00000000 bno:     0 len:     0
ext3:  NULL    boff: 0x00000000 bno:     0 len:     0
ext4:  NULL    boff: 0x00000000 bno:     0 len:     0
ext5:  NULL    boff: 0x00000000 bno:     0 len:     0
>

> 3880b ; p 64 eW      type                offset                block no
blocks
00000f28.0000:  33554432      0      1120      8
00000f28.0010:  33554432      8      1286      2
00000f28.0020:  33554432     10      1312      8
00000f28.0030:  33554432     18      1140      2
00000f28.0040:  33554432     20      1168     16
00000f28.0050:  33554432     36      1216     32
00000f28.0060:  33554432     68      1408     64
00000f28.0070:  33554432    132      1664    128
00000f28.0080:  33554432    260      2048    256
00000f28.0090:  33554432    516      3072    512
00000f28.00a0:  33554432   1028      4096   1024
00000f28.00b0:  33554432   2052      8192   2048
00000f28.00c0:  33554432   4100     12288   2048
00000f28.00d0:  33554432   6148     16384   1892
00000f28.00e0:      0          0          0          0
00000f28.00f0:      0          0          0          0
>

33554432 is 0x02000000

> 3880b; p 16 T
0x00000f28.0000:  DATA  boff: 0x00000000 bno:    1120 len:     8
0x00000f28.0010:  DATA  boff: 0x00000008 bno:    1286 len:     2
0x00000f28.0020:  DATA  boff: 0x0000000a bno:    1312 len:     8
0x00000f28.0030:  DATA  boff: 0x00000012 bno:    1140 len:     2
0x00000f28.0040:  DATA  boff: 0x00000014 bno:    1168 len:    16
0x00000f28.0050:  DATA  boff: 0x00000024 bno:    1216 len:    32
0x00000f28.0060:  DATA  boff: 0x00000044 bno:    1408 len:    64
0x00000f28.0070:  DATA  boff: 0x00000084 bno:    1664 len:   128
0x00000f28.0080:  DATA  boff: 0x00000104 bno:    2048 len:   256
0x00000f28.0090:  DATA  boff: 0x00000204 bno:    3072 len:   512
0x00000f28.00a0:  DATA  boff: 0x00000404 bno:    4096 len:  1024
0x00000f28.00b0:  DATA  boff: 0x00000804 bno:    8192 len:  2048
0x00000f28.00c0:  DATA  boff: 0x00001004 bno:   12288 len:  2048
0x00000f28.00d0:  DATA  boff: 0x00001804 bno:   16384 len:  1892
0x00000f28.00e0:  NULL   boff: 0x00000000 bno:     0 len:     0
0x00000f28.00f0:  NULL   boff: 0x00000000 bno:     0 len:     0

```

Here the type in the inode is *INDIR* and the reference is to an 8 block extent for indirect typed extents.

The **type** column in the above example looks a little strange in decimal, in fact it just uses the high byte. Potentially the remaining 3 bytes can be combined with the offset fields 4 bytes to give a 56bit offset (65536 TB file offset). Of course **fsdb** can display in typed extent format too.

In both the previous examples the file is in fact wholly allocated. The second example uses more extents than could be flexibly handled using the conventional inode layout.

For a sparse file this layout is even more useful. Here a file has a few bytes written at offsets of 0, 1K, 1M, 10M, 100M and 1G.

```
> 17i
inode structure at 0x00000464.0100
type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1073741841
atime 901186104 330013 mtime 901186104 420000 ctime 901186104 420000
aflags 0 orgtype 3 eopflags 8 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 41 gen 0 version 0 50255 iattrino 0 noverlay 0
ext0: DATA boff: 0x00000000 bno: 1177 len: 9
ext1: DATA boff: 0x00000400 bno: 1144 len: 8
ext2: DATA boff: 0x00002800 bno: 1192 len: 8
ext3: DATA boff: 0x00019000 bno: 1200 len: 8
ext4: DATA boff: 0x00100000 bno: 1208 len: 8
ext5: NULL boff: 0x00000000 bno: 0 len: 0
>
```

In UFS this can be achieved by issuing blocks at the required locations and leaving all the other pointers NULL. The first and second piece of data would appear in the first block, with NULLs between. The third and fourth would each require a single indirect block and the last would use a double indirect block.

For VxFS V2 where the typed extent inode format was not available, the variable sized extent system proves problematic. Either all the blocks would have to be allocated, or indirection must be used. However the file does not know where the indirection area starts unless the size of the first 10 extents have previously been selected. This means in effect that VxFS must give up the flexibility and performance advantages of extent-based allocation for sparse files. Files that are created sparse and then back filled show a marked performance difference at HP-UX 10.10 from those either created normally or pre-allocated.

The typed extent layout available with VxFS 3 gets around this problem.

There are a number of different extent types listed in the header file `vx_inode.h`:

```
#define VX_TYPE_IADDR_4 1 /* indirect address extent */
#define VX_TYPE_DATA_4 2 /* data extent */
#define VX_TYPE_IADDR_DEV4 3 /* indirect addr extent with altdev */
#define VX_TYPE_DATA_DEV4 4 /* data extent with altdev */
#define VX_TYPE_PLACEHOLD 5 /* placeholder for empty indirects */

Also the value 0 is NULL to indicate this entry is unused.
```

The values that appear to be used on HP-UX are 0, 1 & 2. HP-UX does not support having a file spread over multiple devices (Logical Volumes). The types 3 and 4 are for cases where the operating system does allow for this. Here the typed extents need not only to specify the disk block and extent length, but also device. The header file defines two data structures for

Module 8
File Systems

use as typed extents **vx_typed_4**, which is the type we have seen, and **vx_typed_dev4** for the multiple-device case.

The overall layout of the inode on disk is described by the **vx_icommon** structure. As we have seen there are different possible layouts for the inodes. When reading an inode, the inode **type** information from the **ic_mode** field and the **ic_orgtype** field tell us how the inode is being used.

```

struct vx_icommon {          /* 0x190 disk inode */
    vx_u32_t    ic_mode;      /* 0x00: mode and type */
    vx_u32_t    ic_nlink;     /* 0x04: number of links */
    vx_u32_t    ic_uid;       /* 0x08: owner's user id */
    vx_u32_t    ic_gid;       /* 0x0c: owner's group id */
    vx_hyper_t  ic_size;      /* 0x10: "disk" num of bytes */
    vx_timeval32_t ic_atime;   /* 0x18: time last accessed */
    vx_timeval32_t ic_mtime;   /* 0x20: time last modified */
    vx_timeval32_t ic_ctime;   /* 0x28: time ino last change */
    vx_u8_t     ic_aflags;     /* 0x30: allocation flags */
    vx_u8_t     ic_orgtype;    /* 0x31: org type */
    vx_u16_t    ic_eopflags;   /* 0x32: extended operations */
    vx_u32_t    ic_eopdata;    /* 0x34: extended operations */

1      union vx_ftarea {      /* 0x38: type specific data */
1.1      struct vx_ftarea_dev { /* device files */
            vx_u32_t    ic_rdev; /* 0x38: device number */
        } ic_ftarea_dev;
1.2      struct vx_ftarea_dir { /* directories */
            vx_ino32_t  ic_dotdot; /* 0x38: parent directory */
        } ic_ftarea_dir;
1.3      struct vx_ftarea_reg { /* regular files */
            vx_blkcnt32_t ic_reserve; /* 0x38: prealloc space */
            vx_blkcnt32_t ic_fixextsize; /* 0x3c: fixed ext size */
        } ic_ftarea_reg;
1.4      struct vx_ftarea_vxspecc { /* vxfs private files */
            vx_ino32_t  ic_matchino; /* 0x38: matching inode */
            vx_u32_t    ic_fsetindex; /* 0x3c: fileset index */
        } ic_ftarea_vxspecc;
    } ic_ftarea;

    vx_blkcnt32_t ic_blocks; /* 0x40: blocks actually held */
    vx_u32_t    ic_gen;       /* 0x44: generation number */
    vx_hyper_t  ic_vversion; /* 0x48: inode serial number */

2      union vx_org {        /* 0x50: */
2.1      struct vx_immed {
            char        ic_immed[NIMMED_N]; /* 0x50 immediate */
        } ic_vx_immed;
2.2      struct vx_ext4 {
            vx_blkcnt32_t ic_ies; /* 0x50: ind ext sz */
            vx_daddr32_t  ic_ie[NIADDR]; /* 0x54: indir exts */
            struct vx_dext { /* 0x60: dir exts */
                vx_daddr32_t  ic_de; /* dir ext */
                vx_blkcnt32_t  ic_des; /* dir ext size */
            } ic_dext[NDADDR_N];
        } ic_vx_e4;
2.3      struct vx_typed_4    ic_td4[VX_NTYP4]; /* typed exts */
2.4      struct vx_typed_dev4 ic_td8[VX_NTYP8]; /* typed exts */
    } ic_org;
    vx_ino32_t  ic_iattrino; /* 0xb0: indirect attribute ino */
    vx_u32_t    ic_noverlay; /* 0xb4: overlay count */
} i_ic;

```

ic_mode	<p>As with the UFS inode, the VxFS one starts with the type and permissions combined into a single field. The low order 12 bits give the permissions. The remainder are then available for defining the <u>type</u> of the inode.</p> <p>VxFS has many more types than UFS, in addition to supporting the standard UNIX ones: -</p> <table border="0"> <tr><td>IFIFO</td><td>0x00001000</td><td>named pipe (fifo)</td></tr> <tr><td>IFCHR</td><td>0x00002000</td><td>character special</td></tr> <tr><td>IFDIR</td><td>0x00004000</td><td>directory</td></tr> <tr><td>IFBLK</td><td>0x00006000</td><td>block special</td></tr> <tr><td>IFREG</td><td>0x00008000</td><td>regular</td></tr> <tr><td>IFLNK</td><td>0x0000a000</td><td>symbolic link</td></tr> <tr><td>IFSOC</td><td>0x0000c000</td><td>socket</td></tr> </table> <p>There are also the internal types used in the structural filesset: -</p> <table border="0"> <tr><td>IFFSH</td><td>0x10000000</td><td>file set header</td></tr> <tr><td>IFILT</td><td>0x20000000</td><td>inode list</td></tr> <tr><td>IFIAU</td><td>0x30000000</td><td>inode allocation unit</td></tr> <tr><td>IFCUT</td><td>0x40000000</td><td>current usage table</td></tr> <tr><td>IFATT</td><td>0x50000000</td><td>attribute inode</td></tr> <tr><td>IFLCT</td><td>0x60000000</td><td>lint count table</td></tr> <tr><td>IFIAT</td><td>0x70000000</td><td>indirect attribute file</td></tr> <tr><td>IFEMR</td><td>0x80000000</td><td>extent map reorg file</td></tr> <tr><td>IFQUO</td><td>0x90000000</td><td>bsd quota file</td></tr> <tr><td>IFLAB</td><td>0xa0000000</td><td>device label file</td></tr> <tr><td>IFOLT</td><td>0xb0000000</td><td>object location table file</td></tr> <tr><td>IFLOG</td><td>0xc0000000</td><td>log file</td></tr> <tr><td>IFEMP</td><td>0xd0000000</td><td>extent map file</td></tr> <tr><td>IFEAU</td><td>0xe0000000</td><td>extent allocation unit file</td></tr> <tr><td>IF AUS</td><td>0xf0000000</td><td>extent au summary file</td></tr> <tr><td>IFDEV</td><td>0x17000000</td><td>device configuration file</td></tr> </table> <p>That we discussed earlier.</p>	IFIFO	0x00001000	named pipe (fifo)	IFCHR	0x00002000	character special	IFDIR	0x00004000	directory	IFBLK	0x00006000	block special	IFREG	0x00008000	regular	IFLNK	0x0000a000	symbolic link	IFSOC	0x0000c000	socket	IFFSH	0x10000000	file set header	IFILT	0x20000000	inode list	IFIAU	0x30000000	inode allocation unit	IFCUT	0x40000000	current usage table	IFATT	0x50000000	attribute inode	IFLCT	0x60000000	lint count table	IFIAT	0x70000000	indirect attribute file	IFEMR	0x80000000	extent map reorg file	IFQUO	0x90000000	bsd quota file	IFLAB	0xa0000000	device label file	IFOLT	0xb0000000	object location table file	IFLOG	0xc0000000	log file	IFEMP	0xd0000000	extent map file	IFEAU	0xe0000000	extent allocation unit file	IF AUS	0xf0000000	extent au summary file	IFDEV	0x17000000	device configuration file
IFIFO	0x00001000	named pipe (fifo)																																																																				
IFCHR	0x00002000	character special																																																																				
IFDIR	0x00004000	directory																																																																				
IFBLK	0x00006000	block special																																																																				
IFREG	0x00008000	regular																																																																				
IFLNK	0x0000a000	symbolic link																																																																				
IFSOC	0x0000c000	socket																																																																				
IFFSH	0x10000000	file set header																																																																				
IFILT	0x20000000	inode list																																																																				
IFIAU	0x30000000	inode allocation unit																																																																				
IFCUT	0x40000000	current usage table																																																																				
IFATT	0x50000000	attribute inode																																																																				
IFLCT	0x60000000	lint count table																																																																				
IFIAT	0x70000000	indirect attribute file																																																																				
IFEMR	0x80000000	extent map reorg file																																																																				
IFQUO	0x90000000	bsd quota file																																																																				
IFLAB	0xa0000000	device label file																																																																				
IFOLT	0xb0000000	object location table file																																																																				
IFLOG	0xc0000000	log file																																																																				
IFEMP	0xd0000000	extent map file																																																																				
IFEAU	0xe0000000	extent allocation unit file																																																																				
IF AUS	0xf0000000	extent au summary file																																																																				
IFDEV	0x17000000	device configuration file																																																																				
ic_size	<p>This is the offset within the file of the last byte.</p> <p>Although it is labelled as the size, this is misleading since sparse files don't have that many blocks associated with them, and on VxFS space can be pre-allocated to files using the Online JFS setext command.</p>																																																																					
ic_aflags	<p>Certain <i>allocation flags</i> can be set for a file through the use of Online JFS setext command. This allows programmers for instance to pre-create a file and then not allow the file to grow when writing to the end.</p>																																																																					
ic_orgtype	<p>This field identifies the layout of the main part of the inode:</p> <table border="0"> <tr><td>IORG_NONE</td><td>0</td><td>inode has no format</td></tr> <tr><td>IORG_EXT4</td><td>1</td><td>inode has 4 byte data block addrs</td></tr> <tr><td>IORG_IMMED</td><td>2</td><td>data stored in inode</td></tr> <tr><td>IORG_TYPED</td><td>3</td><td>inode has typed extent descriptors</td></tr> </table> <p>The preceding discussion looked at these different formats.</p>	IORG_NONE	0	inode has no format	IORG_EXT4	1	inode has 4 byte data block addrs	IORG_IMMED	2	data stored in inode	IORG_TYPED	3	inode has typed extent descriptors																																																									
IORG_NONE	0	inode has no format																																																																				
IORG_EXT4	1	inode has 4 byte data block addrs																																																																				
IORG_IMMED	2	data stored in inode																																																																				
IORG_TYPED	3	inode has typed extent descriptors																																																																				
ic_eopflags	<p>Extended inode opts flag. (see later section on exetend inode ops)</p>																																																																					
ic_eopdata	<p>Extended inode opts data. (see later section on exetend inode ops)</p>																																																																					


vx_ftarea	This section of the inode is used differently by different types of inodes: -	
	Device files	The device number, the major and minor numbers combined.
	Directories	All directories in UNIX have the entries "." and ".." VxFS does not store these entries, since they are known to exist they can be invented when accessed. The "." entry references itself so no further information is needed to invent this. For the ".." entry however the inode number of the parent directory is needed. This inode entry is stored here.
	Regular files	The setext command or ioctl can be used to specify a fixed extent size for a file and also an amount of space to pre-allocate to the file. This space is not initialized! Using this technique extents of greater than one allocation unit can be created.
	VxFS internal types	Where inodes in structural filesets describe information pertaining to an individual fileset, the ic_fsetindex field says which one. Many of these structural inodes are replicated, and the ic_matchino field gives the inode number of the other copy.
ic_blocks	This field is the real <i>size</i> of the file. It details the number of file system blocks the file contains. This field is reported by commands like ls -s and du . These commands normalize the value into SYS V blocks of 512 bytes.	
ic_gen	The inode generation indicates the number of times this inode has been used to map a file. When a file is removed the generation number is incremented. This field is used to form part of the NFS file handle. Modifying the data held in this field helps to give NFS a little more in the way of security. Where the file system does not update this field two problems arise.	
	1	If a file is removed and then a new file then uses the same inode, any NFS client accessing the old file, would suddenly find itself accessing the new one. This is a security violation and can lead to file corruption.
	2	If no generation numbers are used then the NFS file handles can easily be guessed, thus allowing programs to <i>spoof</i> them, and gain access to files without having the correct file permission or even having to mount the file system!

Module 8
File Systems

* vx_org	This is the main part of the inode, the format here depends upon the vx_orgtype value.
IORG_IMMED	For small directories and symbolic link the data is store immediately inside the inode
IORG_EXT4	The standard arrangement with 10 pairs of block pointers and extent size values for the start of the file. Then indirection pointers and an indirect extent size value.
IORG_TYPED	Typed extent records, these could either be in the format vx_typed_4 or vx_typed_dev4 . Both record structures start with a header field from which the correct type can be extracted.

8-26. SLIDE: The Intent Log

The Intent Log



- Before VxFS changes meta data structures it writes the changes to the intent log.
- Once the work has been done, VxFS writes that fact also.

Student Notes

Probably the single biggest problem with the UFS file system is that, in the event of an improper shutdown, a highly time-consuming repair operation is needed. This **fscking** is often blamed on the existence of the buffer cache. This is not really fair. Caching certainly does not help the situation, but **fsck** would still be required even if all writes were to be performed synchronously to the disk.

Module 8
File Systems

The UFS file system consists of 4 major sets of data structures.

- The superblock
- The cylinder group information, which holds the free maps
- The inodes
- The directories

All of these data structures as well as being self consistent must agree with each other.

As an example of the type of problem that can occur when more than one operation is required to be completed, let us look at a problem from the distant past

*In very early UNIX there was no **mkdir** system call. Directories were formed by using **mknode(2)** to make the directory file itself and then **link(2)** was used to set-up the "." and ".." entries. This could lead to a problem that if the **mkdir** command were killed halfway through it could leave a directory with no "." and/or no "..".*

During an improper shutdown similar things could happen since it is not possible for all four of these data structures to be updated simultaneously. For instance, creating a new file one needs to

- Add a directory entry.
- Allocate and populate an inode.
- Remove the inode from the free list.
- Reduce the number of free inodes in the summary field of the superblock.

This is the minimum that needs to happen to create a new file, and it affects all four of the major file system structures. Since not all four can be updated at the same time, if there were to be a system crash or power failure halfway through, then the data structures would not be consistent.

For UFS there is no record of what activities are being performed on the file system. So when recovering from an improper shutdown **fsck** has no choice but to read every single inode, and directory to build up a picture of what the file system should look like and then compare this picture back against the freemaps and superblock summaries. As we know, this is very time consuming.

With VxFS, an intent log is normally kept. Before starting a file system update such as creating a new file. The intended metadata changes are logged. After the changes have happened successfully a record is made of this successful completion.

If a problem were to arise, then **fsck** could use this log to perform the required repair work.

The intent log entries provide not only information about which structures need to be repaired, but also, what data they are now supposed to contain. Since frequently more than one structure will need to be modified as part of a file system operation, each transaction is split into sub-functions. Also the log is made up of fixed size pieces and no entry is allowed to cross the boundary of one of these pieces so sub-functions can be divided into sections.

All intent log entries start with a **header** that contains:

- a log id number of the transaction
- a function number
- the position of this sub-function within the transaction
- the number of sub-functions in the whole transaction
- the piece number
- the number of pieces in the sub-function.
- the length of the entry

The header is followed by details of the changes to be made. The format of these changes varies depending upon the type of sub-function. In the following example an inode is being updated

```

00002000: id 23  func 1  ser 0  lser 5  len 295
Inode Modification  fset 999  ilist 0  dev/bno 0/1120  ino 2  osize 295
New Inode Contents:
type IFDIR mode 40755  nlink 3  uid 0  gid 0  size 1024
atime 901209387 286262  mtime 901209388 440000  ctime 901209388 440000
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0  rdev/reserve/dotdot/matchino 2
blocks 1  gen 0  version 0 13  iattrino 0 noverlay 0
de: 1285  0  0  0  0  0  0  0  0  0
des:  1  0  0  0  0  0  0  0  0  0
ie:   0  0  0
ies:  0

```

Here the overall transaction id is 23, and the operation is an inode update, which is **sub-function type VX_IMTRAN (1)**. This whole transaction has 6 stages, numbered 0 through to 5, and this is the first, 0. Unfortunately when **fsdb** prints out information from the log file it does not show the individual pieces in the sub-function, as it reads the whole record and shows it as a single more readable entity.

Module 8
File Systems

A later sub-function from this transaction uses multiple pieces and these can be viewed if need be by displaying the raw data from the intent log.

```
00002350: id 23  func 3  ser 5  lser 5  len 1052
Directory entry reorg  fset 999  ilist 0  inode 2  bno 1285  oldblen 0
newblen 1024
```

Here the directory is being updated from using the immediate format to using the external format. The **fmtlog** command within **fsdb** does not show what is really happening here. The length it reports however is 1052, and the pieces within the intent log are 512 bytes.

If the data from the intent log were dumped out raw then it is possible to see the individual pieces.

```
> 40b ; + 0x2350 B ; p 300X
      23   3 5 5  0 2 164          23 is the log id
00000030.0350: 00000017 00030505 000200a4 000003e7      3 is the sub function type
00000030.0360: 00000000 00000002 00000000 00000505      5 sub func serial number
00000030.0370: 00000000 00000400 03580020 00000000      of 5 in transaction
00000030.0380: 00000000 00000000 00000000 00000044      0`th part
00000030.0390: 00000000 00000000 00000058 00680078      of 2
00000030.03a0: 00880098 00000000 00000000 00000000      part length 164
00000030.03b0: 00000000 00000000 00000000 00000003

      23   3 5 5  1 2 500          23 is the log id
00000031.0000: 00000017 00030505 010201f4 00000007      3 is the sub function type
00000031.0010: 00100006 00006669 6c652e64 00000008      5 sub func serial number
00000031.0020: 03680006 00006669 6c652e65 00000000      of 5 in transaction
00000031.0030: 00000000 00000000 00000000 00000000      1st part
00000031.0040: 00000000 00000000 00000000 00000000      of 2
00000031.0050: 00000000 00000000 00000000 00000000      part length 500
00000031.0060: 00000000 00000000 00000000 00000000
00000031.0070: 00000000 00000000 00000000 00000000
00000031.0080: 00000000 00000000 00000000 00000000

      23   3 5 5  2 2 388          as above, only this is pt 2 of 2
00000031.0200: 00000017 00030505 02020184 00000000      part length 388
00000031.0210: 00000000 00000000 00000000 00000000
00000031.0220: 00000000 00000000 00000000 00000000
00000031.0230: 00000000 00000000 00000000 00000000
```

The actual data has been cut down as it is not really relevant to this discussion.

Adding up the three length values (164+500+388) gives the total size of 1052 reported by **fsdb** originally.

8-27. SLIDE: Logging Levels

Logging Levels

When to write log entries?

- Always log before modifying the metadata.
- Before ever returning to userland? — *ончун монтураб.*
- Before returning to userland for sensitive updates?
- At some future time?
- Or just don't bother.

Student Notes

No logging (`nolog`)

No attempt at maintaining consistency is made. This mode is used for file systems that contain purely transitory data that should not persist across system failure. An example is a file system used for swap files. In this mode VxFS does not use the intent log.

This mode can also be useful when restoring backups into clean file system from high-speed tape devices. If there were to be a system failure during the recovery it would be normal to clear the file system and restart the recovery. VxFS allows file systems to be remounted if it is desirable to change mount options.

Temporary (`templog`)

The file system is kept in a consistent state, but no operations are guaranteed to persist across a reboot. This mode is useful for the type of data stored in the UNIX `/tmp` file system. The data is of a transitory nature, but it is necessary for the file system to persist across reboots, so that files such as editor temporary files can be recovered in case of a crash. In this mode VxFS uses the intent log to record pending changes, but does not ensure that any operations are physically logged before returning control to the calling application.

Workstation (delaylog)

In this mode some but not all structural operations are guaranteed to persist after a reboot. This mode matches the persistence model of most UFS file systems. In particular, file removal and rename operations are guaranteed to persist across a system failure once the system call returns control to the calling application. Applications from a BSD UNIX environment frequently rely on this feature of UFS to function robustly. In this mode VxFS uses the intent log to record pending changes to the file system, and also ensures that at least file removal and rename operations are physically logged before returning control to the caller.

This mode is similar to using the `fs_async` kernel parameter set to 1, on UFS file systems, except of course that the intent log is available to provide high speed, accurate file system repairs.

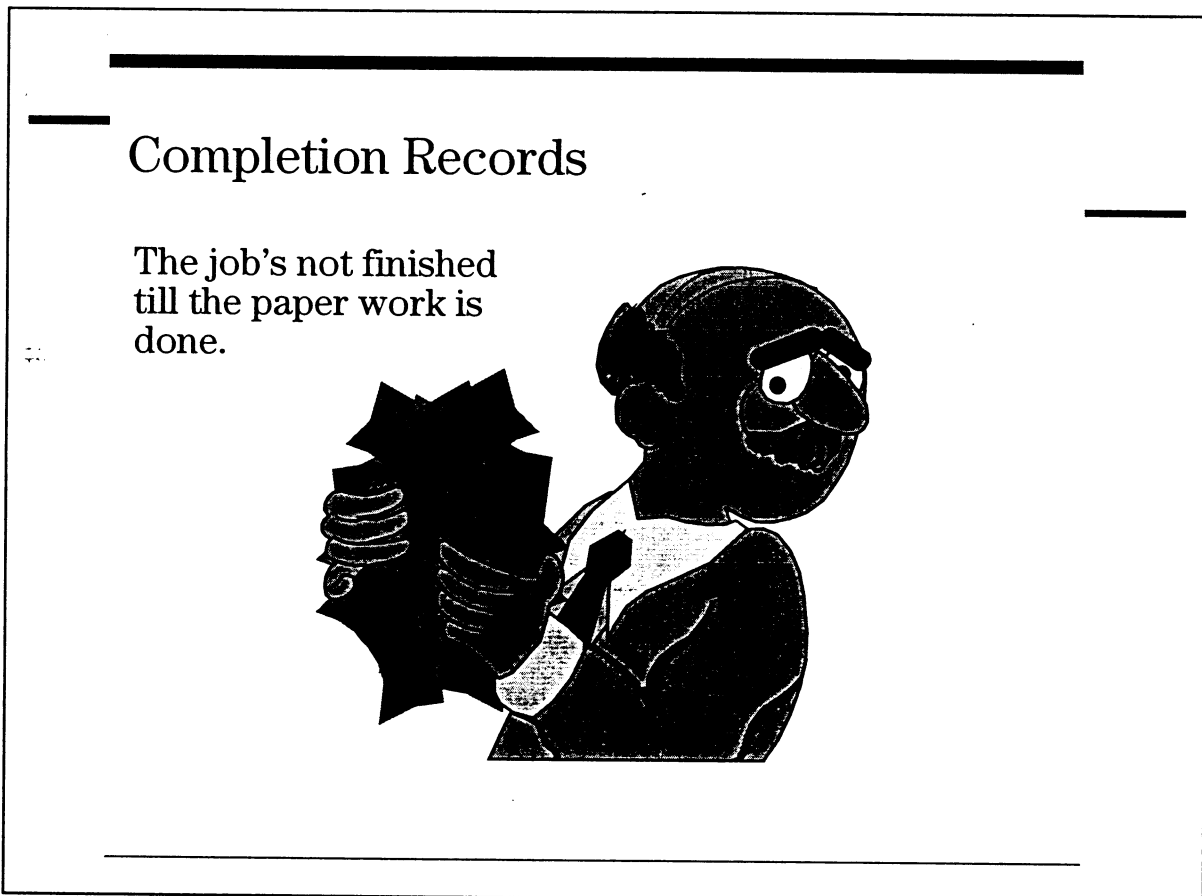
Server (log)

In this mode all structural operations are guaranteed to persist once control is returned to the calling application. VxFS ensures that all structural operations are physically logged in the intent log before returning control to the caller.

This mode is similar to using the `fs_async` kernel parameter set to 0, on UFS file systems.

*ecrb u gpyne basarwle Gpyne
(afine, etc.)*

8-28. SLIDE: Completion Records



Student Notes

Once a transaction has been successfully written to disk then a completion record can be added to the log file.

The completion records contain a **group done transid** field, which indicates which transactions have now been completed. When a completion record is written it indicates that all transactions up to and including the one whose ID matches the done ID have now been completed..

These completion records are a transaction in their own right, and therefore have their own transaction ID number.

VxFS writes 2 types of completion records. In the most common type the done ID is 0. These records do not carry out any real function for the file system. The second types are the real ones where the ID is that of an earlier transaction.

Module 8
File Systems

```
000029a0: id 26  func 103  ser 0  lser 0  len 84
Group Done tranid: 0
  covering 96 bytes

00002a00: id 27  func 103  ser 0  lser 0  len 500
Group Done tranid: 0
  covering 512 bytes
```

Examples of 0 type completion records.

```
00013f00: id 182  func 103  ser 0  lser 0  len 244
Group Done tranid: 177
  covering 256 bytes

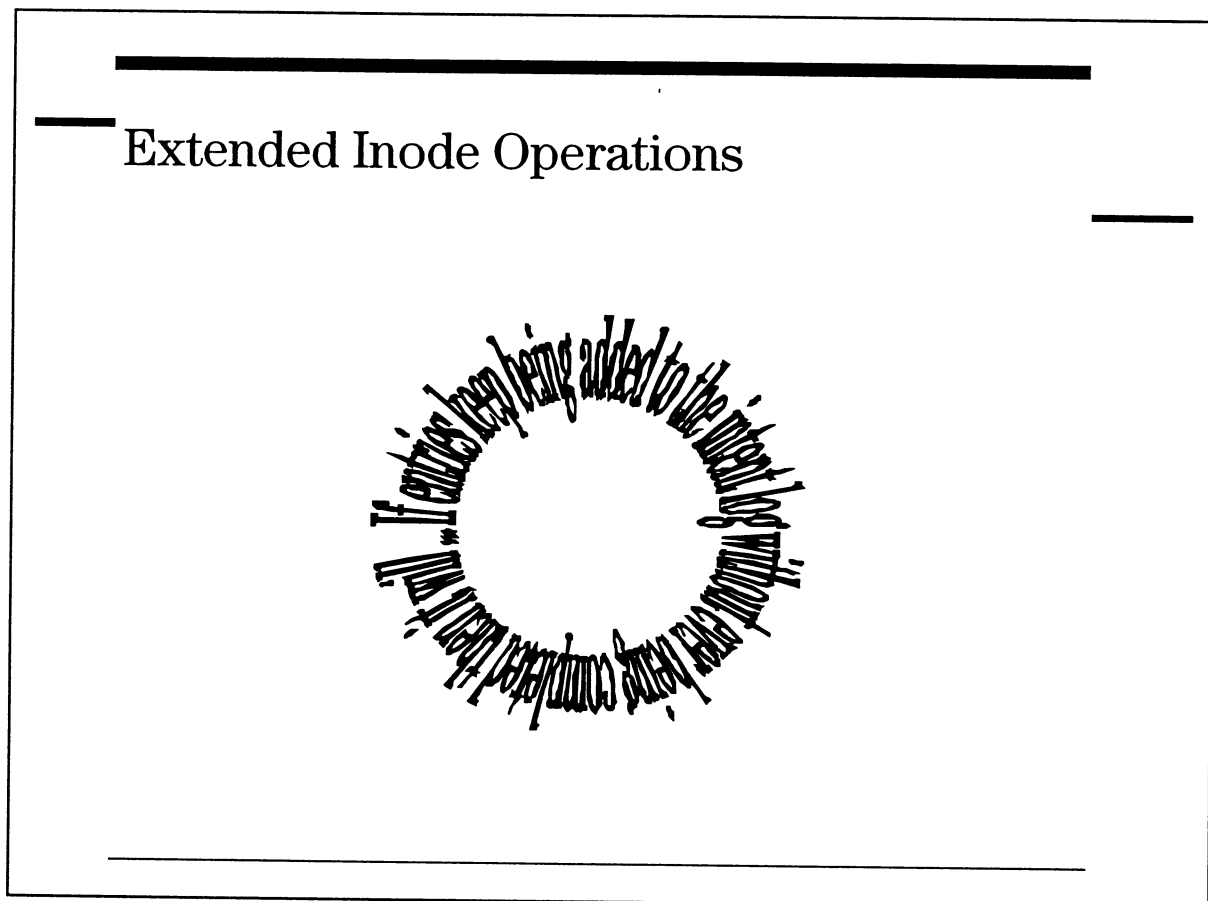
00014000: id 183  func 103  ser 0  lser 0  len 500
Group Done tranid: 182
  covering 512 bytes

00014200: id 184  func 103  ser 0  lser 0  len 500
Group Done tranid: 182
  covering 512 bytes
```

Examples of completion records for an earlier transaction.

Where there are these 0 type records, or multiple normal records, the next record starts on a new block boundary. Since disk IO needs to be performed in whole blocks (or DEV_BSIZE units) this behavior is used to allow the log to be written to disk immediately. In many cases user programs are sleeping, waiting for this IO operation to be completed.

8-29. SLIDE: Extended Inode Operations



Student Notes

The main intent log is circular. The problem with circular logs is that there is a danger of them *eating their own tail*. Should the log wrap around and find an outstanding entry still waiting to be completed, then all further transactions will be forced to wait until there is space in the log.

It is obviously not desirable to have to hold up all new transactions so the file system needs to try and avoid this situation where ever possible.

Most file system operations are very short lived and so are unlikely to cause the log to overflow. However there are a few which might need to be remembered over longer periods of time.

These are known as extended inode operations. There are several **types** of operations that the file system needs to keep track of over longer times, such as when the allocation policies for disk space allocate a larger area than the file has in fact used. This allows the next write to be performed into previously allocated space that has been reserved in the optimum

Module 8
File Systems

position. If however the file is never again extended, this space needs to be returned back to the free pool.

Another, perhaps more important example has to do with the way that files are removed in UNIX. The **rm** command does not directly remove files. The system call it uses has a more accurate name — **unlink**. When a file is unlinked, a name for the file is removed, and its reference count is decremented. When the reference count reaches zero then the file is removed. The main reference count for files is the link count — the number of names the file has. For files that are currently open there is also a second reference count held inside the vnode structure in the kernel. Before the file can be removed, both of these counts must reach zero.

When the file is closed then the reference count in the vnode is decremented. If it and the link count are now zero, then the file is removed. This happens however the file is closed, whether it is closed explicitly by the program calling the **close** system call, or by the kernel calling **close** as part of the exit code.

In the event of a system failure the file will be closed, as the program will have died. However in this case the kernel has also died, so it is unable to check whether a remove operation is required. It would therefore fall to **fsck** to check whether the file is to be deleted.

For VxFS, **fsck** can see this through the use of the extended inode operations facility.

When it runs it can check the clean flag in the superblock and the fileset header to see whether there are any pending extended operations, and then the extended inodes operations map will detail the inodes needing further processing.

```

root@tigger[testDisk3] prealloc f1 $((100*1024*1024))
root@tigger[testDisk3] bdf .
File system      kbytes    used    avail %used Mounted on
/dev/vg00/lvol13 204800 103557 94923  52% /testDisk3
root@tigger[testDisk3] cat >> f1 &
[1] 6666
root@tigger[testDisk3]
[1] + Stopped (SIGTTIN)          cat >> f1 &
root@tigger[testDisk3] rm f1
                                now unlink it
root@tigger[testDisk3] bdf .
File system      kbytes    used    avail %used Mounted on
/dev/vg00/lvol13 204800 103557 94923  52% /testDisk3
root@tigger[testDisk3] ied fsdb -F vxfs /dev/vg00/lvol13
> 8b; p S
                                using fsdb we can see the superblock
super-block at 00000008.0000
magic a501fcf5 version 3 ctime Mon Jul 27 15:03:04 1998
:
:
free 101243 ifree 0
efree 1 1 0 1 1 1 1 2 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
flags 0 mod 0 clean 3c time Mon Jul 27 15:04:41 1998      0x3c means dirty
oltext[0] 33 oltext[1] 1282 oltsize 1
iaumlen 1 iausize 4 dinosize 256
dniaddr 3 checksum2 62c
features 0 checksum3 0
>
> 4i
                                view the files inode
inode structure at 0x00000461.0000
                                note that nlink is now 0
type IFREG mode 100666 nlink 0 uid 0 gid 3 size 104857600
atime 901548255 780001 mtime 901548267 840035 ctime 901548267 840035
aflags 0 orgtype 1 eopflags 1 eopdata 0
                                and that eopflags is 1
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 102400 gen 0 version 0 12867 iattrino 0 noverlay 0
de: 1304 196608 32768 65536 0 0 0 0 0 0
des: 31464 8192 32768 29976 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
> 1fset
                                by moving to the attribute fileset
                                it is possible to look at the inode
                                allocation unit

```

Module 8
File Systems

```
> 64i
inode structure at 0x00000018.0000
type IFIAU mode 6000000777 nlink 1 uid 0 gid 0 size 4096
atime 901548184 581682 mtime 901548184 581682 ctime 901548184 581682
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 0
blocks 4 gen 0 version 0 0 iattrino 0 noverlay 0
de: 36 0 0 0 0 0 0 0 0 0
des: 4 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
> 36b; p IA Print the heade and summary for the IAU
iau header at 0x00000024.0000
magic a505fcf5 fsindx 999 aun 0
au_iextop 1 au_rifree 27 au_ribfree 3 Here the au_iextop flag is set
> 39b; p 8 xW
00000027.0000: 08000000 00000000 00000000 00000000
00000027.0010: 00000000 00000000 00000000 00000000
>
The first byte has a value of 0x08, The maps count their bits from the most significant end, so from here
we can see that bit 4 is set.
```

In this example a file has been removed while still open. See the **rm_log** appendix.

8-30. SLIDE: Directories

Directories

*Сонце Сонцароду
Сонцун но
УгБЕЦТНОМУ АМЕРУ
ИРАНА*

Free space		No hashes					
Hash 1	Hash 2	Hash 3	Hash 4	Hash 5	Hash 6	Hash 7	Hash 8
Hash 9	Hash 10	Hash 11	Hash 12	Hash 13	Hash 14	Hash 15	Hash 16
Hash 17	Hash 18	Hash 19	Hash 20	Hash 21	Hash 22	Hash 23	Hash 24
Hash 25	Hash 26	Hash 27	Hash 28	Hash 29	Hash 30	Hash 31	Hash 32

Inode #	rec len	fn. len	next ptr	filename	padding
Inode #	rec len	fn. len	next ptr	filename	padding
Inode #	rec len	fn. len	next ptr	filename	p
Inode #	rec len	fn. len	next ptr	filename	padding
Inode #	rec len	fn. len	next ptr	filename	
Inode #	rec len	fn. len	next ptr	filename	padding

↓ как в HFS

Student Notes

With VxFS there are two basic forms that directories can take:

- Immediate, for small directories where the directory occupies 96 bytes or less. The directory is stored inside its inode.
- Normal, for most directories. Data extents are allocated from the file system to hold the directory information.

For the immediate directories, the record structure is the same as for UFS.

Допараметры :

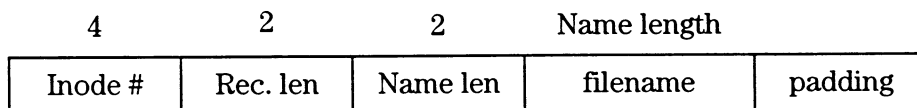
1. Пароль
2. Свод. информация
3. Категория

Module 8
File Systems

```

root@tiger[] ied fsdb -F vxfs /dev/vg00/lvol13
> 2i
inode structure at 0x00000460.0200
type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96
atime 901555768 590001 mtime 901555764 750006 ctime 901555764 750006
aflags 0 orgtype 2 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2
blocks 0 gen 0 version 0 26200 iattrino 0 noverlay 0
> im
00000460.0250: 00000000
> p db
immediate directory block at 00000460.0250 - total free (d_tfree) 24
00000460.0254: d 0 d_ino 3 d_reclen 20 d_namlen 10
                l o s t + f o u n d
00000460.0268: d 1 d_ino 4 d_reclen 16 d_namlen 3
                o n e
00000460.0278: d 2 d_ino 5 d_reclen 16 d_namlen 3
                t w o
00000460.0288: d 3 d_ino 6 d_reclen 40 d_namlen 5
                t h r e e
>

```



The entry starts with the inode number of the file, followed by the records and name lengths. The filename itself is possibly followed by some padding. The padding performs two roles:

- It pads the length of the entries out to a multiple of 4. This way all inode numbers are correctly aligned.
- It absorbs unused space either at the end of the directory or when files are deleted.

For the external directories the VxFS differs from UFS in that it provides an index to improve the performance of filename lookups — a long standing UNIX problem.

Free space		No hashes					
Hash 1	Hash 2	Hash 3	Hash 4	Hash 5	Hash 6	Hash 7	Hash 8
Hash 9	Hash 10	Hash 11	Hash 12	Hash 13	Hash 14	Hash 15	Hash 16
Hash 17	Hash 18	Hash 19	Hash 20	Hash 21	Hash 22	Hash 23	Hash 24
Hash 25	Hash 26	Hash 27	Hash 28	Hash 29	Hash 30	Hash 31	Hash 32
Inode #	rec len	fn. len	next ptr	filename	padding		
Inode #	rec len	fn. len	next ptr	filename	padding		
Inode #	rec len	fn. len	next ptr	filename	p		
Inode #	rec len	fn. len	next ptr	filename	padding		
Inode #	rec len	fn. len	next ptr	filename			
Inode #	rec len	fn. len	next ptr	filename	padding		

The directory now starts with a header that gives the total amount of free space available in the directory currently, the size of the index (this will be 32) and then the index or hash chain pointers.

The directory entries themselves then need to have next hash chain pointers to allow the indexed searches to find the next possible entry.

```

> 2i
inode structure at 0x00000460.0200
type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024
atime 901556710 230001 mtime 901556717 720010 ctime 901556717 720010
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2
blocks 1 gen 0 version 0 26247 iattrino 0 noverlay 0
de: 1285 0 0 0 0 0 0 0 0 0
des: 1 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
> 1285b ; p db
directory block at 00000505.0000 - total free (d_tfree) 888 nhash 32
Hash 0- 7: 0000 0000 0058 0000 0000 0000 0000 0000
Hash 8-15: 0000 0044 0000 0000 0000 0000 0000 0000
Hash 16-23: 0000 0000 0000 0000 0000 0000 0000 0000
Hash 24-31: 0078 0000 0068 0000 0000 0000 0000 0000
                                4 bytes      2 bytes      2 bytes      2 bytes
00000505.0044: d 0      d_ino 3  d_reclen 20  d_namlen 10  d_hashnext 0000
                l o s t + f o u n d
00000505.0058: d 1      d_ino 4  d_reclen 16  d_namlen 3  d_hashnext 0000
                o n e
                + 3 bytes of padding
00000505.0068: d 2      d_ino 5  d_reclen 16  d_namlen 3  d_hashnext 0000
                t w o
                + 3 bytes of padding
00000505.0078: d 3      d_ino 6  d_reclen 904  d_namlen 5  d_hashnext 0000
                t h r e e
                + 889 bytes of padding
>
                                1 for alignment
                                888 free space

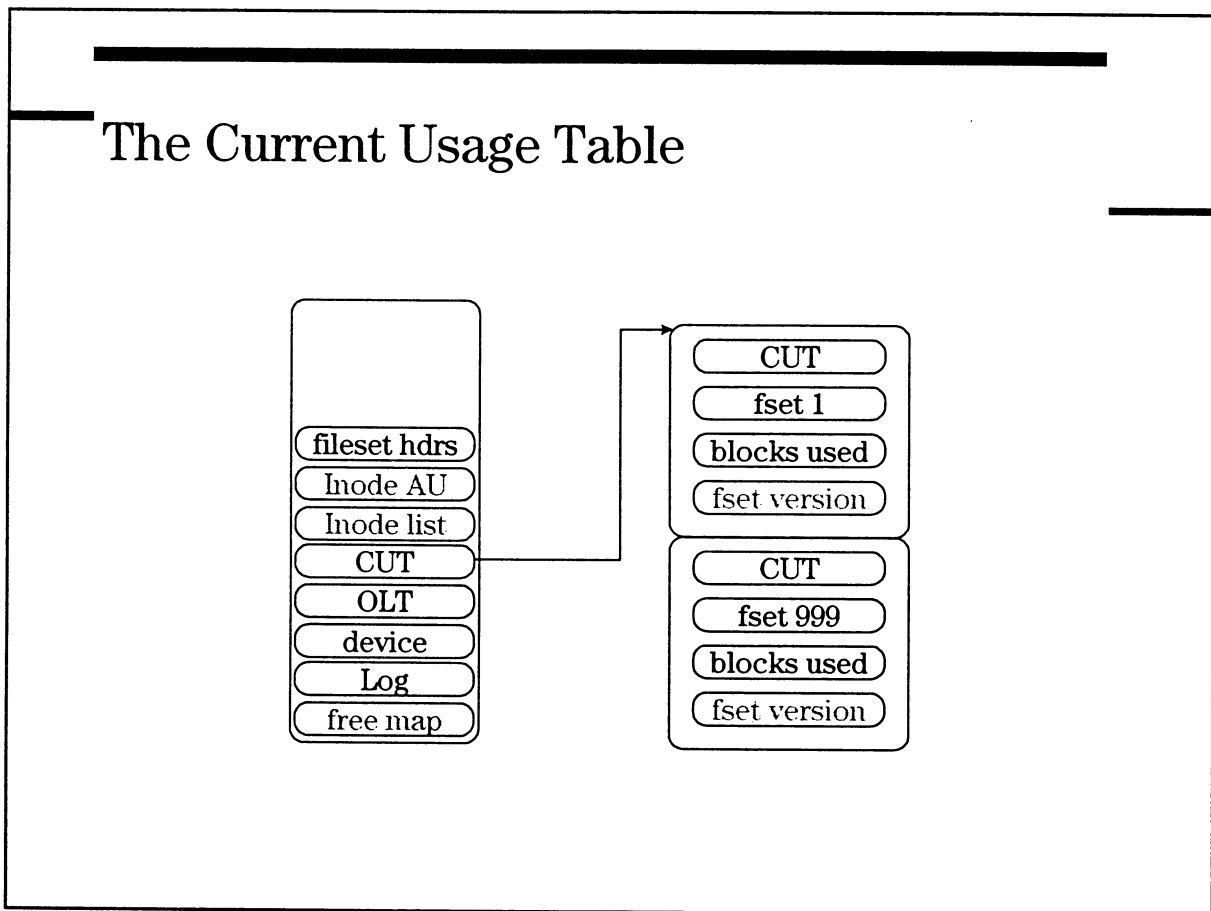
```

Module 8
File Systems

In this example, the earlier directory has been expanded to the point where it no longer fits in the immediate area of the inode. For the sake of brevity it has been reduced to its previous size. VxFS never converts non-immediate files back to immediate ones.

The other feature of VxFS directories is the way they handle "." and "..". Since all directories are known to contain these entries there is little point in storing them. The VxFS directory access functions can provide them when required. The only information the directory can not just generate is the inode number of the parent, so this is stored directly in the inode, as previously discussed.

8-31. SLIDE: The Current Usage Table



Student Notes

The current usage table is used to provide up to date status information about the file system. Nominally it contains the number of blocks used and a fileset version number. The header file `/usr/include/sys/fs/vx_fs.h` details the structure of the entries in the current usage table.

Only the `cu_curused` field that gives the number of blocks used by the fileset is used as documentation. The fileset version number is supposed to be incremented on all changes to the filesets (such as creating, removing or changing a file), however neither the disk nor kernel version of this value follow this rule.

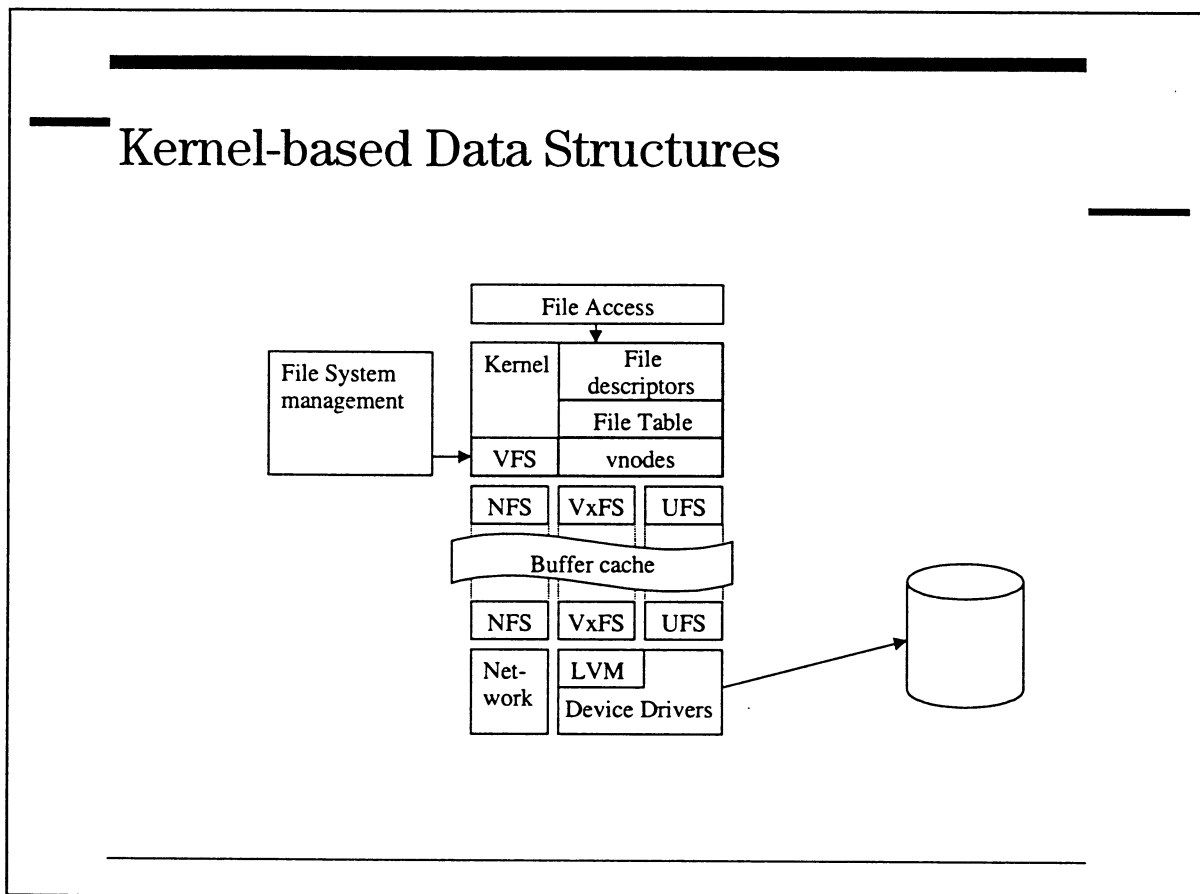
Module 8
File Systems

When using **fsdb** to read the current usage table, the command **cut** only displays the entry for the first fileset. To display the whole table it is necessary to find the block number where the table is stored and then print the two entries from there.

```
> lfset
fset header structure at 0x00000009.0000
.
.
.
fsh_states 0x0 fsh_status ""
>olt
OLT at 0x00000021.0000
.
.
.
OLT cut entry:
    olt_type 3  olt_size 16  olt_cutino 6
.
.
.
>6i
inode structure at 0x00000011.0200
type IFCUT mode 10000000777 nlink 1 uid 0 gid 0 size 1024
atime 901548184 581682 mtime 901548184 581682 ctime 901548184 581682
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 1 gen 0 version 0 0 iattrino 0 noverlay 0
de: 15 0 0 0 0 0 0 0 0 0
des: 1 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>15b; p 2 C
cut entries start at 0x0000000f.0000
cu_fsindex 1 cu_curused 1157 cu_lversion 0 cu_hversion 0
cu_flags 0 cu_dcurused 0 cu_visused 0 cu_update 0 0
cu_fsindex 999 cu_curused 102428 cu_lversion 76347 cu_hversion 0
cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
>
```

Starting in fileset 1 the inode number of the current usage table can be seen from the object location table.

8-32. SLIDE: Kernel-based Data Structures



Student Notes

As well as looking at the arrangement of data on the disks, we also need to take a look at how the kernel keeps track of the different file systems that are mounted and the individual files that are being used.

Since the kernel supports different types of file system design and largely make the type invisible to calling programs, and even to much of the kernel itself, a switching mechanism is needed. This allows file systems and files to be accessed by generic functions and have these mapped onto the specific functions that operate on the current type.

For file systems this switching is achieved through the VFS (virtual file system) structure².

For files the vnode (virtual node) structure³ is used.

Both of these structures were first introduced to the UNIX kernel by Sun Microsystems when they implemented NFS and needed to add support for a second type of file system. Their

² see /usr/include/sys/vfs.h

³ see /usr/include/sys/vnode.h

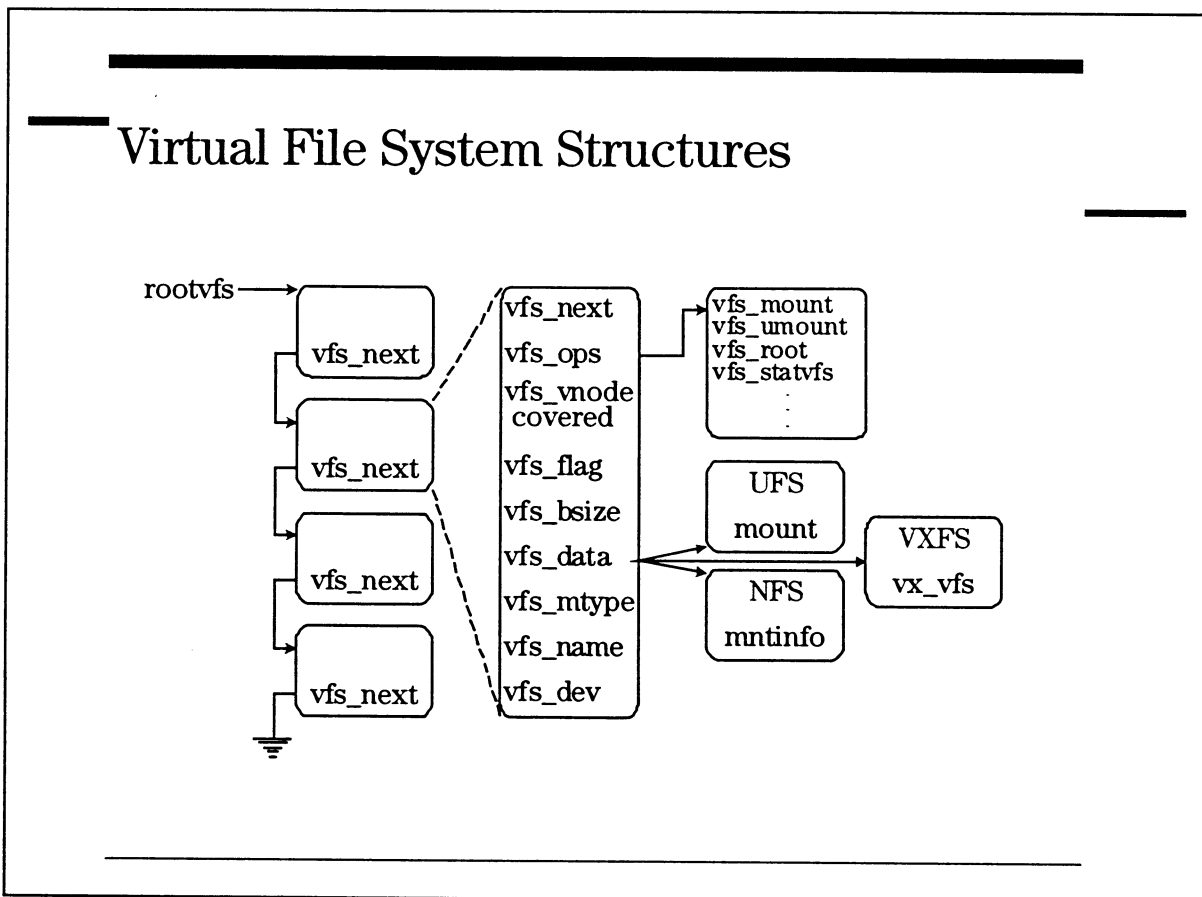
Module 8
File Systems

design concept stems from object-oriented mechanisms although, as with the rest of the kernel, they are just written in C.

These structures have pointers to the functions that operate on this type of file system or file. Thus any higher level code that, for instance, wants to read from a file needs only to know where its vnode is, and that will then point it to the relevant function for reading the data.

This sort of method allows other file system designs to be added to the kernel without the need to make extensive changes to existing code.

8-33. SLIDE: Virtual File System Structures



Student notes

At the heart of the kernel management of the mounted file system is the list of **vfs** structures.

Module 8
File Systems

```

struct vfs {
    struct vfs      *vfs_next;           /* next vfs in vfs list */
    struct vfs      *vfs_hash;          /* hash chain for fast lookup*/
    struct vfsops   *vfs_op;            /* operations on vfs */
    struct vnode    *vfs_vnodecovered;  /* vnode we mounted on */
    int32_t         vfs_flag;           /* flags */
    int32_t         vfs_bsize;          /* native block size */
    uint16_t        vfs_exroot;         /* exported fs uid 0 mapping */
    int16_t         vfs_exflags;        /* exported fs flags */
    caddr_t         vfs_data;          /* private data */
    int32_t         vfs_icount;         /* ref count of processes */
    /* sleeping on the mnt inode */
    int16_t         vfs_mtype;          /* Type of vfs */
    fsid_t          vfs_fsid;           /* file system ID for vfs_get */
    struct log_hdrT *vfs_logp;          /* ptr to WA log */
    time_t          vfs_mnttime;        /* time mounted */
    char            vfs_name[MAXPATHLEN]; /* file sys identifier */
    dev_t           vfs_dev;            /* device (if approp) */
    struct ncache   *vfs_ncachehd;      /* DNLC entries */
};

```

The key field to the **vfs**'s role is the **vfs_op** pointer. This points to a structure that contains the file system- specific functions for operating on this file system.

vfs_next	The vfs structures are held in a link list referenced by the pointer rootvfs . This field is then used to hold the list together.
vfs_op	The pointer to the set of functions that operate on this file system
vfs_vnodecovered	With the exception of the root files system, new file systems are mounted on top directories in existing file systems. The existing (mount point) directory will have a vnode set up within the kernel, this field references that vnode.
vfs_flag	Flags such as read only or largefiles... etc.
vfs_exroot vfs_exflags	While these fields are provided for use by NFS, they are not currently used on HP-UX.
vfs_data	The pointer to the type specific data structure, such as a mount structure for UFS file systems, or vx_vfs structures for vxfs .
vfs_mtype	The file system type number. These values are dynamically allocated at boot time as the drivers are installed into the kernel. The order that the file system drivers are installed is governed by the order that they are listed in the system file, and then appear in the conf.c file made as part of kernel generation.
vfs_ncachehd	Points to the chain of dnlc entries for files on this file system.

The **vfs** operations are

```

struct vfsops {
    vfs_mount_t      *vfs_mount;
    vfs_unmount_t    *vfs_unmount;
    vfs_root_t       *vfs_root;
    vfs_statvfs_t    *vfs_statvfs;
    vfs_sync_t       *vfs_sync;
    vfs_vget_t       *vfs_vget;           used to allow NFS to find vnodes
    vfs_getmount_t   *vfs_getmount;
    vfs_freeze_t     *vfs_freeze;        used by lvsplit to be able to split fs in a
    vfs_thaw_t       *vfs_thaw;          consistent state
    vfs_quota_t      *vfs_quota;
    vfs_mountroot_t *vfs_mountroot;
    vfs_size_t       *vfs_size;
};

```

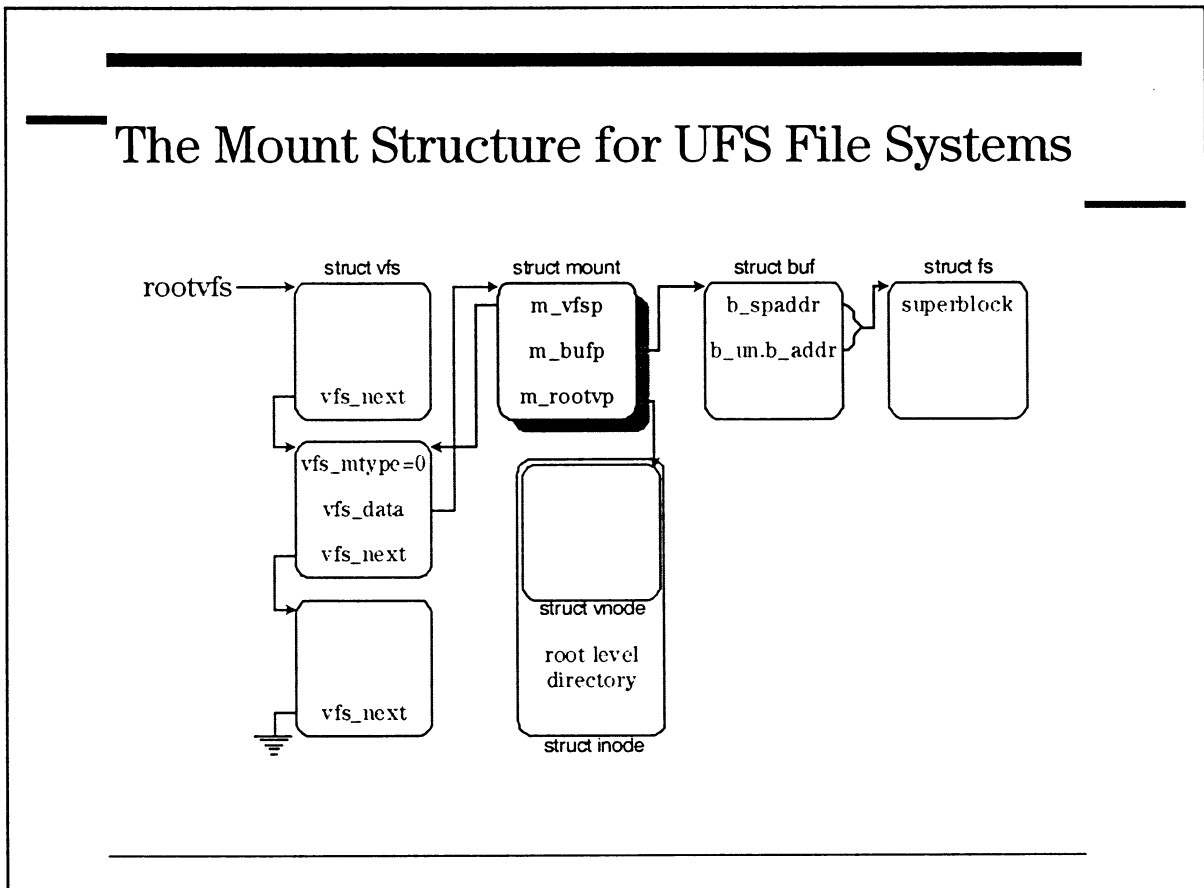
The flags are

```

#define VFS_RDONLY      0x01           /* read only vfs */
#define VFS_MLOCK       0x02           /* lock vfs so that subtree is stable
*/
#define VFS_MWAIT       0x04           /* someone is waiting for lock */
#define VFS_NOSUID      0x08           /* someone is waiting for lock */
#define VFS_EXPORTED    0x10           /* file system is exported (NFS) */
#define VFS_REMOTE      0x20           /* remote file system (NFS) */
#define VFS_HASQUOTA    0x40           /* file system with quotas */
#define VFS_MI_DEV      0x100          /* dev_t has mgr_index in it already */
#define VFS_MOUNTING    0x200          /* set with VFS_MLOCK if mounting */
#define VFS_ORD          0x400          /* use fast readdir for LADDIS */
#define VFS_LARGEFILES  0x800          /* Large File mount */

```

8-34. SLIDE: The Mount Structure for UFS File Systems



Student Notes

Where the file system mounted is of type UFS then the **vfs_data** pointer points to a mount structure. This then hold UFS specific information, since the design of this structure predates that of the VFS it also has some data that is now also held with the VFS structure.

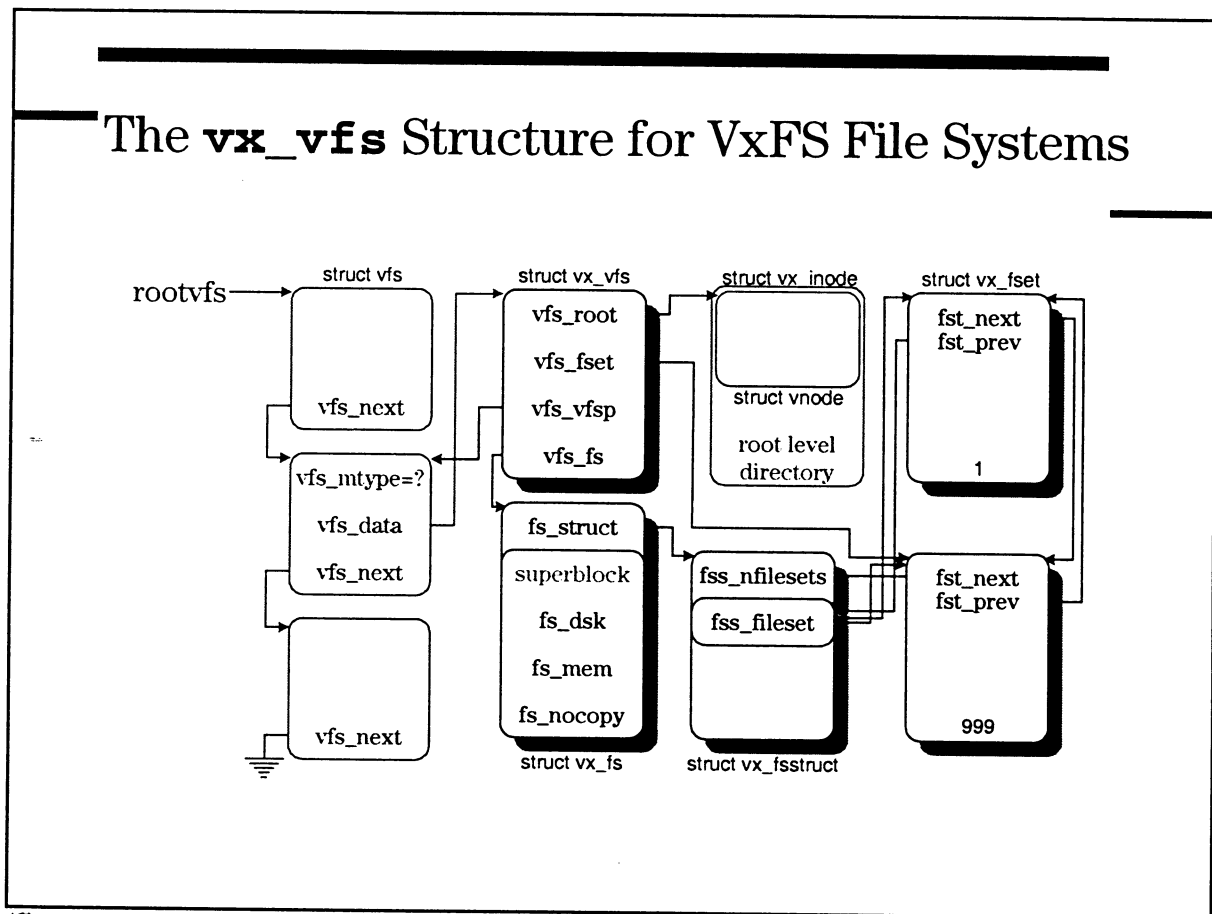
The mount structures provide access to the superblock, which is held permanently in the buffer cache, so access is via a **buf** structure⁴.

Also it is possible to access the vnode for the root level directory of the files ystem via the **m_rootvp** field.

Currently the file system type number used for UFS is hardcoded as 0. Prior to HP-UX 10.30 NFS was also hardcoded as 1. According to the release notes for 11.00 this is no longer the case, but experimentation still seems to always show the type to be 1 for NFS file systems.

⁴ **buf** structures are discussed in the section on the buffer cache.

8-35. SLIDE: The **vx_fs** Structure for VxFS File Systems



Student Notes

For VxFS files systems there are many more data structures, starting with the **vx_vfs** structure. Within the VxFS file system there are the filesets, represented by the list of **vx_fset** structures. The superblock is described using the **vx_fs** structure and the file system-wide structures are described by the **vx_fsstruct** structure.

Module 8 File Systems

Starting with the **vx_vfs** structure.

