3

The Modular Debugger

With contributions from Mike Shapiro, Eric Schrock and Matt Simmons

This chapter broadly explains how to use the Modular Debugger, MDB, to debug systems and applications. It leads to the full reference for MDB, which is available in the Solaris *Modular Debugger Guide*. We have structured this chapter in the following main sections:

- "Introduction to the Modular Debugger" on page 123 — background and theory of MDB.
- "Getting Started with MDB" on page 129 — a hands-on tour of MDB features.
- "Debugging Kernel Cores" on page 154 — a hands-on guide to kernel-related MDB features. Start here if you want skip the longer tutorial on MDB, and jump right into MDB's core commands.
- "`kmdb`, the Kernel Modular Debugger" on page 178 — Debugging a live kernel with `kmdb`.
- "`kmdb` Implementation" on page 185 — The architecture and implementation of the kernel MDB.
- References: *GDB-to-MDB Migration*; *MDB Command Reference*.

3.1 Introduction to the Modular Debugger

If you were a detective investigating the scene of a crime, you might interview witnesses and ask them to describe what happened and who they saw. However, if there were no witnesses or these descriptions proved insufficient, you might consider collecting fingerprints and forensic evidence that could be examined for DNA to help solve the case. Often, software program failures divide into analogous categories: problems that can be solved with source-level debugging tools; and problems that require low-level debugging facilities, examination of core files, and knowledge of assembly language to diagnose and correct. MDB facilitates analysis of this second class of problems.

It might not be necessary to use MDB in every case, just as a detective doesn't need a microscope and DNA evidence to solve every crime. However, when programming a complex low-level software system such as an operating system, you might frequently encounter these situations. That's why MDB is designed as a debugging framework that lets you construct your own custom analysis tools to aid in the diagnosis of these problems. MDB also provides a powerful set of built-in commands with which you can analyze the state of your program at the assembly language level.

3.1.1 MDB

MDB provides a completely customizable environment for debugging programs, including a dynamic module facility that programmers can use to implement their own debugging commands to perform program-specific analysis. Each MDB module can examine the program in several different contexts, including live and postmortem. The Solaris Operating System includes a set of MDB modules that assist programmers in debugging the Solaris kernel and related device drivers and kernel modules. Third-party developers might find it useful to develop and deliver their own debugging modules for supervisor or user software.

3.1.2 MDB Features

MDB offers an extensive collection of features for analyzing the Solaris kernel and other target programs. Here's what you can do:

- Perform postmortem analysis of Solaris kernel crash dumps and user process core dumps.

  MDB includes a collection of debugger modules that facilitate sophisticated analysis of kernel and process state, in addition to standard data display and formatting capabilities. The debugger modules allow you to formulate complex queries to do the following:

  – Locate all the memory allocated by a particular thread

  – Print a visual picture of a kernel STREAM

- Determine what type of structure a particular address refers to
- Locate leaked memory blocks in the kernel
- Analyze memory to locate stack traces

- Use a first-class programming API to implement your own debugger commands and analysis tools without having to recompile or modify the debugger itself.

  In MDB, debugging support is implemented as a set of loadable modules (shared libraries on which the debugger can run `dlopen`(3C)), each of which provides a set of commands that extends the capabilities of the debugger itself. The debugger in turn provides an API of core services, such as the ability to read and write memory and access symbol table information. MDB provides a framework for developers to implement debugging support for their own drivers and modules; these modules can then be made available for everyone to use.
- Learn to use MDB if you are already familiar with the legacy debugging tools `adb` and `crash`.

  MDB is backward compatible with these existing debugging solutions. The MDB language itself is designed as a superset of the adb language; all existing adb macros and commands work within MDB, so developers who use adb can immediately use MDB without knowing any MDB-specific commands. MDB also provides commands that surpass the functionality available from the crash utility.
- Benefit from enhanced usability features.

  MDB provides a host of usability features:
  - Command-line editing
  - Command history
  - Built-in output pager
  - Syntax error checking and handling
  - Online help
  - Interactive session logging

3.1.3 MDB Features

The MDB infrastructure was first added in Solaris 8. Many new features have been added throughout Solaris releases, as shown in Table 0.1.

Table 0.1 MDB History

Solaris Revision	Annotation
Solaris 8	MDB introduced
Solaris 9	Kernel type information (e.g., `::print`)
Solaris 10	User-level type information (Common Type Format)
Solaris Next	kmdb replaces kadb

3.1.4 Terms

Throughout this chapter, MDB is used to describe the common debugger core—the set of functionality common to both `mdb` and `kmdb`. `mdb` refers to the userland debugger. `kmdb` refers to the in-situ kernel debugger.

3.2 MDB Debugger Concepts

This section discusses the significant aspects of MDB's design and the benefits derived from this architecture.

3.2.1 Building Blocks

The target is the program being inspected by the debugger. MDB currently provides support for the following types of targets:

- User processes
- User process core files
- Live operating system without kernel execution control (through `/dev/kmem` and `/dev/ksyms`)
- Live operating system with kernel execution control (through `kmdb`(1))
- Operating system crash dumps
- User process images recorded inside an operating system crash dump
- ELF object files
- Raw data files

Each target exports a standard set of properties, including one or more address spaces, one or more symbol tables, a set of load objects, and a set of threads. Figure 3.1 shows an overview of the MDB architecture, including two of the built-in targets and a pair of sample modules.

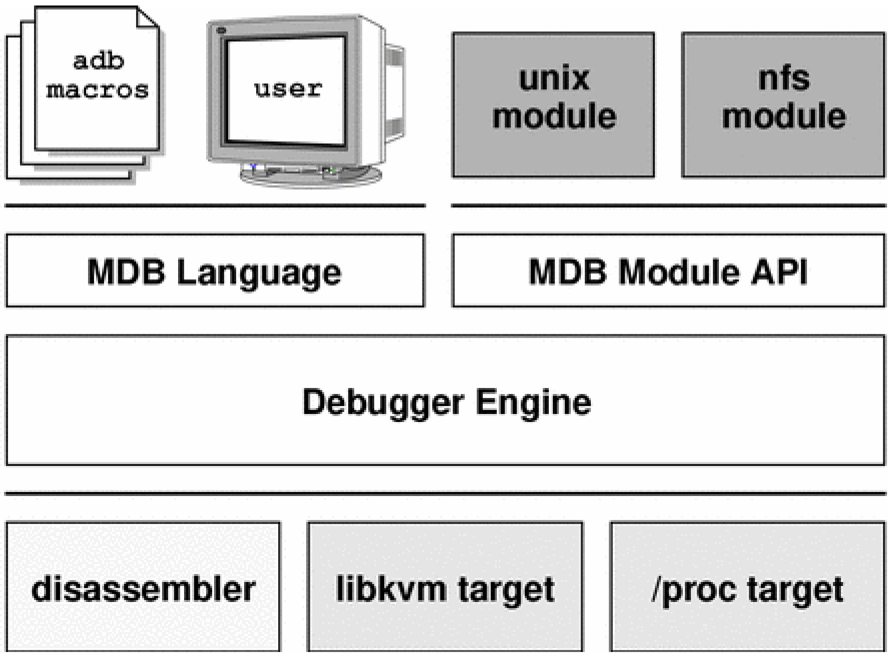


Figure 3.1 MDB Architecture

A debugger command, or *dcmd* (pronounced dee-command) in MDB terminology, is a routine in the debugger that can access any of the properties of the current target. MDB parses commands from standard input, then executes the corresponding dcmds. Each dcmd can also accept a list of string or numerical arguments, as shown in "MDB Command Syntax" on page 131. MDB contains a set of built-in dcmds described in Section 3.3.8 , "dcmds," that are always available. The programmer can also extend the capabilities of MDB itself by writing dcmds, using a programming API provided with MDB.

A *walker* is a set of routines that describe how to walk, or iterate, through the elements of a particular program data structure. A walker encapsulates the data structure's implementation from dcmds and from MDB itself. You can use walkers interactively or as a primitive to build other dcmds or walkers. As with dcmds, you can extend MDB by implementing additional walkers as part of a debugger module.

A debugger module, or *dmod* (pronounced dee-mod), is a dynamically loaded library containing a set of dcmds and walkers. During initialization, MDB attempts to load dmods corresponding to the load objects present in the target. You can subsequently load or unload dmods at any time while running MDB. MDB provides a set of standard dmods for debugging the Solaris kernel.

A macro file is a text file containing a set of commands to execute. Macro files typically automate the process of displaying a simple data structure. MDB provides complete backward compatibility for the execution of macro files written for `adb`. The set of macro files provided with the Solaris installation can therefore be used with either tool.

3.2.2 Modularity

The benefit of MDB's modular architecture extends beyond the ability to load a module containing additional debugger commands. The MDB architecture defines clear interface boundaries between each of the layers shown in Figure 3.1. Macro files execute commands written in the MDB or adb language. Dcmds and walkers in debugger modules are written with the MDB Module API, and this forms the basis of an application binary interface that allows the debugger and its modules to evolve independently.

The MDB namespace of walkers and dcmds also defines a second set of layers between debugging code that maximizes code sharing and limits the amount of code that must be modified as the target program itself evolves. For example, one of the primary data structures in the Solaris kernel is the list of `proc_t` structures representing active processes in the system. The `::ps` dcmd must iterate over this list to produce its output. However, the code to iterate over the list is not in the `::ps` dcmd but is encapsulated in the genunix module's proc walker.

MDB provides both `::ps` and `::ptree` dcmds, but neither has any knowledge of how `proc_t` structures are accessed in the kernel. Instead, they invoke the proc walker programmatically and format the set of returned structures appropriately. If the data structure used for `proc_t` structures ever changed, MDB could provide a new proc walker and none of the dependent dcmds would need to change. You can also access the proc walker interactively with the `::walk` dcmd to create novel commands as you work during a debugging session.

In addition to facilitating layering and code sharing, the MDB Module API provides dcmds and walkers with a single stable interface for accessing various properties of the underlying target. The same API functions access information from user process or kernel targets, simplifying the task of developing new debugging facilities.

In addition, a custom MDB module can perform debugging tasks in a variety of contexts. For example, you might want to develop an MDB module for a user program you are developing. Once you have done so, you can use this module when MDB examines a live process executing your program, a core dump of your program, or even a kernel crash dump taken on a system on which your program was executing.

The Module API provides facilities for accessing the following target properties:

- Address Spaces — The module API provides facilities for reading and writing data from the target's virtual address space. Functions for reading and writing using physical addresses are also provided for kernel debugging modules.
- Symbol Table — The module API provides access to the static and dynamic symbol tables of the target's primary executable file, its runtime link editor, and a set of load objects (shared libraries in a user process or loadable modules in the Solaris kernel).
- External Data — The module API provides a facility for retrieving a collection of named external data buffers associated with the target. For example, MDB provides programmatic access to the `proc`(4) structures associated with a user process or user core file target.

In addition, you can use built-in MDB dcmds to access information about target memory mappings, to load objects, to obtain register values, and to control the execution of user process targets.

3.3 Getting Started with MDB

In this section, we take a tour of MDB basics, from startup through elements (command syntax, expressions, symbols, and other core concepts), through simple procedures illustrated by examples.

3.3.1 Invoking MDB

MDB is available on Solaris systems as two commands that share common features: `mdb` and `kmdb`. You can use the `mdb` command interactively or in scripts to debug live user processes, user process core files, kernel crash dumps, the live operating system, object files, and other files. You can use the `kmdb` command to debug the live operating system kernel and device drivers when you also need to control and halt the execution of the kernel. To start `mdb`, execute the `mdb`(1) command.

The following example shows how mdb can be started to examine a live kernel.

```
sol8# mdb -k
Loading modules: [ unix krtld genunix specfs dtrace ufs ip sctp usba uhci s1394 fcp fctl
emlxs nca lofs zfs random nfs audiosup sppp crypto md fcip logindmux ptm ipc ]
>
```

To start mdb with a kernel crash image, specify the namelist and core image names on the command line.

```
sol8# cd /var/crash/myserver

sol8# ls /var/crash/*
bounds    unix.1    unix.3    unix.5    unix.7    vmcore.1  vmcore.3  vmcore.5  vmcore.7
unix.0    unix.2    unix.4    unix.6    vmcore.0  vmcore.2  vmcore.4  vmcore.6

sol8# mdb -k unix.1 vmcore.1
Loading modules: [ unix krtld genunix specfs dtrace ufs ip sctp usba uhci s1394 fcp fctl
emlxs nca lofs zfs random nfs audiosup sppp crypto md fcip logindmux ptm ipc ]
>
```

To start mdb with a process target, enter either a command to execute or a process ID with the -p option.

```
# mdb /usr/bin/ls
>

# mdb -p 121
Loading modules: [ ld.so.1 libumem.so.1 libc.so.1 libuutil.so.1 ]
```

To start kmdb, boot the system or execute the mdb command with the -K option as described in Section 2.8 , “Diagnosing with kmdb and moddebug,” on page 120.

3.3.2 Logging Output to a File

It’s often useful to log output to a file, so arrange for that early on by using the ::log dcmd.

```
> ::log mymdb.out
mdb: logging to "mymdb.out"
```

3.3.3 MDB Command Syntax

The MDB debugger lets us interact with the target program and the memory image of the target. The syntax is an enhanced form of that used with debuggers like adb, in which basic form is expressed as value and a command.

```
[value] [,count ] command
```

The language syntax is designed around the concept of computing the *value* of an *expression* (typically a memory address in the target), and applying a *command* to that expression. A command in MDB can be of several forms. It can be a *macro file*, a *metacharacter*, or a dcmd *pipeline*. A simple command is a metacharacter or dcmd followed by a sequence of zero or more blank-separated words. The words are typically passed as arguments. Each command returns an exit status that indicates it succeeded, failed, or was invoked with invalid arguments.

For example, if we wanted to display the contents of the word at address fec4b8d0, we could use the / metacharacter with the word X as a format specifier, and optionally a count specifying the number of iterations.

```
> fec4b8d0 /X
lotsfree:
lotsfree:       f5e
> fec4b8d0,4 /X
lotsfree:
lotsfree:       f5e             7af             3d7             28
```

MDB retains the notion of dot (.) as the current address or value, retained from the last successful command. A command with no supplied expression uses the value of dot for its argument.

```
> /X
lotsfree:
lotsfree:       f5e

> . /X
lotsfree:
lotsfree:       f5e
```

A pipeline is a sequence of one or more simple commands separated by |. Unlike the shell, dcmds in MDB pipelines are not executed as separate processes.

After the pipeline has been parsed, each dcmd is invoked in order from left to right. The full definition of a command involving pipelines is as follows.

```
[expr] [,count ] pipeline [words...]
```

Each dcmd's output is processed and stored as described in "dcmd Pipelines" in Section 3.3.8 , "dcmds". After the left-hand dcmd is complete, its processed output is used as input for the next dcmd in the pipeline. If any dcmd does not return a successful exit status, the pipeline is aborted.

For reference, Table 3.2 lists the full set of expression and pipeline combinations that form commands.

Table 3.2 General MDB Command Syntax

Command	Description
`pipeline [!word...] [;]`	basic
`expr pipeline [!word...] [;]`	set dot, run once
`expr, expr pipeline [!word...] [;]`	set dot, repeat
`,expr pipeline [!word...] [;]`	repeat
`expr [!word...] [;]`	set dot, last pipeline, run once
`,expr [!word...] [;]`	last pipeline, repeat
`expr, expr [!word...] [;]`	set dot, last pipeline, repeat
`!word... [;]`	shell escape

3.3.4 Expressions

Arithmetic expansion is performed when an MDB command is preceded by an optional *expression* representing a numerical argument for a dcmd. A list of common expressions is summarized as follows.

Table 3.3 Arithmetic Expressions

Operator	Expression
`integer`	`0i` binary `0o` octal `0t` decimal `0x` hex
`0t[0-9]+\.[0-9]+`	IEEE floating point
`'cccccccc'`	little-endian character const
`<identifier`	variable lookup
`identifier`	symbol lookup
`(expr)`	the value of expr
`.`	the value of dot
`&`	last dot used by dcmd
`+`	dot+increment
`^`	dot-increment (increment is effected by the last formatting dcmd)

Table 3.4 Unary Operators

Operator	Expression
`#expr`	logical NOT
`~expr`	bitwise NOT
`-expr`	integer negation
`%expr`	object-file pointer dereference
`%/[csil]/expr`	object-file typed dereference
`%/[1248]/expr`	object-file sized dereference
`*expr`	virtual-address pointer dereference
`*/[csil]/expr`	virtual-address typed dereference
`*/[1248]/expr`	virtual-address sized dereference
[csil] is char-, short-, int-, or long-sized	

Table 3.5 Binary Operators

Operator	Description
`expr * expr`	integer multiplication
`expr % expr`	integer division
`left # right`	left rounded up to next right multiple
`expr + expr`	integer addition
`expr - expr`	integer subtraction
`expr << expr`	bitwise left shift
`expr >> expr`	bitwise right shift (logical)
`expr == expr`	logical equality
`expr != expr`	logical inequality
`expr & expr`	bitwise AND
`expr ^ expr`	bitwise XOR
`expr | expr`	bitwise OR

An example of a simple expression is adding an integer to an address.

```
> d7c662e0+0t8/X
0xd7c662e8:     d2998b80
> d7c662e0+0t8::print int
0xd7c662e8:     d2998b80
```

3.3.5 Symbols

MDB can reference memory or objects according to the value of a symbol of the target. A symbol is the name of either a function or a global variable in the target.

For example, you compute the address of the kernel's global variable `lotsfree` by entering it as an expression, and display it by using the = metacharacter. You display the value of the `lotsfree` symbol by using the / metacharacter.

```
> lotsfree=X
                fec4b8d0

> lotsfree/D
lotsfree:
lotsfree:       3934
```

Symbol names can be resolved from kernel and userland process targets. In the kernel, the resolution of the symbol names can optionally be defined with a scope by specifying the module or object file name. In a process, symbols' scope can be defined by library or object file names. They take the following form.

Table 3.6 Resolving Symbol Names

Target	Form
kernel	`{module`}{file`}symbol`
process	`{LM[0-9]+`}{library`}{file`}symbol`

The target typically searches the primary executable's symbol tables first, then one or more of the other symbol tables. Notice that ELF symbol tables contain only entries for external, global, and static symbols; automatic symbols do not appear in the symbol tables processed by MDB.

Additionally, MDB provides a private user-defined symbol table that is searched before any of the target symbol tables are searched. The private symbol table is initially empty and can be manipulated with the `::nmadd` and `::nmdel` dcmds.

The `::nm -P` option displays the contents of the private symbol table. The private symbol table allows the user to create symbol definitions for program functions or data that were either missing from the original program or stripped out.

```
> ::nm
Value      Size       Type  Bind  Other Shndx    Name
0x00000000|0x00000000|NOTY |LOCL |0x0  |UNDEF   |
0xfec40038|0x00000000|OBJT |LOCL |0x0  |14      |_END_
0xfe800000|0x00000000|OBJT |LOCL |0x0  |1       |_START_
0xfec00000|0x00000000|NOTY |LOCL |0x0  |10      |__return_from_main
...
```

These definitions are then used whenever MDB converts a symbolic name to an address, or an address to the nearest symbol. Because targets contain multiple symbol tables and each symbol table can include symbols from multiple object files, different symbols with the same name can exist. MDB uses the backquote " ` " character as a symbol-name scoping operator to allow the programmer to obtain the value of the desired symbol in this situation.

3.3.6 Formatting Metacharacters

The /, \, ?, and = metacharacters denote the special output formatting dcmds. Each of these dcmds accepts an argument list consisting of one or more format characters, repeat counts, or quoted strings. A format character is one of the ASCII characters shown in Table 3.7.

Table 3.7 Formatting Metacharacters

Metacharacter	Description
/	Read or write virtual address from . (dot)
\	Read or write physical address from .
?	Read or write primary object file, using virtual address from .
=	Read or write the value of .

3.3.7 Formatting Characters

Format characters read or write and format data from the target. They are combined with the formatting metacharacters to read, write, or search memory. For example, if we want to display or set the value of a memory location, we could represent that location by its hexadecimal address or by its symbol name. Typically, we use a *metacharacter* with a *format* or a *dcmd* to indicate what we want MDB to do with the memory at the indicated address.

