Solaris Modular Debugger Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-1583–10
February 2000

Contents

Preface

The Modular Debugger (MDB) is a new general purpose debugging tool for the Solaris™ Operating Environment. Its primary feature is its extensibility. The *Solaris Modular Debugger Guide* describes how to use MDB to debug complex software systems, with a particular emphasis on the facilities available for debugging the Solaris kernel and associated device drivers and modules. It also includes a complete reference for and discussion of the MDB language syntax, debugger features, and MDB Module Programming API.

Who Should Use This Book

If you were a detective and were investigating at the scene of a crime, you might interview the 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 could be examined for DNA to help solve the case. Often, software program failures divide into analagous 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 is a debugger designed to facilitate 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, these situations can occur frequently. As a result, MDB is designed as a debugging framework that allows you to construct your own custom analysis tools to aid in the diagnosis of these problems. MDB also provides a powerful set of built-in commands that allow you to analyze the state of your program at the assembly language level.

If you are not familiar with assembly language programming and debugging, “Related Books and Papers” on page provides references to materials that you might find useful.

You should also disassemble various functions of interest in the programs you will be debugging in order to familiarize yourself with the relationship between your program’s source code and the corresponding assembly language code. If you are planning to use MDB for debugging Solaris kernel software, you should read carefully Chapter 5 and Chapter 6. These chapters provide more detailed information on the MDB commands and facilities provided for debugging Solaris kernel software.

How This Book Is Organized

Chapter 1 provides an overview of the debugger. This chapter is intended for all users..

Chapter 2 describes the MDB architecture and explains the terminology for the debugger concepts used throughout this book. This chapter is intended for all users.

Chapter 3 describes the syntax, operators and evaluation rules for the MDB language. This chapter is intended for all users.

Chapter 4 describes the set of built-in debugger commands that are always available. This chapter is intended for all users.

Chapter 5 describes the set of loadable debugger commands that are provided for debugging the Solaris kernel. This chapter is intended for users who intend to examine Solaris kernel crash dumps and for kernel software developers.

Chapter 6 describes the debugging features of the Solaris kernel memory allocator and the MDB commands provided to take advantage of these features. This chapter is intended for advanced programmers and kernel software developers.

Chapter 7 describes the facilities for writing loadable debugger modules. This chapter is intended for advanced programmers and software developers who intend to develop custom debugging support for MDB.

Appendix A provides a reference for MDB command-line options.

Appendix B provides a reference for `crash(1M)` commands and their MDB equivalents.

Related Books and Papers

These books and papers are recommended and related to the tasks that you need to perform:

- Vahalia, Uresh. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996. ISBN 0-13-101908-2
- *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1998. ISBN 0–13–099227–5
- *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1994. ISBN 0-13-825001-4
- *Pentium Pro Family Developer's Manual, Volumes 1-3*. Intel Corporation, 1996. ISBN 1-55512-259-0 (Volume 1) , ISBN 1-55512-260-4 (Volume 2) , ISBN 1-55512-261-2 (Volume 3)
- Bonwick, Jeff. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*. Proceedings of the Summer 1994 Usenix Conference, 1994. ISBN 9–99–452010–5
- *SPARC Assembly Language Reference Manual*. Sun Microsystems, 1998.
- *x86 Assembly Language Reference Manual*. Sun Microsystems, 1998.
- *Writing Device Drivers*. Sun Microsystems, 2000.
- *STREAMS Programming Guide*. Sun Microsystems, 2000.
- *Solaris 64-bit Developer's Guide*. Sun Microsystems, 2000.
- *Linker and Libraries Guide*. Sun Microsystems, 2000.

Note - In this document, the term "IA" refers to the Intel 32–bit processor architecture, which includes the Pentium, Pentium Pro, Pentium II, Pentium II Xeon, Celeron, Pentium III, and Pentium III Xeon processors, and compatible microprocessor chips made by AMD and Cyrix.

Note - The Solaris operating environment runs on two types of hardware, or platforms—SPARC™ and IA. The Solaris operating environment also runs on both 64–bit and 32–bit address spaces. The information in this document pertains to both platforms and address spaces unless called out in a special chapter, section, note, bullet, figure, table, example, or code example.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems™, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`
MDB prompt	`>`

CHAPTER 1

Modular Debugger Overview

The Modular Debugger (MDB) is a new general purpose debugging tool for Solaris whose primary feature is its extensibility. This book describes how to use MDB to debug complex software systems, with a particular emphasis on the facilities available for debugging the Solaris kernel and associated device drivers and modules. The book also includes a complete reference for and discussion of the MDB language syntax, debugger features, and MDB Module Programming API.

Introduction

Debugging is the process of analyzing the execution and state of a software program in order to remove defects. Traditional debugging tools provide facilities for execution control so that programmers can re-execute programs in a controlled environment and display the current state of program data or evaluate expressions in the source language used to develop the program. Unfortunately, these techniques are often inappropriate for debugging complex software systems such as:

- An operating system, where bugs might not be reproducible and program state is massive and distributed
- Programs that are highly optimized or have had their debug information removed
- Programs that are themselves low-level debugging tools
- Customer situations where the developer can only access post-mortem information

MDB is a tool that provides a completely customizable environment for debugging these programs and scenarios, including a dynamic module facility that programmers can use to implement their own debugging commands to perform program-specific analysis. Each MDB module can be used to examine the program in

several different contexts, including live and post-mortem. The Solaris Operating Environment includes a set of MDB modules designed to aid 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.

MDB Features

MDB provides an extensive collection of features for analyzing the Solaris kernel and other target programs. You can:

- Perform post-mortem 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:
  - 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 that the debugger can `dlopen`(3DL)), 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(1)` and `crash(1M)`: MDB provides backward compatibility 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, including:
  - Command-line editing

- Command history
- Built-in output pager
- Syntax error checking and handling
- Online help
- Interactive session logging

Future Enhancements

MDB provides a stable foundation for developing advanced post-mortem analysis tools. In the future, the Solaris operating environment will include additional MDB modules that provide even more sophisticated functionality for debugging the kernel and other software programs. You can use MDB to debug existing software programs, and develop your own modules to improve your ability to debug your own Solaris drivers and applications.

CHAPTER 2

Debugger Concepts

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

Architecture

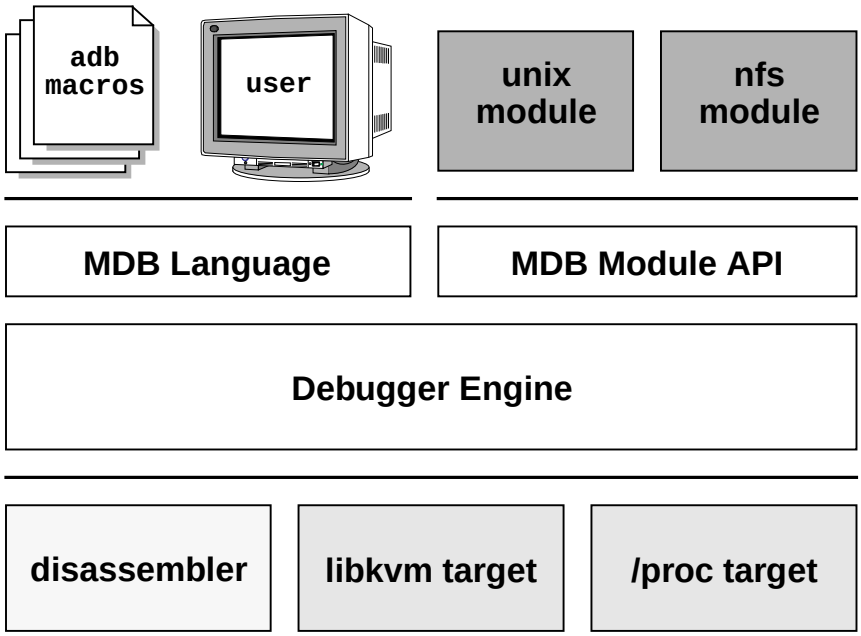

Figure 2–1 MDB architecture

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 (through `/dev/kmem` and `/dev/ksyms`)
- Operating system crash dumps
- User process images recorded inside an operating system crash dump
- ELF object 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 2–1 shows an overview of the MDB architecture, including two of the built-in targets and a pair of sample modules.

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 “Syntax” on page 23. MDB contains a set of built-in dcmds described in Chapter 4, 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 use them as a primitive to build other dcmds or walkers. As with dcmds, the programmer 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 are typically used to 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.

Modularity

The benefit of MDB's modular architecture extends beyond the ability to load a shared library containing additional debugger commands. The MDB architecture defines clear interface boundaries between each of the layers shown in Figure 2–1. Macro files execute commands written in the MDB or adb language. Dcmds and walkers in debugger modules are written using 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 name space 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 in order to produce its output. However, the code to iterate over the list is not in the `::ps` dcmd, it 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. The proc walker can also be accessed interactively using the `::walk` dcmd in order 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 are used to access information from user process or kernel targets, simplifying the task of developing new debugging facilities.

In addition, a custom MDB module can be used to 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 where 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 Tables — 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, load objects, register values, and control the execution of user process targets.

CHAPTER 3

Language Syntax

This chapter describes the MDB language syntax, operators, and rules for command and symbol name resolution.

Syntax

The debugger processes commands from standard input. If standard input is a terminal, MDB provides terminal editing capabilities. MDB can also process commands from macro files and from dcmd pipelines, described below. 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 dcmd to that address. The current address location is referred to as *dot*, and " . " is used to reference its value.

A *metacharacter* is one of the following characters:

```
[ ]| ! / \ ? = > $ : ; NEWLINE SPACE TAB
```

A *blank* is a TAB or a SPACE. A *word* is a sequence of characters separated by one or more non-quoted metacharacters. Some of the metacharacters function only as delimiters in certain contexts, as described below. An *identifier* is a sequence of letters, digits, underscores, periods, or back quotes beginning with a letter, underscore, or period. Identifiers are used as the names of symbols, variables, dcmds, and walkers. Commands are delimited by a NEWLINE or semicolon (;).

A dcmd is denoted by one of the following words or metacharacters:

```
/ \ ? = > $character :character ::identifier
```

dcmds named by metacharacters or prefixed by a single $ or : are provided as built-in operators, and implement complete compatibility with the command set of the legacy adb(1) utility. After a dcmd has been parsed, the /, \, ?, =, >, $, and :

characters are no longer recognized as metacharacters until the termination of the argument list.

A *simple-command* is a dcmd followed by a sequence or zero or more blank-separated words. The words are passed as arguments to the invoked dcmd, except as specified under “Arithmetic Expansion” on page 26 and “Quoting” on page 27. Each dcmd returns an exit status that indicates it was either successful, failed, or was invoked with invalid arguments.

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. Each dcmd's output is processed and stored as described in “dcmd Pipelines” on page 31. 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.

An *expression* is a sequence of words that is evaluated to compute a 64-bit unsigned integer value. The words are evaluated using the rules described in “Arithmetic Expansion” on page 26.

Commands

A *command* is one of the following:

pipeline [! word ...] [;]

A simple-command or pipeline can be optionally suffixed with the `!` character, indicating that the debugger should open a `pipe`(2) and send the standard output of the last dcmd in the MDB pipeline to an external process created by executing `$SHELL -c` followed by the string formed by concatenating the words after the `!` character. For more details, refer to “Shell Escapes” on page 28.

expression pipeline [! word ...] [;]

A simple-command or pipeline can be prefixed with an expression. Before execution of the pipeline, the value of dot (the variable denoted by “ `.` “) is set to the value of the expression.

expression , expression pipeline [! word ...] [;]

A simple-command or pipeline can be prefixed with two expressions. The first is evaluated to determine the new value of dot, and the second is evaluated to determine a repeat count for the first dcmd in the pipeline. This dcmd will be

executed *count* times before the next dcmd in the pipeline is executed. The repeat count applies only to the first dcmd in the pipeline.

, expression pipeline [! word ...] [;]

If the initial expression is omitted, dot is not modified; however, the first dcmd in the pipeline will be repeated according to the value of the expression.

expression [! word ...] [;]

A command can consist only of an arithmetic expression. The expression is evaluated and the dot variable is set to its value, then the previous dcmd and arguments are executed using the new value of dot.

expression , expression [! word ...] [;]

A command can consist only of a dot expression and repeat count expression. After dot is set to the value of the first expression, the previous dcmd and arguments are repeatedly executed the number of times specified by the value of the second expression.

, expression [! word ...] [;]

If the initial expression is omitted, dot is not modified but the previous dcmd and arguments are repeatedly executed the number of times specified by the value of the count expression.

! word ... [;]

If the command begins with the `!` character, no dcmds are executed and the debugger executes `$SHELL –c` followed by the string formed by concatenating the words after the `!` character.

Comments

A word beginning with `//` causes that word and all the subsequent characters up to a `NEWLINE` to be ignored.

Arithmetic Expansion

Arithmetic expansion is performed when an MDB command is preceded by an optional expression representing a start address, or a start address and a repeat count. Arithmetic expansion can also be performed to compute a numerical argument for a dcmd. An arithmetic expression can appear in an argument list enclosed in square brackets preceded by a dollar sign (`$[ expression ]`), and will be replaced by the value of the expression.

Expressions can contain any of the following special words:

integer	The specified integer value. Integer values can be prefixed with `0i` or `0I` to indicate binary values, `0o` or `0O` to indicate octal values, `0t` or `0T` to indicate decimal values, and `0x` or `0X` to indicate hexadecimal values (the default).
0[tT][0-9]+.[0-9]+	The specified decimal floating point value, converted to its IEEE double-precision floating point representation
'*cccccccc*'	The integer value computed by converting each character to a byte equal to its ASCII value. Up to eight characters can be specified in a character constant. Characters are packed into the integer in reverse order (right-to-left), beginning at the least significant byte.
<*identifier*	The value of the variable named by *identifier*
identifier	The value of the symbol named by *identifier*
(*expression*)	The value of *expression*
.	The value of dot
&	The most recent value of dot used to execute a dcmd
+	The value of dot incremented by the current increment
^	The value of dot decremented by the current increment

The increment is a global variable that stores the total bytes read by the last formatting dcmd. For more information on the increment, refer to the discussion of "Formatting dcmds" on page 31.

Unary operators are right associative and have higher precedence than binary operators. The unary operators are:

#expression	Logical negation
~expression	Bitwise complement
-expression	Integer negation
%expression	Value of a pointer-sized quantity at the object file location corresponding to virtual address *expression* in the target's virtual address space
%/[csil]/expression	Value of a char-, short-, int-, or long-sized quantity at the object file location corresponding to virtual address *expression* in the target's virtual address space
%/[1248]/expression	Value of a one-, two-, four-, or eight-byte quantity at the object file location corresponding to virtual address *expression* in the target's virtual address space
****expression***	Value of a pointer-sized quantity at virtual address *expression* in the target's virtual address space
****/[csil]/expression***	Value of a char-, short-, int-, or long-sized quantity at virtual address *expression* in the target's virtual address space
****/[1248]/expression***	Value of a one-, two-, four-, or eight-byte quantity at virtual address *expression* in the target's virtual address space

Quoting

Each metacharacter described above (see Chapter 3) terminates a word unless quoted. Characters can be quoted (forcing MDB to interpret each character as itself without any special significance) by enclosing them in a pair of single (') or double (") quotation marks. A single quote cannot appear within single quotes. Inside double quotes, MDB recognizes the C programming language character escape sequences.

Shell Escapes

The `!` character can be used to create a pipeline between an MDB command and the user's shell. If the $SHELL environment variable is set, MDB will `fork` and `exec` this program for shell escapes; otherwise `/bin/sh` is used. The shell is invoked with the `–c` option followed by a string formed by concatenating the words after the `!` character.

The `!` character takes precedence over all other metacharacters, except semicolon (`;`) and `NEWLINE`. After a shell escape is detected, the remaining characters up to the next semicolon or `NEWLINE` are passed "as is" to the shell. The output of shell commands cannot be piped to MDB dcmds. Commands executed by a shell escape have their output sent directly to the terminal, not to MDB.

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 using the `>` dcmd or `::typeset` dcmd, and its attributes can be manipulated using 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), and tagged (user-defined indicator).