```
struct vx_vfs {
    struct vnode    *vfs_root;        /* 0x00 root vnode */
    struct vx_fset  *vfs_fset;       /* 0x04 pointer to file set */
    struct vfs      *vfs_vfsp;       /* 0x08 pointer to mount */
    struct vx_fs    *vfs_fs;         /* 0x0c pointer to file system */
};
```

The **vx_vfs** structure just provides pointers to the other structures involved:

- The **vnode** for the root level directory of the file system.
- The fileset structure for the user visible fileset.
- A pointer back to the VFS structure.
- The overall file system structure.

```
struct vx_fs {
    struct vx_fsstruct  *fs_struct;    /* 0x000 structural portion */
    struct vx_fsdisk    fs_dsk;        /* 0x008 disk portion */
    struct vx_fsmem     fs_mem;        /* 0x1b8 memory only portion */
    int                 fs_pad;        /* 0x2dc compiler added pad */
    struct vx_fsncopy   fs_ncpy;      /* 0x2e0 non copyable */
    /* 0x436 is length */
};
```

At first glance the **vx_fs** struct is deceptively simple. However it contains other nested structures to contain a copy of the disk based superblock. The kernel-based file system structure is divided into two parts. **vx_fsmem** contains information that can change when the file system is remounted or has its size changed. **fs_ncpy** has the file system information that does not change in these events.

The **vx_fsstruct** structure is used to describe the filesets and the data structures from the attribute filesets that are file system-wide.

```

struct vx_fsstruct {
    struct vx_fs      *fss_fsp;          /* 0x00 pointer to fs structure */
    vx_uq_t           fss_nfileset;     /* 0x04 number of file sets loaded */
    struct vx_fsetlink fss_fileset;     /* 0x08 list of file sets */
    struct vx_device **fss_devlist;     /* 0x18 device structures */
    vx_uq_t           fss_ndevs;       /* 0x1c number of device structures */
    struct vx_fset    *fss_structfset; /* 0x20 ptr to structural fileset */
    struct vx_inode  *fss_superip;     /* 0x24 file containing super-blocks */
    struct vx_inode  *fss_logip[2];    /* 0x28 file containing log */
    struct vx_inode  *fss_fsetip[2];   /* 0x30 files containing fset headers*/
    struct vx_inode  *fss_devconfip[2]; /* 0x38 files containing dev cfg */
    struct vx_inode  *fss_oltip[2];    /* 0x40 files containing OLT */
    struct vx_inode  *fss_cutip;       /* 0x48 file containing cut */
    struct vx_cut    **fss_cutlist;    /* 0x4c pointers to cuts */
    struct vx_fsetid **fss_ident;      /* 0x54 fileset identity info */
    vx_daddr_t       *fss_logblocks;   /* 0x148 map of log blocks */
    vx_blkcnt_t      fss_loglen;       /* 0x14c number of log blocks */
    vx_uq_t          fss_ncuts;        /* 0x50 number of cut pointers */
    vx_uq_t          fss_nident;       /* 0x58 cnt of ident structs */
    /* 0x5c is length */
};

```

So here we find the number of filesets within the file system (there will be two at VxFS version 3) and then a structure that points to the fileset structures.

```

struct vx_fsetlink {
    struct vx_fset *fst_next;
    struct vx_fset *fst_prev;
    struct vx_fset *fst_cforw;
    struct vx_fset *fst_cback;
};

```

Where **fst_next** points to the first filesets within the file system, then using the **fst_next** pointer from there a list of the filesets can be built. Note that this list comes back through the **vx_fsetlink** structure.

The list can also be followed in reverse using the **prev** pointers.

This same link structure is also used from the in-core **CUT** structures, using the **fst_cforw** and **fst_cback** fields. Surprisingly the **vx_fset** structure does not use this link structure, but rather repeats these four fields at its start.

The **vx_fsstruct** structure then provides information on incore **vx_inodes** representing many of the files from the structural fileset.

Module 8
File Systems

For each of the filesets there is a **vx_fset** structure.

```
struct vx_fset {
    struct vx_fset    *fst_next;    /* 0x00 pointer to next file set */
    struct vx_fset    *fst_prev;    /* 0x04 pointer to previous file set */
    struct vx_fset    *fst_cforw;   /* 0x08 forw ptr for cut fset chain */
    struct vx_fset    *fst_cback;   /* 0x0c back ptr for cut fset chain */
    vx_blkcnt_t       fst_curused;   /* 0x10 current blocks used */
    vx_blkcnt_t       fst_visused;  /* 0x14 visible blocks used */
    vx_blkcnt_t       fst_strused;  /* 0x18 structural blocks used */
    struct vx_ias     *fst_ias[2];  /* 0x1a inode lists */
    struct vx_ias     *fst_attrias; /* 0x20 attribute inode list */
    struct vx_mountinfo {
        char    fmi_ronly;    /* 0x28 mounted readonly */
        char    fmi_blkclear; /* 0x29 guarantee cleared storage */
        char    fmi_delaylog; /* 0x2a workstation semantics */
        char    fmi_tmplg;    /* 0x2b tmp file system semantics */
        char    fmi_nolog;    /* 0x2c logging disabled */
        char    fmi_logwrites; /* 0x2d log synchronous writes */
        char    fmi_mounted;  /* 0x2e file set mounted */
        char    fmi_loglevel; /* 0x2f logging level */
        vx_uq_t    fmi_mincache; /* 0x30 mount caching flags */
        vx_uq_t    fmi_convosync; /* 0x34 mount convosync flags */
        caddr_t    fmi_mntpt;  /* 0x38 mount point of file system */
        size_t    fmi_mntlen;  /* 0x3c len of mntpt pathname */
        time_t    fmi_btime[VX_MAXQUOTAS]; /* 040 block time limit */
        time_t    fmi_itime[VX_MAXQUOTAS]; /* 0x48 inode time limit */
        vx_uq_t    fmi_qflags[VX_MAXQUOTAS]; /* 0x50 quota flags */
        struct vnode *fmi_quotvp[VX_MAXQUOTAS]; /* 0x58 quota vnodes */
        struct ucred *fmi_qcred[VX_MAXQUOTAS]; /* 0x60 quota creds */
    }
    fst_mountinfo;
    vx_hyper_t     fst_cversion; /* 0x68 current file set vers */
    vx_hyper_t     fst_dversion; /* 0x70 disk file set version */
    vx_hyper_t     fst_iversion; /* 0x78 next incr version number */
    vx_u32_t       fst_dflags;   /* 0x80 current copy of dflags */
    vx_uq_t        fst_unused;   /* 0x84 field for rent */
    struct vx_fsstruct *fst_fsstruct; /* 0x88 pointer to file system */
    struct vx_fsethead *fst_fshd; /* 0x8c file set disk header */
    struct vx_vfs     *fst_vxp;  /* 0x90 pointer to vfs private data */
    struct vx_cut     *fst_cutp;  /* 0x94 pointer to cut table */
    struct vx_mlinkhd fst_cdlink; /* 0x98 cut mlinks in done order */
    struct vx_mlinkhd fst_cflink; /* 0xa0 cut mlinks being flushed */
    struct vx_mlink   *fst_clink; /* 0xa8 cut mlinks in log order */
    vx_uq_t           fst_nmlink; /* 0xac num mlinks on cdlink chain */
    vx_off_t          fst_cuoffset; /* 0xb0 byte offset in cut table */
    union vx_cuent    fst_cuent;  /* 0xb8 cut updates in log order */
    union vx_cuent    fst_dskent;  /* 0xf8 cut updates in done order */
    struct vx_fsetid  *fst_ident;  /* 0x138 fileset identity structure */
    vx_uq_t           fst_mflags;  /* 0x13c n memory flags */
    struct vx_inode   *fst_quotip[VX_MAXQUOTAS]; /* 0x140 quota inodes */
    vx_sleep_t        fst_dirlock_slk; /* 0x148 ptr to directory lock */
    vx_spin_t         fst_curused_lk; /* 0x158 curused and vversion lock */
    vx_spin_t         fst_cut_lk;    /* 0x160 cut spin lock */
    vx_blkcnt_t       fst_qblocks;   /* 0x168 DFS disk quota limit */
    caddr_t           fst_vol;       /* 0x170 dfs volume ptr */
    vx_uq_t           fst_features;  /* 0x174 fileset features */
    vx_uq_t           fst_largefiles; /* 0x178 large files created */
};
/* 0x17c is length */
```

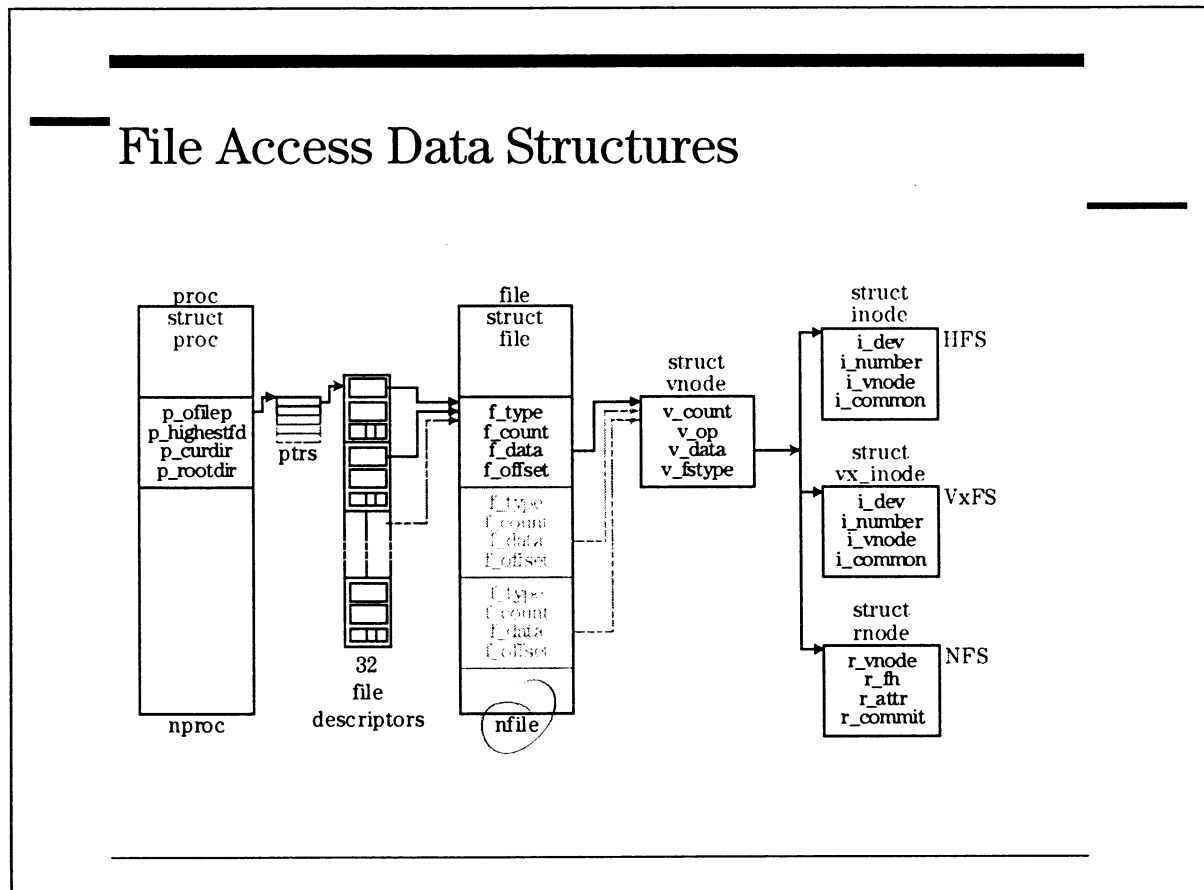
The **vx_fset** structures are held on a circular linked list. This linked list also contains the **vx_fsetlink** structure previously discussed.

The fileset data structures show how the file system was mounted and reference the in-core copies of the various files from the structural fileset that describes this fileset.

10
11
12
13

14
15
16
17

8-36. SLIDE: File Access Data Structures



Student Notes

When a UNIX program accesses a file it does it through a file descriptor. The number of file descriptors per process is limited via the `max_files` and `max_files_lim` kernel parameters. The file descriptor information for a process can be referenced through the pointer `p_ofilep` from the process table entry. Since the number of file descriptors per process can vary wildly, many just have `stdin`, `stdout` and `stderr`. The maximum supported is 60,000 so it would be inefficient to store this information directly in a table. File descriptors are stored in blocks of 32, and new blocks are created only when needed. The `p_ofilep` pointer then points to an array of pointers to these blocks of 32. This array of pointers is then dynamically reallocated should more be needed.

Prior to HP-UX 10.20 the file descriptors used a data structure of type `ofile_t`, and `p_ofilep` was defined as a pointer to an `ofile_t` pointer. At 10.20 the data type was redefined and the new type is not listed in any header file. Now `p_ofilep` is defined as either a `void**` or an `int**`.

The new structure still mainly acts as a pointer to the systems file table. It also has the information that was previously kept in the `ofile_t` structures such as the `close on exec` flag.

When a file is opened a new file descriptor structure is set up and it will point to a new entry within the system file table. The `open(2)` call always returns the lowest available file descriptor.

It is also possible to duplicate file descriptors, such that two or more point to the same file table entry. There are two groups of system calls that duplicate the descriptors:

- The dup calls, `dup(2)`, `dup2(2)` and `fcntl(2)`.
- The fork calls, `fork(2)` and `vfork(2)`.

The dup calls are provided explicitly to allow a process to copy its file descriptors, often so that it can control the file descriptor numbers. For example the shell dup's the file descriptors for a pipe into `stdin` and `stdout` when building pipelines.

The duplication of file descriptors by the `fork` calls is less obvious, but as we will see when we look at the file table, highly important. Without it, simple IO redirection from the shell would not work in a usable fashion.

For each open of a file, there is an entry in the file table. If the same file is opened more than once then several slots in the file table will exist for the same file. The file table entry therefore keeps information specific to this opening and this usage only.

Information about the file is stored in separate structures. Thus all the entries in the file table for a single file can then point to (and share) a single structure for any information that is common to all uses of the file.

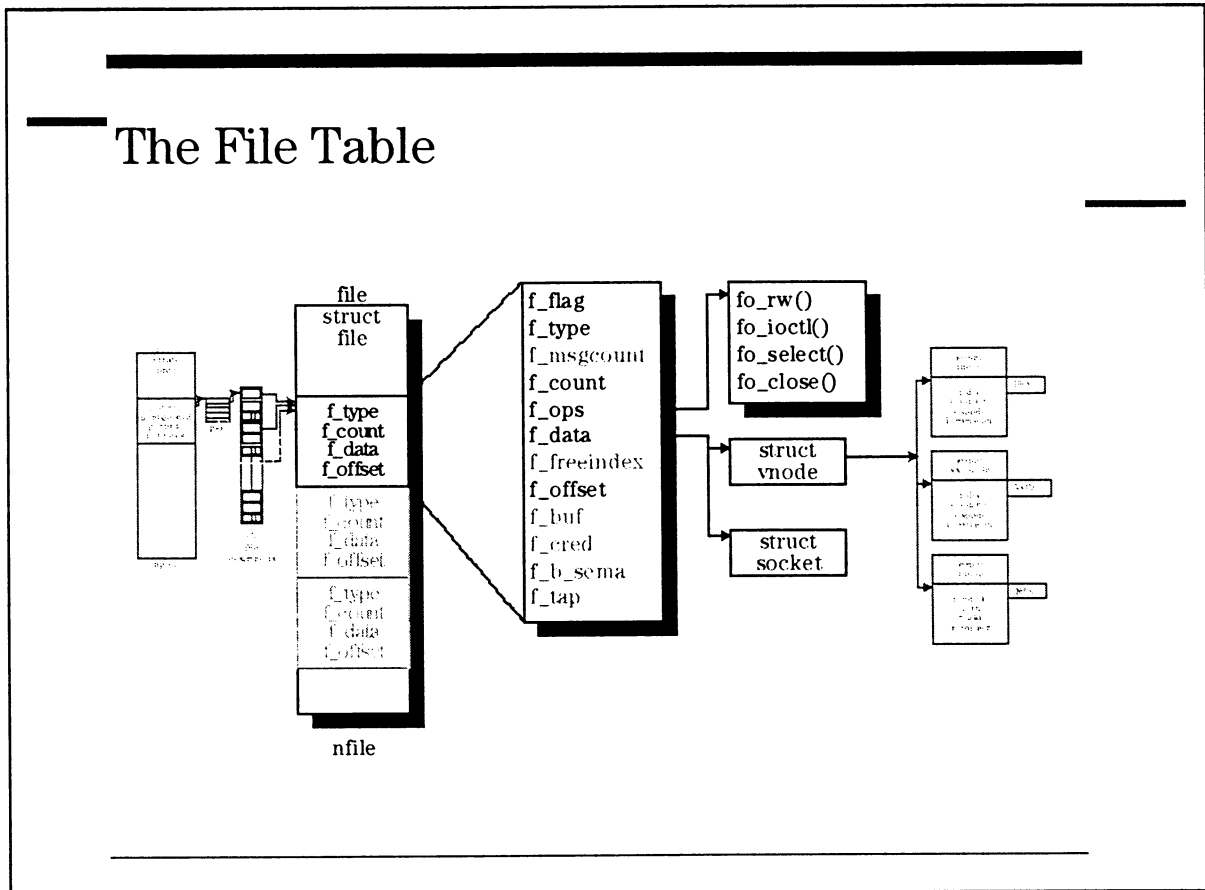
In early UNIX kernels (for HP-UX before 6.0) there was only a single type of file system and so the `file` structures pointed directly to an `inode` structure to describe the file.

When Sun Microsystems added the Networked File System they added a layer of abstraction that meant that any code accessing files at a high level could do so without the need to understand the type of the file. This new layer uses the `vnode` structure to then act as the switch to the different types of file system. The `vnodes` have pointers to all the functions that the kernel performs on files.

After the `vnode` layer there are different structures for the different file system types:

- Inode structures for UFS.
- `Vx_inodes` for VxFS.
- `Rnodes` for NFS.

8-37. SLIDE: The File Table



Student Notes

The system file table keeps track of all the files and sockets opened on the system. Each time a file is opened or a socket is created, a slot in the file table is used. If there are several opens on the same file, each will require a separate entry in the table.

Since file descriptors and file table entries are used for both normal files and sockets, the file structure needs to be able to cope with both usages. The Berkley socket environment predates the **vnode** structure and so it switches at the file structure level.

```

struct file {
    int     f_flag;           /* see below */
    short   f_type;          /* descriptor type */
    short   f_msgcount;      /* references from message queue */
    int     f_count;         /* reference count */
    struct fileops *f_ops;
    caddr_t f_data;          /* ptr to file specific struct (vnode/socket)*/
    int     f_freeindex;     /* index of free file table entry */
                                /* -1 => no more free entries */
#ifdef __STDC_32_MODE__
    off64_t f_offset;        /* off_t would be wrong, since not _KERNEL */
#else
    off32_t f_offset[2];
#endif
    caddr_t f_buf;
    struct ucred *f_cred;    /* credentials of user who opened file */
    b_sema_t f_b_sema;
    short   f_tap;          /* audit flag to allow for data intercept */
};

```

The flags are used when the file is opened or subsequently by **fcntl(2)** to control how the file is to be used.

The **f_count** field indicates how many file descriptors are pointing to this **file** structure. This usage count is need for example when a file is closed. This entry will continue to be used so long as any file descriptor is still pointing to it.

The **f_ops** field then points to the set of functions that apply to this type of file entry. The **fileops** structure contains:

```

struct fileops {
    fo_rw_t      *fo_rw;
    fo_ioctl_t   *fo_ioctl;
    fo_select_t  *fo_select;
    fo_close_t   *fo_close;
};

```

These are the functions that must differentiate between a file and a socket.

The structure that contains the file- or socket-specific data is then pointed to by the **f_data** field.

Module 8 File Systems

Lastly the `f_offset` field holds the current position within the file. It is this field that particularly needs to be shared through a `fork(2)` call. Let us consider what would happen if the output of the following shell script were to be redirected if the current file pointer was held at the process level.

```
#!/usr/bin/sh
# myScript, an example of a simple shell script
date
ls
```

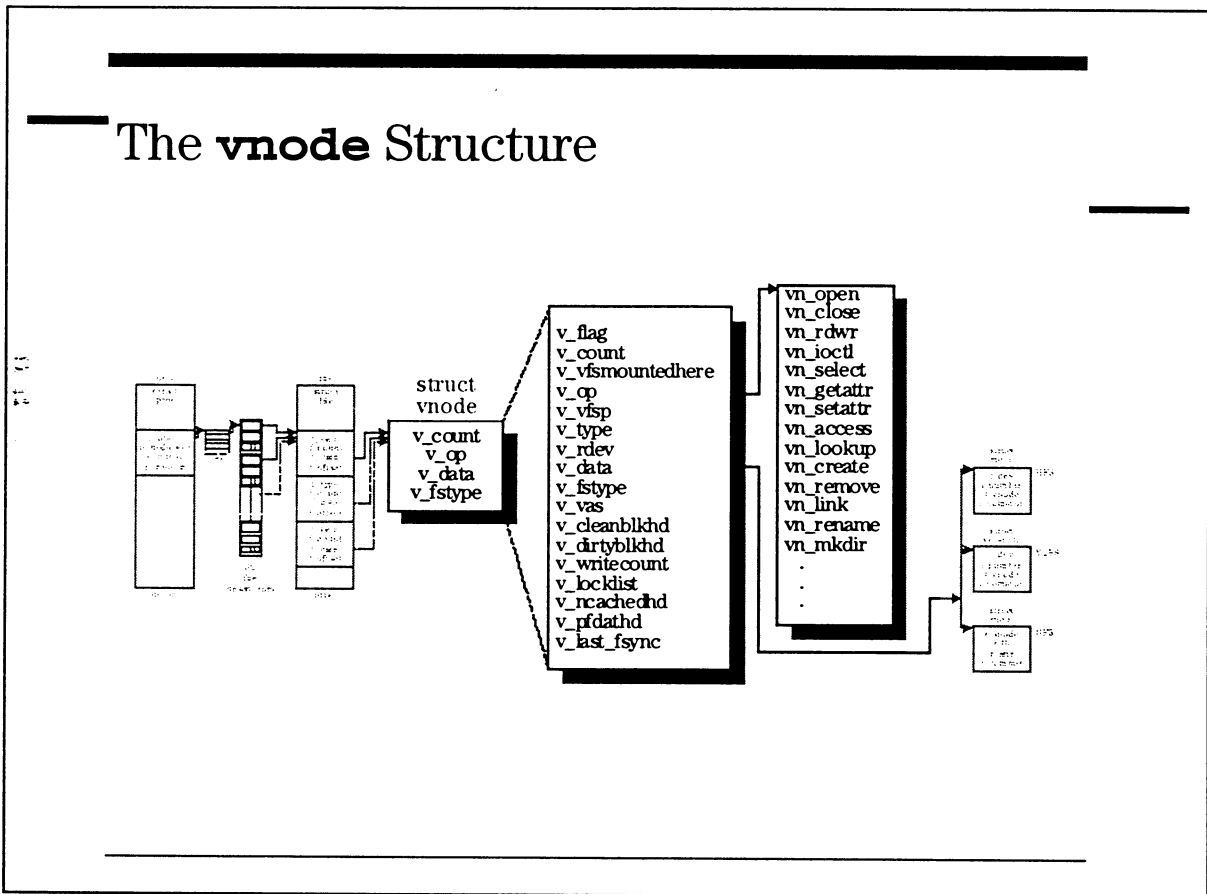
Since this is unlikely to be the result we want another method is obviously needed.

As the file pointer is not held at the process level, this is not the behavior that is observed. As `date` runs it moves the file pointer along. When it exits, there is no movement of the file pointer and so `ls` starts from the current end of file.

Under some conditions it will be desirable for, say a parent and child, to have independent control over the file pointer. In order to achieve this the file will need to be re-opened. This method of having the file pointer managed away from the process means that the simple case works simply, at a slight cost to the more complex case.

In the case of the file descriptor for a file, `f_data` then points to a vnode structure.

8-38. SLIDE: The **vnode** Structure



Student Notes

The **vnode** is at the center of all file activity within the kernel. There will be **one vnode** for each file that is currently being used. It is the heart of the systems ability to transparently use different types of file systems. Any part of the system accessing files at a higher level than the **vnode** is able to so on any file system without having to know the actual functions that are needed for that particular file system type. This is because as well as the **vnode** giving the file type it also has pointers to all of the functions that operate on the particular type of file.

The **vnode** and the **vfs** structures both use this object oriented technique to hide lower level implementation techniques from higher level access code.

Module 8
File Systems

The **vnode** structure is defined as.

```

struct vnode {
#ifdef _KERNEL
    u_short      v_flag;                /* vnode flags (see below)*/
    u_short      v_shlockc;            /* count of shared locks */
    u_short      v_exlockc;            /* count of exclusive locks */
    u_short      v_tcount;              /* private data for fs */
    int          v_count;               /* reference count */
    struct vfs    *v_vfsmountedhere;   /* ptr to vfs mounted here */
    struct vnodeops *v_op;              /* vnode operations */
    struct socket *v_socket;            /* UNIX ipc */
    struct sth_s  *v_stream;            /* stream head pointer */
    struct vfs    *v_vfsp;              /* ptr to vfs we are in */
    enum vtype    v_type;               /* vnode type */
    dev_t         v_rdev;               /* device (VCHR, VBLK) */
    caddr_t       v_data;               /* private data for fs */
    enum vfstype  v_fstype;             /* file system type*/
    struct vas     *v_vas;               /* vm data structures */
    vm_sema_t     v_lock;               /* vnode lock */
    struct buf    *v_cleanblkhd;        /* list of buffers for file */
    struct buf    *v_dirtyblkhd;
    int           v_writecount;         /* vnode write count */
    struct locklist *v_locklist;        /* FS-independent locking */
    int           v_scount;             /* soft hold count */
    int32_t       v_nodeid;             /* copy of va_nodeid */
    struct ncache *v_ncachedhd;         /* DNLC entries (as dp) */
    struct ncache *v_ncachevhd;        /* DNLC entries (as vp) */
    struct pfdat  *v_pfdathd;          /* link list on this vnode */
    unsigned      v_last_fsync;        /* most recent VOP_FSYNC */
#else /* not _KERNEL */
    int           v_opaque;
#endif /* not _KERNEL */
};

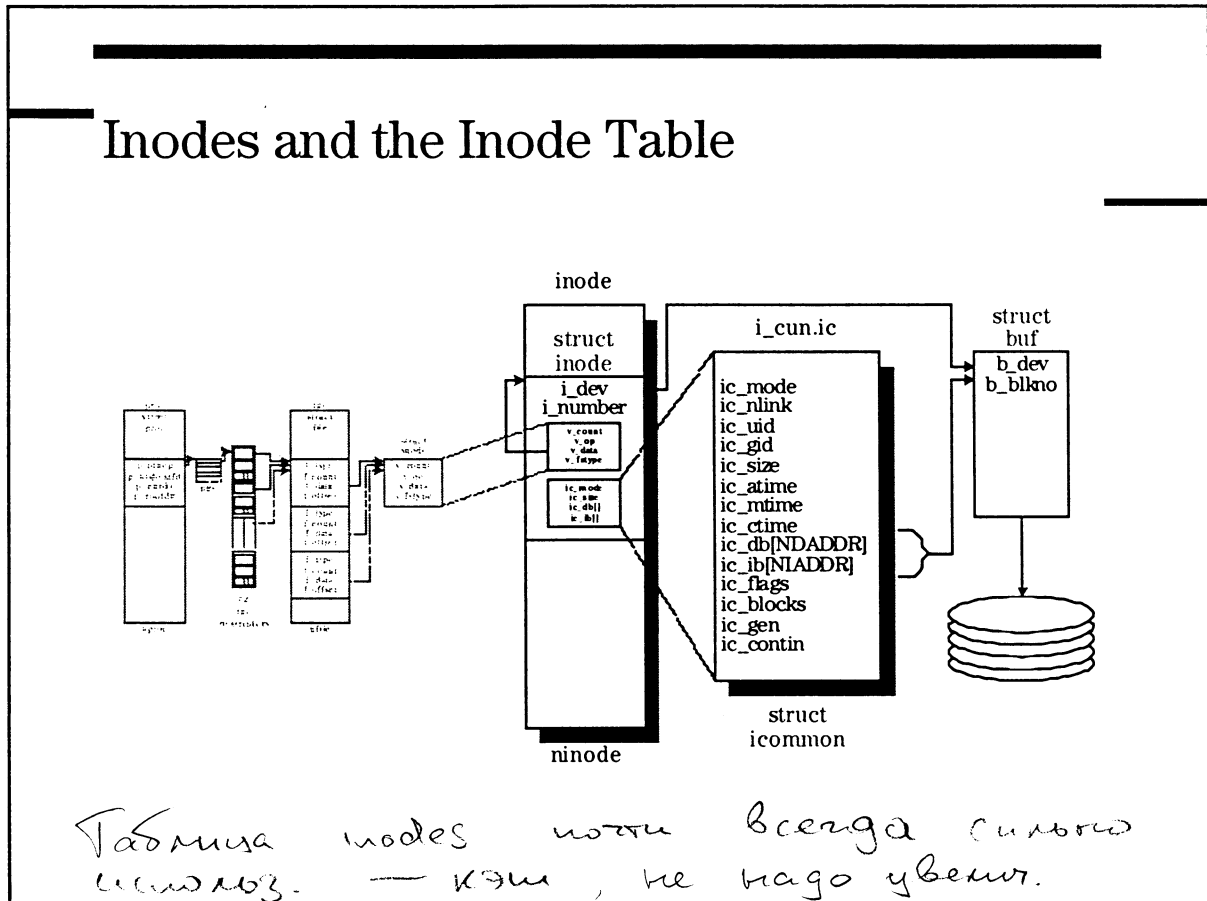
```

Some of the key fields are:

v_flag	Various flags including		
	VROOT	0x001	The root directory of a file system
	VMMF	0x100	A memory mapped file
v_count	The number of pointers pointing to this vnode, from the file table, dnlc , processes table, etc.		
v_vfsmountedhere	Where the vnode represents a mount point directory this field points to the vfs for the mounted file system.		
v_op	A pointer to the table of file system specific functions. (see below)		
v_type	The file type, like those found in the disk inodes, Regular file, Directory, Block device, etc.		
v_rdev	If this is a device file then the device number is stored here. For regular UFS files this field has the first disk block number from the inode, as this occupies the same position within the inode.		

v_data	The pointer to the file system specific data structure, such as the inode or mode. In these case the vnode actual physically resides within these structures.
v_fstype	The type of the file system. Unlike the VFS the type numbers in the vnode are not dynamically assigned at boot time, based upon driver installation orders. For the vnode the types are defined as an enumerated type with all the known type coded into it. While this has the problem of meaning that vnode.h must be modified to add new file system types, when looking at vnode from within Q4 the types are displayed symbolically.
v_vas	Where there is a virtual address space associated with the file, then the vnode will have a pseudo vas structure and a series of pregions . This field is used for memory mapped files and executing text files.
v_cleanblkhd v_dirtyblkhd	Pointer to a linked list of buffer cache headers for this file. The linked list is then formed from the b_blockf field of the buf structure. There are two separate lists one for clean buffers, those that have an up to date copy on the disk, and dirty buffers, those where the data needs to be flushed to disk.
v_writecount	The number of accesses to this file that are open for writing
v_locklist	The pointer to the linked list of file locks currently in place in this file. This field can be very useful when attempting to debug locking problems within applications. As there is no way for a process to find out what locks in currently hold. It is not even possible on HP-UX for a process to know what locks it was given when it asks for them!
v_nodeid	The inode number of the file.
v_ncachedhd	For a directory this pointer is to a linked list of ncache entries representing files in this directory.
v_ncachevhd	If this file is currently described in the dnlc then this field points to its list of ncache entries.
v_pfdathd	If there is an address space associated with this vnode, this field points to the linked list of pfdat entries for the pages currently in memory. This link list can be followed through the pf_vnext field.

8-39. SLIDE: Inodes and the Inode Table



Student Notes

Vnodes gives a file system independent view of the files, but the specific file system drivers also need to have information relevant to this particular type of file. For UFS files this data is held in an **inode** structure.

The **inode** structures within the kernel are held in a table pointed to by the value **inode** and sized by the kernel tuneable parameter **ninode**. This table is used to manage the UFS files only. There will be only one incore inode for any given file, so this table needs to be large enough to hold one entry for each UFS file that is in use at any particular time.

The incore inode also contains a copy of the disk inode (**icommon**) and since fetching any data from disk is so slow the kernel attempts to limit the number of times it needs to fetch new inodes from disk by caching recently accessed ones in the inode table. Thus increasing the size of the table may reduce the number of inode fetches from disk and help with the performance of opening UFS files.

In addition to containing a copy of the disk inode the incore inode structure hold much information that is only need by the system while the file is in use.

```

struct inode {
    struct inode *i_chain[2]; /* must be first */
    dev_t i_dev; /* device where inode resides */
    ino_t i_number; /* i number, 1-to-1 with device address */
    u_int i_flag;
    ushort i_lockword;
    tid_t i_tid; /* tid of last thread to lock this inode */
    struct vnode i_vnode; /* vnode associated with this inode */
    struct vnode *i_devvp; /* vnode for block i/o */
    int i_diroff; /* offset in dir, where we found last entry */
    struct inode *i_contip; /* pointer to the continuation inode */
    struct fs *i_fs; /* file sys associated with this inode */
    struct dqot *i_dquot; /* quota structure controlling this file */

/* Put the i_rdev here so the remote device stuff can change it
 * and still have the real device number around
 */
    dev_t i_rdev; /* if special, the device number */

    union {
        daddr_t if_lastr; /* last read (read-ahead) */
        struct socket *is_socket;
    } i_un;
    struct {
        struct inode *if_freef; /* free list forward */
        struct inode **if_freeb; /* free list back */
    } i_fr;
    struct i_select {
        struct proc *i_selp;
        short i_selflag;
    } i_fselr, i_fselw;
    struct mount *i_mount; /* mount table entry
 * note this can be calculated as:
 * (struct mount *)
 * (ITOV(ip)->v_vfsp->v_data)
 * but since this is a relatively
 * frequent operation in DUX, we
 * save it here to make it more
 * efficient.
 */

    union
    {
        struct icommon i_ic;
        struct icontr i_icc;
    } i_icun;
    ushort i_rcount; /* number of ilock_reader calls */
    b_sema_t i_b_sema;
    int filler[2]; /* pad to make hash work better */
};

```

Module 8
File Systems

Some of the key fields within this structure are:

i_chain[2]	Incore inodes are searched for through a hashing mechanism. These are the forward and backward pointers for the hash lists. These hash chains are discussed in the next section		
i_dev	Since inode numbers are only unique to a particular device the system needs to know which device this inode belongs.		
i_number	In the disk based inode table an entry does not need to know its number since that is just its position within the table. For the incore table though we need to know the inode number since not all the inodes are present, so we can not simply use the position within the table		
i_flag	The flags that give the current status of the inode. (see <code>inode.h</code> for full details)		
	IUPD	0x2	The file has been modified
	IACC	0x4	The inode access time has been changed
	ICHG	0x40	inode has been changed
	IREF	0x400	The file is in use
i_lockword	IACLEXISTS	0x2000	There is an access control list (ACL) continuation inode for this file
	To allow the system calls to act in an atomic fashion, the inode needs to be locked whilst IO's are being performed. This field holds the lock information flags.		
	ILOCKED	0x01	Exclusive access to this inode is required.
	ISHARED	0x02	During read operations a shared lock can be used, which allows other reads but not writes.
	IPAUSED	0x04	Under some situations it is desirable for an exclusive locker to temporarily allow read access, but to be able regain the exclusive lock without the risk of some other usage getting in first.
i_tid	IWANT	0x10	If another thread needs access to the inode, it will be sent to sleep (at priority PINOD, 138 from ps). It sets the IWANT flag to indicate that when the lock is removed, wakeup needs to be called to allow the other threads to return to run mode.
	Either the thread id of the locker of this inode, or		
	ITID_ICS	-3	There is no thread, its an interrupt service routine running.
	ITID_UNOWNED	-2	The inode is not locked.
	ITID_SHARED	-1	For a shared lock, several threads could be holding the lock. The <code>i_rcount</code> field is used to record the number of reading threads locking this inode.

i_vnode	For UFS the vnode is physically stored within the inode.
i_devvp	A pointer to the vnode for the device file this file system is mounted on.
i_diroff	Rather than always starting directory searches from the beginning UFS remembers where it last looked and can then start the next search from there.
i_contip	If ACL's are in use then this field points to the incore inode entry that holds the file continuation inode. In this case the file will use two entries in the incore inode table.
i_fs	Points to the copy of the superblock in the buffer cache. Since this is a short pointer in a 32bit kernel, this entry must reside in the part of the buffer cache mapped by the kernels SR6 register.
i_dquot	Pointer to quota information
i_rdev	If the file is a device file then this field holds the device number. It also holds the first block pointer for regular files.
i_fr.if_freef i_fr.if_freeb	If this inode is currently on the free list, these fields are the free list pointers.
i_mount	Pointer to the mount structure for the file system this file resides upon. The comment appears dated as DUX is not supported since HP-UX 10
i_icun.i_ic i_icun.i_icc	A union that holds either the disk inode, or if this entry represents a continuation inode, then the icont structure.
i_rcount	The number of threads currently holding a shared (read) lock on this inode.

As we have seen the incore inodes are held in a table. As with any table setting the appropriate size is important. If the table is made to large then space is wasted, since the table resides in the permanently memory resident part of the kernel. Alternatively if the table is too small then it will become impossible to access new files.

For the systems administrator the job of sizing the inode table is made more difficult by a number factors

- The inode table is used both to describe the currently in use UFS file, and also as a cache of recently accessed UFS files. So not only should the table be made large enough to handle the actual number of files that might be in use at anyone time, but also it benefits from having sufficient capacity to limit the number of inode look ups from disk.
- File table entries are not the only things that point to the inode table. Since the inodes are bound up with the vnodes, and vnodes are used for all kernel file access functions, there needs to exist a vnode for all file usages. These include: -

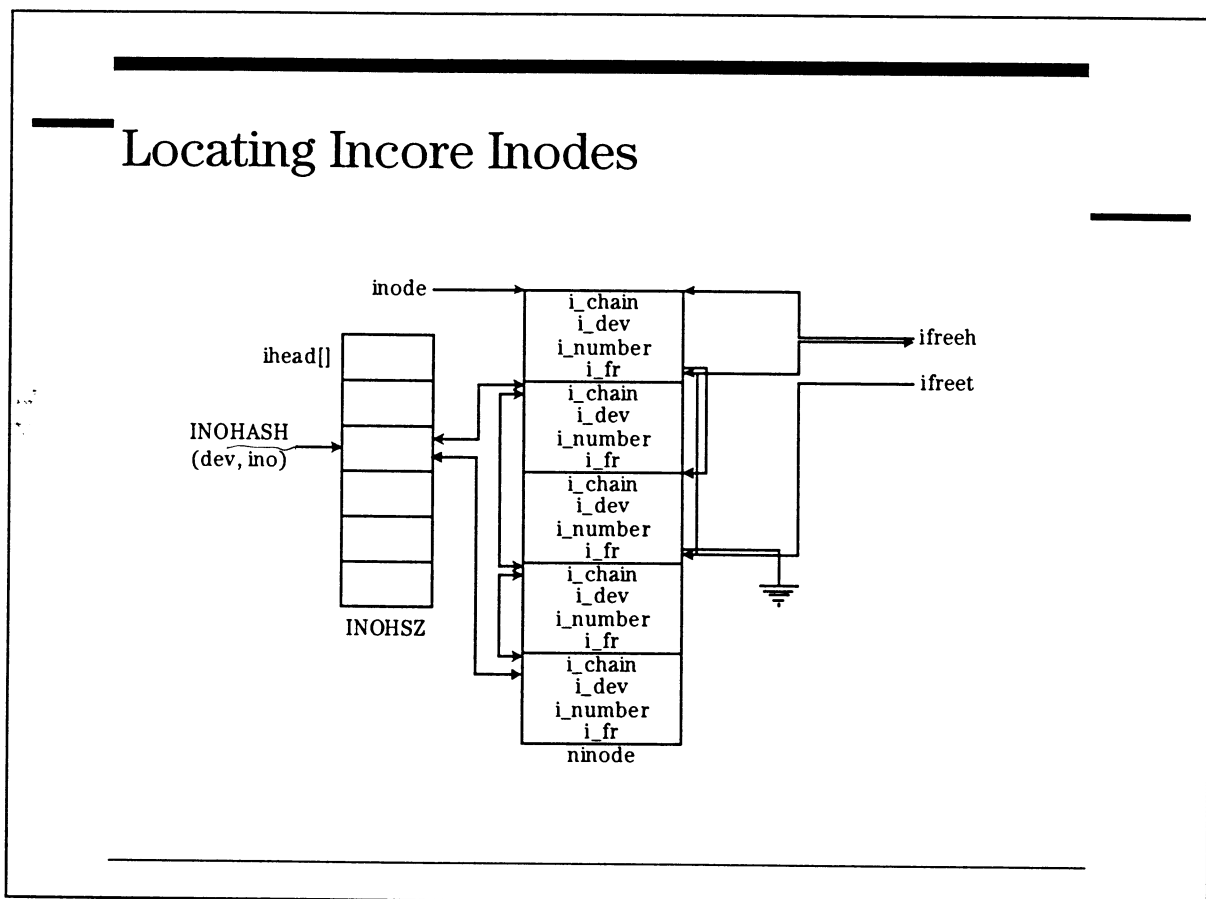
Open files	There will be a file table entry pointing to the vnode.
Current directories	Processes and threads point to the vnode for their current directory through their p_cdir and kt_cdir fields.
The system root directory	The kernel global roortdir points to the vnode for the systems root directory

Module 8
File Systems

Processes root directories	If a process or thread uses <code>chroot(2)</code> to change it's root directory then they will use their <code>p_rdir</code> and <code>kt_rdir</code> pointers to reference the vnode for their new root directory.
File systems root directories	The root directory of a mounted file system is referenced from the mount structure for UFS.
Mount point directories	Where a file system is mounted its <code>vfs</code> structure will point to a vnode for the mount point using the <code>vfs_vnodecovered</code> field.
Memory mapped and text files	These are referenced by the region structures as the front and back store locations.

- Should any of these be on a UFS file system then they need to be taken into account when calculating the correct value for **ninode**.
- In order to improve the performance of file name lookups the kernel caches names in the directory name lookup cache (DNLC). These entries reference the vnodes for the files and directories. Since these vnodes are referenced they are not free even when the file is not in use. Should more space be needed in the inode table, the DNLC is purged until the required space is found.
- The tools, **sar(1M)** and **glance(1)**, that report on the utilisation of the inode table do not report any meaningful data. The values they report for used inodes fluctuates without any changes occurring in the free inode list.

8-40. SLIDE: Locating Incore Inodes



Student Notes

As with most situations where the kernel needs to find data, particularly data structures, quickly it uses a hashing mechanism to locate entries in the inode table.

When trying to find an inode in the table we will have the device number and the inode number. These are hashed together using the `INOHASH` macro to obtain an index into the hash table `ihead`. The hash table is defined directly as an array from the `space.h` header file.

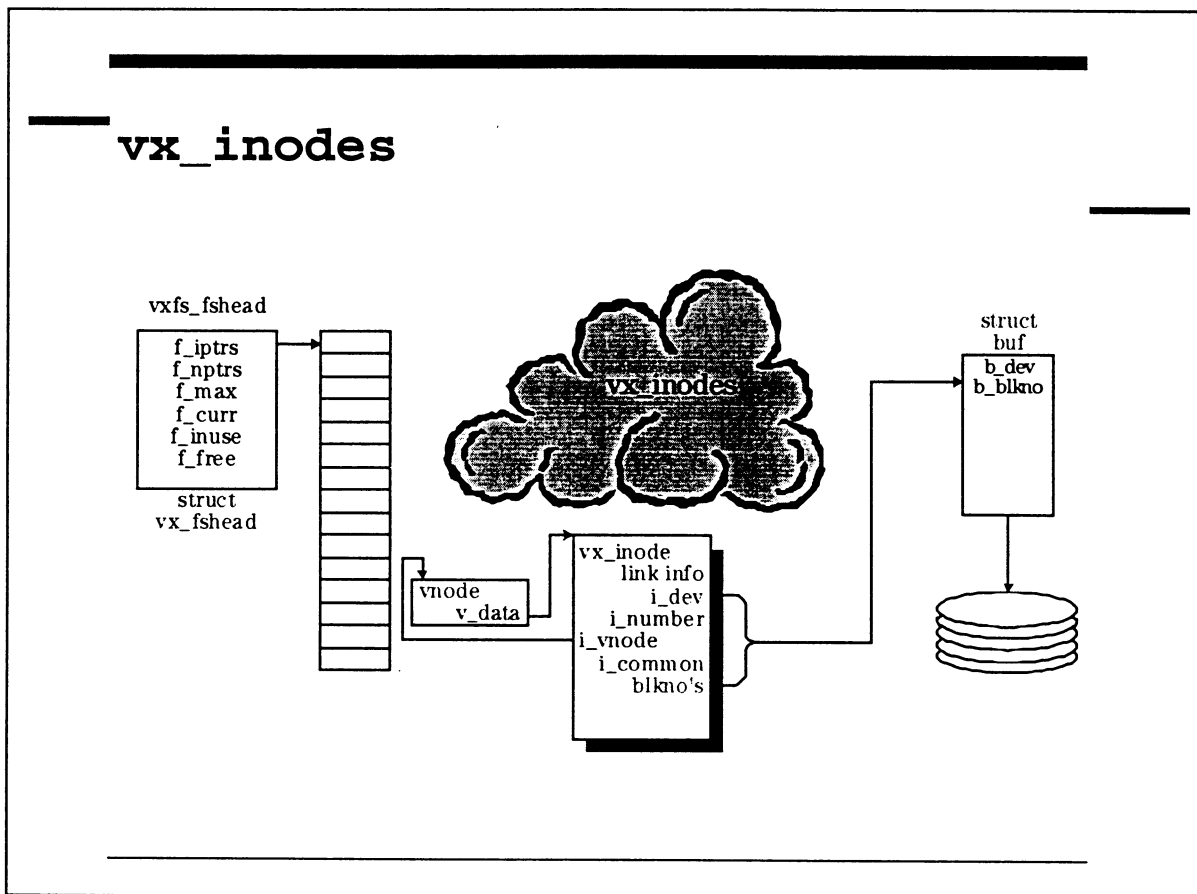
The entries in the hash table then point to the two ends of a linked list of inode table entries that may match our device and inode numbers. Since the hash table is not large enough to hold an entry for every unique combination of device and inode numbers, the first entry in the inode table may not match. So using the `i_chain` field of the inode we can walk the linked list until the correct entry is found.

If the inode is not found then it will need to be read in, and a free entry must be found. The variables `ifreeh` and `ifreet` point to the beginning and the end of the free inode list. The `i_fr.if_freef` and `i_fr.if_freeb` fields in the inode structure then form the linked lists. The forward list points to the inode structures but the backward list instead points to

Module 8
File Systems

the `i_fr.if_freeb` entries themselves, which tends to make them a little more difficult to use for walking the list in debuggers.

8-41.SLIDE: vx_inodes



Student Notes

The management of VxFS's incore file data structures centers around the **vxfs_fshead** structure. This points to a table of pointers to the incore **vx_inodes**. The first stage table is sized at boot time to allow plenty⁵ of space for **vx_inodes**.

The **vx_inodes** themselves are then dynamically created, so avoiding the problems of overflows that are experienced with traditional tables.

The array of pointers is sized according to the amount of RAM within the system, if **ninode** is larger than the chosen size, then **ninode** is used.

All of the inodes are accessed through a hash table similar to the UFS inode cache. The head of this hash table is of type **vx_hinode** and pointed to by a kernel variable of the same name (**vx_hinode**). There are actually two sets of hash chains in the structure. The first set of pointers are the forward and backward for the individual inode hash whose hash function is based on device, file set number, and inode number. The macro **VX_IHASH** defines the inode hash algorithm. The second set of pointers is the hash chain for the block hash. This hash is

⁵ On a system with 256MB of RAM this table can handle a target maximum of 16,000 inodes with space for a further 2,666 overflow **vx_inodes**.

Module 8
File Systems

based on device and inode list block number. The purpose of the block hash is to link together all the inodes in the same inode block. The macro `VX_BHASH` defines the block hash algorithm. This information is used by the allocation routines.

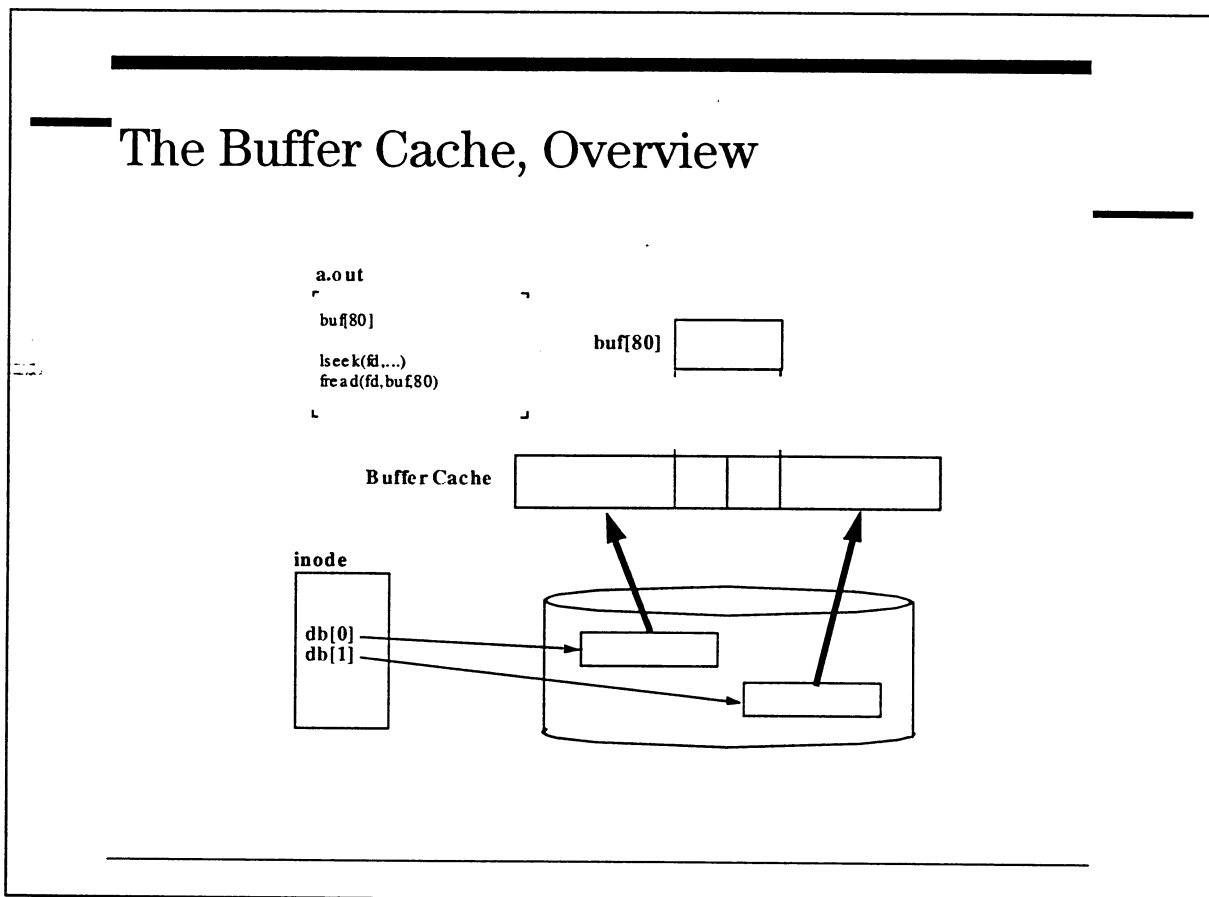
All inodes will appear on both of the inode hash chains contained in `vxfs_fshead`. Additionally, there are several different lists maintained by the inode cache.

List	Kernel Variable	Description
Free List	<code>vx_free_ilst</code>	List of free inodes
Dirty Page and Clean Inode List	<code>vx_dirty_inactlist</code>	List of inodes that have been inactivated that have dirty or clean pages associated with them
Attribute Free List	<code>vx_attrfree_ilst</code>	List of free attribute inodes

Each free/inactive inode will appear on only one of these free lists. The inode is linked into the list through the `av_forw` and `av_back` fields.

As with the UFS incore inodes, VxFS's internal inodes contain a copy of the disk inode and additional information that is only held in the memory based structure.

8-42. SLIDE: The Buffer Cache, Overview



Student Notes

We should now have an understanding of how data is stored in a file system on disk and some of the structures that the kernel maintains to keep track of this data. The kernel also needs a mechanism for accessing the data that will not require it to physically go to disk for each access.

The kernel is able to minimize the performance impact of the disk access through the buffer cache. Data is brought into the buffer cache in units of file system block size. The first access of a file system block will bring the block into the buffer cache. Subsequent access to data within the same block will only require the data be copied from the block in the buffer cache to the process' memory.

The most extreme example of where this would benefit performance is the case of sequentially reading through a file one byte at a time. Without the buffer cache, each request to read a byte from the file would result in a physical disk I/O. So, if our file system block size was 8K, this would result in 8K physical I/Os. With the buffer cache, the first request to read a byte would bring the entire file system block into the buffer cache. All the subsequent

reads would then be satisfied in the buffer cache (assuming conditions we will discuss later do not force the block out of the cache) and only one physical I/O would be needed.

Benefits/Disadvantages

In addition to the obvious performance benefit of the reduced number of disk I/O operations, the file system buffer cache provides many benefits for the kernel :

Recently/Frequently accessed data retained in memory

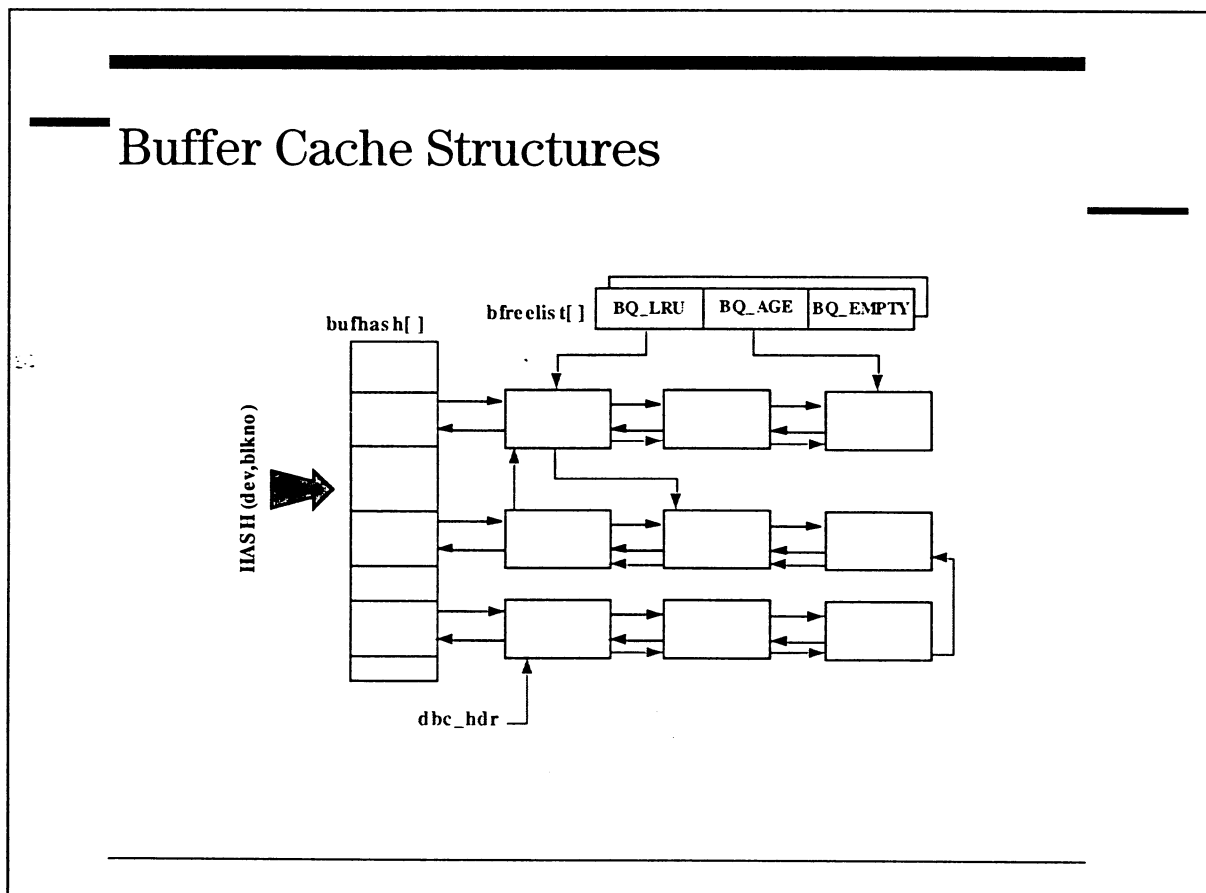
Provides read ahead capabilities

Writes completed asynchronously eliminating the need to wait for I/O completion

Reduced disk traffic since multiple writes to the same block may be committed to disk with a single disk I/O.

There is also one disadvantage to the buffer cache that needs to be recognized. Since writes to the buffer cache are held in memory until conditions occur that trigger a sync to disk, there exists the potential for data loss. In the event of a system failure, data in the buffer cache is not flushed to disk. So any writes not yet committed will be lost.

8-43. Buffer Cache Structures



Student Notes

The buffer cache is managed through a set of buffer headers and a hash chain. The `bufhash[]` array maintains a list of hash chain headers of type `bufhd`