In the following example, we display the address of the kernel's lotsfree symbol. We use the = metacharacter to display the absolute value of the symbol, lotsfree and the X format to display the address in 32-bit hexadecimal notation.

```
> lotsfree=X
fec4b8d0
```

In a more common example, we can use the / metacharacter to format for display the value at the *address* of the lotsfree symbol.

```
> lotsfree/D
lotsfree:
lotsfree:       4062
```

Optionally, a repeat count can be supplied with a format. A repeat count is a positive integer preceding the format character and is always interpreted in base 10 (decimal). A repeat count can also be specified as an expression enclosed in square brackets preceded by a dollar sign ($[ ]). A string argument must be enclosed in double-quotes (" "). No blanks are necessary between format arguments.

```
> lotsfree/4D
lotsfree:
lotsfree:       3934            1967            983             40
```

If MDB is started in writable (-w) mode, then write formats are enabled. Note that this should be considered MDB's dangerous mode, especially if operating on live kernels or applications. For example, if we wanted to rewrite the value indicated by lotsfree to a new value, we could use the W write format with a valid MDB value or arithmetic expression as shown in the summary at the start of this section. For example, the W format writes the 32-bit value to the given address. In this example, we use an integer value, represented by the 0t arithmetic expression prefix.

```
> lotsfree/W 0t5000
lotsfree:
lotsfree:       f5e
```

A complete list of format strings can be found with the ::formats dcmd.

```
> ::formats
+ - increment dot by the count (variable size)
- - decrement dot by the count (variable size)
B - hexadecimal int (1 byte)
C - character using C character notation (1 byte)
D - decimal signed int (4 bytes)
E - decimal unsigned long long (8 bytes)
...
```

A summary of the common formatting characters and the required metacharacters is shown in Table 3.8 through Table 3.10.

Table 3.8 Metacharacters and Formats for Reading

Metacharacter	Description
`[/\?=][BCVbcdhoquDHOQ+-^NnTrtaIiSsE]`	value is immediate or `$[expr]`
`/`	format VA from . (dot)
`\`	format PA from .
`?`	format primary object file, using VA from .
`=`	format value of .

Format	Description	Format	Description
`B (1)`	hex	`+`	dot += increment
`C (1)`	char (C-encoded)	`-`	dot -= increment
`V (1)`	unsigned	`^ (var)`	dot -= incr*count
`b (1)`	octal	`N`	newline
`c (1)`	char (raw)	`n`	newline
`d (2)`	signed	`T`	tab
`h (2)`	hex, swap endianness	`r`	whitespace
`o (2)`	octal	`t`	tab
`q (2)`	signed octal	`a`	dot as symbol+offset
`u (2)`	decimal	`I (var)`	address and instruction
`D (4)`	signed	`i (var)`	instruction
`H (4)`	hex, swap endianness	`S (var)`	string (C-encoded)
`O (4)`	octal	`s (var)`	string (raw)
`Q (4)`	signed octal	`E (8)`	unsigned
`U (4)`	unsigned	`F (8)`	double
`X (4)`	hex	`G (8)`	octal
`Y (4)`	decoded `time32_t`	`J (8)`	hex
`f (4)`	float	`R (8)`	binary
`K (4|8)`	hex `uintptr_t`	`e (8)`	signed
`P (4|8)`	symbol	`g (8)`	signed octal
`p (4|8)`	symbol	`y (8)`	decoded `time64_t`

Table 3.9 Metacharacters and Formats for Writing

Metacharacter	Description
`[/\?][vwWZ] value...`	value is immediate or `$[expr]`
`/`	write virtual addresses
`\`	write physical addresses
`?`	write object file
Format	**Description**
`v (1)`	write low byte of each value, starting at dot
`w (2)`	write low 2 bytes of each value, starting at dot
`W (4)`	write low 4 bytes of each value, starting at dot
`Z (8)`	write all 8 bytes of each value, starting at dot

Table 3.10 Metacharacters and Formats for Searching

Metacharacter	Description
[/\?][lLM] value [mask]	value and mask are immediate or $[expr]
/	search virtual addresses
\	search physical addresses
?	search object file
Format	**Description**
l (2)	search for 2-byte value, optionally masked
L (4)	search for 4-byte value, optionally masked
M (8)	search for 8-byte value, optionally masked

3.3.8 dcmds

The metacharacters we explored in the previous section are actually forms of dcmds. The more general form of a dcmd is *::name* , where name is the command name, as summarized by

```
::{module`}d
expr>var        write the value of expr into var
```

A list of dcmds can be obtained with `::dcmds`. Alternatively, the `::dmods` command displays information about both dcmds and walkers, conveniently grouped per MDB module.

```
> ::dmods -l
genunix
...
  dcmd pfiles               - print process file information
  dcmd pgrep                - pattern match against all processes
  dcmd pid2proc             - convert PID to proc_t address
  dcmd pmap                 - print process memory map
  dcmd project              - display kernel project(s)
  dcmd prtconf              - print devinfo tree
  dcmd ps                   - list processes (and associated thr,lwp)
  dcmd ptree                - print process tree
...
```

Help on individual dcmds is available with the `help` dcmd. Yes, almost everything in MDB is implemented as a dcmd!

```
> ::help ps

NAME
  ps - list processes (and associated thr,lwp)

SYNOPSIS
  ::ps [-fltzTP]

ATTRIBUTES

  Target: kvm
  Module: genunix
  Interface Stability: Unstable
```

For example, we can optionally use `::ps` as a simple dcmd with no arguments.

```
> ::ps
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R      0      0      0      0      0 0x00000001 fffffffffbc23640 sched
R      3      0      0      0      0 0x00020001 ffffffff812278f8 fsflush
R      2      0      0      0      0 0x00020001 ffffffff81228520 pageout
R      1      0      0      0      0 0x42004000 ffffffff81229148 init
R   1782      1   1782   1782      1 0x42000000 ffffffff8121cc38 lockd
R    524      1    524    524      0 0x42000000 ffffffff8b7fd548 dmispd
R    513      1    513    513      0 0x42010000 ffffffff87bd2878 snmpdx
R    482      1      7      7      0 0x42004000 ffffffff87be90b8 intrd
R    467      1    466    466      0 0x42010000 ffffffff87bd8020 smcboot
```

Optionally, we could use the same `::ps` dcmd with an address supplied in hexadecimal.

```
> ffffffff87be90b8::ps
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R    482      1      7      7      0 0x42004000 ffffffff87be90b8 intrd

> ffffffff87be90b8 ::ps -ft
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R    482      1      7      7      0 0x42004000 ffffffff87be90b8 /usr/perl5/bin/perl
/usr/lib/intrd
        T  0xffffffff8926d4e0 <TS_SLEEP>
```

3.3.9 Walkers

A *walker* is used to traverse a connect set of data. Walkers are a type of plugin that is coded to iterate over the specified type of data. In addition to the `::dcmds` dcmd, the `::walkers` dcmd lists walkers.

```
> ::walkers
Client_entry_cache       - walk the Client_entry_cache cache
DelegStateID_entry_cache - walk the DelegStateID_entry_cache cache
File_entry_cache         - walk the File_entry_cache cache
HatHash                  - walk the HatHash cache
...
```

For example, the `::proc` walker could be used to traverse set of process structures (`proc_ts`). Many walkers also have a default data item to walk if none is specified.

```
> ::walk proc
fffffffffbc23640
ffffffff812278f8
ffffffff81228520
...
```

There are walkers to traverse common generic data structure indexes. For example, simple linked lists can be traversed with the `::list` walker, and AVL trees with the `::avl` walker.

```
> ffffffff9a647ae0::walk avl
ffffffff9087a990
fffffe85ad8aa878
fffffe85ad8aa170
...
> fffffffffbc23640::list proc_t p_prev
fffffffffbc23640
ffffffff81229148
ffffffff81228520
...
```

3.3.10 Macros

MDB provides a compatibility mode that can interpret macros built for `adb`. A macro file is a text file containing a set of commands to execute. Macro files typically automate the process of displaying a simple data structure. These older macros can therefore be used with either tool. The development of macros is discouraged, since they are difficult to construct and maintain. Following is an example of using a macro to display a data structure.

```
> d8126310$<ce
                ce instance structure
0xd8126310:     dip                 instance          dev_regs
                d8c8e840            d84b65c8          d2999900
...
```

3.3.11 Pipelines

Walkers and dcmds can build on each other, combining to do more powerful things by placement into an mdb “pipeline.”

The purpose of a pipeline is to pass a list of values, typically virtual addresses, from one dcmd or walker to another. Pipeline stages might map a pointer from one type of data structure to a pointer to a corresponding data structure, sort a list of addresses, or select the addresses of structures with certain properties.

MDB executes each dcmd in the pipeline in order from left to right. The leftmost dcmd executes with the current value of dot or with the value specified by an explicit expression at the start of the command. When a | operator is encountered, MDB creates a pipe (a shared buffer) between the output of the dcmd to its left and the MDB parser, and an empty list of values.

To give you a taste of the power of pipelines, here's an example, running against the live kernel. The `::pgrep` dcmd allows you to find all processes matching a pattern, the thread walker walks all of the threads in a process, and the `::findstack` dcmd gets a stack trace for a given thread. Connecting them into a pipeline, you can yield the stack traces of all `sshd` threads on the system (note that the middle one is swapped out). MDB pipelines are quite similar to standard UNIX pipelines and afford debugger users a similar level of power and flexibility.

```
> ::pgrep sshd
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R 100174      1 100174 100174      0 0x42000000 0000030009216790 sshd
R 276948 100174 100174 100174      0 0x42010000 000003002d9a9860 sshd
R 276617 100174 100174 100174      0 0x42010000 0000030013943010 sshd
> ::pgrep sshd | ::walk thread
3000c4f0c80
311967e9660
30f2ff2c340
> ::pgrep sshd | ::walk thread | ::findstack
stack pointer for thread 3000c4f0c80: 2a10099d071
[ 000002a10099d071 cv_wait_sig_swap+0x130() ]
  000002a10099d121 poll_common+0x530()
  000002a10099d211 pollsys+0xf8()
  000002a10099d2f1 syscall_trap32+0x1e8()
stack pointer for thread 311967e9660: 2a100897071
[ 000002a100897071 cv_wait_sig_swap+0x130() ]
stack pointer for thread 30f2ff2c340: 2a100693071
[ 000002a100693071 cv_wait_sig_swap+0x130() ]
  000002a100693121 poll_common+0x530()
  000002a100693211 pollsys+0xf8()
  000002a1006932f1 syscall_trap32+0x1e8()
```

The full list of built-in dcmds can be obtained with the `::dmods` dcmd.

```
> ::dmods -l mdb

mdb
  dcmd $<                   - replace input with macro
  dcmd $<<                  - source macro
  dcmd $>                   - log session to a file
  dcmd $?                   - print status and registers
  dcmd $C                   - print stack backtrace
...
```

3.3.12 Piping to UNIX Commands

MDB can pipe output to UNIX commands with the `!` pipe. A common task is to use `grep` to filter output from a dcmd. We've shown the output from `::ps` for illustration; actually, a handy `::pgrep` command handles this common task.

```
> ::ps !grep inet
R    255      1    255    255      0 0x42000000 ffffffff87be9ce0 inetd
```

3.3.13 Obtaining Symbolic Type Information

The MDB environment exploits the Compact Type Format (CTF) information in debugging targets. This provides symbolic type information for data structures in the target; such information can then be used within the debugging environment.

Several dcmds consume CTF information, most notably ::print. The ::print dcmd displays a target data type in native C representation. The following example shows ::print in action.

```
/* process ID info */

struct pid {
        unsigned int pid_prinactive :1;
        unsigned int pid_pgorphaned :1;
        unsigned int pid_padding :6;    /* used to be pid_ref, now an int */
        unsigned int pid_prslot :24;
        pid_t pid_id;
        struct proc *pid_pglink;
        struct proc *pid_pgtail;
        struct pid *pid_link;
        uint_t pid_ref;
};
                                                                  See sys/proc.h


> ::print -t "struct pid"
{
    unsigned pid_prinactive :1
    unsigned pid_pgorphaned :1
    unsigned pid_padding :6
    unsigned pid_prslot :24
    pid_t pid_id
    struct proc *pid_pglink
    struct proc *pid_pgtail
    struct pid *pid_link
    uint_t pid_ref
}
```

The ::print dcmd is most useful to print data structures in their typed format. For example, using a pipeline we can look up the address of the p_pidp member of the supplied proc_t structure and print its structure's contents.

```
> ::pgrep inet
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R   1595      1   1595   1595      0 0x42000400 d7c662e0 inetd

> d7c662e0::print proc_t p_pidp |::print -t "struct pid"
{
    unsigned pid_prinactive :1 = 0
    unsigned pid_pgorphaned :1 = 0x1
    unsigned pid_padding :6 = 0
    unsigned pid_prslot :24 = 0xae
    pid_t pid_id = 0x63b
    struct proc *pid_pglink = 0xd7c662e0
    struct proc *pid_pgtail = 0xd7c662e0
    struct pid *pid_link = 0
    uint_t pid_ref = 0x3
}
```

The `::print` command also understands how to traverse more complex data structures. For example, here we traverse an element of an array.

```
> d7c662e0::print proc_t p_user.u_auxv[9]
{
    p_user.u_auxv[9].a_type = 0x6
    p_user.u_auxv[9].a_un = {
        a_val = 0x1000
        a_ptr = 0x1000
        a_fcn = 0x1000
    }
}
```

Several other dcmds, listed below, use the CTF information. Starting with Solaris 9, the kernel is compiled with CTF information, making type information available by default. Starting with Solaris 10, CTF information is also available in userland, and by default some of the core system libraries contain CTF. The CTF-related commands are summarized in Table 3.11.

Table 3.11 CTF-related dcmds

dcmd	Description
`addr::print [type] [field...]`	Use CTF info to print out a full structure or particular fields thereof.

`::sizeof type`
`::offsetof type field`
`::enum enumname`
	Get information about a type.

`addr::array [type count] [var]`	Walk the count elements of an array of type `type`, starting at `addr`.
`addr::list type field [var]`	Walk a circular or NULL-terminated list of type `type`, which starts at `addr` and uses `field` as its linkage.

`::typegraph`
`addr::whattype`
`addr::istype type`
`addr::notype`
	Use the type inference engine—works on non-debug text.

3.3.14 Variables

A *variable* is a variable name, a corresponding integer value, and a set of attributes. A variable name is a sequence of letters, digits, underscores, or periods.

A variable can be assigned a value with `> dcmd` and read with `< dcmd`. Additionally, the variable can be the `::typeset` dcmd, and its attributes can be manipulated with the `::typeset` dcmd. Each variable's value is represented as a 64-bit unsigned integer. A variable can have one or more of the following attributes:

- Read-only (cannot be modified by the user)
- Persistent (cannot be unset by the user)
- Tagged (user-defined indicator)

The following examples shows assigning and referencing a variable.

```
> 0t27>myvar

> <myvar=D
                27
> $v
myvar = 1b
. = 1b
0 = f5e
b = fec00000
d = 85737
e = fe800000
m = 464c457f
t = 1a3e70
```

The CPU's registers are also exported as variables.

```
> ::vars
uesp = 0
eip = 0
myvar = 1b
cs = 0
savfp = 0
ds = 0
trapno = 0
es = 0
. = 1b
0 = f5e
1 = 0
2 = 0
ss = 0
9 = 0
fs = 0
gs = 0
_ = 0
eax = 0
b = fec00000
d = 85737
e = fe800000
eflags = 0
ebp = 0
m = 464c457f
ebx = 0
t = 1a3e70
ecx = 0
hits = 0
edi = 0
edx = 0
err = 0
esi = 0
esp = 0
savpc = 0
thread = 0
```

Commands for working with variables are summarized in Table 3.12.

Table 3.12 Variables

Variable	Description
0	Most recent value [/\?=]ed
9	Most recent count for $< dcmd
b	Base VA of the data section
d	Size of the data
e	VA of entry point
hits	Event callback match count

Table 3.12 Variables (*continued*)

Variable	Description
`m`	Magic number of primary object file, or zero
`t`	Size of text section
`thread`	TID of current representative thread

3.3.15 Walkers, Variables, and Expressions Combined

Variables can be combined with arithmetic expressions and evaluated to construct more complex pipelines, in which data is manipulated between stages. In a simple example, we might want to iterate only over processes that have a uid of zero. We can easily iterate over the processes by using a pipeline consisting of a walker and type information, which prints the `cr_uid`s for every process.

```
> ::walk proc | ::print proc_t p_cred->cr_uid
cr_uid = 0
cr_uid = 0x19
cr_uid = 0x1
cr_uid = 0
...
```

Adding an expression allows us to select only those that match a particular condition. The `::walk` dcmd takes an optional variable name, in which to place the value of the walk. In this example, the walker sets the value of `myvar` and also pipes the output of the same addresses into `::print`, which extracts the value of `proc_t->p_cred->cr_uid`. The `::eval` dcmd prints the variable `myvar` only when the expression is true; in this case when the result of the previous dcmd (the printed value of `cr_uid`) is equal to 1. The statement given to `::eval` to execute retrieves the value of the variable myvar and formats it with the K format (uint_ptr_t).

```
> ::walk proc myvar |::print proc_t p_cred->cr_uid |::grep .==1 |::eval <myvar=K
fec1d280
d318d248
d318daa8
d318e308
...
> ::walk proc myvar | ::print proc_t p_cred->cr_uid |::grep .==1 |::eval <myvar=K
|::print -d proc_t p_pidp->pid_id
p_pidp->pid_id = 0t4189
p_pidp->pid_id = 0t4187
p_pidp->pid_id = 0t4067
p_pidp->pid_id = 0t4065
...
```

3.3.16 Working With Debugging Targets

MDB can control and interact with live mdb processes or kmdb kernel targets. Typical debugging operations include starting, stopping, and stepping the target. We discuss more about controlling specific process or kmdb targets in Section 2.5 , “Debugging Processes within a Kernel Image” and Section 2.8 , “Diagnosing with kmdb and moddebug”. The common commands for controlling targets are summarized in Table 3.13.

Table 3.13 Debugging Target dcmds

dcmd	Description
::status	Print summary of current target.
$r ::regs	Display current register values for target.
$c ::stack $C	Print current stack trace ($C: with frame pointers).
addr[,b] ::dump [-g sz] [-e]	Dump at least b bytes starting at address addr. -g sets the group size; for 64-bit debugging, -g 8 is useful.
addr::dis	Disassemble text, starting around addr.

Table 3.13 Debugging Target dcmds

dcmd	Description

`[ addr ] :b`
`[ addr ] ::bp [+/-dDestT] [-c cmd] [-n count] sym ...  addr  [cmd ... ]`
	Set breakpoint at `addr`.

`$b`
`::events [-av] $b [-av]`
	Display all breakpoints.

`addr ::delete [id | all]`
`addr :d [id | all]`
	Delete a breakpoint at `addr`.

`:z`	Delete all breakpoints.

`::cont [SIG]`
`:c [SIG]`
	Continue the target program, and wait for it to terminate.

`id ::evset [+/-dDestT] [-c cmd] [-n count] id ...`	Modify the properties of one or more software event specifiers.

`::next [SIG]`
`:e [SIG]`
	Step the target program one instruction, but step over subroutine calls.

`::step [branch | over | out] [SIG]`
`:s SIG`
`:u SIG`
	Step the target program one instruction.

`addr [,len]::wp [+/-dDestT] [-rwx] [-ip] [-c cmd] [-n count]`
 	
`addr [,len]:a [cmd... ]`
`addr [,len]:p [cmd... ]`
`addr [,len]:w [cmd... ]`
	Set a watchpoint at the specified address.

3.3.17 Displaying Stacks

We can print a stack of the current address with the $c command or with $C, which also prints the stack frame address for each stack level.

```
> $c
atomic_add_32+8(0)
nfs4_async_inactive+0x3b(dc1c29c0, 0)
nfs4_inactive+0x41()
fop_inactive+0x15(dc1c29c0, 0)
vn_rele+0x4b(dc1c29c0)
snf_smap_desbfree+0x59(dda94080)

> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
d2a588c0 snf_smap_desbfree+0x59(dda94080)
```

3.3.18 Displaying Registers

We can print a stack of the current address with the $c command or with $C, which also prints the stack frame address for each stack level.

```
> ::regs (or $r)
%cs = 0x0158            %eax = 0x00000000
%ds = 0xd9820160                %ebx = 0xde453000
%ss = 0x0000            %ecx = 0x00000001
%es = 0xfe8d0160                %edx = 0xd2a58de0
%fs = 0xfec30000                %esi = 0xdc062298
%gs = 0xfe8301b0                %edi = 0x00000000

%eip = 0xfe82ca58 atomic_add_32+8
%ebp = 0xd2a58828
%esp = 0xd2a58800

%eflags = 0x00010282
  id=0 vip=0 vif=0 ac=0 vm=0 rf=1 nt=0 iopl=0x0
  status=<of,df,IF,tf,SF,zf,af,pf,cf>

  %uesp = 0xfe89ab0d
%trapno = 0xe
   %err = 0x2
```

3.3.19 Disassembling the Target

We can print a stack of the current address with the `$c` command or with `$C`, which also prints the stack frame address for each stack level.

```
> atomic_add_32+8::dis
atomic_add_32:                  movl   0x4(%esp),%eax
atomic_add_32+4:                movl   0x8(%esp),%ecx
atomic_add_32+8:                lock addl %ecx,(%eax)
atomic_add_32+0xb:              ret
```

Note that in this example, the contents of `%eax` from `$r` is zero, causing the `movl` instruction to trap with a `NULL` pointer reference at `atomic_add_32+4`.