The following variables are defined as persistent:

0	Most recent value printed using the `/`, `\`, `?`, or `=` dcmd.
9	Most recent count used with the `$<` dcmd
b	Virtual address of the base of the data section
d	Size of the data section in bytes
e	Virtual address of the entry point
m	Initial bytes (magic number) of the target's primary object file, or zero if no object file has been read yet
t	Size of the text section in bytes

In addition, the MDB kernel and process targets export the current values of the representative thread's register set as named variables. The names of these variables depend on the target's platform and instruction set architecture.

Symbol Name Resolution

As explained in "Syntax" on page 23, a symbol identifier present in an expression context evaluates to the value of this symbol. The value typically denotes the virtual address of the storage associated with the symbol in the target's virtual address space. A target can support multiple symbol tables including, but not limited to,

- Primary executable symbol table
- Primary dynamic symbol table
- Runtime link-editor symbol table
- Standard and dynamic symbol tables for each of a number of load objects (such as shared libraries in a user process, or kernel modules in the Solaris kernel)

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 prior to any of the target symbol tables. The private symbol table is initially empty, and can be manipulated using the `::nmadd` and `::nmdel` dcmds.

The `::nm` `-P` option can be used to display 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. 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.

You can specify the scope used to resolve a symbol name as either: *object`name*, or *file`name*, or *object`file`name*. The object identifier refers to the name of a load object. The file identifier refers to the basename of a source file that has a symbol of type `STT_FILE` in the specified object's symbol table. The object identifier's interpretation depends on the target type.

The MDB kernel target expects *object* to specify the base name of a loaded kernel module. For example, the symbol name:

```
specfs‘_init
```

evaluates to the value of the `_init` symbol in the `specfs` kernel module.

The `mdb` process target expects *object* to specify the name of the executable or of a loaded shared library. It can take any of the following forms:

- Exact match (that is, a full pathname): `/usr/lib/libc.so.1`
- Exact basename match: `libc.so.1`
- Initial basename match up to a ‘‘.’’ suffix: `libc.so` or `libc`
- Literal string `a.out` which is accepted as an alias for the executable

In the case of a naming conflict between symbols and hexadecimal integer values, MDB attempts to evaluate an ambiguous token as a symbol first, before evaluating it as an integer value. For example, the token `f` can refer either to the decimal integer value `15` specified in hexadecimal (the default base), or to a global variable named `f` in the target’s symbol table. If a symbol with an ambiguous name is present, the integer value can be specified by using an explicit `0x` or `0X` prefix.

dcmd and Walker Name Resolution

As described earlier, each MDB dmod provides a set of dcmds and walkers. dcmds and walkers are tracked in two distinct, global namespaces. MDB also keeps track of a dcmd and walker namespace associated with each dmod. Identically named dcmds or walkers within a given dmod are not allowed: a dmod with this type of naming conflict will fail to load.

Name conflicts between dcmds or walkers from different dmods are allowed in the global namespace. In the case of a conflict, the first dcmd or walker with that particular name to be loaded is given precedence in the global namespace. Alternate definitions are kept in a list in load order.

The backquote character “ ‘ “ can be used in a dcmd or walker name as a scoping operator to select an alternate definition. For example, if dmods `m1` and `m2` each provide a dcmd `d`, and `m1` is loaded prior to `m2`, then:

`::d`	Executes `m1`’s definition of `d`
`::m1‘d`	Executes `m1`’s definition of `d`
`::m2‘d`	Executes `m2`’s definition of `d`

If module `m1` were now unloaded, the next dcmd on the global definition list (`m2'd`) would be promoted to global visibility. The current definition of a dcmd or walker can be determined using the `::which` dcmd, described below. The global definition list can be displayed using the `::which –v` option.

dcmd Pipelines

dcmds can be composed into a pipeline using the `|` operator. 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 be used to map a pointer from one type of data structure to a pointer to a corresponding data structure, to sort a list of addresses, or to select the addresses of structures with certain properties.

MDB executes each dcmd in the pipeline in order from left to right. The left-most dcmd is executed using the current value of dot, or using 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.

As the dcmd executes, its standard output is placed in the pipe and then consumed and evaluated by the parser, as if MDB were reading this data from standard input. Each line must consist of an arithmetic expression terminated by a `NEWLINE` or semicolon (`;`). The value of the expression is appended to the list of values associated with the pipe. If a syntax error is detected, the pipeline is aborted.

When the dcmd to the left of a `|` operator completes, the list of values associated with the pipe is then used to invoke the dcmd to the right of the `|` operator. For each value in the list, dot is set to this value and the right-hand dcmd is executed. Only the rightmost dcmd in the pipeline has its output printed to standard output. If any dcmd in the pipeline produces output to standard error, these messages are printed directly to standard error and are not processed as part of the pipeline.

Formatting dcmds

The `/`, `\`, `?`, and `=` metacharacters are used to 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 the table below.

Format characters are used to read and format data from the target. A repeat count is a positive integer preceding the format character that 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.

The formatting dcmds are:

`/`	Display data from the target's virtual address space starting at the virtual address specified by dot.
`\`	Display data from the target's physical address space starting at the physical address specified by dot.
`?`	Display data from the target's primary object file starting at the object file location corresponding to the virtual address specified by dot.
`=`	Display the value of dot itself in each of the specified data formats. The `=` dcmd is therefore useful for converting between bases and performing arithmetic.

In addition to dot, MDB keeps track of another global value called the *increment*. The increment represents the distance between dot and the address following all the data read by the last formatting dcmd.

For example, if a formatting dcmd is executed with dot equal to address A, and displays a 4-byte integer, then after this dcmd completes, dot is still A, but the increment is set to 4. The `+` character (described "Arithmetic Expansion" on page 26) would now evaluate to the value `A + 4`, and could be used to reset dot to the address of the next data object for a subsequent dcmd.

Most format characters increase the value of the increment by the number of bytes corresponding to the size of the data format, shown in the table. The table of format characters can be displayed from within MDB using the `::formats` dcmd. The format characters are:

+	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)
F	Double (8 bytes)
G	Octal unsigned long long (8 bytes)
H	Swap bytes and shorts (4 bytes)
I	Address and disassembled instruction (variable size)

J	Hexadecimal long long (8 bytes)
K	Hexadecimal uintptr_t (4 or 8 bytes)
O	Octal unsigned int (4 bytes)
P	Symbol (4 or 8 bytes)
Q	Octal signed int (4 bytes)
S	String using C string notation (variable size)
U	Decimal unsigned int (4 bytes)
V	Decimal unsigned int (1 byte)
W	Default radix unsigned int (4 bytes)
X	Hexadecimal int (4 bytes)
Y	Decoded time32_t (4 bytes)
Z	Hexadecimal long long (8 bytes)
^	Decrement dot by increment * count (variable size)
a	Dot as symbol+offset
b	Octal unsigned int (1 byte)
c	Character (1 byte)
d	Decimal signed short (2 bytes)
e	Decimal signed long long (8 bytes)
f	Float (4 bytes)
g	Octal signed long long (8 bytes)
h	Swap bytes (2 bytes)
i	Disassembled instruction (variable size)
n	Newline
o	Octal unsigned short (2 bytes)
p	Symbol (4 or 8 bytes)
q	Octal signed short (2 bytes)
r	Whitespace
s	Raw string (variable size)
t	Horizontal tab
u	Decimal unsigned short (2 bytes)
v	Decimal signed int (1 byte)
w	Default radix unsigned short (2 bytes)

x	Hexadecimal short (2 bytes)
y	Decoded time64_t (8 bytes)

The `/`, `\`, and `?` formatting dcmds can also be used to write to the target's virtual address space, physical address space, or object file by specifying one of the following modifiers as the first format character, and then specifying a list of words that are either immediate values or expressions enclosed in square brackets preceded by a dollar sign (`$[ ]`).

The write modifiers are:

v, w	Write the lowest 2 bytes of the value of each expression to the target beginning at the location specified by dot
W	Write the lowest 4 bytes of the value of each expression to the target beginning at the location specified by dot
Z	Write the complete 8 bytes of the value of each expression to the target beginning at the location specified by dot

The `/`, `\`, and `?` formatting dcmds can also be used to search for a particular integer value in the target's virtual address space, physical address space, and object file, respectively, by specifying one of the following modifiers as the first format character, then specifying a value and optional mask. The value and mask are each specified as either immediate values or expressions enclosed in square brackets preceded by a dollar sign.

If only a value is specified, MDB reads integers of the appropriate size and stops at the address containing the matching value. If a value `V` and mask `M` are specified, MDB reads integers of the appropriate size and stops at the address containing a value `X` where `(X & M) == V`. At the completion of the dcmd, dot is updated to the address containing the match. If no match is found, dot is left at the last address that was read.

The search modifiers are:

l	Search for the specified 2-byte value
L	Search for the specified 4-byte value
M	Search for the specified 8-byte value

For both user and kernel targets, an address space is typically composed of a set of discontiguous segments. It is not legal to read from an address that does not have a corresponding segment. If a search reaches a segment boundary without finding a match, it aborts when the read past the end of the segment boundary fails.

CHAPTER 4

Built-in Commands

MDB provides a set of built-in dcmds that are always defined. Some of these dcmds are applicable only to certain targets: if a dcmd is not applicable to the current target, it fails and prints a message indicating "command is not supported by current target".

In many cases, MDB provides a mnemonic equivalent (`::identifier`) for the legacy `adb`(1) dcmd names. For example, `::quit` is provided as the equivalent of `$q`. Programmers who are experienced with `adb`(1) or who appreciate brevity or arcana might prefer the `$` or `:` forms of the built-ins. Programmers who are new to `mdb` might prefer the more verbose `::` form. The builtiins are shown in alphabetical order. If a `$` or `:` form has a `::identifier` equivalent, it is shown under the `::identifier` form.

Built-in dcmds

> *variable-name*
>/*modifier*/ *variable-name*

Assign the value of dot to the specified named variable. Some variables are read-only and cannot be modified. If the `>` is followed by a modifier character surrounded by `//`, then the value is modified as part of the assignment. The modifier characters are:

`c` Unsigned char quantity (1-byte)

`s` Unsigned short quantity (2-byte)

`i` Unsigned int quantity (4-byte)

`l` Unsigned long quantity (4-byte in 32-bit, 8-byte in 64-bit)

Notice that these operators do not perform a cast; they instead fetch the specified number of low-order bytes (on little-endian architectures) or high-order bytes (big-endian architectures). These modifiers are provided for backward compatibility; the MDB **/*modifier*/ and %/*modifier*/ syntax should be used instead.

$< ***macro-name***

Read and execute commands from the specified macro file. The file name can be given as an absolute or relative path. If the file name is a simple name (that is, if it does not contain a '/'), MDB searches for it in the macro file include path. If another macro file is currently being processed, this file is closed and replaced with the new file.

$<< ***macro-name***

Read and execute commands from the specified macro file (as with $<), but do not close the current open macro file.

$?

Print the process-ID and current signal of the target if it is a user process or core file, then print the general register set of the representative thread.

[***address*** **]** $C **[** ***count*** **]**

Print a C stack backtrace, including stack frame pointer information. If the dcmd is preceded by an explicit *address*, a backtrace beginning at this virtual memory address is displayed. Otherwise, the stack of the representative thread is displayed. If an optional count value is given as an argument, no more than *count* arguments are displayed for each stack frame in the output.

For 64-bit systems (SPARC) - The biased frame pointer value (that is, the virtual address plus `0x7ff`) should be used as the address when requesting a stack trace.

[***base*** **]** $d

Get or set the default output radix. If the dcmd is preceded by an explicit expression, the default output radix is set to the given *base*; otherwise, the current radix is printed in base 10 (decimal). The default radix is base 16 (hexadecimal).

$e

Print a list of all known external (global) symbols of type object or function, the value of the symbol, and the first 4 (32-bit `mdb`) or 8 (64-bit `mdb`) bytes stored at this location in the target's virtual address space. The `::nm` dcmd provides more flexible options for displaying symbol tables.

`$P` ***prompt-string***

Set the prompt to the specified *prompt-string*. The default prompt is '> '. The prompt can also be set using `::set` –`P` or the –`P` command-line option.

distance `$s`

Get or set the symbol matching *distance* for address-to-symbol-name conversions. The symbol matching distance modes are discussed along with the –`s` command-line option in Appendix A. The symbol matching distance can also be modified using the `::set` –`s` option. If no distance is specified, the current setting is displayed.

`$v`

Print a list of the named variables that have non-zero values. The `::vars` dcmd provides other options for listing variables.

width `$w`

Set the output page *width* to the specified value. Typically, this command is not necessary, as `mdb` queries the terminal for its width and handles resize events.

`$W`

Reopen the target for writing, as if `mdb` had been executed with the –`w` option on the command line. Write mode can also be enabled with the `::set` –`w` option.

[***pid*** **]** `::attach` **[** ***core*** **|** ***pid*** **]**
[***pid*** **]** `:A` **[** ***core*** **|** ***pid*** **]**

If the user process target is active, attach to and debug the specified process-ID or *core* file. The core file path name should be specified as a string argument. The process-ID can be specified as the string argument, or as the value of the expression preceding the dcmd. Recall that the default base is hexadecimal, so decimal PIDs obtained using `pgrep`(1) or `ps`(1) should be preceded with "`0t`" when specified as expressions.

`::cat` ***filename ...***

Concatenate and display files. Each file name can be specified as a relative or absolute path name. The file contents will print to standard output, but will not pass through the output pager. This dcmd is intended to be used with the `|` operator; the programmer can initiate a pipeline using a list of addresses stored in an external file.

address `::context`
address `$p`

Context switch to the specified process. A context switch operation is valid only when using the kernel target. The process context is specified using the *address* of its proc structure in the kernel's virtual address space. The special context address "`0`" is used to denote the context of the kernel itself. MDB can perform only a context switch when examining a crash dump if the dump contains all physical memory pages (as opposed to just kernel pages). The kernel crash dump facility can be configured to dump all pages using `dumpadm`(1M).

When the user requests a context switch from the kernel target, MDB constructs a new target representing the specified user process. After the switch occurs, the new target interposes its dcmds at the global level: thus the `/` dcmd can now format and display data from the virtual address space of the user process, the `::mappings` dcmd can display the mappings in the address space of the user process, and so on. The kernel target can be restored by executing `0::context`.

`::dcmds`

List the available dcmds and print a brief description for each one.

[***address*** **]** `::dis` **[** `–fw` **] [** `–n` ***count*** **] [** ***address*** **]**

Disassemble starting at or around the *address* specified by the final argument, or the current value of dot. If the address matches the start of a known function, the entire function is disassembled; otherwise, a "window" of instructions before and after the specified address is printed in order to provide context. By default, instructions are read from the target's virtual address space; if the `–f` option is present, instructions are read from the target's object file instead. The `–w` option can be used to force "window"-mode, even if the address is the start of a known function. The size of the window defaults to ten instructions; the number of instructions can be specified explicitly using the `–n` option.

`::disasms`

List the available disassembler modes. When a target is initialized, MDB attempts to select the appropriate disassembler mode. The user can change the mode to any of the modes listed using the `::dismode` dcmd.

`::dismode` **[** ***mode*** **]**
`$V` **[** ***mode*** **]**

Get or set the disassembler mode. If no argument is specified, print the current disassembler mode. If a *mode* argument is specified, switch the disassembler to the specified mode. The list of available disassemblers can be displayed using the `::disasms` dcmd.

`::dmods` **[** `-l` **] [** ***module-name*** **]**

List the loaded debugger modules. If the `-l` option is specified, the list of the dcmds and walkers associated with each dmod is printed below its name. The output can be restricted to a particular dmod by specifying its name as an additional argument.

`::dump`

Print a hexadecimal and ASCII memory dump of the 16-byte aligned region of virtual memory containing the address specified by dot. If a repeat count is specified for `::dump`, this is interpreted as a number of bytes to dump rather than a number of iterations.

`::echo` **[** ***string*** | ***value ...*** **]**

Print the arguments separated by blanks and terminated by a `NEWLINE` to standard output. Expressions enclosed in `$[ ]` will be evaluated to a value and printed in the default base.

`::eval` ***command***

Evaluate and execute the specified string as a command. If the command contains metacharacters or white space, it should be enclosed in double or single quotes.

`::files`
`$f`

Print a list of the known source files (symbols of type `STT_FILE` present in the various target symbol tables).