```

struct bufhd
{
    int32_t b_flags;           /* see defines below */
    struct buf *b_forw, *b_back; /* fwd/bkwd pointer in
                                chain */
    int     b_checkdup;       /* modification "time" stamp */
    lock_t  *bh_lock;        /* lock for the hash chain */
};
    
```

The `b_forw` and `b_back` pointers create a circular list of buffers. The `bh_lock` is the spinlock associated with this hash chain that must be locked when any entry on the chain is being modified. Each buffer in the hash chain will share the same spinlock.

Buffers are hashed onto these chains through the device and block number. Each **bufhd** in the **bufhash[]** points to several buffers of type **buf**. Because of the size of this structure, it is shown at the end of this discussion rather than here in the middle.

Each buffer is linked onto one of the hash chains through the **b_forw** and **b_back** fields.

On a system with Dynamic Buffer Cache enabled, there is also a buffer pointed to by **dbc_hdr** that links all buffer headers together.

Free Lists

In addition to the hash chain, each buffer will additionally appear on a freelist. To reduce the potential for contention on the freelist lock there exists one freelist for each spu. Additionally, each freelist has three queues within it known as the **BQ_LRU**, **BQ_AGE**, and **BQ_EMPTY**.

The freelists are maintained in a two dimensional array called **bfreelist[]**. The first dimension of the array is the **spu** index and the second is one of the three queues. So, the entry for **BQ_AGE** on spu 3 would be

```
bfreelist[3][BQ_AGE]
```

Each entry in **bfreelist[]** is of type **buf** and the **av_forw** and **av_back** fields are used to record the head and tail of the queue.

The **BQ_AGE** and **BQ_LRU** queues attempt to isolate those buffers that are likely to be reused. Buffers are placed on the **BQ_AGE** queue when they are flagged as invalid (**B_INVALID**) or they have not been frequently accessed while in the buffer cache. If a buffer has been referenced more than **count_to_be_preferred** times while in the cache, then it is placed on the **BQ_LRU** queue.

Our algorithm for allocating buffers from the free list is to first search the **BQ_AGE** queue for our spu and then for other spus. If no buffers are found there only then do we take buffers from the **BQ_LRU** queues.

The **BQ_EMPTY** queue holds buffers that have no blocks associated with them. We never search the **BQ_EMPTY** queue but would sleep waiting on this queue if we request the allocation of new buffers through the dynamic buffer cache.

Buffer Pages

Each buffer header points to a buffer that holds the data from the file system block. The `b_addr` field in the header holds the virtual address of that buffer in memory. The full `buf` structure is shown below:

Key fields in the buffer header (`struct buf`) include:

`b_flags` : various buffer flags include:

```

#define B_WRITE      0x00000000    /* non-read pseudo-flag */
#define B_READ       0x00000001    /* read when I/O occurs */
#define B_DONE       0x00000002    /* transaction finished */
#define B_ERROR      0x00000004    /* transaction aborted */
#define B_BUSY       0x00000008    /* not on av_forw/back list */
#define B_PHYS       0x00000010    /* physical IO (may not be
                                     physio()) */
#define B_WANTED     0x00000040    /* issue wakeup when BUSY goes
                                     off */
#define B_ASYNC      0x00000100    /* don't wait for I/O
                                     completion */
#define B_PRIVATE    0x00001000    /* private, not part of
                                     buffers - LVM */
#define B_PFTIMEOUT  0x00004000    /* power failure time out-LVM*/
#define B_CACHE      0x00008000    /* did bread find us in the
                                     cache ? */
#define B_INVAL      0x00010000    /* does not contain valid info
                                     */
#define B_FSYSIO     0x00020000    /* buffer from b(read,write)*/
#define B_CALL       0x00040000    /* call b_iodone from iodone*/
#define B_NOCACHE    0x00080000    /* don't cache block when
                                     released */
#define B_RAW        0x00100000    /* raw interface
                                     (LVM/disc3/autoch) */
#define B_BCACHE     0x00200000    /* The buffer is from the
                                     buffer cache */
#define B_SYNC       0x00400000    /* buffer write is synchronous
                                     */
#define B_PAGEOUT    0x01000000    /* a buffer header, not a
                                     buffer */

```

`b_forw, b_back` : linked list for buffer hash chain.

`av_forw, av_back` : linked list for buffer free list if not `B_BUSY`.

`b_blockf, b_blockb` : linked list of buffers for a vnode's clean/dirty list.

`b_strategy` : strategy routine for this I/O.

`b_un.b_addr` : virtual offset of the actual buffer.

`b_dev` : major+minor device number.

`b_blkno` : block number on device

`b_bcount` : number of bytes for transfer.

*Запамяти LVM
на special
определенные параметры*

Module 8
File Systems

b2_flags : used to control how the buffer is handled in routines like `bread()`, `getblk()`, `brealloc()` and `getnewbuf()`, which will be discussed later.

0x0000	<code>BX_BUFWAIT</code>	pseudo-flag; wait for buf if locked
0x0001	<code>BX_NOBUFWAIT</code>	don't wait for locked buf
0x0002	<code>BX_INCORE</code>	only get buffer if incore
0x0004	<code>BX_NONBLOCK</code>	non blocking read for pageout
0x0008	<code>BX_READASYNC</code>	disowning read for bitmaps
0x0010	<code>BX_BUFCLEAR</code>	clear the buffer
0x0020	<code>BX_ANYSIZE</code>	don't check the size on the buffer
0x0040	<code>BX_MEMWAIT</code>	wait for memory
0x0080	<code>BX_NOMEMWAIT</code>	do not wait for memory
0x0100	<code>BX_PHYSICAL</code>	must have phys space attached (<code>getnewbuf</code>)
0x0200	<code>BX_PURGE</code>	purge from cache after IO (<code>blkflush</code>)
0x0400	<code>BX_FORCE</code>	do IO even if <code>NOFLUSH</code>
0x0800	<code>BX_READ</code>	Request is for a Read
0x1000	<code>BX_ANYSPACE</code>	<code>b_spaddr</code> need not be 0
0x2000	<code>BX_PUSHDIRTY</code>	don't clobber data in <code>brealloc1()</code>

b_error : error number returned, if any. `B_ERROR` flag should also be set in *b_flags* field.

b_bufsize : size of allocated buffer

b_iodone : function called by `iodone()` when I/O is complete.

b_resid : bytes not transferred after I/O is complete. *b_error* field should indicate why transfer did not complete.

b_bptype : buffer type. Valid values are:

b_qstart : queue start time.

b_iostart : I/O start time.

b_spaddr : space id associated with `b_un.b_addr`.

b_proc : process which initiated the physical or swap I/O.

b_vp : vnode associated with this buffer.

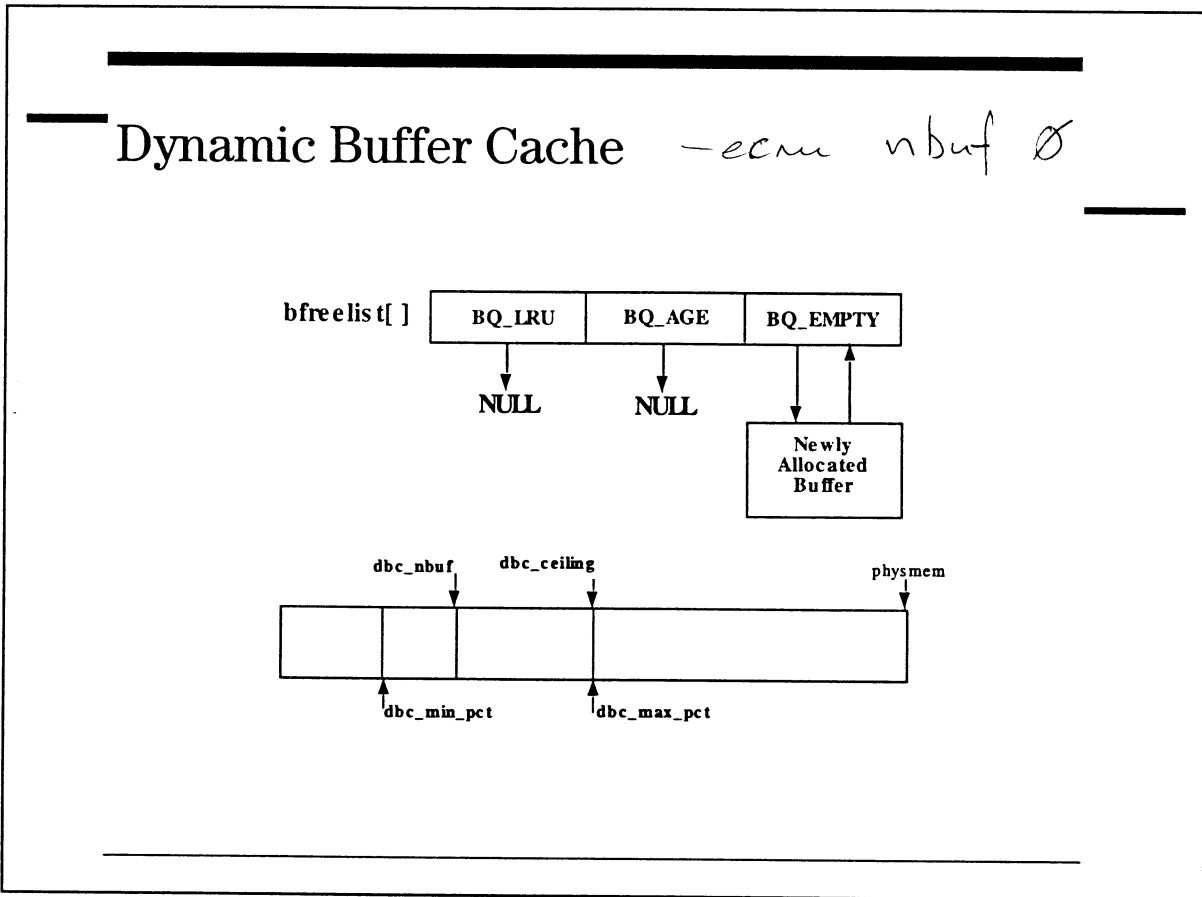
b_rp : region associated with this buffer.

b_nexthdr : linked list of all buffer headers.

b_timestamp : used to age and "steal" buffers. This will be discussed more in Module 11.

b_private : file system private buffer information.

8-44. SLIDE: Dynamic Buffer Cache



Student Notes

The buffer cache can be managed in one of two ways

- Fixed Buffer Cache
- Dynamic Buffer Cache

With the fixed buffer cache, the number of buffer headers and the number of pages allocated for the buffers is static based on user configuration.

In the case of the Dynamic Buffer Cache (DBC) the number of buffers and pages can increase or decrease in response to the level of usage.

There are several kernel parameters which affect the operation of the dynamic buffer cache

dbc_nbuf
dbc_bufpages
dbc_ceiling
dbc_min_pct

Minimum number of buffer headers
Minimum number of buffer pages
Maximum number of buffer pages
Minimum percentage of memory that can be allocated to the buffer cache.

*Монитор оуазабсе
монитор оуазабсе*

`dbc_max_pct` 50

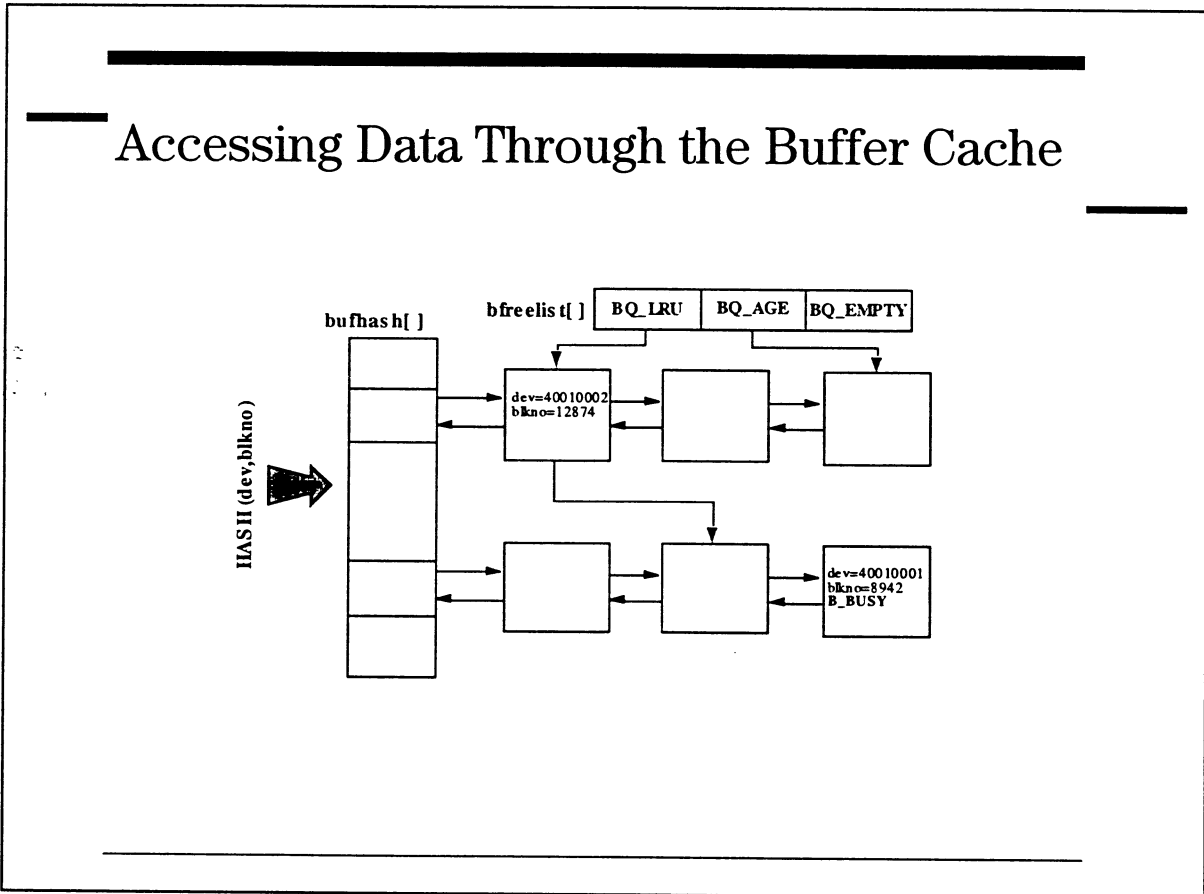
Maximum percentage of memory that can be allocated to the buffer cache.

Allocating buffers

In a fixed buffer cache environment, a process requesting a buffer that cannot find one on the BQ_AGE or BQ_LRU queues will sleep waiting for a buffer to be freed. In the case of the DBC, if there are no free buffers then we call `alloc_more_headers()` to allocate a new buffer and buffer header which is placed on the BQ_EMPTY queue. We then grab the new buffer from this queue and continue processing.

Pages in the buffer cache are aged and stolen through `vhand()`. As a result, dynamically allocated pages are returned to the virtual memory system.

8-45. SLIDE: Accessing Data Through the Buffer Cache



Student Notes

When a process requests data from a file through a system call such as `read()`, the kernel requests the data from the buffer cache through the `bread()` function.

When `bread()` is called, it attempts to locate a buffer that matches the requested device, block number, and vnode as the request. If none are found then a new buffer is allocated.

There are several scenarios that can result when searching for a given block in the buffer cache

Scenario	Results
Buffer found on the hash chain and is not busy.	Buffer marked busy and returned to requester
Buffer found on the hash chain but is busy.	Sleep waiting on buffer to be freed
Matching buffer not found on the hash chain.	Allocate a new buffer

The overall algorithm that **bread()** follows is to

Hash on device and block number and lock appropriate hash chain
Begin searching hash chain for buffer that matches request. If no matching entry found, release lock and call **getnewbuf()** to request new buffer

If a matching entry is found on the hash chain check to see if it is available (not set to B_BUSY). The action taken depends on whether the buffer is busy.

B_BUSY :

```
If set to NONBLOCK or NOBUFWAIT
    return NULL
else
    increment syswait[] .iowait
    sleep on buffer pointer
```

Not B_BUSY :

```
Remove buffer from free list
Mark buffer B_BUSY and B_CACHE
Increment access_cnt
set b_proc to make this process the owner
```

Allocating New Buffers

In the case that **bread()** does not find a buffer in the hash chain, then **getnewbuf()** is called to allocate a new one. The **getnewbuf()** routine will first search the head of the current spu BQ_AGE queue for an invalid entry. If a buffer does not meet this condition then we check the BQ_EMPTY queue to see if an empty buffer is available.

We check the BQ_EMPTY because when a new buffer is requested through the DBC we do not wait for the buffer to be allocated. The allocation is instead done in the background and the new buffer put on the BQ_EMPTY queue. It is therefore necessary for us to check this queue when requesting a new buffer.

If a buffer has not been found by these first two checks then we proceed to check the BQ_AGE queue for each **spu** and then the BQ_LRU queue for each.

If no free buffers are found anywhere then the process will sleep on **&bfreelist[0][0]**.

Reallocating Buffers

When allocating new buffers it is necessary to check existing buffers to make sure no data in the buffer cache overlaps the blocks we are about to bring in. In overlapping buffers are found then these buffers must be flushed to disk and marked as B_INVALID.

Any disk block can exist in only one buffer at any given time.

There are several buffer management routines in the kernel in addition to the ones we mentioned. The table below details some of these routines

bread()	Read disk block into buffer cache
breada()	Read disk block into buffer cache and request read-ahead.
bwrite()	Write buffer to disk waiting for completion
bdwrite()	Release a buffer and mark it for delayed write to disk.
bawrite()	Write buffer to disk without waiting for completion.
brlse()	Release buffer without performing any I/O.
brealloc()	Allocate a buffer in the buffer cache

4. Load the file structure for fd3 and display its contents:

```
q4> load struct file from <FD3>
```

```
q4> print -tx
```

What is the value of the "data" field? _____

This is a pointer to the vnode.

5. Load the vnode structure and display its contents:

```
q4> load struct vnode from f_data
```

```
q4> print -tx | more
```

What type of file system is this? _____

What is the value of the "data" field? _____

This is a pointer to the inode.

6. Load the inode structure and display the inode number and the device number:

For an HFS file system:

```
q4> load struct inode from v_data
```

For a VXFS fielsystem:

```
q4> load struct vx_inode from v_data
```

```
q4> print i_number i_dev%X
```

i_number = _____

i_dev = _____

Major number = _____ (first byte of i_dev), in decimal = _____
minor number = _____ (remaining three bytes)

7. In a new window, let's find out which file this is. First find out which file system has this device number:

```
# find /dev -exec ls -l {} \; | grep "< MAJOR# in dec >" | grep <MINOR#>
```

File system device file name = _____

Module 8
File Systems

8. Then find out the mount point for this file system:

bdf OR # mount

Mount point = _____

9. Finally, find the file within that file system:

find /<MOUNTPOINT> -inum <INODE#> -xdev

File name = _____

Is this the file you did the "more" on? _____

If not, repeat steps 4 through 9 for fd4.

8-47. Lab: VxFS Structures

1. Execute the script "vxfs_setup" in "/usr/local/bin" to create an VxFS file system, mount it, and populate it with a variety of files. See the script on the last page for the steps to accomplish all this. It will help answer several of the questions in this exercise.

```
# /usr/local/bin/vxfs_setup
```

2. Dump the contents of the root directory:

```
# xd -c /vxfs | more
```

Is there a file called "."? _____

Is there a file called ".."? _____

3. Display a long listing of all the files in the root directory with inode numbers and note the following:

```
# ll -ia /vxfs
```

Is there a file called "."? _____

Is there a file called ".."? _____

Why didn't they show up in the dump output? _____

4. In a separate window, invoke the "vxfs" version of "fsdb" using the interactive editor "ied":

```
# ied -h $HOME/.fsdb_vxfs_hist -p "fsdb_vxfs>" fsdb -F vxfs \ /dev/vg00/vxfsvol
```

Module 8
File Systems

5. Display the superblock information for this file system and note the following values:

fsdb_vxfs> 8 b;p S

version (version of the VxFS file system)	_____
size (size of complete file system)	_____
dsize (size of data area)	_____
bsize (block size)	_____
logstart (start of intent log)	_____
logend (end of intent log)	_____
nau (number of allocation units)	_____
aublocks (data blocks per allocation unit)	_____
aulen (blocks per cylinder group)	_____
oltsize (size of Object Location Table)	_____

6. Display the root inode:

fsdb_vxfs> 2i

What type of inode is this? _____

How many extents does this directory take up? _____

What is the size of the first extent? _____

What is the number of the first block in the extent? _____

7. Display the list of files in the root directory:

fsdb_vxfs> a0; p db

What is the inode number of the "adir1" directory? _____

8. Display the inode for that directory:

fsdb_vxfs> 8i

Compare the format of this inode to that of the root directory's inode. Are they the same?

9. Display the immediate field of this directory:

```
fsdb_vxfs> im; p db
```

How many files are listed? _____

10. List the attributes of the current fileset:

```
fsdb_vxfs> fset
```

fsindex (fileset number) _____

fsname (fileset name) _____

11. List the available filesets:

```
fsdb_vxfs> listfset
```

How many are there? _____

12. Change to the Structural Fileset:

```
fsdb_vxfs> 1 fset
```

fsname (fileset name) _____

13. Display the contents of the OLT:

```
fsdb_vxfs> olt
```

How many entries are there? _____

What is the inode number of the first FileSet Header? _____

In addition to the type, size and inode fields, what other fields are in the superblock olt entry?

Module 8
File Systems

14. Print out the inode for the fileset header of this fileset:

```
fsdb_vxfs> 3i
```

What type of inode is this? _____

How many extents does this file take up? _____

What is the size of the first extent? _____

What is the number of the first block in the extent? _____

15. Display the Current Usage Table:

```
fsdb_vxfs> cut
```

How many blocks are currently in use? _____

Extra Credit:

16. Dump the contents of the intent log to a file:

```
fsdb_vxfs> q  
# fsdb -F vxfs /dev/vg00/rvxfsvol > fnt.log  
fntlog  
q
```

17. Display the contents of the intent log:

```
# more fnt.log
```

Compare the contents of the intent log file with the contents of the setup script. Can you identify the functions in the intent log that correspond to the creation of the first file "afile"?

Which id number was assigned to this transaction? _____

How many functions were executed to create the file? _____

Were any data blocks allocated? _____

Which inode number was assigned to the file? _____

Script for vxfs_setup:

```
#!/usr/bin/sh

lvcreate -L 12 -n vxfsvol /dev/vg00

newfs -F vxfs /dev/vg00/rvxfsvol

mkdir /vxfs

mount /dev/vg00/vxfsvol /vxfs

touch /vxfs/afile /vxfs/rmfile1 /vxfs/rmfile2

chmod 7777 /vxfs/afile

echo "This file is not empty" >> /vxfs/notempty

mkdir /vxfs/adir1 /vxfs/adir2

echo "hello" >> /vxfs/afile2

echo "some stuff" >> /vxfs/adir1/file1

echo "some stuff" >> /vxfs/adir2/file1

echo "some more stuff" >> /vxfs/adir1/file2

echo "some more stuff" >> /vxfs/adir2/file2

ln /vxfs/notempty /vxfs/name2

ln -s /vxfs/notempty /vxfs/name3

mknod /vxfs/device c 64 0x010000

rm /vxfs/rmfile1 /vxfs/rmfile2

touch /vxfs/newfile
```

8-48. LAB: VxFS

- 1) Find the inode numbers of deleted files on vxfs.

Using the example program filehider in /home/tops/labs/mod8 to make some files work out the inode numbers of the deleted files.

Create a new vxfs file system and mount it.

Run filehider and pass it the new mount point directory as an argument.

Sync the file system and then using fsdb locate the open but removed files.

By looking in the fileset header for fileset 999 you can get the inode allocation unit inode, and also the number of allocated inodes.

Switch to fset 1 and read that inode, get the block numbers and switch back to fset 999.

The inode allocation unit uses 4 block extents, the last of which is the extended inode operations map, there is 1 bit for each inode allocated. A 1 indicates that there is a pending extended operation. By reading the inode, check if the link count is now zero.

```
root@tiger[] fsdb -F vxfs /dev/vg00/tmp
>
```

We need to know the inode allocation unit inode for fset 999 and also the number of inodes in the fileset

```
> 999 fset
fset header structure at 0x0000000a.0000
fsh_fsindex 999  fsh_volname "UNNAMED"
fsh_version 2  fsh_checksum 0x36c00741  fsh_time Fri Jan 29 03:48:35 1999
fsh_ninode 1056  fsh_nau 1  fsh_old_ilesize 0
fsh_fsextop 0x0  fsh_dflags 0x1  fsh_quota 0  fsh_maxinode 4294967295
fsh_ilistino[65 97]  fsh_iauino 64  fsh_lctino 0
fsh_uquotino 69  fsh_gquotino 0
fsh_attr_ninode 0  fsh_attr_nau 0
fsh_attr_ilistino[67 99]  fsh_attr_iauino 66  fsh_attr_lctino 68
fsh_features 0  fsh_fsetid 0x0 00000001
clone fsh_next 0x0 00000000  fsh_prev 0x0 00000000
fsh_volid 0x0 00000000  fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000  fsh_create 0x0 00000000
fsh_backup 0x0 00000000  fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000  fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000  fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000  fsh_vislim 0x0 00000000
fsh_states 0x0  fsh_status ""
>
```

so the inode allocation inode is 64
and there are 1056 inodes allocated (so the maps will need 1056/32 33 words)

Now move to fset 1 so we can look at it.

```
> 1 fset
fset header structure at 0x00000009.0000
fsh_fsindex 1 fsh_volname "STRUCTURAL"
fsh_version 2 fsh_checksum 0x36b43b48 fsh_time Fri Jan 29 03:47:01 1999
fsh_ninode 128 fsh_nau 1 fsh_old_ilesize 0
fsh_fsextop 0x0 fsh_dflags 0xa fsh_quota 0 fsh_maxinode 4294967295
fsh_ilstino[5 37] fsh_iauino 4 fsh_lctino 0
fsh_uquotino 0 fsh_gquotino 0
fsh_attr_ninode 0 fsh_attr_nau 0
fsh_attr_ilstino[0 0] fsh_attr_iauino 0 fsh_attr_lctino 0
fsh_features 0 fsh_fsetid 0x0 00000000
clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000
fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000
fsh_backup 0x0 00000000 fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000
fsh_states 0x0 fsh_status ""
>
```

```
> 64i
inode structure at 0x00000018.0000
type IFIAU mode 6000000777 nlink 1 uid 0 gid 0 size 4096
atime 917581621 199512 mtime 917581621 199512 ctime 917581621 199512
aflags 0 orgtype 1 eopflags 0 eopdata 0
fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 0
blocks 4 gen 0 version 0 0 iattrino 0 noverlay 0
de: 36 0 0 0 0 0 0 0 0 0
des: 4 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
```

Now that we know the block numbers, switch back to fset 999 so we can read it's inodes.

```
> 999 fset
fset header structure at 0x0000000a.0000
fsh_fsindex 999 fsh_volname "UNNAMED"
fsh_version 2 fsh_checksum 0x36c00741 fsh_time Fri Jan 29 03:48:35 1999
fsh_ninode 1056 fsh_nau 1 fsh_old_ilesize 0
fsh_fsextop 0x0 fsh_dflags 0x1 fsh_quota 0 fsh_maxinode 4294967295
fsh_ilstino[65 97] fsh_iauino 64 fsh_lctino 0
fsh_uquotino 69 fsh_gquotino 0
fsh_attr_ninode 0 fsh_attr_nau 0
fsh_attr_ilstino[67 99] fsh_attr_iauino 66 fsh_attr_lctino 68
fsh_features 0 fsh_fsetid 0x0 00000001
clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000
fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000
fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000
fsh_backup 0x0 00000000 fsh_copy 0x0 00000000
fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000
fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000
fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000
fsh_states 0x0 fsh_status ""
```

Module 8
File Systems

>

read the inode allocation unit header from block 36

```
> 36b ; p IA
iau header at 0x00000024.0000
magic a505fcf5  fsindx  999 aun 0
au_iextop 1  au_rifree 27  au_ribfree 3
>
```

There are extended inode operations pending.

```
> 38b ; p 33xW
00000026.0000: 00000000 00000000 00000000 00000000
00000026.0010: 00000000 00000000 00000000 00000000
00000026.0020: 00000000 00000000 00000000 00000000
00000026.0030: 00000000 00000000 00000000 00000000
00000026.0040: 00000000 00000000 00000000 00000000
00000026.0050: 00000000 00000000 00000000 00000000
00000026.0060: 00000000 00000000 00000000 00000000
00000026.0070: 00000000 00000000 0001ffff ffffffff
00000026.0080: ffffffff
```

>

```
> 39b ; p 33xW
00000027.0000: 00000000 00000010 00000100 00020040
00000027.0010: 00000800 02000002 30000004 10400000
00000027.0020: 40001008 00000000 00000800 00000000
00000027.0030: 00000000 00000000 00000000 00000000
00000027.0040: 00000000 00000000 00000000 00000000
00000027.0050: 00000000 00000000 00000000 00000000
00000027.0060: 00000000 00000000 00000000 00000000
00000027.0070: 00000000 00000000 00000000 00000000
00000027.0080: 00000000
```

>

Then just count up the bits. So from the first row we get inodes
59,87,110 & 121

```
> 59i
inode structure at 0x00000586.0300
type IFREG mode 100666 nlink 0 uid 101 gid 20 size 62783488
atime 917581714 140010 mtime 917581717 510017 ctime 917581717 510017
aflags 0 orgtype 1 eopflags 9 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 62624 gen 0 version 0 13431 iattrino 0 noverlay 0
de:  9056 98304 65536 32768    0    0    0    0    0    0
des: 23712 4096 32768 2048    0    0    0    0    0    0
ie:   0    0    0
ies:  0
```

>

```
> 87i
inode structure at 0x0000058d.0300
type IFREG mode 100666 nlink 0 uid 101 gid 20 size 0
atime 917581714 170013 mtime 917581714 170013 ctime 917581714 170013
aflags 0 orgtype 1 eopflags 1 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
```

```
blocks 0 gen 0 version 0 2110 iattrino 0 noverlay 0
de: 0 0 0 0 0 0 0 0 0 0
des: 0 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
> 110i
inode structure at 0x00000593.0200
type IFREG mode 100666 nlink 0 uid 101 gid 20 size 0
atime 917581714 220003 mtime 917581714 220003 ctime 917581714 220003
aflags 0 orgtype 1 eopflags 1 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 0 version 0 2155 iattrino 0 noverlay 0
de: 0 0 0 0 0 0 0 0 0 0
des: 0 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
> 121i
inode structure at 0x00000596.0100
type IFREG mode 100666 nlink 0 uid 101 gid 20 size 0
atime 917581714 220047 mtime 917581714 220047 ctime 917581714 220047
aflags 0 orgtype 1 eopflags 1 eopdata 0
fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
blocks 0 gen 0 version 0 2144 iattrino 0 noverlay 0
de: 0 0 0 0 0 0 0 0 0 0
des: 0 0 0 0 0 0 0 0 0 0
ie: 0 0 0
ies: 0
>
```

Module 8
File Systems

- 2) Reading the history your history file out of the buffer cache.
One of the nice features of ksh or posix-sh is that they allow the use of a history file to remember the commands we have used to provide the ability to recall and edit them. Make sure that your shell is configured to be using a history file and then using Q4 find it within the buffer cache.

From the process table entry view the field p_highestfd, this tells us what the largest file descriptor open on the process is.

Using the description of file descriptors in the module, find the pointer to the file table for the history file. It will probably be 29, but you can check them with glance.

From there load the file table entry and follow this through to the vnode. The vnode has a pointer to a linked list of buf structures for the file.

From the buf structures print out the b_spaddr and b_un.b_addr fields (in hex) and then view what they point to.

It is normally easier to read unstructured information like your history file in adb.

Use

```
adb -k /stand/vmunix /dev/mem  
space.offset/s
```

then just keep hitting return

```
q4>  
q4> load struct proc from proc max nproc  
loaded 276 struct proc's as an array (stopped by max count)  
q4>  
q4>  
q4> keep p_stat && p_pid == 1886  
kept 1 of 276 struct proc's, discarded 275  
q4> print p_highestfd  
p_highestfd  
29
```

fd 29 is actually the shells history file, you can work these out using glance.

with a 32bit kernel you could use

```
q4> examine *p_ofilep using 90X  
but on this 64bit one you need to use
```

```
q4> examine p_ofilep using 2X  
0x1 0xc02600
```

```
q4> examine 0x100c02600 using 120X  
0 0xe77ca8 0 0 0 0xe77ca8 0 0 0 0xe77ca8 0 0 0 0xe792a8 0 0x1 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0xe79930 0 0x1 0 0 0 0 0 0xe77d58 0 0x5
```



```

q4>
q4>
q4> load struct file from 0xe77d58
loaded 1 struct file as an array (stopped by max count)
q4> print -tx
    indexof  0
    mapped   0x1
    spaceof  0
    addrof   0xe77d58
    physaddrof 0xe77d58
    realmode 0
    f_flag   0xb
    f_type   0x1
    f_msgcount 0
    f_count  0x1
    f_ops    0x6f3658
    f_data   0x1007c8000
    f_freeindex 0xbc
    f_offset 0x21ea
    f_buf    0
    f_cred   0x100a19500
    f_b_sema.b_lock 0
    f_b_sema.order 0x32
    f_b_sema.owner 0
    f_tap    0
q4>

```

Check the type, it should be 1 for a file.

```

q4>
q4> load struct vnode from f_data
loaded 1 struct vnode as an array (stopped by max count)
q4> print -tx
    indexof  0
    mapped   0x1
    spaceof  0
    addrof   0x1007c8000
    physaddrof 0x35c8000
    realmode 0
    v_flag   0x8008
    v_shlockc 0
    v_exlockc 0
    v_tcount 0
    v_count  0x2
    v_vfsmountedhere 0
    v_op     0x740c78
    v_socket 0
    v_stream 0
    v_vfsp   0x100743000
    v_type   VREG
    v_rdev   0
    v_data   0x1007c80b0
    v_fstype VVXFS
    v_vas    0
    v_lock.b_lock 0
    v_lock.order 0x5a
    v_lock.owner 0
    v_cleanblkhd 0x100356db0
    v_dirtyblkhd 0
    v_writecount 0x2
    v_locklist 0x7dcab0
    v_scount  0x1

```

Module 8 File Systems

```
    v_nodeid 0x274
    v_ncachedhd 0
    v_ncachevhd 0xea02e0
    v_pfdathd 0
    v_last_fsync 0
q4>
```

Now follow the linked list through the buffer cache.

```
q4> load struct buf from v_cleanblkhd next b_blockf max 100
loaded 3 struct buf's as a linked list (stopped by null pointer)
q4>
q4> print -x b_spaddr b_un.b_addr
    b_spaddr      b_un.b_addr
0xf478c00 0x8000000000746000
0xf478c00 0x80000000008c0000
0xf478c00 0x8000000001fc0000
q4>
q4> examine tospace(0xf478c00) | 0x8000000000746000 using s
er.c
```

This is data from the buffer cache, but in this case not too helpful

```
q4> examine tospace(0xf478c00) | 0x8000000000746000 using 20s
er.c
make filehider
./filehider /lab
./filehider /lab | more
vi filehider.c
make filehider
./filehider /lab | more
vi filehider.c
make filehider
./filehider /lab | more
ls
vi filehider.c
make filehider
./filehider /lab | more
bc
vi filehider.c
```

Using adb is easier in this case

```
q4> root@tigger[] ied adb -k /stand/vmunix /dev/mem
0xf478c00.8000000000746000/s
bzskipped:
bzskipped:      er.c

bzskipped+6:    make filehider

ra_zeromem+2:  ./filehider /lab

nfs_srvr_debug+8:      ./filehider /lab | more

nfs_rwl_order+1:
```

The output in front of the colon is adb trying to be helpful and interpret the address for you, unfortunately it is in the wrong space, so ignore it.

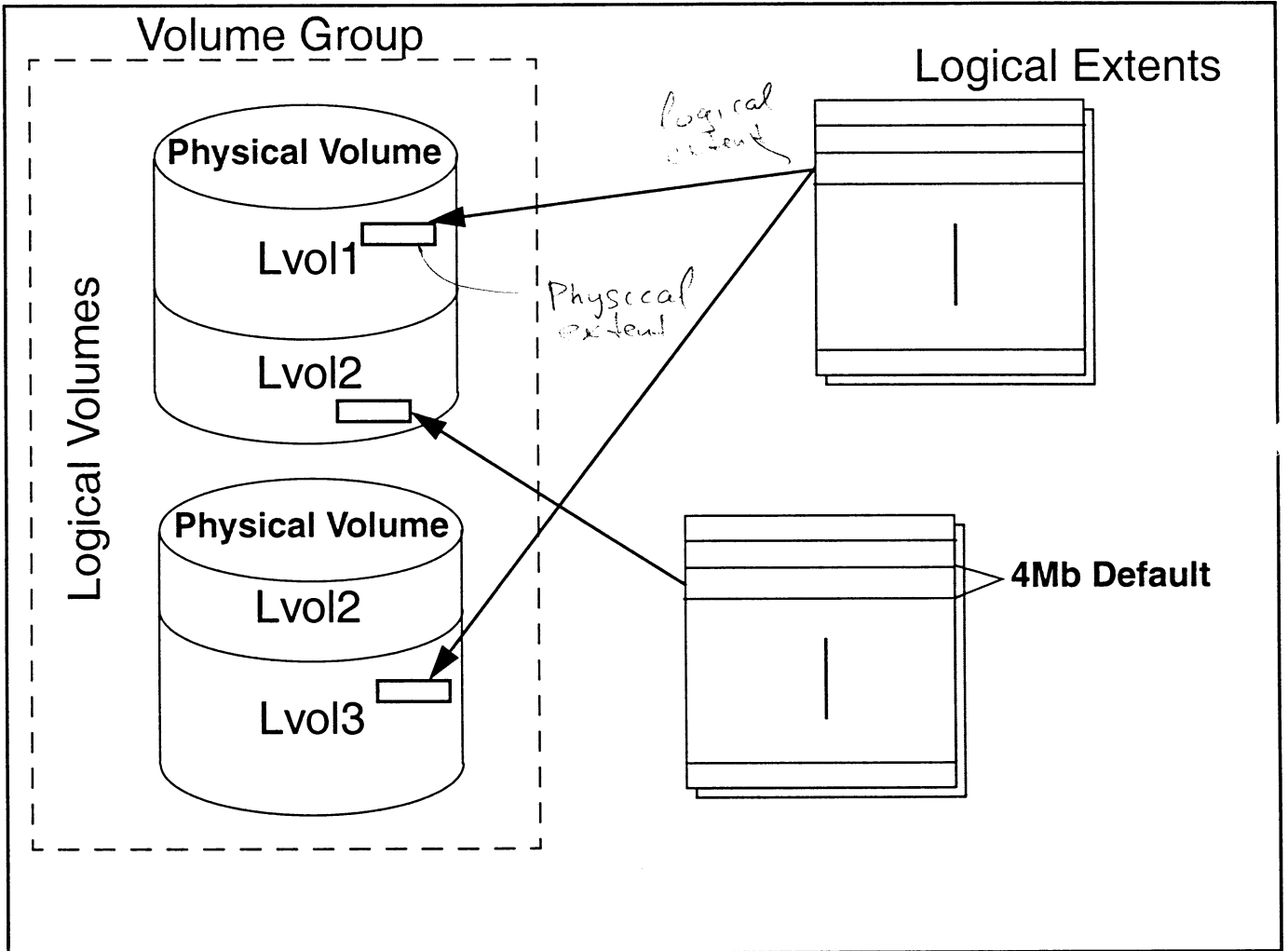
Module 9

Logical Volume Manager

Objectives :

- Understanding of LVM driver architecture.
- Familiarity with LVM disk and in-core structures.
- Understanding of how Mirror Write Consistency is maintained.
- Understand how LVM handles Bad Block Relocation.

Slide: LVM Overview



Notes:

Можно задавать mirroring
с помощью LVM — это уменьшит
размер lvol

Slide: LVM Overview

The **Logical Volume Manager (LVM)** is not a file system but rather a disk management subsystem that offers access to file systems as well as features such as disk mirroring, disk spanning, and dynamic partitioning. Since LVM is the preferred method of managing disks on HP-UX 10.0, a discussion of it is relevant to our study of file systems.

It is assumed that prior to taking this class you already have familiarity with LVM and its command interface. Our purpose here is to investigate the kernel level design for the various components of LVM, including algorithms and data structures.

While it is assumed that you already have working knowledge of LVM, we will define the basic terms used within LVM so that there is common understanding.

Physical Volume Physical device where data is stored

Physical Extent Set of physical sectors(blocks) contained within a single physical volume. A physical extent is a specific, contiguous region of the disk where data resides. A physical extent is restricted in size to a power of 2 between 1Mb and 256Mb

Logical Volume Virtual mapping of data to physical volumes or to areas within one or more physical volumes. A logical volume is a conceptual construct and not, in itself, a physical extent. Consequently, a logical volume can be conceptually viewed as a storage device without physical boundaries. It can be larger than any one physical device and can consist of several physical devices or portions of physical devices.

Logical Extent Refers to a set of virtual sectors (blocks) contained with a logical volume. Logical extents map to one, two, or three physical extents. If the logical extent maps to one physical extent, the stored data is not being mirrored. If the logical extent maps to two or three physical extents, the stored data is being mirrored.

Volume Group A set of physical and logical volumes and the data mapping that exists among them. A volume group can be thought of as a medium between the storage capabilities offered by physical volumes and the abstraction of the logical volumes. It is a storage repository in the sense that it groups together all of the physical extents of physical volumes that are part of the volume group itself. Membership in a specific volume group allows for mapping among the logical and physical volumes of which the group is comprised. Logical volumes cannot map to physical volumes that are members of different volume groups and each physical volume can belong to exactly one volume group.

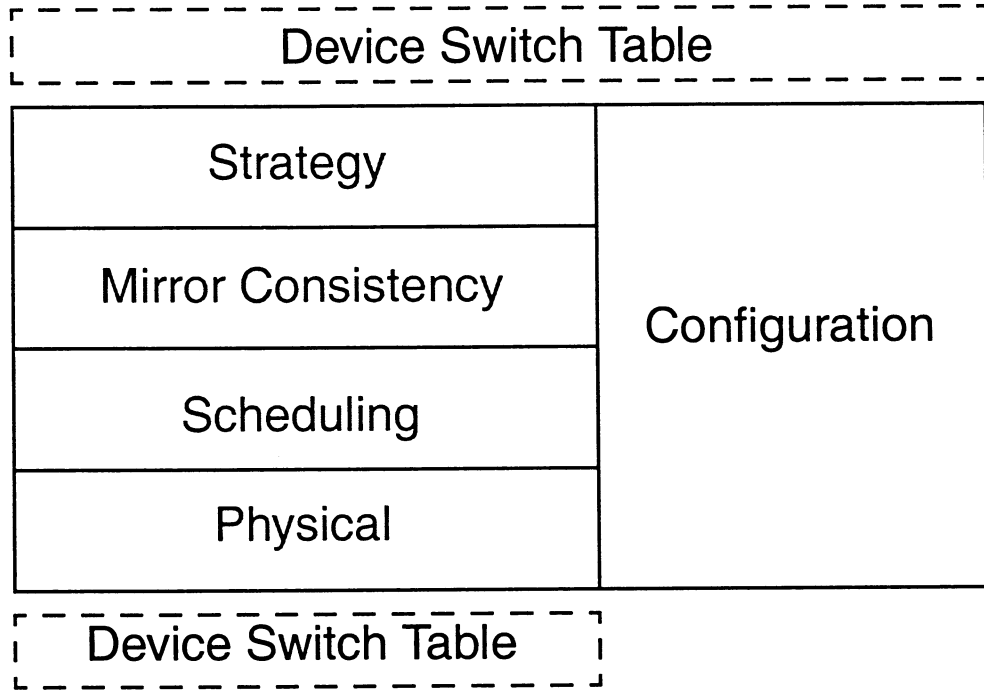
Mirrored Data Refers to copies of data stored in physical extents (blocks) that map to unique logical extents. Data can be single mirrored (one additional copy) or doubly mirrored (two additional copies). If the data is single mirrored, two physical extents are allocated for each logical extent.

Logical Track Group (LTG) The unit used to keep track of mirror consistency. Currently this is 256K.

ноз транзакций зеркалирование
есть на всех дисках внутри VG

Slide: LVM Architecture

LVM Architecture



Notes:

SWAP TOUKE KPICTYTHA MIRRORATE POLU
KPICTYTHA SECTYTHA

Slide: LVM Architecture

The kernel portion of LVM is broken up into five main sections as pictured in the slide. Normal I/O flows through the four layers shown by the boxes on the left hand side. Configuration is performed through various *ioctl()* calls and interface directly with the device driver.

Strategy Layer

The strategy layer guarantees that all I/O to a given block on disk is serialized. In addition, the strategy layer provides request validation, initialization and termination of requests.

The *lv_strategy()* routine is passed a *buf* structure to describe the request. Within the *buf* the following fields are expected to be valid

b_dev	Logical Volume Device Number
b_flags	Request Flags
b_blkno	Block number with logical volume
b_bcount	Byte count of request
b_options	Options passed through from raw I/O

Within the strategy layer, checks are made to make sure that the request is a valid size (multiple of DEV_BSIZE) and that the logical volume is open. If an error occurs in the strategy layer, the following fields are set in the *buf* and *biodone()* is called

b_error	Set to indicate the type of error
b_resid	Set to the number of bytes <i>not</i> transferred
b_flags	Sets B_ERROR bit

Mirror Consistency Layer

The Mirror Write Layer maintains data in the **Mirror Write Cache (MWC)**. If the logical volume is not mirrored (struct *lv* *lv_maxmirror* field is zero) or the request is a read or it is not using the MWC, this layer is skipped and the request is passed through to the scheduling layer (see macro LV_MWC_TEST()). In the case of a write to MWC mirrored LV, the mirror consistency layer maintains the data to ensure all the mirror copies of a given LTG are identical should the LV not be closed cleanly (e.g a system crash occurs). Mirrored LVs that do not use the MWC but do have mirror consistency recovery specified

Module 9 — Logical Volume Manager

Slide: LVM Architecture

(lvchange -M n -c y) have to synchronize all LEs under similar circumstances. Whereas one using the MWC only has to read and write the entries from the latest MWC.

The MWC is a per VG structure. It holds 32 entries (each being a different LTG). Before a write of a new LTG to a MWC mirrored LV can be progressed to the next layer, it has to be both recorded in the MWC and written to the disc based Mirror Consistency record (MCR). If the MWC is full then the LRU entry is overwritten.

Before a write to a mirrored logical track group, the “in transition” status of that track group is recorded in the volume group. The status is held MCR on the chosen physical volume.

Scheduling Layer

The scheduler converts the logical request into one or more physical requests and then schedules those requests through the physical layer on to the device drivers. The *lv_schedule()* routine accepts a buf pointer and places all the requests onto the physical buffer (pbuf) pending queue¹ and then calls *lv_reschedule()* to do the work of scheduling the request.

A logical volume has several different strategies it may follow for scheduling. The particular strategy to use is recorded in the logical volume structures to be discussed later. The strategy determines which scheduling routine to call as follows:

no garbage

0x0000	LVM_RESERVED	<i>lv_reserved(vg, lv, lb, pb)</i> Initiate I/O to the physical volume reserved area. This is only used to non-mirrored areas. This is the scheduling policy for lvol 0 in any group.
0x0001	LVM_SEQUENTIAL	<i>lv_sequential(vg, lv, lb, pb)</i> Initiates sequential mirrored physical operations.
0x0002	LVM_PARALLEL Merge magenta no stripe - no qmstr	<i>lv_parallel(vg, lv, lb, free_q)</i> Initiates parallel mirrored physical operations
0x0003	LVM_STRIPE	<i>lv_stripe(vg, lv, lb, free_q)</i> Initiates parallel striped operations

1. LVM request queues will be discussed on page 10-33

no refer to
SEQ - TOUO magenta
PAR - Merge magenta

Module 9 — Logical Volume Manager

Slide: LVM Architecture

Нотера урдузб.
- 1% бурик
- 3% MIRROR

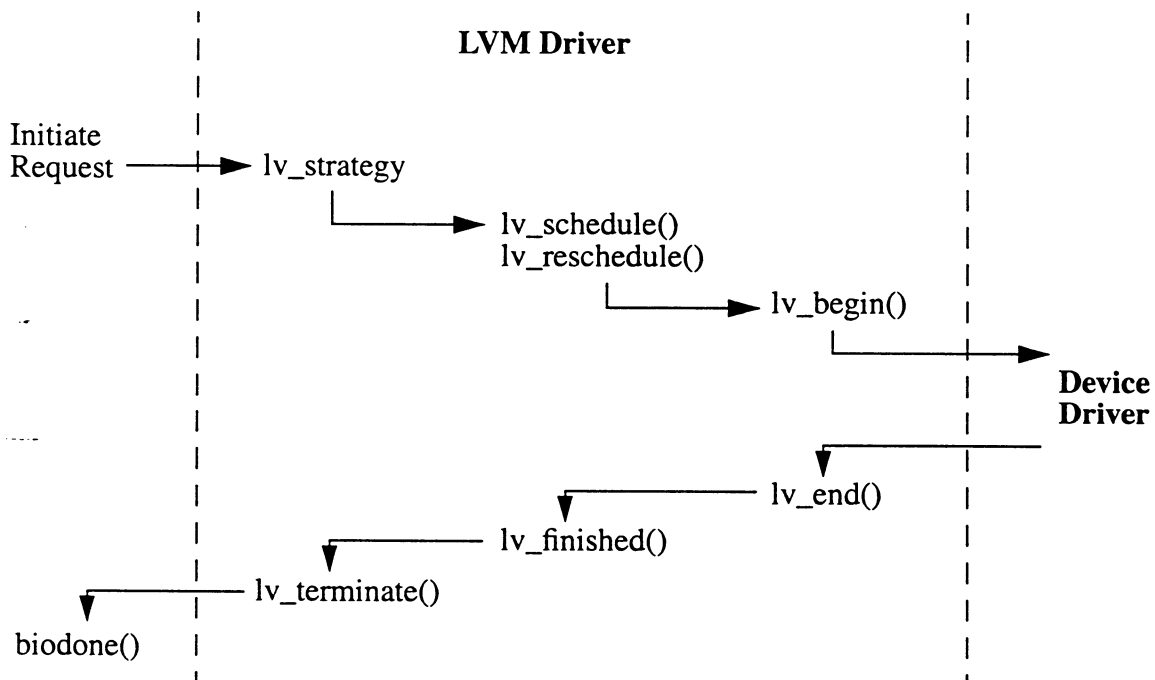
Physical Layer

The physical layer is the layer which converts the logical request to a physical request and calls the disk strategy routine.

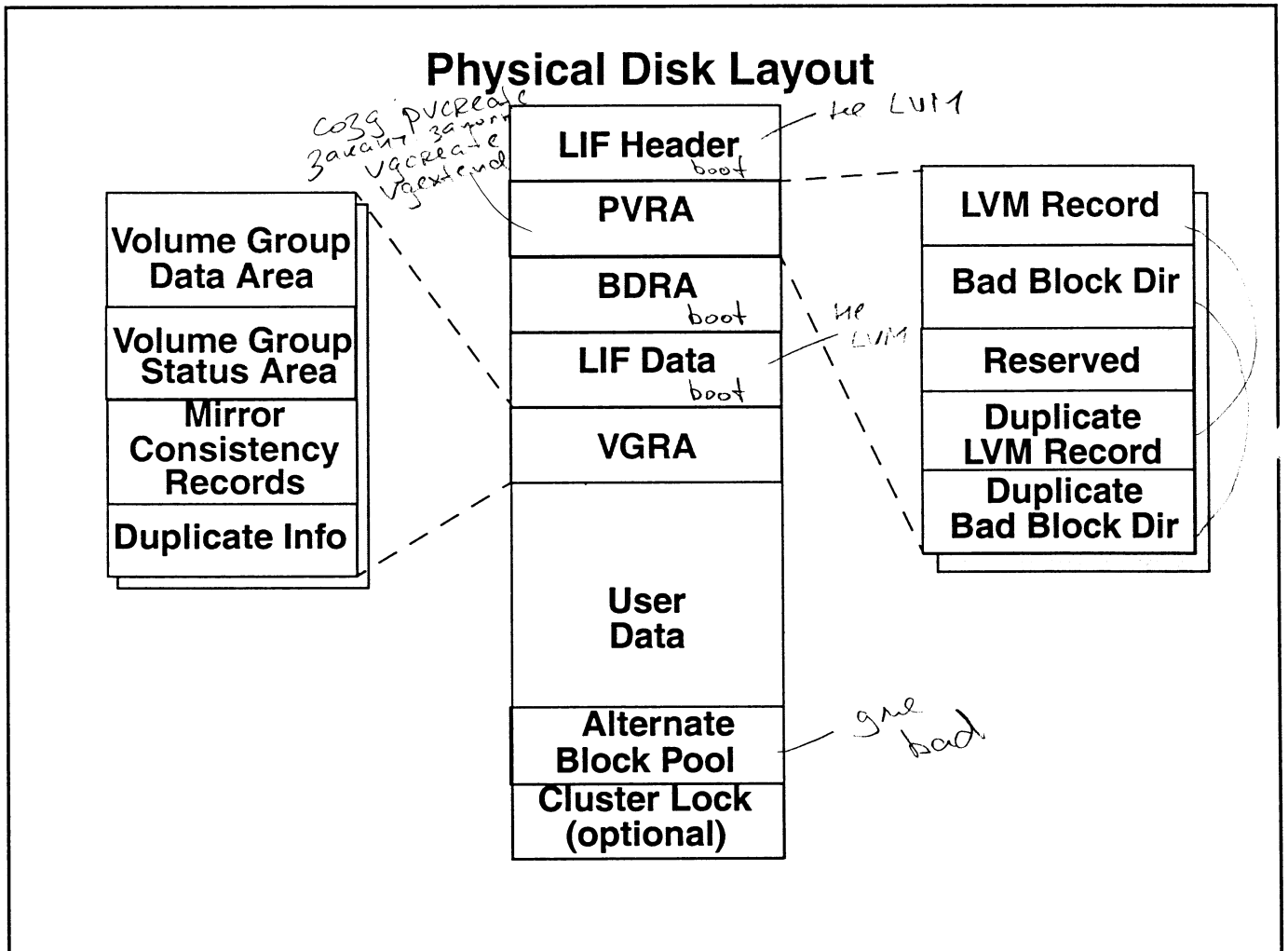
The `lv_begin()` routine is passed a physical buffer (pbuf) that describes the I/O to be performed.

Any write request to a volume group that is activated as read-only should be caught in a higher layer. If detected in the physical layer, this condition will result in a panic. It is also in this routine that we check for bad block relocation

A request that goes through the LVM layers will generally follows the sequence below



Slide: LVM Disk Layout: Bootable



Notes:

Лезко ybugers репер од -xc

bootable — торко pvcreate -B

Slide: LVM Disk Layout: Bootable

LVM Bootable Disk Layout

The slides show the general layout of the an LVM disk. A disk which has been prepared using `pvcreate -B` is termed a bootable disk. We'll discuss this layout first.

A bootable disk starts with the LIF header in the first 8K. The LIF header points to the LIF data which is held in a reserved space in the LVM structures.

The first LVM data on disk is the **Physical Volume Reserved Area (PVRA)**. From a user viewpoint the PVRA is normally discussed as an individual structure. The kernel, however, has no PVRA-specific data type and instead we discuss the PVRA as the area of disk that contains

- information describing this particular physical volume
- pointers to other LVM structures on the disk.
- the Bad Block Directory for the physical volume

Following the PVRA is the **Boot Data Reserved Area (BDRA)**. The BDRA contains information necessary to located the root, swap, and dump volumes during boot. This area is created with the command `pvcreate -B`.

The **Volume Group Reserved Area (VGRA)** contains the **Volume Group Descriptor Area (VGDA)**, the **Volume Group Status Area (VGSA)**, and Mirror Consistency Records.

The Volume Group Descriptor Area which has information about the volume group the disk belongs to, information about each logical volume on the disk, and the physical extent mapping.

The Volume Group Status Area has status contains information about missing PVols and stale extents.

The Mirror Consistency Records are for the Mirror Write Cache and will be discussed on page 10-37

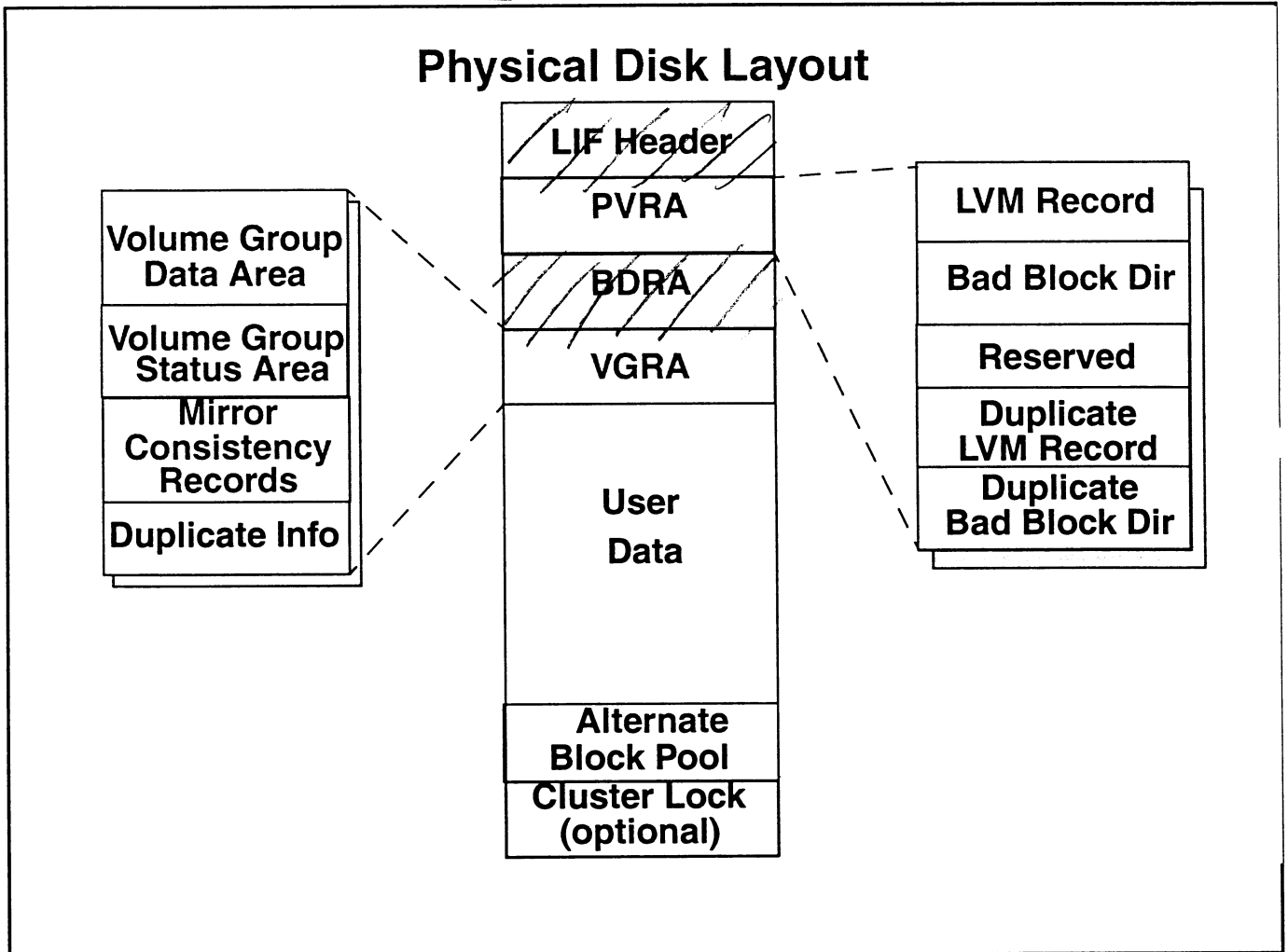
At the end of the disk is the **Alternate Block Pool** used for Bad Block Relocation. The details of this will be discussed on page 10-49.

The remaining area between the LVM structures and the bad block pool is the area of disk used for user data.

2K Block Size Support

In HP-UX 9.04, changes were made to LVM to support reading and writing I/O in 2K sizes. When a request is initiated, LVM determines the block size of the device and issues the I/O in that increment. The examples in this module assume the default 1K block/sector size.

Slide: LVM Disk Layout: Non-Bootable



Notes:

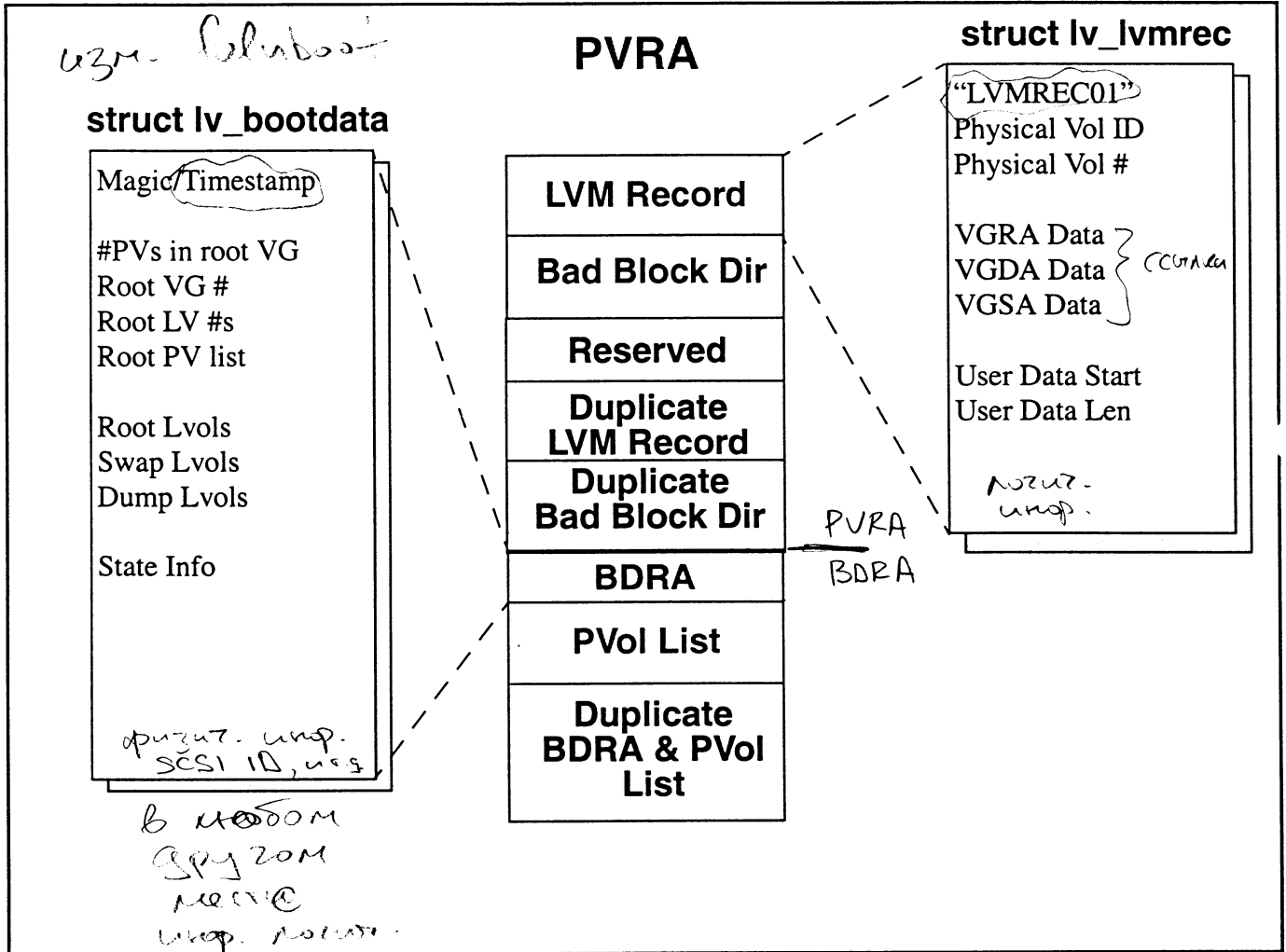
Slide: LVM Disk Layout: Non-Bootable

Non-Bootable Disk Layout

A non-bootable disk differs from that described for the bootable case in the following respects:

- The 8K at the front of the disk contains both LIF header and LIF data (i.e. enough space for the LABEL file should it be required)
- It has no BDRA

Slide: LVM Disk Structures: PVRA and BDRA



Notes:

При загрузке системы читается Root VG, где это хранится BDRA, где хранятся данные о физических дисках

VG ID — обязательно указывается! (но не в main.)

Slide: LVM Disk Structures: PVRA and BDRA

The PVRA contains a primary LVM record of type *lv_lvmrec* that describes the physical volume followed by the primary bad block directory. Secondary copies of both of these structures also exist. The table below shows the kernel definitions for each structure starting position.

	Kernel #define	Sector #
Primary LVM Record	PVRA_LVM_REC_SN1	8
Primary Bad Block Directory	PVRA_BBDIR_SN1	9 (10 if 2K device)
Secondary LVM Record	PVRA_LVM_REC_SN2	72
Secondary Bad Block Directory	PVRA_BBDIR_SN2	73 (74 if 2K device)

The kernel also defines the overall PVRA size (*PVRA_SIZE*) as 128 sectors and the Bad Block Directory length (*PVRA_BBDIR_LENGTH*) as 55 sectors.

The first field of the PVRA (*bd_magic*) is a magic ID for the structure. It is an eight character field that has a value of "LVMREC01". The two IDs (*pv_id* and *vg_id*) are double words of type *lv_uniqueID* which is defined as