3.3.20 Setting Breakpoints

We can set breakpoints in MDB by using `:b`. Typically, we pass a symbol name to `:b` (the name of the function of interest).

We can start the target program and then set a breakpoint for the `printf` function.

```
> printf:b

> :r

mdb: stop at 0x8050694
mdb: target stopped at:
PLT:printf:     jmp    *0x8060980
```

In this example, we stopped at the first symbol matching “`printf`”, which is actually in the procedure linkage table (PLT) (see the Linker and Libraries manual for a description of how dynamic linking works in Solaris). To match the `printf` we likely wanted, we can increase the scope of the symbol lookup. The `:c` command continues execution until the next breakpoint or until the program finishes.

```
> libc`printf:b

> :c
mdb: stop at libc.so.1`printf
mdb: target stopped at:
libc.so.1`printf:       pushl  %ebp
```

3.4 Debugging Kernel Cores

In this section we explore the rudimentary facilities within MDB for analyzing kernel crash images. The objective is not to provide an all-encompassing kernel crash analysis tutorial, but rather to introduce the most relevant MDB dcmds and techniques.

A more comprehensive guide to crash dump analysis can be found in some of the recommended reference texts, for example, *Panic!* by Chris Drake and Kimberly Brown for SPARC [8], and *Crash Dump Analysis* by Frank Hoffman for x86/x64 [12].

3.4.1 Locating and Attaching the Target

If a system has crashed, then we should have a core image saved in `/var/crash` on the target machine. The `mdb` debugger should be invoked from a system with the same architecture and Solaris revision as the crash image. The first steps are to locate the appropriate saved image and then to invoke `mdb`.

```
# cd /var/crash/nodename

# ls
bounds    unix.1    unix.3    unix.5    unix.7    vmcore.1  vmcore.3  vmcore.5  vmcore.7
unix.0    unix.2    unix.4    unix.6    vmcore.0  vmcore.2  vmcore.4  vmcore.6

# mdb -k unix.7 vmcore.7
Loading modules: [ unix krtld$c
 genunix specfs dtrace ufs ip sctp usba uhci s1394 fcp fctl nca lofs zfs random nfs
audiosup sppp crypto md fcip logindmux ptm ipc ]
>
```

3.4.2 Examining Kernel Core Summary Information

The kernel core contains important summary information from which we can extract the following:

- Revision of the kernel
- Hostname
- CPU and platform architecture of the system
- Panic string
- Module causing the panic

We can use the `::showrev` and `::status` dcmds to extract this information.

```
> ::showrev
Hostname: zones-internal
Release: 5.11
Kernel architecture: i86pc
Application architecture: i386
Kernel version: SunOS 5.11 i86pc snv_27
Platform: i86pc
> ::status
debugging crash dump vmcore.2 (32-bit) from zones-internal
operating system: 5.11 snv_27 (i86pc)
panic message: BAD TRAP: type=e (#pf Page fault) rp=d2a587c8 addr=0 occurred in module
"unix" due to a NULL pointer dereference
dump content: kernel pages only
> ::panicinfo
             cpu        0
          thread d2a58de0
        message BAD TRAP: type=e (#pf Page fault) rp=d2a587c8 addr=0 occurred in module
"unix" due to a NULL pointer dereference
              gs fe8301b0
              fs fec30000
              es fe8d0160
              ds d9820160
             edi        0
             esi dc062298
             ebp d2a58828
             esp d2a58800
             ebx de453000
             edx d2a58de0
             ecx        1
             eax        0
          trapno        e
             err        2
             eip fe82ca58
              cs      158
          eflags    10282
            uesp fe89ab0d
              ss        0
             gdt fec1f2f002cf
             idt fec1f5c007ff
             ldt      140
            task      150
             cr0 8005003b
             cr2        0
             cr3  4cb3000
             cr4      6d8
```

3.4.3 Examining the Message Buffer

The kernel keeps a cyclic buffer of the recent kernel messages. In this buffer we can observe the messages up to the time of the panic. The `::msgbuf` dcmd shows the contents of the buffer.

```
> ::msgbuf
MESSAGE
/pseudo/zconsnex@1/zcons@5 (zcons5) online
/pseudo/zconsnex@1/zcons@6 (zcons6) online
/pseudo/zconsnex@1/zcons@7 (zcons7) online
pseudo-device: ramdisk1024
...
panic[cpu0]/thread=d2a58de0:
BAD TRAP: type=e (#pf Page fault) rp=d2a587c8 addr=0 occurred in module "unix" due to a
NULL pointer dereference


sched:
#pf Page fault
Bad kernel fault at addr=0x0
pid=0, pc=0xfe82ca58, sp=0xfe89ab0d, eflags=0x10282
cr0: 8005003b<pg,wp,ne,et,ts,mp,pe> cr4: 6d8<xmme,fxsr,pge,mce,pse,de>
cr2: 0 cr3: 4cb3000
         gs: fe8301b0  fs: fec30000  es: fe8d0160  ds: d9820160
        edi:        0 esi: dc062298 ebp: d2a58828 esp: d2a58800
        ebx: de453000 edx: d2a58de0 ecx:        1 eax:        0
        trp:        e err:        2 eip: fe82ca58  cs:      158
        efl:    10282 usp: fe89ab0d  ss:        0
...
```

3.4.4 Obtaining a Stack Trace of the Running Thread

We can obtain a stack backtrace of the current thread by using the `$C` command. Note that the displayed arguments to each function are not necessarily accurate. On each platform, the meaning of the shown arguments is as follows:

- SPARC: The values of the arguments if they are available from a saved stack frame, assuming they are not overwritten by use of registers during the called function. With SPARC architectures, a function's input argument registers are sometimes saved on the way out of a function—if the input registers are reused during the function, then values of the input arguments are overwritten and lost.
- x86: Accurate values of the input arguments. Input arguments are always saved onto the stack and can be accurately displayed

- x64: The values of the arguments, assuming they are available. As with the SPARC architectures, input arguments are passed in registers and may be overwritten.

```
> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
d2a588c0 snf_smap_desbfree+0x59(dda94080)
d2a588dc dblk_lastfree_desb+0x13(de45b520, d826fb40)
d2a588f4 dblk_decref+0x4e(de45b520, d826fb40)
d2a58918 freemsg+0x69(de45b520)
d2a5893c FreeTxSwPacket+0x3b(d38b84f0)
d2a58968 CleanTxInterrupts+0xb4(d2f9cac0)
d2a589a4 e1000g_send+0xf6(d2f9cac0, d9ffba00)
d2a589c0 e1000g_m_tx+0x22()
d2a589dc dls_tx+0x16(d4520f68, d9ffba00)
d2a589f4 str_mdata_fastpath_put+0x1e(d3843f20, d9ffba00)
d2a58a40 tcp_send_data+0x62d(db0ecac0, d97ee250, d9ffba00)
d2a58aac tcp_send+0x6b6(d97ee250, db0ecac0, 564, 28, 14, 0)
d2a58b40 tcp_wput_data+0x622(db0ecac0, 0, 0)
d2a58c28 tcp_rput_data+0x2560(db0ec980, db15bd20, d2d45f40)
d2a58c40 tcp_input+0x3c(db0ec980, db15bd20, d2d45f40)
d2a58c78 squeue_enter_chain+0xe9(d2d45f40, db15bd20, db15bd20, 1, 1)
d2a58cec ip_input+0x658(d990e554, d3164010, 0, e)
d2a58d40 i_dls_link_ether_rx+0x156(d4523db8, d3164010, db15bd20)
d2a58d70 mac_rx+0x56(d3520200, d3164010, db15bd20)
d2a58dac e1000g_intr+0xa6(d2f9cac0, 0)
d2a58ddc intr_thread+0x122()
```

3.4.5 Which Process?

If the stack trace is of a kernel housekeeping or interrupt thread, the process reported for the thread will be that of `p0`—"`sched`." The process pointer for the thread can be obtained with `::thread`, and `::ps` will then display summary information about that process. In this example, the thread is an interrupt thread (as indicated by the top entry in the stack from `$C`), and the process name maps to `sched`.

```
> d2a58de0::thread -p
    ADDR     PROC      LWP     CRED
d2a58de0 fec1d280        0 d9d1cf38
> fec1d280::ps -t
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R      0      0      0      0      0 0x00000001 fec1d280 sched
        T        t0 <TS_STOPPED>
```

3.4.6 Disassembling the Suspect Code

Once we've located the thread of interest, we often learn more about what happened by disassembling the target and looking at the instruction that reportedly caused the panic. MDB's `::dis` dcmd will disassemble the code around the target instruction that we extract from the stack backtrace.

```
> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
...
> nfs4_async_inactive+0x3b::dis
nfs4_async_inactive+0x1a:       pushl  $0x28
nfs4_async_inactive+0x1c:       call   +0x51faa30       <kmem_alloc>
nfs4_async_inactive+0x21:       addl   $0x8,%esp
nfs4_async_inactive+0x24:       movl   %eax,%esi
nfs4_async_inactive+0x26:       movl   $0x0,(%esi)
nfs4_async_inactive+0x2c:       movl   -0x4(%ebp),%eax
nfs4_async_inactive+0x2f:       movl   %eax,0x4(%esi)
nfs4_async_inactive+0x32:       movl   0xc(%ebp),%edi
nfs4_async_inactive+0x35:       pushl  %edi
nfs4_async_inactive+0x36:       call   +0x51b7cdc       <crhold>
nfs4_async_inactive+0x3b:       addl   $0x4,%esp
nfs4_async_inactive+0x3e:       movl   %edi,0x8(%esi)
nfs4_async_inactive+0x41:       movl   $0x4,0xc(%esi)
nfs4_async_inactive+0x48:       leal   0xe0(%ebx),%eax
nfs4_async_inactive+0x4e:       movl   %eax,-0x8(%ebp)
nfs4_async_inactive+0x51:       pushl  %eax
nfs4_async_inactive+0x52:       call   +0x51477f4       <mutex_enter>
nfs4_async_inactive+0x57:       addl   $0x4,%esp
nfs4_async_inactive+0x5a:       cmpl   $0x0,0xd4(%ebx)
nfs4_async_inactive+0x61:       je     +0x7e    <nfs4_async_inactive+0xdf>
nfs4_async_inactive+0x63:       cmpl   $0x0,0xd0(%ebx)
> crhold::dis
crhold:                         pushl  %ebp
crhold+1:                       movl   %esp,%ebp
crhold+3:                       andl   $0xfffffff0,%esp
crhold+6:                       pushl  $0x1
crhold+8:                       movl   0x8(%ebp),%eax
crhold+0xb:                     pushl  %eax
crhold+0xc:                     call   -0x6e0b8 <atomic_add_32>
crhold+0x11:                    movl   %ebp,%esp
crhold+0x13:                    popl   %ebp
crhold+0x14:                    ret
> atomic_add_32::dis
atomic_add_32:                  movl   0x4(%esp),%eax
atomic_add_32+4:                movl   0x8(%esp),%ecx
atomic_add_32+8:                lock addl %ecx,(%eax)
atomic_add_32+0xb:              ret
```

3.4.7 Displaying General-Purpose Registers

In this example, the system had a NULL pointer reference at `atomic_add_32+8(0)`. The faulting instruction was atomic, referencing the memory at the location pointed to by `%eax`. By looking at the registers at the time of the panic, we can see that `%eax` was indeed NULL. The next step is to attempt to find out *why* `%eax` was NULL.

```
> ::regs
%cs = 0x0158            %eax = 0x00000000
%ds = 0xd9820160                %ebx = 0xde453000
%ss = 0x0000            %ecx = 0x00000001
%es = 0xfe8d0160                %edx = 0xd2a58de0
%fs = 0xfec30000                %esi = 0xdc062298
%gs = 0xfe8301b0                %edi = 0x00000000

%eip = 0xfe82ca58 atomic_add_32+8
%ebp = 0xd2a58828
%esp = 0xd2a58800

%eflags = 0x00010282
  id=0 vip=0 vif=0 ac=0 vm=0 rf=1 nt=0 iopl=0x0
  status=<of,df,IF,tf,SF,zf,af,pf,cf>

  %uesp = 0xfe89ab0d
%trapno = 0xe
   %err = 0x2
```

3.4.8 Navigating the Stack Backtrace

The function prototype for `atomic_add_32()` reveals that the first argument is a pointer to the memory location to be added. Since this was an x86 machine, the

```
void
atomic_add_32(volatile uint32_t *target, int32_t delta)
{
        *target += delta;
}


> atomic_add_32::dis
atomic_add_32:                  movl   0x4(%esp),%eax
atomic_add_32+4:                movl   0x8(%esp),%ecx
atomic_add_32+8:                lock addl %ecx,(%eax)
atomic_add_32+0xb:              ret
> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
...
```

arguments reported by the stack backtrace are known to be useful, and we can look to see where the NULL pointer was handed down—in this case nfs4_async_inactive().

```
> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
...
> nfs4_async_inactive+0x3b::dis
nfs4_async_inactive+0x1a:       pushl  $0x28
nfs4_async_inactive+0x1c:       call   +0x51faa30       <kmem_alloc>
nfs4_async_inactive+0x21:       addl   $0x8,%esp
nfs4_async_inactive+0x24:       movl   %eax,%esi
nfs4_async_inactive+0x26:       movl   $0x0,(%esi)
nfs4_async_inactive+0x2c:       movl   -0x4(%ebp),%eax
nfs4_async_inactive+0x2f:       movl   %eax,0x4(%esi)
nfs4_async_inactive+0x32:       movl   0xc(%ebp),%edi
nfs4_async_inactive+0x35:       pushl  %edi
nfs4_async_inactive+0x36:       call   +0x51b7cdc       <crhold>
nfs4_async_inactive+0x3b:       addl   $0x4,%esp
nfs4_async_inactive+0x3e:       movl   %edi,0x8(%esi)
nfs4_async_inactive+0x41:       movl   $0x4,0xc(%esi)
nfs4_async_inactive+0x48:       leal   0xe0(%ebx),%eax
nfs4_async_inactive+0x4e:       movl   %eax,-0x8(%ebp)
nfs4_async_inactive+0x51:       pushl  %eax
nfs4_async_inactive+0x52:       call   +0x51477f4       <mutex_enter>
nfs4_async_inactive+0x57:       addl   $0x4,%esp
nfs4_async_inactive+0x5a:       cmpl   $0x0,0xd4(%ebx)
nfs4_async_inactive+0x61:       je     +0x7e    <nfs4_async_inactive+0xdf>
nfs4_async_inactive+0x63:       cmpl   $0x0,0xd0(%ebx)
...
```

Looking at the disassembly, it appears that there is an additional function call, which is omitted from the stack backtrack (typically due to tail call compiler optimization). The call is to crhold(), passing the address of a credential structure from the arguments to `nfs4_async_inactive()`. Here we can see that `crhold()` does in fact call `atomic_add_32()`.

```
/*
 * Put a hold on a cred structure.
 */
void
crhold(cred_t *cr)
{
        atomic_add_32(&cr->cr_ref, 1);
}


> crhold::dis
crhold:                         pushl  %ebp
crhold+1:                       movl   %esp,%ebp
crhold+3:                       andl   $0xfffffff0,%esp
crhold+6:                       pushl  $0x1
crhold+8:                       movl   0x8(%ebp),%eax
crhold+0xb:                     pushl  %eax
crhold+0xc:                     call   -0x6e0b8 <atomic_add_32>
crhold+0x11:                    movl   %ebp,%esp
crhold+0x13:                    popl   %ebp
crhold+0x14:                    ret
```

Next, we look into the situation in which `nfs4_async_inactive()` was called. The first argument is a `vnode` pointer, and the second is our suspicious credential pointer. The `vnode` pointer can be examined with the CTF information and the `::print` dcmd. We can see that we were performing an `nfs4_async_inactive` function on the `vnode` referencing a pdf file in this case.

```
 */
void
nfs4_async_inactive(vnode_t *vp, cred_t *cr)
{



> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
> dc1c29c0::print vnode_t
{
...
    v_type = 1 (VREG)
    v_rdev = 0
...
    v_path = 0xdc3de800 "/zones/si/root/home/ftp/book/solarisinternals_projtaskipc.pdf"
...
}
```

Looking further at the stack backtrace and the code, we can try to identify where the credentials were derived from. nfs4_async_inactive() was called by nfs4_inactive(), which is one of the standard VOP methods (VOP_INACTIVE).

```
> $C
d2a58828 atomic_add_32+8(0)
d2a58854 nfs4_async_inactive+0x3b(dc1c29c0, 0)
d2a58880 nfs4_inactive+0x41()
d2a5889c fop_inactive+0x15(dc1c29c0, 0)
d2a588b0 vn_rele+0x4b(dc1c29c0)
```

The credential can be followed all the way up to vn_rele(), which derives the pointer from CRED(), which references the current thread's t_cred.

```
vn_rele(vnode_t *vp)
{
        if (vp->v_count == 0)
                cmn_err(CE_PANIC, "vn_rele: vnode ref count 0");
        mutex_enter(&vp->v_lock);
        if (vp->v_count == 1) {
                mutex_exit(&vp->v_lock);
                VOP_INACTIVE(vp, CRED());
...

#define CRED()          curthread->t_cred
```

We know which thread called vn_rele()—the interrupt thread with a thread pointer of d2a58de0. We can use ::print to take a look at the thread's t_cred.

```
> d2a58de0::print kthread_t t_cred
t_cred = 0xd9d1cf38
```

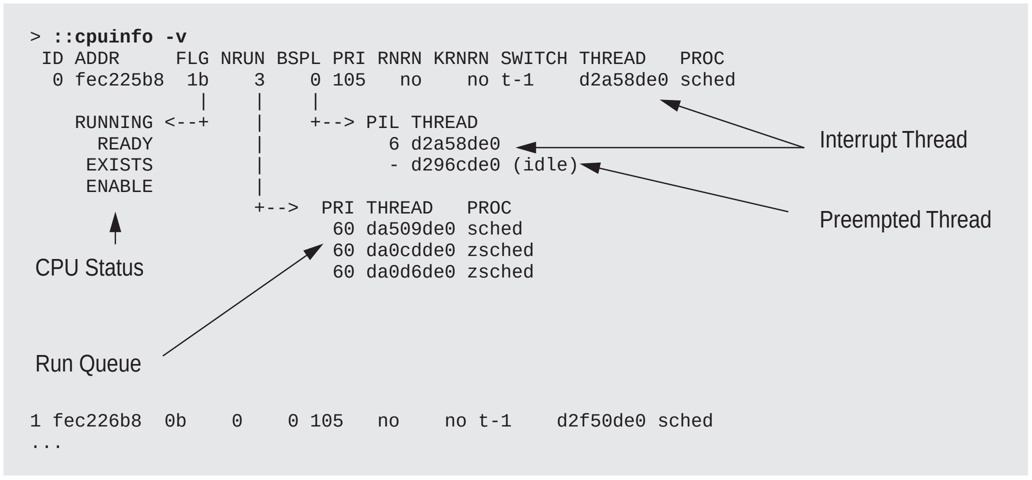
Interestingly, it's not NULL! A further look around the code gives us some clues as to what's going on. In the initialization code during the creation of an interrupt thread, the `t_cred` is set to `NULL`:

```
/*
 * Create and initialize an interrupt thread.
 *      Returns non-zero on error.
 *      Called at spl7() or better.
 */
void
thread_create_intr(struct cpu *cp)
{
...
        /*
         * Nobody should ever reference the credentials of an interrupt
         * thread so make it NULL to catch any such references.
         */
        tp->t_cred = NULL;
```

Our `curthread->t_cred` is not `NULL`, but `NULL` was passed in when `CRED()` accessed it in the not-too-distant past—an interesting situation indeed. It turns out that the NFS client code wills credentials to the interrupt thread's `t_cred`, so what we are in fact seeing is a race condition, where `vn_rele()` is called from the interrupt thread with no credentials. In this case, a bug was logged accordingly and the problem was fixed!