`::fpregs`
`$x`, `$X`, `$y`, `$Y`

Print the floating-point register set of the representative thread.

`::formats`

List the available output format characters for use with the `/`, `\`, `?`, and `=` formatting dcmds. The formats and their use is described in “Formatting dcmds” on page 31.

`::grep` ***command***

Evaluate the specified command string, then print the old value of dot if the new value of dot is non-zero. If the *command* contains white space or metacharacters, it must be quoted. The `::grep` dcmd can be used in pipelines to filter a list of addresses.

`::help` **[** ***dcmd-name*** **]**

With no arguments, the `::help` dcmd prints a brief overview of the help facilities available in `mdb`. If a *dcmd-name* is specified, MDB prints a usage summary for that dcmd.

`::load` ***module-name***

Load the specified dmod. The module name can be given as an absolute or relative path. If *module-name* is a simple name (that is, does not contain a '/'), MDB searches for it in the module library path. Modules with conflicting names cannot be loaded; the existing module must be unloaded first.

`::log` **[** `–d` | **[** `–e` **]** ***filename*** **]**
`$>` **[** ***filename*** **]**

Enable or disable the output log. MDB provides an interactive logging facility where both the input commands and standard output can be logged to a file while still interacting with the user. The `–e` option enables logging to the specified file, or re-enables logging to the previous log file if no file name is given. The `–d` option disables logging. If the `$>` dcmd is used, logging is enabled if a file name argument is specified; otherwise, logging is disabled. If the specified log file already exists, MDB appends any new log output to the file.

`::map` ***command***

Map the value of dot to a corresponding value using the *command* specified as a string argument, then print the new value of dot. If the command contains white space or metacharacters, it must be quoted. The `::map` dcmd can be used in pipelines to transform the list of addresses into a new list of addresses.

[***address*** **]** `::mappings` **[** ***name*** **]**
[***address*** **]** `$m` **[** ***name*** **]**

Print a list of each mapping in the target's virtual address space, including the address, size, and description of each mapping. If the dcmd is preceded by an *address*, MDB shows only the mapping that contains the given address. If a string *name* argument is given, MDB shows only the mapping that matched the description.

`::nm` **[** `–DPdghnopuvx` **] [** ***object*** **]**

Print the symbol tables associated with the current target. If an *object* name argument is specified, only the symbol table for this load object is displayed. The `::nm` dcmd also recognizes the following options:

`–D` Print `.dynsym` (dynamic symbol table) instead of `.symtab`.

–P Print the private symbol table instead of `.symtab`.

–d Print value and size fields in decimal.

–g Print only global symbols.

–h Suppress the header line.

–n Sort symbols by name.

–o Print value and size fields in octal.

–p Print symbols as a series of `::nmadd` commands. This option can be used with –P to produce a macro file that can be subsequently read into the debugger with `$<`.

–u Print only undefined symbols.

–v Sort symbols by value.

–x Print value and size fields in hexadecimal.

value `::nmadd` **[** –fo **] [** –e ***end*** **] [** –s ***size*** **]** ***name***

Add the specified symbol *name* to the private symbol table. MDB provides a private, configurable symbol table that can be used to interpose on the target's symbol table, as described in "Symbol Name Resolution" on page 29. The `::nmadd` dcmd also recognizes the following options:

–e Set the size of the symbol to *end* - *value*.

–f Set the type of the symbol to `STT_FUNC`.

–o Set the type of the symbol to `STT_OBJECT`.

–s Set the size of the symbol to *size*.

`::nmdel` ***name***

Delete the specified symbol *name* from the private symbol table.

`::objects`

Print a map of the target's virtual address space, showing only those mappings that correspond to the primary mapping (usually the text section) of each of the known load objects.

```
::quit
$q
```

Quit the debugger.

```
::regs
$r
```

Print the general-purpose register set of the representative thread.

```
::release
:R
```

Release the previously attached process or core file.

`::set` **[** –wF **] [** +/-o ***option*** **] [** – s ***distance*** **] [** –I ***path*** **] [** –L ***path*** **] [** –P ***prompt*** **]**

Get or set miscellaneous debugger properties. If no options are specified, the current set of debugger properties is displayed. The `::set` dcmd recognizes the following options:

–F Forcibly take over the next user process that `::attach` is applied to, as if `mdb` had been executed with the –F option on the command line.

–I Set the default path for locating macro files. The path argument can contain any of the special tokens described for the –I command-line option in Appendix A.

–L Set the default path for locating debugger modules. The path argument can contain any of the special tokens described for the –I command-line option in Appendix A.

–o Enable the specified debugger option. If the +o form is used, the option is disabled. The option strings are described along with the –o command-line option in Appendix A.

–P Set the command prompt to the specified prompt string.

–s Set the symbol matching distance to the specified distance. Refer to the description of the –s command-line option in Appendix A for more information.

–w Re-open the target for writing, as if `mdb` had been executed with the –w option on the command line.

[***address*** **]** `::stack` **[** ***count*** **]**
[***address*** **]** `$c` **[** ***count*** **]**

Print a C stack back trace. If the dcmd is preceded by an explicit *address*, a back trace beginning at this virtual memory address is displayed. Otherwise, the stack of the representative thread is displayed. If an optional count value is given as an argument, no more than *count* arguments are displayed for each stack frame in the output.

For 64-bit systems (SPARC) - The biased frame pointer value (that is, the virtual address plus `0x7ff`) should be used as the address when requesting a stack trace.

`::status`

Print a summary of information related to the current target.

`::typeset` **[**`+/-t`**]** ***variable-name ...***

Set attributes for named variables. If one or more variable names are specified, they are defined and set to the value of dot. If the `–t` option is present, the user-defined tag associated with each variable is set. If the `+t` option is present, the tag is cleared. If no variable names are specified, the list of variables and their values is printed.

`::unload` ***module-name***

Unload the specified dmod. The list of active dmods can be printed using the `::dmods` dcmd. Built-in modules cannot be unloaded. Modules that are busy (that is, provide dcmds that are currently executing) cannot be unloaded.

`::unset` ***variable-name ...***

Unset (remove) the specified variable(s) from the list of defined variables. Some variables are exported by MDB are marked as persistent, and cannot be unset by the user.

`::vars` **[**`–npt`**]**

Print a listing of named variables. If the `–n` option is present, the output is restricted to variables that currently have non-zero values. If the `–p` option is present, the variables are printed in a form suitable for re-processing by the debugger using the `$<` dcmd. This option can be used to record the variables to a macro file, then restore these values later. If the `–t` option is present, only the tagged variables are printed. Variables can be tagged using the `–t` option of the `::typeset` dcmd.

`::version`

Print the debugger version number.

address `::vtop`

Print the physical address mapping for the specified virtual address, if possible. The `::vtop` dcmd is available only when examining a kernel target, or when examining a user process inside a kernel crash dump (after a `::context` dcmd has been issued).

[*address*] `::walk` ***walker-name*** **[*variable-name*]**

Walk through the elements of a data structure using the specified walker. The available walkers can be listed using the `::walkers` dcmd. Some walkers operate on a global data structure and do not require a starting address. For example, walk the list of proc structures in the kernel.

Other walkers operate on a specific data structure whose address must be specified explicitly. For example, given a pointer to an address space, walk the list of segments.

When used interactively, the `::walk` dcmd will print the address of each element of the data structure in the default base. The dcmd can also be used to provide a list of addresses for a pipeline. The walker name can use the backquote " ` " scoping operator described in "dcmd and Walker Name Resolution" on page 30. If the optional *variable-name* is specified, the specified variable will be assigned the value returned at each step of the walk when MDB invokes the next stage of the pipeline.

`::walkers`

List the available walkers and print a brief description for each one.

`::whence` **[**–v**]** ***name ...***
`::which` **[**–v**]** ***name ...***

Print the dmod that exports the specified dcmds and walkers. These dcmds can be used to determine which dmod is currently providing the global definition of the given dcmd or walker. Refer to "dcmd and Walker Name Resolution" on page 30 for more information on global name resolution. The –v option causes the dcmd to print the alternate definitions of each dcmd and walker in order of precedence.

`::xdata`

List the external data buffers exported by the current target. External data buffers represent information associated with the target that cannot be accessed through standard target facilities (that is, an address space, symbol table, or register set). These buffers can be consumed by dcmds; for more information, refer to "`mdb_get_xdata()`" on page 112.

CHAPTER 5

Kernel Debugging Modules

This chapter describes the debugger modules, dcmds, and walkers provided to debug the Solaris kernel. Each kernel debugger module is named after the corresponding Solaris kernel module, so that it will be loaded automatically by MDB. The facilities described here reflect the current Solaris kernel implementation and are subject to change in the future; writing shell scripts that depend on the output of these commands is not recommended. In general, the kernel debugging facilities described in this chapter are meaningful only in the context of the corresponding kernel subsystem implementation. See “Related Books and Papers” on page for a list of references that provide more information about the Solaris kernel implementation.

Note - This guide reflects the Solaris 8 operating environment implementation; these modules, dcmds, and walkers may not be relevant, correct, or applicable to past or future releases, since they reflect the current kernel implementation. They do not define a permanent public interface of any kind. All of the information provided about modules, dcmds, walkers, and their output formats and arguments is subject to change in future releases of the Solaris operating environment.

Generic Kernel Debugging Support (`genunix`)

Kernel Memory Allocator

This section discusses the dcmds and walkers used to debug problems identified by the Solaris kernel memory allocator and to examine memory and memory usage. The dcmds and walkers described here are discussed in more detail in Chapter 6.

Dcmds

thread `::allocdby`

Given the address of a kernel thread, print a list of memory allocations it has performed in reverse chronological order.

bufctl `::bufctl` **[**–a ***address***] **[**–c ***caller***] **[**–e ***earliest***] **[**–l ***latest***] **[**–t ***thread***]

Print a summary of the *bufctl* information for the specified bufctl *address*. If one or more options are present, the bufctl information is printed only if it matches the criteria defined by the option arguments; in this way, the dcmd can be used as a filter for input from a pipeline. The –a option indicates that the bufctl's corresponding buffer address must equal the specified address. The –c option indicates that a program counter value from the specified caller must be present in the bufctl's saved stack trace. The –e option indicates that the bufctl's timestamp must be greater than or equal to the specified earliest timestamp. The –l option indicates that the bufctl's timestamp must be less than or equal to the specified latest timestamp. The –t option indicates that the bufctl's thread pointer must be equal to the specified thread address.

[***address*** **]** `::findleaks` **[**–v**]**

The `::findleaks` dcmd provides powerful and efficient detection of memory leaks in kernel crash dumps where 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.

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.

thread `::freedby`

Given the address of a kernel thread, print a list of memory frees it has performed, in reverse chronological order.

value `::kgrep`

Search the kernel address space for pointer-aligned addresses that contain the specified pointer-sized value. The list of addresses that contain matching values is then printed. Unlike MDB's built-in search operators, `::kgrep` searches every segment of the kernel's address space and searches across discontiguous segment boundaries. On large kernels, `::kgrep` can take a considerable amount of time to execute.

`::kmalog` **[** `slab | fail` **]**

Display events in a kernel memory allocator transaction log. Events are displayed in time-reverse order, with the most recent event displayed first. For each event, `::kmalog` displays the time relative to the most recent event in T-minus notation (for example, T-0.000151879), the bufctl, the buffer address, the kmem cache name, and the stack trace at the time of the event. Without arguments, `::kmalog` displays the kmem transaction log, which is present only if `KMF_AUDIT` is set in `kmem_flags`. `::kmalog fail` displays the allocation failure log, which is always present; this can be useful in debugging drivers that don't cope with allocation failure correctly. `::kmalog slab` displays the slab create log, which is always present. `::kmalog slab` can be useful when searching for memory leaks.

`::kmastat`

Display the list of kernel memory allocator caches and virtual memory arenas, along with corresponding statistics.

`::kmausers` **[**–`ef`**] [***cache* **...]**

Print information about the medium and large users of the kernel memory allocator that have current memory allocations. The output consists of one entry for each unique stack trace specifying the total amount of memory and number of allocations that was made with that stack trace. This dcmd requires that the `KMF_AUDIT` flag is set in `kmem_flags`.

If one or more cache names (for example, `kmem_alloc_256`) are specified, the scan of memory usage is restricted to those caches. By default all caches are included. If the –`e` option is used, the small users of the allocator are included. The small users are allocations that total less than 1024 bytes of memory or for which there are less than 10 allocations with the same stack trace. If the –`f` option is used, the stack traces are printed for each individual allocation.

[***address*** **]** `::kmem_cache`

Format and display the `kmem_cache` structure stored at the specified address, or the complete set of active `kmem_cache` structures.

`::kmem_log`

Display the complete set of kmem transaction logs, sorted in reverse chronological order. This dcmd uses a more concise tabular output format than `::kmalog`.

[***address*** **]** `::kmem_verify`

Verify the integrity of the `kmem_cache` structure stored at the specified address, or the complete set of active `kmem_cache` structures. If an explicit cache address is

specified, the dcmd displays more verbose information regarding errors; otherwise, a summary report is displayed. The `::kmem_verify` dcmd is discussed in more detail in “Kernel Memory Caches” on page 71.

[*address*] `::vmem`

Format and display the `vmem` structure stored at the specified address, or the complete set of active `vmem` structures. This structure is defined in `<sys/vmem_impl.h>`.

address `::vmem_seg`

Format and display the `vmem_seg` structure stored at the specified address. This structure is defined in `<sys/vmem_impl.h>`.

address `::whatis` **[**`–abv`**]**

Report information about the specified address. In particular, `::whatis` will attempt 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 report its findings. If the `–a` option is present, the dcmd reports all matches instead of just the first match to its queries. If the `–b` option is present, the dcmd also attempts to determine if the address is referred to by a known kmem `bufctl`. If the `–v` option is present, the dcmd reports its progress as it searches various kernel data structures.

Walkers

allocdby — Given the address of a `kthread_t` structure as a starting point, iterate over the set of `bufctl` structures corresponding to memory allocations performed by this kernel thread.

bufctl — Given the address of a `kmem_cache_t` structure as a starting point, iterate over the set of allocated `bufctl`s associated with this cache.

freectl — Given the address of a `kmem_cache_t` structure as a starting point, iterate over the set of free `bufctl`s associated with this cache.

freedby — Given the address of a `kthread_t` structure as a starting point, iterate over the set of `bufctl` structures corresponding to memory deallocations performed by this kernel thread.

freemem	Given the address of a `kmem_cache_t` structure as a starting point, iterate over the set of free buffers associated with this cache.
kmem	Given the address of a `kmem_cache_t` structure as a starting point, iterate over the set of allocated buffers associated with this cache.
kmem_cache	Iterate over the active set of `kmem_cache_t` structures. This structure is defined in `<sys/kmem_impl.h>`.
kmem_cpu_cache	Given the address of a `kmem_cache_t` structure as a starting point, iterate over the per-CPU `kmem_cpu_cache_t` structures associated with this cache. This structure is defined in `<sys/kmem_impl.h>`.
kmem_slab	Given the address of a `kmem_cache_t` structure as a starting point, iterate over the set of associated `kmem_slab_t` structures. This structure is defined in `<sys/kmem_impl.h>`.
kmem_log	Iterate over the set of `bufctl`s stored in the kmem allocator transaction log.

File Systems

The MDB file systems debugging support includes a built-in facility to convert vnode pointers to the corresponding file system path name. This conversion is performed using the Directory Name Lookup Cache (DNLC); because the cache does not hold all active vnodes, some vnodes might not be able to be converted to path names and "??" is displayed instead of a name.

dcmds

`::fsinfo`

Display a table of mounted file systems, including the `vfs_t` address, `ops` vector, and mount point of each file system.

`::lminfo`

Display a table of vnodes with active network locks registered with the lock manager. The pathname corresponding to each vnode is shown.

address `::vnode2path` **[**–v**]**

Display the pathname corresponding to the given vnode address. If the –v option is specified, the dcmd prints a more verbose display, including the vnode pointer of each intermediate path component.

Walkers

buf — Iterate over the set of active block I/O transfer structures (`buf_t` structures). The `buf` structure is defined in `<sys/buf.h>` and is described in more detail in `buf`(9S).