```

struct lv_uniqueID {
    ulong_t id1;           /* First part of ID. */
    ulong_t id2;           /* Second part of ID. */
};

```

They represent the CPU ID and the time when the *pvcreate* and *vgcreate* was performed respectively. i.e. *vg_id.id1* is the CPU ID of the CPU where the *vgcreate* was performed and *vg_id.id2* is the time when the *vgcreate* was done.

Within the LVM record are pointers to other LVM structures on the disk. The fields suffixed by *_psn* hold the starting sectors number for the VGRA, VGDA, VGSA, user data, Alternate Block Pool, BDRA, Boot Data Records, and PVol list. For structures which maintain primary and secondary copies, their starting sectors are indicated by the *_psn1* and *_psn2* fields. The length in sectors of each of these structures is indicated by the fields suffixed with *_len*.

The LVM record is analogous to the FS superblock.

At the end of the structure are fields for the MC/Serviceguard cluster lock and finally spare PVs (newly added feature at 10.30).

Slide: LVM Disk Structures: PVRA and BDRA

Boot Data Reserved Area (bootable disks only)

The BDRA contains a primary and secondary boot data record of type *lv_bootdata* that describes the root, swap, and dump volumes to be used during boot. Also in the BDRA is a primary and secondary PVol list. The table below shows the kernel definitions for each structure's starting location.

	Kernel #define	Sector #
Primary Boot Data Record	BDRA_BDR_SN1	128
Secondary Boot Data Record	BDRA_BDR_SN2	136

There is not a definition of the starting position for the PVol list but the kernel does define the overall size of the BDRA (*BDRA_SIZE*) to be 16 sectors and the individual lengths of the boot data record (*BDRA_BDR_LENGTH*) and the PVol list (*BDRA_PVL_LENGTH*) to be 2 and 6 respectively. So you can conclude that the PVol lists begin at sectors 130 and 138.

The BDRA and **LIF LABEL** file holds all the information required to boot. These data include:

root, swap, boot, dump — гонимые данные (группы устройств)

- Physical paths to all the disks in the VG - PV list area.
- VGID and VG number - boot data record.
- LV numbers for root, swap and dump - boot data record.
- Size and starting block for root, boot, swap and dump LVs - LABEL file.

The user data on a bootable disk starts at a fixed position at 2912k. This fact is used by both the maintenance mode boot and support tape/CD to bypass LVM.

While a maintenance mode boot can overcome a missing LABEL file it will use it if it exists. Therefore if it is corrupted, a maintenance mode boot may fail and it may be necessary to use the support tape/CD to remove it.

If root and boot are on separate LVs then the file rootconf in the boot LV is required on a maintenance mode boot. This file contains 3 words:

Word 0	Magic label 0xdeadbeef
Word 1	Start location of root LV

Slide: LVM Disk Structures: PVRA and BDRA

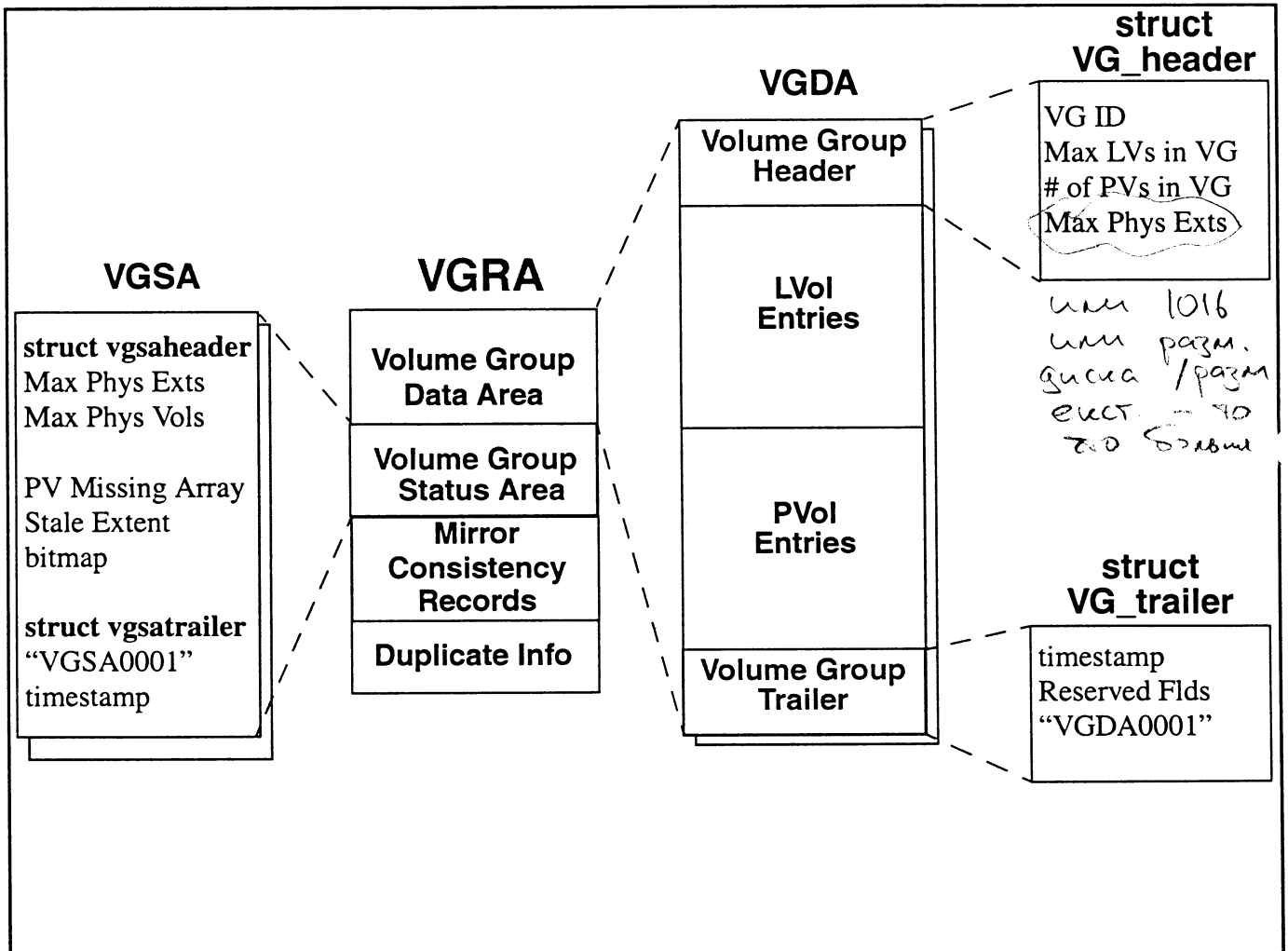
Word 2	Size of root LV
--------	-----------------

When a maintenance mode boot is performed `BD_PREVBOOTMAINT` is set in the boot data record in the `bd_flags` field. This flag is used to tell the kernel (on the next normal boot) to mark mirror copies of the root stale. This is required as a maintenance mode boot bypasses LVM and therefore changes made in the `-lm` boot will not have been propagated to the mirrors.

The magic value (*bd_magic*) for the boot record will always be "HPLVMBDR". The other fields in the BDRA describe the root volume group and the root, swap, and dump volumes within the group. There are several unused fields in this structure present for the future possibility of allowing separate swap and dump volume groups.

The *bd_rootVGID* is the Volume Group ID for the root volume group and has the same *lv_uniqueID* format discussed in the PVRA. If this value is zero it indicates a null BDRA and thus a volume that is not bootable.

Slide: LVM Disk Structures: VGRA (Header/Trailer)



Notes:

Slide: LVM Disk Structures: VGRA (Header/Trailer)

The Volume Group Reserved Area contains information about the volume group and is made up of the VGDA, the VGSA, and the Mirror Consistency Records. The starting location and size for each of these structures can be found in the LVM record in the PVRA.

At the start of the VGRA is the VGDA which has entries to describe each logical volume and each physical volume in the group. The VGDA has a header of type *VG_header* and a trailer of type *VG_trailer*

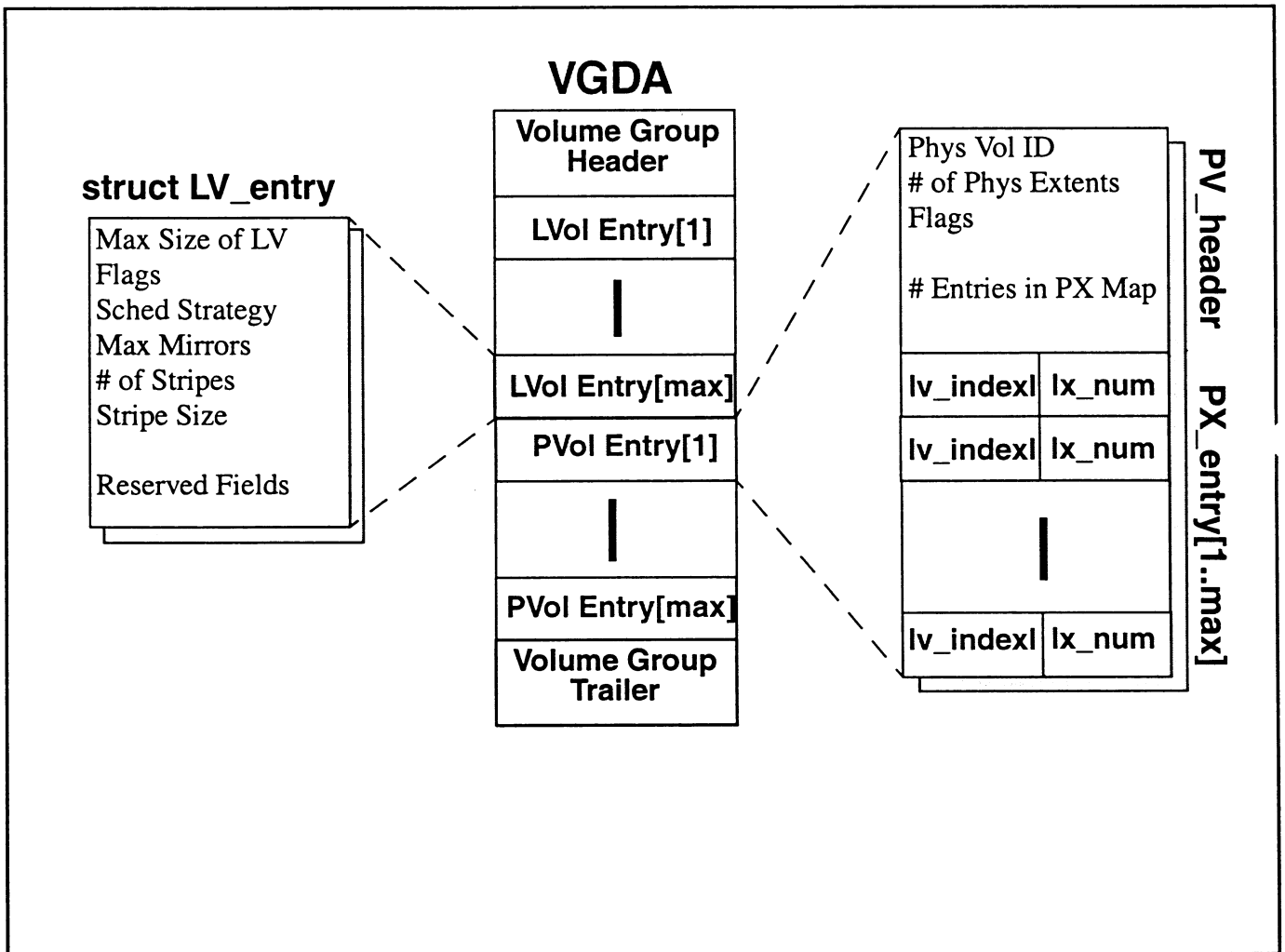
In between the header and trailer are LVol and PVol entries that will be discussed on the next slide. In the header there is a flags field which has one possible value. If the flags have `LVM_VGCFGRSTORD` (0x01) set, this is an indication that the physical volume has had its LVM configuration restored.

Each component of the VGDA (header, trailer, LVol, and PVol entries) starts on a sector boundary. The header can be located from the `vgda_psn1` value in the LVM record. Using this starting sector and the individual component lengths, the sector number for other parts can be found.

VGDA Header	<code>lv_lvmrec.vgda_psn</code>
VGDA LVol Entries	<code>lv_lvmrec.vgda_psn + 1</code>
VGDA PVol Entries	<code>(lv_lvmrec.vgda_psn + 1) + ROUNDUP(MAXLVS * sizeof(LV_entry))</code>
VGDA Trailer	<code>lv_lvmrec.vgda_psn + lv_lvmrec.vgda_len - 1</code>

Following the VGDA is the VGSA.

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)



Notes:

Module 9 — Logical Volume Manager

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)

The only remaining detail for the on disk LVM structures is the LVol and PVol entries in the VGDA. On the previous slide we already discuss the location of these two structures in the VGDA.

LVol Entries in VGDA (LV_entry) :

```
struct LV_entry {
    ushort_t    maxlxs;           /* Maximum size of the LV.          */
    ushort_t    lv_flags;        /* Logical volume flags.            */
    uchar_t     sched_strat;     /* The scheduling strategy.         */
    uchar_t     maxmirrors;     /* The maximum number of mirrors.  */
    ushort_t    stripes;        /* Number of stripes.              */
    ushort_t    stripe_size;    /* Size of each stripe in Kb.      */
    ushort_t    reserved2;     /* RESERVED */
    uint_t      lv_io_timeout;  /* # secs before declare LV I/O failed */
};
```

The area set aside for the LVol entries is large enough to hold MAXLVS entries. The maxlxs field indicates the maximum number of logical extents in the logical volume and sched_strat indicates the scheduling strategy for writes to this volume. The different strategies are discussed on page <>>

The lv_flags field contains logical volume specific flags. Possible values are

LVM_LVDEFINED	0x0001	Logical volume entry defined.
LVM_DISABLED	0x0002	Logical volume unavailable for use.
LVM_RDONLY	0x0004	<u>Read-only logical volume.</u>
LVM_NORELOC	0x0008	Bad blocks are not relocated.
LVM_VERIFY	0x0010	Verify all writes.
LVM_STRICT	0x0020	Allocate mirrors on distinct PV's
LVM_NOMWC	0x0040	No mirror consistency on this LV
LVM_PVG_STRICT	0x0080	Allocate mirrors on distinct PVG's
LVM_CONSISTENCY	0x0100	Mirror consistency recovery required
LVM_CLEAN	0x0200	LV has no mirror write in transition when
LVM_CONTIGUOUS	0x0400	Allocate contiguous PXs for this LV

Монитор
на агном
опер. гуна
Зеруага.

PVVol Entries in VGDA

The PVol entries in the VGDA are preceded by a PV header of type *PV_header*

```
struct PV_header {
    lv_uniqueID_t  pv_id;           /* The physical volume ID.          */
    ushort_t       px_count;        /* Number of physical extents.      */
    ushort_t       pv_flags;        /* The physical volume flags.       */
    ushort_t       pv_msize;        /* Size of PX entry map, in entries */
    ushort_t       pv_defectlim;    /* Max relocated defects allowed */
};
```

and is followed by the physical volume extent map. Possible values for the physical volume flags (pv_flags) are

LVM_PVDEFINED	0x0001	This entry is used
---------------	--------	--------------------

Module 9 — Logical Volume Manager

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)

LVM_PVNOALLOC	0x0002	No extent allocation is allowed.
LVM_NOVGDA	0x0004	Physical volume contains a VGDA.
LVM_PVRORELOC	0x0008	No new defects relocated
LVM_PVMISSING	0x0010	Physical volume is missing.
LVM_NOTATTACHED	0x0020	Physical volume is not attached.
LVM_PVPOWERFAIL	0x0040	Physical volume is power failing.
LVM_PVNEEDSYN	0x0080	Physical volume needs re-sync.
LVM_PVALTLINK	0x0100	Physical volume is not the primary link
LVM_PVINUSE	0x0200	Physical volume is being configured
LVM_PVCFGRSTOR	0x0400	Physical Volume had config restored on it
LVM_PVSWITCHLINK	0x0800	PV path requires switching
LVM_PVNOBACK	0x1000	Don't automatically switch links back
LVM_PVSPARE	0x2000	PV is a spare PV
LVM_PVDATA_SPARED	0x4000	PV has failed but data has been spared

На какому
ф.т. есть
инф. о брэх
прямик

Following the PV_header will be several one word physical volume extent entries of type *PX_entry*

```
} struct PX_entry {
    ushort_t    lv_index;    /* The logical volume index.    */
    ushort_t    lx_num;      /* The logical extent number.    */
};
```

Each entry correlates to one physical extent. If you are interested in the mapping for physical extent 24 on a particular volume, you look at *PX_entry[24]* to get the *lv_index* and *lx_num*.

All of the structures we have discussed so far reside on the disk. Most of the data that is used by the kernel is help in memory resident structures that combine many the on disk structures.

Structure timestamps

крит. в структуре не
нужно, а лучше сразу

These serve several purposes namely,

- As all PVs in a VG have the same data, and each PV has two copies, it is the timestamps that LVM uses to determine which structure is most recent and should therefore be used when the VG is activated.
- When a structure is updated (e.g. VGDA when an *lvcreate(1m)* is performed), in the case of a multi-PV VG, the structure with the oldest timestamp on each PV in the VG is overwritten. For single PV VG's both copies of the structures are updated.
- By comparing the header to the trailing timestamp LVM can ensure that the structure was successfully written to the disk.

Копиям VG — 50%, если обновить
то VG не уникализир., тогда
неправильно не будет записано

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)

Cluster lock

Если в кластере одна сеть, но работает другая служба. — те кто не работает уходят в перезагрузку.

ServiceGuard makes strenuous efforts to ensure it never gets into the situation where two or more nodes believe they have exclusive access to a VG. For instance, in the case of a two node cluster a tie-breaker is required to ensure that two clusters, containing one node each, do not result from network communication problems. The tie-break mechanism is performed by specifying a cluster lock PV.

The designated PV uses some additional fields in the lvmrecord to specify the cluster lock state and the location of the cluster lock. In actual fact the cluster lock “steals” the last few blocks from the alternate block pool.

An extract from the lvm program (which will be used in the lab exercises shows the LVM disc based fields it affects).

```
# ./lvm.11 -Pcld /dev/rdisk/c2t5d0
/* Last physical sector number. */ 2082635
....
/* Alternate block pool length. */ 832
/* Alternate block pool start. */ 2081792
/* max no. of defects expected */ 254
....
/* Cluster-lock flags. */ LVM_CLUSTER_LOCK_INITIALIZED
/* Cluster-lock area start. */ 2082624
/* Cluster that can use this vg */ 895683413
/* Configured activation Modes */ 0x100 LVM_VG_EXCL_ACT
.....
CLUSTER LOCK fields
lock_state LVM_DISK_CL_LOCK_FREE
cl_nodes[] 1 0 0 0 0 0 0 0
Mutex fields x_val=0x0 y_val=0x0
Node_ids[] 0 0 0 0 0 0 0 0
```

Отрабатывает только разряд, но и лам. не работает — поднимает борншмид

Here we can see that the alternate block pool extends from 1Kb blocks 2081792 to 2082623. The cluster lock from 2082624 to 2082634 (11 blocks).

11 блоков

The “Cluster-lock flags” field is set to LVM_CLUSTER_LOCK_INITIALIZED if the cluster lock is valid (i.e. initialized).

Cluster ID (895683413) is generated by cmapplyconf. It is the timestamp when cmapplyconf was run (echo 0d895683413=Y | adb will decode it). cmapplyconf generates a new Cluster ID whenever it is run and the /etc/cmcluster/cmclconfig is missing or corrupt. i.e. It does NOT create a new one every time it is run.

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)

Activation modes values and meanings are:

0x1000	LVM_VG_EXCL_ACT	Exclusive mode Activation
0x2000	LVM_VG_SHAR_ACT	Shared mode Activation.
0x4000	LVM_VG_SHARED_SERVER	This node is the Server, for this shared VG.

Then we come to the actual cluster lock itself. It is held in 11 blocks (1Kb normally, or 2Kb if this minimum I/O size for this disc). In actual fact almost all the relevant data is held in the first word of each block but 1Kb or 2Kb is the minimum request size that can be performed to discs and hence this size is used.

The cluster lock blocks contain the following information:

Block 0

A struct `lvm_disk_cl_lock` which comprises of members:

```
lock_state - LVM_DISK_CL_LOCK_FREE or LVM_DISK_CL_LOCK_OWNED.  
cl_nodes[LVM_MAX_NODES].
```

If `lock_state == LVM_DISK_CL_LOCK_OWNED` then one of `cl_nodes` should be non-zero to indicate the node which has it locked. The syslog file shows the mapping of node ids (which start from 1) to hostnames. For example:

```
cmclid[1305]: The new active cluster membership is: hppinf13(id=2)  
hppinf12(id=1)
```

Block 1

`x_value` mutex

Block 2

`y_value` mutex

Blocks 3-10

nodes 1 to 8 mutex

Blocks 1 through 10 are used to do the actual mutual exclusion locking. The locking details are described in the comments for the kernel function which performs the locking (`lv_obtain_cluster_lock`).

Finally here is a cluster lock which is locked.

Module 9 — Logical Volume Manager

Slide: LVM Disk Structures: VGRA (PVOL/LVOL Structures)

CLUSTER LOCK fields

lock_state	LVM_DISK_CL_LOCK_OWNED							
cl_nodes[]	0	2	0	0	0	0	0	0
Mutex fields	x_val=0x2 y_val=0x0							
Node_ids[]	0	0	0	0	0	0	0	0

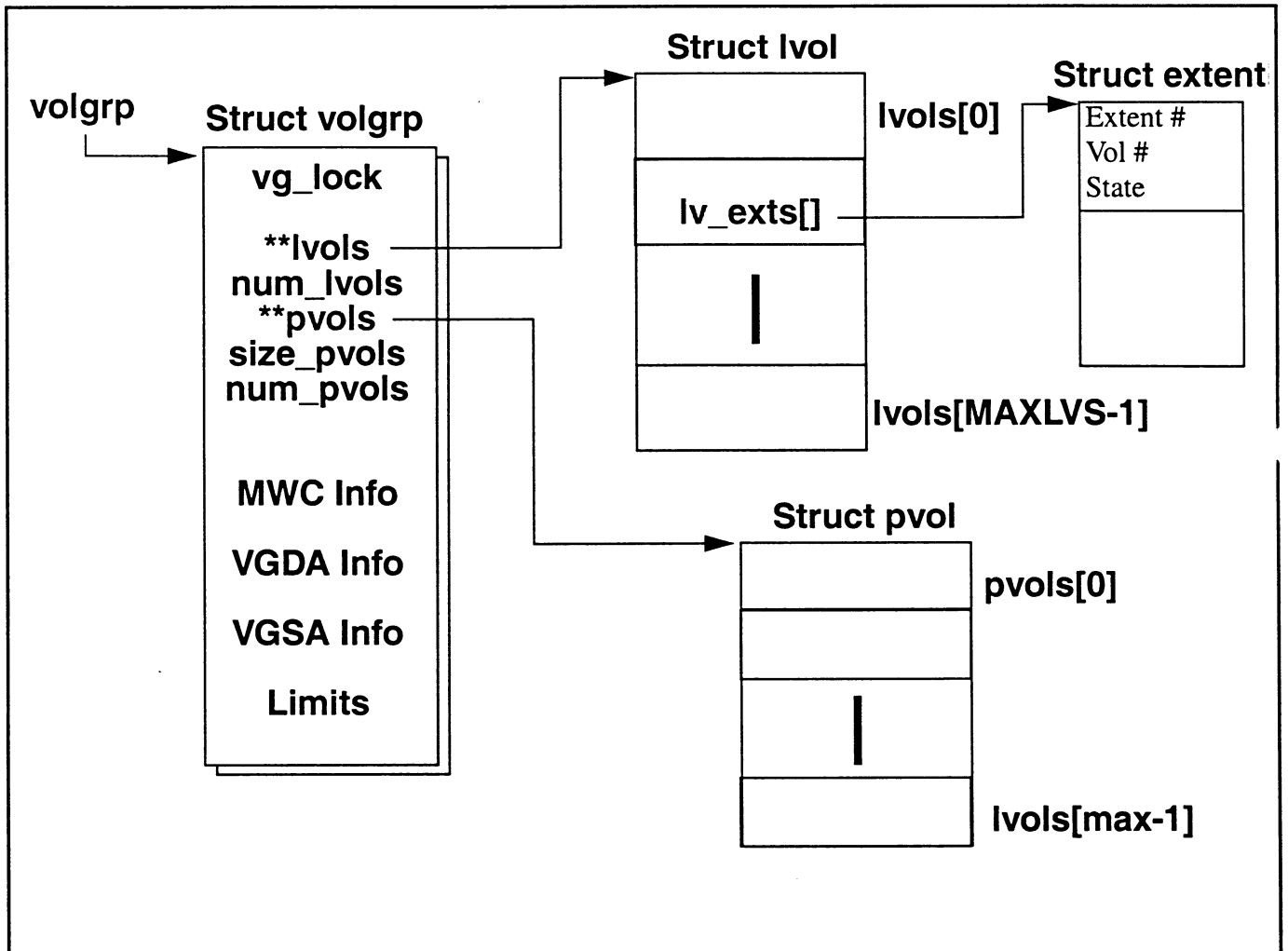
LAB: LVM Disk Structures

Setup : This lab requires the program lvm.11. The latest copy is held at URL <ftp://hppine34.uk.sr.hp.com/pub/lvm.11.Z>

1. Create or use an existing VG containing a bootable PV (mkboot to install LIF files). Then answer the following questions using the lvm.11 and lifls programs.
 - a. Where does the LIF data start?
 - b. How many 1Kb blocks does the LIF data occupy?
 - c. What is the PVID?
 - d. What is the VGID?
 - e. What is the PV number?
 - f. At what block does PE zero start
 - g. Answer 1c to 1f using standard commands rather than the lvm program.

Left blank intentionally

Slide: Memory Resident Structures



Notes:

Преобразование структуры
 массива - структура.

Module 9 — Logical Volume Manager

Slide: Memory Resident Structures

Memory Resident Structures

The three main LVM structures are volgrp, pvols and lvols. These respectively hold the Volume Group, Physical Volume and Logical Volume details in memory.

The volgrp structure contains the pointers to the other two structures so it is the obvious starting point.

Memory Resident Structures - volgrp

The number of volgrp structures is determined by the maxvgs kernel parameter, the default is 10. The entry point volgrp points to the array of pointers to the volgrp structures. The VG number in the group file minor number corresponds to the volgrp array index. e.g. minor number 0x020000 is volgrp[2]. The macro DEV2VG does this mapping.

The volgrp structure is one of the largest in the kernel. Some of the most interesting and useful fields are:

lvols and num_lvols	Pointers and size of lvols array (num_lvols is fixed at 256 when initialized).
pvols and num_pvols	Pointers and size of pvols array.
major_num	Should be 0x40 (good/fast sanity check)
vg_id	Two word Volume Group IDentification
vg_extshift	log base 2 of extent size in 1Kb blocks Default extent size of 4Mb would appear as 12. ($2^{12} == 4096 == 4Mb$)
vg_opencount	number of LVs open
vg_flags	VG_LOST_QUORUM 0x01 running quorum is lost VG_ACTIVATED 0x04 VG is activated VG_NOLVOPENS 0x08 Don't allow logical volume opens VG_READONLY 0x10 VG was activated as read-only
vg_totalcount	total count of requests. Incremented in lv_strategy()
vg_requestcount	outstanding requests. Incremented in lv_strategy() decremented on completion.
vg_num	VG number. (sanity check)
vg_cluster_id	Cluster ID which can use this VG.

Можно посмотреть на xogy, vg едем гук
wogu m02en на xogy, vgchange -ay

Module 9 — Logical Volume Manager

Slide: Memory Resident Structures

vg_config_mode	CLV_VG_CONF_STD 0x000 Non-special CLV_VG_CONF_EXCL 0x100 Exclusive mode activation. CLV_VG_CONF_SHAR 0x200 Shared mode activation
----------------	---

Refer to “lvm/lvmd.h” for more details.

Memory Resident Structures - lvol

The lvol structure holds the LV information for the kernel. As we have seen the lvol structures can be located from the lvols field in the volgrp. The macro VG_DEV2LV is used to select the appropriate LV from the device minor number. Some useful fields:

lv_ext	logical extent array
lv_exts	physical extent arrays
lv_stripes	number of stripes
lv_stripesize	size of each stripe in Kb
lv_totalcount	total count of requests. Incremented in lv_strategy()
lv_requestcount	outstanding requests. Incremented in lv_strategy() decremented in lv_complete()
lv_status	Twelve different flags (e.g. LV_OPEN, LV_BLOCKOPEN)
lv_curpxs	number of physical extents
lv_maxlxs	maximum number of logical extents
lv_curlxs	in use number of logical extents (rarely will lv_maxlxs != lv_curlxs and when so is indicative of a problem).
lv_sched_strat	LVM_RESERVED 0x0000 group file lvol0 strategy LVM_SEQUENTIAL 0x0001 sequential LVM_PARALLEL 0x0002 parallel LVM_STRIPE 0x0003 stripe LVM_DYNAMIC 0x0004 dynamic (not used) LVM_STRIPE_NEW 0x0008 stripe (not used)
lv_maxmirrors	Number of mirrors
lv_vg	address of volgrp (sanity check)

Module 9 — Logical Volume Manager

Slide: Memory Resident Structures

lv_io_timeout	max. number of seconds before timing out requests
---------------	---

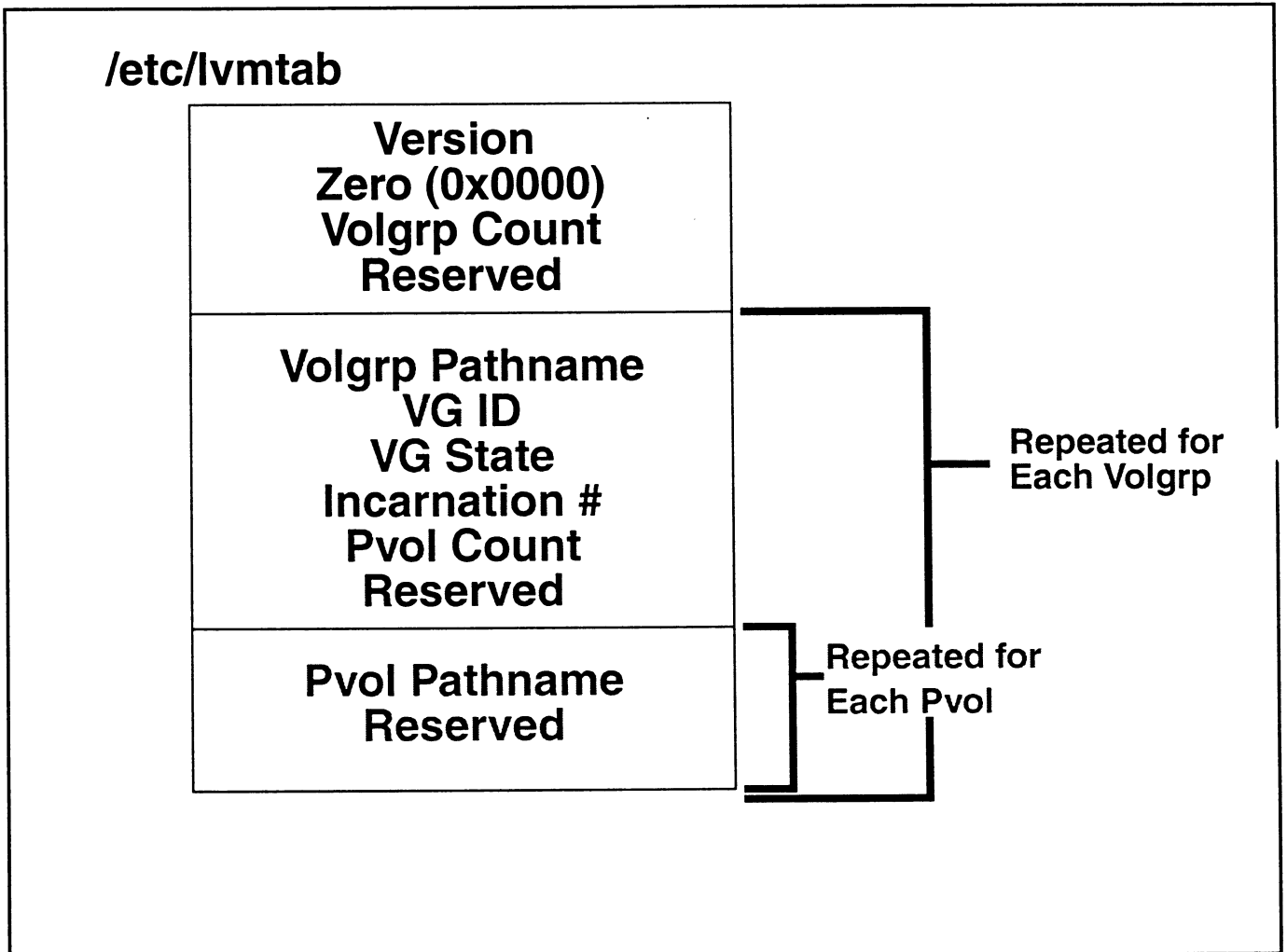
Refer to “lvm/lvmd.h” for more details.

Memory Resident Structures - pvol

As for the lvol structure the pointers to these structures are held in the volgrp. Some of the useful fields are:

pv_vg	Pointer to volgrp structure
pv_lvmrec	Pointer to PVs lvmrec
pv_bbdir	Pointer to PVs bad block directory (BBDIR)
pv_curdefects	Current number of entries in BBDIR
pv_freepxs	Free PEs
pv_ready_Q	Requests to pass to physical driver
pv_totxf	Total number of transfers to this PV
pv_curxfs	Number of outstanding requests
pv_flags	See page 10-27 for possible settings.
pv_num	PV index number
currentPhysicalLink	Pointer to vnode and PV link information
pv_blkfactor	Minimum number of 1Kb blocks in an I/O request
pv_spare_info	Spare PV structure

Slide: LVMTAB File



Notes:

Используется всегда, при любом
манипулировании томами

VGSCAN

Slide: LVMTAB File

LVM maintains a file that records information about the volume groups defined on the system. This file is called `/etc/lvmtab`. The data from this file is loaded into memory and written back out when changes are made to the LVM configuration.

Note that the kernel itself knows nothing of the `lvmtab` file or the in-core copy of its data. It is the `lvm` commands that access this data. Each command reads the `lvmtab` into memory to carry out the desired task and then writes back any changes.

The slide shows the layout of the `lvmtab` file on disk. The file begins with some header information

<code>0x00</code>	<code>short</code>	<code>vers_no</code>
<code>0x02</code>	<code>short</code>	<code>zero</code>
<code>0x04</code>	<code>int</code>	<code>vg_cnt</code>
<code>0x08</code>	<code>int [3]</code>	<code>reserved</code>

The `vers_no` for `lvmtab` will always be `LVMTAB_VERSNO` (#1000). Prior to 10.0, this field was an integer which caused the version number to be stored in the second half of the word. To distinguish a 10.0 `lvmtab` from pre-10.0, the `vers_no` is now a short and the second half of the word will always be zero. If we try to read a pre-10.0 version of `lvmtab` it will fail because the `vers_no` will come back as zero.

The `vg_cnt` field is the count of the number of volume groups defined in `/etc/lvmtab`.

Volume Group Information

The header is followed by a volume group entry for each volume group defined.

<code>0x00</code>	<code>char[1024]</code>	<code>vg_path</code>
<code>0x400</code>	<code>lv_uniqueID_t</code>	<code>vg_id</code>
<code>0x408</code>	<code>short</code>	<code>vg_state</code>
<code>0x40c</code>	<code>int</code>	<code>vg_incno</code>
<code>0x410</code>	<code>int</code>	<code>pv_cnt</code>
<code>0x414</code>	<code>int [3]</code>	<code>reserved</code>

The volume group records begin with the pathname for the volume group in `vg_path`. This is the device file name for the volume group, such as `"/dev/vg00"`. The state of the volume group is recorded in `vg_state` with possible values of

<code>0x0</code>	<code>STANDARD</code>
<code>0x1</code>	<code>SHARED</code>
<code>0x2</code>	<code>EXCLUSIVE</code>

The count for the number of physical volumes in the volume group is recorded in `pv_cnt`.

Slide: LVMTAB File

Physical Volume Information

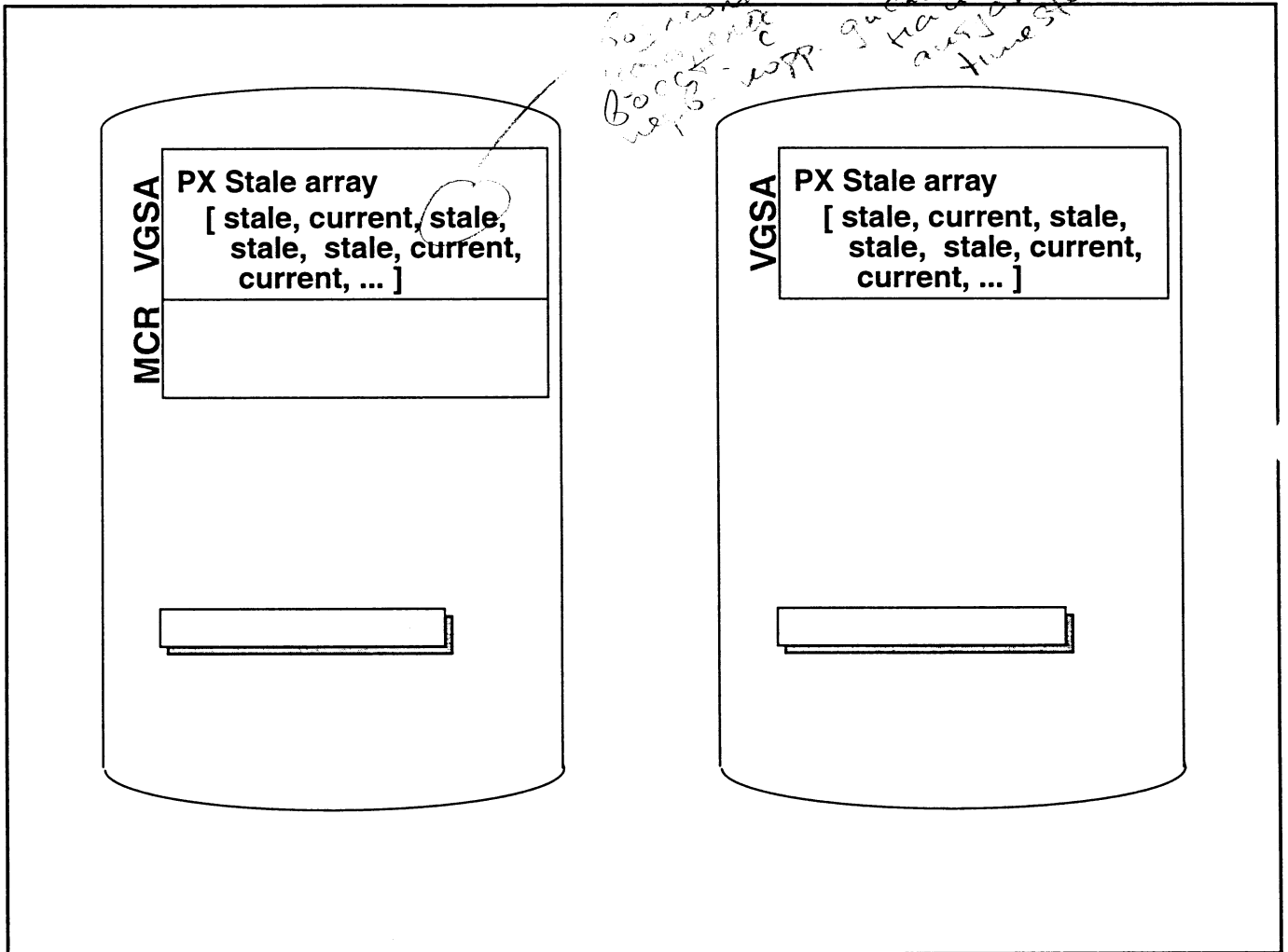
Each volume group entry will also have a repeated physical volume entry for each pvol belonging to the volume group

0x00	char[1024]	pv_path
0x400	int	reserved

The path name for the physical device file is recorded here in pv_path.

Left blank intentionally

Slide: Mirror Write Cache



Notes:

Slide: Mirror Write Cache

LVM maintains mirror consistency for LVs which have not been shutdown cleanly by one of 3 methods. A logical volume may be setup for MWC, NOMWC, or NONE. The method of consistency deals more with how mirrored volumes are resynchronized than how a write is handled.

NOMWC

Resynchronization of mirrored volumes occurs at activation time. When NOMWC is specified all but one physical extent for each logical extent is marked stale. Then, the entire logical volume is resynchronized from the non-stale copies.

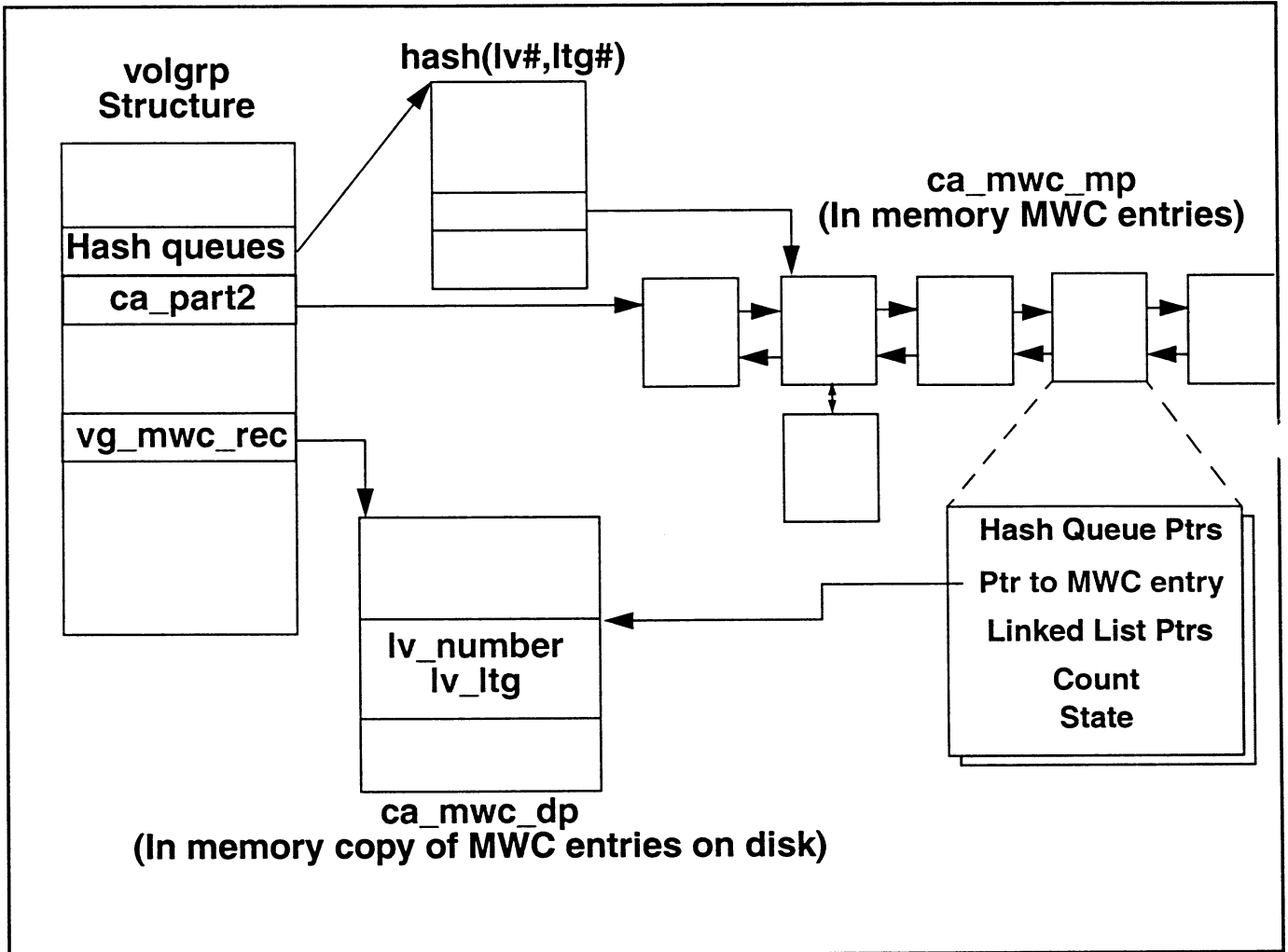
NONE

When the method of NONE is specified there is no internal consistency protection. No extents are marked stale at activation time. In this case the only way a physical extent will be marked stale is as the result of a disk write failure. Useful for swap devices.

MWC

What we are most concerned with is the structures and method of handling resynchronization when a volume is setup for MWC. In a MWC setup, for any request to proceed it must have an entry in the cache on one of the disk. At activation time, all logical tracks represented in the cache must be resynchronized. This is done by marking all but one of the physical tracks stale and resynchronizing from the one that is left. There is no way for LVM to know which extent contains the latest data so MWC simply ensures that all physical extents contain the same data, not necessarily the latest data.

Slide: Mirror Write Cache Data Structures



Notes:

Slide: Mirror Write Cache Data Structures

Mirror Write Cache Disk Structure

The MWC contains 32 consecutive entries on disk in the following form:

lv_number	ltgshift
lv_ltg	

...

The lv_number represents an index into the array of logical volumes for the volume group being accessed. The MWC maintains information in units of one logical track group. The lv_ltg value refers to the logical track group containing elements of a possible outstanding write. If the request spans several logical track groups, multiple MWC entries must be obtained -- one for each ltg. The ltgshift value is used to convert the lv_ltg to a disk block number.

These records exist on disk but there is also a memory copy of this disk part of cache. The memory copy is found through the volume group structure using volgrp->vg_mwc_rec and is of type mwc_rec.

Mirror Write Cache Memory Structure.

The memory part of the cache is also linked into the volume group structure, this one through volgrp->ca_part2.

The ca_pvol value is the physical volume that this MWC entry is first written to. The ca_mwc_ent points to the memory copy of the MWC disk structure for this entry. The ca_next and ca_prev pointers link all MWC entries onto a circular list.

The ca_iocnt value represents the number of I/O requests that currently rely on this cache entry. A value other than zero indicates that a second request for the same ltg has come through and is waiting on a request ahead of it to complete.

There are two possible states that can be reflected in ca_state.

CACHE_ENTRY_CHANGING

CACHE_ENTRY_FROZEN

Slide: Mirror Write Cache Data Structures

MWC fields in the Volume Group Structure

We have already looked at the memory resident volume group structure on page 10-27. The table below list a description of all the MWC fields in this structure

field	meaning
vg_ca_intlock	The interrupt level spinlock used to synchronize access to these fields between process context and interrupt context.
vg_mwc_lbuf	The logical buffer header used when writing the MWC disk part to one of the physical volumes.
vg_cache_wait	A queue of all logical requests awaiting a free MWC cache entry.
vg_cache_write	A queue of physical volumes that are ready to have the MWC written to them.
vg_mwc_rec	A pointer to the memory copy of the disk part of the cache.
ca_part2	A pointer to the beginning of the memory space allocated for the memory part of the cache.
ca_lst	A pointer to the least recently used cache entry.
ca_hash[MWCHSIZE]	The anchors for the hash lists of entries used to speed up searching for cache entries.
ca_free	The number of currently unused entries.
ca_size	The total number of cache entries.
ca_chgcount	The number of changed entries.
ca_flags	Flags used to describe the state of the cache as a whole. Possible values are: CACHE_ACTIVATED meaning the cache has been set up, CACHE_INFLIGHT meaning the cache is in the process of being written to a disk, CACHE_CHANGED meaning the memory copy of the disk part has been changed to differ from the actual disk image, and CACHE_CLEAN meaning that something is sleeping for cache entries to be cleaned out and is waiting for the cache write to have completed to the disk (in other words, the cache was INFLIGHT when the process attempted to clean it).
ca_clean_lvnum	The logical volume being cleaned out of the cache. This is used whenever extents are being removed from a logical volume to make sure they are removed from the cache.

When a volume group is first created, or when it is activated with a clean cache, LVM sets up the cache in the following manner:

Slide: Mirror Write Cache Data Structures

1. Allocate memory for the disk part and the memory part and point to them via `vg_mwc_rec` and `ca_part2`.
2. Connect the `ca_mwc_ent` pointers to point to the corresponding areas in the memory copy of the disk part.
3. Build the circular list of entries in the memory part using the `ca_next` and `ca_prev` pointers.
4. Hang this circular list off the `ca_lst` field in the struct `volgrp`.
5. Initialize the hash pointers in the struct `volgrp` (`ca_hash[]`) to be NULL.
6. Set `ca_free` equal to `ca_size`.
7. Flag the cache as `CACHE_ACTIVATED`.

Writes Through MWC

When using the MWC consistency mode, LVM maintains a “cache” per volume group of possible outstanding writes. Before any write request can proceed, it must have an entry in this cache on one of the disks in the volume group. When the volume group is subsequently activated, all areas that are referred to in the cache are resynchronized. LVM does not know which physical extent contains the latest data, it merely ensures that all physical extents for a logical extent contain the same data.

The first step is to check whether the MWC is enabled for the logical volume. This is done by checking the `lv->lv_flags` for a value of `LVM_NOMWC`. If this is set, then MWC is disabled for the volume and the request is passed straight to the scheduling layer.

If the MWC is not disabled, the request goes into the Mirror Consistency Layer. When a write request reaches the MWC layer of the LVM driver, the driver uses the following steps to ensure that a cache entry exists for it on disk. The steps we follows are

1. Search through the appropriate hash list to see if the area of this request already has a cache entry. If so, increment the `ca_iocnt` in the memory structure and put it at the front of the MRU chain. The request can now proceed to disk.
2. If the request isn't already covered by the cache, it needs to grab a cache entry for itself. It searches the LRU chain looking for entries with a `ca_iocnt` of zero and a clear `ca_state`. If it doesn't find one, it puts itself on the `vg_cache_wait` queue.
3. Once it has grabbed an unused cache entry for itself, it marks the MWC as `CACHE_CHANGED` and looks for a `pvol` to send the MWC request to. It then puts itself on the `pv_cache_wait` queue and starts the MWC I/O.
4. When this I/O completes, all requests waiting in the `pv_cache_wait` queue for the physical volume being used are allowed to proceed.

Slide: Mirror Write Cache Data Structures

When the write request completes, the LVM MWC layer looks at the `vg_cache_wait` queue and uses the above algorithm to get these requests going. The order of this queue is maintained ensuring that requests are never starved.

Module 10

The I/O Subsystem

“Need input. Must have input!”

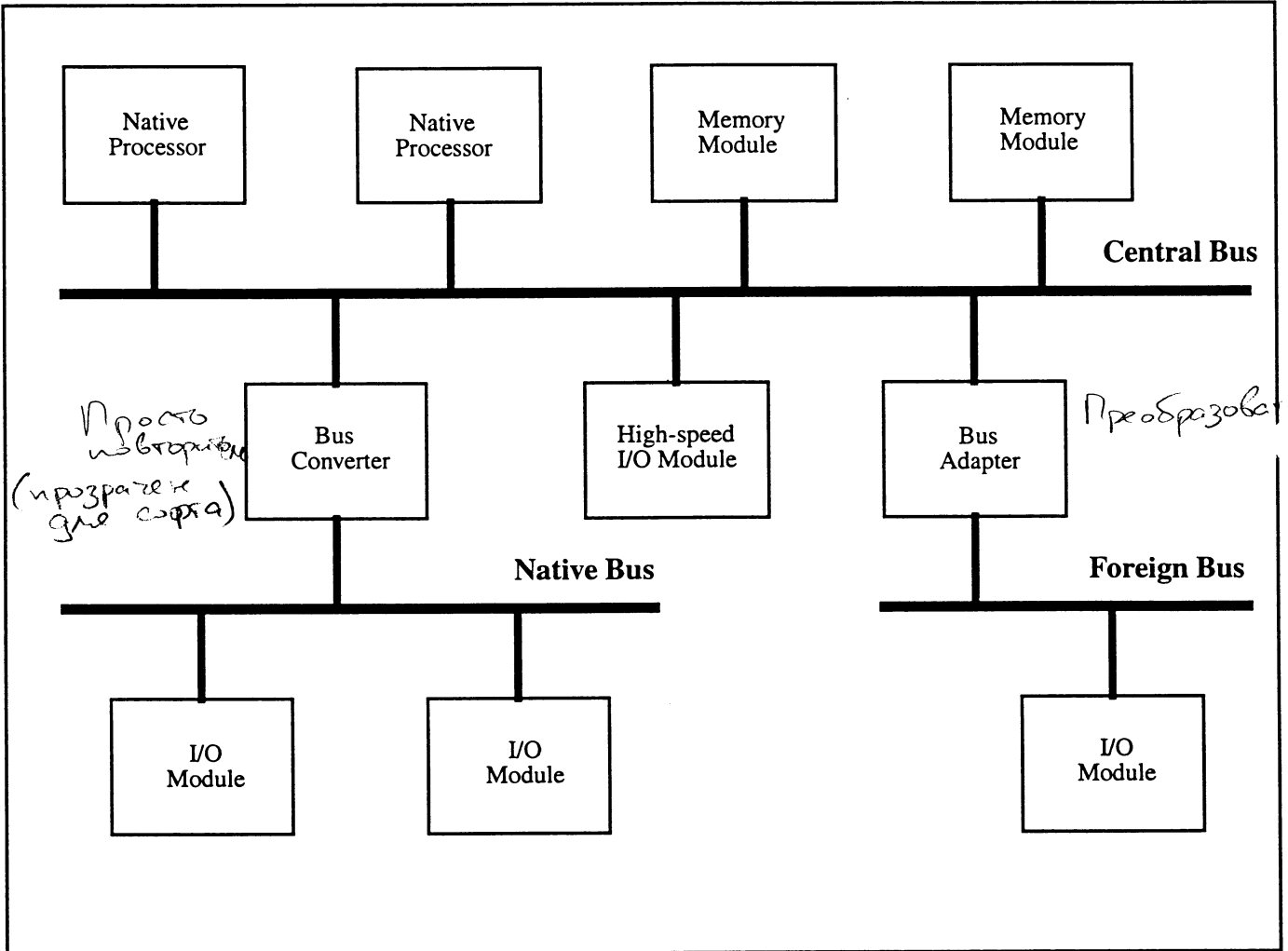
-- Johnny Five, “Short Circuit”

Objectives :

- Review the PA-RISC I/O Architecture.
- Review the relationship between Device Files and Switch Tables.
- Discuss the Converged I/O system model.
- Define the General I/O (GIO) Interface and associated objects, including the I/O Tree.
- Define the Context Dependent I/O (CDIO) Interface.
- Define WCIO and SIO CDIO Interfaces.
- Discuss the operation of driver environment and bus nexus CDIOs.

Module 10 — The I/O Subsystem

Slide: Example Topology of a PA-RISC System



Notes:

Slide: Example Topology of a PA-RISC System

PA-RISC I/O Architecture

The PA-RISC I/O Architecture is defined in the **PA-RISC 1.1 I/O Architecture Reference Specification**¹ (also referred to as the **I/O Architecture Control Document** or **I/O ACD**). Changes for PA-RISC 2.0 are identified in the **PA-RISC 2.0 I/O Architecture Reference Specification**²: these are primarily related to the use of 64-bit registers and the 64-bit address space layout.

These documents outline the objectives of the architecture and the constraints which it imposes. These include:

- all processor and memory modules must reside on the central bus
- each PA-RISC system must have at least one processor module, memory module and I/O module
- the maximum number of modules on a native bus is 64
- there may be up to three levels of Bus Converters

System Components

- native bus -** one which meets the requirements of the connect protocol defined by the architecture (e.g. can interrupt the processor). The **Central Bus**, also referred to as the **Processor Memory Bus (PMB)**, must be a native bus.
- Examples:
- **Runway** (central bus for J, K-class)
 - **Summit** (central bus for T-class)
 - **HP-PB (HP Precision Bus)**, also **Native I/O (NIO)**
 - **GSC (Gecko System Connect)**
 - **HP-HSC (High Speed Connect) or GSC+**
- Details on the implementation of each bus are given in the External Reference Specification (ERS) for that bus.
- foreign bus -** one which does not meet the requirements of the connect protocol.
- Examples:
- **EISA (Extended Industry Standard Architecture)**
 - **VME (Versa Module Eurocard)**
 - **PCI (Peripheral Component Interconnect)**
 - **CIO (Channel Input/Output)** - from the HP9000 500-series (*not* a native bus because the modules do not understand the memory addressing model used by PA-RISC).
 - **core** (midplane bus used for workstation core I/O)
 - **Quix** (midplane bus used for Nova MFIO cards)
- module -** an entity which is configured into the system address space and which adheres to the PA-RISC I/O Architecture.

1. PA-RISC 1.1 I/O ACD, v0.96, SADL - http://arch.cup.hp.com/programs/doc/io_acd/v0.96/

2. PA-RISC 2.0 I/O ACD, v0.36, SADL - http://arch.cup.hp.com/programs/doc/io_acd/2.0v0.36/

Slide: Example Topology of a PA-RISC System

- module set -** a group of two or more modules completely contained on a single card. A card containing a module set is known as a **multi-module card**.
Example: 28655A HP-PB SCSI / parallel Centronics “Skunk” interface is two modules on one card: SCSI-2 Single-ended and Centronics.
- card -** a physical entity containing the components which are necessary for module operation. A card has one and only one slot address. A module is always implemented on a single card, but a card may consist of more than one printed circuit boards (PCBs).
Cards may be divided into two functional areas:
- a backplane interface to handle the I/O bus to which it is connected (e.g. bus protocol, DMA requirements)
- a frontplane interface which understands the device(s) which it drives (e.g. the SCSI protocol)
- native processor -** a module which implements the Precision instruction set and can execute the standard operating system software (e.g. HP-UX, MPE/iX). For a complete description of the internal processor architecture and instruction set, refer to the **Precision Architecture and Instruction Set Reference Manual (PA-RISM)** for PA-RISC 1.1¹ or PA-RISC 2.0².
- bus converter -** a pair of **bus converter port** modules, linked together by a **bus converter link**, which connect two native busses. A bus converter (bc) can be used to:
- increase the total number of modules in a system
- connect a high-speed PMB to a lower speed I/O bus
- allow two busses to be physically separated
During normal system operation, bus converters are completely transparent to software: i.e. software running on any processor can control the modules on any bus.
Examples:
- **GeckoBOA** - a bus converter between *HP-HSC* and *HP-PB* used in K-class systems.
- **U2** - actually a pair of bus converters known as **IOAs**, which connect the *Runway* PMB to *HP-HSC* in J-class, K-class and high-end D-class systems.

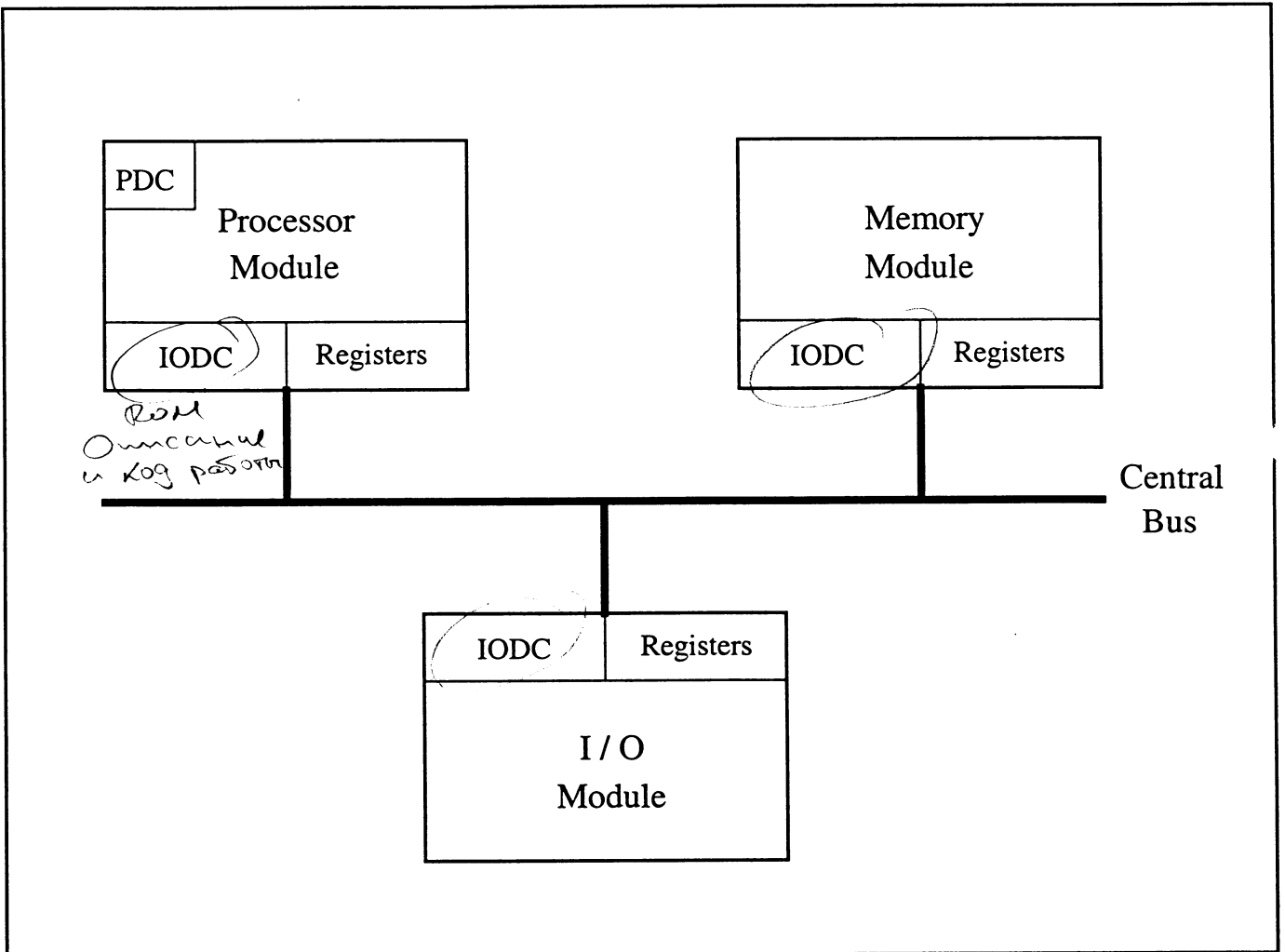
1. PA-RISC 1.1 Architecture and Instruction Set Reference Manual 3rd Ed, February 1994 09740-90039
2. PA-RISC 2.0 Architecture, Gerry Kane, Prentice Hall 1996

Slide: Example Topology of a PA-RISC System

- bus adapter -** a module which connects a **native bus** to a **foreign bus** and handles the conversion of the native *I/O* bus protocol to and from that of the foreign bus. Software (a bus adapter manager) is required.
Examples: - **LASI** - bus adapter between *HP-HSC* and the foreign *core* bus used on the *MFIOC* chip in J-class, K-class, D-class, B-class and C-class systems.
 - **WAX** - bus adapter between *HP-HSC* and *EISA*, *RS-232* for D-class and J-class and other systems.
- bus bridge -** a special case of a bus adapter, which provides a more complete implementation of the modes and protocols used on the native bus.
Example: - **DINO PCI** bridge from *HP-HSC* used in B-class and C-class.
- device adapter -** an entity which connects to an *I/O* bus and provides connection to a device.
- device -** the object to which input and/or output operations are done. Devices are connected to but are not part of an *I/O* module. They are accessed directly or indirectly through the address space of that module and they may optionally be independently powered. Each *I/O* module may have any number of devices connected to it. For the purposes of architectural discussion, the device includes all entities between the module and the device (e.g. cables, controllers, link adapters).
Examples: terminals, disks, tape drives, network connections.

Module 10 — The I/O Subsystem

Slide: I/O and Processor Dependent Code



Notes:

Slide: I/O and Processor Dependent Code

I/O Dependent Code (IODC)

IODC, typically contained in a ROM, is used to provide a uniform, architected mechanism to obtain module-type dependent information from a module. It is composed of two parts:

- a set of up to sixteen bytes which identify and characterize the module.
These contain sufficient module-type dependent information to allow a configuration to be determined automatically during system initialization. This enables the installation of new modules and I/O devices without modification of the processor configuration or boot ROM.
- a set of entry points that provide a standard procedural interface for performing module-type dependent operations.
These entry points provide a consistent interface for operations such as module initialization and testing. Special IODC entry points are defined to support boot.

Processor Dependent Code (PDC)

Processor Dependent Code (PDC) is used to provide a uniform, architected context in which to perform processor-dependent operations. Two PDC mechanisms are used:

- a set of entry points that are triggered when special events are recognized by the processor hardware. The entry points relate to initialization and error recovery and are defined as:
 - PDCE_TOC** Transfer of Control
 - PDCE_CHECK** Machine Check Preparation (HPMC or LPMC)
 - PDCE_RESET** Processor reset
- a software entry point which provides a variety of options to execute specific procedures. These procedures access processor-dependent hardware and return parameters that characterize or identify the processor. PDC procedures are provided as options to the entry point **PDC_PROC**. The architected procedures include:
 - PDC_CHASSIS** update chassis display
 - PDC_MODEL** return processor model information
 - PDC_IODC** access a module's IODC
 - PDC_TOD** access time-of-day clock
 - PDC_BLOCK_TLB** manage a block Translation Lookaside Buffer (TLB)
 - PDC_TLB** manage hardware TLB miss handling

The most natural implementation for PDC is via Precision code stored in a processor ROM. This does not, however, preclude special hardware support for PDC operations. Any of the PDC procedures may be performed by any combination of code and hardware. In particular, a support processor may be valuable in performing some PDC procedures.

A single copy of PDC may be shared between separate processors, provided that some mechanism is employed within PDC so that each processor appears to have its own copy.

Slide: I/O and Processor Dependent Code

Module Registers

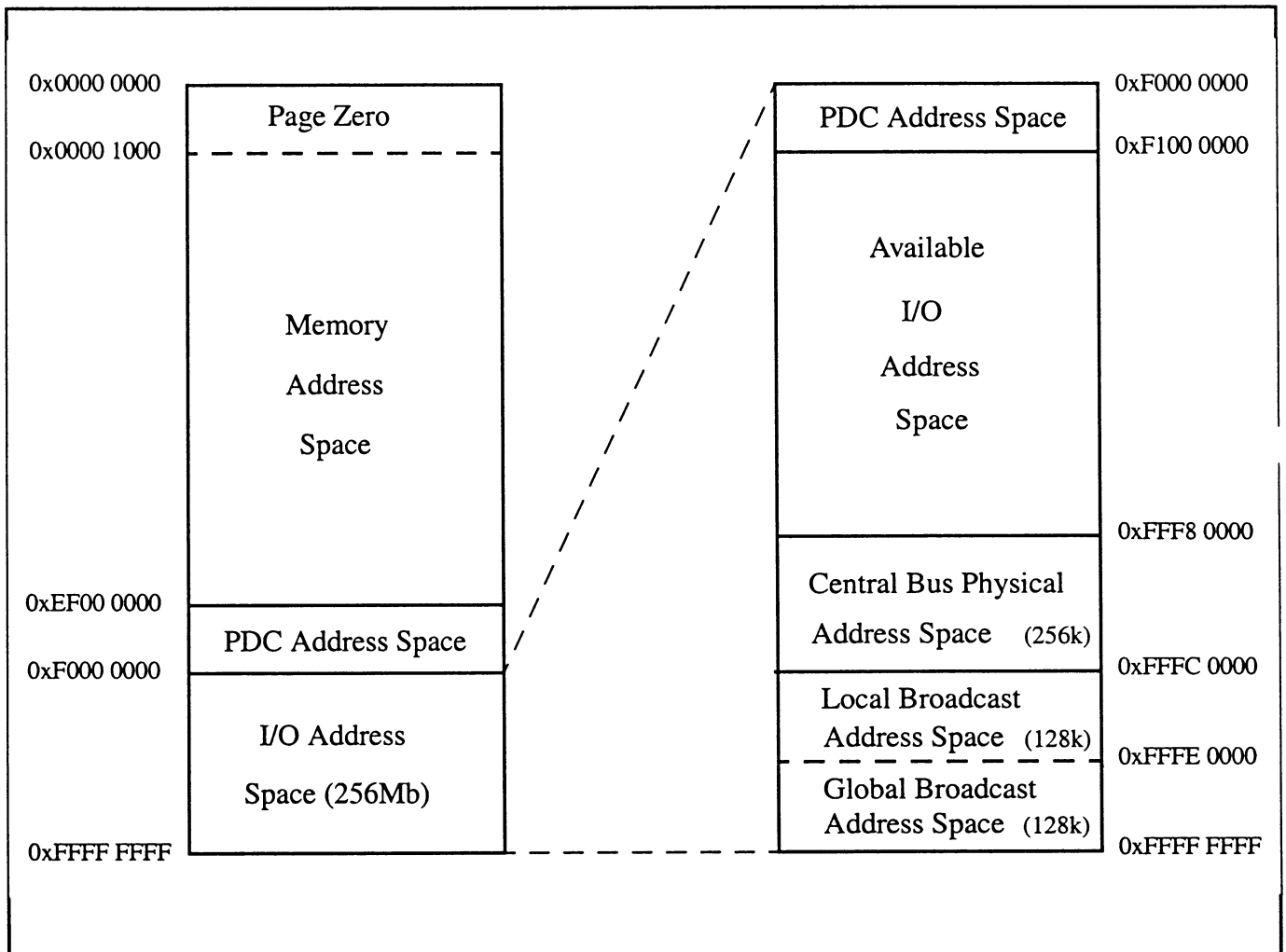
Each module connected to a native bus contains a set of registers which are directly addressable by other modules connected to a native bus on that system. The registers are accessed using the same LOAD and STORE instructions which are used for memory. They allow the processor to configure and control the modules. Processor modules also have such registers: in this case, they are used to allow other modules to interrupt the processor, as well as enabling support for multi-processor configurations.

Module 10 — The I/O Subsystem

Left blank intentionally

Module 10 — The I/O Subsystem

Slide: System Address Space: 32-bit



Notes:

Slide: System Address Space: 32-bit

The address space of a PA-RISC 1.x system, or PA-RISC 2.0 system running in narrow mode, uses 32-bit physical addresses. The system address space is logically subdivided into pages, each of which is 4Kbytes in size and aligned on a 4Kbyte boundary.¹ The 32-bit address is referred to as a **real** or **absolute** address, since it represents data which can be addressed directly by the processor via the address lines on the central bus. It can be viewed as a 20-bit physical page number, with a 12-bit offset into the page. The use of 32-bit addresses and byte addressing results in a total addressable space of 4 gigabytes.

The PA-RISC I/O architecture is **memory-mapped** - i.e. complete control of all modules is performed using memory read and write commands. Processors control I/O modules by executing **LOAD** and **STORE** instructions to **I/O registers**, which function like storage locations from the perspective of the software, using either real or virtual addresses. There are no specific I/O instructions in the instruction set. The addresses for modules are assigned during system initialization.

If the target address for an operation is a memory location, then cache will be updated.
If it represents an I/O or memory controller location, cache will *not* be updated: all pages in the I/O address space are uncacheable.

Page Zero (the first 4kbytes of memory - addresses 0x0 through 0x7FF) is restricted for use by the **boot** and **Initial Program Load (IPL)** software and is used for communication between the IPL and the user kernel being booted².

Addresses in the **Memory Address Space** from 0x0 to 0xEEFF FFFF are available only for assignment to memory modules.

The area from 0xEF00 0000 to 0xEFFF FFFF is reserved for **PDC**. Although PDC does not have to reside here, these addresses cannot be used for any other purpose.

The 256Kbyte space from 0xFFFC 0000 through 0xFFFF FFFF is called the **Broadcast Physical Address Space**: addresses in this range are known as **Broadcast Physical Addresses (BPAs)**. The space is split into two portions:

- **Local Broadcast Address Space** (0xFFFC 0000 - 0xFFFD FFFF)
Transactions sent to this area will be broadcast to all modules on a bus.
All modules on the same bus as the requestor will respond to a request sent to these addresses.
- **Global Broadcast Address Space** (0xFFFE 0000 - 0xFFFF FFFF)
Transactions sent to this area will be broadcast to all the modules on every native bus on the system.
All native modules in the system will respond to a request sent to these addresses.

In a single-bus system, global and local have the same effect.

1. PA-RISC 1.1 I/O ACD v0.96, section 1.3 "System Address Space"

2. PA-RISC 1.1 I/O ACD v0.96, Appendix C "Memory Data Formats"

Slide: System Address Space: 32-bit

Three functions require the use of the broadcast address space:

1. Broadcast interrupts can be sent to all processors (on the local bus or in the entire system).
2. A Global Broadcast Reset command can be used to reboot the system by resetting all the processors in the system.
3. A local broadcast can be used to assign a portion of the system address space to each of the modules on the local bus.

Each native bus in the system is allocated a bus address space of 256Kbytes. This represents the modules on that bus and is subdivided into units of 4kbytes for each of the 64 potential modules which may be present. This 4Kbyte **Hard Physical Address (HPA)** space represents the registers on the module. The module's position (offset) within the address space for the bus is determined by its location on the bus (slot).

The **Central Bus Physical Address Space**, also referred to as the **Fixed Physical Address Space**, is reserved for the 256Kbyte bus address space for the Central Bus. Architected memory modules have HPAs in this area.

The **Available I/O Address Space** is used for the allocation of the 256Kbyte bus address space for additional native busses.

The 4K HPA which is available may not be sufficient to support the normal module functions for some modules. If this is the case, an **Extended Address Space** allows the module to extend the range of addresses to which it responds: this is known as the **Soft Physical Address (SPA)** space for the module. All of the information needed for the allocation of SPA space for a module is provided by that module's IODC. The space is allocated from the **Available I/O Address Space**. The base address of the SPA space allocated to a module is stored in a register in its HPA space.

An example of a module requiring an extended address space would be a bus adapter, which may control many devices on a foreign bus.

The SPA space for memory modules comprises the **Memory Address Space**.

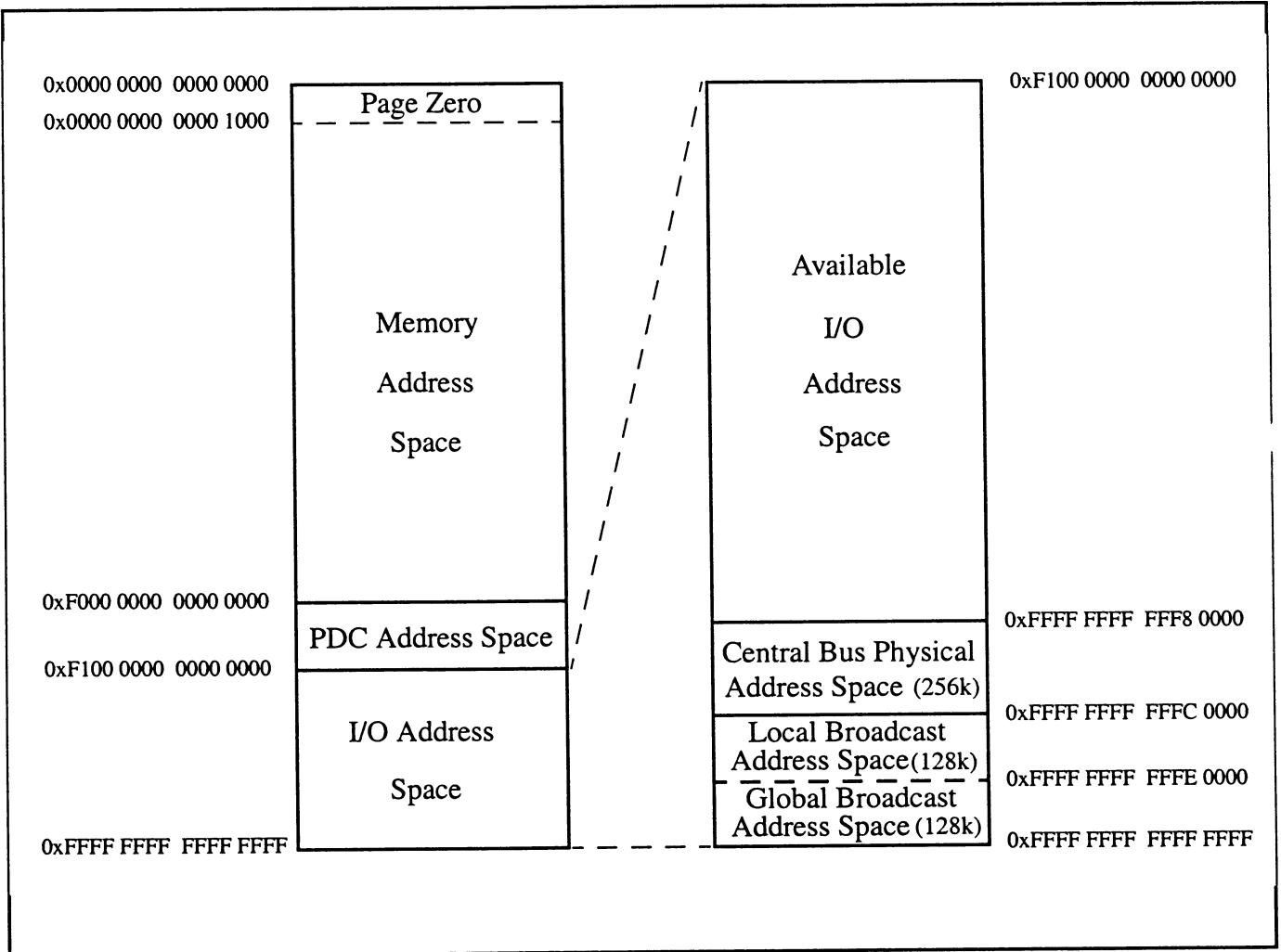
Native processors have the **PDC address space** as their extended address space.

Module 10 — The I/O Subsystem

Left blank intentionally

Module 10 — The I/O Subsystem

Slide: System Address Space: 64-bit



Notes:

Module 10 — The I/O Subsystem

Slide: System Address Space: 64-bit

The address space of a PA-RISC 2.0 system running in **wide mode** uses 64-bit physical addresses¹. The I/O offsets of all modules are the same when the hardware is running in wide mode or narrow mode: the only difference between the two is “F” extending the 32-bit value to 64 bits when running in wide mode.²

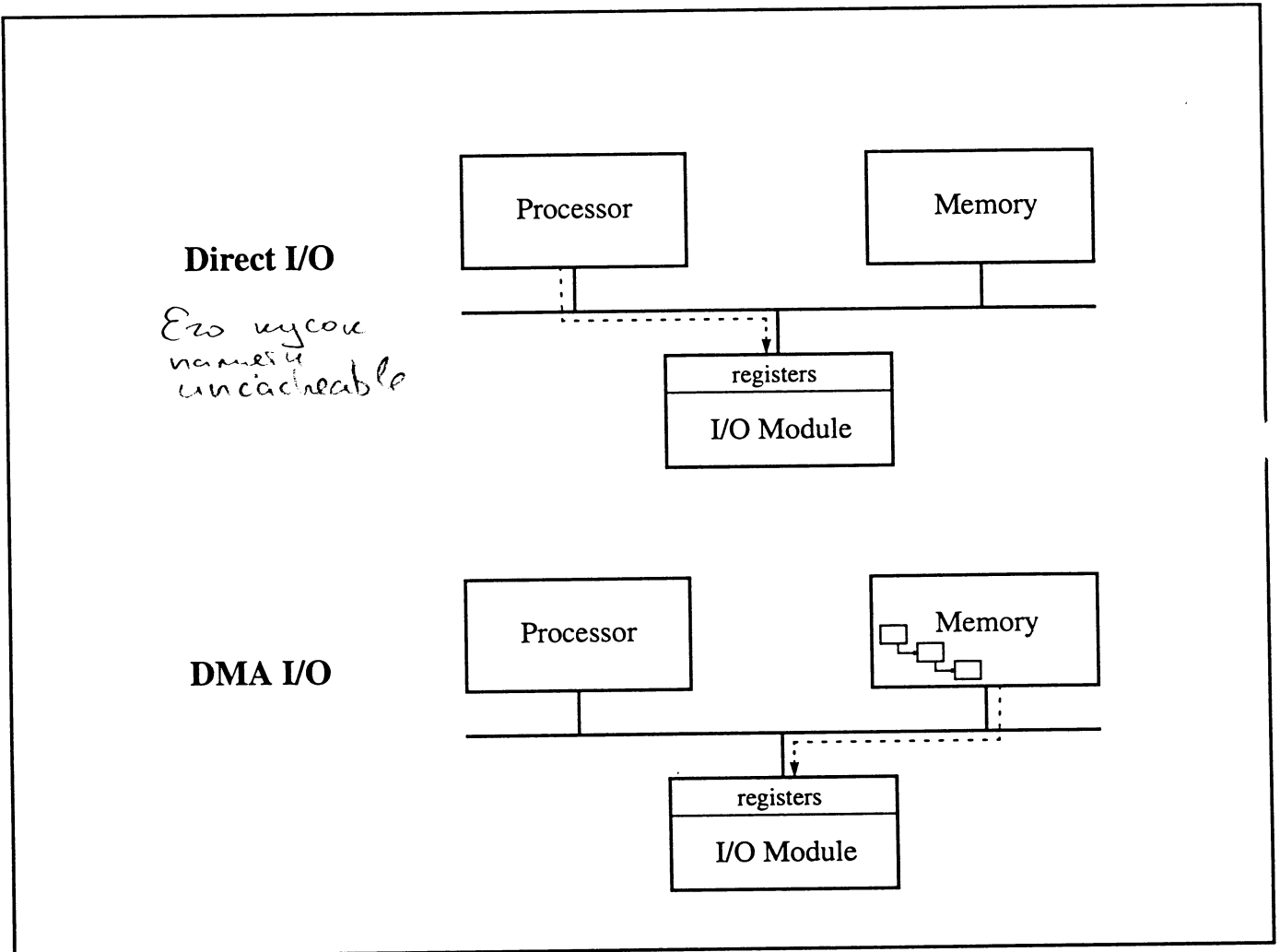
The same subdivisions of the I/O Address Space are used in 64-bit as in 32-bit. Each native bus is still allocated a 256Kbyte HPA which is subdivided into a 4Kbyte HPA for each of the 64 possible modules on that bus.

1. Ref: PA-RISC 2.0 I/O ACD v0.36, section 1.3 “System Address Space”

2. Ref: “HP-UX PA-RISC 2.0 64-bit Address Space Layout Design Overview and Specification” v1.1, OSSD/COSL

Module 10 — The I/O Subsystem

Slide: Modes of I/O Operation



Notes:

Slide: Modes of I/O Operation

Direct I/O

Data transfer to and from the I/O module is accomplished by the processor, using STORE and LOAD instructions to the I/O registers on the module.

Advantages:

- easy to implement in hardware
- inexpensive (allows the connection of industry-standard I/O chips to precision systems, also boards can be less complex)
- high performance for small data transfers

Disadvantages:

- involves significant processor overhead
- lower performance for large data transfers

Modules implementing this mode of operation are known as **Type-A Direct¹** I/O modules. Such modules cannot master any bus operations and send only broadcast interrupts, to a fixed bit of every processor's External Interrupt Request Register (EIRR).

Sequencing of I/O Operations and multiple data transfers must be controlled entirely by the software (the driver).

Examples:

- Centronics module on the 28655A "Skunk" HP-PB Single-ended SCSI / parallel card
- J2094A "Solaris" HP-PB 16-channel modem multiplexer

Direct Memory Access (DMA) I/O

cache coherency support!

Data are transferred to and from the I/O module directly to and from memory.

Advantages:

- I/O transfer is managed by the I/O module, with minimal processor intervention
- more efficient than Direct I/O (for larger transfers)
- reduces interrupt overhead associated with multiple-block transfers

Disadvantages:

- more complex to implement in hardware
- more expensive than Direct I/O modules

1. PA-RISC 1.1 I/O ACD v0.96, Section 1.4 "Modules and Pseudo-modules"

Slide: Modes of I/O Operation

Two different types of DMA I/O module are defined. Both types access I/O registers in order to generate interrupts, and can be programmed to send interrupts to any bit of any processor's EIRR.

Type-B DMA

no master

I/O modules are used to control high-speed I/O devices such as networks. Such modules can fetch commands from memory, transfer data to or from memory, and return status in **completion lists** semi-autonomously. DMA is used to execute an architected linked list structure known as a **DMA chain (command chaining)** and move multiple blocks of data (**data chaining**) without incurring continual processor overhead.

Examples:

- J2146A "Miura" HP-PB LAN card
- "Diablo" LAN module in the "Nova" family SCSI/Console/LAN Multifunction I/O (MFIO) card

Type-A DMA

HP-OSB

I/O modules implement a simpler form of DMA than Type-B. Command chaining and completion lists are not supported, but data chaining is.

Examples:

- SCSI module in the 28655A "Skunk" HP-PB Single-ended SCSI / parallel host adapter. This module uses the "SPIFI" SCSI controller chip.
- 28969A "Wizard" HP-PB Fast-wide SCSI host adapter. This module uses the NCR "Zalon" SCSI controller chip.
- J2468A "Alaxis" HP-HSC ATM/622 card
- A3404A "Baby Hugo" HP-HSC Fiber Channel Mass Storage card.

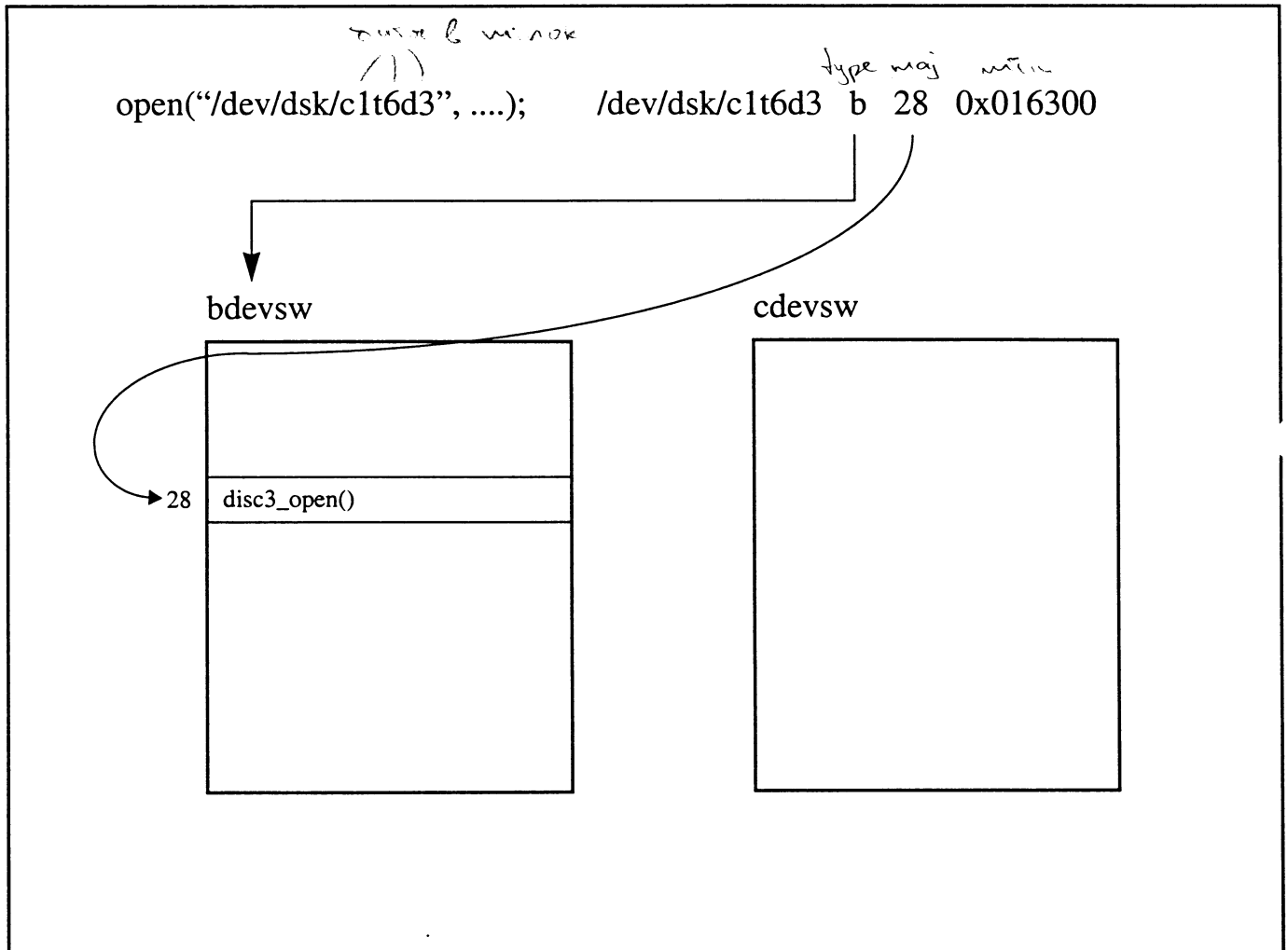
Note: the DMA engine on HP-PB cards is typically either "Lo-Quix" or "Shazam".

Module 10 — The I/O Subsystem

Left blank intentionally

Module 10 — The I/O Subsystem

Slide: Device Files and the Switch Tables



Notes:

На minor' засто передаются логические параметры устройства

Major + minor = 2 байта
1 байт

Динамический major number

Slide: Device Files and the Switch Tables

Device File Naming Convention

There are no real conventions for the naming of device files on Unix systems: IBM, SGI, USL (on Motorola and Intel platforms) and SunOS 4.x all use different formats. The current convention for HP-UX was adopted at revision 10.0¹ and comes from USL's SVR4 (Intel version), which is also followed in Sun's Solaris 2 (SunOS 5.x) for disks. This uses the format: `c#t#d#[s#]`

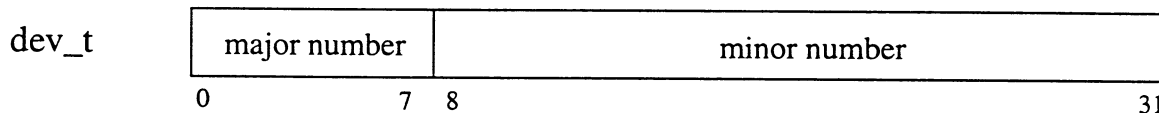
The significance of each field is:

- c - card instance
- t - target (e.g. SCSI address)
- d - device (e.g. SCSI LUN)
- s - section

It is not necessary for there to be any relationship between the name of the device file and the actual minor number for the device (as was the case in HP-UX 9.X and earlier releases). Responsibility for interpreting the minor number lies with the driver, while the naming convention is system-wide.

Device Number

Each device file has an associated device number (`dev_t`), which contains both major and minor number information for the device:



The minor number has no significance to the general I/O system: its use is specific to the environment in which the driver operates and it may either be interpreted by the CDIO (qv) or the individual driver.

Major Numbers

The major number associated with a device and specified in the device file identifies the driver which controls that device: it is used as an index to one of two *switch tables*, which vector to the appropriate driver entry points for block (`bdevsw`) and character (`cdevsw`) device operations. The contents of these tables will be discussed in more detail later in this section.

The 8-bit representation of major numbers results in a limit of 256 discrete major numbers for both block and character devices. To reduce the impact of this restriction, and to avoid the problems resulting when multiple drivers use the same fixed major number, HP-UX provides a mechanism for the dynamic allocation of major numbers². This operation is carried out during the I/O system configuration phase of the system initialization process³ and the mappings are recorded in the `ioconfig` file.

`lsdev(1M)` displays major number allocations and `rmsf(1M)` allows for the dissociation of mappings. For devices associated with a driver which uses dynamic major number assignment, `lsdev(1M)` should be used to determine the actual major number, before device files are created.

1. HP-UX 10.0 I/O User Interface ERS, v1.4 OSSD/COSL

2. `io/gio_kdev.c` - procedure `gio_allocate_majors()`

3. `io/gio_lvl2.c` - procedure `gio_load_ioconfig_file()`

Slide: The ioconfig File

/etc/ioconfig and /stand/ioconfig

ioconfig_t	name	class	hw_path	instance
	c720	ext_bus	10/0	0
	sdisk	disk	10/0.5.0	0
	sdisk	disk	10/0.6.0	1
	mux2	tty	10/4/0	0
	scsi3	ext_bus	10/4/4	1
	target	target	10/4/4.6	3
	disc3	disk	10/4/4.6.0	2
	lanmux0	lanmux	10/4/8	0
	lan3	lan	10/4/8/1	0
	lan2	lan	10/12/6	1

dyn_major_t	name	b_major	c_major
	telm	75	-1
	tels	76	-1

Notes:

Coordinates major's to the hardware
For general use the general code
code

Slide: The ioconfig File

The ioconfig file

This file is used to retain information on the system's I/O configuration across reboots. Two copies are kept: `/etc/ioconfig` and `/stand/ioconfig`

A second copy was placed in `/stand` because NFS diskless clients¹ are not guaranteed to have a reliable `/etc` directory at boot time. The two files are compared by `ioinit(1M)`² which is run from `/etc/iointrc`, invoked from the "ioin" label of `/etc/inittab`. If they differ, the copy in `/stand` is overwritten with that from `/etc` and the system is (optionally) rebooted.

The `ioconfig` file³ contains two types of information⁴:

- mappings of dynamically allocation major numbers to drivers (`dyn_major_t`)
- mappings of instance numbers to hardware paths

Instance Numbers

Card instance numbers uniquely identify individual cards within a specific *class* of cards: a class is a logical grouping of cards which have similar attributes. These include:

- `ext_bus` - cards which support remote devices (e.g. SCSI)
- `lan` - LAN interfaces (e.g. 100BT, FDDI, Ethernet)
- `tty` - cards which support terminal devices (e.g. multiplexers)

The class to which a card belongs is determined by the interface driver which claims that card. The actual instance number values are assigned in the order in which cards are bound to their drivers (which may not be the expected order). Values may not be consecutive: they are always unique within a class, but may be repeated across classes.

As long as the `ioconfig` files remain intact, the card instance number associated with a particular hardware path will remain the same, even if the system configuration is modified by the addition of removal of other cards.

Only **card** instance numbers are guaranteed to remain unchanged across boots.

1. No longer supported at HP-UX 11.0

2. `ioinit(1M)` source module `ioinit.c`, procedures `check_consistency()` and `force_consistency()`

3. `man ioconfig(4)` - contents of the `ioconfig` file. Note the version shipped with 11.0 is outdated

4. `io/ioparams.h` - header file containing definitions actually used in the `ioconfig` file

Slide: The Converged I/O System

Goals of I/O Convergence

- converge ^{workstation} WSIO and ^{server} SIO systems
- protect current driver investment
- extent high availability features to new I/O system
- allow for future movement to industry-standard driver environment
- provide open environment for third-party driver developers
- provide a viable upgrade path to customers
- avoid performance degradation

Notes:

Slide: The Converged I/O System

The I/O subsystems on 700-series (Workstation I/O or WSIO) and 800-series (Server or SIO) systems were separate at HP-UX 9.X and earlier releases. At HP-UX 10.X the **Converged I/O System**¹ was introduced, with the following goals:

1. **Converge WSIO and SIO**

It was recognized that the hardware was converging to such an extent that the support the I/O in the forthcoming K-class and other systems would require aspects of both WSIO and SIO (e.g. the core I/O previously used in the 712 Gecko workstation exists in the same chassis as an HP-PB expansion bus). The goal was to converge the I/O systems such that drivers from both I/O systems could co-exist in the same system. This included the convergence of the processes used to generate and configure a kernel.

2. **Protect current driver investment**

Much time and effort was already invested in the development of drivers for both WSIO and SIO systems. To avoid the rewriting of drivers from either system to the semantics of the other, the converged I/O system was required to support both driver environments with minimal impact on the drivers themselves.

3. **Extend high availability features to new I/O system**

Features which had hitherto only been available on SIO systems (such as LVM, device powerfail recovery, SwitchOver/ServiceGuard and diagnostics) were also to become available on the WSIO driver set.

4. **Allow for future movement to industry-standard driver environment**

The I/O system was required to be flexible enough to adopt industry-standard driver environments as definitions became available.

5. **Provide an open environment for third-party driver developers**

It was noted that third parties were developing drivers for WSIO systems. This open environment was to be preserved in the converged I/O system and extended to cover similar interfaces in the SIO model.

6. **Provide a viable upgrade path to customers**

Although it was understood that some customer-visible differences between the previous systems and the converged I/O system were inevitable, the upgrade path was required to be well understood and as painless as possible. Any changes to the I/O system resulting in a behavioral change which would be visible to customers was to be well documented.

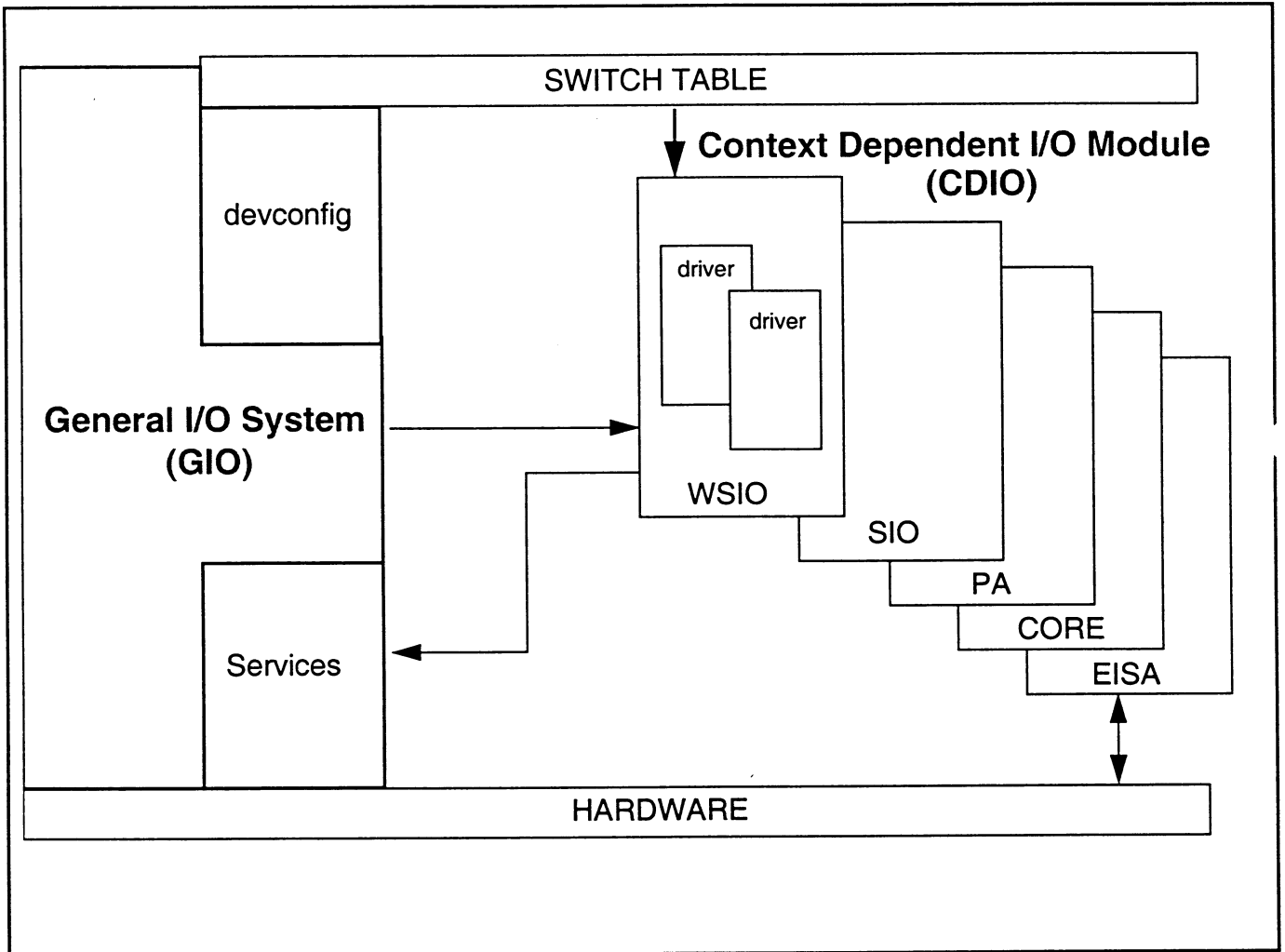
7. **Avoid performance degradation**

The changes were not to cause measurable performance degradation for either system type.

1. HP-UX Converged I/O System ERS v1.2, OSSD/COSL

Module 10 — The I/O Subsystem

Slide: Overview: General I/O & Context Dependent I/O



Notes:

Slide: Overview: General I/O & Context Dependent I/O

The concept behind the converged I/O system is that hardware-specific or driver environment specific functionality should be hidden from the rest of the system behind well defined interfaces.

There are two main elements:

- **General I/O System (GIO)**
- **Context Dependent I/O Modules (CDIOs)**

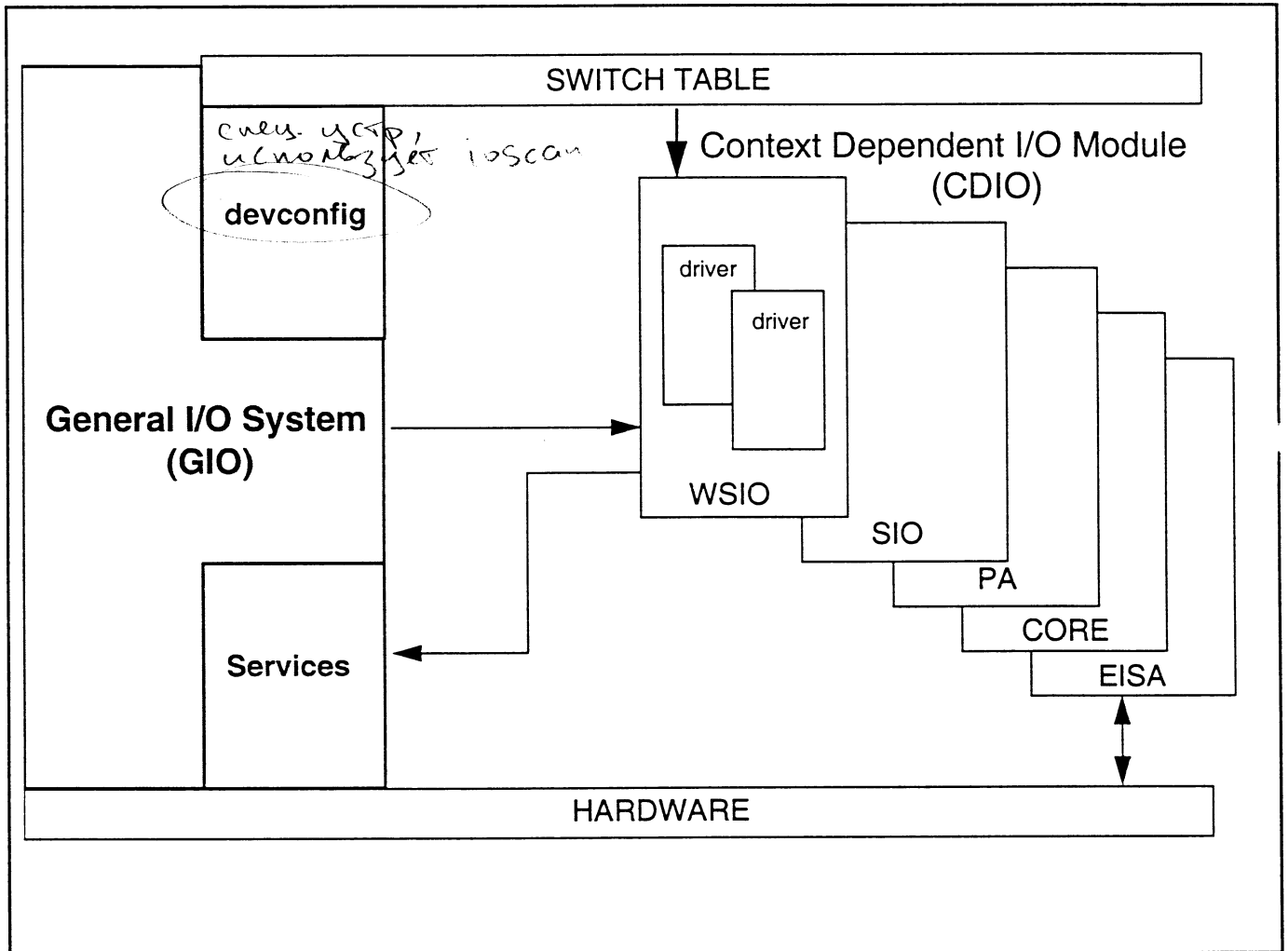
The **GIO** is responsible for all functionality which is global to the system and provides primitive services for the CDIOs.

The **CDIO** contains all bus-specific and/or driver environment specific functionality. CDIOs are configured into to system only as necessary: e.g.

- the kernel on a 700-series Workstation will contain the WSIO, CORE, and PA CDIOs, and might contain EISA, but will not contain SIO
- the kernel on an 800-series server, such as a model G50 “Nova”, will contain the SIO and PA CDIOs but not WSIO or CORE.

Module 10 — The I/O Subsystem

Slide: Overview: General I/O System (GIO)



Notes:

Slide: Overview: General I/O System (GIO)

General I/O System (GIO)

The GIO contains no device specific or environment specific functionality. Its purpose is to manage global I/O resources and data structures, to drive the system configuration process, and provide an interface to the system administration utilities.

Examples of GIO functionality are:

- **Management of I/O Configuration Data Structures**

Data structures which can be manipulated by system administration utilities or which are global to the system must be maintained by the GIO. Examples of these are:

- The I/O Tree node
- Block and Character switch tables
- The Kernel Device Table (KDT)

- **The Algorithms Driving System Configuration**

System configuration is driven by the GIO (although all interaction with interface cards and devices are handled by the CDIOs).

- **The System Administration Interface**

The system administration utilities must see a consistent view of the system, independent of the driver environments. As in the current systems, the *devconfig* pseudo driver will be provided as the system administration interface.

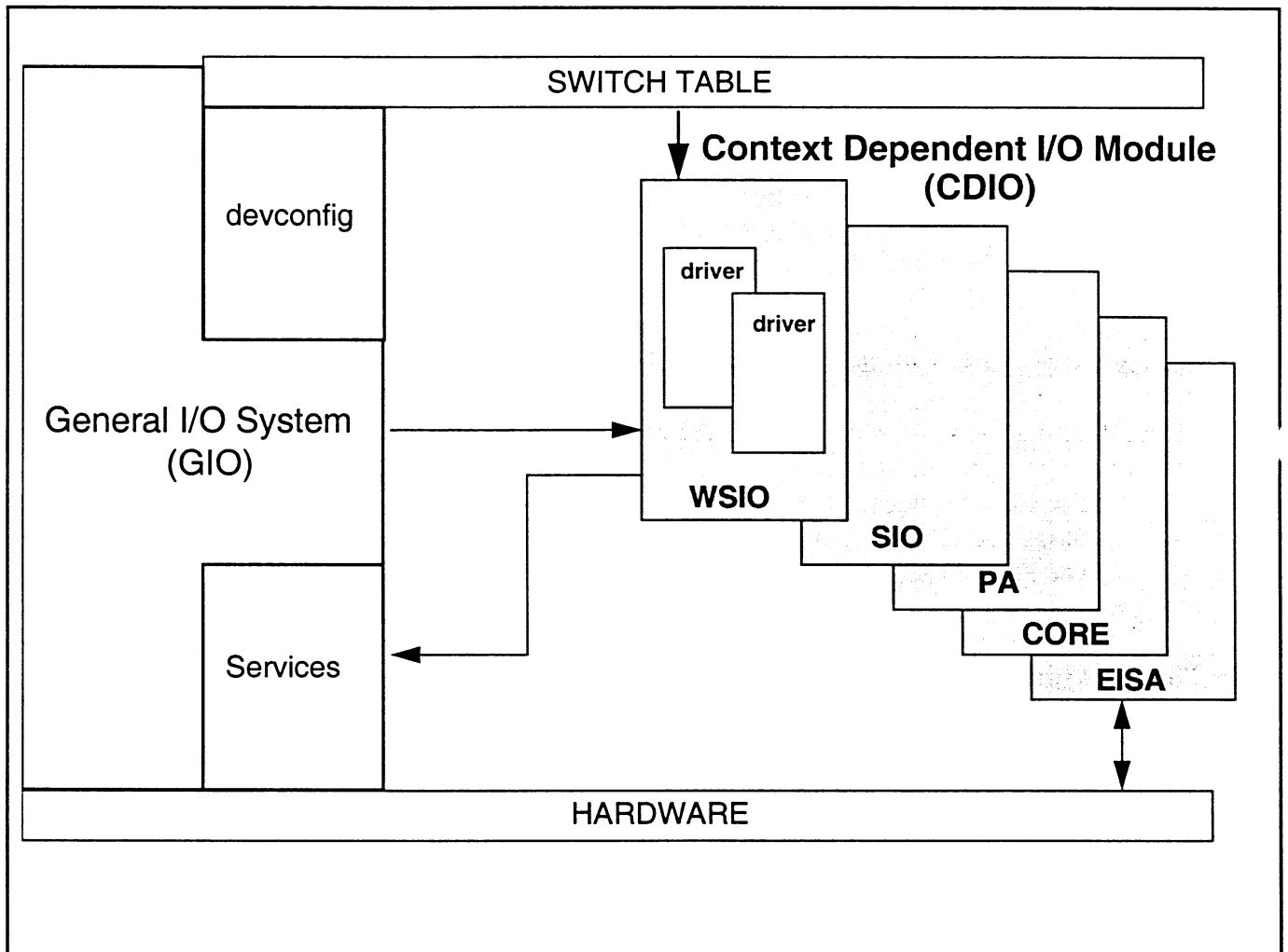
- **Primitive Services**

Other services which are global to the system like inter-CDIO communication services and dynamic driver loading/unloading services will be part of the GIO layer.

The GIO will “not” directly interact with device drivers or devices. With the exception of some GIO services, the GIO is not directly involved with I/O transactions and will not impact system performance.

Module 10 — The I/O Subsystem

Slide: Context Dependent I/O Module (CDIOs)



Notes:

Slide: Context Dependent I/O Module (CDIOs)

Context Dependent I/O Modules (CDIOs)

CDIOs contain functionality specific to a particular bus or driver environment (the context is defined by the bus or driver environment). They will be either pre-configured into the system or loaded dynamically at boot time. CDIOs consist of two forms:

- **Bus-nexus CDIOS**
Provide bus-dependent services to other CDIOs. They may have bus-nexus drivers to control bus adapters or bus converters (e.g. PA, CORE, EISA).
- **Driver Environment CDIOS**
Provide drivers with a defined environment. Drivers within an environment CDIO share a common set of services and defined entry points (e.g. for configuration and diagnostics). Existing versions are:
 - SIO** - each driver has a *port server*, which is required for autoconfig and system administration
 - the main means of communication is message-passing
 - WSIO** - there is no port server concept
 - standard entry points *xxx_init()*, *xxx_attach()* etc. are used to perform configuration-related activities.

The basic components of a CDIO are:

1. **GIO Interface**
The GIO interface contains the entry points invoked by the GIO. Generic configuration requests are converted by the CDIO into the appropriate context dependent functions.
2. **Inter-CDIO Communication Interface**
Inter-CDIO communication is provided via properties and services. Services which allow one CDIO to claim hardware modules found by another CDIO, or to gain access to hardware resources maintained by another CDIO must be provided.
3. **Driver Services**
Services which define a driver environment are part of the CDIO. For example, the SIO service *io_send* will be part of the SIO CDIO. There may be cases where a service in one CDIO is called by a driver in another CDIO. This will be the case, for instance, with some drivers which control the EISA cards. They are part of the WSIO CDIO but call bus-dependent functions from the EISA CDIO. Where possible, bus-dependent functions will be hidden by services in the driver environment to reduce dependencies. For example, the EISA CDIO should not be required to be present on non-EISA systems just because the CORE SCSI is also used for EISA SCSI. This will be accomplished by hiding some of the EISA services behind WSIO services.
4. **Drivers**
A CDIO will in most cases contain drivers. In a *bus-nexus* CDIO like EISA the driver will be the EISA *bus-nexus* manager which is only used to configure the EISA adaptor and provide services specific to EISA. A driver environment CDIO like WSIO can support many drivers which are configurable and may be dynamically loaded or unloaded.
5. **Management of I/O Resources**
PA specific resources like interrupt bits, and I/O PDIR entries will be managed by the PA CDIO.

Slide: GIO Objects

GIO Objects

- I/O Tree**
- Classes**
- Properties**
- Kernel Device Table (KDT)**

Notes:

Slide: GIO Objects

Now that we have the introductions out of the way let us take a closer look at the interfaces and data structures of the **General I/O System** (GIO). Also we will review the interface between the GIO and CDIO.

Data Structures and Objects

Some of the key GIO data structures are “*encapsulated*” as objects. This means that in addition to the data structures themselves there are functions defined which operate on the data structures. Only the defined operations can actually read or modify the data structure for the object.

The GIO objects are:

I/O Tree Object

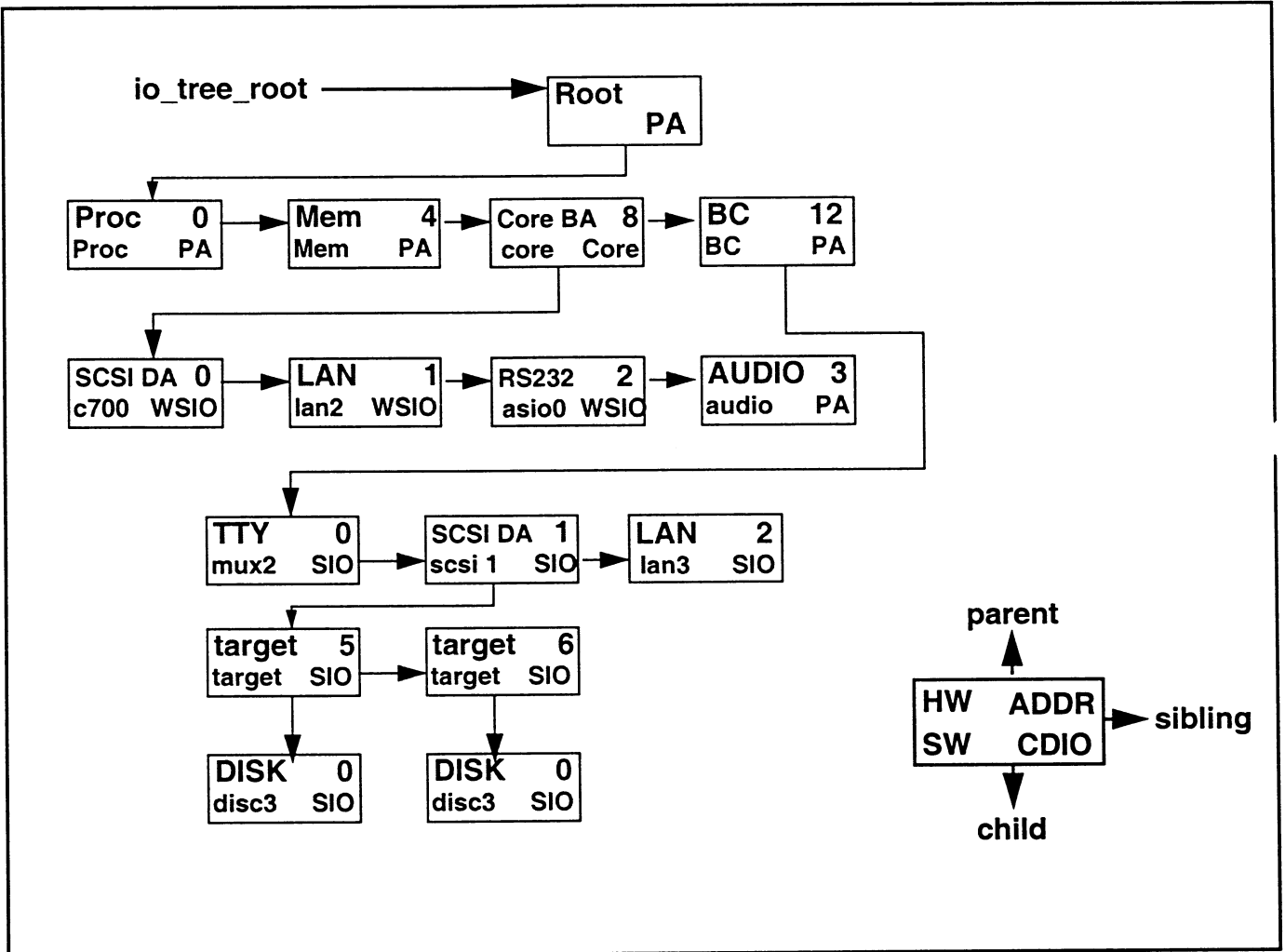
Classes Object

Properties Object

Kernel Device Table (KDT) Object

Module 10 — The I/O Subsystem

Slide: General I/O Tree Objects



Notes:

Класс - номерная единица

Slide: General I/O Tree Objects

The **I/O tree node**¹ is the central data structure of the I/O system. Individual nodes represent elements in the hardware path and contain all of the information necessary for the GIO to manage the associated hardware. The nodes are connected to form a tree which models the hardware in the system. In cases where the hardware has been removed or a CDIO needs a place holder in the tree, there may be nodes in the tree which do not have hardware associated with them. The slide on the previous page shows an example of an I/O tree based on an hypothetical system.

Each node in the tree points to its parent's node. Each **parent** has a pointer to the **child** with the lowest address. This child then points to its next **sibling**. Each node can be owned by one CDIO. The first CDIO to "claim" ownership of a node becomes its owner. This is based on the **io_claim()** function:

io_claim() - *<n>.config.c*

Parameters Passed:

node (input):	the I/O tree node token being claimed
name (input):	name to assign to the node
type (input):	node type (T_DEVICE, T_INTERFACE, etc.)
drv_info (input):	header for the claiming driver

Description: This procedure is called to associate a driver (and the CDIO in which it resides) with an I/O tree node. The **name** and **drv_info** structure must be static since pointers to them will be stored in the I/O tree node. If errors are detected in the **drv_info** structure then GIO_ERROR is returned. Otherwise the state of the node is changed to **S_CLAIMED** and an instance number is assigned to the node (if one is not already assigned).

Significant fields in the **struct io_tree_node** include:

name	The name of this node, as specified by the CDIO which claimed it. Corresponds to the module name of the SIO environment.
parent	pointer to the io_node_t of the parent node
sibling	pointer to the io_node_t for the next sibling node
child	pointer to the io_node_t for the first child node
state	The current state of this node. Defined states are: TRANSPARENT node is for CDIO bookkeeping, there is no associated hardware. Used for software entities such as SIO Logical Device Managers (LDMs) which do not directly control hardware, but do fit into the tree hierarchy. UNCLAIMED The node does not have a software module associated with it SCAN A scan() operation is on progress for this node DIFF_HW The hardware found does not match the associated software NO_HW The hardware at this address is no longer responding ERROR The hardware at this address is responding but is in an error state CLAIMED The node has software associated and no errors.

1. io/gio_priv.h - definition of the I/O tree note, **io_node_t**

Slide: General I/O Tree Objects

<i>type</i>	The type of node as determined by the associated CDIO:
UNKNOWN	There is no hardware associated with this node, or the type of hardware is unknown
PROCESSOR	The node corresponds to a processor
MEMORY	The node corresponds to a memory module
BUS_NEXUS	The node corresponds to a bus converter or bus adapter
INTERFACE	The node corresponds to an interface card
DEVICE	The node corresponds to a device in a remote bus
<i>flags</i>	Flags set for this node:
NEW	The node has been recently created. This condition is cleared by a call to <i>io_clear_new()</i> .
<i>instance</i>	The instance number of the node.
<i>hw_addr</i>	The hardware address of this node <i>relative to its parent</i>
<i>ioconfig</i>	pointer to the <i>ioconfig</i> file data associated with this node
<i>init_func</i>	pointer to the context-dependent initialization function for this node
<i>drv_info</i>	pointer to the <i>drv_info</i> structure for the associated driver. This element defines the driver and CDIO which are associated with the node.
<i>properties</i>	pointer to the property list for this node.
<i>cdio_private</i>	pointer to CDIO private data for this node: only the CDIO which claims the node can use this field.

Module 10 — The I/O Subsystem

Left blank intentionally

Slide: I/O Tree Node State Transition

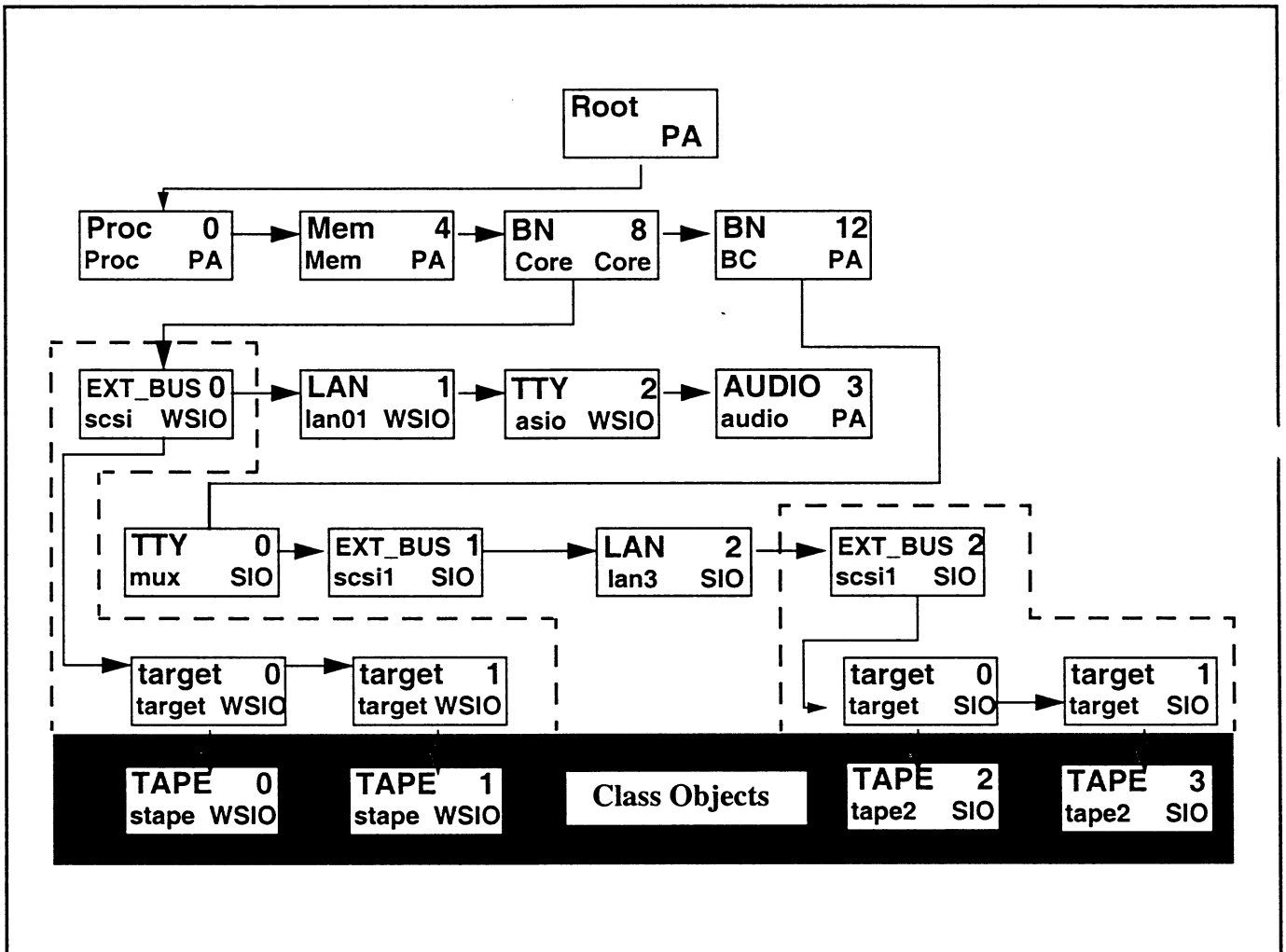
The *scan()* entry point is called by the GIO to either find devices or verify that a device is still responding. This routine will cause the creation of I/O nodes for new devices and cause state changes in the I/O node based on the results of the *scan()*. All existing I/O nodes involved in the *scan()* operation are put into the SCAN state before *scan()* is invoked.

The slide on the previous page shows a state diagram for an I/O node. The state transitions indicated by the bold solid lines are caused by the *scan()* routine. The actions indicated on the previous slide correspond to services which should be called by the *scan()* routine. An example, if the *scan* routine finds a device which is different that the device which was at the same address last time, it should call *io_scan_found_different()* to indicate that fact to the system. If there were no errors and a device was found, then *io_scan_ok()* should be called if the node already exists, or *io_create_child()* should be called if a new node must be created. If the scan routine uses *io_create_child()* to create a node, *prop_create* must be used to create the *id*, and name properties for the new node.

If the child node has no *id* property, it should be treated as if it does not exist. This is the situation for nodes that were created based on */etc/ioconfig* information, and that do not physically exist.

Module 10 — The I/O Subsystem

Slide: GIO Class Objects



Notes:

Slide: GIO Class Objects

Class Object

A class of devices is a logical group of similar devices: for example all tape devices, whether connected via HP-PB HP-IB or EISA SCSI, belong to the tape class, even though they are controlled by different drivers. Instance numbers are assigned by the GIO using `class_assign_instance()`¹ and `io_create_child()`² and are unique within this device class.

The class object is used by the GIO and CDIOs to assign instance numbers and provide the mapping between instance numbers and I/O tree nodes. I/O Tree Node queries reference an I/O tree using pointers of type void. These pointers, which are associated with a particular I/O node, are referred to as tokens: as with the I/O tree nodes, the class data structure must only be accessed via the defined functions in the object.

I/O Class information is stored in a struct `io_class (io_class_t)`³. Useful fields are:

name The class name
node The I/O tree node token for this instance within the class
data Private data for this instance
class_data The base of a dynamically allocated array of node and data pointers which is indexed by the instance number.
num_instance The allocated size of the class_data array

The functions defined for the class object are as follows:

Function	Operation	Description	Used By
<i>class_assign_instance</i>	Operator	Assign an instance within the given class. It is used by <i>io_create_child()</i>	GIO
<i>class_get_data</i>	Query	Gets private data for given class and instance	CDIOs
<i>class_get_name</i>	Query	Returns the name of the given class	
<i>class_get_node</i>	Query	Gets I/O tree node token for given class and instance	GIO and CDIOs
<i>class_put_data</i>	Operator	Assoc. data with given class and instance	CDIOs
<i>class_remove_instance</i>	Operator	Removes the instance for the class. It is used by <i>io_destroy</i>	GIO
<i>io_dev_to_class</i>	Query	Gets a class token given a <i>dev_t</i>	GIO and CDIOs
<i>io_str_to_class</i>	Operator	Gets a class token given the class name	GIO and CDIOs

1. io/gio_class.c

2. io/gio_node.c

3. io/gio_priv.h

Slide: Kernel Device Table (KDT) Objects

Kernel Device Table - KDT

Contains Information about the System's Devices:

- Root Device - rootdev**
- Console Device - cons_mux_dev**
- Dump Device - dumpdevt[]**
- Swap Device - swdevt[]**

Notes:

Slide: Kernel Device Table (KDT) Objects

Kernel Device Table (KDT)

The **Kernel Device Table** (KDT) contains information about the system's special devices (*root, console, swap, and dump*). Entries are generated in the **special devices table**¹ in `/stand/build/conf.c` when the kernel is built and the table is set up during level 2 initialization² based on the information found there³, together with that specified when the system was booted⁴ (which is passed to the booting kernel by the kernel loader⁵ via page zero).

The KDT is not actually a single structure, although the general-purpose routines identified below provide an abstraction layer and conceal this from areas of the kernel which manipulate this data. The record types and associated structures are⁶:

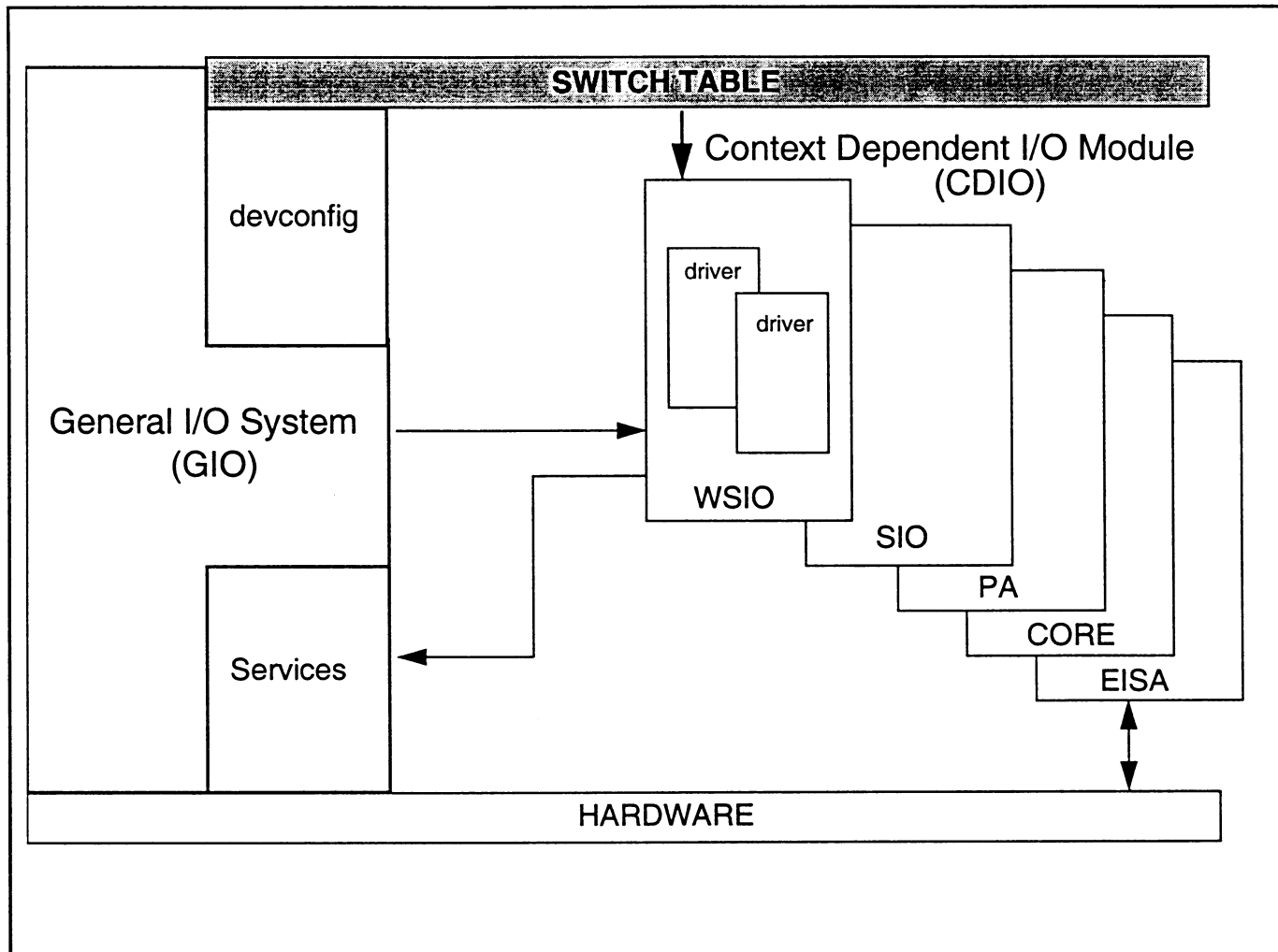
KDT_CONSOLE `cons_mux_dev` (`dev_t` - for the *character* device) and `consflags` (`int`)
KDT_ROOT `rootdev` (`dev_t` - for the *block* device) and `rootflags`
KDT_SWAP `swdevt[0]` (`struct swdevt`) and `swapflags` (`int`)
KDT_DUMP `dumpdevt[]` (`struct dumpdevt`)

The functions to manipulate KDT entries are⁷:

Function	Operation	Description	Used By
<i>kdt_delete</i>	Operator	Remove definition of a kernel special device	LVM
<i>kdt_insert</i>	Operator	Insert definition of a kernel special device	GIO and LVM
<i>kdt_modify</i>	Operator	Modify definition of a kernel special device	GIO and LVM
<i>kdt_read</i>	Operator	get information about a kernel special device	GIO, LVM, and devconfig

1. `h/conf.h` - `struct special_devices`
2. `io/gio_lvl2.c` - procedure `gio_configure_special_devices()`
3. `io/gio_lvl2.c` - procedure `gio_config_from_sdt()`
4. `io/pa_gio.c` - procedure `gio_configure_device()`
5. `mach.800/vm_machdep.c` - definition of `boot_info` and associated struct `bt_info`
6. `io/gio_kdev.c` - declarations of some files which comprise the KDT
7. `io/gio_kdev.c` - for the source to these routines

Slide: Switch Tables



Notes:

Clone driver — это minor драйв
и устройство как major группа и
выбирает первую свободную
(невыключенную, ...)

Module 10 — The I/O Subsystem

Slide: Switch Tables

Two switch tables, **bdevsw** (for block device accesses) and **cdevsw** (for character device accesses), are used to invoke the appropriate driver routines. The tables are indexed by device major number and entries in the table specify the driver functions to be called for particular operations, together with information on the driver. Significant members of the structures are:

- character device switch table entry - **struct cdevsw**

<code>d_open</code>	open processing for character device
<code>d_close</code>	close processing for character device
<code>d_read</code>	read routine for the character device
<code>d_write</code>	write routine for the character device
<code>d_ioctl</code>	<code>ioctl()</code> routine
<code>d_select</code>	select routine for this device type
<code>d_option1</code>	optional "miscellaneous operations" routine for this device type
<code>d_flags</code>	see below for possible flag values
<code>d_drv_info</code>	identifies the driver associated with this entry - a <code>struct drv_info*</code> , see below
<code>d_aio_ops</code>	POSIX Async I/O operations information - a <code>struct aio_ops*</code> ¹

это преамбула
универсальной структуры
39006 не угадал
ф-ция универсальной
(работает или
уже сработала и т.д.)

- block device switch table entry - **struct bdevsw**²

<code>d_open</code>	open processing for block device
<code>d_close</code>	close processing for block device
<code>d_strategy</code>	strategy routine used for read and write accesses to the block device <i>Оутумиз.</i>
<code>d_dump</code>	dump routine (no longer used, formerly used in panic processing)
<code>d_psize</code>	routine to return size information about this device
<code>d_flags</code>	see below for possible values
<code>d_drv_info</code>	identifies the driver associated with this entry - a <code>struct drv_info*</code> , see below
<code>d_aio_ops</code>	POSIX Async I/O operations information - a <code>struct aio_ops*</code>

Values defined for the **d_flag** element are:

<code>C_ALLCLOSES</code>	Call device close on all closes — <i>много раз. где open</i>
<code>C_NODELAY</code>	No write delay on block devices <i>опоздат.</i>
<code>C_MGR_IS_MP</code>	Identifies that the device driver is MP-safe <i>(нельзя...)</i>
<code>C_CLONESMAJOR</code>	driver clones major and minor number (primarily used by Streams)
<code>C_DYN_MAJOR</code>	entry is reserved for dynamic allocation
<code>C_ASSIGNED</code>	major number has been dynamically assigned
<code>C_RQ_CB_AIO</code>	driver has support for POSIX Async I/O request/callback functionality ³
<code>C_MAP_BUFFER_TO_KERNEL</code>	driver needs to remap a buffer to kernel space in physio()

1. POSIX 1003.1b-1993 "Asynchronized I/O", UNIX98 - <http://www.unix-systems.org/version2/unix98.html>

2. `h/conf.h` for the definitions of `struct bdevsw` and `struct cdevsw`

3. Only currently implemented by the LVM, `sdisk` and `disc3` drivers

Slide: Switch Tables

Prior to release 10.X of HP-UX, the block and character switch tables were created during the kernel generation process and placed in *conf.c*, where the associations between major numbers and driver entry points could be observed. In the converged system they are dynamically populated by the driver install routine¹ *install_driver()*, but used by the system in the same way as they were before the adoption of this paradigm. The size of the block and character switch tables is given by the global variables *nblkdev* and *nchrdev* respectively: both are initialized in *conf.c* when the kernel is generated.

The *drv_info* structure

This structure, **struct *drv_info***², is used to define a device driver to the system. It may be built by a CDIO or an individual driver and is passed as an argument to the service routines *install_driver()* and *io_claim()*³. Significant elements of the structure are:

<i>name</i>	name of the driver (module name, for SIO)
<i>class</i>	pointer to the class with which this driver is associated
<i>flags</i>	see below for possible values
<i>b_major</i>	major number for this driver in block mode
<i>c_major</i>	major number for this driver in character mode
<i>cdio</i>	pointer to
<i>gio_private</i>	pointer to additional GIO information for this driver
<i>cdio_private</i>	pointer to additional CDIO information for this driver

Possible values of the flags field are:

<i>DRV_CHAR</i>	character device driver
<i>DRV_BLOCK</i>	block device driver
<i>DRV_PSEUDO</i>	pseudo-driver - i.e. is not directly associated with a physical device
<i>DRV_SCAN</i>	driver provides a <i>scan()</i> ⁴ function
<i>DRV_MP_SAFE</i>	driver provides its own MP protection
<i>DRV_SAVE_CONF</i>	save configuration information to the <i>ioconfig</i> file
<i>DRV_BLKLIST</i>	supports list requests (scatter/gather lists) - disc drivers and LVM only
<i>DRV_LOADABLE</i>	<u>driver is a loadable module</u> ⁵

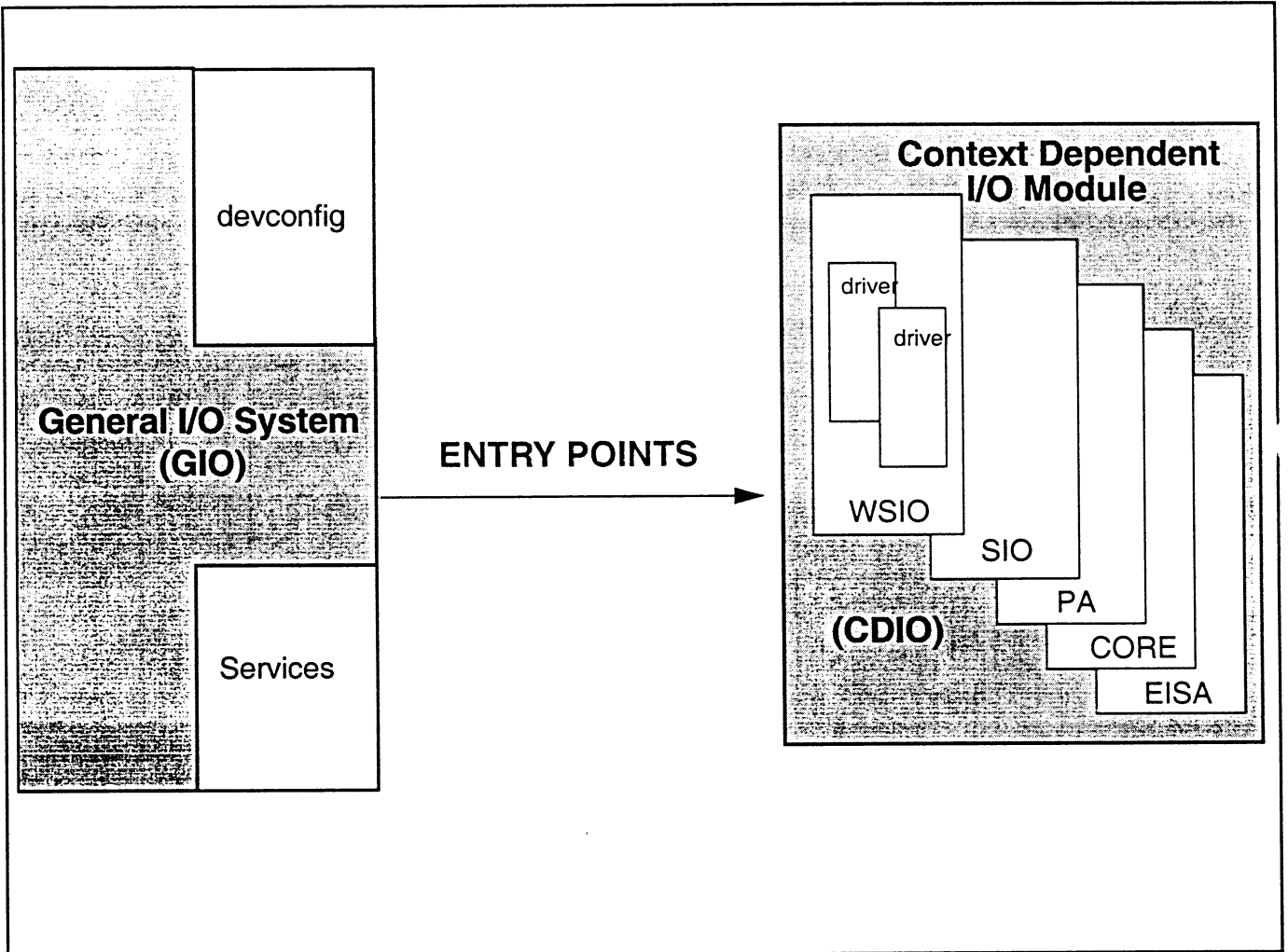
Note: since a pointer to the structure is stored in the I/O tree node, this data structure must be static

-
1. *io/gio_kdev.c*, for the procedure *install_driver()*, *h/conf.h* for the struct *drv_ops* used by this routine
 2. *h/conf.h* for the structure definition of struct *drv_info*
 3. *io/gio_node.c* for the procedure *io_claim()*
 4. man *scan(CDIO2)* in the HP-UX Converged I/O System ERS
 5. *io/gio_modfuncs.c* - *gio_mod_drv_reg()* for DLKM, *streams/str_modfuncs.c* *str_mod_sdrv_reg()* for Streams

Module 10 — The I/O Subsystem

Left blank intentionally

Slide: GIO-CDIO Interface



Notes:

Slide: GIO-CDIO Interface

GIO-CDIO Interface

The GIO invokes CDIOs using the entry points. The naming convention for the entry point is to prefix the function with a unique handle which identifies the CDIO: e.g. the *scan()* entry point for the SIO CDIO is *sio_scan()*.

Entry points are divided into four categories:

- System Configuration
- Node Configuration
- Mapping
- Miscellaneous

System Configuration

These are six system configuration functions which are used to allow the CDIO to perform internal tasks at different points in the initial system boot:

<i>install()</i>	is the first call made to a CDIO. It may be called while the system is still in realmode. Typically just calls the service function <i>install_cdio()</i> to register the CDIO with the GIO.
<i>module_init()</i>	performs <i>real mode I/O</i> configuration tasks. Used to perform configuration tasks which must occur before the Virtual Memory system is configured.
<i>init_begin()</i>	the first call made to the CDIO after the system is in virtual mode: it is intended to allow the CDIO to initialize its internal data structures prior to configuration.
<i>install_drv()</i>	called to pre-install driver into the CDIO. This entry will be called only for drivers which have their header structure built by <i>config(IM)</i> and do not support dynamic loading (WSIO drivers only).
<i>init_middle()</i>	This entry point is called after all pre-loaded drivers have been installed. CDIOs can expect that no more drivers will be added until the system is up and running.
<i>init_end()</i>	This entry point is called at the end of system configuration: it allows CDIOs to perform post-configuration cleanup.

Slide: GIO-CDIO Interface

Node Configuration Functions

There are three functions used for configuration of nodes which may be called during system boot or during normal operation as the result of a system administration operation.

- scan()*** called to scan the hardware to find the children of the given *parent_node*. If new hardware is found, *scan()* will cause the creation of I/O tree nodes via calls to *io_create_child()*. *scan()* must notify the GIO of the results of the hardware scan via calls to *io_scan_found_different()* and *io_scan_error()*.
- config()*** This entry point is called indirectly by the GIO via the **init_func()* field of the node. The appropriate software module is associated with the node using CDIO-dependent algorithms. The **init_func()* pointer is passed to the GIO when the node is created via *io_create_child()*.
- unconfig()*** Un-configures the node. It is called either because the hardware was changed or in preparation for dynamically unloading a driver. All resources in use by the software module must be freed generation

Mapping Functions

These functions are used to provide mapping between the *dev_t* and *io_node*. The GIO has no knowledge of minor number format, but will occasionally have to convert between a *dev_t* and an I/O node.

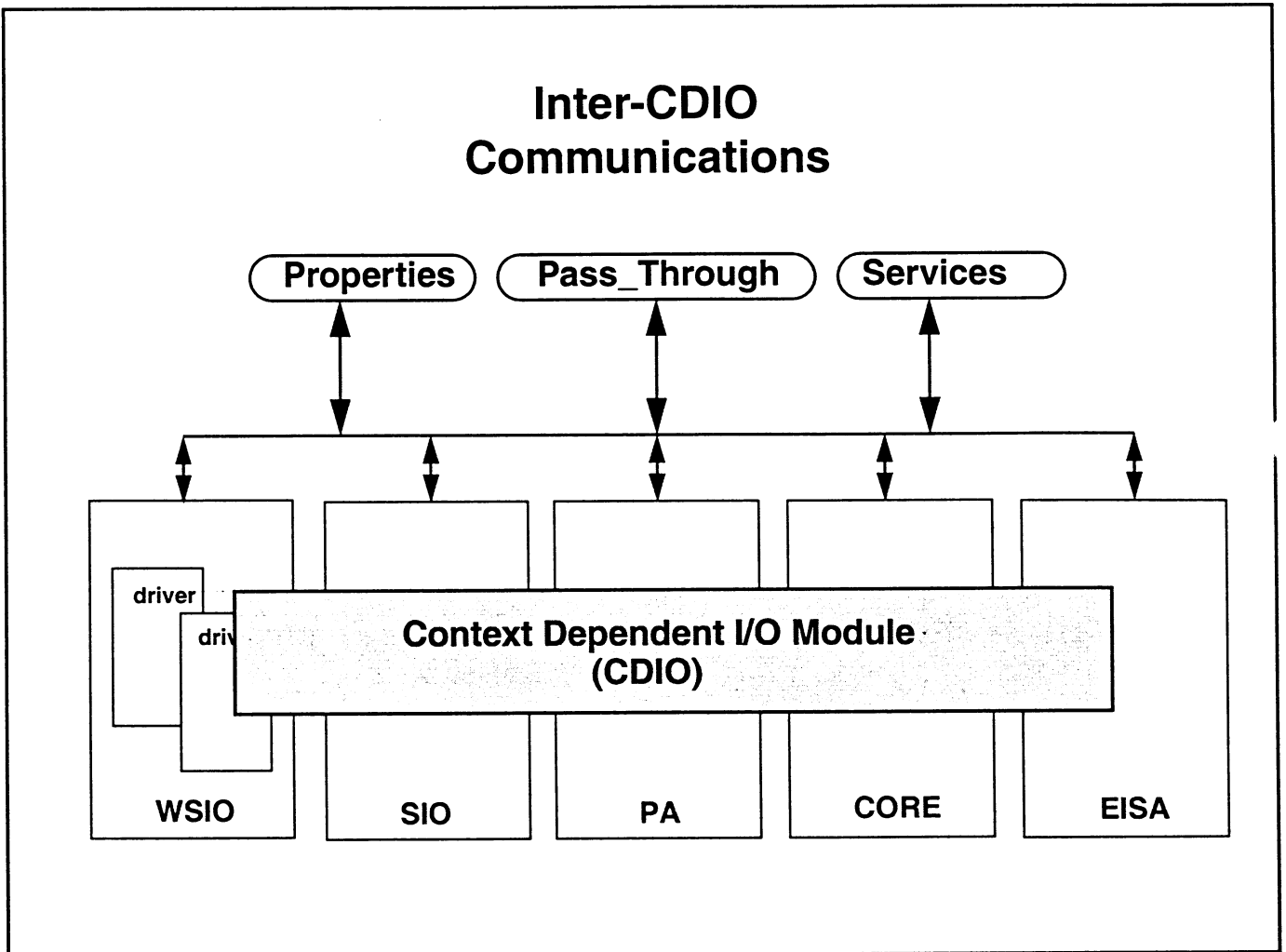
- get_node()*** Returns the node associated with the given *dev_t*
- mkminor()*** Returns the minor number of a *dev_t*, from an options string and the node

Miscellaneous Functions

The *pass_thru()* entry point provides access to CDIO-specific functionality.

Left blank intentionally

Slide: Inter-CDIO Communications



Notes:

It a foreign bus using managed module

Slide: Inter-CDIO Communications

Inter-CDIO Communication

The primary means of Inter-CDIO communication are Properties, Pass-through, and Services.

Properties

Properties are used to propagate information from the CDIO of a parent node to the CDIO of a child.

Two properties are pre-defined:

description A description of the hardware associated with the node: should be an ASCII string containing a product ID or other identifier.

Suggested for all I/O tree nodes

id The identification bytes of the hardware associated with the node. The actual data associated with this property is context-dependent (e.g. IODC bytes for PA modules, inquiry data for SCSI devices).

Required for all I/O tree nodes.

description and **id** must be set up by the *scan()* routines when the I/O node has been created via a call to *io_create_child()*. These properties will be used by the CDIO associated with the child node to identify and claim the node.

Pass-Through Functions

These are a mechanism which allows a function from another CDIO to be invoked, without requiring that CDIO to be present when the kernel is linked. The required function is called using its literal string name, thus avoiding the linker dependency which would result if it were called directly. Pass-through functions are used when no parent-child relationship has been established, but a CDIO needs to access services provided by another CDIO (which may or may not be configured into the system). For example, WSIO drivers may need to claim EISA or CORE modules, therefore WSIO needs to register with the EISA and CORE CDIOs to get a chance to claim the modules. On a given system, however, the EISA CDIO may not be present. To allow WSIO to conditionally register with EISA a pass-through function is used.

WSIO registers with the EISA and CORE using the following calls:

```
cdio_command("eisa","set_claim_func",wsio_config,NULL);
```

```
cdio_command("core","set_claim_func",wsio_config,NULL);
```

If the EISA CDIO is present, then WSIO is registered with EISA and CORE, otherwise it is registered only with CORE.

Services

Services provided by one CDIO may be called directly by another CDIO if the service provider is required to be present in the system for the service user to function. Direct call between CDIOs is discouraged because they cause dependencies which force the presence of one CDIO for another to function. An example is a WSIO driver which controls an EISA card (but not a CORE function): this driver may need to call EISA-specific services which are not hidden behind WSIO services. This is acceptable since the EISA CDIO is required to be present in order for the driver to function.

Slide: Kernel Generation Structure

Kernel Generation Structure

- CDIO Install List**
- Driver Install List**
- Special Device Table**

Notes:

Slide: Kernel Generation Structure

Kernel Generation Structure

The following structures are built by *config(1M)* and put into *conf.c* when the kernel is being created. There is currently no dynamic loading of drivers, so all drivers and CDIOs are configured into the kernel when it is generated, via these structures.

CDIO Install List All CDIOs will be configured into the kernel via the CDIO Install list in *conf.c*. The list contains *install()* entry points for all CDIOs specified in the input file for *config(1M)*. The list is pointed to by the global variable *cdio_install* and must be NULL terminated: order is not important. The GIO uses this list to call the *install()* entry point for each CDIO during the first level of configuration (see System Initialization).
e.g. if the *sio* keyword is present in the */stand/system* file, *config(1M)* will add *sio_install()* to the *cdio_install[]* array in the *conf.c* which is created. This results in the presence of the unresolved symbol *sio_install* in the *conf.o* which is generated: resolving this dependency causes the linker to pull in the *sio* module.

Driver Install List All drivers are pre-loaded either via the Driver Install List or the compatibility mechanism. The driver install list will contain install entry points for all drivers specified in the input file of *config(1M)* which have install entry points. The list is pointed to by the global variable *driver_install* and is NULL terminated: order is not important. The GIO uses the list to call indicated functions with no arguments.
e.g. if the *sdisk* keyword is present in the */stand/system* file, *config(1M)* will add *sdisk_install()* to the *driver_install[]* array in the *conf.c* which is created. This results in the presence of the unresolved symbol *sdisk_install* in the *conf.o* which is generated: resolving this dependency causes the linker to pull in the *sdisk* module.

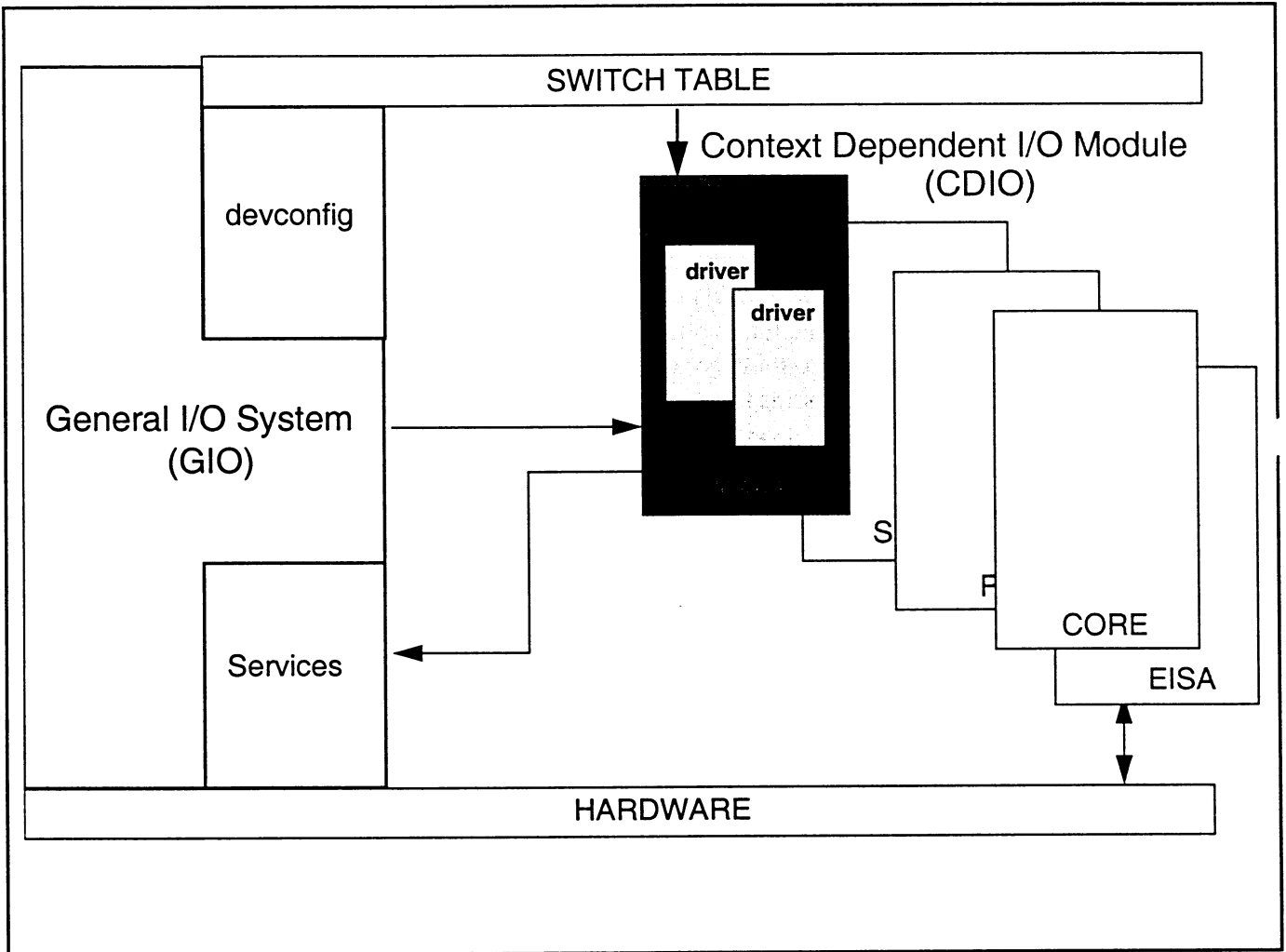
Special Device Table The **Special Device Table (SDT)** indicates to the system which devices are used for primary swap and dump: the **Kernel Device Table (KDT)** is initialized from the information in this table. Each entry in the table is a *struct special_devices*: significant fields are:

<i>special_device_name</i>	type of special device: swap or dump
<i>hw_path</i>	hardware path for the special device, or the string "lvol" if specifying a logical volume
<i>offset</i>	The offset from the beginning of the device, in <i>DEV_BSIZ</i> (1Kbyte) units. -1 indicates that the area after the filesystem is used
<i>blocks</i>	The number of <i>DEV_BSIZ</i> blocks in this specification. 0 indicates all available space

e.g. the presence of the keywords *dump lvol* in the */stand/system* file will cause *config(1M)* to add an entry to the *special_device_table[]* in the resulting *conf.c*.

Module 10 — The I/O Subsystem

Slide: WSIO CDIO



Notes:

STREAMS:

`/usr/include/sys/stropts.h`
 (команды ioctl, b r.2. uspegnara fd)

One messages - copy-on-write

Messages - gannell see komu pytotel
 1. gosabnet pre-headers, post-headers
 2. 3a crez nromexyrob b nnyngic messag

Slide: WSIO CDIO

The WSIO CDIO¹ is a “*Driver Environment CDIO*” whose purpose is to provide drivers with a defined environment: all drivers within the WSIO CDIO share a common set of defined entry points and services. It can be viewed as a buffer zone which protect drivers in its environment from the peculiarities of the bus-nexus CDIOs with which it is configured.

Overall, the CDIO can be conceptually broken into five components:

- GIO Interface
- Inter-CDIO Communication Interface
- Driver Services
- Drivers
- Management of I/O resources

Since the WSIO CDIO is a driver environment CDIO, it is required to provide a consistent environment no matter how it is configured with *Bus-Nexus* CDIOs. Drivers residing within the WSIO CDIO must operate smoothly without knowledge of the underlying configuration: it is the task of the WSIO CDIO to interpret the service calls and take the appropriate actions for the actual configuration.

The WSIO CDIO is responsible for:

- absorbing all interface discrepancies for the driver
- handling configuration issues
- monitoring resources
- hiding all WSIO specific functionality from the generic GIO system configuration, so that the I/O system can continue to run even if WSIO is completely unconfigured

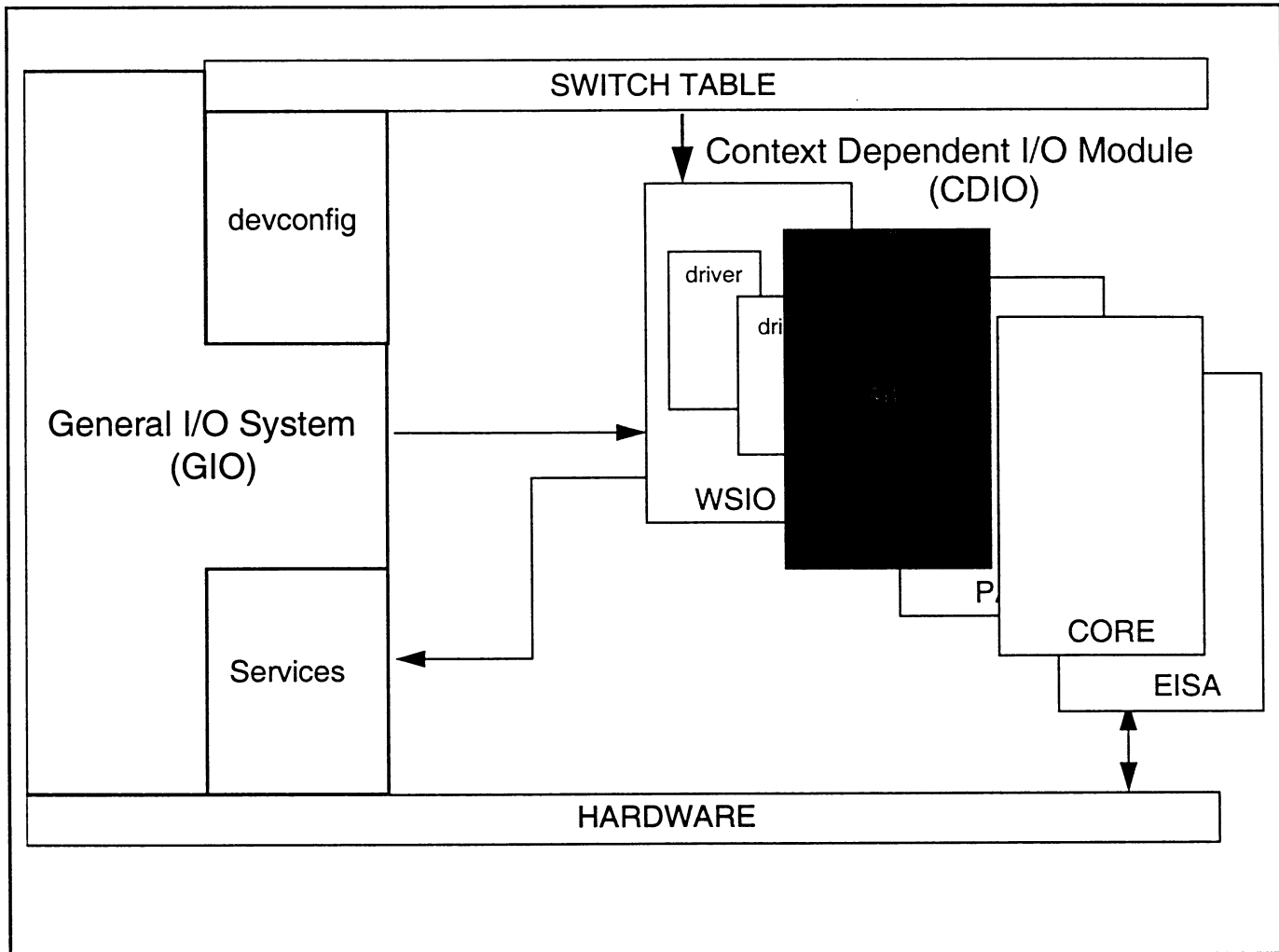
Управление ресурсами в STREAMS
— мост

Служ. могут организовать гал. односторон.
Read(), write() — могут обходить

1. Workstation I/O Context-Dependent I/O Module (WSIO CDIO) ERS, v3.0, COSL/OSSD

Module 10 — The I/O Subsystem

Slide: SIO CDIO



Notes:

Slide: SIO CDIO

The SIO CDIO¹ contains functionality necessary for the support of 800-series style drivers², which can either be pre-configured into the system or loaded dynamically at boot time. The basic components of the SIO CDIO are:

GIO Interface - contains the entry points invoked by the GIO to configure and initialize the driver environment. Generic configuration requests are converted by the CDIO into the appropriate context-dependent functions.

Inter-CDIO Communication Interface - properties and services provide a mechanism for inter-CDIO communication. Services are provided to allow one CDIO to claim hardware modules found by another CDIO, or to gain access to hardware resources maintained by another CDIO.

Driver Services - are services which define a driver environment. Services are routines global to the Series 800 I/O managers that provide the drivers with an interface between other I/O managers and the operating system. For example, the service *io_send()* is part of SIO CDIO.

Drivers - software modules which control I/O devices, e.g. scsi1 (interface driver for the HP-PB Single-ended SCSI host adapter).

Management of I/O resources - resources common to drivers in the SIO environment, such as interrupts, powerfail recovery, diagnostics, I/O configuration, messages and ports, are managed by the SIO CDIO.

The SIO model

Device control is achieved using a hierarchy of driver modules known as **managers**, each of which implements a **port server** to receive requests. The layers of drivers are known as:

Logical Device Manager (LDM) - implements functionality which is generic to a class of devices - e.g. disc3 is the LDM for disk devices

Device Manager (DM) - implements functionality which is specific to a subset of these devices - e.g. disc30 is the DM for SCSI direct-access devices

Device Adapter Manager (DAM) - implements functionality which is specific to a particular interface type - e.g. scsi3 is the DAM for the HP-PB fast-wide SCSI host adapter (“Wizard”).

Channel Adapter Manager (CAM) - not generally used, will be present if a Channel Adapter module is in the I/O path to the device - e.g. cio_ca0 is the CAM for the HP-PB to CIO “ChanSpan” channel adapter³.

Communication between the managers is achieved by a message-passing mechanism implemented by standard routines such as *io_send()*. Information on the allocation of port numbers and structures used in this mechanism is stored in the *Msg_control* structure, which is defined as a **struct msg_control**⁴.

1. SIO CDIO External Specification, v3.1, COSL/OSSD

2. HP-UX I/O Services Definition, v8.0, HP-UX Kernel Lab/General Systems Division

3. Only supported for use with the CIO HP-FL interface and HP-FL is not supported as of 11.0

4. sioserv/ioserv.h - definition of *Msg_control* and the struct *msg_control*

Slide: SIO CDIO

Context-dependent data structures

- The private data area for nodes in the I/O tree which belong to the SIO CDIO is represented by a **struct sio_tree_entry**¹.
- For the SIO CDIO, the context-dependent private data area in the `drv_info` structure is represented by a **struct sio_drv_entry**.
- **sio_hdr_list** is the head of a linked list² of SIO driver headers.
- information regarding the I/O module associated with an SIO node in the I/O tree is stored in a **struct sio_card_info**³.
- the bus type associated with an interface node is represented by a **struct sio_bus_type**⁴.
- pass-through functions which may be invoked by `sio_pass_thru()`⁵ are held in a structure **table[]**⁶ made up of elements which are of type **struct sio_pass_thru_table**⁷. This table is searched using a binary search defined by `io_lookup()`⁸.
- information on interpreting the minor number for a device is held in **mtable[]**⁹ which has elements of type **struct sio_minor_table**¹⁰. `io_lookup()` is used to match entries in this table.

1. `sioserv/sio_cdio.h` - `struct sio_tree_entry (sio_node_entry_t)`

2. `sioserv/sio_cdio.h` - `struct sio_list (sio_drv_info_t)`

3. `sioserv/sio_cdio.h` - `struct sio_card_info (sio_card_info_t)`

4. `sioserv/sio_cdio.h` - `struct sio_bus_type (sio_bus_type_t)`

5. `sioserv/sio_util.c` - for the declaration of the routine `sio_pass_thru()`

6. `sioserv/sio_util.c` - declaration of `table[]`

7. `sioserv/sio_cdio.h` - `struct sio_pass_thru_table (sio_pass_thru_table_t)`

8. `io/io_minor.h` - macro definition for `io_lookup()`

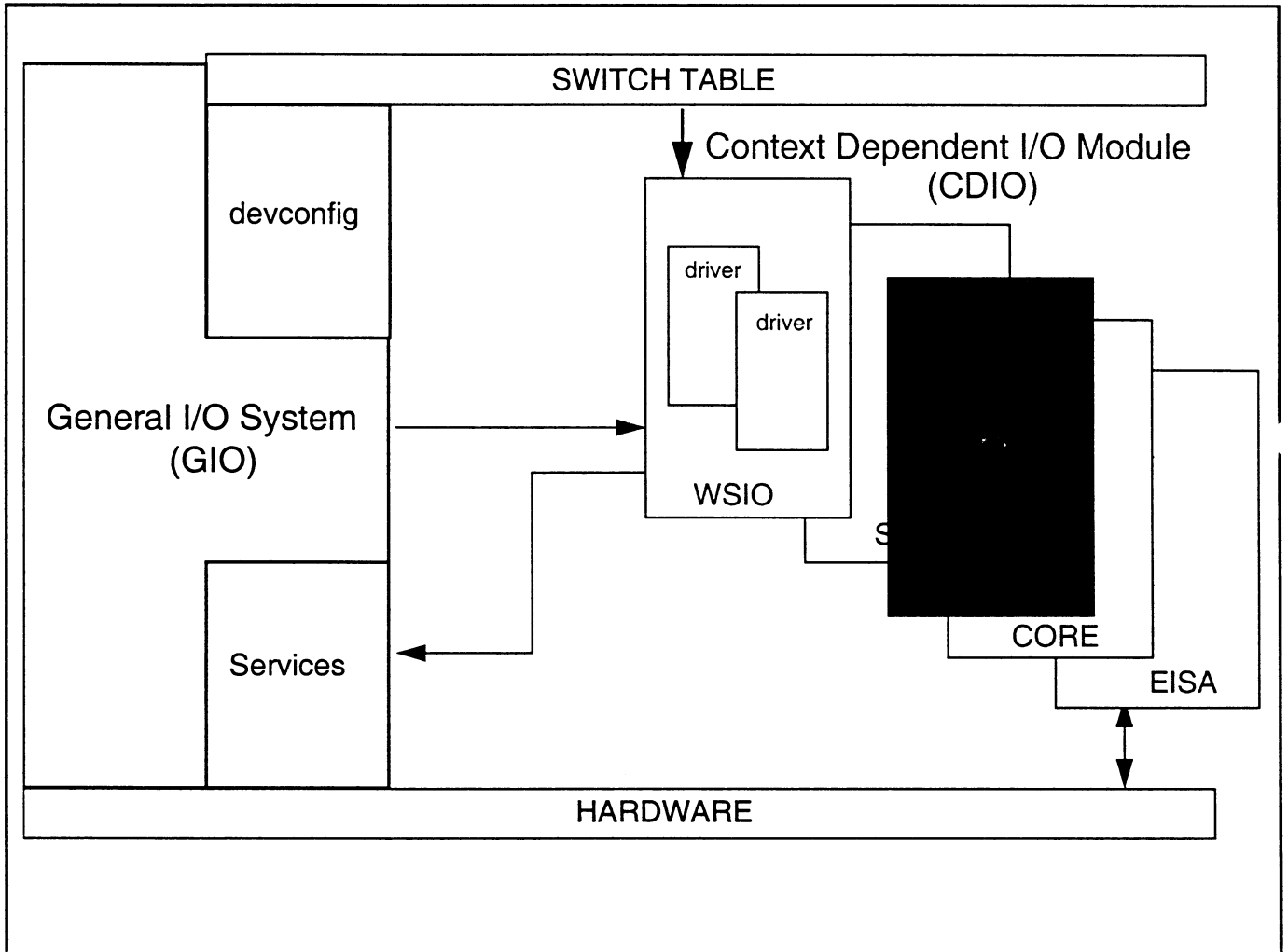
9. `sioserv/sio_node_config.c` - declaration of `mtable[]`

10. `sioserv/sio_cdio.h` - `struct sio_minor_table (sio_minor_table_t)`

Left blank intentionally

Module 10 — The I/O Subsystem

Slide: PA CDIO



Notes:

Slide: PA CDIO

PA CDIO

This section describes the interfaces and data structures for the **Precision Architecture (PA) CDIO**. The PA CDIO is responsible for finding and identifying HP-PA modules, managing PA bus-converters, and managing PA-dependent resources such as PA interrupts and address mapping functions across PA bus-converters.

IODC Data

This structure, **struct iodc_data**¹, represents the IODC information² for all PA modules: it is also used as the “*id*” property for all such modules. Significant fields include:

hv_model	hardware version number (HVERSION) - model field. Specifies the hardware implementation.
hv_rev	HVERSION - revision field
spa_io	soft physical address (SPA) capabilities - io flag. Specifies if SPA is in memory or I/O space.
spa_shift	SPA - shift field. Specifies maximum SPA space size - type-dependent.
tp_type	type. Specifies the module type ³ .
sv_rev:	software version number (SVERSION) - revision field.
sv_model	SVERSION - model field.
sv_opt	SVERSION - opt field.
iodc_length	length of the entry point table, in words.

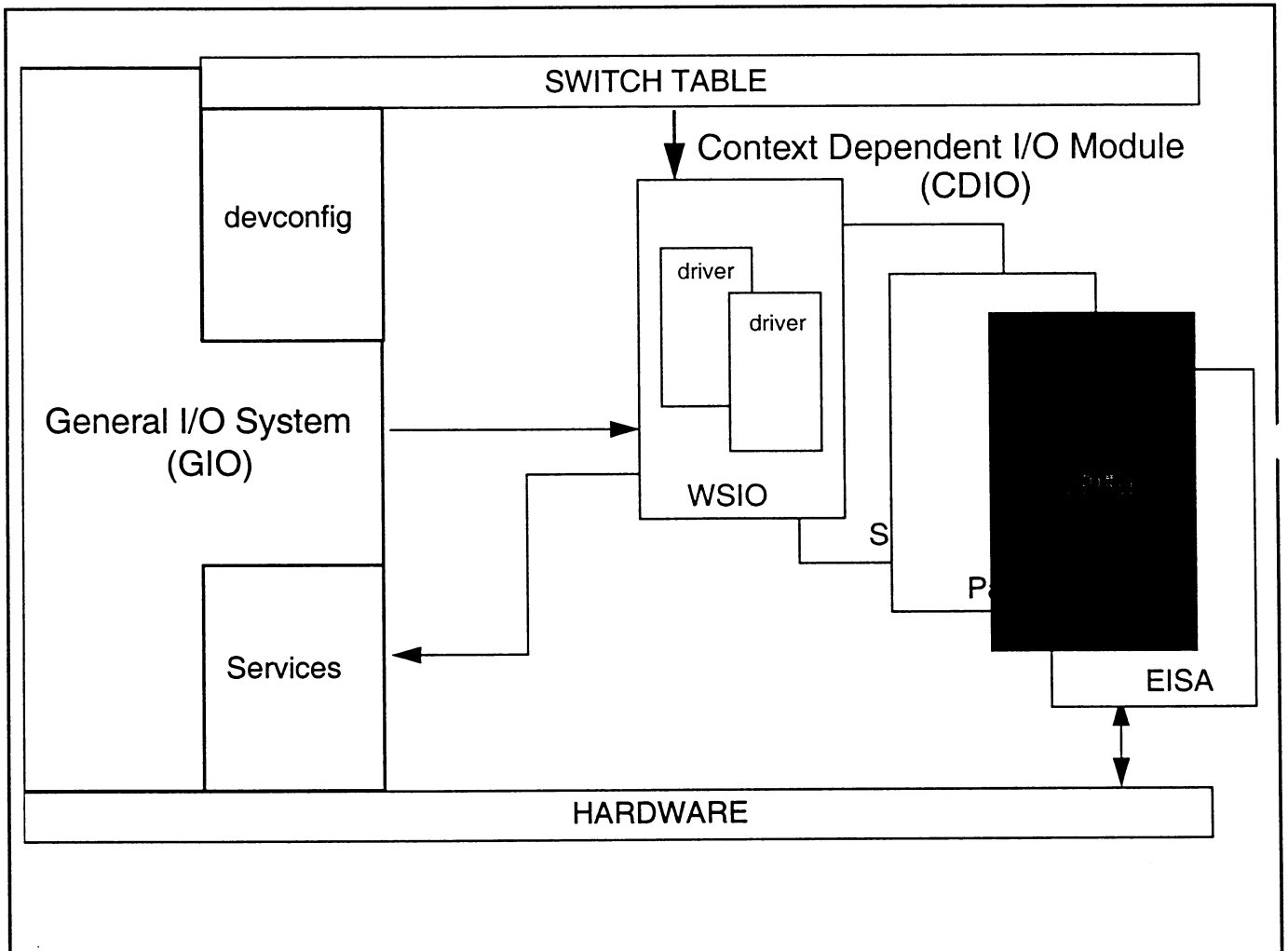
1. io/pa.h - declaration of struct iodc_data (iodc_data_t)

2. PA-RISC 1.1 I/O ACD Section 13 “IODC”, page 13-2 “IODC Data Bytes”

3. mach.800/cpu.h - #defines for MOD_TYPE_*

Module 10 — The I/O Subsystem

Slide: CORE CDIO



Notes:

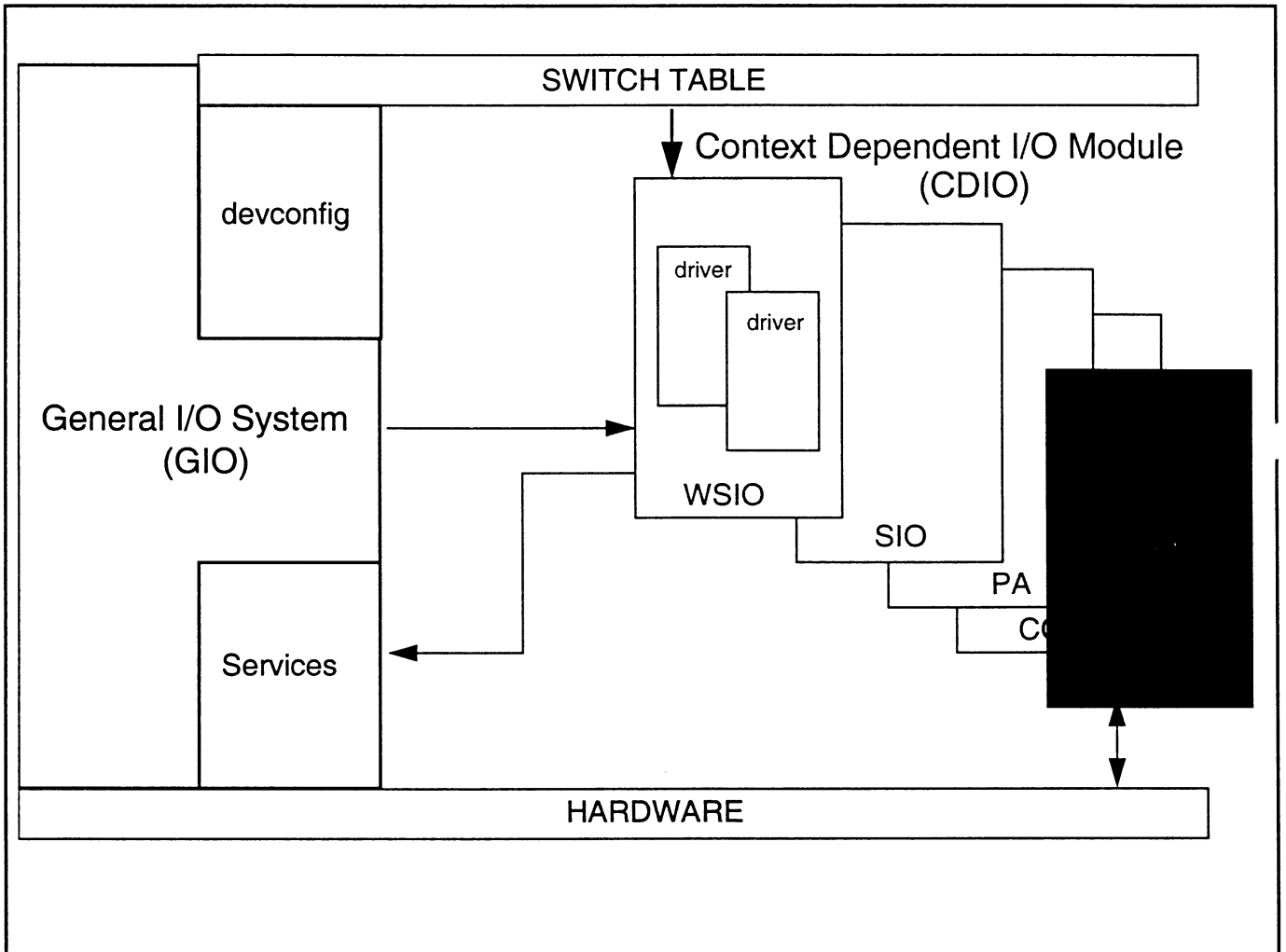
Slide: CORE CDIO

CORE CDIO

The CDIO controls the multi-function core I/O modules based on the LASI, WAX, ASP, and Stiletto (and compatible) ASICS. The CDIO is responsible for managing the Core I/O adapter and interrupt controller, and providing a mechanism for drivers to become associated with the Core I/O functions.

Module 10 — The I/O Subsystem

Slide: EISA CDIO



Notes:

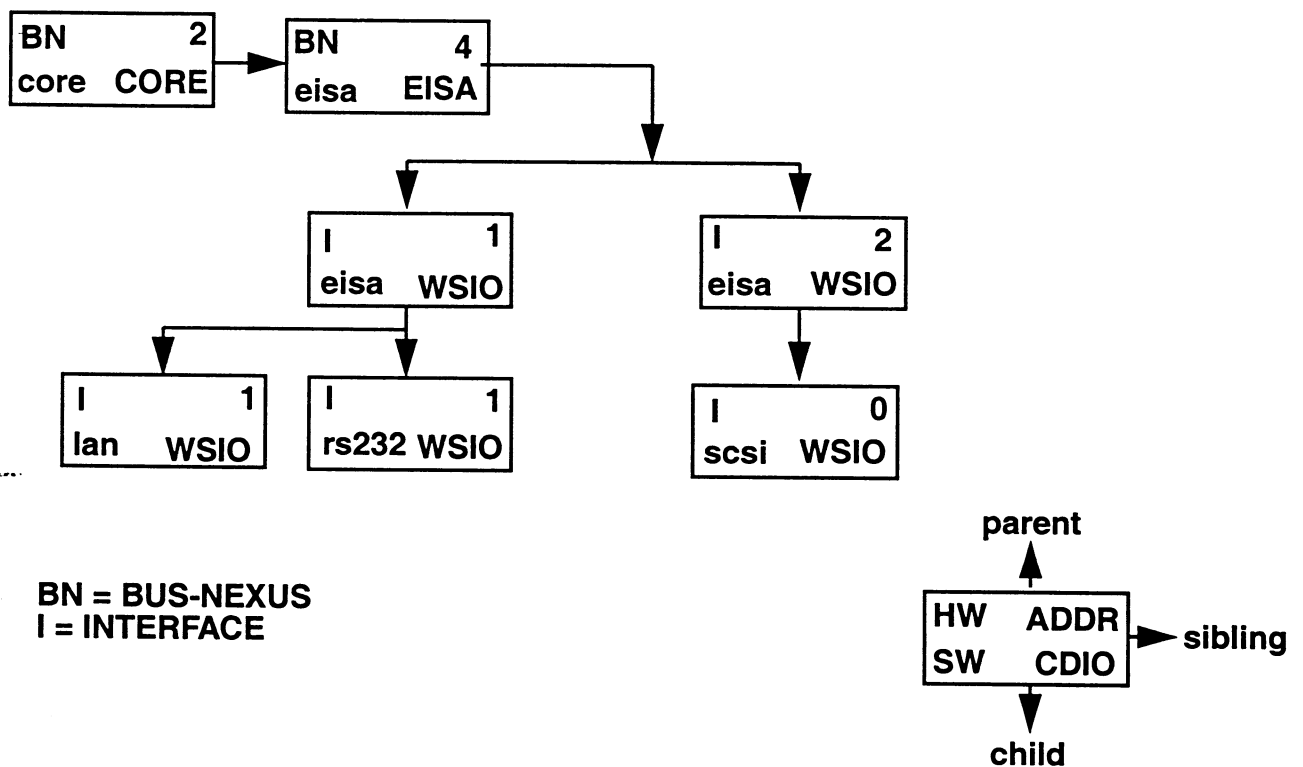
Устройство и на зарпуге ера,
загб и3 ит

Slide: EISA CDIO

EISA CDIO

This slide describes the interfaces and data structures of the EISA CDIO. This CDIO is responsible for managing the EISA I/O adapter and providing a mechanism for drivers to become associated with the EISA I/O sub-system and functions.

An EISA sub-system consists of a system board and one or more slots that can hold interface cards. Each interface card consists of one or more functions that may be controlled by different drivers. In 10.X this hierarchy is modeled by the I/O tree. The diagram below illustrates a branch of the I/O tree that represents a hypothetical EISA sub-system.



In the above example, the EISA sub-system lies beneath CORE. It consists of a system board and two interface cards in slot 1 and 2. The first card is a multi-function card that has both lan and rs232 interfaces and the second is a single function card with a SCSI interface. The I/O tree node that describes the system board is the second from the top and is of type *Bus_Nexus*. The two nodes directly underneath it represent the interface cards and are of type INTERFACE. Beneath these are the nodes that represent the card functions, they are also of type INTERFACE. Note that the card and function nodes belong to the WSIO CDIO whereas the *Bus_Nexus* belongs to EISA. This is because an interface driver in a driver environment CDIO needs to recognize and configure the card function.

Module 10 — The I/O Subsystem

Slide: EISA CDIO

Module 10 — The I/O Subsystem

Lab: The I/O Subsystem

The io tree is made out of struct `io_tree_nodes`, and pointed to by `io_tree_root`. Each node has a pointer to its parent, child, and a sibling. Using Q4 to read the `io_tree_nodes` draw your system's `io_configuration`. Check out what you draw against the output of `ioscan -f`.

Use the commands

```
load struct io_tree_node from <<where ever>>
print -x name_buffer instance %d hw_addr %d parent sibling child
```

Q4 can not follow two sets of linked lists at once but you can walk either down through all children or sideways through all siblings in one go.

Load the root.