3.4.9 Looking at the Status of the CPUs

Another good source of information is the `::cpuinfo` dcmd. It shows a rich set of information of the processors in the system. For each CPU, the details of the thread currently running on each processor are shown. If the current CPU is handling an interrupt, then the thread running the interrupt and the preempted thread are shown. In addition, a list of threads waiting in the run queue for this processor is shown.



```
> ::cpuinfo -v
 ID ADDR     FLG NRUN BSPL PRI RNRN KRNRN SWITCH THREAD   PROC
  0 fec225b8  1b    3    0 105   no    no t-1    d2a58de0 sched
               |    |    |
    RUNNING <--+    |    +--> PIL THREAD
      READY         |           6 d2a58de0
     EXISTS         |           - d296cde0 (idle)
     ENABLE         |
                    +-->  PRI THREAD   PROC
                           60 da509de0 sched
                           60 da0cdde0 zsched
                           60 da0d6de0 zsched

1 fec226b8  0b    0    0 105   no    no t-1    d2f50de0 sched
...
```

In this example, we can see that the idle thread was preempted by a level 6 interrupt. Three threads are on the run queue: the thread that was running immediately before preemption and two other threads waiting to be scheduled on the run queue. We can traverse these manually, by traversing the stack of the thread pointer with `::findstack`.

```
> :da509de0:findstack
stack pointer for thread da509de0: da509d08
  da509d3c swtch+0x165()
  da509d60 cv_timedwait+0xa3()
  da509dc8 taskq_d_thread+0x149()
  da509dd8 thread_start+8()
```

The CPU containing the thread that caused the panic will, we hope, be reported in the panic string and, furthermore, will be used by MDB as the default thread for other dcmds in the core image. Once we determine the status of the CPU, we can observe which thread was involved in the panic.

Additionally, we can use the CPU's run queue (cpu_dispq) to provide a stack list for other threads queued up to run. We might do this just to gather a little more information about the circumstance in which the panic occurred.

```
> fec225b8::walk cpu_dispq |::thread
    ADDR    STATE  FLG PFLG SFLG   PRI  EPRI PIL     INTR DISPTIME BOUND PR
da509de0 run         8    0   13    60     0   0      n/a   7e6f9c    -1  0
da0cdde0 run         8 2000   13    60     0   0      n/a   7e8452    -1  0
da0d6de0 run         8 2000   13    60     0   0      n/a   7e8452    -1  0
> fec225b8::walk cpu_dispq |::findstack
stack pointer for thread da509de0: da509d08
  da509d3c swtch+0x165()
  da509d60 cv_timedwait+0xa3()
  da509dc8 taskq_d_thread+0x149()
  da509dd8 thread_start+8()
stack pointer for thread da0cdde0: da0cdd48
  da0cdd74 swtch+0x165()
  da0cdd84 cv_wait+0x4e()
  da0cddc8 nfs4_async_manager+0xc9()
  da0cddd8 thread_start+8()
stack pointer for thread da0d6de0: da0d6d48
  da0d6d74 swtch+0x165()
  da0d6d84 cv_wait+0x4e()
  da0d6dc8 nfs4_async_manager+0xc9()
  da0d6dd8 thread_start+8()
```

3.4.10 Traversing Stack Frames in SPARC Architectures

We briefly mentioned in Section 3.4.4 , "Obtaining a Stack Trace of the Running Thread" some of the problems we encounter when trying to glean argument values from stack backtraces. In the SPARC architecture, the values of the input arguments' registers are saved into register windows at the exit of each function. In most cases, we can traverse the stack frames to look at the values of the registers as they are saved in register windows. Historically, this was done by manually traversing the stack frames (as illustrated in *Panic!*). Conveniently, MDB has a dcmd that understands and walks SPARC stack frames. We can use the

`::stackregs` dcmd to display the SPARC input registers and locals (`%l0-%l7`) for each frame on the stack.

```
> ::stackregs
000002a100d074c1 vpanic(12871f0, e, e, fffffffffffffffe, 1, 185d400)
  %l0-%l3:                0      2a100d07f10      2a100d07f40         ffffffff
  %l4-%l7: fffffffffffffffe                0          1845400          1287000
  px_err_fabric_intr+0xbc: call      -0x1946c0     <fm_panic>

000002a100d07571 px_err_fabric_intr+0xbc(600024f9880, 31, 340, 600024d75d0,
30000842020, 0)
  %l0-%l3:                0      2a100d07f10      2a100d07f40         ffffffff
  %l4-%l7: fffffffffffffffe                0          1845400          1287000
  px_msiq_intr+0x1ac:      call      -0x13b0       <px_err_fabric_intr>

000002a100d07651 px_msiq_intr+0x1ac(60002551db8, 0, 127dcc8, 6000252e9e0, 30000828a58,
30000842020)
  %l0-%l3:                0      2a100d07f10      2a100d07f40      2a100d07f10
  %l4-%l7:                0               31      30000842020      600024d21d8
  current_thread+0x174:    jmpl      %o5, %o7

000002a100d07751 current_thread+0x174(16, 2000, ddf7dfff, ddf7ffff, 2000, 12)
  %l0-%l3:          100994c      2a100cdf021                e              7b9
  %l4-%l7:                0                0                0      2a100cdf8d0
  cpu_halt+0x134:          call      -0x29dcc      <enable_vec_intr>

000002a100cdf171 cpu_halt+0x134(16, d, 184bbd0, 30001334000, 16, 1)
  %l0-%l3:      60001db16c8                0      60001db16c8 ffffffffffffffff
  %l4-%l7:                0                0                0          10371d0
  idle+0x124:              jmpl      %l7, %o7

000002a100cdf221 idle+0x124(1819800, 0, 30001334000, ffffffffffffffff, e, 1818400)
  %l0-%l3:      60001db16c8               1b                0 ffffffffffffffff
  %l4-%l7:                0                0                0          10371d0
  thread_start+4:          jmpl      %i7, %o7

000002a100cdf2d1 thread_start+4(0, 0, 0, 0, 0, 0)
  %l0-%l3:                0                0                0                0
  %l4-%l7:                0                0                0                0
```

SPARC input registers become output registers, which are then saved on the stack. The common technique when trying to qualify registers as valid arguments is to ascertain, before the registers are saved in the stack frame, whether they have been overwritten during the function. A common technique is to disassemble the target function, looking to see if the input registers (`%i0-%i7`) are reused in the function's code body. A quick and dirty way to look for register usage is to use `::dis` piped to a UNIX `grep`; however, at this stage, examining the code for use of input registers is left as an exercise for the reader. For example, if we are looking to see if the values of the first argument to `cpu_halt()` are valid, we could see if

%i0 is reused during the cpu_halt() function, before we branch out at cpu_halt+0x134.

```
> cpu_halt::dis !grep i0
cpu_halt+0x24:                ld        [%g1 + 0x394], %i0
cpu_halt+0x28:                cmp       %i0, 1
cpu_halt+0x90:                add       %i2, 0x120, %i0
cpu_halt+0xd0:                srl       %i4, 0, %i0
cpu_halt+0x100:               srl       %i4, 0, %i0
cpu_halt+0x144:               ldub      [%i3 + 0xf9], %i0
cpu_halt+0x150:               and       %i0, 0xfd, %l7
cpu_halt+0x160:               add       %i2, 0x120, %i0
```

As we can see in this case, %i0 is reused very early in cpu_halt() and would be invalid in the stack backtrace.

3.4.11 Listing Processes and Process Stacks

We can obtain the list of processes by using the `::ps` dcmd. In addition, we can search for processes by using the `pgrep`(1M)-like `::pgrep` dcmd.

```
> ::ps -f
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R      0      0      0      0      0 0x00000001 fec1d280 sched
R      3      0      0      0      0 0x00020001 d318d248 fsflush
R      2      0      0      0      0 0x00020001 d318daa8 pageout
R      1      0      0      0      0 0x42004000 d318e308 /sbin/init
R   9066      1   9066   9066      1 0x52000400 da2b7130 /usr/lib/nfs/nfsmapid
R   9065      1   9063   9063      1 0x42000400 d965a978 /usr/lib/nfs/nfs4cbd
R   4125      1   4125   4125      0 0x42000400 d9659420 /local/local/bin/httpd -k start
R   9351   4125   4125   4125  40000 0x52000000 da2c0428 /local/local/bin/httpd -k start
R   4118      1   4117   4117      1 0x42000400 da2bc988 /usr/lib/nfs/nfs4cbd
R   4116      1   4116   4116      1 0x52000400 d8da7240 /usr/lib/nfs/nfsmapid
R   4105      1   4105   4105      0 0x42000400 d9664108 /usr/apache/bin/httpd
R   4263   4105   4105   4105  60001 0x52000000 da2bf368 /usr/apache/bin/httpd
...
> ::ps -t
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R      0      0      0      0      0 0x00000001 fec1d280 sched
        T        t0 <TS_STOPPED>
R      3      0      0      0      0 0x00020001 d318d248 fsflush
        T  0xd3108a00 <TS_SLEEP>
R      2      0      0      0      0 0x00020001 d318daa8 pageout
        T  0xd3108c00 <TS_SLEEP>
R      1      0      0      0      0 0x42004000 d318e308 init
        T  0xd3108e00 <TS_SLEEP>
R   9066      1   9066   9066      1 0x52000400 da2b7130 nfsmapid
        T  0xd942be00 <TS_SLEEP>
        T  0xda68f000 <TS_SLEEP>
        T  0xda4e8800 <TS_SLEEP>
        T  0xda48f800 <TS_SLEEP>
...

::pgrep httpd
> ::pgrep http
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R   4125      1   4125   4125      0 0x42000400 d9659420 httpd
R   9351   4125   4125   4125  40000 0x52000000 da2c0428 httpd
R   4105      1   4105   4105      0 0x42000400 d9664108 httpd
R   4263   4105   4105   4105  60001 0x52000000 da2bf368 httpd
R   4111   4105   4105   4105  60001 0x52000000 da2b2138 httpd
...
```

We can observe several aspects of the user process by using the *ptool-like* dcmds.

```
> ::pgrep nscd
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R    575      1    575    575      0 0x42000000 ffffffff866f1878 nscd

> 0t575 |::pid2proc |::walk thread |::findstack
(or)
> ffffffff82f5f860::walk thread |::findstack
stack pointer for thread ffffffff866cb060: fffffe8000c7fdd0
[ fffffe8000c7fdd0 _resume_from_idle+0xde() ]
  fffffe8000c7fe10 swtch+0x185()
  fffffe8000c7fe80 cv_wait_sig_swap_core+0x17a()
  fffffe8000c7fea0 cv_wait_sig_swap+0x1a()
  fffffe8000c7fec0 pause+0x59()
  fffffe8000c7ff10 sys_syscall32+0x101()
...

> ffffffff866f1878::ptree
fffffffffbc23640  sched
     ffffffff82f6b148  init
          ffffffff866f1878  nscd

> ffffffff866f1878::pfiles
FD   TYPE            VNODE INFO
   0  CHR ffffffff833d4700 /devices/pseudo/mm@0:null
   1  CHR ffffffff833d4700 /devices/pseudo/mm@0:null
   2  CHR ffffffff833d4700 /devices/pseudo/mm@0:null
   3 DOOR ffffffff86a0eb40 [door to 'nscd' (proc=ffffffff866f1878)]
   4 SOCK ffffffff835381c0

> ffffffff866f1878::pmap
             SEG             BASE     SIZE      RES PATH
ffffffff85e416c0 0000000008046000       8k       8k [ anon ]
ffffffff866ab5e8 0000000008050000      48k          /usr/sbin/nscd
ffffffff839b1950 000000000806c000       8k       8k /usr/sbin/nscd
ffffffff866ab750 000000000806e000     520k     480k [ anon ]
...
```

3.4.12 Global Memory Summary

The major buckets of memory allocation are available with the `::memstat` dcmd.

```
> ::memstat
Page Summary                Pages                MB  %Tot
------------     ----------------  ----------------  ----
Kernel                      49022               191   19%
Anon                        68062               265   27%
Exec and libs                3951                15    2%
Page cache                   4782                18    2%
Free (cachelist)             7673                29    3%
Free (freelist)            118301               462   47%

Total                      251791               983
Physical                   251789               983
```

3.4.13 Listing Network Connections

We can use the `::netstat` dcmd to obtain the list of network connections.

```
> ::netstat
TCPv4    St   Local Address        Remote Address       Zone
da348600  6      10.0.5.104.63710       10.0.5.10.38189    7
da348a80  0      10.0.5.106.1016        10.0.5.10.2049     2
da34fc40  0      10.0.5.108.1018        10.0.5.10.2049     3
da3501c0  0      10.0.4.106.22       192.18.42.17.64836    2
d8ed2800  0      10.0.4.101.22       192.18.42.17.637
...
```

3.4.14 Listing All Kernel Threads

A stack backtrace of all threads in the kernel can be obtained with the `::threadlist` dcmd. (If you are familiar with `adb`, this is a modern version of `adb`'s `$<threadlist` macro). With this dcmd, we can quickly and easily capture a useful snapshot of all current activity in text form, for deeper analysis.

```
> ::threadlist
    ADDR     PROC      LWP CMD/LWPID
fec1dae0 fec1d280 fec1fdc0 sched/1
d296cde0 fec1d280        0 idle()
d2969de0 fec1d280        0 taskq_thread()
d2966de0 fec1d280        0 taskq_thread()
d2963de0 fec1d280        0 taskq_thread()
d2960de0 fec1d280        0 taskq_thread()
d29e3de0 fec1d280        0 taskq_thread()
d29e0de0 fec1d280        0 taskq_thread()
...
> ::threadlist -v
    ADDR     PROC      LWP CLS PRI    WCHAN
fec1dae0 fec1d280 fec1fdc0   0  96        0
  PC: 0xfe82b507    CMD: sched
  stack pointer for thread fec1dae0: fec33df8
    swtch+0x165()
    sched+0x3aa()
    main+0x365()

d296cde0 fec1d280        0   0  -1        0
  PC: 0xfe82b507    THREAD: idle()
  stack pointer for thread d296cde0: d296cd88
    swtch+0x165()
    idle+0x32()
    thread_start+8()
...

# echo "::threadlist" |mdb -k >mythreadlist.txt
```

3.4.15 Other Notable Kernel dcmds

The ::findleaks dcmd efficiently detects memory leaks in kernel crash dumps when the full set of kmem debug features has been enabled. The first execution of ::findleaks processes the dump for memory leaks (this can take a few minutes), then coalesces the leaks by the allocation stack trace. The findleaks report shows a bufctl address and the topmost stack frame for each memory leak that was identified. See Section 16.4.9.1 , “Finding Memory Leaks,” on page 718 for more information on ::findleaks.

```
> ::findleaks
CACHE     LEAKED   BUFCTL CALLER
70039ba8       1 703746c0 pm_autoconfig+0x708
70039ba8       1 703748a0 pm_autoconfig+0x708
7003a028       1 70d3b1a0 sigaddq+0x108
7003c7a8       1 70515200 pm_ioctl+0x187c
------------------------------------------------------
   Total       4 buffers, 376 bytes
```

If the -v option is specified, the dcmd prints more verbose messages as it executes. If an explicit address is specified prior to the dcmd, the report is filtered and only leaks whose allocation stack traces contain the specified function address are displayed.

The ::vatopfn dcmd translates virtual addresses to physical addresses, using the appropriate platform translation tables.

```
> fec4b8d0::vatopfn
        level=1 htable=d9d53848 pte=30007e3
Virtual fec4b8d0 maps Physical 304b8d0
```

The ::whatis dcmd attempts to determine if the address is a pointer to a kmem-managed buffer or another type of special memory region, such as a thread stack, and reports its findings. When the -a option is specified, the dcmd reports all matches instead of just the first match to its queries. When the -b option is specified, the dcmd also attempts to determine if the address is referred to by a known kmem bufctl. When the -v option is specified, the dcmd reports its progress as it searches various kernel data structures. See Section 16.4.9.2 , “Finding References to Data,” on page 719 for more information on `::whatis`.

```
> 0x705d8640::whatis
705d8640 is 705d8640+0, allocated from streams_mblk
```

The `::kgrep` dcmd lets you search the kernel for occurrences of a supplied value. This is particularly useful when you are trying to debug software with multiple instances of a value.

```
> 0x705d8640::kgrep
400a3720
70580d24
7069d7f0
706a37ec
706add34
```

3.5 Examining User Process Stacks Within a Kernel Image

A kernel crash dump can save memory pages of user processes in Solaris. We explain how to save process memory pages and how to examine user processes by using the kernel crash dump.

3.5.1 Enabling Process Pages in a Dump

We must modify the dump configuration to save process pages. We confirm the dump configuration by running `dumpadm` with no option.

```
# /usr/sbin/dumpadm
         Dump content: all pages
          Dump device: /dev/dsk/c0t0d0s1 (swap)
   Savecore directory: /var/crash/example
     Savecore enabled: yes
```

If `Dump content` is not `all pages` or `curproc`, no process memory page will be dumped. In that case, we run `dumpadm -c all` or `dumpadm -c curproc`.

3.5.2 Invoking MDB to Examine the Kernel Image

We gather a crash dump and confirm that user pages are contained.

```
# /usr/bin/mdb unix.0 vmcore.0
   Loading modules: [ unix krtld genunix ufs_log ip nfs random ptm
   logindmux ]

> ::status
debugging crash dump vmcore.0 (64-bit) from rmcferrari
operating system: 5.11 snv_31 (i86pc)
panic message: forced crash dump initiated at user request
dump content: all kernel and user pages
```

The `dump content` line shows that this dump includes user pages.

3.5.3 Locating the Target Process

Next, we search for process information with which we are concerned. We use `nscd` as the target of this test case. The first thing to find is the address of the process.

```
> ::pgrep nscd
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R    575      1    575    575      0 0x42000000 ffffffff866f1878 nscd
```

The address of the process is `ffffffff866f1878`. As a sanity check, we can look at the kernel thread stacks for each process—we'll use these later to double-check that the user stack matches the kernel stack, for those threads blocked in a system call.

```
> 0t575::pid2proc |::print proc_t p_tlist |::list kthread_t t_forw |::findstack
stack pointer for thread ffffffff866cb060: fffffe8000c7fdd0
[ fffffe8000c7fdd0 _resume_from_idle+0xde() ]
  fffffe8000c7fe10 swtch+0x185()
  fffffe8000c7fe80 cv_wait_sig_swap_core+0x17a()
  fffffe8000c7fea0 cv_wait_sig_swap+0x1a()
  fffffe8000c7fec0 pause+0x59()
  fffffe8000c7ff10 sys_syscall32+0x101()
stack pointer for thread ffffffff866cc140: fffffe8000c61d70
[ fffffe8000c61d70 _resume_from_idle+0xde() ]
  fffffe8000c61db0 swtch+0x185()
  fffffe8000c61e10 cv_wait_sig+0x150()
  fffffe8000c61e50 door_unref+0x94()
  fffffe8000c61ec0 doorfs32+0x90()
  fffffe8000c61f10 sys_syscall32+0x101()
stack pointer for thread ffffffff866cba80: fffffe8000c6dd10
[ fffffe8000c6dd10 _resume_from_idle+0xde() ]
  fffffe8000c6dd50 swtch_to+0xc9()
  fffffe8000c6ddb0 shuttle_resume+0x376()
  fffffe8000c6de50 door_return+0x228()
  fffffe8000c6dec0 doorfs32+0x157()
  fffffe8000c6df10 sys_syscall32+0x101()
stack pointer for thread ffffffff866cb720: fffffe8000c73cf0
[ fffffe8000c73cf0 _resume_from_idle+0xde() ]
  fffffe8000c73d30 swtch+0x185()
  fffffe8000c73db0 cv_timedwait_sig+0x1a3()
  fffffe8000c73e30 cv_waituntil_sig+0xab()
  fffffe8000c73ec0 nanosleep+0x141()
  fffffe8000c73f10 sys_syscall32+0x101()
...
```

It appears that the first few threads on the process are blocked in the `pause()`, `door()`, and `nanosleep()` system calls. We'll double-check against these later when we traverse the user stacks.

3.5.4 Extracting the User-Mode Stack Frame Pointers

The next things to find are the stack pointers for the user threads, which are stored in each thread's `lwp`.

```
> ffffffff866f1878::walk thread |::print kthread_t t_lwp->lwp_regs|::print "struct
regs" r_rsp |=X
             8047d54        fecc9f80       febbac08      fea9df78       fe99df78
fe89df78        fe79df78
             fe69df78       fe59df78       fe49df78      fe39df58       fe29df58
fe19df58        fe09df58
             fdf9df58       fde9df58       fdd9df58      fdc9df58       fdb9df58
fda9df58        fd99df58
                fd89d538        fd79bc08
```

Each entry is a thread's stack pointer in the user process's address space. We can use these to traverse the stack in the user process's context.