Virtual Memory

This section describes the debugging support for the kernel virtual memory subsystem.

dcmds

address `::addr2smap` **[*offset*]**

Print the `smap` structure address that corresponds to the given address in the kernel's segmap address space segment.

as `::as2proc`

Display the `proc_t` address for the process corresponding to the `as_t` address *as*.

seg `::seg`

Format and display the specified address space segment (`seg_t` address).

vnode `::vnode2smap` **[*offset*]**

Print the `smap` structure address that corresponds to the given `vnode_t` address and offset.

Walkers

anon — Given the address of an `anon_map` structure as a starting point, iterate over the set of related anon structures. The anon map implementation is defined in `<vm/anon.h>`.

seg — Given the address of an `as_t` structure as a starting point, iterate over the set of address space segments (`seg` structures) associated with the specified address space. The `seg` structure is defined in `<vm/seg.h>`.

CPUs and the Dispatcher

Thsi section describes the facilities for examining the state of the cpu structures and the kernel dispatcher.

dcmds

`::callout`

Display the callout table. The function, argument, and expiration time for each callout is displayed.

`::class`

Display the scheduling class table.

[***cpuid*** **]** `::cpuinfo` **[**–v**]**

Display a table of the threads currently executing on each CPU. If an optional CPU ID number is specified prior to the dcmd name, only the information for the specified CPU is displayed. If the –v option is present, `::cpuinfo` also displays the runnable threads waiting to execute on each CPU as well as the active interrupt threads.

Walkers

cpu — Iterate over the set of kernel CPU structures. The `cpu_t` structure is defined in `<sys/cpuvar.h>`.

Device Drivers and DDI Framework

This section describes dcmds and walkers that are useful for kernel developers as well as third-party device driver developers.

dcmds

`::devbindings` ***device-name***

Display the list of all instances of the named driver. The output consists of an entry for each instance, beginning with the pointer to the struct `dev_info` (viewable with `$<devinfo` or `::devinfo`), the driver name, the instance number, and the driver and system properties associated with that instance.

address `::devinfo` **[** `–q` **]**

Print the system and driver properties associated with a devinfo node. If the `–q` option is specified, only a quick summary of the device node is shown.

[***address*** **]** `::devnames` **[** `–v` **]**

Display the kernel's `devnames` table along with the `dn_head` pointer, which points at the driver instance list. If the `–v` flag is specified, additional information stored at each entry in the `devnames` table is displayed.

[***devinfo*** **]** `::prtconf` **[** `–cpv` **]**

Display the kernel device tree starting at the device node specified by *devinfo*. If *devinfo* is not provided, the root of the device tree is assumed by default. If the `–c` option is specified, only children of the given device node are displayed. If the `–p` option is specified, only ancestors of the given device node are displayed. If `–v` is specified, the properties associated with each node are displayed.

[***major-num*** **]** `::major2name` **[** ***major-num*** **]**

Display the driver name corresponding to the specified major number. The major number can be specified as an expression preceding the dcmd or as a command-line argument.

[***address*** **]** `::modctl2devinfo`

Print all of the device nodes that correspond to the specified `modctl` address.

`::name2major` ***driver-name***

Given a device driver name, display its major number.

[*address*] `::softstate` **[*instance-number*]**

Given a softstate state pointer (see `ddi_soft_state_init`(9F)) and a device instance number, display the soft state for that instance.

Walkers

devinfo — First, iterate over the parents of the given devinfo and return them in order of seniority from most to least senior. Second, return the given devinfo itself. Third, iterate over the children of the given devinfo in order of seniority from most to least senior. The `dev_info` struct is defined in `<sys/ddi_impldefs.h>`.

devinfo_children — First, return the given devinfo, then iterate over the children of the given devinfo in order of seniority from most to least senior. The `dev_info` struct is defined in `<sys/ddi_impldefs.h>`.

devinfo_parents — Iterate over the parents of the given devinfo in order of seniority from most to least senior, and then return the given devinfo. The `dev_info` struct is defined in `<sys/ddi_impldefs.h>`.

devi_next — Iterate over the siblings of the given devinfo. The `dev_info` struct is defined in `<sys/ddi_impldefs.h>`.

devnames — Iterate over the entries in the devnames array. This structure is defined in `<sys/autoconf.h>`.

STREAMS

This section describes dcmds and walkers that are useful for kernel developers as well as developers of third-party STREAMS modules and drivers.

address `::queue` **[**`–v`**] [**`–f` ***flag***] [**`–F` ***flag***] [**`–m` ***modname***]**

Filter and display the specified `queue_t` data structure. With no options, various properties of the `queue_t` are shown. If the `–v` option is present, the queue flags are decoded in greater detail. If the `–f`, `–F`, or `–m` options are present, the queue is displayed only if it matches the criteria defined by the arguments to these options; in this way, the dcmd can be used as a filter for input from a pipeline. The `–f` option indicates that the specified flag (one of the Q flag names from `<sys/stream.h>`) must be present in the queue flags. The `–F` option indicates that the specified flag must be absent from the queue flags. The `–m` option indicates that the module name associated with the queue must match the specified modname.

address `::q2syncq`

Given the address of a `queue_t`, print the address of the corresponding `syncq_t` data structure.

address `::q2otherq`

Given the address of a `queue_t`, print the address of the peer read or write queue structure.

address `::q2rdq`

Given the address of a `queue_t`, print the address of the corresponding read queue.

address `::q2wrq`

Given the address of a `queue_t`, print the address of the corresponding write queue.

[***address*** **]** `::stream`

Display a visual picture of a kernel STREAM data structure, given the address of the `stdata_t` structure representing the STREAM head. The read and write queue pointers, byte count, and flags for each module are shown, and in some cases additional information for the specific queue is shown in the margin.

address `::syncq` **[**`–v`**] [**`–f` ***flag***] [**`–F` ***flag***] [**`–t` ***type***] [**`–T` ***type***]**

Filter and display the specified `syncq_t` data structure. With no options, various properties of the `syncq_t` are shown. If the `–v` option is present, the syncq flags are decoded in greater detail. If the `–f`, `–F`, `–t`, or `–T` options are present, the syncq is displayed only if it matches the criteria defined by the arguments to these options; in this way, the dcmd can be used as a filter for input from a pipeline. The `–f` option

indicates that the specified flag (one of the SQ_ flag names from <sys/strsubr.h>) must be present in the syncq flags. The –F option indicates that the specified flag must be absent from the syncq flags. The –t option indicates that the specified type (one of the SQ_CI or SQ_CO type names from <sys/strsubr.h>) must be present in the syncq type bits. The –T option indicates that the specified type must be absent from the syncq type bits.

address ::syncq2q

Given the address of a syncq_t, print the address of the corresponding queue_t data structure.

Walkers

qlink	Given the address of a queue_t structure, walk the list of related queues using the q_link pointer. This structure is defined in <sys/stream.h>.
qnext	Given the address of a queue_t structure, walk the list of related queues using the q_next pointer. This structure is defined in <sys/stream.h>.
readq	Given the address of an stdata_t structure, walk the list of read-side queue structures.
writeq	Given the address of an stdata_t structure, walk the list of write-side queue structures.

Files, Processes, and Threads

This section describes dcmds and walkers used to format and examine various fundamental file, process, and thread structures in the Solaris kernel.

dcmds

process ::fd ***fd-num***

Print the file_t address corresponding to the file descriptor *fd-num* associated with the specified process. The process is specified using the virtual address of its proc_t structure.

thread `::findstack` **[** ***command*** **]**

Print the stack trace associated with the given kernel thread, identified by the virtual address of its `kthread_t` structure. The dcmd employs several different algorithms to locate the appropriate stack backtrace. If an optional command string is specified, the dot variable is reset to the frame pointer address of the topmost stack frame, and the specified command is evaluated as if it had been typed at the command line. The default command string is "`<.$C0`"; that is, print a stack trace including frame pointers but no arguments.

pid `::pid2proc`

Print the `proc_t` address corresponding to the specified PID. Recall that MDB's default base is hexadecimal, so decimal PIDs obtained using `pgrep`(1) or `ps`(1) should be prefixed with `0t`.

process `::pmap`

Print the memory map of the process indicated by the given process address. The dcmd displays output using a format similar to `pmap`(1).

[***process*** **]** `::ps` **[**`–flt`**]**

Print a summary of the information related to the specified process, or all active system processes, similar to `ps`(1). If the `–f` option is specified, the full command name and initial arguments are printed. If the `–l` option is specified, the LWPs associated with each process are printed. If the `–t` option is specified, the kernel threads associated with each process LWP are printed.

`::ptree`

Print a process tree, with child processes indented from their respective parent processes. The dcmd displays output using a format similar to `ptree`(1).

vnode `::whereopen`

Given a `vnode_t` address, print the `proc_t` addresses of all processes that have this vnode currently open in their file table.

Walkers

file — Given the address of a `proc_t` structure as a starting point, iterate over the set of open files (`file_t` structures) associated with the specified

process. The `file_t` structure is defined in `<sys/file.h>`.

proc — Iterate over the active process (`proc_t`) structures. This structure is defined in `<sys/proc.h>`.

thread — Iterate over a set of kernel thread (`kthread_t`) structures. If the global walk is invoked, all kernel threads are returned by the walker. If a local walk is invoked using a `proc_t` address as the starting point, the set of threads associated with the specified process is returned. The `kthread_t` structure is defined in `<sys/thread.h>`.

Synchronization Primitives

This section describes dcmds and walkers used to examine particular kernel synchronization primitives. The semantics of each primitive are discussed in the corresponding *man pages section 9F: DDI and DKI Kernel Functions.*

dcmds

rwlock `::rwlock`

Given the address of a readers-writers lock (see `rwlock`(9F)), display the current state of the lock and the list of waiting threads.

[***address*** **]** `::wchaninfo` **[**–v**]**

Given the address of a condition variable (see `condvar`(9F)) or semaphore (see `semaphore`(9F)), display the current number of waiters on this object. If no explicit address is specified, display all such objects that have waiting threads. If the –v option is specified, display the list of threads that are blocked on each object.

Walkers

blocked — Given the address of a synchronization object (such as a `mutex`(9F) or `rwlock`(9F)), iterate over the list of blocked kernel threads.

wchan — Given the address of a condition variable (see `condvar`(9F)) or semaphore (see `semaphore`(9F)), iterate over the list of blocked kernel threads.

Cyclics

The cyclic subsystem is a low-level kernel subsystem that provides high resolution, per-CPU interval timer facilities to other kernel services and programming interfaces.

dcmds

`::cycinfo` **[**–vV**]**

Display the cyclic subsystem per-CPU state for each CPU. If the –v option is present, a more verbose display is shown. If the –V option is present, an even more verbose display than –v is shown.

address `::cyclic`

Format and display the `cyclic_t` at the specified address.

`::cyccover`

Display cyclic subsystem code coverage information. This information is available only in a DEBUG kernel.

`::cyctrace`

Display cyclic subsystem trace information. This information is available only in a DEBUG kernel.

Walkers

cyccpu — Iterate over the per-CPU `cyc_cpu_t` structures. This structure is defined in `<sys/cyclic_impl.h>`.

cyctrace — Iterate over the cyclic trace buffer structures. This information is only available in a DEBUG kernel.

Interprocess Communication Debugging Support (ipc)

The ipc module provides debugging support for the implementation of the message queue, semaphore, and shared memory interprocess communication primitives.

dcmds

::ipcs **[**–l**]**

Display a listing of system-wide IPC identifiers, corresponding to known message queues, semaphores, and shared memory segments. If the –l option is specified, a longer listing of information is shown.

[***address*** **]** ::msqid_ds **[**–l**]**

Print the specified msqid_ds structure or a table of the active msqid_ds structures (message queue identifiers). If the –l option is specified, a longer listing of information is displayed.

[***address*** **]** ::semid_ds **[**–l**]**

Print the specified semid_ds structure or a table of the active semid_ds structures (semaphore identifiers). If the –l option is specified, a longer listing of information is displayed.

[***address*** **]** ::shmid_ds **[**–l**]**

Print the specified shmid_ds structure or a table of the active shmid_ds structures (shared memory segment identifiers). If the –l option is specified, a longer listing of information is displayed.

Walkers

msg — Walk the active msqid_ds structures corresponding to message queue identifiers. This structure is defined in <sys/msg.h>.

sem	Walk the active `semid_ds` structures corresponding to semaphore identifiers. This structure is defined in `<sys/sem.h>`.
shm	Walk the active `shmid_ds` structures corresponding to shared memory segment identifiers. This structure is defined in `<sys/shm.h>`.

Loopback File System Debugging Support (`lofs`)

The `lofs` module provides debugging support for the `lofs`(7FS) file system.

dcmds

[*address*] `::lnode`

Print the specified `lnode_t`, or a table of the active `lnode_t` structures in the kernel.

address `::lnode2dev`

Print the `dev_t` (`vfs_dev`) for the underlying loopback mounted filesystem corresponding to the given `lnode_t` address.

address `::lnode2rdev`

Print the `dev_t` (`li_rdev`) for the underlying loopback mounted file system corresponding to the given `lnode_t` address.

Walkers

lnode	Walk the active `lnode_t` structures in the kernel. This structure is defined in `<sys/fs/lofs_node.h>`.

Internet Protocol Module Debugging Support (ip)

The ip module provides debugging support for the ip(7P) driver

dcmds

[*address*] ::ire **[**–q**]**

Print the specified ire_t, or a table of the active ire_t structures in the kernel. If the –q flag is specified, the send and receive queue pointers are printed instead of the source and destination addresses.

Walkers

ire — Walk the active ire (Internet Route Entry) structures in the kernel. This structure is defined in <inet/ip.h>.

Kernel Runtime Link Editor Debugging Support (krtld)

This section describes the debugging support for the kernel runtime link editor, which is responsible for loading kernel modules and drivers.

dcmds

[*address*] ::modctl

Print the specified modctl, or a table of the active modctl structures in the kernel.

address `::modhdrs`

Given the address of a `modctl` structure, print the module's ELF executable header and section headers.

`::modinfo`

Print information about the active kernel modules, similar to the output of the `/usr/sbin/modinfo` command.

Walkers

modctl — Walk the list of active modctl structures in the kernel. This structure is defined in `<sys/modctl.h>`.

IA: Platform Debugging Support (`unix`)

These dcmds and walkers are specific to IA.

dcmds

[***cpuid*** **|** ***address*** **]** `::ttrace` **[**–x**]**

Display trap trace records in reverse chronological order. The trap trace facility is available only in DEBUG kernels. If an explicit dot value is specified, this is interpreted as either a CPU ID number or a trap trace record address, depending on the precise value. If a CPU ID is specified, the output is restricted to the buffer from that CPU. If a record address is specified, only that record is formatted. If the –x option is specified, the complete raw record is displayed.

Walkers

ttrace — Walk the list of trap trace record addresses in reverse chronological order. The trap trace facility is available only in DEBUG kernels.

SPARC: sun4d Platform Debugging Support (unix)

These dcmds and walkers are specific to the SPARC sun4d platform.

dcmds

[***cpuid*** **|** ***address*** **]** `::ttrace` **[**–x**]**

Display trap trace records in reverse chronological order. The trap trace facility is available only in DEBUG kernels. If an explicit dot value is specified, this is interpreted as either a CPU ID number or a trap trace record address, depending on the precise value. If a CPU ID is specified, the output is restricted to the buffer from that CPU. If a record address is specified, only that record is formatted. If the –x option is specified, the complete raw record is displayed.

Walkers

ttrace — Walk the list of trap trace record addresses in reverse chronological order. The trap trace facility is available only in DEBUG kernels.

SPARC: sun4m Platform Debugging Support (unix)

These dcmds and walkers are specific to the SPARC sun4m platform.

dcmds

[***cpuid*** **|** ***address*** **]** `::ttrace` **[**–x**]**

Display trap trace records in reverse chronological order. The trap trace facility is available only in DEBUG kernels. If an explicit dot value is specified, this is

interpreted as either a CPU ID number or a trap trace record address, depending on the precise value. If a CPU ID is specified, the output is restricted to the buffer from that CPU. If a record address is specified, only that record is formatted. If the –x option is specified, the complete raw record is displayed.

Walkers

ttrace — Walk the list of trap trace record addresses in reverse chronological order. The trap trace facility is only available in DEBUG kernels.