```
q4>
q4> load struct io_tree_node from io_tree_root
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr parent sibling child
"root" 0 -1 0 0 0x100005100
```

The root has no parent or sibling, but does have a child, so follow that.

```
q4> load struct io_tree_node from child
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr parent sibling child
"ccio" 1 8 0x89e000 0x100005d00 0x100005180
```

Here there is a sibling so walk along this level. it's actually the top level in the system.

```
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr parent sibling child
"ccio" 2 10 0x89e000 0x100170780 0x100005d80
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr parent sibling child
"processor" 0 32 0x89e000 0x100170800 0
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr parent sibling child
"memory" 0 49 0x89e000 0 0
q4>
```

Module 10 — The I/O Subsystem

Lab: The I/O Subsystem

Here there are no more siblings, so we have reached the end of that level in the system, go down to the next level through the child pointer of the first ccio.

```
q4>
q4> load struct io_tree_node from 0x100005180
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent  sibling  child
"GSCToPCI"          0           0 0x100005100 0x100005680 0x100009500
```

This has both sibling and child pointers, so lets carry along this level by following the siblings.

```
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent  sibling  child
"c720"              3           8 0x100005100 0x100005700 0x100009080
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent  sibling  child
"bus_adapter"       1           16 0x100005100      0 0x100005780
q4>
```

No More siblings, so thats the end of this bus. There was another bus at this level though as the second ccio had a child pointer, so follow that one to.

```
q4> load struct io_tree_node from 0x100005d80
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent  sibling  child
"GSCToPCI"          2           0 0x100005d00 0x100005f00 0x100005e00
q4>
```

Again there is both a child and sibling pointer, so we'll follow the sibling one (ioscan goes down then accross so we are down across and then down -:)

```
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent  sibling  child
"graph3"            0           8 0x100005d00      0      0
q4>
```

That's it on this bus.

Going back to the first "GSCToPCI". it's child was at 0x100009500

```
q4> load struct io_tree_node from 0x100009500
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
```


Module 10 — The I/O Subsystem

Lab: The I/O Subsystem

```
name_buffer instance hw_addr      parent      sibling      child
""           -1           1 0x100005180 0x100005200 0x100009580
q4>
```

This node has no name, this happens if you look at ioscan out put you will see missing levels. Anyway there is both sibling and child pointers.

```
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
""           -1           19 0x100005180 0x100005500 0x100005280
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
""           -1           20 0x100005180 0x100005600 0x100005520
q4> load struct io_tree_node from sibling
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
"asio0"      1           63 0x100005180      0           0
q4>
```

Here we have the end of another bus, going back to the first nameless node we can follow it's child pointer

```
q4> load struct io_tree_node from 0x100009580
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
""           -1           0 0x100009500      0           0
q4>
```

Not very exciting, there is no name and not more pointers to follow.

```
q4> load struct io_tree_node from 0x100005280
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
"c720"       0           0 0x100005200      0 0x100005300
q4>
```

Another scsi interface, with a child.

```
q4> load struct io_tree_node from 0x100005580
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent      sibling      child
"btlan3"     0           0 0x100005500      0           0
q4>
```

Module 10 — The I/O Subsystem

Lab: The I/O Subsystem

Well that's all the previous layers children, now we can follow one of the scsi "c720" nodes, the first one's child pointer was 0x100009080.

```
q4> load struct io_tree_node from 0x100009080
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent sibling      child
"tgt"          3          7 0x100005620      0 0x100009100
q4>
```

No sibling so let's follow its child

```
q4> load struct io_tree_node from child
loaded 1 struct io_tree_node as an array (stopped by max count)
q4> print -x name_buffer instance %d hw_addr %d parent sibling child
name_buffer instance hw_addr      parent sibling child
"sctl"          2          0 0x100009080      0      0
q4>
```

This method can be followed until there are no more pointers either down or across.

Lab: I/O Structures

1. In this exercise, we will probe the I/O tree. Fill in the table on the last page for each entity in the I/O tree. The intent here is not to complete the table necessarily, but to do enough of it so you see how the information is organized. First lets gather some information on the “root” node structure.

```
q4> load io_node_t from io_tree_root
```

```
q4> name it root_0
```

```
q4> print -x name_buffer parent sibling child type%d hw_addr%d instance%d
```

```
q4> load drv_info_t from drv_info ; ex class using s ; ex name using s
```

From these outputs, fill in the information on the root node in the table.

2. From the root node, let's investigate its children.

```
q4> recall root_0
```

```
q4> load io_node_t from child next sibling max 100
```

```
q4> name it root_0_children
```

```
q4> print -x name_buffer parent sibling child type%d hw_addr%d instance%d  
drv_info
```

Now for each different drv_info, get the corresponding class and driver name.

```
q4> load drv_info_t from <DRV_INFO_ADDR> ; ex class using s ;  
ex name using s
```

From these outputs, fill in the information on the root node's children in the table.

```
q4> recall root_0_children
```

Module 10 — The I/O Subsystem

Lab: I/O Structures

3. Note that some of the children of the root node have children of their own (as indicated by the non-NULL “child” pointer. These in general will be bus converters and bus adapters. Making adjustments for different starting points in the syntax of commands, repeat steps 1 and 2, recursively, for each child found that has children of its own and continue filling in the rest of the table. Again, only continue until you are satisfied you understand the layout of the I/O tree.
4. Extra Credit: Draw the resulting I/O tree from the information in the table.

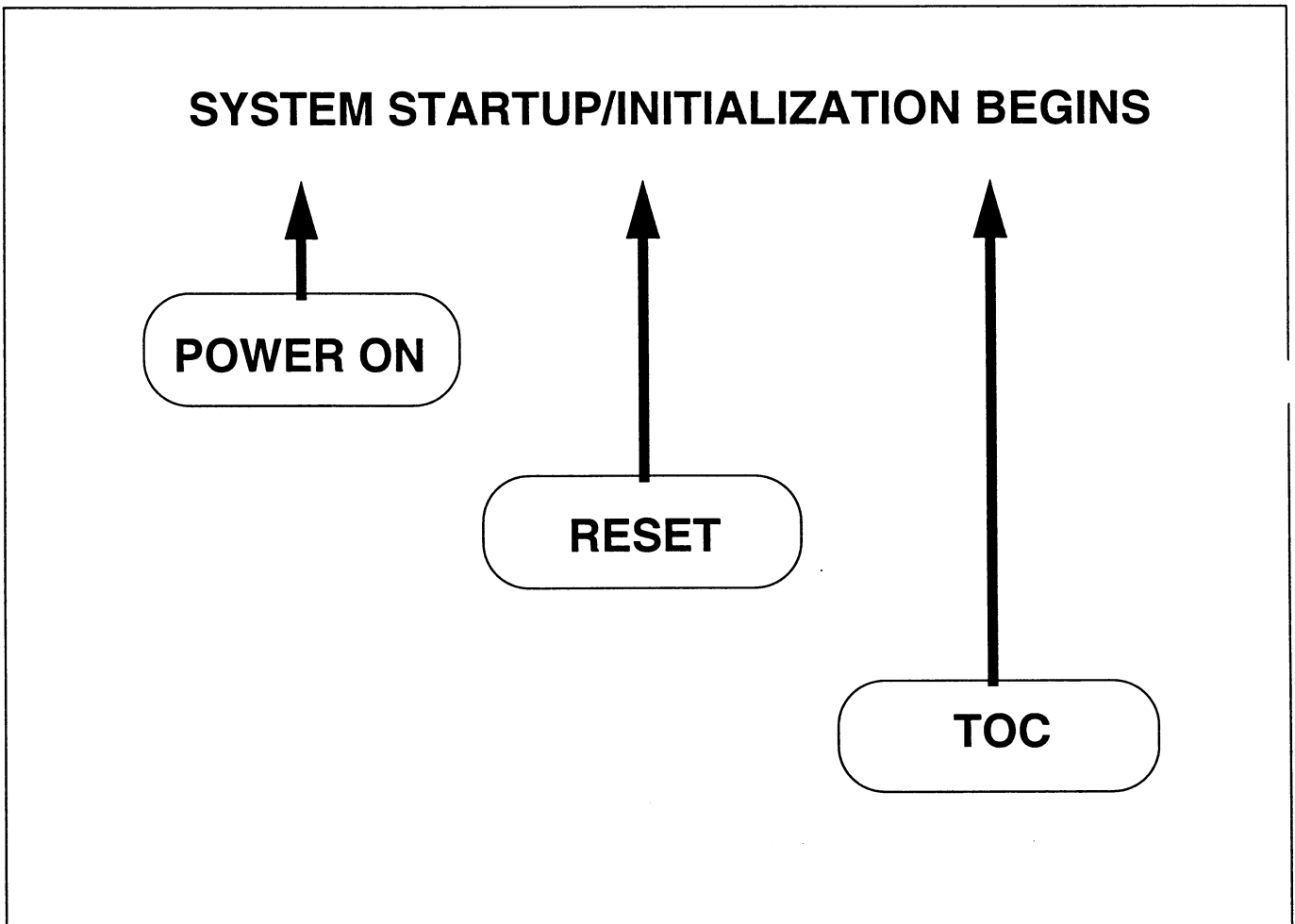
Module 11

System Initialization

Objectives:

- Describe what actions invoke System Initialization
- Describe the firmware components and their role in System Initialization
- Describe the boot process and list the steps involved to boot an HP-UX operating system
- List the steps taken during I/O configuration

Slide: The Beginning of System Initialization



The beginning of System Start-up and Initialization occurs based on one of the following 3 conditions:

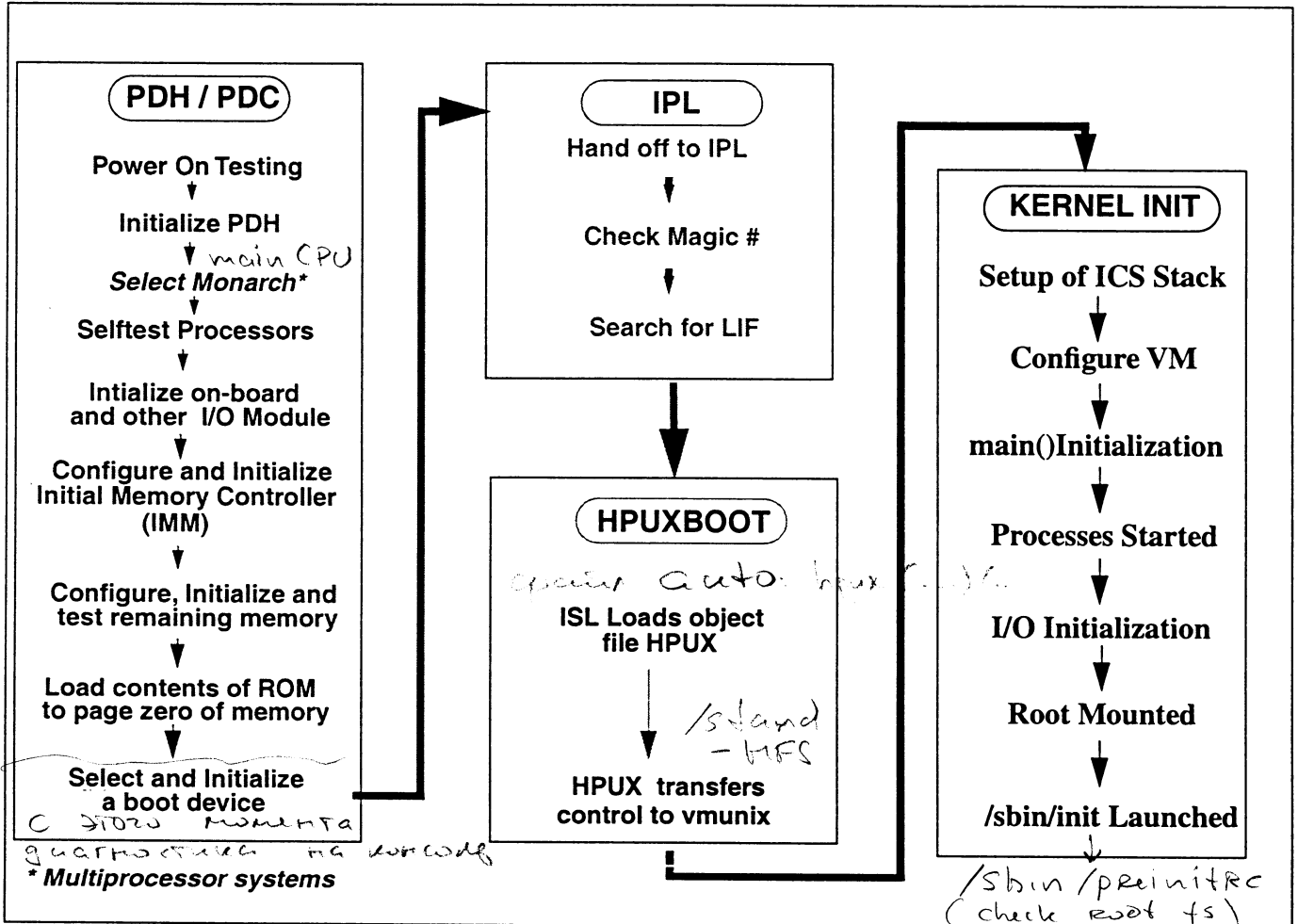
- **POWER ON** - power applied to the system or the result of a powerfail condition.
- **RESET** - Issued via reset switch and causes the broadcast of the software routine "CMD_RESET". This in turn triggers the Processor Dependent Code (PDC) routine PDCE_RESET to reset the system processor. This may occur as a result of a reboot command being issued or a system fault.
- **TRANSFER OF CONTROL (TOC)** - Invoked by a user to clear a hung system and force a bus reset but saves the contents of memory if space is available to write the contents to disk.

The system will travel through several stages before it is actually running a complete HP-UX operating system.

due sq - STO re sq/ba
due sq - no a30bae sq/ba
STUX

Module 11 — System Initialization

Slide: Overview - "The Big Picture"



System Start-up occurs as a result of system level broadcast of the software routine "CMD_RESET". This will alarm PDCE_RESET and start in motion several steps to boot the HP-UX operating system. A brief review of the BIG PICTURE will help us dive deeper into exactly what happens on bootup. Several components make up the start-up process:

PDH/PDC

Processor Dependent Hardware (PDH) contains firmware that allows the system processor(s) to initialize and select the controlling processor, in a multi-processor environment. In single processor systems the selection of the monarch processor is skipped. Once the selection is complete the processor will invoke its **Processor Dependent Code (PDC)** to perform the following:

- Initialize the all other I/O modules.
- Setup and initialize physical memory for Page 0.
- Load the contents from the PDC ROM to Page 0 of physical memory.
- Use the **Boot Console Handler (BCH)** to select and initialize the console and boot device.

Module 11 — System Initialization

IPL (ISL)

Once the boot device has been initialized, PDC procedures load in from disk and handoff to the **Initial Program Loader (IPL)**. The main objective of the IPL is to load the **Initial System Load (ISL)** from disk. This is achieved by determining the boot device magic number and loading ISL into memory.

HPUXBOOT (MONGOOSE)

IPL completes its task, turns over the next steps to the secondary loader or HPUXBOOT. HPUXBOOT setup the environment to allow C routines to execute and loads the vmunix object file.

KERNEL INITIALIZATION

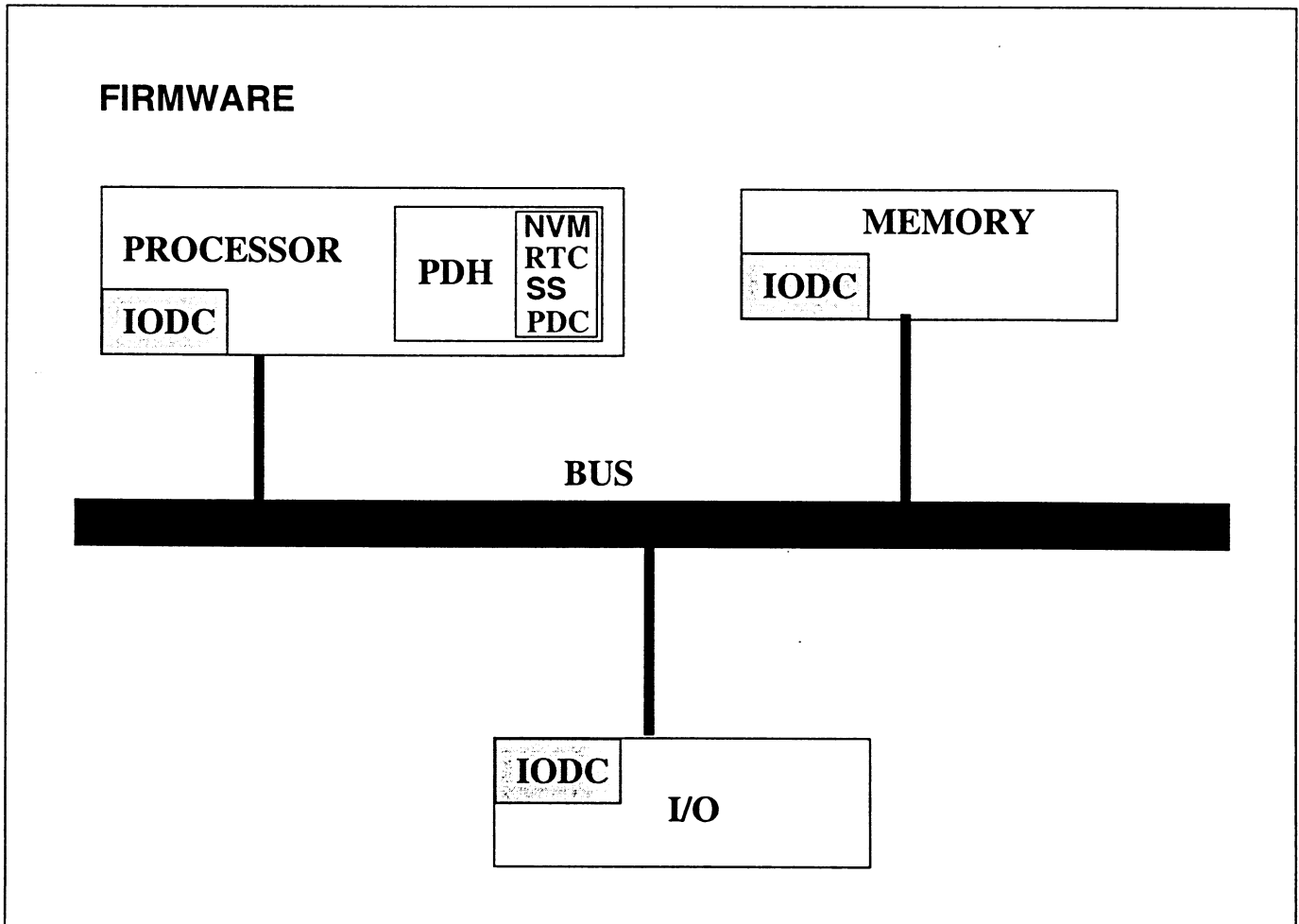
Once vmunix is loaded it will execute and perform the following:

- Initialize and configuration the I/O System
- Set up Virtual Memory
- Set up Kernel Data Structures
- `exec /sbin/init`

The final result will be a login prompt awaiting the user.

Module 11 — System Initialization

Slide: Firmware - PDH/PDC/IODC



A given system will have several hardware modules, such as processor(s), memory and I/O cards. A module may have different firmware components.

Each of these hardware modules is likely to have firmware. Firmware is a term used to denote the computer programs encoded permanently into ROM. On a HP-UX system, firmware is found in two locations:

- Input/Output Dependent Code (IODC)
- Processor Dependent Hardware (PDH)

Input/Output Dependent Code (IODC)

IODC provides drivers for the I/O devices that can act as a console or boot device. Each hardware module contains a separate IODC which helps it to be initialized and identified. This code allows the processor to perform I/O with the module. This establishes a communication path between the processor and the module.

Module 11 — System Initialization

Every hardware module has its own IODC, which provides a unique identification of the card and specifies any card-specific peculiarities. The HP-UX I/O system first acknowledges the IODC when a hardware scan or probe is done (that is, when the system is trying to learn what is at a certain hardware address). The I/O configuration code sends an IO_INQUIRY request (or something similar) to the specific hardware address being probed. HP devices return an *iodc_t* structure read from the IODC of the card.

NOTE: The IODC never communicates with anything else; it merely contains the ROM data associated with the card.

The I/O configuration code reads and interprets that information. For example, on a S800 card, the IODC might contain information on how much SPA space the card might need allocated, so the SIO CDIO would block out that space as it configures that device. Once the I/O subsystem has obtained the ID info from the card's IODC, the CDIO is responsible for finding the driver that knows how to talk to the device with that ID. On a WSIO subsystem, the ID is passed down through a linked list of *driver_attach()* routines until one of the drivers on the list recognizes the ID and "claims" the card. (On the SIO side, this is implemented a little differently, but the notion of having the drivers match only certain card IDs is the same.) Also, HP provides IODC for all their cards, using a consistent format; those cards manufactured by other companies might follow a different format or not even have any.

The actual I/O is carried out by means of the device drivers. In the case of system initialization, the IODC of a disk enables it to be used as a boot device. The IODC of a terminal enables it to be used as a console.

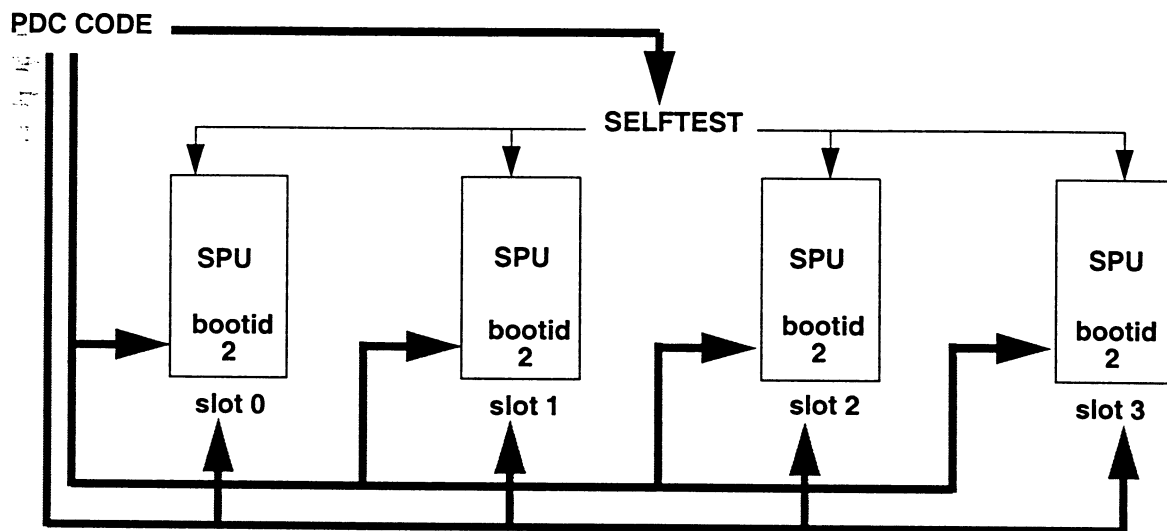
Processor Dependent Hardware (PDH)

PDH basically consist of all the components necessary for a given SPU. The PDH is made up of the following:

- **Processor Dependent Code (PDC)** contains the "bootstrap" code that performs the testing, initialization and booting of the system. The PDC is also responsible for providing routines which will allow memory resident programs to obtain IODC based drivers, determine Hversion dependent hardware parameters, and reconfigure the system hardware. It also contains the HPMC handler routines.
- **RTC (Real Time Clock)**
- **Stable Storage (SS)** contains critical system parameters during power fail and the hardware address of the boot device and console.
- **Non-Volatile Memory (NVM)** stores non-critical system information. NVM usually consists of a table made up modules on the system.

Slide: Firmware - Monarch Selection

MONARCH SELECTION



Start-up - Monarch Selection

The term Monarch basically means “The Controlling Native Processor”. Monarch selection only applies to systems with multiple processors. This step would be skipped in a single processor system and the start-up process would continue.

The purpose of this slide is to show how a module layout may look and how the processors are connected to a central bus.

In a multiple processor system, like the J and K Series HP-PA systems, after the completion of the CPU level selftest we must select a ruler processor (or Monarch). If a processor does not complete selftest it will be removed as a candidate via the PDC routine PDC_CONFIG.

Monarch selection is achieved in two ways:

- PDC_CONFIG has the ability to configure and deconfigure a given processor based on selftest results, etc. If a processor does not complete self-test the PDC routine PDC_CONFIG removes it from consideration.

Module 11 — System Initialization

- A Monarch can also be selected via a user within the Boot Console Handler (BCH) code.

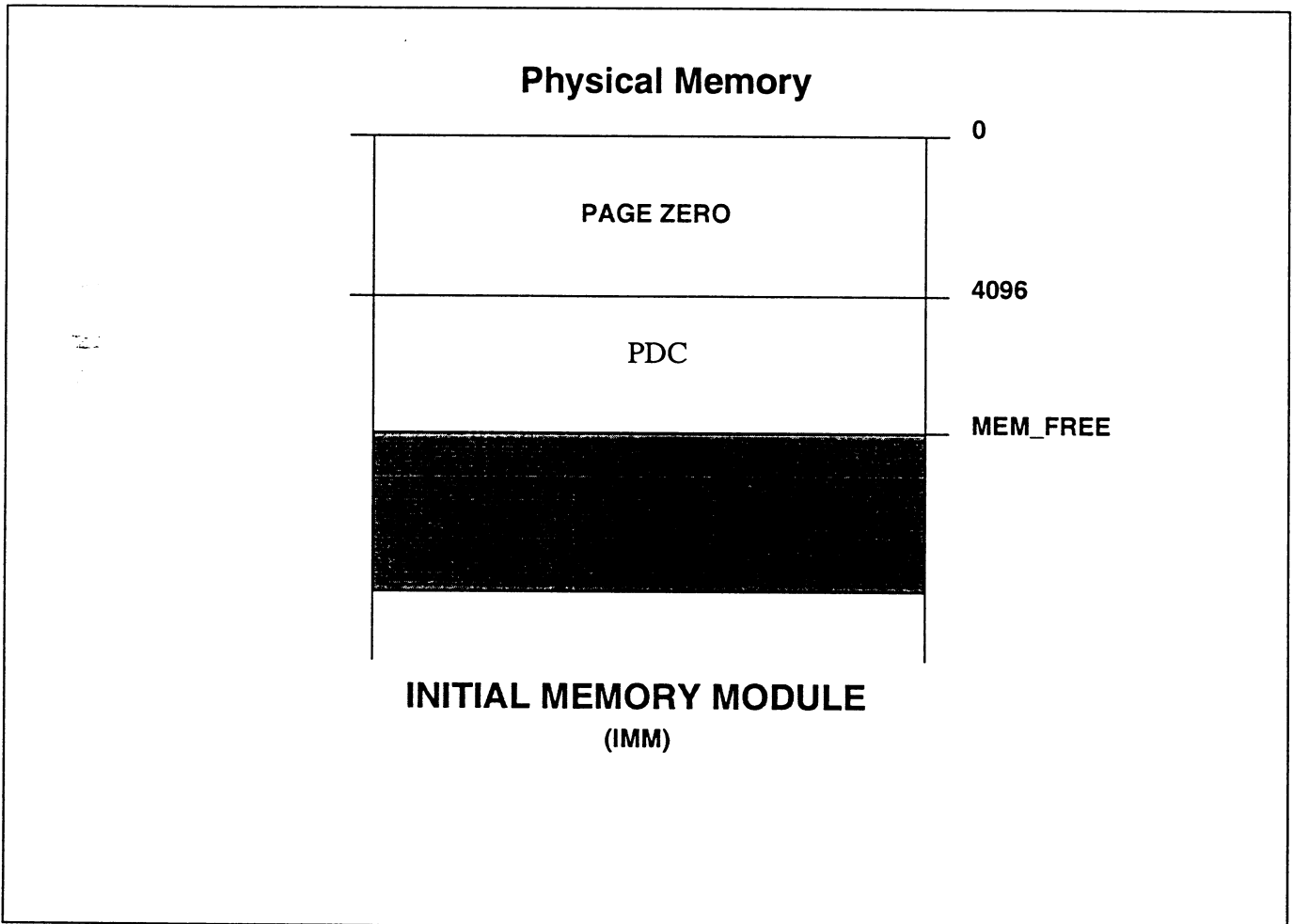
The BCH provides a user interface, via the PDC, to the outside world.

Typically, Monarch selection or arbitration is determined based on the following:

- Writes to the **Local Broadcast Register Set (LBRS) IO_FLEX** register on the central bus to initialize the Hard Physical Address (HPA) of the processors to the central bus physical address space. A write to this register will affect each module on the local bus.
- Processor `BOOT_ID` is set to a default value of 2 by the factory. PDC will attempt to determine the `BOOT_ID` via the `REV_PORT` information scanned into the processor. In the case of the K Class Systems, the `BOOT_ID` will be stored with the EEPROM
- The selection process is simple: who has the highest `BOOT_ID` value. If there is a tie, the processor with the lowest slot number on the `RUNWAY` bus is selected. The winner will be crowned Monarch.
- The Monarch processor will take charge and initialize the local bus by sending a `CMD.RESET.ST` call to each module, thereby excluding the remaining processors.
- The remaining processors (now called slaved processors) will go into the rendezvous code where all interrupts and interruptions are cleared. The slave processors have the responsibility to monitor the actions of the Monarch processor. If the Monarch fails, the slave processors will deconfigure the Monarch and force a system reboot. On boot up the selection process will restart and a new Monarch will be selected.

Module 11 — System Initialization

Slide: PDC - Initial Memory Module



At this point no physical memory has existed for the PDC. The PDC has utilized the area within the EEPROM RAM. The PDC will now allow the CPU to configure and initialize physical memory.

The following steps review the setup of the Initial Memory Module (IMM) and Page Zero.:

- PDC calls "PDC_MEMORY_MAP" to determine the Hard Physical Address (HPA) for all modules via their IODC code on board.
- PDC determines which memory module has the largest array and selects it to be the home for the IMM.
- If all the memory modules sizes are the same then the PDC code will select the memory module in the lowest slot on the bus.
- PDC instructs the IMM to initialize to address 0 in its address space. This allows the creation of Page Zero.

Module 11 — System Initialization

PDC

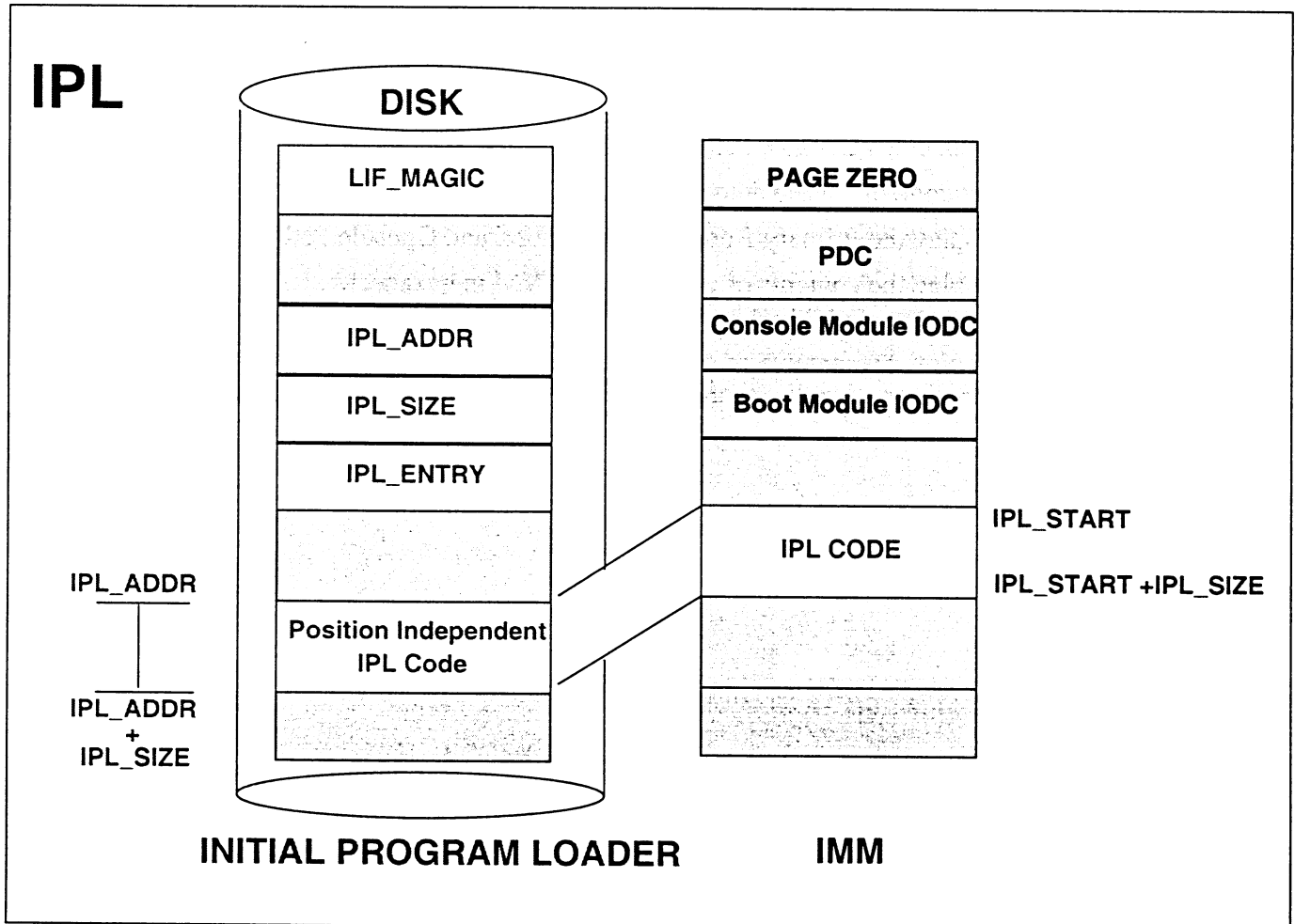
Utilizes the free area below page zero to store instructions and data for PDC procedures.

Page Deallocation Table (PDT)

During memory testing and initialization the Page Deallocation Table is created via a PDC call. This table contains an entry for each page that is scheduled for deallocation based on selftest results.

Module 11 — System Initialization

Slide: IPL - Handoff From PDC



Now we load and handoff to **Initial Program Load (IPL)**. Once the boot device has been selected it will be searched for a **Logical Interchange Format (LIF)** volume. The first code loaded from the boot device will be the IPL code. The boot process creates a data stack in IMM to prepare for IPL program to take over. First the magic number is checked then the IPL code, from disk, is loaded from IPL_START to $IPL_START + IPL_SIZE$ into IMM at starting location IPL_START . It is then executed by passing the $IPL_START + IPL_ENTRY$ locations, **ISL** is invoked.

ISL provides the “first step” for the beginning of the o/s boot process. ISL will now allow a user to interact with the system. This may be achieved if the AUTOBOOT flag has been “disabled” within stable storage or the boot process has been interrupted via the console during the loading of IPL from the disk. The user may provide the PDC a new path location to load IPL from at this point, also.

If this takes place the following commands can be used from ISL:

- help Help - List commands and available utilities
- ls List available utilities
- autoboot Enable or disable the autoboot sequence

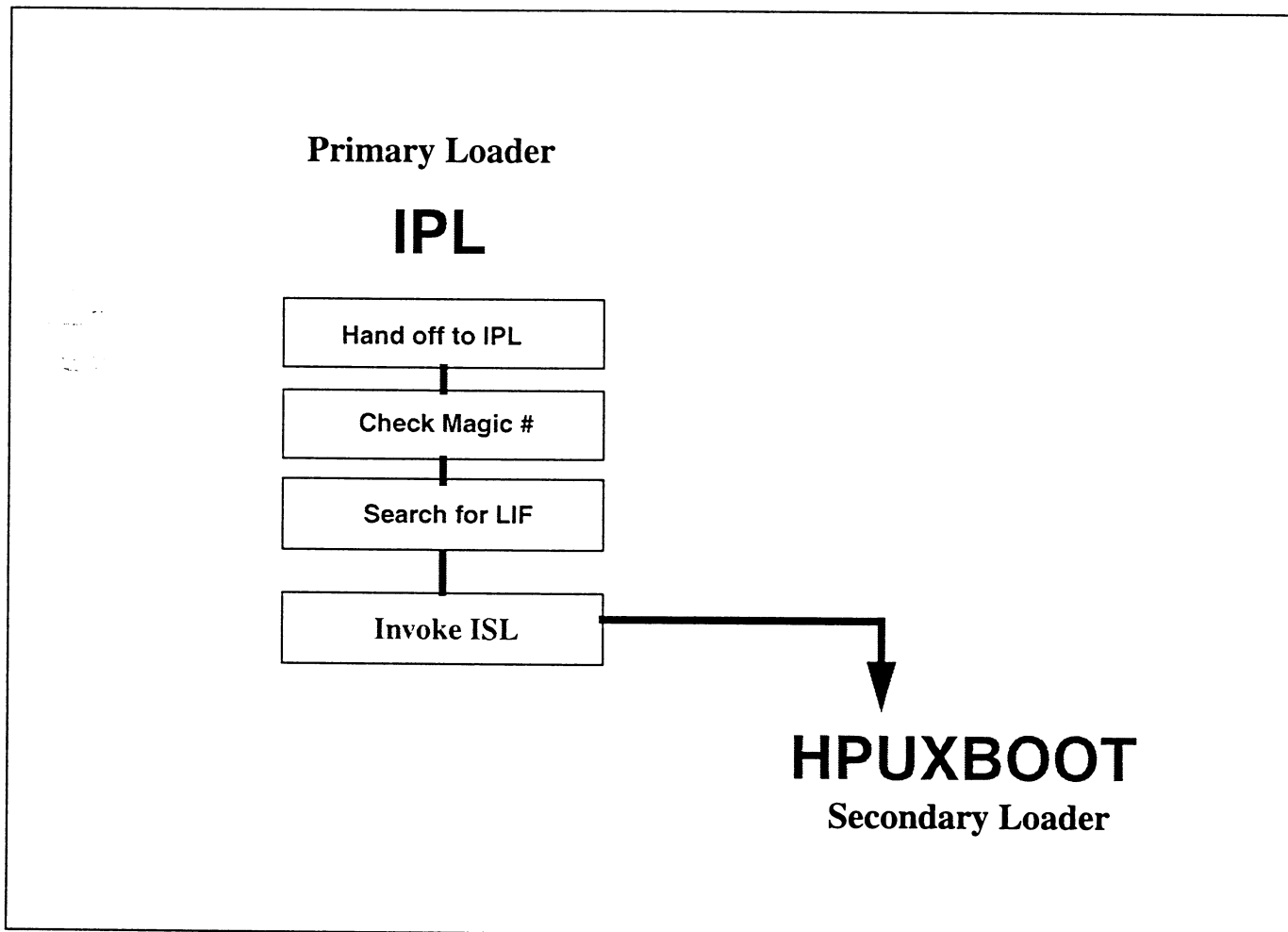
Module 11 — System Initialization

- **primpath** Modify the Primary Boot Path
Parameter - Primary Boot Path in decimal
- **altpath** Modify the Alternate Boot Path
Parameter - Alternate Boot Path in decimal
- **conspath** Modify the Console Path
Parameter - Console Path in decimal
- **listautof** List contents of the autoexecute file
- **display** Display the Primary Boot, Alternate Boot, and Console Paths
- **readnvm** Display the contents of one word of NVM in hexadecimal
Parameter - NVM address in decimal or standard hexadecimal notation
- **readss** Display the contents of one word of Stable Storage in hexadecimal
Parameter - Stable Storage address in decimal or standard hexadecimal notation

selfboot - 8 ONLY

Module 11 — System Initialization

Slide: HPUXBOOT - Handoff From IPL



HPUXBOOT

HPUXBOOT will process a boot request in the following manner:

- `isl` loads `hpuxboot` and branches to it.
- `arg0` (r26) will contain the `pd_c_entry` point address for page 0
- `arg1` (r25) will contain the `isl_command` runstring passed. An example of this would be "vmunix"

The major structures utilized by `hpuxboot`:

`devicefile()` - `hpuxboot.h`

An HPUXBOOT devicefile is equivalent to a file table entry. HPUXBOOT has one of these for each device it needs to access.

hpux - lq - oiaaz of vBopyna.

Module 11 — System Initialization

bootoperation() - *hpuxboot.h*

This structure defines what operation is to be done and what devices are involved with the operation.

```
/*
 * HPUXBOOT command codes
 */

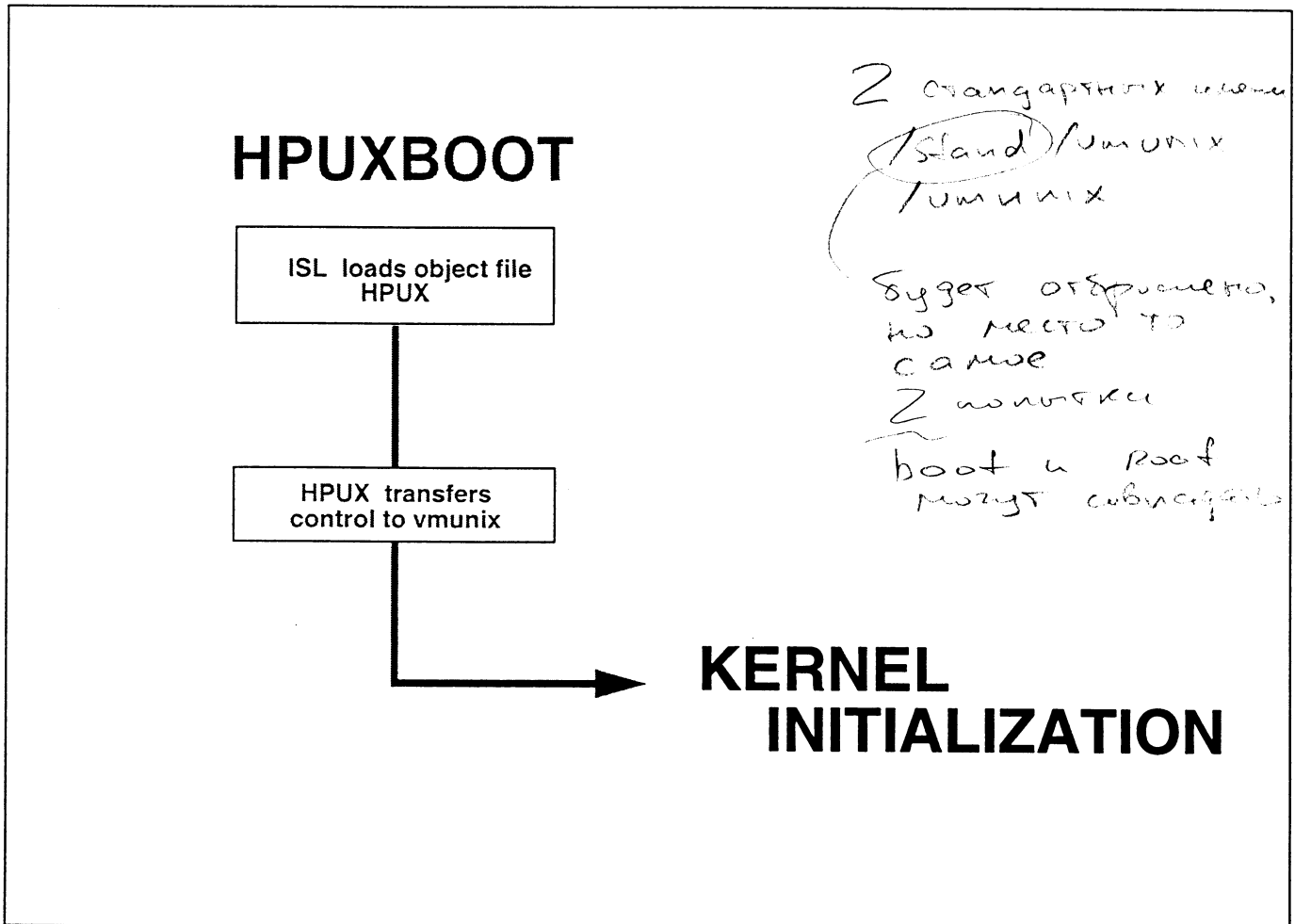
#define BT_COMMAND      0xff
#define BT_NULL        0
#define BT_DEFAULT_BOOT 1
#define BT_BOOT        2
#define BT_COPY        3
#define BT_LS          4
#define BT_INTERACT    5
#define BT_UNUSED1     6
#define BT_SHOW        7
#define SHOW_CONFIG    0x100
#define SHOW_VERSION   0x200
#define SHOW_DEVICES   0x400
#define SHOW_AUTOFILE  0x800
#define BT_SET         8
#define SET_AUTOFILE   0x800
#define BT_LIFLS       9
```

bt_info() - *boot.h*

This structure is completed during the I/O configuration. It provides the address and driver information for the console and boot device. This information is then forwarded to the kernel. *hpuxboot* proceeds next by loading the kernel. Once loaded *hpuxboot* uses procedure calling conventions to branch to the kernel.

Module 11 — System Initialization

Slide: Kernel Initialization



Several steps take place during the initialization of the kernel. Hpxboot has set up the environment for the kernel. Let's look at what happens at this point in time based on the source code that is used and routines called.

Source Code and Routines Used during Kernel Initialization:

rbd_bootstrap()

Is the first code executed when the kernel initializes. This routine contains the IVA vectors, the TLB miss handler code and the initial system bootup. We create a stack frame for calling C, and kernel space is set up. Next we test the arg1 value passed by the loader. This should contain the boot flags which will be used by the kernel. Once complete we jump to *\$locore*.

\$locore

Sets up the **Interrupt Control Stack (ICS)** and checks that the **Processor Status Word (PSW)** has Q-bit set. *\$locore* calls *realmain()* to determine a page number for the next available physical address.

Module 11 — System Initialization

realmain()

The loader (HPUXBOOT) has placed the kernel into physical memory and control was transferred to a low level routine. This setup the stack pointer and global data pointer for *realmain()*. The purpose of this function is to set up the virtual memory system. The following assumptions are made at this point:

- In Real Mode
- Interrupts are off
- Cache is Consistent

The first thing initialized when calling *realmain()* is crash dump code. We must create an environment to handle a failure. The transfer of control address is set to point to the crash dump code. We also determine what the timing variables are and how fast.

Once this is completed it is passed on to *save_boot_args()* routine.

Next we setup and initialize spinlocks in preparation of the first PDC call. The *pdc_entry* point is passed, then we determine the location of the HPA and SPA of the memory modules.

realmain()

realmain will allocate the *pdir* and hash table and reset their values.

kmeminit()

Initializes kernel memory allocator (spinlock)

\$locore

realmain() passes the next available physical page and now we transfer into virtual mode since the *pdir* and hash table is now setup. We call *rfi()* to help transfer us into a virtual state. Once in virtual mode we again flush the entire cache. At this point we are now running on the interrupt stack. We move off the stack in order to initialize the real ICS. We move to the beginning of the kernel stack and load *arg0* with the physical page value provided from *realmain()*. Then we call *\$vstart*.

rfi()

rfi builds a new *psw* from *global_psw*, the *setmask* and the *clear mask* and then *rfi*'s back to the caller (this being *realmain()*). This routine is here since we can not take a *tlb* miss once we turn off the *pc* queue. All code for this routine must reside on the same page.

\$lvstart

This provides the first address passed used when running in virtual mode. *\$lvstart* sets up the stack frame and initializes the *pcb*. Next we determine the address of the *u_area*. then we initialize the *uptr* and setup the pointer to the *UAREA*.

Module 11 — System Initialization

main()

Is the main initialization code for the system. It's functions consist of:

- Clear and free user core
- turn on clock
- Create Process 0
- Call all Initialization Routines
- fork *messengers*
 - process 0 to schedule
 - process 2 to page out
 - process 1 execute bootstrap

Many steps are taken to achieve all of the above.

vfs_mountroot() - *vfs.h*

Mount the root, get the rootdir

xicode_start() - *xicode.c*

The routine is called by */sys/init_main.c* to start up */sbin/init*.

*оган уз станоб - эрнеу BDPA
(u UGRA)*

Slide: I/O Configuration Process - Level 1

FIRST LEVEL I/O CONFIGURATION

- Setup Dynamic Memory
- Load ioconfig file
- Locate Native Modules
- Determine I/O addressing
- Determine page requirements
- Build Native Module I/O Tree

First Level Configuration

The configuration process is broken down into multiple levels. The **First Level** of configuration occurs before the system's Virtual Memory has been initialized. We are still in real mode and early in the system initialization phase.

The following reviews the routines, entry points, and functions called during first level of initialization:

io_mem_zalloc

Dynamically allocates memory. The space is setup after the kernel image is in real memory.

io_real_mode_config()

- Is the call from *realmain()* to initiate the first level of configuration
- The *ioconfig* file is read in by the GIO using system calls provided by the loader (Mongoose / HPUX-BOOT)

Module 11 — System Initialization

io_module_init

- This entry point is called by the GIO or other CDIOs to perform the bulk of I/O configuration task. These consist of initialization of bus converters, adapters, memory modules, scanning lower buses, and mapping I/O pages.
- Locate all fixed address modules and build tables. The table is created by using the **PDC_MEMORY_MAP** mechanism for the Snakes or Gecko systems. **PDC_SYSTEM_MAP** is used for the remaining systems.

niopdir

Used to determine number of I/O entries in PDIR and the final size needed. Once the PA Modules have been claimed the global variable *niopdir* is increased based on the addressing required for the found modules.

At this point the bus adapters can initialize foreign bus via *io_module_init* routines that allow CDIOs to claim these buses. This occurs only if needed. The Central Bus is scanned to locate configurable modules. This is the last bus initialization step since we have already setup the fixed PA and Fixed modules. Similar steps take place in that *niopdir* is increased by the amount of addressing required by the modules. The native module tree is updated with the new entries, and the module type is initialized.

Slide: I/O Configuration Process - Level 2

SECOND LEVEL I/O CONFIGURATION

- I/O System Initialized
- I/O Tree created
- Drivers mapped to hardware
- Devices initialized

Second Level Configuration

Occurs after the Virtual Memory system has been initialized and the system is running in virtual mode. The root, dump, and console devices will be configured in this phase. The algorithms used in the second level is not only used during system initialization but every time new hardware is discovered by *io_scan*.

The following steps occur during the **Second Level I/O Configuration**:

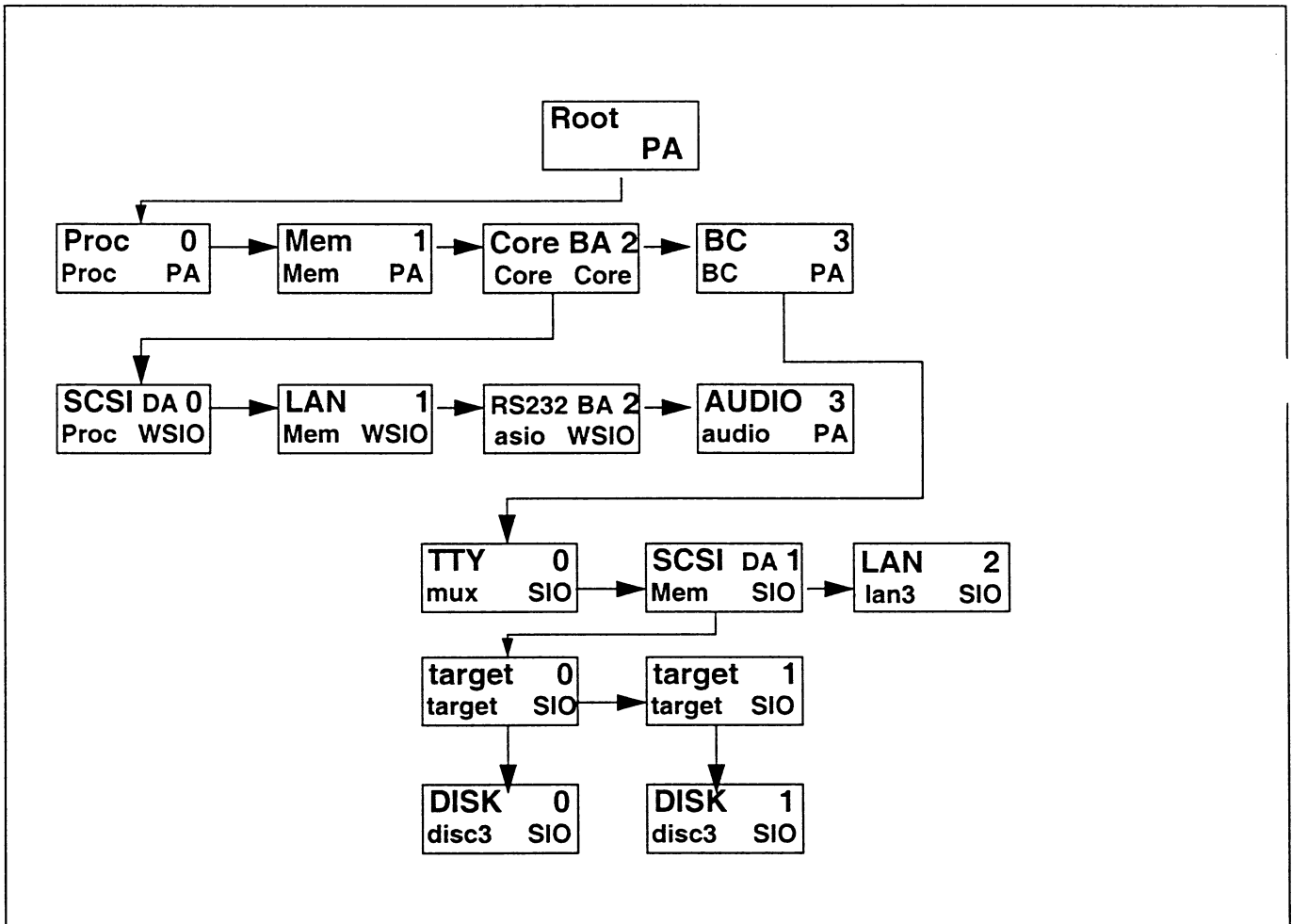
1. All GIO Internal Structures are initialized
2. **init_begin** - Called for each CDIO in the system. This is the first call made to a CDIO after the system goes into virtual mode.
3. **install_drv** - Called to register device drivers with the system.
4. **init_middle** - Is the next call made after the drivers have been installed. Each CDIO is called and initialized.

Module 11 — System Initialization

5. The I/O Tree nodes are created from the */etc/ioconfig*. This only represents consistency of instance numbers across the system and does not associate drivers with nodes at this point.
6. Nodes are scanned to local children nodes. This step only occurs at the interface level and nodes of type **BUS_NEXUS** are scanned. Once the children are located they call *init_func* to bind to drivers and CDIOs.
7. Configure special devices. The root, dump, and console drivers are installed.
8. Kernel Device Table is initialized

Module 11 — System Initialization

Slide: Completed I/O Tree



The slide shows an example of the finished product. The I/O tree consists of central data structure of the I/O system based on individual nodes. The nodes provide the hardware path and important information to allow the GIO to manage their configuration.

Appendix A — Q4 Quick Reference Guide

Viewing data

Variables and Values

To view a variable just type in it's name

```
q4> max_pdir
0350000000 60817408 0x3a00000
```

or for a sum

```
q4> max_pdir-base_pdir
0300000000 50331648 0x3000000
```

Directly

Data can be viewed directly from a location in memory using any format use the **examine** command.

```
q4> examine &msgbuf using DDs
405600 2034 WARNING: Kernel sy...
NOTICE: nfs3_link(): File syst...
```

Q4 expects to follow pointers, msgbuf is not, and so the '&' is needed. **Using** provides the format, two 32bit decimals and a string in this case

As structures

The main way to view data is by loading structures and then printing them.

Loading single structures

```
q4> load struct msginfo from &msginfo
loaded 1 struct msginfo as an array (stopped by max count)
```

Loading tables (arrays of structures)

```
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
```

Loading linked lists

```
q4> load struct kthread from p_firstthreadp next kt_nextp max 100
```

Appendix A Q4 Quick Reference Guide

Viewing, once loaded (print)

Once some data has been loaded it can then be printed. The print command can print whole structures, or specific fields. The fields can be formatted using 'C' style format strings after the field. The default format can be changed using options

print -t prints one field per line
print (without -t) prints fields in columns

```
q4> print -tx | more
indexof 0
mapped 0x1
spaceof 0
addrrof 0x6b0e20
physaddrrof 0x6b0e20
realmode 0
msgmap 0x2a
msgmax 0x2000
msgmnb 0x4000
msgmni 0x32
msgssz 0x8
msgtql 0x28
msgseg 0x800
q4>
```

Extra 'fields' added by q4.

```
q4> print -x p_type p_space p_vaddr p_count %d
p_type p_space p_vaddr p_count
PT_UAREA 0xb7d4000 0x400003fffffce000 7
PT_MMAP 0xc0eb000 0x80b08000 2923
PT_STACK 0xf2f5800 0x7f7e6000 6
PT_DATA 0xf2f5800 0x40001000 2967
PT_TEXT 0xd534400 0x1000 141
0 0x1 0x100948d00 9735424
q4>
```

Q4 1.79a only formats the first line of column output correctly. Subsequent rows are compacted.

What structures are available

The catalog command searches for structures, it uses regular expressions.