3.5.5 Switching MDB to Debug a Specific Process

An `mdb` command, `<proc address>::context`, switches a context to a specified user process.

```
> ffffffff866f1878::context
debugger context set to proc ffffffff866f1878
```

After the context is switched, several `mdb` commands return process information rather than kernel information. For example:

```
> ::nm
Value              Size               Type  Bind  Other Shndx    Name
0x0000000000000000|0x0000000000000000|NOTY |LOCL |0x0  |UNDEF   |
0x0000000008056c29|0x0000000000000076|FUNC |GLOB |0x0  |10      |gethost_revalidate
0x0000000008056ad2|0x0000000000000024|FUNC |GLOB |0x0  |10      |getgr_uid_reaper
0x000000000805be5f|0x0000000000000000|OBJT |GLOB |0x0  |14      |_etext
0x0000000008052778|0x0000000000000000|FUNC |GLOB |0x0  |UNDEF   |strncpy
0x0000000008052788|0x0000000000000000|FUNC |GLOB |0x0  |UNDEF   |_uncached_getgrnam_r
0x000000000805b364|0x000000000000001b|FUNC |GLOB |0x0  |12      |_fini
0x0000000008058f54|0x0000000000000480|FUNC |GLOB |0x0  |10      |nscd_parse
0x0000000008052508|0x0000000000000000|FUNC |GLOB |0x0  |UNDEF   |pause
0x00000000080554e0|0x0000000000000076|FUNC |GLOB |0x0  |10      |getpw_revalidate
...

> ::mappings
            BASE            LIMIT             SIZE NAME
         8046000          8048000             2000 [ anon ]
         8050000          805c000             c000 /usr/sbin/nscd
         806c000          806e000             2000 /usr/sbin/nscd
         806e000          80f0000            82000 [ anon ]
        fd650000         fd655000             5000 /lib/nss_files.so.1
        fd665000         fd666000             1000 /lib/nss_files.so.1
        fd680000         fd690000            10000 [ anon ]
        fd6a0000         fd79e000            fe000 [ anon ]
        fd7a0000         fd89e000            fe000 [ anon ]
...
```

3.5.6 Constructing the Process Stack

Unlike examining the kernel, where we would ordinarily use the stack-related `mdb` commands like `::stack` or `::findstack`, we need to use stack pointers to traverse a process stack. In this case, `nscd` is an x86 32-bit application. So a "stack pointer + 0x38" and a "stack pointer + 0x3c" shows the stack pointer and the program counter of the previous frame.

```
/*
 * In the Intel world, a stack frame looks like this:
 *
 * %fp0->|                               |
 *       |-------------------------------|
 *       |  Args to next subroutine      |
 *       |-------------------------------|-\
 * %sp0->|  One-word struct-ret address  | |
 *       |-------------------------------|  > minimum stack frame
 * %fp1->|  Previous frame pointer (%fp0)| |
 *       |-------------------------------|-/
 *       |  Local variables              |
 * %sp1->|-------------------------------|
 *
 * For amd64, the minimum stack frame is 16 bytes and the frame pointer must
 * be 16-byte aligned.
 */

struct frame {
        greg_t  fr_savfp;               /* saved frame pointer */
        greg_t  fr_savpc;               /* saved program counter */
};

#ifdef _SYSCALL32

/*
 * Kernel's view of a 32-bit stack frame.
 */
struct frame32 {
        greg32_t fr_savfp;              /* saved frame pointer */
        greg32_t fr_savpc;              /* saved program counter */
};
                                                               See sys/stack.h
```

Each individual stack frame is defined as follows:

```
/*
 * In the x86 world, a stack frame looks like this:
 *
 *              |--------------------------|
 * 4n+8(%ebp) ->| argument word n          |
 *              | ...                      |    (Previous frame)
 *    8(%ebp) ->| argument word 0          |
 *              |--------------------------|--------------------
 *    4(%ebp) ->| return address           |
 *              |--------------------------|
 *    0(%ebp) ->| previous %ebp (optional) |
 *              |--------------------------|
 *   -4(%ebp) ->| unspecified              |    (Current frame)
 *              | ...                      |
 *    0(%esp) ->| variable size            |
 *              |--------------------------|
 */
                                                               See sys/stack.h
```

We can explore the stack frames from "Extracting the User-Mode Stack Frame Pointers" on page 174.

```
> ffffffff866f1878::walk thread |::print kthread_t t_lwp->lwp_regs|::print "struct
regs" r_rsp |=X
             8047d54        fecc9f80       febbac08       fea9df78       fe99df78
fe89df78        fe79df78
             fe69df78       fe59df78       fe49df78       fe39df58       fe29df58
fe19df58        fe09df58
             fdf9df58       fde9df58       fdd9df58       fdc9df58       fdb9df58
fda9df58        fd99df58
                fd89d538        fd79bc08
```

```
> 8047d54/X
0x8047d54:      fedac74f
> fedac74f/
libc.so.1`pause+0x67:          8e89c933        = xorl   %ecx,%ecx

> febbac08/X
0xfebbac08:     feda83ec
> feda83ec/
libc.so.1`_door_return+0xac:   eb14c483        = addl   $0x14,%esp

> fea9df78/X
0xfea9df78:     fedabe4c
> fedabe4c/
libc.so.1`_sleep+0x88:         8908c483        = addl   $0x8,%esp
```

Thus, we observe user stacks of `pause()`, `door_return()`, and `sleep()`, as we expected.

3.5.7 Examining the Process Memory

In the process context, we can examine process memory as usual. For example:

```
> libc.so.1`_sleep+0x88::dis
libc.so.1`_sleep+0x67:          pushq  $-0x13
libc.so.1`_sleep+0x69:          call   -0x5cb59 <0xfed4f2d4>
libc.so.1`_sleep+0x6e:          addl   $0x4,%esp
libc.so.1`_sleep+0x71:          movl   %esp,%eax
libc.so.1`_sleep+0x73:          movl   %eax,0x22c(%rsi)
libc.so.1`_sleep+0x79:          leal   0x14(%rsp),%eax
libc.so.1`_sleep+0x7d:          pushq  %rax
libc.so.1`_sleep+0x7e:          leal   0x10(%rsp),%eax
libc.so.1`_sleep+0x82:          pushq  %rax
libc.so.1`_sleep+0x83:          call   +0xc419  <0xfedb8260>
libc.so.1`_sleep+0x88:          addl   $0x8,%esp
libc.so.1`_sleep+0x8b:          movl   %edi,0x22c(%rsi)
libc.so.1`_sleep+0x91:          movb   0xb3(%rsi),%cl
libc.so.1`_sleep+0x97:          movb   %cl,0xb2(%rsi)
libc.so.1`_sleep+0x9d:          jmp    +0x14    <libc.so.1`_sleep+0xb1>
libc.so.1`_sleep+0x9f:          leal   0x14(%rsp),%eax
libc.so.1`_sleep+0xa3:          pushq  %rax
libc.so.1`_sleep+0xa4:          leal   0x10(%rsp),%eax
libc.so.1`_sleep+0xa8:          pushq  %rax
libc.so.1`_sleep+0xa9:          call   +0xc3f3  <0xfedb8260>
libc.so.1`_sleep+0xae:          addl   $0x8,%esp
```

3.6 kmdb, the Kernel Modular Debugger

The userland debugger, mdb, debugs the running kernel and kernel crash dumps. It can also control and debug live user processes as well as user core dumps. kmdb extends the debugger's functionality to include instruction-level execution control of the kernel. mdb, by contrast, can only observe the running kernel.

The goal for kmdb is to bring the advanced debugging functionality of mdb, to the maximum extent practicable, to in-situ kernel debugging. This includes loadable-debugger module support, debugger commands, ability to process symbolic debugging information, and the various other features that make mdb so powerful.

kmdb is often compared with tracing tools like DTrace. DTrace is designed for tracing in the large—for safely examining kernel and user process execution at a function level, with minimal impact upon the running system. kmdb, on the other hand, grabs the system by the throat, stopping it in its tracks. It then allows for micro-level (per-instruction) analysis, allowing users observe the execution of individual instructions and allowing them to observe and change processor state. Whereas DTrace spends a great deal of energy trying to be safe, kmdb scoffs at safety, letting developers wreak unpleasantness upon the machine in furtherance of the debugging of their code.

3.6.1 Diagnosing With kmdb and moddebug

Diagnosing problems with kmdb builds on the techniques used with mdb. In this section, we cover some basic examples of how to use kmdb to boot the system.

3.6.1.1 Starting kmdb From the Console

kmdb can be started from the command line of the console login with mdb and the -K option.

```
# mdb -K

Welcome to kmdb
Loaded modules: [ audiosup cpc uppc ptm ufs unix zfs krtld s1394 sppp nca lofs
genunix ip logindmux usba specfs pcplusmp nfs md random sctp ]
[0]> $c
kmdbmod`kaif_enter+8()
kdi_dvec_enter+0x13()
kmdbmod`kctl_modload_activate+0x112(0, fffffe85ad938000, 1)
kmdb`kdrv_activate+0xfa(4c6450)
kmdb`kdrv_ioctl+0x32(ab00000000, db0001, 4c6450, 202001, ffffffff8b483570,
fffffe8000c48edc)
cdev_ioctl+0x55(ab00000000, db0001, 4c6450, 202001, ffffffff8b483570,
fffffe8000c48edc)
specfs`spec_ioctl+0x99(ffffffffbc4cc880, db0001, 4c6450, 202001,
ffffffff8b483570, fffffe8000c48edc)
fop_ioctl+0x2d(ffffffffbc4cc880, db0001, 4c6450, 202001, ffffffff8b483570,
fffffe8000c48edc)
ioctl+0x180(4, db0001, 4c6450)
sys_syscall+0x17b()
[0]> :c
```

3.6.2 Booting With the Kernel Debugger

If you experience hangs or panics during Solaris boot, whether during installation or after you've already installed, using the kernel debugger can be a big help in collecting the first set of “what happened” information.

You invoke the kernel debugger by supplying the -k switch in the kernel boot arguments. So a common request from a kernel engineer starting to examine a problem is often “try booting with kmdb.”

Sometimes it’s useful either to set a breakpoint to pause the kernel startup and examine something, or to just set a kernel variable to enable or disable a feature or to enable debugging output. If you use -k to invoke kmdb but also supply the -d switch, the debugger will be entered before the kernel really starts to do anything of consequence, so you can set kernel variables or breakpoints.

To enter the debugger at boot with Solaris 10, enter `b -kd` at the appropriate prompt; this is slightly different whether you're installing or booting an already installed system.

```
ok boot kmdb -d
Loading kmdb...

Welcome to kmdb
[0]>
```

If, instead, you're doing this with a system where GRUB boots Solaris, you add the `-kd` to the “kernel” line in the GRUB menu entry (you can edit GRUB menu entries for this boot by using the GRUB menu interface, and the “e” (for edit) key).

Either way, you'll drop into the kernel debugger in short order, which will announce itself with this prompt:

```
[0]>
```

Now we're in the kernel debugger. The number in square brackets is the CPU that is running the kernel debugger; that number might change for later entries into the debugger.

3.6.3 Investigating Hangs

For investigating hangs, try turning on module debugging output. You can set the value of a kernel variable by using the `/W` command (“write a 32-bit value”). Here's how you set `moddebug` to 0x80000000 and then continue execution of the kernel.

```
[0]> moddebug/W 80000000
[0]> :c
```

This command gives you debug output for each kernel module that loads. The bit masks for moddebug are shown below. Often, 0x80000000 is sufficient for the majority of initial exploratory debugging.

```
/*
 * bit definitions for moddebug.
 */
#define MODDEBUG_LOADMSG        0x80000000      /* print "[un]loading..." msg */
#define MODDEBUG_ERRMSG         0x40000000      /* print detailed error msgs */
#define MODDEBUG_LOADMSG2       0x20000000      /* print 2nd level msgs */
#define MODDEBUG_FINI_EBUSY     0x00020000      /* pretend fini returns EBUSY */
#define MODDEBUG_NOAUL_IPP      0x00010000      /* no Autounloading ipp mods */
#define MODDEBUG_NOAUL_DACF     0x00008000      /* no Autounloading dacf mods */
#define MODDEBUG_KEEPTEXT       0x00004000      /* keep text after unloading */
#define MODDEBUG_NOAUL_DRV      0x00001000      /* no Autounloading Drivers */
#define MODDEBUG_NOAUL_EXEC     0x00000800      /* no Autounloading Execs */
#define MODDEBUG_NOAUL_FS       0x00000400      /* no Autounloading File sys */
#define MODDEBUG_NOAUL_MISC     0x00000200      /* no Autounloading misc */
#define MODDEBUG_NOAUL_SCHED    0x00000100      /* no Autounloading scheds */
#define MODDEBUG_NOAUL_STR      0x00000080      /* no Autounloading streams */
#define MODDEBUG_NOAUL_SYS      0x00000040      /* no Autounloading syscalls */
#define MODDEBUG_NOCTF          0x00000020      /* do not load CTF debug data */
#define MODDEBUG_NOAUTOUNLOAD   0x00000010      /* no autounloading at all */
#define MODDEBUG_DDI_MOD        0x00000008      /* ddi_mod{open,sym,close} */
#define MODDEBUG_MP_MATCH       0x00000004      /* dev_minorperm */
#define MODDEBUG_MINORPERM      0x00000002      /* minor perm modctls */
#define MODDEBUG_USERDEBUG      0x00000001      /* bpt after init_module() */

                                                                See sys/modctl.h
```

3.6.4 Collecting Information About Panics

When the kernel panics, it drops into the debugger and prints some interesting information; usually, however, the most interesting thing is the stack backtrace; this shows, in reverse order, all the functions that were active at the time of panic. To generate a stack backtrace, use

```
[0]> $c
```

A few other useful information commands during a panic are ::msgbuf and ::status, as shown in Section 3.4 , "Debugging Kernel Cores," on page 154.

```
[0]> ::msgbuf   - which will show you the last things the kernel printed onscreen, and
[0]> ::status   - which shows a summary of the state of the machine in panic.
```

If you're running the kernel while the kernel debugger is active and you experience a hang, you may be able to break into the debugger to examine the system state; you can do this by pressing the <F1> and <A> keys at the same time (a sort of "F1-shifted-A" keypress). (On SPARC systems, this key sequence is <Stop>-<A>.) This should give you the same debugger prompt as above, although on a multi-CPU system you may see that the CPU number in the prompt is something other than 0. Once in the kernel debugger, you can get a stack backtrace as above; you can also use `::switch` to change the CPU and get stack backtraces on the different CPU, which might shed more light on the hang. For instance, if you break into the debugger on CPU 1, you could switch to CPU 0 with

```
[1]> 0::switch
```

3.6.5 Working With Debugging Targets

For the most part, the execution control facilities provided by `kmdb` for the kernel mirror those provided by the `mdb` process target. Breakpoints (`:bp`), watchpoints (`::wp`), `::continue`, and the various flavors of `::step` can be used.

We discuss more about debugging targets in Section 3.3 , “Getting Started with MDB” and Section 3.4 , “Debugging Kernel Cores”. The common commands for controlling `kmdb` targets are summarized in Table 3.14.

Table 3.14 Core kmdb dcmds

dcmd	Description
`::status`	Print summary of current target.
`$r` `::regs`	Display current register values for target.

`$c`
`::stack`
`$C`
	Print current stack trace ($C: with frame pointers).

`addr[,b]`
`::dump [-g sz] [-e]`
	Dump at least b bytes starting at address `addr`. `-g` sets the group size; for 64-bit debugging, `-g 8` is useful.

`addr::dis`	Disassemble text, starting around `addr`.

Table 3.14 Core kmdb dcmds (*continued*)

dcmd	Description

`[ addr ] :b`
`[ addr ] ::bp [+/-dDestT] [-n count] sym ...  addr`
	Set breakpoint at addr.

`$b`	Display all breakpoints.
`::branches`	Display the last branches taken by the CPU. (x86 only)

`addr ::delete [id | all]`
`addr :d [id | all]`
	Delete a breakpoint at addr.

`:z`	Delete all breakpoints.
`function ::call [arg [arg ...]]`	Call the specified function, using the specified arguments.
`[cpuid] ::cpuregs [-c cpuid]`	Display the current general-purpose register set.
`[cpuid] ::cpustack [-c cpuid]`	Print a C stack backtrace for the specified CPU.
`::cont` `:c`	Continue the target program.
`$M`	List the macro files that are cached by kmdb for use with the $< dcmd
`::next` `:e`	Step the target program one instruction, but step over subroutine calls.
`::step [branch | over | out]`	Step the target program one instruction.
`$<systemdump`	Initiate a panic/dump.

`::quit [-u]`
`$q`
	Cause the debugger to exit. When the -u option is used, the system is resumed and the debugger is unloaded.

`addr [,len]::wp [+/-dDestT] [-rwx] [-ip] [-n count]`
 	
`addr [,len]:a [cmd ...]`
`addr [,len]:p [cmd ...]`
`addr [,len]:w [cmd ...]`
	Set a watchpoint at the specified address.

3.6.6 Setting Breakpoints

Setting breakpoints with kmdb is done in the same way as with generic mdb targets, using the :b dcmd. Refer to Table 3.13 for a complete list of debugger dcmds.

```
# mdb -K
Loaded modules: [ crypto ]
kmdb: target stopped at:
kmdbmod`kaif_enter+8:   popfq
[0]> resume:b
[0]> :c
kmdb: stop at resume
kmdb: target stopped at:
resume:         movq   %gs:0x18,%rax
[0]> :z
[0]> :c
#
```

3.6.7 Forcing a Crash Dump With halt -d

This example shows how to force a crash dump and reboot of the x86-based system by using the halt -d and boot commands. Use this method to force a crash dump of the system. Afterwards, reboot the system manually.

```
# halt -d
4ay 30 15:35:15 wacked.Central.Sun.COM halt: halted by user

panic[cpu0]/thread=ffffffff83246ec0: forced crash dump initiated at user request

fffffe80006bbd60 genunix:kadmin+4c1 ()
fffffe80006bbec0 genunix:uadmin+93 ()
fffffe80006bbf10 unix:sys_syscall32+101 ()

syncing file systems... done
dumping to /dev/dsk/c1t0d0s1, offset 107675648, content: kernel
NOTICE: adpu320: bus reset
100% done: 38438 pages dumped, compression ratio 4.29, dump succeeded

Welcome to kmdb
Loaded modules: [ audiosup crypto ufs unix krtld s1394 sppp nca uhci lofs
genunix ip usba specfs nfs md random sctp ]
[0]>
kmdb: Do you really want to reboot? (y/n) y
```

3.6.8 Forcing a Dump With kmdb

If you cannot use the reboot -d or the halt -d command, you can use the kernel debugger, kmdb, to force a crash dump. The kernel debugger must have been loaded, either at boot or with the mdb -k command, for the following procedure to work. Enter kmdb by using L1–A on SPARC, F1-A on x86, or break on a tty.

```
[0]> $<systemdump
panic[cpu0]/thread=ffffffff83246ec0: forced crash dump initiated at user request

fffffe80006bbd60 genunix:kadmin+4c1 ()
fffffe80006bbec0 genunix:uadmin+93 ()
fffffe80006bbf10 unix:sys_syscall32+101 ()

syncing file systems... done
dumping to /dev/dsk/c1t0d0s1, offset 107675648, content: kernel
NOTICE: adpu320: bus reset
100% done: 38438 pages dumped, compression ratio 4.29, dump succeeded
```

3.7 kmdb Implementation

The best way to understand kmdb is by first understanding how mdb does things. We begin with an overview of the portions of mdb that are relevant to our later discussion of kmdb. For more information about mdb and its operation, consult the Modular Debugger AnswerBook. Having set the stage, we next discuss the major design goals behind kmdb. With those goals in mind, we return to the list of components we discussed from an mdb perspective, analyzing them this time from the point of view of kmdb, showing how their implementation fulfills kmdb's design goals. Finally, we embark on a whirlwind tour of some of the lower-level components of kmdb that weren't described in earlier sections.

3.7.1 MDB Components and Their Implementation in MDB

In this section, we review the parts of MDB that are particularly relevant for our later discussion of kmdb, focusing on how those components are implemented in mdb. That is, we concentrate only on those components whose implementation changes significantly in kmdb. The design of MDB is sufficiently modular that we could replace the components requiring change without disrupting the remainder of the debugger. The components described are shown in Figure 3.2.