SPARC: sun4u Platform Debugging Support (unix)

These dcmds and walkers are specific to the SPARC sun4u platform.

dcmds

[***address*** **]** `::softint`

Display the soft interrupt vector structure at the specified address, or display all the active soft interrupt vectors. The pending count, PIL, argument, and handler function for each structure is displayed.

`::ttctl`

Display trap trace control records. The trap trace facility is available only in DEBUG kernels.

[***cpuid*** **]** `::ttrace` **[**–x**]**

Display trap trace records in reverse chronological order. The trap trace facility is available only in DEBUG kernels. If an explicit dot value is specified, this is interpreted as a CPU ID number, and the output is restricted to the buffer from that CPU. If the –x option is specified, the complete raw record is displayed.

[*address*] `::xc_mbox`

Display the cross-call mailbox at the specified address, or format all the cross-call mailboxes that have pending requests.

`::xctrace`

Format and display trace trace records in reverse chronological order that are related to CPU cross-call activity. The trap trace facility is available only in DEBUG kernels.

Walkers

softint	Iterate over the soft interrupt vector table entries.
ttrace	Iterate over the trap trace record addresses in reverse chronological order. The trap trace facility is only available in DEBUG kernels.
xc_mbox	Iterate over the mailboxes used for CPU handshake and cross-call (x-call) requests.

CHAPTER 6

Debugging With the Kernel Memory Allocator

The Solaris kernel memory (kmem) allocator provides a powerful set of debugging features that can facilitate analysis of a kernel crash dump. This chapter discusses these debugging features, and the MDB dcmds and walkers designed specifically for the allocator. Bonwick (see “Related Books and Papers” on page) provides an overview of the principles of the allocator itself. Refer to the header file `<sys/kmem_impl.h>` for the definitions of allocator data structures. The kmem debugging features can be enabled on a production system to enhance problem analysis, or on development systems to aid in debugging kernel software and device drivers.

Note - This guide reflects Solaris 8 implementation; this information might not be relevant, correct, or applicable to past or future releases, since it reflects the current kernel implementation. It does not define a public interface of any kind. All of the information provided about the kernel memory allocator is subject to change in future Solaris releases.

Getting Started: Creating a Sample Crash Dump

This section shows you how to obtain a sample crash dump, and how to invoke MDB in order to examine it.

Setting kmem_flags

The kernel memory allocator contains many advanced debugging features, but these are not enabled by default because they can cause performance degradation. In order to follow the examples in this guide, you should turn on these features. You should enable these features only on a test system, as they can cause performance degradation or expose latent problems.

The allocator's debugging functionality is controlled by the kmem_flags tunable. To get started, make sure kmem_flags is set properly:

```
# mdb -k
> kmem_flags/X
kmem_flags:
kmem_flags:     f
```

If kmem_flags is not set to 'f', you should add the line:

```
set kmem_flags=0xf
```

to /etc/system and reboot the system. When the system reboots, confirm that kmem_flags is set to 'f'. Remember to remove your /etc/system modifications before returning this system to production use.

Forcing a Crash Dump

The next step is to make sure crash dumps are properly configured. First, confirm that dumpadm is configured to save kernel crash dumps and that savecore is enabled. See dumpadm(1M) for more information on crash dump parameters.

```
# dumpadm
        Dump content: kernel pages
         Dump device: /dev/dsk/c0t0d0s1 (swap)
  Savecore directory: /var/crash/testsystem
    Savecore enabled: yes
```

Next, reboot the system using the '-d' flag to reboot(1M), which forces the kernel to panic and save a crash dump.

```
# reboot -d
Sep 28 17:51:18 testsystem reboot: rebooted by root

panic[cpu0]/thread=70aacde0: forced crash dump initiated at user request

401fbb10 genunix:uadmin+55c (1, 1, 0, 6d700000, 5, 0)
  %l0-7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000
...
```

When the system reboots, make sure the crash dump succeeded:

```
$ cd /var/crash/testsystem
$ ls
bounds    unix.0    unix.1    vmcore.0  vmcore.1
```

If the dump is missing from your dump directory, it could be that the partition is out of space. You can free up space and run `savecore`(1M) manually as root to subsequently save the dump. If your dump directory contains multiple crash dumps, the one you just created will be the `unix.[n]` and `vmcore.[n]` pair with the most recent modification time.

Starting MDB

Now, run `mdb` on the crash dump you created, and check its status:

```
$ mdb unix.1 vmcore.1
Loading modules: [ unix krtld genunix ip nfs ipc ]
> ::status
debugging crash dump vmcore.1 (32-bit) from testsystem
operating system: 5.8 Generic (sun4u)
panic message: forced crash dump initiated at user request
```

In the examples presented in this guide, a crash dump from a 32-bit kernel is used. All of the techniques presented here are applicable to a 64-bit kernel, and care has been taken to distinguish pointers (sized differently on 32- and 64-bit systems) from fixed-sized quantities, which are invariant with respect to the kernel data model.

A Sun Ultra-1 workstation was used to generate the example presented. Your results can vary depending on the architecture and model of system you use.

Allocator Basics

The kernel memory allocator's job is to parcel out regions of virtual memory to other kernel subsystems (these are commonly called *clients*). This section explains the basics of the allocator's operation and introduces some terms used later in this guide.

Buffer States

The functional domain of the kernel memory allocator is the set of *buffers* of virtual memory that make up the kernel heap. These buffers are grouped together into sets of uniform size and purpose, known as *caches*. Each cache contains a set of buffers. Some of these buffers are currently *free*, which means that they have not yet been allocated to any client of the allocator. The remaining buffers are *allocated*, which means that a pointer to that buffer has been provided to a client of the allocator. If no

client of the allocator holds a pointer to an allocated buffer, this buffer is said to be *leaked*, because it cannot be freed. Leaked buffers indicate incorrect code that is wasting kernel resources.

Transactions

A kmem *transaction* is a transition on a buffer between the allocated and free states. The allocator can verify that the state of a buffer is valid as part of each transaction. Additionally, the allocator has facilities for logging transactions for post-mortem examination.

Sleeping and Non-Sleeping Allocations

Unlike the Standard C Library's `malloc`(3C) function, the kernel memory allocator can block (or *sleep*), waiting until enough virtual memory is available to satisfy the client's request. This is controlled by the 'flag' parameter to `kmem_alloc`(9F). A call to `kmem_alloc`(9F), which has the `KM_SLEEP` flag set, can never fail; it will block forever waiting for resources to become available.

Kernel Memory Caches

The kernel memory allocator divides the memory it manages into a set of *caches*. All allocations are supplied from these caches, which are represented by the `kmem_cache_t` data structure. Each cache has a fixed *buffer size*, which represents the maximum allocation size satisfied by that cache. Each cache has a string name indicating the type of data it manages.

Some kernel memory caches are special purpose and are initialized to allocate only a particular kind of data structure. An example of this is the "thread_cache," which allocates only structures of type `kthread_t`. Memory from these caches is allocated to clients by the `kmem_cache_alloc( )` function and freed by the `kmem_cache_free( )` function.

Note - `kmem_cache_alloc( )` and `kmem_cache_free( )` are not public DDI interfaces. Do NOT write code that relies on them, because they are subject to change or removal in future releases of Solaris.

Caches whose name begins with "`kmem_alloc_`" implement the kernel's general memory allocation scheme. These caches provide memory to clients of `kmem_alloc`(9F) and `kmem_zalloc`(9F). Each of these caches satisfies requests whose size is between the buffer size of that cache and the buffer size of the next largest cache. For example, the kernel has `kmem_alloc_8` and `kmem_alloc_16`

caches. In this case, the kmem_alloc_16 cache handles all client requests for 9-16 bytes of memory. Remember that the size of each buffer in the kmem_alloc_16 cache is 16 bytes, regardless of the size of the client request. In a 14 byte request, two bytes of the resulting buffer are unused, since the request is satisfied from the kmem_alloc_16 cache.

The last set of caches are those used internally by the kernel memory allocator for its own bookkeeping. These include those caches whose names start with "kmem_magazine_" or "kmem_va_", the kmem_slab_cache, the kmem_bufctl_cache and others.

Kernel Memory Caches

This section explains how to find and examine kernel memory caches. You can learn about the various kmem caches on the system by issuing the ::kmastat command.

```
> ::kmastat
cache                        buf    buf    buf    memory     alloc alloc
name                        size in use  total    in use   succeed  fail
------------------------- ------ ------ ------ --------- --------- -----
kmem_magazine_1               8     24   1020      8192        24     0
kmem_magazine_3              16    141    510      8192       141     0
kmem_magazine_7              32     96    255      8192        96     0
...
kmem_alloc_8                  8   3614   3751     90112   9834113     0
kmem_alloc_16                16   2781   3072     98304   8278603     0
kmem_alloc_24                24    517    612     24576    680537     0
kmem_alloc_32                32    398    510     24576    903214     0
kmem_alloc_40                40    482    584     32768    672089     0
...
thread_cache                368    107    126     49152    669881     0
lwp_cache                   576    107    117     73728       182     0
turnstile_cache              36    149    292     16384    670506     0
cred_cache                   96      6     73      8192   2677787     0
...
```

If you run ::kmastat you get a feel for what a "normal" system looks like. This will help you to spot excessively large caches on systems that are leaking memory. The results of ::kmastat will vary depending on the system you are running on, how many processes are running, and so forth.

Another way to list the various kmem caches is with the ::kmem_cache command:

```
> ::kmem_cache
ADDR     NAME                       FLAG  CFLAG  BUFSIZE  BUFTOTL
70036028 kmem_magazine_1            0020 0e0000        8     1020
700362a8 kmem_magazine_3            0020 0e0000       16      510
70036528 kmem_magazine_7            0020 0e0000       32      255
...
70039428 kmem_alloc_8               020f 000000        8     3751
700396a8 kmem_alloc_16              020f 000000       16     3072
```

```
70039928 kmem_alloc_24             020f 000000       24      612
70039ba8 kmem_alloc_32             020f 000000       32      510
7003a028 kmem_alloc_40             020f 000000       40      584
...
```

This command is useful because it maps cache names to addresses, and provides the debugging flags for each cache in the FLAG column. It is important to understand that the allocator's selection of debugging features is derived on a per-cache basis from this set of flags. These are set in conjunction with the global `kmem_flags` variable at cache creation time. Setting `kmem_flags` while the system is running has no effect on the debugging behavior, except for subsequently created caches (which is rare after boot-up).

Next, walk the list of kmem caches directly using MDB's `kmem_cache` walker:

```
> ::walk kmem_cache
70036028
700362a8
70036528
700367a8
...
```

This produces a list of pointers that correspond to each kmem cache in the kernel. To find out about a specific cache, apply the `kmem_cache` macro:

```
> 0x70039928$<kmem_cache
0x70039928:     lock
0x70039928:     owner/waiters
                0
0x70039930:     flags           freelist        offset
                20f             707c86a0        24
0x7003993c:     global_alloc    global_free     alloc_fail
                523             0               0
0x70039948:     hash_shift      hash_mask       hash_table
                5               1ff             70444858
0x70039954:     nullslab
0x70039954:     cache           base            next
                70039928        0               702d5de0
0x70039960:     prev            head            tail
                707c86a0        0               0
0x7003996c:     refcnt          chunks
                -1              0
0x70039974:     constructor     destructor      reclaim
                0               0               0
0x70039980:     private         arena           cflags
                0               104444f8        0
0x70039994:     bufsize         align           chunksize
                24              8               40
0x700399a0:     slabsize        color           maxcolor
                8192            24              32
0x700399ac:     slab_create     slab_destroy    buftotal
                3               0               612
0x700399b8:     bufmax          rescale         lookup_depth
                612             1               0
0x700399c4:     kstat           next            prev
                702c8608        70039ba8        700396a8
0x700399d0:     name    kmem_alloc_24
```

```
0x700399f0:     bufctl_cache    magazine_cache  magazine_size
                70037ba8        700367a8        15
...
```

Important fields for debugging include 'bufsize', 'flags' and 'name'. The name of the `kmem_cache` (in this case "`kmem_alloc_24`") indicates its purpose in the system. Bufsize indicates the size of each buffer in this cache; in this case, the cache is used for allocations of size 24 and smaller. 'flags' indicates what debugging features are turned on for this cache. You can find the debugging flags listed in `<sys/kmem_impl.h>`. In this case 'flags' is `0x20f`, which is `KMF_AUDIT | KMF_DEADBEEF | KMF_REDZONE | KMF_CONTENTS | KMF_HASH`. This document explains each of the debugging features in subsequent sections.

When you are interested in looking at buffers in a particular cache, you can walk the allocated and freed buffers in that cache directly:

```
> 0x70039928::walk kmem
704ba010
702ba008
704ba038
702ba030
...

> 0x70039928::walk freemem
70a9ae50
70a9ae28
704bb730
704bb2f8
...
```

MDB provides a shortcut to supplying the cache address to the kmem walker: a specific walker is provided for each kmem cache, and its name is the same as the name of the cache. For example:

```
> ::walk kmem_alloc_24
704ba010
702ba008
704ba038
702ba030
...

> ::walk thread_cache
70b38080
70aac060
705c4020
70aac1e0
...
```

Now you know how to iterate over the kernel memory allocator's internal data structures and examine the most important members of the `kmem_cache` data structure.

Detecting Memory Corruption

One of the primary debugging facilities of the allocator is that it includes algorithms to recognize data corruption quickly. When corruption is detected, the allocator immediately panics the system.

This section describes how the allocator recognizes data corruption; you must understand this to be able to debug these problems. Memory abuse typically falls into one of the following categories:

- Writing past the end of a buffer
- Accessing uninitialized data
- Continuing to use a freed buffer
- Corrupting kernel memory

Keep these problems in mind as you read the next three sections. They will help you to understand the allocator's design, and enable you to diagnose problems more efficiently.

Freed Buffer Checking: `0xdeadbeef`

When the `KMF_DEADBEEF (0x2)` bit is set in the flags field of a `kmem_cache`, the allocator tries to make memory corruption easy to detect by writing a special pattern into all freed buffers. This pattern is `0xdeadbeef`. Since a typical region of memory contains both allocated and freed memory, sections of each kind of block will be interspersed; here is an example from the "`kmem_alloc_24`" cache:

```
0x70a9add8:     deadbeef        deadbeef
0x70a9ade0:     deadbeef        deadbeef
0x70a9ade8:     deadbeef        deadbeef
0x70a9adf0:     feedface        feedface
0x70a9adf8:     70ae3260        8440c68e
0x70a9ae00:     5               4ef83
0x70a9ae08:     0               0
0x70a9ae10:     1               bbddcafe
0x70a9ae18:     feedface        4fffed
0x70a9ae20:     70ae3200        d1befaed
0x70a9ae28:     deadbeef        deadbeef
0x70a9ae30:     deadbeef        deadbeef
0x70a9ae38:     deadbeef        deadbeef
0x70a9ae40:     feedface        feedface
0x70a9ae48:     70ae31a0        8440c54e
```

The buffer beginning at `0x70a9add8` is filled with the `0xdeadbeef` pattern, which is an immediate indication that the buffer is currently free. At `0x70a9ae28` another free buffer begins; at `0x70a9ae00` an allocated buffer is located between them.

Note - You might have observed that there are some holes on this picture, and that 3 24–byte regions should occupy only 72 bytes of memory, instead of the 120 bytes shown here. This discrepancy is explained in the next section “Redzone: `0xfeedface`” on page 75.

Redzone: `0xfeedface`

The pattern `0xfeedface` appears frequently in the buffer above. This pattern is known as the "redzone" indicator. It enables the allocator (and a programmer debugging a problem) to determine if the boundaries of a buffer have been violated by “buggy” code. Following the redzone is some additional information. The contents of that data depends upon other factors (see “Memory Allocation Logging” on page 78). The redzone and its suffix are collectively called the *buftag* region. Figure 6–1 summarizes this information.

Figure 6–1 The Redzone

The buftag is appended to each buffer in a cache when any of the `KMF_AUDIT`, `KMF_DEADBEEF`, `KMF_REDZONE`, or `KMF_CONTENTS` flags are set in that buffer’s cache. The contents of the buftag depend on whether `KMF_AUDIT` is set.