```
q4> catalog horse
BYTES FIELDS SHAPE NAME
24 3 struct _horse
24 3 typedef horse_t
2 types matched
q4>
```

Where catalog lists items as 'struct _horse' then 'load struct _horse' will be needed. Where 'typedef horse_t' is used no struct is needed so 'load horse_t' would be used.

What variables are there

The symbols command search the symbol table using regular expressions. The symbol table contains the variable names along with function names...

```
q4> symbols ^cow
      OCTAL      DECIMAL      HEX      DIST TYPE  NAME
011004160      2361456      0x240870      0 FUNC  cowin_error
011004250      2361512      0x2408a8      0 FUNC  cowin
011005200      2361984      0x240a80      0 FUNC  cowcopyto
011005220      2362000      0x240a90      0 FUNC  cowcopyfrom
011005670      2362296      0x240bb8      0 FUNC  cowphysaddr
011005730      2362328      0x240bd8      0 FUNC  cowfreebuf
011006030      2362392      0x240c18      0 FUNC  cowargs_free
011006110      2362440      0x240c48      0 FUNC  cowargs_alloc
032767170      7073400      0x6bee78      8 OBJT  cow_ops
035063160      7628400      0x746670      4 OBJT  cow
10 matches
q4>
```

What fields are there

The fields command is able to display the data structures that q4 knows.

The fields -c option prints the structures as defined in 'C'.

The fields -cv option adds size and offset information as comments.

```
q4> fields -cv struct vfd
struct vfd {
    u_int pg_v:1;          /* off 0 bytes, len 4 bytes */
    u_int pg_cw:1;        /* off 0 bytes, len 1 bit */
    u_int pg_lock:1;      /* off 0 bytes + 1 bit, len 1 bit */
    u_int pg_mlock:1;     /* off 0 bytes + 2 bits, len 1 bit */
    u_int pg_swresv:1;    /* off 0 bytes + 3 bits, len 1 bit */
    u_int pg_pfnum:27;    /* off 0 bytes + 4 bits, len 1 bit */
    u_int pg_swresv:1;    /* off 0 bytes + 4 bits, len 1 bit */
    u_int pg_pfnum:27;    /* off 0 bytes + 5 bits, len 27 bits */
};
q4>
```

More on loading structures

The load command load data structures either signally, from tables or linked lists. Once a set of structures has been loaded, it can be filtered.

Keeping data that matches certain criteria.

The keep command takes a 'C' style logical expression and keeps all the loaded structures that match the condition.

```
q4> load proc_t from proc max nproc
loaded 276 proc_t's as an array (stopped by max count)
q4> keep p_stat && p_pid == 28
kept 1 of 276 proc_t's, discarded 275
q4>
```

If no test is given for a field, it is effectively testing for a non-zero value. To test equality a 'C' style double '=' is needed.

The double '&' is a logical and a single one would be bitwise one.

```
q4> load proc_t from proc max nproc
loaded 276 proc_t's as an array (stopped by max count)
q4> keep p_stat
kept 76 of 276 proc_t's, discarded 200
q4> keep p_flag & SSYS
kept 17 of 76 proc_t's, discarded 59
q4>
```

There is no test for strings so grep and another field that can be used to uniquely identify a structure have to be used.

```
q4> print p_pid p_comm | grep vxfs
28 "vxfsd"
q4> keep p_pid == 28
kept 1 of 17 proc_t's, discarded 16
q4>
```

Discarding data that matches certain criteria

The discard command is the opposite of the keep command.

```
q4> discard p_flag & SSYS
kept 59 of 76 proc_t's, discarded 17
q4>
```

Using fields from loaded structures

When more than one structure is loaded then the fields can not be used by the load and examine commands.

To use individual fields there must be just one of a structure loaded.

```
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
q4> keep p_stat
kept 76 of 276 struct proc's, discarded 200
q4> load struct vas from p_vas
loaded 0 struct vas's as an array (stopped by bad read at 0x0.0x535c0230'e840d0)
q4> keep p_pid == 28
nothing to keep
q4> forget 0      ### discussed later, it gets rid of the 0 structures loaded
                  above
q4> keep p_pid == 28
kept 1 of 76 struct proc's, discarded 75
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
q4>
```

Managing what has been loaded

When data is loaded it is loaded as 'piles' and Q4 maintains a stack of these piles, the commands then refer to the most pile on top of the stack.

History prints the stack

Forget removes piles from the stack

Name it gives piles names by which they can be referred to otherwise numbers are used.

Pushhistory remembers the current position in the stack so that
 Pophistory can throw away every pile back to the previously remembered point (forget only
 deletes one pile at a time).

Recall copies an earlier pile on to the top of the stack, for further use.

```
q4> load struct proc from proc max nproc
loaded 276 struct proc's as an array (stopped by max count)
q4> keep p_stat
kept 76 of 276 struct proc's, discarded 200
q4> name it allprocs
so named
q4> print p_pid p_comm | grep q4
2094 "q4"
q4>
q4> keep p_pid == 2094
kept 1 of 76 struct proc's, discarded 75
q4> name it myproc
so named
q4> pushhistory
q4> load struct vas from p_vas
loaded 1 struct vas as an array (stopped by max count)
```

Appendix A

Q4 Quick Reference Guide

```

q4>
q4> load struct pregion from va_ll.lle_prev next p_ll.lle_prev max 1000
loaded 6 struct pregion's as a linked list (stopped by loop)
q4> history
HIST NAME      LAYOUT COUNT TYPE      COMMENTS
  1 <none>      array  276 struct proc  stopped by max count
  2 allprocs mixed?  76 struct proc  subset of 1
  3 myproc mixed?   1 struct proc  subset of "allprocs"
  4 <none>      array   1 struct vas   stopped by max count
  5 <none>      list    6 struct pregion stopped by loop
q4> keep p_type == PT_UAREA
kept 1 of 6 struct pregion's, discarded 5
q4> history
HIST NAME      LAYOUT COUNT TYPE      COMMENTS
  1 <none>      array  276 struct proc  stopped by max count
  2 allprocs mixed?  76 struct proc  subset of 1
  3 myproc mixed?   1 struct proc  subset of "allprocs"
  4 <none>      array   1 struct vas   stopped by max count
  5 <none>      list    6 struct pregion stopped by loop
  6 <none>      mixed?  1 struct pregion subset of 5
q4>
q4> forget 0
q4>
q4> history
HIST NAME      LAYOUT COUNT TYPE      COMMENTS
  1 <none>      array  276 struct proc  stopped by max count
  2 allprocs mixed?  76 struct proc  subset of 1
  3 myproc mixed?   1 struct proc  subset of "allprocs"
  4 <none>      array   1 struct vas   stopped by max count
  5 <none>      list    6 struct pregion stopped by loop
q4>
q4> pophistory
2 piles popped
q4> history
HIST NAME      LAYOUT COUNT TYPE      COMMENTS
  1 <none>      array  276 struct proc  stopped by max count
  2 allprocs mixed?  76 struct proc  subset of 1
  3 myproc mixed?   1 struct proc  subset of "allprocs"
q4>
q4> recall 2
copied 76 items
q4> recall myproc
copied 1 item
q4> history
HIST NAME      LAYOUT COUNT TYPE      COMMENTS
  1 <none>      array  276 struct proc  stopped by max count
  2 allprocs mixed?  76 struct proc  subset of 1
  3 myproc mixed?   1 struct proc  subset of "allprocs"
  7 <none>      mixed?  76 struct proc  copy of "allprocs"
  8 <none>      mixed?  1 struct proc  copy of "myproc"
q4>

```


Appendix B—Format Log

		logmin 3 logmax 184 logmbno 40 loglbn0 120 logloff 0x200 logmoff 0x0
3	Update inode number 2, it is an immediate directory at this stage and the fsdb command fntlog does not show the immediate data!, but the entry for <u>file.a</u> was added.	00000000: id 3 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901209387 286262 mtime 901209388 310000 ctime 901209388 310000 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 1 iattrino 0 noverlay 0
	Cu_flags ? means 1 user quota check required 4 Fset_dirty	00000140: id 3 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	Remove inode 4 from the free list	000001a0: id 3 func 5 ser 2 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 4
	Set-up inode 4 initially for the new file	000001e0: id 3 func 1 ser 3 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 310001 mtime 901209388 310001 ctime 901209388 310001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 2 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
4	Completion record, note done id = 0, so this record does nothing.	00000320: id 4 func 103 ser 0 lser 0 len 212 Group Done tranid: 0 covering 224 bytes

Appendix B
Format Log

5		<pre>00000400: id 5 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 330000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 3 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00000540: id 5 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
6	NOP	<pre>000005a0: id 6 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes</pre>
7	NOP	<pre>00000600: id 7 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes</pre>
8	Creation of directory entry for file.b, again fsdb does not show the immediate data from the inode	<pre>00000800: id 8 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901209387 286262 mtime 901209388 350008 ctime 901209388 350008 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 4 iattrino 0 noverlay 0</pre>
		<pre>00000940: id 8 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	Remove inode 5 from the free list	<pre>000009a0: id 8 func 5 ser 2 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 5</pre>

	Set-up inode 5	000009e0: id 8 func 1 ser 3 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 5 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 350009 mtime 901209388 350009 ctime 901209388 350009 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 5 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
9	NOP	00000b20: id 9 func 103 ser 0 lser 0 len 212 Group Done tranid: 0 covering 224 bytes
10		00000c00: id 10 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 5 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 360000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 6 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00000d40: id 10 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
11		00000da0: id 11 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes
12		00000e00: id 12 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes

Appendix B
Format Log

Create directory entry for file.c	<pre>00001000: id 13 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901209387 286262 mtime 901209388 380000 ctime 901209388 380000 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 7 iattrino 0 noverlay 0</pre>
	<pre>00001140: id 13 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	<pre>000011a0: id 13 func 5 ser 2 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 6</pre>
	<pre>000011e0: id 13 func 1 ser 3 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 6 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 380001 mtime 901209388 380001 ctime 901209388 380001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 8 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 0 ies: 0</pre>
	<pre>00001320: id 14 func 103 ser 0 lser 0 len 212 Group Done tranid: 0 covering 224 bytes</pre>
	<pre>00001400: id 15 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 6 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 380002 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 9 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<pre>00001540: id 15 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>

		000015a0: id 16 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes
		00001600: id 17 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes
	Add file.d to the directory	00001800: id 18 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901209387 286262 mtime 901209388 400000 ctime 901209388 400000 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 10 iattrino 0 noverlay 0
		00001940: id 18 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
		000019a0: id 18 func 5 ser 2 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 7 000019e0: id 18 func 1 ser 3 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 7 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 400001 mtime 901209388 400001 ctime 901209388 400001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0
		blocks 0 gen 0 version 0 11 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
19		00001b20: id 19 func 103 ser 0 lser 0 len 212 Group Done tranid: 0 covering 224 bytes
20		00001c00: id 20 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 7 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 400002 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 12 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0

Appendix B
Format Log

		00001d40: id 20 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
21		00001da0: id 21 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes
22		00001e00: id 22 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes
23	To add another file to the directory it will need to stop using the immediate format and move to the "normal" directory format using some data extents.	00002000: id 23 func 1 ser 0 lser 5 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 440000 ctime 901209388 440000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 13 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00002140: id 23 func 13 ser 1 lser 5 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
		000021a0: id 23 func 5 ser 2 lser 5 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 aulum dev/bno 0/37 op alloc ino 8
		000021e0: id 23 func 1 ser 3 lser 5 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 8 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 440001 mtime 901209388 440001 ctime 901209388 440001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 14 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00002320: id 23 func 6 ser 4 lser 5 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1285 length 1

	Reorganise the directory to the new format.	00002350: id 23 func 3 ser 5 lser 5 len 1052 Directory entry reorg fset 999 ilist 0 inode 2 bno 1285 oldblen 0 newblen 1024
24		00002790: id 24 func 103 ser 0 lser 0 len 100 Group Done tranid: 0 covering 112 bytes
25		00002940: id 25 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
26		000029a0: id 26 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes
27		00002a00: id 27 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes
28	Update the directory inode as we add the new entry to it.	00002c00: id 28 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 470012 ctime 901209388 470012 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 16 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00002d40: id 28 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	Mark inode 9 in use	00002da0: id 28 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 9
	Set-up inode 9 for new file	00002de0: id 28 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 9 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 470013 mtime 901209388 470013 ctime 901209388 470013 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 17 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0

Appendix B
Format Log

	Add directory entry for new file	00002f20: id 28 func 2 ser 4 lser 4 len 56 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 168 previous d_ino 8 d_reclen 872 d_namlen 6 d_hashnext 0000 added d_ino 9 d_reclen 856 d_namlen 6 d_hashnext 0000 f i l e . f
29		00002f70: id 29 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes
30	Update inode 9	00003000: id 30 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 9 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 480000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 18 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00003140: id 30 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
31		000031a0: id 31 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes
32		00003200: id 32 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes
33		00003400: id 33 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 500000 ctime 901209388 500000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 19 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0

		<pre>00003540: id 33 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
		<pre>000035a0: id 33 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 10</pre>
		<pre>000035e0: id 33 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 10 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 500001 mtime 901209388 500001 ctime 901209388 500001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 20 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00003720: id 33 func 2 ser 4 lser 4 len 56 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 184 previous d_ino 9 d_reclen 856 d_namlen 6 d_hashnext 0000 added d_ino 10 d_reclen 840 d_namlen 6 d_hashnext 0000 file.g</pre>
34		<pre>00003770: id 34 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes</pre>
35		<pre>00003800: id 35 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 10 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 500002 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 21 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00003940: id 35 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
36		<pre>000039a0: id 36 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes</pre>

Appendix B
Format Log

37	<pre>00003a00: id 37 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes</pre>
38	<pre>00003c00: id 38 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 520000 ctime 901209388 520000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 22 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00003d40: id 38 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00003da0: id 38 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 asum dev/bno 0/37 op alloc ino 11 00003de0: id 38 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 11 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 520001 mtime 901209388 520001 ctime 901209388 520001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 23 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00003f20: id 38 func 2 ser 4 lser 4 len 56 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 200 previous d_ino 10 d_reclen 840 d_namlen 6 d_hashnext 0000 added d_ino 11 d_reclen 824 d_namlen 6 d_hashnext 0000 f i l e . h 00003f70: id 39 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes</pre>

40		<pre>00004000: id 40 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1122 ino 11 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 520002 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 24 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00004140: id 40 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
41		<pre>000041a0: id 41 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes</pre>
42		<pre>00004200: id 42 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes</pre>
		<pre>Cut out large number of repeats...</pre>
128		<pre>0000cc00: id 128 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 940000 ctime 901209388 940000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 76 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>0000cd40: id 128 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
		<pre>0000cda0: id 128 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 aumsum dev/bno 0/37 op alloc ino 29</pre>

Appendix B
Format Log

	<pre> 0000cde0: id 128 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 29 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 940001 mtime 901209388 940001 ctime 901209388 940001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 77 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
	<pre> 0000cf20: id 128 func 2 ser 4 lser 4 len 56 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 488 previous d_ino 28 d_reclen 552 d_namlen 6 d_hashnext 0000 added d_ino 29 d_reclen 536 d_namlen 6 d_hashnext 0000 f i l e . z </pre>
129	<pre> 0000cf70: id 129 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes </pre>
130	<pre> 0000d000: id 130 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 29 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 0 mtime 901209388 0 ctime 901209388 940002 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 78 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
	<pre> 0000d140: id 130 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
131	<pre> 0000d1a0: id 131 func 103 ser 0 lser 0 len 84 Group Done tranid: 0 covering 96 bytes </pre>
132	<pre> 0000d200: id 132 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes </pre>

133	Start to create file first, this file is going to be over 30MB	<pre>0000d400: id 133 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209388 960000 ctime 901209388 960000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 79 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>0000d540: id 133 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
		<pre>0000d5a0: id 133 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 30</pre>
		<pre>0000d5e0: id 133 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 960001 mtime 901209388 960001 ctime 901209388 960001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 80 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre> <pre>0000d720: id 133 func 2 ser 4 lser 4 len 55 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 504 previous d_ino 29 d_reclen 536 d_namlen 6 d_hashnext 0000 added d_ino 30 d_reclen 520 d_namlen 5 d_hashnext 01e8 f i r s t</pre>
134	Done creating file	<pre>0000d770: id 134 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes</pre>

Appendix B
Format Log

135	<p>Perform the first extending operation on the inode.</p> <p>Note the extended inode operations flag set to 8</p>	<pre>0000d800: id 135 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 960001 mtime 901209388 960001 ctime 901209388 980000 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8 gen 0 version 0 81 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 8 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<p>Add 8 to the blocks used figure in the CUT.</p>	<pre>0000d940: id 135 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 9 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	<p>Update inode extended operations map</p>	<pre>0000d9a0: id 135 func 5 ser 2 lser 3 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 aulum dev/bno 0/37 op set ino 30</pre>
	<p>Update free map</p>	<pre>0000d9e0: id 135 func 6 ser 3 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1304 length 8</pre>
136	<p>Extend the space allocated to the file. Note the file size has not started to move yet</p>	<pre>0000da20: id 136 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 960001 mtime 901209388 980001 ctime 901209388 980002 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 24 gen 0 version 0 83 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 24 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<p>Add another 16 blocks onto the used figure.</p>	<pre>0000db60: id 136 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 25 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>

	<p>Manipulate the free extent map</p>	<pre>0000dbc0: id 136 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1312 length 16 0000dbf0: id 137 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 960001 mtime 901209388 980004 ctime 901209388 980005 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 88 gen 0 version 0 86 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 88 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<p>Add another 64 blocks to the used counter</p>	<pre>0000dd30: id 137 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 89 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	<p>And take them out of the free map</p>	<pre>0000dd90: id 137 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1328 length 64</pre>
	<p>Give the file another 256 blocks Still in the first extent. And the size is still zero</p>	<pre>0000ddc0: id 138 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209388 960001 mtime 901209388 980013 ctime 901209388 980014 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 344 gen 0 version 0 95 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 344 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<p>Add them to the used counter</p>	<pre>0000df00: id 138 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 345 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	<p>And remove them from the free map</p>	<pre>0000df60: id 138 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1392 length 256</pre>

Appendix B
Format Log

<p>139</p> <p>This time it is 1024 blocks</p> <p>Now the size starts to grow, blocks it's a long way behind.</p>		<pre>0000df90: id 139 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 65536 atime 901209388 960001 mtime 901209388 990005 ctime 901209388 990006 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 1368 gen 0 version 0 128 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 1368 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e0d0: id 139 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 1369 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000e130: id 139 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1648 length 1024</pre>
<p>140</p> <p>Now 2048 blocks</p>		<pre>0000e160: id 140 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 237568 atime 901209388 960001 mtime 901209389 20011 ctime 901209389 20012 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 3416 gen 0 version 0 257 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 3416 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e2a0: id 140 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 3417 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000e300: id 140 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 2672 length 2048</pre>

141	<p>Again 2048 blocks</p> <p>Remember that the free map organises free space using 2048 block pieces</p>	<pre>0000e330: id 141 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 368640 atime 901209388 960001 mtime 901209389 60033 ctime 901209389 60034 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 5464 gen 0 version 0 514 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 5464 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e470: id 141 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 5465 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000e4d0: id 141 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 4720 length 2048</pre>
142	<p>And another 2048 blocks</p>	<pre>0000e500: id 142 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 499712 atime 901209388 960001 mtime 901209389 100045 ctime 901209389 100046 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 7512 gen 0 version 0 771 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 7512 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e640: id 142 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 7513 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000e6a0: id 142 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 6768 length 2048</pre>

Appendix B
Format Log

143	<pre> 0000e6d0: id 143 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 679936 atime 901209388 960001 mtime 901209389 150006 ctime 901209389 150007 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 9560 gen 0 version 0 1028 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 9560 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e810: id 143 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 9561 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000e870: id 143 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 8816 length 2048 </pre>
144	<pre> 0000e8a0: id 144 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 827392 atime 901209388 960001 mtime 901209389 190024 ctime 901209389 190025 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 11608 gen 0 version 0 1285 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 0 des: 11608 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000e9e0: id 144 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 11609 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000ea50: id 144 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 10864 length 2048 </pre>

145		<pre>0000ea80: id 145 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 958464 atime 901209388 960001 mtime 901209389 230051 ctime 901209389 230052 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 13656 gen 0 version 0 1542 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 13656 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>0000ebc0: id 145 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 13657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000ec30: id 145 func 6 ser 2 lser 2 len 36</pre>
		<pre>Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 12912</pre>
146		<pre>0000ec60: id 146 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1155072 atime 901209388 960001 mtime 901209389 280007 ctime 901209389 280008 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 15704 gen 0 version 0 1799 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 15704 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>0000eda0: id 146 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 15705 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
		<pre>0000ee00: id 146 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 14960 length 2048</pre>

Appendix B
Format Log

147	<pre> 0000ee30: id 147 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1286144 atime 901209388 960001 mtime 901209389 320056 ctime 901209389 320057 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 17752 gen 0 version 0 2056 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 17752 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
	<pre> 0000ef70: id 147 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 17753 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
	<pre> 0000efd0: id 147 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 17008 length 2048 </pre>
148	<pre> 0000f000: id 148 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1482752 atime 901209388 960001 mtime 901209389 370030 ctime 901209389 370031 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 19800 gen 0 version 0 2313 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 19800 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
	<pre> 0000f140: id 148 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 19801 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
	<pre> 0000f1a0: id 148 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 19056 length 2048 </pre>

149	<pre> 0000f1d0: id 149 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1613824 atime 901209388 960001 mtime 901209389 420030 ctime 901209389 420031 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 21848 gen 0 version 0 2570 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 21848 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000f310: id 149 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 21849 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000f370: id 149 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 21104 length 2048 </pre>
150	<pre> 0000f3a0: id 150 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1810432 atime 901209388 960001 mtime 901209389 470022 ctime 901209389 470023 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 23896 gen 0 version 0 2827 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 23896 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 0000f4e0: id 150 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 23897 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 0000f540: id 150 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 23152 length 2048 </pre>

Appendix B
Format Log

151		<pre> 0000f570: id 151 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 1941504 atime 901209388 960001 mtime 901209389 520006 ctime 901209389 520007 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 25944 gen 0 version 0 3084 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 25944 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
		<pre> 0000f6b0: id 151 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 25945 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
		<pre> 0000f710: id 151 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 25200 length 2048 </pre>
152		<pre> 0000f740: id 152 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 2138112 atime 901209388 960001 mtime 901209389 570020 ctime 901209389 570021 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 27992 gen 0 version 0 3341 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 27992 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
		<pre> 0000f880: id 152 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 27993 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
		<pre> 0000f8e0: id 152 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 27248 length 2048 </pre>

153		<pre> 0000f910: id 153 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 2203648 atime 901209388 960001 mtime 901209389 610032 ctime 901209389 610033 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 30040 gen 0 version 0 3598 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 30040 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre> <pre> 0000fa50: id 153 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 30041 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre> <pre> 0000fab0: id 153 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 29296 length 2048 </pre>
154	<p>This is finally the last piece to be added to the inode.</p> <p>It is still in the first extent</p> <p>The size was chosen to allow this to happen. The size value still has not caught up.</p>	<pre> 0000fae0: id 154 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 2203648 atime 901209388 960001 mtime 901209389 650015 ctime 901209389 650016 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 31464 gen 0 version 0 3855 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 31464 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre> <pre> 0000fc20: id 154 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31465 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre> <pre> 0000fc80: id 154 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 31344 length 1424 </pre>

Appendix B
Format Log

155	Finally some completion records	0000fcb0: id 155 func 103 ser 0 lser 0 len 324 Group Done tranid: 0 covering 336 bytes
156		0000fe00: id 156 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes
	Add another file, called "second". This will end up 8MB.	00010000: id 157 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209399 670000 ctime 901209399 670000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 4062 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00010140: id 157 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31465 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	Remove inode 31 from the free map (this inode block is now full)	000101a0: id 157 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 aulum dev/bno 0/37 op alloc ino 31
	Set-up inode 31	000101e0: id 157 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 670001 ctime 901209399 670001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 4063 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
	Fill in the directory entry for the new file.	00010320: id 157 func 2 ser 4 lser 4 len 56 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 520 previous d_ino 30 d_reclen 520 d_namlen 5 d_hashnext 01e8 added d_ino 31 d_reclen 504 d_namlen 6 d_hashnext 0128 s e c o n d
158		00010370: id 158 func 103 ser 0 lser 0 len 132 Group Done tranid: 0 covering 144 bytes

159	<p>Start to add space to the new file.</p> <p>Again note the 0 size</p> <p>And</p> <p>The extended inode operations flag</p>	<pre>00010400: id 159 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 670001 ctime 901209399 670002 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8 gen 0 version 0 4064 iattrino 0 noverlay 0 de: 1136 0 0 0 0 0 0 0 0 0 des: 8 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre> <pre>00010540: id 159 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31473 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre> <pre>000105a0: id 159 func 5 ser 2 lser 3 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op set ino 31</pre> <pre>000105e0: id 159 func 6 ser 3 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1136 length 8</pre>
160	<p>Add more space</p>	<pre>00010620: id 160 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 670003 ctime 901209399 670004 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 24 gen 0 version 0 4066 iattrino 0 noverlay 0 de: 1136 0 0 0 0 0 0 0 0 0 des: 24 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre> <pre>00010760: id 160 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31489 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre> <pre>000107c0: id 160 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1144 length 16</pre>

Appendix B
Format Log

161	Add another 64 blocks	<pre> 000107f0: id 161 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 670006 ctime 901209399 670007 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 88 gen 0 version 0 4069 iattrino 0 noverlay 0 de: 1136 0 0 0 0 0 0 0 0 0 des: 88 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00010930: id 161 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31553 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00010990: id 161 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1160 length 64 </pre>
162	<p>And now 56 blocks</p> <p>The first allocation unit is now full. In a 200MB filesystem the last AU will have only 8MB, it must therefore have an expanded freemap.</p>	<pre> 000109c0: id 162 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 680002 ctime 901209399 680003 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 144 gen 0 version 0 4078 iattrino 0 noverlay 0 de: 1136 0 0 0 0 0 0 0 0 0 des: 144 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00010b00: id 162 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 31609 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00010b60: id 162 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1224 length 56 </pre>

163	Move on to using a second extent in the last allocation unit	<pre>00010b90: id 163 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901209399 670001 mtime 901209399 680010 ctime 901209399 750000 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 656 gen 0 version 0 4086 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 512 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00010cd0: id 163 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 32121 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00010d30: id 163 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 196608 length 512</pre>
164	Add 1024 blocks and start to move the size value	<pre>00010d60: id 164 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 212992 atime 901209399 670001 mtime 901209399 760036 ctime 901209399 760037 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 1680 gen 0 version 0 4151 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 1536 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00010ea0: id 164 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 33145 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00010f00: id 164 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 197120 length 1024</pre>

Appendix B
Format Log

165	Add 2028 blocks	<pre> 00010f30: id 165 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 344064 atime 901209399 670001 mtime 901209399 790042 ctime 901209399 790043 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 3728 gen 0 version 0 4280 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 3584 0 0 0 0 0 0 0 0 ie: 0 0 0 0 ies: 0 </pre> <pre> 00011070: id 165 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 35193 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre> <pre> 000110d0: id 165 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 198144 length 2048 </pre>
	And again	<pre> 00011100: id 166 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 475136 atime 901209399 670001 mtime 901209399 830063 ctime 901209399 830064 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 5776 gen 0 version 0 4537 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 5632 0 0 0 0 0 0 0 0 ie: 0 0 0 0 ies: 0 </pre> <pre> 00011240: id 166 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 37241 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre> <pre> 000112a0: id 166 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 200192 length 2048 </pre>

167	And again	<pre> 000112d0: id 167 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 671744 atime 901209399 670001 mtime 901209399 880019 ctime 901209399 880020 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 7824 gen 0 version 0 4794 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 7680 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 00011410: id 167 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39289 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00011470: id 167 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 202240 length 2048 </pre>
168	Add 512 blocks, this overshoots the requested size.	<pre> 000114a0: id 168 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 868352 atime 901209399 670001 mtime 901209399 920038 ctime 901209399 920039 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8336 gen 0 version 0 5051 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 8192 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>

Appendix B
Format Log

	Bring down the size of the second extent so that the file will be the right size	<pre> 000115e0: id 168 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39801 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 00011650: id 168 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 204288 length 512 00011680: id 169 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 868352 atime 901209399 670001 mtime 901209399 930028 ctime 901209399 930028 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8192 gen 0 version 0 5098 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 des: 144 8048 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
169		<pre> 000117c0: id 169 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>
	Return the space back to the free map	<pre> 00011830: id 169 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent free bno 204656 length 144 </pre>
170	Completion records	<pre> 00011860: id 170 func 103 ser 0 lser 0 len 404 Group Done tranid: 0 covering 416 bytes </pre>
171		<pre> 00011a00: id 171 func 103 ser 0 lser 0 len 500 Group Done tranid: 0 covering 512 bytes </pre>
172		<pre> 00011c00: id 172 func 103 ser 0 lser 0 len 500 Group Done tranid: 171 covering 512 bytes </pre>
173		<pre> 00011e00: id 173 func 103 ser 0 lser 0 len 500 Group Done tranid: 171 covering 512 bytes </pre>

174	<p>Now to create a directory called "new". The first block of inodes however is full. Another inode block needs to be added. The number of allocated inodes in the fileset header needs to be increased.</p>	<pre>00012000: id 174 func 14 ser 0 lser 13 len 536 Fileset Header Modification fset 1 ilist 0 ino 3 dev/bno 0/10 New Fileset Header Contents: fsh_fsindex 999 fsh_volname "UNNAMED" fsh_version 2 fsh_checksum 0x35be90d2 fsh_time Thu Jul 23 16:56:52 1998 fsh_ninode 288 fsh_nau 1 fsh_old_ilesize 0 fsh_fsextop 0x0 fsh_dflags 0x1 fsh_quota 0 fsh_maxinode 4294967295 fsh_ilistino[65 97] fsh_iauino 64 fsh_lctino 0 fsh_uquotino 69 fsh_gquotino 0 fsh_attr_ninode 0 fsh_attr_nau 0 fsh_attr_ilistino[67 99] fsh_attr_iauino 66 fsh_attr_lctino 68 fsh_features 0 fsh_fsetid 0x0 00000001 clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000 fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000 fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000 fsh_backup 0x0 00000000 fsh_copy 0x0 00000000 fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000 fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000 fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000 fsh_states 0x0 fsh_status ""</pre>
	<p>? Why twice?</p>	<pre>00012230: id 174 func 14 ser 1 lser 13 len 536 Fileset Header Modification fset 1 ilist 0 ino 35 dev/bno 0/1281 New Fileset Header Contents: fsh_fsindex 999 fsh_volname "UNNAMED" fsh_version 2 fsh_checksum 0x35be90d2 fsh_time Thu Jul 23 16:56:52 1998 fsh_ninode 288 fsh_nau 1 fsh_old_ilesize 0 fsh_fsextop 0x0 fsh_dflags 0x1 fsh_quota 0 fsh_maxinode 4294967295 fsh_ilistino[65 97] fsh_iauino 64 fsh_lctino 0 fsh_uquotino 69 fsh_gquotino 0 fsh_attr_ninode 0 fsh_attr_nau 0 fsh_attr_ilistino[67 99] fsh_attr_iauino 66 fsh_attr_lctino 68 fsh_features 0 fsh_fsetid 0x0 00000001 clone fsh_next 0x0 00000000 fsh_prev 0x0 00000000 fsh_volid 0x0 00000000 fsh_parentid 0x0 00000000 fsh_clonetime 0x0 00000000 fsh_create 0x0 00000000 fsh_backup 0x0 00000000 fsh_copy 0x0 00000000 fsh_backupid 0x0 00000000 fsh_cloneid 0x0 00000000 fsh_llbackid 0x0 00000000 fsh_llfwdid 0x0 00000000 fsh_alloclim 0x0 00000000 fsh_vislim 0x0 00000000 fsh_states 0x0 fsh_status ""</pre>

Appendix B
Format Log

	<p>Add another extent to the inode list file for fileset 999. It is going to have 256 inodes in it</p>	<pre>00012460: id 174 func 1 ser 2 lser 13 len 295 Inode Modification fset 1 ilist 0 dev/bno 0/24 ino 65 osize 295 New Inode Contents: type IFILT mode 4000000777 nlink 1 uid 0 gid 0 size 73728 atime 901209387 286262 mtime 901209387 286262 ctime 901209387 286262 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 97 blocks 72 gen 0 version 0 1 iattrino 0 noverlay 0 de: 1120 204672 0 0 0 0 0 0 0 0 des: 8 64 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
	<p>Remove the blocks from the free map.</p>	<pre>000125a0: id 174 func 13 ser 3 lser 13 len 80 CUT Modification dev/bno 0/15 offset 0 New CUT Contents: cu_fsindex 1 cu_curused 1221 cu_lversion 20002 cu_hversion 0 cu_flags 0 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre> <pre>00012600: id 174 func 6 ser 4 lser 13 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent alloc bno 204672 length 64</pre> <pre>00012630: id 174 func 1 ser 5 lser 13 len 295 Inode Modification fset 1 ilist 0 dev/bno 0/1296 ino 97 osize 295 New Inode Contents: type IFILT mode 4000000777 nlink 1 uid 0 gid 0 size 73728 atime 901209387 286262 mtime 901209387 286262 ctime 901209387 286262 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 999 rdev/reserve/dotdot/matchino 65 blocks 72 gen 0 version 0 2 iattrino 0 noverlay 0 de: 1120 204672 0 0 0 0 0 0 0 0 des: 8 64 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>

	<p>Mark all the new inodes as being free in the IAU</p>	<pre> 00012770: id 174 func 5 ser 6 lser 13 len 288 free inode map changes fset 999 ilst 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 32 op free ino 33 op free ino 34 op free ino 35 op free ino 36 op free ino 37 op free ino 38 op free ino 39 op free ino 40 op free ino 41 op free ino 42 op free ino 43 op free ino 44 op free ino 45 op free ino 46 op free ino 47 op free ino 48 op free ino 49 op free ino 50 op free ino 51 op free ino 52 op free ino 53 op free ino 54 op free ino 55 op free ino 56 op free ino 57 op free ino 58 op free ino 59 op free ino 60 op free ino 61 op free ino 62 op free ino 63 </pre>
--	---	---

Appendix B
Format Log

		000128b0: id 174 func 5 ser 7 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 64 op free ino 65 op free ino 66 op free ino 67 op free ino 68 op free ino 69 op free ino 70 op free ino 71 op free ino 72 op free ino 73 op free ino 74 op free ino 75 op free ino 76 op free ino 77 op free ino 78 op free ino 79 op free ino 80 op free ino 81 op free ino 82 op free ino 83 op free ino 84 op free ino 85 op free ino 86 op free ino 87 op free ino 88 op free ino 89 op free ino 90 op free ino 91 op free ino 92 op free ino 93 op free ino 94 op free ino 95
--	--	---

		<pre>000129e0: id 174 func 5 ser 8 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 96 op free ino 97 op free ino 98 op free ino 99 op free ino 100 op free ino 101 op free ino 102 op free ino 103 op free ino 104 op free ino 105 op free ino 106 op free ino 107 op free ino 108 op free ino 109 op free ino 110 op free ino 111 op free ino 112 op free ino 113 op free ino 114 op free ino 115 op free ino 116 op free ino 117 op free ino 118 op free ino 119 op free ino 120 op free ino 121 op free ino 122 op free ino 123 op free ino 124 op free ino 125 op free ino 126 op free ino 127</pre>
--	--	---

Appendix B
Format Log

		<pre>00012b20: id 174 func 5 ser 9 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 128 op free ino 129 op free ino 130 op free ino 131 op free ino 132 op free ino 133 op free ino 134 op free ino 135 op free ino 136 op free ino 137 op free ino 138 op free ino 139 op free ino 140 op free ino 141 op free ino 142 op free ino 143 op free ino 144 op free ino 145 op free ino 146 op free ino 147 op free ino 148 op free ino 149 op free ino 150 op free ino 151 op free ino 152 op free ino 153 op free ino 154 op free ino 155 op free ino 156 op free ino 157 op free ino 158 op free ino 159</pre>
--	--	--

		<pre>00012c60: id 174 func 5 ser 10 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 160 op free ino 161 op free ino 162 op free ino 163 op free ino 164 op free ino 165 op free ino 166 op free ino 167 op free ino 168 op free ino 169 op free ino 170 op free ino 171 op free ino 172 op free ino 173 op free ino 174 op free ino 175 op free ino 176 op free ino 177 op free ino 178 op free ino 179 op free ino 180 op free ino 181 op free ino 182 op free ino 183 op free ino 184 op free ino 185 op free ino 186 op free ino 187 op free ino 188 op free ino 189 op free ino 190 op free ino 191</pre>
--	--	--

Appendix B
Format Log

		00012d90: id 174 func 5 ser 11 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 192 op free ino 193 op free ino 194 op free ino 195 op free ino 196 op free ino 197 op free ino 198 op free ino 199 op free ino 200 op free ino 201 op free ino 202 op free ino 203 op free ino 204 op free ino 205 op free ino 206 op free ino 207 op free ino 208 op free ino 209 op free ino 210 op free ino 211 op free ino 212 op free ino 213 op free ino 214 op free ino 215 op free ino 216 op free ino 217 op free ino 218 op free ino 219 op free ino 220 op free ino 221 op free ino 222 op free ino 223
--	--	--

		<pre>00012ed0: id 174 func 5 ser 12 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 224 op free ino 225 op free ino 226 op free ino 227 op free ino 228 op free ino 229 op free ino 230 op free ino 231 op free ino 232 op free ino 233 op free ino 234 op free ino 235 op free ino 236 op free ino 237 op free ino 238 op free ino 239 op free ino 240 op free ino 241 op free ino 242 op free ino 243 op free ino 244 op free ino 245 op free ino 246 op free ino 247 op free ino 248 op free ino 249 op free ino 250 op free ino 251 op free ino 252 op free ino 253 op free ino 254 op free ino 255</pre>

Appendix B
Format Log

	As I said there are 256 of the things.	00013000: id 174 func 5 ser 13 lser 13 len 288 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op free ino 256 op free ino 257 op free ino 258 op free ino 259 op free ino 260 op free ino 261 op free ino 262 op free ino 263 op free ino 264 op free ino 265 op free ino 266 op free ino 267 op free ino 268 op free ino 269 op free ino 270 op free ino 271 op free ino 272 op free ino 273 op free ino 274 op free ino 275 op free ino 276 op free ino 277 op free ino 278 op free ino 279 op free ino 280 op free ino 281 op free ino 282 op free ino 283 op free ino 284 op free ino 285 op free ino 286 op free ino 287
--	--	--

175	Now the entry can be added to the root level directory, so firstly work on its inode	<pre>00013130: id 175 func 1 ser 0 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 4 uid 0 gid 0 size 1024 atime 901209387 286262 mtime 901209412 500000 ctime 901209412 500000 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 1 gen 0 version 0 5101 iattrino 0 noverlay 0 de: 1285 0 0 0 0 0 0 0 0 0 des: 1 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00013270: id 175 func 13 ser 1 lser 4 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	Take a new inode from the free map	<pre>000132d0: id 175 func 5 ser 2 lser 4 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 32</pre>
	And set it up. It will use the immediate format	<pre>00013310: id 175 func 1 ser 3 lser 4 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/204672 ino 32 osize 295 New Inode Contents: type IFDIR mode 40777 nlink 2 uid 0 gid 3 size 96 atime 901209412 500001 mtime 901209412 500001 ctime 901209412 500001 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 5102 iattrino 0 noverlay 0</pre>
	Add the directory entry,	<pre>00013450: id 175 func 2 ser 4 lser 4 len 53 directory fset 999 ilist 0 inode 2 bno 1285 blen 1024 boff 536 previous d_ino 31 d_reclen 504 d_namlen 6 d_hashnext 0128 added d_ino 32 d_reclen 488 d_namlen 3 d_hashnext 0000 n e w</pre>
176		<pre>000134a0: id 176 func 103 ser 0 lser 0 len 340 Group Done tranid: 171 covering 352 bytes</pre>
177		<pre>00013600: id 177 func 103 ser 0 lser 0 len 500 Group Done tranid: 171 covering 512 bytes</pre>

Appendix B
Format Log

178	Finally the size value for the 30ish meg file catches up	<pre>00013800: id 178 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 32219136 atime 901209388 960001 mtime 901209389 680066 ctime 901209389 680066 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 31464 gen 0 version 0 5105 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 0 des: 31464 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00013940: id 178 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	And so its extended ops flag can be cleared	<pre>000139a0: id 179 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 30 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 32219136 atime 901209388 960001 mtime 901209389 680066 ctime 901209389 680066 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 31464 gen 0 version 0 5106 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 0 des: 31464 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00013ae0: id 179 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	And from the eiops map in block 39, and the summary in block 37	<pre>00013b40: id 179 func 5 ser 2 lser 2 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op clear ino 30</pre>

	Then the size of the 8M file does the same	<pre>00013b80: id 180 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 8388608 atime 901209399 670001 mtime 901209399 930028 ctime 901209399 930028 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8192 gen 0 version 0 5107 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 8048 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00013cc0: id 180 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
181	Clear the flags	<pre>00013d20: id 181 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1127 ino 31 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 8388608 atime 901209399 670001 mtime 901209399 930028 ctime 901209399 930028 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8192 gen 0 version 0 5108 iattrino 0 noverlay 0 de: 1136 196608 0 0 0 0 0 0 0 0 des: 144 8048 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00013e60: id 181 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39657 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
		<pre>00013ec0: id 181 func 5 ser 2 lser 2 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op clear ino 31</pre>
182		<pre>00013f00: id 182 func 103 ser 0 lser 0 len 244 Group Done tranid: 177 covering 256 bytes</pre>
183		<pre>00014000: id 183 func 103 ser 0 lser 0 len 500 Group Done tranid: 182 covering 512 bytes</pre>
184		<pre>00014200: id 184 func 103 ser 0 lser 0 len 500 Group Done tranid: 182 covering 512 bytes</pre>

Appendix B
Format Log

Appendix C – Raw Notes

```

> 40b                               The intent log starts on block 40
00000028.0000: 00000003

> p 74 xW                             The first entry is 295 bytes, round up and /4
00000028.0000: 00000003 00010003 00000127 000003e7
00000028.0010: 00000000 00000002 00000000 00000000
00000028.0020: 00000460 00000000 00000060 000041ed
00000028.0030: 00000003 00000000 00000000 00000000
00000028.0040: 00000060 35b75d2b 00045e36 35b75d2c
00000028.0050: 0004baf0 35b75d2c 0004baf0 00020000
00000028.0060: 00000000 00000002 00000000 00000000
00000028.0070: 00000000 00000000 00000001 00380000
00000028.0080: 00000003 0014000a 00006c6f 73742b66
00000028.0090: 6f756e64 00000004 00480006 00006669
00000028.00a0: 6c652e61 00000000 00000000 00000000
00000028.00b0: 00000000 00000000 00000000 00000000
00000028.00c0: 00000000 00000000 00000000 00000000
00000028.00d0: 00000000 00000000 00000000 00000000
00000028.00e0: 00000000 00000000 00000000 00000000
00000028.00f0: 00000000 00000000 00000000 00000000
00000028.0100: 00000000 00000000 00000000 00000000
00000028.0110: 00000000 00000000 00000000 00000000
00000028.0120: 00000000 00000000

>

> p L                               Print this location formatted as a log entry
0000a000: id 3  func 1  ser 0  lser 3  len 295
Inode Modification fset 999  ilist 0  dev/bno 0/1120  ino 2  osize 295
New Inode Contents:
type IFDIR mode 4755  nlink 3  uid 0  gid 0  size 96
atime 901209387 286262  mtime 901209388 310000  ctime 901209388 310000
aflags 0  orgtype 2  eopflags 0  eopdata 0
fixextsize/fsindex 0  rdev/reserve/dotdot/matchino 2
blocks 0  gen 0  version 0 1  iattrino 0  noverlay 0

> + 0x2c B                             The log header is 0x2c bytes
00000028.002c: 00

> p I                               print this location as a inode
inode structure at 0x00000028.002c
type IFDIR mode 40755  nlink 3  uid 0  gid 0  size 96
atime 901209387 286262  mtime 901209388 310000  ctime 901209388 310000
aflags 0  orgtype 2  eopflags 0  eopdata 0
fixextsize/fsindex 0  rdev/reserve/dotdot/matchino 2
blocks 0  gen 0  version 0 1  iattrino 0  noverlay 0
> + 0x54 B                             The header of the inode is 0x54 bytes, im doesn't work here
00000028.0080: 00

> p dent                             print as a directory entry, db crashes fsdb
00000028.0080:  d 0      d_ino 3  d_reclen 20  d_namlen 10  d_hashnext 0000
                l o s t + f o u n d

>

> + 20 B                             that entry was 20 bytes log
00000028.0094: 00

> p dent                             print as a directory entry
00000028.0094:  d 0      d_ino 4  d_reclen 72  d_namlen 6  d_hashnext 0000
                f i l e . a

>

```

This entry must be the last as its length is 72, add that to the first length, 20 and we get a length of 92. Since the file is only 96 bytes long there is not enough room for another entry. It is strange that the lengths do not add up to 96, but this is what was found on the disk.

Appendix C
Raw Notes

```
> 40b
00000028.0000: 00000003
> + 0x1a0 B          move to the start of the log entry
00000028.01a0: 00
> p L              print as a log entry
0000a1a0: id 3  func 5  ser 2  lser 3  len 40
free inode map changes  fset 999  ilist 0  aun 0
  map dev/bno 0/38  ausum dev/bno 0/37
    op alloc  ino 4

> p 10 xW          using the length print as Hex
00000028.01a0: 00000003 00050203 00000028 00000002
00000028.01b0: 000003e7 00000000 00000000 00000026
00000028.01c0: 00000000 00000000
> p 20 xW          that does not seem to be enough
00000028.01a0: 00000003 00050203 00000028 00000002
00000028.01b0: 000003e7 00000000 00000000 00000026
00000028.01c0: 00000000 00000000 00000025 00000004
00000028.01d0: 00000002 00010002 00010024 000003e7

00000028.01e0: 00000003 00010303 00010014 000003e7
>
```

directory reorg entry

> 40b ; + 0x2350 B ; p 300X

```

                23   3 5 5 0 2 164
00000030.0350: 00000017 00030505 000200a4 000003e7
00000030.0360: 00000000 00000002 00000000 00000505
00000030.0370: 00000000 00000400 03580020 00000000
00000030.0380: 00000000 00000000 00000000 00000044
00000030.0390: 00000000 00000000 00000058 00680078
00000030.03a0: 00880098 00000000 00000000 00000000
00000030.03b0: 00000000 00000000 00000000 00000003
00000030.03c0: 0014000a 00006c6f 73742b66 6f756e64
00000030.03d0: 00000004 00100006 00006669 6c652e61
00000030.03e0: 00000005 00100006 00006669 6c652e62
00000030.03f0: 00000006 00100006 00006669 6c652e63

```

23 is the log id
3 is the sub function type
5 sub func serial number
of 5 in transaction
0'th part
of 2
part length 164

```

                23   3 5 5 1 2 500
00000031.0000: 00000017 00030505 010201f4 00000007
00000031.0010: 00100006 00006669 6c652e64 00000008
00000031.0020: 03680006 00006669 6c652e65 00000000
00000031.0030: 00000000 00000000 00000000 00000000
00000031.0040: 00000000 00000000 00000000 00000000
00000031.0050: 00000000 00000000 00000000 00000000
00000031.0060: 00000000 00000000 00000000 00000000
00000031.0070: 00000000 00000000 00000000 00000000
00000031.0080: 00000000 00000000 00000000 00000000
00000031.0090: 00000000 00000000 00000000 00000000
00000031.00a0: 00000000 00000000 00000000 00000000
00000031.00b0: 00000000 00000000 00000000 00000000
00000031.00c0: 00000000 00000000 00000000 00000000
00000031.00d0: 00000000 00000000 00000000 00000000
00000031.00e0: 00000000 00000000 00000000 00000000
00000031.00f0: 00000000 00000000 00000000 00000000
00000031.0100: 00000000 00000000 00000000 00000000
00000031.0110: 00000000 00000000 00000000 00000000
00000031.0120: 00000000 00000000 00000000 00000000
00000031.0130: 00000000 00000000 00000000 00000000
00000031.0140: 00000000 00000000 00000000 00000000
00000031.0150: 00000000 00000000 00000000 00000000
00000031.0160: 00000000 00000000 00000000 00000000
00000031.0170: 00000000 00000000 00000000 00000000
00000031.0180: 00000000 00000000 00000000 00000000
00000031.0190: 00000000 00000000 00000000 00000000
00000031.01a0: 00000000 00000000 00000000 00000000
00000031.01b0: 00000000 00000000 00000000 00000000
00000031.01c0: 00000000 00000000 00000000 00000000
00000031.01d0: 00000000 00000000 00000000 00000000
00000031.01e0: 00000000 00000000 00000000 00000000
00000031.01f0: 00000000 00000000 00000000 00000000

```

23 is the log id
3 is the sub function type
5 sub func serial number
of 5 in transaction
1st part
of 2
part length 500

```

                23   3 5 5 2 2 388
00000031.0200: 00000017 00030505 02020184 00000000
00000031.0210: 00000000 00000000 00000000 00000000
00000031.0220: 00000000 00000000 00000000 00000000
00000031.0230: 00000000 00000000 00000000 00000000
00000031.0240: 00000000 00000000 00000000 00000000
00000031.0250: 00000000 00000000 00000000 00000000
00000031.0260: 00000000 00000000 00000000 00000000

```

as above, only this is pt 2 of 2
part length 388

Appendix C
Raw Notes

```
00000031.0270: 00000000 00000000 00000000 00000000
00000031.0280: 00000000 00000000 00000000 00000000
00000031.0290: 00000000 00000000 00000000 00000000
00000031.02a0: 00000000 00000000 00000000 00000000
00000031.02b0: 00000000 00000000 00000000 00000000
00000031.02c0: 00000000 00000000 00000000 00000000
00000031.02d0: 00000000 00000000 00000000 00000000
00000031.02e0: 00000000 00000000 00000000 00000000
00000031.02f0: 00000000 00000000 00000000 00000000
00000031.0300: 00000000 00000000 00000000 00000000
00000031.0310: 00000000 00000000 00000000 00000000
00000031.0320: 00000000 00000000 00000000 00000000
00000031.0330: 00000000 00000000 00000000 00000000
00000031.0340: 00000000 00000000 00000000 00000000
00000031.0350: 00000000 00000000 00000000 00000000
00000031.0360: 00000000 00000000 00000000 00000000
00000031.0370: 00000000 00000000 00000000 00000000
00000031.0380: 00000000 00000000 00000000 00000000
```

```
new entry          24 103 0 0 100
00000031.0390: 00000018 00670000 00000064 00000000
00000031.03a0: 00000000 00000000 00000000 00000000
00000031.03b0: 00000000 00000000 00000000 00000000
00000031.03c0: 00000000 00000000 00000000 00000000
00000031.03d0: 00000000 00000000 00000000 00000000
00000031.03e0: 00000000 00000000 00000000 00000000
00000031.03f0: 00000000 00000000 00000000 00000000
```

> 40b ; + 0x2350 B

```
00000030.0350: 00
```

>

> + 36 B

```
00000030.0374: 00
```

> p 10 xw dir len 1024

```
00000030.0374: 00000400 03580020 00000000 00000000
                    856 32          free bytes and
                    number of hash entries
```

```
00000030.0384: 00000000 00000000 00000044 00000000
```

```
00000030.0394: 00000000 00000058
```

> + 4 B

move passed the dir length into dir itself

```
00000030.0378: 03
```

> p db

print as directory

directory block at 00000030.0378 - total free (d_tfree) 856 nhash 32

```
Hash 0- 7: 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
Hash 8-15: 0000 0044 0000 0000 0000 0000 0000 0000 0058
```

```
Hash 16-23: 0068 0078 0088 0098 0000 0000 0000 0000
```

```
Hash 24-31: 0000 0000 0000 0000 0000 0000 0000 0000
```

```
00000030.03bc: d 0 d_ino 3 d_reclen 20 d_namlen 10 d_hashnext 0000
                l o s t + f o u n d
```

```
00000030.03d0: d 1 d_ino 4 d_reclen 16 d_namlen 6 d_hashnext 0000
```

```
                f i l e . a
```

```
00000030.03e0: d 2 d_ino 5 d_reclen 16 d_namlen 6 d_hashnext 0000
```

```
                f i l e . b
```

```
00000030.03f0: d 3 d_ino 6 d_reclen 16 d_namlen 6 d_hashnext 0000
```

```
                f i l e . c
```

```
00000031.0000: d 4 d_ino 23 d_reclen 3 d_namlen 1285 d_hashnext 0102
```

```
                01ffffffff400
```


Then fsdb crashes when it gets to the end of the part as the print directory block function was not intended to be able to print from the intent log.

The intent log is made out of 512 byte pieces, an entry can not cross the boundary of one of these pieces so it get divided into to parts, as above.

Appendix C
Raw Notes

Appendix D — RM Log

		logmin 3 logmax 85 logmbno 40 loglbn0 76 logloff 0x200 logmoff 0x0
3	Update the parent directories' inode as the file is created.	00000000: id 3 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901548184 581682 mtime 901548255 780000 ctime 901548255 780000 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 1 iattrino 0 noverlay 0
		00000140: id 3 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	Remove the new inode from the free map	000001a0: id 3 func 5 ser 2 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 ausum dev/bno 0/37 op alloc ino 4
	Set-up the new inode for the file	000001e0: id 3 func 1 ser 3 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548255 780001 ctime 901548255 780001 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 0 version 0 2 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
4		00000320: id 4 func 103 ser 0 lser 0 len 212 Group Done tranid: 0 covering 224 bytes
5	The file starts to be allocated disk space. An extend inode operation has been set here to manage the size of the file.	00000400: id 5 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548255 780001 ctime 901548255 840000 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 8 gen 0 version 0 3 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 8 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0

Appendix D
RM Log

		00000540: id 5 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 8 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	So the extended inode operations map needs updating	000005a0: id 5 func 5 ser 2 lser 3 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op set ino 4
	As does the free extent map.	000005e0: id 5 func 6 ser 3 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1304 length 8
6	The file keeps growing, it will reach 100MB	00000620: id 6 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548255 840001 ctime 901548255 840002 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 24 gen 0 version 0 5 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 24 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00000760: id 6 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 24 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
		000007c0: id 6 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1312 length 16
7		000007f0: id 7 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548255 840004 ctime 901548255 840005 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 88 gen 0 version 0 8 iattrino 0 noverlay 0 de: 1304 0 0 0 0 0 0 0 0 0 des: 88 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00000930: id 7 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 88 cu_lversion 20002 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0

		00000990: id 7 func 6 ser 2 lser 2 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent alloc bno 1328 length 64
		The files grows usually 2M at a time
70	The file has now grown to 100MB	00007800: id 70 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 104857600 atime 901548255 780001 mtime 901548267 840035 ctime 901548267 840035 aflags 0 orgtype 1 eopflags 8 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 102400 gen 0 version 0 12862 iattrino 0 nooverlay 0 de: 1304 196608 32768 65536 0 0 0 0 0 0 des: 31464 8192 32768 29976 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00007940: id 70 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 102400 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
71	Since the size and the number of allocated blocks agree the extended inodes operations flag is removed	000079a0: id 71 func 1 ser 0 lser 2 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 1 uid 0 gid 3 size 104857600 atime 901548255 780001 mtime 901548267 840035 ctime 901548267 840035 aflags 0 orgtype 1 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 102400 gen 0 version 0 12863 iattrino 0 nooverlay 0 de: 1304 196608 32768 65536 0 0 0 0 0 0 des: 31464 8192 32768 29976 0 0 0 0 0 0 ie: 0 0 0 ies: 0
		00007ae0: id 71 func 13 ser 1 lser 2 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 102400 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	And the map is updated	00007b40: id 71 func 5 ser 2 lser 2 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op clear ino 4

Appendix D
RM Log

72	Most of this is now safely on disk.	00007b80: id 72 func 103 ser 0 lser 0 len 116 Group Done tranid: 67 covering 128 bytes
73	Now the file is being removed. The filename is deleted so inode 2 is updated. Note it is using the immediate format.	00007c00: id 73 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1120 ino 2 osize 295 New Inode Contents: type IFDIR mode 40755 nlink 3 uid 0 gid 0 size 96 atime 901548184 581682 mtime 901548339 890000 ctime 901548339 890000 aflags 0 orgtype 2 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 2 blocks 0 gen 0 version 0 12864 iattrino 0 noverlay 0
		00007d40: id 73 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 102400 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	Then the file's own inode is updated. Setting the link count to 0 and the extended operations flag to 1.	00007da0: id 73 func 1 ser 2 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 0 uid 0 gid 3 size 104857600 atime 901548255 780001 mtime 901548267 840035 ctime 901548267 840035 aflags 0 orgtype 1 eopflags 1 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 102400 gen 0 version 0 12865 iattrino 0 noverlay 0 de: 1304 196608 32768 65536 0 0 0 0 0 0 des: 31464 8192 32768 29976 0 0 0 0 0 0 ie: 0 0 0 ies: 0
	As there is once again an extended operation the IAU needs updating	00007ee0: id 73 func 5 ser 3 lser 3 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 ausum dev/bno 0/37 op set ino 4
74		00007f20: id 74 func 103 ser 0 lser 0 len 212 Group Done tranid: 67 covering 224 bytes
75	These changes get out to the disk.	00008000: id 75 func 103 ser 0 lser 0 len 500 Group Done tranid: 74 covering 512 bytes
76	Move upto the next block boundary	00008200: id 76 func 103 ser 0 lser 0 len 500 Group Done tranid: 74 covering 512 bytes

77	At this time the program is killed and the kernel closes the file. This causes the real deletion.	<pre>00008400: id 77 func 1 ser 0 lser 1 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 0 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548267 840035 ctime 901548863 420000 aflags 0 orgtype 1 eopflags 1 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 102400 gen 1 version 0 12868 iattrino 0 noverlay 0 de: 1304 196608 32768 65536 0 0 0 0 0 0 des: 31464 8192 32768 29976 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>00008540: id 77 func 13 ser 1 lser 1 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 102400 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
78	The space is returned to the free maps	<pre>000085a0: id 78 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 0 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548267 840035 ctime 901548863 430000 aflags 0 orgtype 1 eopflags 1 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 39656 gen 1 version 0 12869 iattrino 0 noverlay 0 de: 1304 196608 0 0 0 0 0 0 0 0 des: 31464 8192 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0</pre>
		<pre>000086e0: id 78 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 39656 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0</pre>
	The extent map for AU 3 is updated	<pre>00008740: id 78 func 6 ser 2 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1080 extent free bno 65536 length 29976</pre>
	AU 2 goes back to being unexpanded as it all got freed in l operation	<pre>00008770: id 78 func 6 ser 3 lser 3 len 36 Map Type: SMAP Summary dev/blk: 0/0 Map dev/blk: 0/32 extent free bno 32768 length 32768</pre>
79		<pre>000087a0: id 79 func 103 ser 0 lser 0 len 84 Group Done tranid: 74 covering 96 bytes</pre>

Appendix D
RM Log

80	All the space has now been freed	<pre> 00008800: id 80 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type IFREG mode 100666 nlink 0 uid 0 gid 3 size 0 atime 901548255 780001 mtime 901548267 840035 ctime 901548863 500000 aflags 0 orgtype 1 eopflags 1 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 1 version 0 12870 iattrino 0 noverlay 0 de: 0 0 0 0 0 0 0 0 0 0 des: 0 0 0 0 0 0 0 0 0 0 ie: 0 0 0 ies: 0 </pre>
		<pre> 00008940: id 80 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0 </pre>

	The free maps for AU 1 and	000089a0: id 80 func 6 ser 2 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1112 extent free bno 196608 length 8192
	7 are updated to reflect this.	000089d0: id 80 func 6 ser 3 lser 3 len 36 Map Type: EMAP Summary dev/blk: 0/35 Map dev/blk: 0/1064 extent free bno 1304 length 31464
81	The inode can now be freed	00008a00: id 81 func 1 ser 0 lser 3 len 295 Inode Modification fset 999 ilist 0 dev/bno 0/1121 ino 4 osize 295 New Inode Contents: type FREE mode 0 nlink 0 uid 0 gid 0 size 0 atime 0 0 mtime 0 0 ctime 0 0 aflags 0 orgtype 0 eopflags 0 eopdata 0 fixextsize/fsindex 0 rdev/reserve/dotdot/matchino 0 blocks 0 gen 2 version 0 12871 iattrino 0 noverlay 0
		00008b40: id 81 func 13 ser 1 lser 3 len 80 CUT Modification dev/bno 0/15 offset 64 New CUT Contents: cu_fsindex 999 cu_curused 0 cu_lversion 26190 cu_hversion 0 cu_flags 5 cu_dcurused 0 cu_visused 0 cu_update 0 0
	This gets rid of the extended operation, and so it is removed from the map.	00008ba0: id 81 func 5 ser 2 lser 3 len 40 inode extop map changes fset 999 ilist 0 aun 0 map dev/bno 0/39 aulum dev/bno 0/37 op clear ino 4
	And the inode map	00008be0: id 81 func 5 ser 3 lser 3 len 40 free inode map changes fset 999 ilist 0 aun 0 map dev/bno 0/38 aulum dev/bno 0/37 op free ino 4
82		00008c20: id 82 func 103 ser 0 lser 0 len 468 Group Done tranid: 74 covering 480 bytes
83		00008e00: id 83 func 103 ser 0 lser 0 len 500 Group Done tranid: 74 covering 512 bytes
84	These operations have now been written to disk.	00009000: id 84 func 103 ser 0 lser 0 len 500 Group Done tranid: 83 covering 512 bytes
85		00009200: id 85 func 103 ser 0 lser 0 len 500 Group Done tranid: 83 covering 512 bytes

Appendix D
RM Log