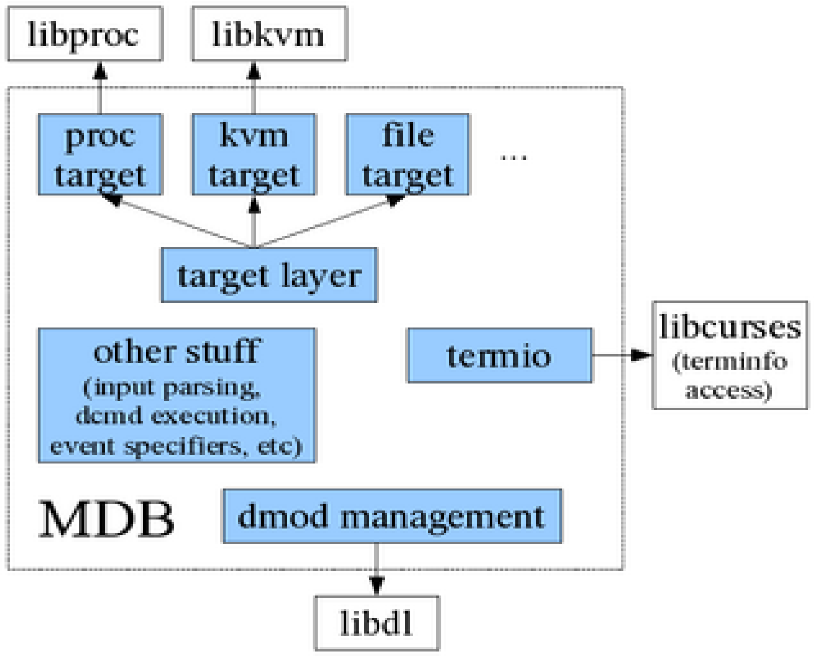

libproc
libkvm
proc target
kvm target
file target
...
target layer
other stuff (input parsing, dcmd execution, event specifiers, etc)
termio
libcurses (terminfo access)
MDB
dmod management
libdl

Figure 3.2 MDB Components

3.7.1.1 The Target Layer

The MDB answerbook describes targets as follows:

> *The target is the program being inspected by the debugger. [...] Each target exports a standard set of properties, including one or more address spaces, one or more symbol tables, a set of load objects, and a set of threads.*

Targets are implemented by means of an ops vector, with each target implementing a subset of the functions in the vector. In-situ targets, such as the user process or `proc`, implement virtually all operations. Targets that debug entities whose execution cannot be controlled, such as the kvm target used for crash dump analysis, implement a smaller subset of the operations. As with many other parts of MDB, the targets are modular and are designed to be easily replaceable depending on the requirements of the debugging environment.

Figure 3.2 shows three of the targets used by MDB. The first is the `proc` target, which is used for the debugging and control of user processes as well as the analysis of user core dumps. The `proc` target is implemented on top of `libproc`, which provides the primitives used for process control. The interfaces provided by `libproc` simplify the implementation of the `proc` target by hiding the differences between in-situ and postmortem debugging (one is done with a live process, whereas the other uses a corefile). The target itself is largely concerned with mapping the requests of the debugger to the interfaces exposed by `libproc`.

Also shown in Figure 3.2 is the `kvm` target, which is used for both live and postmortem kernel debugging. Like the `proc` target, the `kvm` target uses a support library (`libkvm`) to abstract the differences between live and postmortem debugging. While the capabilities of the `kvm` and `proc` targets are largely the same when used for postmortem debugging, they differ when the subjects are live. The `proc` target fully controls process execution, whereas the `kvm` target allows only the inspection and alteration of kernel state. Allowing the debugger to control the execution of the kernel that is responsible for running the debugger would be difficult at best. Consequently, most debugging done with the `kvm` target is of the postmortem variety.

The third target shown in Figure 3.2 is used for the “debugging” of raw files. This allows the data-presentation abilities of MDB to be brought to bear upon flat (usually binary) files. This target lays the foundation for the eventual replacement of something like `fsdb`, the filesystem debugger.

3.7.1.2 Debugger Module Management

Today’s kernels are made up of a great many modules, each implementing a different subsystem and each requiring different tools for analysis and debugging. The same can be said for modern, large-scale user processes, which can incorporate tens or even hundreds of shared libraries and subsystems. A modern modular debugger should, therefore, allow for the augmentation of its basic tool set as needed. MDB allows subsystem-specific debugging facilities to be provided through shared objects known as debugger modules, or dmods. Each dmod provides debugging commands (also known as dcmds) and walkers (iterators) that debug a given subsystem. These modules interface with MDB through the module API layer and use well-defined interfaces for data retrieval and analysis. This is enforced by the fact that, in the case of both major targets (`kvm` and `proc`), the debugger runs in a separate address space from the entity being analyzed. The dcmds are therefore forced to use the module API to access the target. While some dmods link with other support libraries to reduce the duplication of code, most dmods stand alone, consuming only the header files from the subsystems they support.

While the core debugger uses its own code for the management of debugger modules and their metadata, it relies upon a system library, libdl, for the mechanics of module unloading and unloading. It is libdl, for example, that knows how to load the dmod into memory, and it is libdl that knows how to integrate that dmod into the debugger's address space.

3.7.1.3 Terminal I/O

MDB was designed with an eye toward the eventual implementation of something like kmdb and thus performs most terminal interaction directly. Having built up a list of terminal attributes, MDB handles cursor and character manipulation directly. The MDB subsystem that performs terminal I/O is known as termio.

While termio handles a great deal itself, there is one aspect of terminal management that is provided by a support library. MDB uses libcurses to retrieve the list of terminal attributes for the current terminal from the terminfo database. The current terminal type is retrieved from the environment variable TERM.

3.7.1.4 Other Stuff

MDB is a large program, with many more subsystems than are described here. One of the benefits arising from the modular design of the debugger is that these other subsystems don't need to change even when used in an environment as radically different as kmdb is from MDB. For example, MDB implements its own routines for the management of ELF symbol tables. ELF being ELF regardless of source, the same subsystem can be used, as is, in both MDB and kmdb. A description of the MDB subsystems unaffected by kmdb is beyond the scope of this document.

3.7.2 Major kmdb Design Decisions

In this section we explore the major design rationale.

3.7.2.1 The Kernel/Debugger Interface (KDI)

When we implement an in-situ kernel debugger, we must determine the extent to which the debugger will be intermingled with the kernel being debugged. Should the debugger call kernel functions to accomplish its duties, or should the debugger be entirely self-contained? The legacy Solaris in-situ kernel debugger, kadb, hewed to the latter philosophy to a significant extent. The kadb module was as self-contained as possible, to the point where it contained copies of certain low-level kernel routines. That said, there were some kernel routines to which kadb needed access. During debugger startup, it would search for a number of functions by name, saving pointers to them for later use.

There are a number of problems with kadb's approach. First of all, by duplicating low-level kernel code in the debugger, we introduce duplication. Furthermore, this duplication, due to the layout of the Solaris source code, results in the copies being significantly separated. It's hard enough to maintain code rife with duplication when the duplicates are co-located. Maintaining duplicates located in wildly disparate locations is next to impossible. During initial analysis of kadb as part of the kmdb project, we discovered several duplicated functions in kadb that had not kept up with hardware-specific changes to the versions in the kernel. The second problem concerns the means by which kadb gained access to the kernel functions it did use. Searching for those functions by name is dangerous because it leaves the debugger vulnerable to changes in the kernel. A change in the signature of a kernel function used by kadb, for example, would not be caught until kadb failed while trying to use said function.

To some extent, the nature of a kernel debugger requires duplication. The kernel debugger cannot, for example, hold locks, and therefore requires lock-free versions of any kernel code that it must call. The lock-free version of a function may not be safe when used in a running kernel context and therefore must be kept separate from the normal version. Rather than placing that duplicate copy within the debugger itself, we decided to co-locate the duplicate with the original. This reduces the chances of code rot, since an engineer changing the normal version is much more likely to notice the debugger-specific version sitting right next to it.

Access to kernel functionality was formalized through an interface known as the KDI, or Kernel/Debugger Interface. The KDI is an ops vector through which all kernel function calls must pass. Each function called by the debugger has a member in this vector. Whereas an assessment of kernel functionality used by kadb required a search for symbol lookup routines and their consumers, a similar assessment in kmdb simply requires the review of the single ops vector. Furthermore, our use of an ops vector allowed us to use the compiler to monitor the evolution of kernel functions used by kmdb. Any change to a KDI function significant enough to change the function signature will be caught by the compiler during the initialization of the KDI ops vector. Furthermore, the initialization of said vector is easily visible to code analysis tools such as cscope, allowing engineers to quickly determine whether kmdb is a consumer of a given function. With kadb, such a check would require a check of the symbol lookup routines, something that is not automatically done by the code analysis tools used today.

3.7.2.2 Implementation As a Kernel Module

kadb was implemented as a stand-alone module. In Solaris, this means that the kadb module was an executable, directly loadable by the boot loader. It had no static dependencies on other modules, thus leading to the symbol lookup problems discussed above. When the use of kadb was requested, the boot process ran something like this:

1. Boot loader loads kadb.
2. kadb initializes.
3. kadb loads normal stand-alone, UNIX.
4. kadb loads the UNIX interpreter, krtld.
5. kadb passes control to krtld.
6. krtld loads the UNIX dependencies (genunix, CPU module, platform module, etc.).
7. krtld transfers control to UNIX.

While this allowed the debugger to take early control of the system (it could debug from the first instruction in krtld), that ability came with some significant penalties. The decision to load a 32-bit or 64-bit kernel being made after kadb had loaded and initialized, kadb had to be prepared to debug either variety. The need for kadb to execute prior to the loading of UNIX itself meant that it could not use any functions located in the kernel until the kernel was loaded. While some essential functions were dynamically located later, the result of this restriction was the location of many low-level kernel functions in the debugger itself. A further penalty comes in the form of increased debugger complexity. kadb's need to load UNIX and krtld requires that it know how to process ELF files and how to load modules into the address space. The boot loader already needs to know how to do that, as does krtld. With kadb as a stand-alone module, the number of separate copies of ELF-processing and module-loading code goes up to three.

The remaining limitations have to do with the timing of the decision to load kadb. As stated above, kadb was a stand-alone module and as such could only be loaded at boot. Moreover, an administrator was required to decide, before rebooting, whether to load kadb. Once loaded, it could not be unloaded. While the inability to unload the debugger isn't a major limitation, the inability to dynamically load it, is. Not knowing whether kadb would be needed during the life of a given system boot, administrators would be faced with an unfortunate choice. On the one hand, they could always load kadb at boot. This kept it always ready for use, but at the cost of the wiring down of a chunk of kernel address space. This could be avoided, of course, by making the other choice—not loading the debugger at boot. Administrators then ran the risk of not having the debugger around when they needed it.

The implementation of kmdb as a normal kernel module solves all of these problems, with only a minor activation-time penalty compared to kadb. When kmdb is loaded at boot, the boot process looks something like this:

1. Boot loader loads UNIX.
2. Boot loader loads the UNIX interpreter, krtld.
3. Boot loader passes control to krtld.
4. krtld loads the UNIX dependencies (genunix, CPU module, platform module, etc.).
5. krtld loads kmdb.
6. krtld transfers control to UNIX.

As shown above, kmdb loads after the primary kernel modules have been selected and loaded. kmdb can therefore assume that it will be running with the same bit width as that of the underlying kernel. That is, a 32-bit kmdb will never have to deal with a 64-bit kernel, and vice versa.

By loading after the primaries, kmdb can have static symbol dependencies on the other primary kernel modules. It is this ability that allows the KDI to exist. Even better, kmdb can rely on krtld's selection of the proper CPU and platform modules for this machine. Rather than having to carry around several processor-specific implementations of the same function (or compiling one module for each of four platform types, as kadb did), kmdb can, using the KDI, simply use the proper implementation of a given function from the proper module. When a new platform-specific KDI function is implemented, the developer implements it in a platform-specific way in each platform module. krtld selects the proper platform module on boot, and kmdb automatically ends up using the proper version for the host machine.

Last but certainly not least, the implementation of kmdb as a normal kernel module allows it to be dynamically loaded and unloaded. It can still be loaded at boot, but it can also be loaded on-demand by the administrator. If dynamically loaded, it can also be unloaded when no longer needed. This can be a consolation to wary administrators who would otherwise object to the running of a kernel debugger on certain types of machines.

The only disadvantage of the use of a normal kernel module versus a standalone one is the loss of the ability to debug the early stages of krtld. In practice, this has not turned out to be a problem, because the early stages of krtld are fairly straightforward and stable.

Every attempt has been made to minimize the effects of the two load types (boot and runtime). Obviously initialization differs in some respects, a number of common kernel subsystems simply won't be available during the initialization of boot-loaded kmdb. Largely, though, these differences are dealt with under the covers and are not visible to the user.

3.7.3 The Structure of kmdb

We can best understand the inner workings of kmdb by first reviewing the debugger's external structure. kmdb's external structure is dictated, to some extent, by the environments in which it will be used. Those requirements are

- The debugger must be loadable at boot.
- The debugger must be loadable at runtime.
- The debugger must restrict its contact with the running kernel to a set of operations defined in advance.

To satisfy the first two requirements, kmdb exists as two separate kernel modules. The first, misc/kmdbmod, contains the meat of the debugger; it is the module loaded by krtld when kmdb is loaded at boot. The second module, drv/kmdb, exists solely to gather property values from the device tree and to present an ioctl-based interface to controlling userland programs such as mdb(1). When kmdb is to be loaded at runtime, mdb opens /dev/kmdb and uses the ioctl interface to command it to activate. The opening of /dev/kmdb causes drv/kmdb to load. drv/kmdb has a dependency on misc/kmdbmod, which gets loaded as well. Upon receipt of the appropriate ioctl, drv/kmdb calls into misc/kmdbmod, and the debugger is initialized.

If the debugger was loaded at boot, only misc/kmdbmod will be loaded. The module loading subsystem is not fully initialized at that point. Userland does not exist yet, and given that drv/kmdb exists only to convey ioctl requests from userland to misc/kmdbmod, there is no need to force drv/kmdb to load until an attempt is made to open /dev/kmdb. When someone does attempt to control the debugger through ioctls to /dev/kmdb, drv/kmdb is loaded. It then sends commands to misc/kmdbmod as in the runtime case above.

We now focus our attention more closely on misc/kmdbmod, which itself is composed of two parts. The first, referred to as the debugger, contains the core debugger functionality, as well as the primary subsystems needed to allow the core to control the kernel. The second, referred to as the controller, interacts with the running kernel.

The debugger interacts with the outside world only through a set of well-defined interfaces. One of these is the KDI; the other is composed of a set of functions passed during initialization by the controller. Aside from these interactions, the debugger must, by nature, function as a fully self-contained entity. Put in compilation terms, the debugger, which is built separately from the controller, must not have any unresolved symbols at link time. It is the debugger, and only the debugger, that is active when kmdb has control of the machine.

Behind the scenes, as it were, the controller works to ensure that the debugger's runtime needs are met. The debugger has a limited set of direct interactions with the kernel. And it can only be active when the world has stopped. Those two facts necessarily limit the sorts of things the debugger can do. For example, it can neither perform the early stages of kmdb initialization nor load or unload kernel modules.

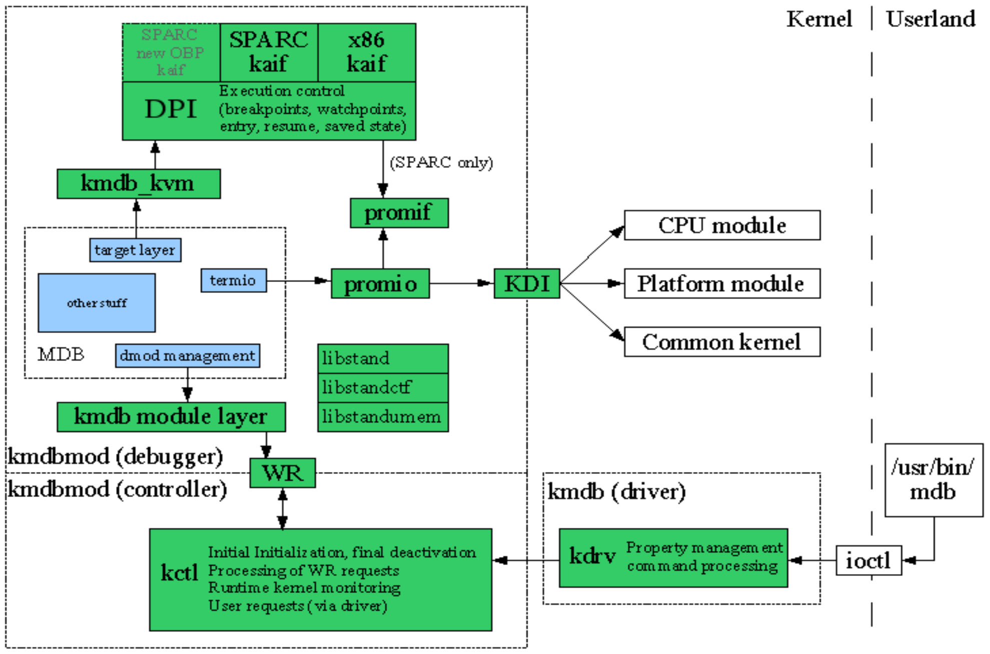


Figure 3.3 KMDB Structure

The former takes place before debugger initialization starts and is taken care of by the controller. A memory region, known as Oz, is allocated and is set aside for use by the debugger. Other initialization tasks performed by the controller include the creation of trap tables or IDTs, as appropriate, after which control is passed to the debugger for the completion of initialization.

Kernel module loading and unloading, which is discussed in more detail below, is a task that must be performed by the running kernel. The debugger must rely on the controller to perform these sorts of tasks for it.

In the text that follows, we use the words driver, debugger, and controller to refer to the components we've just discussed. These three components are indicated in Figure 3.3 by regions surrounded by dotted lines. When we discuss the entire entity, we refer to it as kmdb. References to the core debugger refer to the set of blue boxes labeled MDB. One unfortunate note: The term "controller" is a relatively recent invention. In many instances, the source code refers to the driver when it means the controller. This doesn't cause nearly as many issues as one might imagine because of the minor role played by the entity we refer to as the driver.

3.7.4 MDB Components and Their Implementation in kmdb

We now use our earlier discussion of mdb to motivate our review of the major subsystems used by kmdb. Recall that the three subsystems discussed were the target layer, module management, and terminal management (termio). The implementation of kmdb is largely the story of the replacement of support libraries with subsystems designed to work in kmdb's unique environment. Figure 3.3 shows how these replacement subsystems relate to the core debugger.

3.7.4.1 The Target Layer

The target layer itself is unchanged in kmdb. What changes is the target implementation itself. Gone are the proc, kvm, and file targets, replaced with a single target called kmdb_kvm. We continue to call it kmdb_kvm to avoid confusion with the kvm target used by mdb.

kmdb_kvm can be thought of as a hybrid of the proc and kvm targets. It includes the execution control aspects of proc, such as the ability to set breakpoints and watchpoints, as well as support for single-stepping, continuation, and so forth. This functionality is coupled with the kernel-oriented aspects of the kvm target. The kmdb_kvm target is common between SPARC and x86 machines and for the most part handles the bits of kernel analysis, management, and control that are generic to the two architectures. With the exceptions of stack trace construction and the display of saved registers, all architecture-specific functionality is abstracted into the DPI. The DPI's relationship to kmdb_kvm is very similar to that of libkvm to the kvm target or to that of libproc to the proc target.

A significant portion of kmdb_kvm is devoted to the monitoring of kernel state. As an example, target implementations are required to provide symbol lookup routines for use by the core debugger. Provision of this information requires access to kernel module symbol tables, which are easily accessed by kmdb_kvm. What is not so simple, however, is dealing with the constant churn in the set of loaded modules. Whenever kmdb regains control of the machine, kmdb_kvm scans the entire module list, looking for modules that have loaded or unloaded. The tracking state (symbol table references, and so forth) of kmdb_kvm modules that have unloaded is destroyed, while new state is created for modules that have been loaded. Challenges arise when a module has unloaded and then reloaded since kmdb last had control. This churn must be detected, and tracking state rebuilt.

The tracking of module movement, for lack of a better term, illustrates the interaction between the debugger and the controller. While the debugger could certainly rescan the entire list upon every entry, that approach would be wasteful. Instead, the controller subscribes to the kernel's module change notification service and bumps a counter whenever a change has occurred. kmdb_kvm can, upon reentry, check the value of that counter. If the value has changed since kmdb_kvm last saw it, a module list rescan is necessary.

While this interaction with the controller results in a useful optimization for module state management, it becomes crucial for the management of deferred breakpoints. Deferred breakpoints are breakpoints requested for modules that haven't yet loaded. The user's expectation is that the breakpoint will activate when the named module loads. The debugger is responsible for the creation, deletion, enabling, disabling, activation, and deactivation of breakpoints. The user creates the breakpoint by using the breakpoint command (::bp). This being a deferred breakpoint for a module that hasn't been loaded, the debugger leaves the breakpoint in a disabled state. When that module has loaded, the breakpoint is enabled. Enabled breakpoints are activated by the debugger when the world is resumed. The activation is what makes the breakpoint actually happen. In kmdb_kvm, the DPI installs a breakpoint instruction at the specified virtual address. The key design question: How do we detect the loading of the requested module?