Decomposing the memory region presented above into distinct buffers is now simple:

```
0x70a9add8:     deadbeef        deadbeef  \
0x70a9ade0:     deadbeef        deadbeef   +- User Data (free)
0x70a9ade8:     deadbeef        deadbeef  /
0x70a9adf0:     feedface        feedface  -- REDZONE
0x70a9adf8:     70ae3260        8440c68e  -- Debugging Data

0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        4fffed    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data

0x70a9ae28:     deadbeef        deadbeef  \
0x70a9ae30:     deadbeef        deadbeef   +- User Data (free)
0x70a9ae38:     deadbeef        deadbeef  /
0x70a9ae40:     feedface        feedface  -- REDZONE
0x70a9ae48:     70ae31a0        8440c54e  -- Debugging Data
```

In the free buffers at `0x70a9add8` and `0x70a9ae28`, the redzone is filled with `0xfeedfacefeedface`. This a convenient way of determining that a buffer is free.

In the allocated buffer beginning at `0x70a9ae00`, the situation is different. The first half of the redzone is used to indicate the end of the buffer `(at 0x70a9ae18)`, and the second half tracks the redzone byte. Recall from "Allocator Basics" on page 69 that there are two allocation types:

1) The client requested memory using `kmem_cache_alloc( )`, in which case the size of the requested buffer is equal to the bufsize of the cache.

2) The client requested memory using `kmem_alloc`(9F), in which case the size of the requested buffer is less than or equal to the bufsize of the cache. For example, a request for 20 bytes will be fulfilled from the `kmem_alloc_24` cache. The allocator enforces the buffer boundary by placing the redzone byte immediately following the client data:

```
0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        4fffed    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data
```

`0xfeedface` at `0x70a9ae18` is followed by a 32-bit word containing what seems to be a random value. This number is actually an encoded representation of the size of the buffer. To decode this number and find the size of the allocated buffer, use the formula:

```
size = redzone_value / (UINT_MAX / KMEM_MAXBUF)
```

The value of `KMEM_MAXBUF` is `16384`, and the value of `UINT_MAX` is `4294967295`. So, in this example,

```
size = 0x4fffed / (4294967295 / 16384) = 20 bytes.
```

This indicates that the buffer requested was of size 20 bytes. The allocator performs this decoding operation and finds that the redzone byte should be at offset 20. The redzone byte is the hex pattern `0xbb`, which is present at `0x729084e4 (0x729084d0 + 0t20)` as expected.

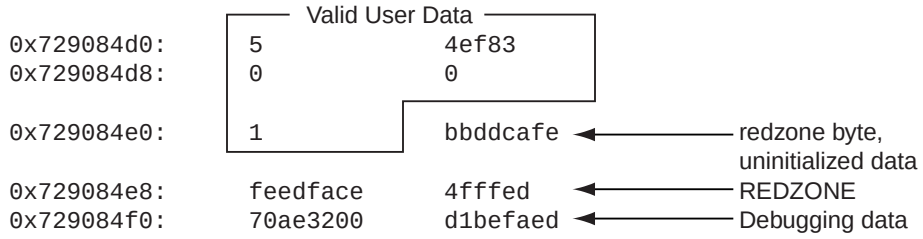

Figure 6–2 Sample `kmem_alloc(9F)` Buffer

Figure 6–3 shows the general form of this memory layout..

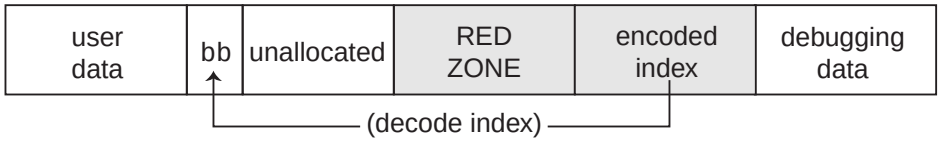

Figure 6–3 Redzone Byte

If the allocation size is the same as the bufsize of the cache, the redzone byte overwrites the first byte of the redzone itself, as shown in Figure 6–4.

Figure 6–4 Redzone Byte at the Beginning of the Redzone

This overwriting results in the first 32-bit word of the redzone being `0xbbedface`, or `0xfeedfabb` depending on the endianness of the hardware on which the system is running.

Note - Why is the allocation size encoded this way? To encode the size, the allocator uses the formula ((UINT_MAX / KMEM_MAXBUF) * size + 1). When the size decode occurs, the integer division discards the remainder of '+1'. However, the addition of 1 is valuable because the allocator can check whether the size is valid by testing whether (size % (UINT_MAX / KMEM_MAXBUF) == 1). In this way, the allocator defends against corruption of the redzone byte index.

Uninitialized Data: `0xbaddcafe`

You might be wondering what the suspicious `0xbbddcafe` at address `0x729084d4` was *before* the redzone byte got placed over the first byte in the word. It was `0xbaddcafe`. When the `KMF_DEADBEEF` flag is set in the cache, allocated but *uninitialized* memory is filled with the `0xbaddcafe` pattern. When the allocator performs an allocation, it loops across the words of the buffer and verifies that each word contains `0xdeadbeef`, then fills that word with `0xbaddcafe`.

A system can panic with a message such as:

```
panic[cpu1]/thread=e1979420: BAD TRAP: type=e (Page Fault)
rp=ef641e88 addr=baddcafe occurred in module "unix" due to an
illegal access to a user address
```

In this case, the address that caused the fault was `0xbaddcafe`: the panicking thread has accessed some data that was never initialized.

Associating Panic Messages With Failures

The kernel memory allocator emits panic messages corresponding to the failure modes described earlier. For example, a system can panic with a message such as:

```
kernel memory allocator: buffer modified after being freed
modification occurred at offset 0x30
```

The allocator was able to detect this case because it tried to validate that the buffer in question was filled with `0xdeadbeef`. At offset `0x30`, this condition was not met. Since this condition indicates memory corruption, the allocator panicked the system.

Another example failure message is:

```
kernel memory allocator: redzone violation: write past end of buffer
```

The allocator was able to detect this case because it tried to validate that the redzone byte (`0xbb`) was in the location it determined from the redzone size encoding. It failed to find the signature byte in the correct location. Since this indicates memory corruption, the allocator panicked the system. Other allocator panic messages are discussed later.

Memory Allocation Logging

This section explains the logging features of the kernel memory allocator and how you can employ them to debug system crashes.

Buftag Data Integrity

As explained earlier, the second half of each buftag contains extra information about the corresponding buffer. Some of this data is debugging information, and some is data private to the allocator. While this auxiliary data can take several different forms, it is collectively known as "Buffer Control" or *bufctl* data.

However, the allocator needs to know whether a buffer's bufctl pointer is valid, since this pointer might also have been corrupted by malfunctioning code. The allocator confirms the integrity of its auxiliary pointer by storing the pointer *and* an encoded version of that pointer, and then cross-checking the two versions.

As shown in Figure 6–5, these pointers are the *bcp* (buffer control pointer) and *bxstat* (buffer control XOR status). The allocator arranges bcp and bxstat so that the expression `bcp XOR bxstat` equals a well-known value.

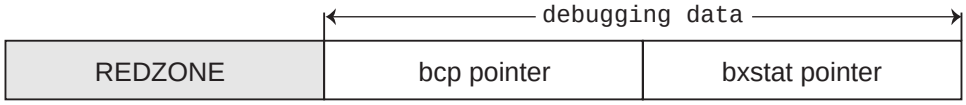

Figure 6–5 Extra Debugging Data in the Buftag

In the event that one or both of these pointers becomes corrupted, the allocator can easily detect such corruption and panic the system. When a buffer is *allocated*, `bcp XOR bxstat = 0xa110c8ed` ("allocated"). When a buffer is free, `bcp XOR bxstat = 0xf4eef4ee` ("freefree").

Note - You might find it helpful to re-examine the example provided in "Freed Buffer Checking: `0xdeadbeef`" on page 74, in order to confirm that the buftag pointers shown there are consistent.

In the event that the allocator finds a corrupt buftag, it panics the system and produces a message similar to the following:

```
kernel memory allocator: boundary tag corrupted
    bcp ^ bxstat = 0xffeef4ee, should be f4eef4ee
```

Remember, if bcp is corrupt, it is still possible to retrieve its value by taking the value of `bxstat XOR 0xf4eef4ee` or `bxstat XOR 0xa110c8ed`, depending on whether the buffer is allocated or free.

The `bufctl` Pointer

The buffer control (bufctl) pointer contained in the buftag region can have different meanings, depending on the cache's `kmem_flags`. The behavior toggled by the `KMF_AUDIT` flag is of particular interest: when the KMF_AUDIT flag is *not* set, the kernel memory allocator allocates a `kmem_bufctl_t` structure for each buffer. This structure contains some minimal accounting information about each buffer. When the `KMF_AUDIT` flag *is* set, the allocator instead allocates a `kmem_bufctl_audit_t`, an extended version of the `kmem_bufctl_t`.

This section presumes the `KMF_AUDIT` flag is set. For caches that do not have this bit set, the amount of available debugging information is reduced.

The `kmem_bufctl_audit_t` (`bufctl_audit` for short) contains additional information about the last transaction that occurred on this buffer. The following example shows how to apply the `bufctl_audit` macro to examine an audit record. The buffer shown is the example buffer used in "Detecting Memory Corruption" on page 74:

```
> 0x70a9ae00,5/KKn
0x70a9ae00:     5               4ef83
                0               0
                1               bbddcafe
                feedface        4fffed
                70ae3200        d1befaed
```

Using the techniques presented above, it is easy to see that `0x70ae3200` points to the `bufctl_audit` record: it is the first pointer following the redzone. To examine the `bufctl_audit` record it points to, apply the `bufctl_audit` macro:

```
> 0x70ae3200$<bufctl_audit
0x70ae3200:     next            addr            slab
                70378000        70a9ae00        707c86a0
0x70ae320c:     cache           timestamp       thread
                70039928        e1bd0e26afe     70aac4e0
0x70ae321c:     lastlog         contents        stackdepth
                7011c7c0        7018a0b0        4
0x70ae3228:
                kmem_zalloc+0x30
                pid_assign+8
                getproc+0x68
                cfork+0x60
```

The 'addr' field is the address of the buffer corresponding to this bufctl_audit record. This is the original address: `0x70a9ae00`. The 'cache' field points at the kmem_cache that allocated this buffer. You can use the `::kmem_cache` dcmd to examine it as follows:

```
> 0x70039928::kmem_cache
ADDR     NAME                      FLAG  CFLAG  BUFSIZE  BUFTOTL
70039928 kmem_alloc_24             020f 000000       24      612
```

The 'timestamp' field represents the time this transaction occurred. This time is expressed in the same manner as `gethrtime`(3C).

'thread' is a pointer to the thread that performed the last transaction on this buffer. The 'lastlog' and 'contents' pointers point to locations in the allocator's *transaction logs*. These logs are discussed in detail in "Allocator Logging Facility" on page 83.

Typically, the most useful piece of information provided by `bufctl_audit` is the stack trace recorded at the point at which the transaction took place. In this case, the transaction was an allocation called as part of executing `fork`(2).

Advanced Memory Analysis

This section describes facilities for performing advanced memory analysis, including locating memory leaks and sources of data corruption.

Finding Memory Leaks

The `::findleaks` dcmd provides powerful and efficient detection of memory leaks in kernel crash dumps where 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), and 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:

```
> ::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
```

Using the bufctl pointers, you can obtain the complete stack backtrace of the allocation by applying the `bufctl_audit` macro:

```
> 70d3b1a0$<bufctl_audit
0x70d3b1a0:     next            addr            slab
                70a049c0        70d03b28        70bb7480
0x70d3b1ac:     cache           timestamp       thread
                7003a028        13f7cf63b3      70b38380
0x70d3b1bc:     lastlog         contents        stackdepth
                700d6e60        0               5
0x70d3b1c8:
                kmem_alloc+0x30
                sigaddq+0x108
                sigsendproc+0x210
                sigqkill+0x90
                kill+0x28
```

The programmer can usually use the `bufctl_audit` information and the allocation stack trace to quickly track down the code path that leaks the given buffer.

Finding References to Data

When trying to diagnose a memory corruption problem, you should know what other kernel entities hold a copy of a particular pointer. This is important because it can reveal which thread accessed a data structure after it was freed. It can also make it easier to understand what kernel entities are sharing knowledge of a particular (valid) data item. The `::whatis` and `::kgrep` dcmds can be used to answer these questions. You can apply `::whatis` to a value of interest:

```
> 0x705d8640::whatis
705d8640 is 705d8000+640, allocated from kmem_va_8192
705d8640 is 705d8640+0, allocated from streams_mblk
```

In this case, `0x705d8640` is revealed to be a pointer to a STREAMS `mblk` structure. Notice that this allocation also appears in the `kmem_va_8192` cache—a kmem cache that is fronting the kmem_va virtual memory arena. The complete list of kmem caches and vmem arenas is displayed by the `::kmastat` dcmd. You can use `::kgrep` to locate other kernel addresses that contain a pointer to this mblk. This illustrates the hierarchical nature of memory allocations in the system; in general,

you can determine the type of object referred to by the given address from the name of the most specific kmem cache.

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

and investigate them by applying `::whatis` again:

```
> 400a3720::whatis
400a3720 is in thread 7095b240's stack

> 706add34::whatis
706add34 is 706ac000+1d34, allocated from kmem_va_8192
706add34 is 706add20+14, allocated from streams_dblk_120
```

Here one pointer is located on the stack of a known kernel thread, and another is the mblk pointer inside of the corresponding STREAMS `dblk` structure.

Finding Corrupt Buffers With `::kmem_verify`

MDB's `::kmem_verify` dcmd implements most of the same checks that the kmem allocator does at runtime. `::kmem_verify` can be invoked in order to scan every kmem cache with appropriate `kmem_flags`, or to examine a particular cache.

Here is an example of using `::kmem_verify` to isolate a problem:

```
> ::kmem_verify
Cache Name                      Addr     Cache Integrity
kmem_alloc_8                    70039428 clean
kmem_alloc_16                   700396a8 clean
kmem_alloc_24                   70039928 1 corrupt buffer
kmem_alloc_32                   70039ba8 clean
kmem_alloc_40                   7003a028 clean
kmem_alloc_48                   7003a2a8 clean
...
```

It is easy to see here that the `kmem_alloc_24` cache contains what `::kmem_verify` believes to be a problem. With an explicit cache argument, the `::kmem_verify` dcmd provides more detailed information about the problem:

```
> 70039928::kmem_verify
Summary for cache 'kmem_alloc_24'
  buffer 702babc0 (free) seems corrupted, at 702babc0
```

The next step is to examine the buffer which `::kmem_verify` believes to be corrupt:

```
> 0x702babc0,5/KKn
0x702babc0:     0               deadbeef
                deadbeef        deadbeef
                deadbeef        deadbeef
```

```
                feedface        feedface
                703785a0        84d9714e
```

The reason that `::kmem_verify` flagged this buffer is now clear: The first word in the buffer (at `0x702babc0`) should probably be filled with the `0xdeadbeef` pattern, not with a `0`. At this point, examining the `bufctl_audit` for this buffer might yield clues about what code recently wrote to the buffer, indicating where and when it was freed.

Another useful technique in this situation is to use `::kgrep` to search the address space for references to address `0x702babc0`, in order to discover what threads or data structures are still holding references to this freed data.

Allocator Logging Facility

When `KMF_AUDIT` is set for a cache, the kernel memory allocator maintains a log that records the recent history of its activity. This *transaction log* records `bufctl_audit` records. If the `KMF_AUDIT` and the `KMF_CONTENTS` flags are both set, the allocator generates a *contents log* that records portions of the actual contents of allocated and freed buffers. The structure and use of the contents log is outside the scope of this document. The transaction log is discussed in this section.

MDB provides several facilities for displaying the transaction log. The simplest is `::walk kmem_log`, which prints out the transaction in the log as a series of `bufctl_audit_t` pointers:

```
> ::walk kmem_log
70128340
701282e0
70128280
70128220
701281c0
...
> 70128340$<bufctl_audit
0x70128340:     next            addr            slab
                70ac1d40        70bc4ea8        70bb7c00
0x7012834c:     cache           timestamp       thread
                70039428        e1bd7abe721     70aacde0
0x7012835c:     lastlog         contents        stackdepth
                701282e0        7018f340        4
0x70128368:
                kmem_cache_free+0x24
                nfs3_sync+0x3c
                vfs_sync+0x84
                syssync+4
```

A more elegant way to view the entire transaction log is by using the `::kmem_log` command:

```
> ::kmem_log
CPU ADDR     BUFADDR         TIMESTAMP THREAD
  0 70128340 70bc4ea8      e1bd7abe721 70aacde0
```

```
  0 701282e0 70bc4ea8      e1bd7aa86fa 70aacde0
  0 70128280 70bc4ea8      e1bd7aa27dd 70aacde0
  0 70128220 70bc4ea8      e1bd7a98a6e 70aacde0
  0 701281c0 70d03738      e1bd7a8e3e0 70aacde0
  ...
  0 70127140 70cf78a0      e1bd78035ad 70aacde0
  0 701270e0 709cf6c0      e1bd6d2573a 40033e60
  0 70127080 70cedf20      e1bd6d1e984 40033e60
  0 70127020 70b09578      e1bd5fc1791 40033e60
  0 70126fc0 70cf78a0      e1bd5fb6b5a 40033e60
  0 70126f60 705ed388      e1bd5fb080d 40033e60
  0 70126f00 705ed388      e1bd551ff73 70aacde0
  ...
```

The output of `::kmem_log` is sorted in descending order by timestamp. The `ADDR` column is the `bufctl_audit` structure corresponding to that transaction; `BUFADDR` points to the actual buffer.

These figures represent *transactions* on buffers (both allocations and frees). When a particular buffer is corrupted, it can be helpful to locate that buffer in the transaction log, then determine in which other transactions the transacting thread was involved. This can help to assemble a picture of the sequence of events that occurred prior to and after the allocation (or free) of a buffer.

You can employ the `::bufctl` command to filter the output of walking the transaction log. The `::bufctl -a` command filters the buffers in the transaction log by buffer address. This example filters on buffer `0x70b09578`:

```
> ::walk kmem_log | ::bufctl -a 0x70b09578
ADDR     BUFADDR   TIMESTAMP   THREAD   CALLER
70127020 70b09578  e1bd5fc1791 40033e60 biodone+0x108
70126e40 70b09578  e1bd55062da 70aacde0 pageio_setup+0x268
70126de0 70b09578  e1bd52b2317 40033e60 biodone+0x108
70126c00 70b09578  e1bd497ee8e 70aacde0 pageio_setup+0x268
70120480 70b09578  e1bd21c5e2a 70aacde0 elfexec+0x9f0
70120060 70b09578  e1bd20f5ab5 70aacde0 getelfhead+0x100
7011ef20 70b09578  e1bd1e9a1dd 70aacde0 ufs_getpage_miss+0x354
7011d720 70b09578  e1bd1170dc4 70aacde0 pageio_setup+0x268
70117d80 70b09578  e1bcff6ff27 70bc2480 elfexec+0x9f0
70117960 70b09578  e1bcfea4a9f 70bc2480 getelfhead+0x100
...
```

This example illustrates that a particular buffer can be used in numerous transactions.

Note - Remember that the kmem transaction log is an incomplete record of the transactions made by the kernel memory allocator. Older entries in the log are evicted as needed in order to keep the size of the log constant.

The `::allocdby` and `::freedby` dcmds provide a convenient way to summarize transactions associated with a particular thread. Here is an example of listing the recent allocations performed by thread `0x70aacde0`:

```
> 0x70aacde0::allocdby
BUFCTL      TIMESTAMP CALLER
```

```
70d4d8c0  e1edb14511a allocb+0x88
70d4e8a0  e1edb142472 dblk_constructor+0xc
70d4a240  e1edb13dd4f allocb+0x88
70d4e840  e1edb13aeec dblk_constructor+0xc
70d4d860  e1ed8344071 allocb+0x88
70d4e7e0  e1ed8342536 dblk_constructor+0xc
70d4a1e0  e1ed82b3a3c allocb+0x88
70a53f80  e1ed82b0b91 dblk_constructor+0xc
70d4d800  e1e9b663b92 allocb+0x88
```

By examining `bufctl_audit` records, you can understand the recent activites of a particular thread.

CHAPTER 7

Module Programming API

This chapter describes the structures and functions contained in the MDB debugger module API. The header file <sys/mdb_modapi.h> contains prototypes for these functions, and the SUNWmdbdem package provides source code for an example module in the directory /usr/demo/mdb.

Debugger Module Linkage

_mdb_init()

```
const mdb_modinfo_t *_mdb_init(void);
```

Each debugger module is required to provide, for linkage and identification purposes, a function named _mdb_init(). This function returns a pointer to a persistent (that is, not declared as an automatic variable) mdb_modinfo_t structure, as defined in </sys/mdb_modapi.h>:

```
typedef struct mdb_modinfo {
   ushort_t mi_dvers;              /* Debugger API version number */
   const mdb_dcmd_t *mi_dcmds;     /* NULL-terminated list of dcmds */
   const mdb_walker_t *mi_walkers; /* NULL-terminated list of walks */
} mdb_modinfo_t;
```

The *mi_dvers* member is used to identify the API version number, and should always be set to MDB_API_VERSION. The current version number is therefore compiled into each debugger module, allowing the debugger to identify and verify the application binary interface used by the module. The debugger does not load modules that are compiled for an API version that is more recent than the debugger itself.

The *mi_dcmds* and *mi_walkers* members, if not NULL, point to arrays of dcmd and walker definition structures, respectively. Each array must be terminated by a NULL element. These dcmds and walkers are installed and registered with the debugger as part of the module loading process. The debugger will refuse to load the module if one or more dcmds or walkers are defined improperly or if they have conflicting or invalid names. Dcmd and walker names are prohibited from containing characters that have special meaning to the debugger, such as quotation marks and parentheses.

The module can also execute code in `_mdb_init()` using the module API to determine if it is appropriate to load. For example, a module can only be appropriate for a particular target if certain symbols are present. If these symbols are not found, the module can return NULL from the `_mdb_init()` function. In this case, the debugger will refuse to load the module and an appropriate error message is printed.

`_mdb_fini()`

```
void _mdb_fini(void);
```

If the module performs certain tasks prior to unloading, such as freeing persistent memory previously allocated with `mdb_alloc()`, it can declare a function named `_mdb_fini()` for this purpose. This function is not required by the debugger. If declared, it is called once prior to unloading the module. Modules are unloaded when the user requests that the debugger terminate or when the user explicitly unloads a module using the `::unload` built-in dcmd.

Dcmd Definitions

```
int dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv);
```

A dcmd is implemented with a function similar to the `dcmd()` declaration. This function receives four arguments and returns an integer status. The function arguments are:

addr Current address, also called dot. At the start of the dcmd, this address corresponds to the value of the dot " `.` " variable in the debugger.

flags Integer containing the logical OR of one or more of the following flags:

`DCMD_ADDRSPEC` An explicit address was specified to the left of `::dcmd`.

DCMD_LOOP	The dcmd was invoked in a loop using the `,count` syntax, or the dcmd was invoked in a loop by a pipeline.
DCMD_LOOPFIRST	This invocation of the dcmd function corresponds to the first loop or pipeline invocation.
DCMD_PIPE	The dcmd was invoked with input from a pipeline.
DCMD_PIPE_OUT	The dcmd was invoked with output set to a pipeline.

As a convenience, the `DCMD_HDRSPEC( )` macro is provided to allow a dcmd to test its flags to determine if it should print a header line (that is, it was not invoked as part of a loop, or it was invoked as the first iteration of a loop or pipeline).

argc Number of arguments in the *argv* array.

argv Array of arguments specified to the right of `::dcmd` on the command line. These arguments can be either strings or integer values.

The dcmd function is expected to return one of the following integer values, defined in `<sys/mdb_modapi.h>`.

DCMD_OK	The dcmd completed successfully.
DCMD_ERR	The dcmd failed for some reason.
DCMD_USAGE	The dcmd failed because invalid arguments were specified. When this value is returned, the dcmd usage message (described below) prints automatically.
DCMD_NEXT	The next dcmd definition (if one is present) is automatically invoked with the same arguments.
DCMD_ABORT	The dcmd failed, and the current loop or pipeline should be aborted. This is like `DCMD_ERR`, but indicates that no further progress is possible in the current loop or pipe.

Each dcmd consists of a function defined according to the example `dcmd( )` prototype, and a corresponding `mdb_dcmd_t` structure, as defined in `<sys/mdb_modapi.h>`. This structure consists of the following fields:

`const char *dc_name`	The string name of the dcmd, without the leading "`::`". The name cannot contain any of the MDB meta-characters, such as $ or `‘`.
`const char *dc_usage`	An optional usage string for the dcmd, to be printed when the dcmd returns `DCMD_USAGE`. For example, if the dcmd accepts options `–a` and `–b`, `dc_usage` might be specified as "`[–ab]`". If the dcmd accepts no arguments, `dc_usage` can be set to NULL. If the usage string begins with "`:`", this is shorthand for indicating that the dcmd requires an explicit address (that is, it requires `DCMD_ADDRSPEC` to be set in its flags parameter). If the usage string begins with "?", this indicates that the dcmd optionally accepts an address. These hints modify the usage message accordingly.
`const char *dc_descr`	A mandatory description string, briefly explaining the purpose of the dcmd. This string should consist of only a single line of text.
`mdb_dcmd_f *dc_funcp`	A pointer to the function that will be called to execute the dcmd.
`void (*dc_help)(void)`	An optional function pointer to a help function for the dcmd. If this pointer is not NULL, this function will be called when the user executes `::help` *dcmd*. This function can use `mdb_printf()` to display further information or examples.

Walker Definitions

```
int walk_init(mdb_walk_state_t *wsp);
int walk_step(mdb_walk_state_t *wsp);
void walk_fini(mdb_walk_state_t *wsp);
```

A walker is composed of three functions, `init`, `step`, and `fini`, which are defined according to the example prototypes above. A walker is invoked by the debugger when one of the walk functions (such as `mdb_walk()`) is called, or when the user executes the `::walk` built-in dcmd. When the walk begins, MDB calls the walker's init function, passing it the address of a new `mdb_walk_state_t` structure, as defined in `<sys/modapi.h>`:

```
typedef struct mdb_walk_state {
   mdb_walk_cb_t walk_callback;    /* Callback to issue */
   void *walk_cbdata;              /* Callback private data */
   uintptr_t walk_addr;            /* Current address */
   void *walk_data;                /* Walk private data */
   void *walk_arg;                 /* Walk private argument */
   void *walk_layer;               /* Data from underlying layer */
} mdb_walk_state_t;
```

A separate `mdb_walk_state_t` is created for each walk, so that multiple instances of the same walker can be active simultaneously. The state structure contains the callback the walker should invoke at each step (*walk_callback*), and the private data for the callback (*walk_cbdata*), as specified to `mdb_walk()`, for example. The *walk_cbdata* pointer is opaque to the walker: it must not modify or dereference this value, nor can it assume it is a pointer to valid memory.

The starting address for the walk is stored in *walk_addr*. This is either NULL if `mdb_walk()` was called, or the address parameter specified to `mdb_pwalk()`. If the `::walk` built-in was used, *walk_addr* will be non-NULL if an explicit address was specified on the left-hand side of `::walk`. A walk with a starting address of NULL is referred to as *global*. A walk with an explicit non-NULL starting address is referred to as *local*.

The *walk_data* and *walk_arg* fields are provided for use as private storage for the walker. Complex walkers might need to allocate an auxiliary state structure and set *walk_data* to point to this structure. Each time a walk is initiated, *walk_arg* is initialized to the value of the `walk_init_arg` member of the corresponding walker's `mdb_walker_t` structure.

In some cases, it is useful to have several walkers share the same init, step, and fini routines. For example, the MDB `genunix` module provides walkers for each kernel memory cache. These share the same init, step, and fini functions, and use the `walk_init_arg` member of the `mdb_walker_t` to specify the address of the appropriate cache as the *walk_arg*.

If the walker calls `mdb_layered_walk()` to instantiate an underlying layer, then the underlying layer will reset *walk_addr* and *walk_layer* prior to each call to the walker's step function. The underlying layer sets *walk_addr* to the target virtual address of the underlying object, and set *walk_layer* to point to the walker's local copy of the underlying object. For more information on layered walks, refer to the discussion of `mdb_layered_walk()` below.

The walker init and step functions are expected to return one of the following status values:

`WALK_NEXT`	Proceed to the next step. When the walk init function returns `WALK_NEXT`, MDB invokes the walk step function. When the walk step function returns `WALK_NEXT`, this indicates that MDB should call the step function again.

WALK_DONE	The walk has completed successfully. WALK_DONE can be returned by either the step function to indicate that the walk is complete, or by the init function to indicate that no steps are needed (for example, if the given data structure is empty).
WALK_ERR	The walk has terminated due to an error. If WALK_ERR is returned by the init function, mdb_walk( ) (or any of its counterparts) returns –1 to indicate that the walker failed to initialize. If WALK_ERR is returned by the step function, the walk terminates but mdb_walk( ) returns success.

The *walk_callback* is also expected to return one of the values above. Therefore, the walk step function's job is to determine the address of the next object, read in a local copy of this object, call the *walk_callback* function, then return its status. The step function can also return WALK_DONE or WALK_ERR without invoking the callback if the walk is complete or if an error occurred.

The walker itself is defined using the mdb_walker_t structure, defined in :

```
typedef struct mdb_walker {
   const char *walk_name;                 /* Walk type name */
   const char *walk_descr;                /* Walk description */
   int (*walk_init)(mdb_walk_state_t *);  /* Walk constructor */
   int (*walk_step)(mdb_walk_state_t *);  /* Walk iterator */
   void (*walk_fini)(mdb_walk_state_t *); /* Walk destructor */
   void *walk_init_arg;                   /* Constructor argument */
} mdb_walker_t;
```

The walk_name and walk_descr fields should be initialized to point to strings containing the name and a brief description of the walker, respectively. A walker is required to have a non-NULL name and description, and the name cannot contain any of the MDB meta-characters. The description string is printed by the ::walkers and ::dmods built-in dcmds.

The walk_init, walk_step, and walk_fini members refer to the walk functions themselves, as described earlier. The walk_init and walk_fini members can be set to NULL to indicate that no special initialization or cleanup actions need to be taken.

The walk_step member cannot be set to NULL. The walk_init_arg member is used to initialize the walk_arg member of each new mdb_walk_state_t created for the given walker, as described earlier. See Figure 7–1 for the steps of a typical walker.

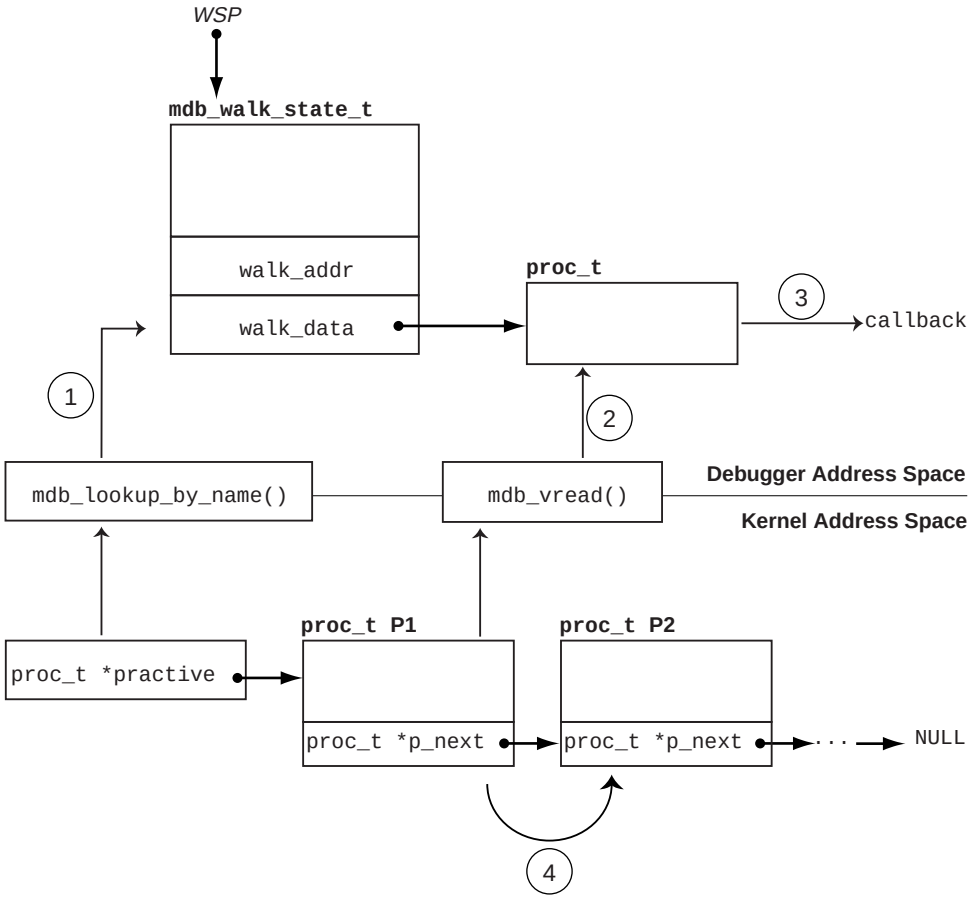

Figure 7–1 Sample Walker

The walker is designed to iterate over the list of `proc_t` structures in the kernel. The head of the list is stored in the global `practive` variable, and each element's `p_next` pointer points to the next `proc_t` in the list. The list is terminated with a NULL pointer. In the walker's `init` routine, the `practive` symbol is located using `mdb_lookup_by_name()` step (1), and its value is copied into the `mdb_walk_state_t` pointed to by *wsp*.