The simplest, cleanest, and slowest approach would be to have kmdb_kvm place an internal breakpoint on the kernel's module loading routine. Whenever a module is loaded, the debugger would activate, would check the identity of the loaded module, and would decide whether to enable the breakpoint. Debugger entry isn't cheap. All CPUs must be stopped, and their state must be saved. This particular stop would happen after a module load, so we would need to rescan the module list. All in all, this is something that we really don't want to have to do every time a module is loaded or unloaded.

If we involve the controller, we can eliminate the unnecessary debugger activations, entering the debugger only when a module named in a deferred breakpoint is loaded or unloaded. How do we do this? We bend the boundaries between the debugger and controller slightly, exposing the list of deferred breakpoints to code that runs when the world is turning. Tie this into the controller's registration with the kernel's module change notification service, and we end up entering the debugger only when a change has occurred in a module named in a deferred breakpoint. We use a quasi-lock-free data structure to allow access to the deferred breakpoint list both from within the debugger (when the world is stopped) and within the module change check (when the world is running).

Like the `proc` and `kvm` targets, `kmdb_kvm` is also home to dcmds that could not be implemented elsewhere. Implemented in the target, they have access to everything the target does and can thus do things that dcmds implemented in dmods could only dream of doing. As implied above, `kmdb_kvm` (as well as `kvm` and `proc`) implement dcmds that provide stack tracing and register access.

3.7.4.2 Debugger Module Management

As discussed earlier, `mdb` uses `libdl` for the management of dmods, which are implemented as shared objects. The implementation of `kmdb` is similar, but without `libdl`. Nor does the debugger have the way to actually load or unload modules. Other than that, `kmdb` and `mdb` are the same.

We decompose module management into two pieces: the requesting of module loads and unloads, and the implementation of a `libdl` replacement atop the results of the loading and unloading.

3.7.4.3 Module Loads and Unloads: the Work Request Queue (WR)

`kmdb` implements debugger modules as kernel modules. While we engage in some sleight of hand to keep the dmods off the kernel's main module list, the mechanics of loading and unloading dmods is largely the same as that used for "normal" kernel modules. The primary difference is in the means by which a load or unload is requested. Recall that the debugger, which will receive the load or unload request from the user, can only run when the world is stopped. Also note that the loading or unloading of a kernel module is a process that uses many different kernel subsystems. The kernel runtime linker (`krtld`), the disk driver, VM system, file system, and many others come into play. Use of these subsystems of course entails the use of locks, threads, and various other things that are anathema to the debugger.

To load a dmod, the debugger must therefore ask the controller to do it. The controller runs when the world is turning and is more than capable of loading and unloading kernel modules. The only thing we need is a channel for communication between the two. That channel is provided by the Work Request Queue, or WR. The WR consists of two queues: one for messages from the debugger to the controller and one for messages from the controller to the debugger. The rough sequence of events for a module load is as follows:

1. User requests a dmod load with `::load`.
2. The `kmdb` module layer receives the request and passes it to the WR debugger → controller queue.
3. The world is resumed.

4. The controller receives the request.
5. The controller loads the module.
6. The controller returns the requests to the debugger as a (successful) reply on the controller → debugger queue.
7. The controller initiates a debugger reentry.
8. The debugger receives the reply and makes the contents of the dmod available to the debugger core.

A few details bear mentioning. The debugger can be activated at any time—even in the midst of the controller's processing of a load request. The controller must keep this in mind when checking and manipulating the WR queues. The queues themselves are lock-free and have very strict rules regarding the methods used to access them. For example, the controller may only add to the end of the controller → debugger queue. It sets the next pointer on its request and updates the tail pointer for the queue. Even though the queue is doubly linked, there's no easy way for the controller, which may be interrupted at any time by the debugger, to set the prev pointer. Accordingly, the debugger's first action upon preparing to process the controller → debugger queue is to traverse it, from tail to head, building the `prev` pointers. The debugger doesn't have to worry about being interrupted by the controller and can thus take its time. Similar rules are in place for the debugger → controller queue.

Every request must be tracked and sent back as a reply at some point. Even fire-and-forget requests, such as those establishing new module search paths, must be returned as replies, even if those replies don't come until the debugger is unloaded. To see why this is necessary, consider the source of the memory underlying the requests. Requests from the debugger are allocated from debugger memory by the debugger's allocator and can thus only be freed by the debugger. Requests initiated by the controller (for example, an automatic dmod load triggered by the loading of the corresponding kernel module) are allocated by the controller from kernel memory and can thus be freed only by the kernel. Replies therefore serve a dual purpose—they provide status to the requester and also return the request to the requester for freeing.

We'd like to minimize the impact of the debugger on the running system to the extent practicable and so don't want the controller to poll for updates to the WR queues. Instead, we want the debugger to tell the controller when work is available for processing. This isn't as simple as it may seem. In the real world, we would use semaphores or condition variables to signal the availability of work. To use kernel synchronization objects, the debugger would need to call into the kernel to release them. The kernel is most definitely not prepared for a `cv_broadcast()` call with every CPU stuck in the debugger. Unpleasantness would ensue. The lightest-weight way to communicate with the controller is to post a soft interrupt, the implementation of which is essentially the setting of a bit in the kernel's cpu_t structure. When the world has resumed, normal Normal interrupt processing will encounter this bit and will call the soft interrupt handler registered by the controller. That handler bangs on a semaphore, which triggers the controller's WR processing. Note that these problems apply only for communications from the debugger to the controller. The debugger can simply poll for messages sent in the opposite direction. Since the debugger is activated relatively infrequently, the occasional check of a message-waiting bit doesn't impose a burden. When users request a debugger activation, the last thing on their mind is whether the debugger is wasting a few cycles to check for messages.

libdl supplies a synchronous loading and unloading interface to mdb, thus considerably simplifying its management of dmods. kmdb has no such luxury. As the reader might surmise from the preceding discussion, kmdb's loading and unloading of dmods is decidedly asynchronous. Every attempt is made to preserve the user's illusion of a blocking load, but the asynchronous nature occasionally pokes its head into the open. A breakpoint encountered before the completion of the load, for example, causes an early debugger reentry. The user is told that a load or an unload is still pending and is told how to allow it to complete.

3.7.4.4 libdl Wrapper

MDB's dmod management code uses the libdl interfaces for manipulating dmods. dlopen() loads modules, dlclose() unloads them, and dlsym() looks up symbols. The debugger implements its own versions of these functions (using the same function signatures) to support the illusion of libdl. Underneath, the debugger's symbol table facilities are retargeted to implement dlsym()'s searches of dmod symbol tables.

3.7.4.5 Terminal I/O

To implement terminal I/O handling, we need three things: access to the terminal type, the ability to manipulate that terminal, and routines for actually sending I/O to and from that terminal. The second of these can be further subdivided into the retrieval of terminal characteristics and the use of that knowledge to manipulate the terminal. mdb implements the most difficult of these—the routines that actually manipulate the terminal according to the gathered characteristics. mdb handles the tracking of cursor position, in-line editing, and the implementation of a parser and knows how to use the individual terminal attributes (echo this to make the cursor move right, echo that to enable bold, etc.) to accomplish those tasks.

Left to mdb and kmdb are terminal type determination, attribute retrieval, and I/O to the terminal itself. For mdb, this is relatively straightforward. The terminal type can be gathered from the environment, attributes can be retrieved from the terminfo database with libcurses, and I/O accomplished with stdin, stdout, and stderr.

kmdb, as is its wont, has a more difficult time of things. There is no environment from which to gather the current terminal type. There's no easy access to the terminfo database. Completing the trifecta, the I/O methods vary with the type of platform, progress of the boot process, and phase of the moon. As a bonus, kmdb's termio implementation handles interrupt (^C) processing. We discuss each in turn. While the preceding sections had happy endings, in that pleasing solutions were found for the enumerated problems, the reader is warned that there are no happy endings in terminal management. Tales of wading through terminal types, to say nothing of the terminfo/termcap databases, are generally suitable only for frightening small children and always end in woe and the gnashing of teeth.

3.7.4.6 Retrieving the Terminal Type

At first glance, gaining access to the terminal type would seem straightforward. Sadly, no. kmdb can be loaded at boot or at runtime. It can be used on a locally attached console/framebuffer, or it can be used through a serial console. If loaded at runtime, the invocation could be made from a console login, or it could be made from an rsh (or telnet or ...) session. Boot-loaded kmdb on a serial console is the worst because we have no information regarding the type of terminal attached to the other end of the serial connection. We end up assuming the worst, which is a 80 × 24 VT100. Boot-loaded kmdb on a machine with a locally attached console or framebuffer is easier because we know the terminal type and terminal dimensions for SPARC and x86 consoles. Also easy is a runtime-loaded kmdb from a console login. Assuming that the user set the terminal type correctly, we can use the value of the TERM environment variable. But unfortunately we can't trust $TERM to be set correctly, so we ignore $TERM if the console is locally attached. We end up with a pile of heuristics, which generally come up with the right answer. If they don't, they can always be overridden.

3.7.4.7 Terminal Attributes

After considering the mess that is access to $TERM, retrieval of terminfo data is almost trivial. We don't want to compile in a copy of the terminfo database, and we can't rely on the ability to gain access to it while the debugger is running. We compromise by hard-coding a selection of terminal types into the debugger. The build process extracts the attributes for each selected terminal from the terminfo database and compiles them into the debugger. Terminal type selection in kmdb is thus limited to the types selected during the build. It turns out, though, that the vast majority of common terminal types can be covered by a set of 15 terminal types.

3.7.4.8 Console I/O

Access to the terminal entails the reading of input, the writing of output, and the retrieval of hardware parameters (terminal size and so forth), generally through an ioctl-based interface. MDB's modular I/O subsystem makes our job somewhat easier. Each I/O module provides an ops vector, exposing interfaces for reading, writing, ioctls, and so forth. `kmdb` has its own I/O module, called promio. promio acts as a front end for `promif`, which we discuss in a moment. For the most part, promio is a pass-through, with the exception of the ioctl function. promio interprets the ioctls sent from termio and invokes the appropriate `promif` functions to gather the necessary information. In addition to the aforementioned terminal size ioctl (TIOCGWINSZ), promio's ioctl handler is prepared to deal with requests to get (TCGETS) and set (TCSETSW) hardware parameters. The parameters of interest to `kmdb` are largely concerned with echoing and newlines.

`promif` interfaces the debugger with the system's OpenBoot PROM (OBP). While x86 systems don't have PROMs, Solaris (and thus `kmdb`) try very hard to pretend that they do. For the most part, this means functions called `prom_`*something*`()` are named to mimic their SPARC counterparts. Whereas the SPARC versions jump into OBP, the x86 versions do whatever is necessary to implement the same functionality without a PROM. `promif` exposes two classes of interface: those that deal with console (terminal) I/O, and those that are merely wrappers around PROM routines. We cover the former group here.

Both SPARC and x86 systems get help from the boot loader (OBP on SPARC) for console I/O during the initial stages of boot. SPARC systems without USB keyboards can use OBP for console I/O even after boot. x86 systems and SPARC systems with USB keyboards use a kernel subsystem known as polled I/O. Exposed to `kmdb` through the KDI, polled I/O is a method for interacting directly with the I/O hardware, be it a serial driver, the USB stack, or something completely different without blocking. Rather than waiting for interrupts, as can be done while the world is turning, the polled I/O subsystem is designed to poll I/O devices until input is available or output has been sent. The bottom line is that the method used for console I/O changes during the boot process. The portion of `promif` dedicated to console I/O hides this complexity from consumers, exposing only routines for reading and writing bytes. Consumers need not concern themselves with where those bytes come from or go to.

3.7.4.9 Interrupt (^C) Management

Given that `kmdb` console I/O is synchronous, there is no easy way for an interrupt (^C) from a user to get to the core debugger. In userland, the kernel detects interrupts asynchronously, generates a signal, and inflicts it upon the process. There is no parallel in `kmdb`. The debugger doesn't know about pending interrupts until it reads the interrupt character from the keyboard. With a simplistic I/O implementation, reading only when we need to, a user would never be able to interrupt anything.

promif works around this limitation by implementing a read-ahead buffer. That buffer is drained when the debugger needs input from the user. It is filled whenever input is available by a nonblocking reader. Attempts are made to fill the buffer whenever input is requested, when data is to be output, or when an attempt is made to read or write the kernel's address space. If an interrupt character is discovered during a buffer fill, control passes to the interrupt-handling routine, which halts the command that was executing. Debugger commands that aren't constantly writing to the console, reading from the kernel, or writing to the kernel are very rare (and probably of questionable utility). In practice, this means that a buffer fill attempt will be made soon after the user presses ^C. As a future enhancement, we could, barring the implementation of an asynchronous interrupt-delivery mechanism, expand the number of fill points. In practice, though, this doesn't seem like it would be necessary.

3.7.5 Conclusion

A significant portion of the design and implementation of kmdb was spent filling in the gaping holes left when mdb was separated from its supporting libraries. Certainly, we didn't realize how much is provided by those supporting libraries until we attempted to take them away. These gaps were filled by replacement subsystems whose operations were complicated by the restrictive environment in which kmdb operates. The balance of kmdb's implementation was spent in the development of the KDI functions and in the implementation of the DPI, more on which below. The DPI provides the low-level code that allows the remainder of kmdb to be largely architecture neutral.

3.7.6 Remaining Components

In this section, we cover some remaining discussion items related to the implementation of kmdb.

3.7.6.1 The Debugger/PROM Interface (DPI)

The DPI has a somewhat sordid history, the twists and turns of which have influenced the way it appears today.

kadb on x86, having no PROM, did everything itself. The SPARC version on the other hand, depended on a great many services provided by OBP. OBP provided trap handling for the debugger. It also took care of debugger entry, the saving of a portion of processor state, among other things.

kmdb was initially planned to be released in conjunction with an enhanced OBP. This new OBP would accord more sophisticated debugging facilities, thus freeing kmdb from having to deal with many low-level, hardware-specific details. For example, the new OBP would manage software breakpoints itself. It would capture and park processors during debugger execution. It would also manage watchpoints.

Recognizing that not all systems would have this new OBP, we initially designed kmdb with a pluggable interface that would allow for its use on systems with both types of OBP. That interface is called the Debugger/PROM interface, or DPI. SPARC would have one module for the old-style OBP interface, which we called the kadb-style interface (or kaif). SPARC would have a second module for the new-style OBP interface, the name for which has been buried in the sands of time. The debugger would choose between the two modules according to an assessment of OBP features. x86 systems would have a single module, also called kaif.

Some time into the implementation of kmdb (well after the terms DPI and kaif had cemented themselves throughout the source code), the plans for the new-style OBP were dropped. This turned out to be for the best, the reasons for which are beyond the scope of this document. As a result, modern-day kmdb has one module for each architecture. The intervening layer, the DPI, is not strictly necessary. It may not have been invented had it not been for our earlier plans to accommodate multiple styles of OBP interaction. It remains, though, and serves as a useful repository for some functionality common to the two kaif implementations.

The bulk of the kaif module is devoted to the performance of the following five tasks:

1. Coordination of debugger entry
2. Manipulation of processor state
3. Source analysis for execution control
4. Management of breakpoints and watchpoints
5. Trap handling

3.7.6.2 Coordination of Debugger Entry

kmdb is single threaded and establishes a master-slave relationship between the CPUs on the machine. The first CPU to encounter an event that triggers debugger entry, such as a breakpoint, watchpoint, or deliberate entry, becomes the master. The master then cross-traps the remaining CPUs, causing them to enter the debugger as slaves. Slaves spin in busy loops until the world is resumed or until one of them switches places with the master. If multiple CPUs encounter debugger entry events at the same time and thus race for debugger entry, only one will win. The first to grab the master lock wins, with the remainder becoming slaves.

3.7.6.3 Manipulation of Processor State

When processors enter the debugger, they save their register state into per-processor save areas. This state is then exposed to the user of the debugger. The `kaif` module coordinates the saving of this state and also implements the search routines that allow for its retrieval.

3.7.6.4 Source Analysis for Execution Control

MDB supports a number of execution control primitives. In addition to breakpoints and watchpoints, which we discuss^ shortly, it provides for single-step, step-over, step-out, and continue. Single-step halts execution at the next instruction. Step-over is similar, except that it does not step into subroutines. That is, it steps to the next instruction in the current routine. Step-out steps to the next instruction in the calling routine. Continue resumes system execution on all processors (single-step resumes execution only on the processor being stepped).

Single-step is implemented directly by the `kaif` module. On x86, this entails the setting of `EFLAGS.TF`. On SPARC, we set breakpoints at the next possible execution points. If the next instruction is a branch, for example, we may have to set two breakpoints to cover both possible results of the branch.

Step-over and step are implemented independently of single-step. For step-over, MDB calls into the target, which calls into the DPI and `kaif`, asking whether the next instruction requires special processing. If the next instruction is a call, `kaif` returns with the address of the instruction after the call. MDB places a breakpoint at that location and uses continue to "step" over the call. If the next instruction is not a call, the `kaif` module so indicates, and MDB uses normal single-step. When the user requests a step-out, MDB requests, through the target and the DPI, that the `kaif` module locate the next instruction in the calling function.

Whereas single-step releases a single processor to execute a single instruction, continue releases all processors and fully resumes the world. Continue also posts the soft interrupt to the controller if necessary, in support of debugger module management.

3.7.6.5 Management of Breakpoints and Watchpoints

Both SPARC and x86 rely on software breakpoints. That is, a specific instruction (`int $3` on x86, and `ta 0x7e` on SPARC) is written at a given location. When control reaches that location, the debugger is entered. Breakpoints are activated by installation of one of these instructions and are deactivated by restoration of the original instruction.

Watchpoints are implemented by hardware on both platforms. Space on processors being at a premium and watchpoints being relatively rarely used (though oh-so-helpful), processors don't provide many of them and impose restrictions on the ones they do. SPARC, for example, has two watchpoints—one physical and one virtual. SPARC watchpoint sizes are restricted to 8 bytes or any non-zero power of 256. x86 implements four watchpoints, even allowing watchpoints on individual I/O port numbers, but imposes restrictions on their size and access type. Hardware activates watchpoints by writing to the appropriate hardware registers and deactivates them by clearing those registers. The `kaif` ensures that the target activates only the supported number of watchpoints. It also checks to make sure that the watchpoints requested meet the hardware limitations. No attempt is made to synthesize more flexible watchpoints.

3.7.6.6 Trap Handling

On SPARC, `kmdb` has drastically reduced its dependency upon OBP as the project has progressed. This is somewhat ironic in light of our earlier attempts to increase that dependency. Whereas `kadb` allowed OBP to handle traps and to coordinate entrance into the debugger, `kmdb` has its own trap table, handles its own debugger entry, and even handles its own MMU misses.

`kmdb` also installs its own trap table on x86, although the trap table there is called an IDT. Not having ever had an OBP upon which to become dependent, Solaris x86 in-situ debuggers have always handled their own traps and debugger entry.

When `kmdb` gains control of the machine, it switches to its trap table. When the world resumes, the trap table used prior to debugger entry is restored. While `kmdb` is running, traps that are immediately resolvable by the handler (MMU misses to valid addresses, for example) are handled and control is returned to the execution stream that caused the trap. Traps that are not resolvable by the handler cause a debugger reentry. In some cases, such as when an access is being made to the kernel's address space, the debugger takes precautions against traps resulting from those accesses. Reentry caused by such a trap would cause control to be transferred back to the code that initiated the access, with a return code set indicating that an error occurred. Unexpected traps are signs that something has gone wrong and are grounds for entry into a debugger fault state. The stack trace leading up to the access is displayed, and the user is offered the option to induce a crash dump.

3.8 Kernel Built-in MDB dcmds

```
dcmd $<                    - replace input with macro
dcmd $<<                   - source macro
dcmd $>                    - log session to a file
dcmd $?                    - print status and registers
dcmd $C                    - print stack backtrace
dcmd $G                    - enable/disable C++ demangling support
dcmd $M                    - list macro aliases
dcmd $P                    - set debugger prompt string
dcmd $Q                    - quit debugger
dcmd $V                    - get/set disassembly mode
dcmd $W                    - reopen target in write mode
dcmd $X                    - print floating-point registers
dcmd $Y                    - print floating- point registers
dcmd $b                    - list traced software events
dcmd $c                    - print stack backtrace
dcmd $d                    - get/set default output radix
dcmd $e                    - print listing of global symbols
dcmd $f                    - print listing of source files
dcmd $g                    - get/set C++ demangling options
dcmd $i                    - print signals that are ignored
dcmd $l                    - print the representative thread's lwp id
dcmd $m                    - print address space mappings
dcmd $p                    - change debugger target context
dcmd $q                    - quit debugger
dcmd $r                    - print general-purpose registers
dcmd $s                    - get/set symbol matching distance
dcmd $v                    - print non-zero variables
dcmd $w                    - get/set output page width
dcmd $x                    - print floating-point registers
dcmd $y                    - print floating-point registers
dcmd /                     - format data from virtual as
dcmd :A                    - attach to process or core file
dcmd :R                    - release the previously attached process
dcmd :a                    - set read access watchpoint
dcmd :b                    - set breakpoint at the specified address
dcmd :c                    - continue target execution
dcmd :d                    - delete traced software events
dcmd :e                    - step target over next instruction
dcmd :i                    - ignore signal (delete all matching events)
dcmd :k                    - forcibly kill and release target
dcmd :p                    - set execute access watchpoint
dcmd :r                    - run a new target process
dcmd :s                    - single-step target to next instruction
dcmd :t                    - stop on delivery of the specified signals
dcmd :u                    - step target out of current function
dcmd :w                    - set write access watchpoint
dcmd :z                    - delete all traced software events
dcmd =                     - format immediate value
dcmd >                     - assign variable
dcmd ?                     - format data from object file
dcmd @                     - format data from physical as
dcmd \                     - format data from physical as
dcmd array                 - print each array element's address
dcmd attach                - attach to process or corefile
dcmd bp                    - set breakpoint at the specified addresses or symbols
dcmd cat                   - concatenate and display files
dcmd cont                  - continue target execution
dcmd context               - change debugger target context
dcmd dcmds                 - list available debugger commands
dcmd delete                - delete traced software events
dcmd dem                   - demangle C++ symbol names
dcmd dis                   - disassemble near addr
```

```
  dcmd disasms              - list available disassemblers
  dcmd dismode              - get/set disassembly mode
  dcmd dmods                - list loaded debugger modules
  dcmd dump                 - dump memory from specified address
  dcmd echo                 - echo arguments
  dcmd enum                 - print an enumeration
  dcmd eval                 - evaluate the specified command
  dcmd events               - list traced software events
  dcmd evset                - set software event specifier attributes
  dcmd files                - print listing of source files
  dcmd fltbp                - stop on machine fault
  dcmd formats              - list format specifiers
  dcmd fpregs               - print floating point registers
  dcmd grep                 - print dot if expression is true
  dcmd head                 - limit number of elements in pipe
  dcmd help                 - list commands/command help
  dcmd kill                 - forcibly kill and release target
  dcmd list                 - walk list using member as link pointer
  dcmd load                 - load debugger module
  dcmd log                  - log session to a file
  dcmd map                  - print dot after evaluating expression
  dcmd mappings             - print address space mappings
  dcmd next                 - step target over next instruction
  dcmd nm                   - print symbols
  dcmd nmadd                - add name to private symbol table
  dcmd nmdel                - remove name from private symbol table
  dcmd objects              - print load objects information
  dcmd offsetof             - print the offset of a given struct or union member
  dcmd print                - print the contents of a data structure
  dcmd quit                 - quit debugger
  dcmd regs                 - print general-purpose registers
  dcmd release              - release the previously attached process
  dcmd run                  - run a new target process
  dcmd set                  - get/set debugger properties
  dcmd showrev              - print version information
  dcmd sigbp                - stop on delivery of the specified signals
  dcmd sizeof               - print the size of a type
  dcmd stack                - print stack backtrace
  dcmd stackregs            - print stack backtrace and registers
  dcmd status               - print summary of current target
  dcmd step                 - single-step target to next instruction
  dcmd sysbp                - stop on entry or exit from system call
  dcmd term                 - display current terminal type
  dcmd typeset              - set variable attributes
  dcmd unload               - unload debugger module
  dcmd unset                - unset variables
  dcmd vars                 - print listing of variables
  dcmd version              - print debugger version string
  dcmd vtop                 - print physical mapping of virtual address
  dcmd walk                 - walk data structure
  dcmd walkers              - list available walkers
  dcmd whence               - show source of walk or dcmd
  dcmd which                - show source of walk or dcmd
  dcmd wp                   - set a watchpoint at the specified address
  dcmd xdata                - print list of external data buffers

krtld
  dcmd ctfinfo              - list module CTF information
  dcmd modctl               - list modctl structures
  dcmd modhdrs              - given modctl, dump module ehdr and shdrs
  dcmd modinfo              - list module information
  walk modctl               - list modctl structures
```

```
mdb_kvm
  ctor 0x8076f20            - target constructor
  dcmd $?                   - print status and registers
  dcmd $C                   - print stack backtrace
  dcmd $c                   - print stack backtrace
  dcmd $r                   - print general-purpose registers
  dcmd regs                 - print general-purpose registers
  dcmd stack                - print stack backtrace
  dcmd stackregs            - print stack backtrace and registers
  dcmd status               - print summary of current target
```

3.9 GDB-to-MDB Reference

Table 3-15 GDB-to-MDB Migration

GDB	MDB	Description
Starting Up		
`gdb program`	`mdb path mdb -p pid`	Start debugging a command or running process. GDB will treat numeric arguments as pids, while MDB explicitly requires the `-p` option
`gdb program core`	`mdb [ program ] core`	Debug a corefile associated with `program`. For MDB, the program is optional and is generally unnecessary given the corefile enhancements made during Solaris 10.
Exiting		
`quit`	`::quit`	Both programs also exit on Ctrl-D.
Getting Help		
`help`		
`help command`	`::help ::help dcmd ::dcmds ::walkers`	List all the available walkers or dcmds, as well as get help on a specific dcmd (MDB). Another useful trick is `::dmods -l` mod-ule, which lists walkers and dcmds pro-vided by a specific module.
Running Programs		
`run arglist`	`::run arglist`	Run the program with the given argu-ments. If the target is currently running or is a corefile, MDB will restart the program if possible.
`kill`	`::kill`	Forcibly kill and release target.
`show env`	`::getenv`	Display current environment.
`set env var string`	`::setenv var=string`	Set an environment variable.
`get env var`	`::getenv var`	Get a specific environment variable.

Table 3-15 GDB-to-MDB Migration

GDB	MDB	Description
Shell Commands		
`shell cmd`	`! cmd`	Execute the given shell command.
Breakpoints and Watchpoints		
`break func`		
`break *addr`	`addr::bp`	Set a breakpoint at the given address or function.
`break file:line`	—	Break at the given line of the file. MDB does not support source-level debugging.
`break ... if expr`	—	Set a conditional breakpoint. MDB doesn't support conditional breakpoints, though you can get a close approximation with the `-c` option (though its complicated enough to warrant its own post).
`watch expr`	`addr::wp -rwx [-L size]`	Set a watchpoint on the given region of memory.
`info break`		
`info watch`	`::events`	Display active watchpoints and breakpoints. MDB shows you signal events as well.
`delete [n]`	`::delete n`	Delete the given breakpoint or watchpoints.
Program Stack		
`backtrace n`	`::stack $C`	Display stack backtrace for the current thread.
—	`thread::findstack -v`	Display a stack for a given thread. In the kernel, thread is the address of `kthread_t`. In userland, it's the thread identifier.
`info ...`	—	Display information about the current frame. MDB doesn't support the debugging data necessary to maintain the frame abstraction.

Table 3-15 GDB-to-MDB Migration

GDB	MDB	Description
Execution Control		
`continue`		
`c`	`:c`	Continue target.
`stepi`		
`si`	`::step ]`	Step to the next machine instruction. MDB does not support stepping by source lines.
`nexti ni`	`::step over [`	Step over the next machine instruction, skipping any function calls.
`finish`	`::step out`	Continue until returning from the current frame.
`jump *address`	`address>reg`	Jump to the given location. In MDB, `reg` depends on your platform. For SPARC it's `pc`, for i386 its `eip`, and for amd64 it's `rip`.
Display		
`print expr`	`addr::print expr`	Print the given expression. In GDB you can specify variable names as well as addresses. For MDB, you give a particular address and then specify the type to display (which can include dereferencing of members, etc.).
`print /f`	`addr/f`	Print data in a precise format. See `::formats` for a list of MDB formats.
`disassem addr`	`addr::dis`	Disassemble text at the given address or the current PC if no address is specified.

3.10 dcmd and Walker Reference

3.10.1 Commands

```
pipeline [!word...] [;]                basic
expr pipeline [!word...] [;]           set dot, run once
expr, expr pipeline [!word...] [;]     set dot, repeat
,expr pipeline [!word...] [;]          repeat
expr [!word...] [;]                    set dot, last pipeline, run once
,expr [!word...] [;]                   last pipeline, repeat
expr, expr [!word...] [;]              set dot, last pipeline, repeat
!word... [;]                           shell escape
```

3.10.2 Comments

```
//                              Comment to end of line
```

3.10.3 Expressions

```
Arithmetic
        integer                 0i binary, 0o octal, 0t decimal, 0x hex
        0t[0-9]+\.[0-9]+        IEEE floating point
        'cccccccc'              Little-endian character const
        <identifier             variable lookup
        identifier              symbol lookup
        (expr)                  the value of expr
        .                       the value of dot
        &                       last dot used by dcmd
        +                       dot+increment
        ^                       dot-increment
        increment is effected by the last formatting dcmd.

Unary Ops
        #expr                   logical NOT
        ~expr                   bitwise NOT
        -expr                   integer negation
        %expr                   object file pointer dereference
        %/[csil]/expr           object file typed dereference
        %/[1248]/expr           object file sized dereference
        *expr                   virtual address pointer dereference
        */[csil]/expr           virtual address typed dereference
        */[1248]/expr           virtual address sized dereference

        [csil] is char-, short-, int-, or long-sized

Binary Ops
        expr *  expr            integer multiplication
        expr %  expr            integer division
        left #  right           left rounded up to next right multiple
        expr +  expr            integer addition
        expr -  expr            integer subtraction
        expr << expr            bitwise left shift
        expr >> expr            bitwise right shift (logical)
        expr == expr            logical equality
        expr != expr            logical inequality
        expr &  expr            bitwise AND
        expr ^  expr            bitwise XOR
        expr |  expr            bitwise OR
```

3.10.4 Symbols

```
kernel          {module`}{file`}symbol
proc            {LM[0-9]+`}{library`}{file`}symbol
```

3.10.5 dcmds

```
::{module`}d
expr>var        write the value of expr into var
```

3.10.6 Variables

```
0               Most recent value [/\?=]ed.
9               Most recent count for $< dcmd
b               base VA of the data section
d               size of the data
e               VA of entry point
hits            Event callback match count
m               magic number of primary object file, or zero
t               size of text section
thread          TID of current representative thread.

registers are exported as variables (g0, g1, ...)
```

3.10.7 Read Formats

```
/               format VA from .
\               format PA from .
?               format primary object file, using VA from .
=               format value of .

B (1)   hex                     +       dot += increment
C (1)   char (C-encoded)        -       dot -= increment
V (1)   unsigned                ^ (var) dot -= incr*count
b (1)   octal                   N       newline
c (1)   char (raw)              n       newline
d (2)   signed                  T       tab
h (2)   hex, swap endianness    r       whitespace
o (2)   octal                   t       tab
q (2)   signed octal            a       dot as symbol+offset
u (2)   decimal                 I (var) address and instruction
D (4)   signed                  i (var) instruction
H (4)   hex, swap endianness    S (var) string (C-encoded)
O (4)   octal                   s (var) string (raw)
Q (4)   signed octal            E (8)   unsigned
U (4)   unsigned                F (8)   double
X (4)   hex                     G (8)   octal
Y (4)   decoded time32_t        J (8)   hex
f (4)   float                   R (8)   binary
K (4|8) hex uintptr_t           e (8)   signed
P (4|8) symbol                  g (8)   signed octal
p (4|8) symbol                  y (8)   decoded time64_t
```

3.10.8 Write Formats

```
[/\?][vwWZ] value...            value is immediate or $[expr]

/       write virtual addresses
\       write physical addresses
?       write object file

v (1)   write low byte of each value, starting at dot
w (2)   write low 2 bytes of each value, starting at dot
W (4)   write low 4 bytes of each value, starting at dot
Z (8)   write all 8 bytes of each value, starting at dot
```

3.10.9 Search Formats

```
[/\?][lLM] value [mask]         value and mask are immediate or $[expr]

/       search virtual addresses
\       search physical addresses
?       search object file

l (2)   search for 2-byte value, optionally masked
L (4)   search for 4-byte value, optionally masked
M (8)   search for 8-byte value, optionally masked
```

3.10.10 General dcmds

```
::help dcmd
        Give help text for 'dcmd.'
::dmods -l [module...]
        List dcmds and walkers grouped by the dmod which provides them.
::log -e file
        Log session to file.
::quit / $q
        Quit.
```

3.10.11 Target-Related dcmds

```
::status
        Print summary of current target.
$r / ::regs
        Display current register values for target.
$c / ::stack / $C
        Print current stack trace ($C: with frame pointers).
addr[,b]::dump [-g sz] [-e]
        Dump at least b bytes starting at address addr.  -g sets
        the group size -- for 64-bit debugging, '-g 8' is useful.
addr::dis
        Disassemble text, starting around addr.

[ addr ] :b
[ addr ] ::bp [+/-dDestT] [-c cmd] [-n count] sym ...  addr  [cmd ... ]
        Set breakpoint at addr.
$b
::events [-av]
$b [-av]
        Display all the breakpoints.
addr ::delete [id | all]
addr :d [id | all]
        Delete a breakpoint at addr.
:z
        Deletes all breakpoints
::cont [SIG]
:c [SIG]
        Continue the target program, and wait for it to terminate
id ::evset [+/-dDestT] [-c cmd] [-n count] id ...
        Modify the properties of one or more software event specifiers.
::next [SIG]
:e [SIG]
        Step the target program one instruction, but step over subroutine calls.
::step [branch | over | out] [SIG]
:s SIG
:u SIG
        Step the target program one instruction.
addr [,len]::wp [+/-dDestT] [-rwx] [-ip] [-c cmd] [-n count]
addr [,len]:a [cmd... ]
addr [,len]:p [cmd... ]
addr [,len]:w [cmd... ]
        Set a watchpoint at the specified address.
```

3.10.12 CTF-Related

```
addr::print [type] [field...]
        Use CTF info to print out a full structure, or
        particular fields thereof.
::sizeof type / ::offsetof type field / ::enum enumname
        Get information about a type
addr::array [type count] [var]
        Walk the count elements of an array of type 'type'
        starting at address.
addr::list type field [var]
        Walk a circular or NULL-terminated list of type 'type',
        which starts at addr and uses 'field' as its linkage.
::typegraph / addr::whattype / addr::istype type / addr::notype
        bmc's type inference engine -- works on non-debug
```

3.10.13 Kernel: `proc`-Related

```
0tpid::pid2proc
        Convert the process ID 'pid' (in decimal) into a proc_t ptr.
as::as2proc
        Convert a 'struct as' pointer to its associated proc_t ptr.
vn::whereopen
        Find all processes with a particular vnode open.
::pgrep pattern
        Print out proc_t ptrs which match pattern.
[procp]::ps
        Process table, or (with procp) the line for particular proc_t.
::ptree
        Print out a ptree(1)-like indented process tree.
procp::pfiles
        Print out information on a process' file descriptors.

[procp]::walk proc
        walks all processes, or the tree rooted at procp
```

3.10.14 Kernel: Thread-Related

```
threadp::findstack
        Print out a stack trace (with frame pointers) for threadp.
[threadp]::thread
        Give summary information about all threads or a particular thread.

[procp]::walk thread
        Walk all threads, or all threads in a process (with procp).
```

3.10.15 Kernel: Synchronization-Related

```
[sobj]::wchaninfo [-v]
        Get information on blocked-on condition variables.  With
        sobj, info about that wchan.  With -v, lists all threads
        blocked on the wchan.
sobj::rwlock
        Dump out a rwlock, including detailed blocking information.

sobj::walk blocked
        Walk all threads blocked on sobj, a synchronization object.
```

3.10.16 Kernel: CPU-Related

```
::cpuinfo [-v]
        Give information about CPUs on the system and what they
        are doing.  With '-v', show threads on the run queues.
::cpupart
        Give information about CPU partitions (psrset(1m)s).
addr::cpuset
        Print out a cpuset as a list of included CPUs.
[cpuid]::ttrace
        Dump out traptrace records, which are generated in DEBUG
        kernels.  These include all traps and various other events of
        interest.

::walk cpu
        Walk all cpu_ts on the system.
```

3.10.17 Kernel: Memory-Related

```
::memstat
        Display memory usage summary.
pattern::kgrep [-d dist|-m mask|-M invmask]
        Search the kernel heap for pointers equal to pattern.
addr::whatis [-b]
        Try to identify what a given kernel address is.  With
        '-b', give bufctl address for the buffer (see
        $<bufctl_audit, below).
```

3.10.18 Kernel: kmem-Related

```
::kmastat
        Give statistics on the kmem caches and vmem arenas in the system
::kmem_cache
        Information about the kmem caches on the system
[cachep]::kmem_verify
        Validate all buffers in the system, checking for corruption.
        With cachep, shows the details of a particular cache.
threadp::allocdby / threadp::freedby
        Show buffers that were last allocated/freed by a particular
        thread, and are still in that state.
::kmalog [fail | slab]
        Dump out the transaction log, showing recent kmem activity.
        With fail/slab, outputs records of allocation failures and
        slab creations (which are always enabled)
::findleaks [-dvf]
        Find memory leaks, coalesced by stack trace.
::bufctl [-v]
        Print a summary line for a bufctl -- can also filter them
        -v dumps out a kmem_bufctl_audit_t.

::walk cachename
        Print out all allocated buffers in the cache named cachename.

[cp]::walk kmem/[cp]::walk freemem/[cp]::walk bufctl/[cp]::walk freectl
        Walk {allocated,freed}{buffers,bufctls} for all caches,
        or the particular kmem_cache_t cp.
```

3.10.19 Process: Target-Related

```
flt ::fltbp [+/-dDestT] [-c cmd] [-n count] flt ...
        Trace the specified machine faults.
signal :i
        Ignore the specified signal and allow it to be delivered
        transparently to the target.
$i
        Display the list of signals that are ignored by the debugger and
        will be handled directly by the target.
$l
      Print the LWPID of the representative thread if the target is a user process.

$L
        Print the LWPIDs of each LWP in the target if the target is a user
        process.
::kill
:k
        Forcibly terminate the target if it is a live user process.
::run [args ... ]
:r [args ... ]
        Start a new target program running with the specified arguments and
        attach to it.
[signal] ::sigbp [+/-dDestT] [-c cmd] [-n count] SIG ...
[signal] :t [+/-dDestT] [-c cmd] [-n count] SIG ...
        Trace delivery of the specified signals.
::step [branch | over | out] [SIG]
:s SIG
:u SIG
        Step the target program one instruction.
[syscall] ::sysbp [+/-dDestT] [-io] [-c cmd] [-n count] syscall ...
        Trace entry to or exit from the specified system calls.
```

3.10.20 Kernel: kmdb-Related

```
::help dcmd
        gives help text for 'dcmd'
::dmods -l [module...]
        Lists dcmds and walkers grouped by the dmod which provides them

::status
        Print summary of current target.
$r
::regs
        Display current register values for target.
$c
::stack
$C
        Print current stack trace ($C: with frame pointers).
addr[,b]
::dump [-g sz] [-e]
        Dump at least b bytes starting at address addr.  -g sets the group size;
        for 64-bit debugging, -g 8 is useful.
addr::dis
        Disassemble text, starting around addr.
[ addr ] :b
[ addr ] ::bp [+/-dDestT] [-n count] sym ...  addr
        Set breakpoint at addr.
$b
        Display all the breakpoints.
::branches
        Display the last branches taken by the CPU. (x86 only)
```

```
addr ::delete [id | all]
addr :d [id | all]
        Delete a breakpoint at addr.
:z
        Delete all breakpoints.
function ::call [arg [arg ...]]
        Call the specified function, using the specified arguments.
[cpuid] ::cpuregs [-c cpuid]
        Display the current general-purpose register set.
[cpuid] ::cpustack [-c cpuid]
        Print a C stack backtrace for the specified CPU.
::cont
:c
        Continue the target program.
$M
        List the macro files that are cached by kmdb for use with the $< dcmd
::next
:e
        Step the target program one instruction, but step over subroutine calls.
::step [branch | over | out]
        Step the target program one instruction.
$<systemdump
        Initiate a panic/dump.
::quit [-u]
$q
        Cause the debugger to exit. When the -u option is used,
        the system is resumed and the debugger is unloaded.
        addr [,len]::wp [+/-dDestT] [-rwx] [-ip] [-n count]

addr [,len]:a [cmd ...]
addr [,len]:p [cmd ...]
addr [,len]:w [cmd ...]
        Set a watchpoint at the specified address.
```