In the walker's step function, the next `proc_t` structure in the list is copied into the debugger's address space using `mdb_vread()` step (2), the callback function is invoked with a pointer to this local copy, step (3), and then the `mdb_walk_state_t` is updated with the address of the `proc_t` structure for the next iteration. This update corresponds to following the pointer, step (4) to the next element in the list.

These steps demonstrate the structure of a typical walker: the init routine locates the global information for a particular data structure, the step function reads in a local copy of the next data item and passes it to the callback function, and the address of

the next element is read. Finally, when the walk terminates, the fini function frees any private storage.

API Functions

mdb_pwalk()

```
int mdb_pwalk(const char *name, mdb_walk_cb_t func, void *data, uintptr_t addr);
```

Initiate a local walk starting at *addr* using the walker specified by *name*, and invoke the callback function *func* at each step. If *addr* is NULL, a global walk is performed (that is, the `mdb_pwalk()` invocation is equivalent to the identical call to `mdb_walk()` without the trailing *addr* parameter). This function returns 0 for success, or -1 for error. The `mdb_pwalk()` function fails if the walker itself returns a fatal error, or if the specified walker name is not known to the debugger. The walker name may be scoped using the backquote (‘) operator if there are naming conflicts. The *data* parameter is an opaque argument that has meaning only to the caller; it is passed back to *func* at each step of the walk.

mdb_walk()

```
int mdb_walk(const char *name, mdb_walk_cb_t func, void *data);
```

Initiate a global walk starting at *addr* using the walker specified by *name*, and invoke the callback function *func* at each step. This function returns 0 for success, or -1 for error. The `mdb_walk()` function fails if the walker itself returns a fatal error, or if the specified walker name is not known to the debugger. The walker name can be scoped using the backquote (‘) operator if there are naming conflicts. The data parameter is an opaque argument that has meaning only to the caller; it is passed back to *func* at each step of the walk.

mdb_pwalk_dcmd()

```
int mdb_pwalk_dcmd(const char *wname, const char *dcname, int argc,
   const mdb_arg_t *argv, uintptr_t addr);
```

Initiate a local walk starting at *addr* using the walker specified by *wname*, and invoke the dcmd specified by *dcname* with the specified *argc* and *argv* at each step. This function returns 0 for success, or -1 for error. The function fails if the walker itself returns a fatal error, if the specified walker name or dcmd name is not known to the debugger, or if the dcmd itself returns `DCMD_ABORT` or `DCMD_USAGE` to the walker.

The walker name and dcmd name can each be scoped using the backquote (‘) operator if there are naming conflicts. When invoked from `mdb_pwalk_dcmd()`, the dcmd will have the `DCMD_LOOP` and `DCMD_ADDRSPEC` bits set in its flags parameter, and the first call will have `DCMD_LOOPFIRST` set.

mdb_walk_dcmd()

```
int mdb_walk_dcmd(const char *wname, const char *dcname, int argc,
   const mdb_arg_t *argv);
```

Initiate a global walk using the walker specified by *wname*, and invoke the dcmd specified by *dcname* with the specified *argc* and *argv* at each step. This function returns 0 for success, or -1 for error. The function fails if the walker itself returns a fatal error, if the specified walker name or dcmd name is not known to the debugger, or if the dcmd itself returns `DCMD_ABORT` or `DCMD_USAGE` to the walker. The walker name and dcmd name can each be scoped using the backquote (‘) operator if there are naming conflicts. When invoked from `mdb_walk_dcmd()`, the dcmd will have the `DCMD_LOOP` and `DCMD_ADDRSPEC` bits set in its flags parameter, and the first call will have `DCMD_LOOPFIRST` set.

mdb_call_dcmd()

```
int mdb_call_dcmd(const char *name, uintptr_t addr, uint_t flags,
   int argc, const mdb_arg_t *argv);
```

Invoke the specified dcmd name with the given parameters. The dot variable is reset to *addr*, and *addr*, *flags*, *argc*, and *argv* are passed to the dcmd. The function returns 0 for success, or -1 for error. The function fails if the dcmd returns `DCMD_ERR`, `DCMD_ABORT`, or `DCMD_USAGE`, or if the specified dcmd name is not known to the debugger. The dcmd name can be scoped using the backquote (‘) operator if there are naming conflicts.

mdb_layered_walk()

```
int mdb_layered_walk(const char *name, mdb_walk_state_t *wsp);
```

Layer the walk denoted by *wsp* on top of a walk initiated using the specified walker *name*. The name can be scoped using the backquote (‘) operator if there are naming conflicts. Layered walks can be used, for example, to facilitate constructing walkers for data structures that are embedded in other data structures.

For example, suppose that each CPU structure in the kernel contains a pointer to an embedded structure. To write a walker for the embedded structure type, you could replicate the code to iterate over CPU structures and derefence the appropriate

member of each CPU structure, or you could layer the embedded structure's walker on top of the existing CPU walker.

The `mdb_layered_walk()` function is used from within a walker's init routine to add a new layer to the current walk. The underlying layer is initialized as part of the call to `mdb_layered_walk()`. The calling walk routine passes in a pointer to its current walk state; this state is used to construct the layered walk. Each layered walk is cleaned up after the caller's walk fini function is called. If more than one layer is added to a walk, the caller's walk step function will step through each element returned by the first layer, then the second layer, and so forth.

The `mdb_layered_walk()` function returns 0 for success, or -1 for error. The function fails if the specified walker name is not known to the debugger, if the *wsp* pointer is not a valid, active walk state pointer, if the layered walker itself fails to initialize, or if the caller attempts to layer the walker on top of itself.

mdb_add_walker()

```
int mdb_add_walker(const mdb_walker_t *w);
```

Register a new walker with the debugger. The walker is added to the module's namespace, and to the debugger's global namespace according to the name resolution rules described in "dcmd and Walker Name Resolution" on page 30. This function returns 0 for success, or -1 for error if the given walker name is already registered by this module, or if the walker structure *w* is improperly constructed. The information in the `mdb_walker_t` *w* is copied to internal debugger structures, so the caller can reuse or free this structure after the call to `mdb_add_walker()`.

mdb_remove_walker()

```
int mdb_remove_walker(const char *name);
```

Remove the walker with the specified *name*. This function returns 0 for success, or -1 for error. The walker is removed from the current module's namespace. The function fails if the walker name is unknown, or is registered only in another module's namespace. The `mdb_remove_walker()` function can be used to remove walkers that were added dynamically using `mdb_add_walker()`, or walkers that were added statically as part of the module's linkage structure. The scoping operator cannot be used in the walker name; it is not legal for the caller of `mdb_remove_walker()` to attempt to remove a walker exported by a different module.

mdb_vread() and mdb_vwrite()

```
ssize_t mdb_vread(void *buf, size_t nbytes, uintptr_t addr);
ssize_t mdb_vwrite(const void *buf, size_t nbytes, uintptr_t addr);
```

These functions provide the ability to read and write data from a given target virtual address, specified by the *addr* parameter. The `mdb_vread()` function returns *nbytes* for success, or -1 for error; if a read is truncated because only a portion of the data can be read from the specified address, -1 is returned. The `mdb_vwrite()` function returns the number of bytes actually written upon success; -1 is returned upon error.

mdb_pread() and mdb_pwrite()

```
ssize_t mdb_pread(void *buf, size_t nbytes, uint64_t addr);
ssize_t mdb_pwrite(const void *buf, size_t nbytes, uint64_t addr);
```

These functions provide the ability to read and write data from a given target physical address, specified by the addr parameter. The `mdb_pread()` function returns *nbytes* for success, or -1 for error; if a read is truncated because only a portion of the data can be read from the specified address, -1 is returned. The `mdb_pwrite()` function returns the number of bytes actually written upon success; -1 is returned upon error.

mdb_readstr()

```
ssize_t mdb_readstr(char *s, size_t nbytes, uintptr_t addr);
```

The `mdb_readstr()` function reads a null-terminated C string beginning at the target virtual address *addr* into the buffer addressed by *s*. The size of the buffer is specified by *nbytes*. If the string is longer than can fit in the buffer, the string is truncated to the buffer size and a null byte is stored at `s[nbytes - 1]`. The length of the string stored in *s* (not including the terminating null byte) is returned upon success; otherwise -1 is returned to indicate an error.

mdb_writestr()

```
ssize_t mdb_writestr(const char *s, uintptr_t addr);
```

The `mdb_writestr()` function writes a null-terminated C string from *s* (including the trailing null byte) to the target's virtual address space at the address specified by *addr*. The number of bytes written (not including the terminating null byte) is returned upon success; otherwise, -1 is returned to indicate an error.

mdb_readsym( )

```
ssize_t mdb_readsym(void *buf, size_t nbytes, const char *name);
```

mdb_readsym( ) is similar to mdb_vread( ), except that the virtual address at which reading begins is obtained from the value of the symbol specified by *name*. If no symbol by that name is found or a read error occurs, -1 is returned; otherwise *nbytes* is returned for success.

The caller can first look up the symbol separately if it is necessary to distinguish between symbol lookup failure and read failure. The primary executable's symbol table is used for the symbol lookup; if the symbol resides in another symbol table, you must first apply mdb_lookup_by_obj( ), then mdb_vread( ).

mdb_writesym( )

```
ssize_t mdb_writesym(const void *buf, size_t nbytes, const char *name);
```

mdb_writesym( ) is identical to mdb_vwrite( ), except that the virtual address at which writing begins is obtained from the value of the symbol specified by name. If no symbol by that name is found, -1 is returned. Otherwise, the number of bytes successfully written is returned on success, and -1 is returned on error. The primary executable's symbol table is used for the symbol lookup; if the symbol resides in another symbol table, you must first apply mdb_lookup_by_obj( ), then mdb_vwrite( ).

mdb_readvar( ) and mdb_writevar( )

```
ssize_t mdb_readvar(void *buf, const char *name);
ssize_t mdb_writevar(const void *buf, const char *name);
```

mdb_readvar( ) is similar to mdb_vread( ), except that the virtual address at which reading begins and the number of bytes to read are obtained from the value and size of the symbol specified by *name*. If no symbol by that name is found, -1 is returned. The symbol size (the number of bytes read) is returned on success; -1 is returned on error. This is useful for reading well-known variables whose sizes are fixed. For example:

```
int hz;  /* system clock rate */
mdb_readvar(&hz, "hz");
```

The caller can first look up the symbol separately if it is necessary to distinguish between symbol lookup failure and read failure. The caller must also carefully check the definition of the symbol of interest in order to make sure that the local declaration is the exact same type as the target's definition. For example, if the caller declares an int, and the symbol of interest is actually a long, and the debugger is

examining a 64-bit kernel target, `mdb_readvar()` copies back 8 bytes to the caller's buffer, corrupting the 4 bytes following the storage for the `int`.

`mdb_writevar()` is identical to `mdb_vwrite()`, except that the virtual address at which writing begins and the number of bytes to write are obtained from the value and size of the symbol specified by name. If no symbol by that name is found, -1 is returned. Otherwise, the number of bytes successfully written is returned on success, and -1 is returned on error.

For both functions, the primary executable's symbol table is used for the symbol lookup; if the symbol resides in another symbol table, you must first apply `mdb_lookup_by_obj()`, then `mdb_vread()` or `mdb_vwrite()`.

`mdb_lookup_by_name()` and `mdb_lookup_by_obj()`

```
int mdb_lookup_by_name(const char *name, GElf_Sym *sym);
int mdb_lookup_by_obj(const char *object, const char *name, GElf_Sym *sym);
```

Look up the specified symbol name and copy the ELF symbol information into the `GElf_Sym` pointed to by *sym*. If the symbol is found, the function returns 0; otherwise, -1 is returned. The *name* parameter specifies the symbol name. The *object* parameter tells the debugger where to look for the symbol. For the `mdb_lookup_by_name()` function, the object file defaults to `MDB_OBJ_EXEC`. For `mdb_lookup_by_obj()`, the object name should be one of the following:

`MDB_OBJ_EXEC`	Look in the executable's symbol table (`.symtab` section). For kernel crash dumps, this corresponds to the symbol table from the `unix.X` file or from `/dev/ksyms`.
`MDB_OBJ_RTLD`	Look in the runtime link-editor's symbol table. For kernel crash dumps, this corresponds to the symbol table for the `krtld` module.
`MDB_OBJ_EVERY`	Look in all known symbol tables. For kernel crash dumps, this includes the `.symtab` and `.dynsym` sections from the `unix.X` file or `/dev/ksyms`, as well as per-module symbol tables if these have been processed.
`object`	If the name of a particular load object is explicitly specified, the search is restricted to the symbol table of this object. The object can be named according to the naming convention for load objects described in "Symbol Name Resolution" on page 29.

mdb_lookup_by_addr()

```
int mdb_lookup_by_addr(uintptr_t addr, uint_t flag, char *buf,
    size_t len, GElf_Sym *sym);
```

Locate the symbol corresponding to the specified address and copy the ELF symbol information into the `GElf_Sym` pointed to by *sym* and the symbol name into the character array addressed by *buf*. If a corresponding symbol is found, the function returns 0; otherwise -1 is returned.

The flag parameter specifies the lookup mode and should be one of the following:

`MDB_SYM_FUZZY`	Allow fuzzy matching to take place, based on the current symbol distance setting. The symbol distance can be controlled using the `::set -s` built-in. If an explicit symbol distance has been set (absolute mode), the address can match a symbol if the distance from the symbol's value to the address does not exceed the absolute symbol distance. If smart mode is enabled (symbol distance = 0), then the address can match the symbol if it is in the range [symbol value, symbol value + symbol size).
`MDB_SYM_EXACT`	Disallow fuzzy matching. The symbol can match only the address if the symbol value exactly equals the specified address.

If a symbol match occurs, the name of the symbol is copied into the *buf* supplied by the caller. The *len* parameter specifies the length of this buffer in bytes. The caller's *buf* should be at least of size `MDB_SYM_NAMLEN` bytes. The debugger copies the name to this buffer and appends a trailing null byte. If the name length exceeds the length of the buffer, the name is truncated but always includes a trailing null byte.

mdb_getopts()

```
int mdb_getopts(int argc, const mdb_arg_t *argv, ...);
```

Parse and process options and option arguments from the specified argument array (*argv*). The *argc* parameter denotes the length of the argument array. This function processes each argument in order, and stops and returns the array index of the first argument that could not be processed. If all arguments are processed successfully, *argc* is returned.

Following the *argc* and *argv* parameters, the `mdb_getopts()` function accepts a variable list of arguments describing the options that are expected to appear in the *argv* array. Each option is described by an option letter (`char` argument), an option type (`uint_t` argument), and one or two additional arguments, as shown in the

table below. The list of option arguments is terminated with a NULL argument. The type should be one of one of the following:

`MDB_OPT_SETBITS` The option will `OR` the specified bits into a flag word. The option is described by these parameters:

```
char c, uint_t type, uint_t bits, uint_t *p
```

If type is `MDB_OPT_SETBITS` and option *c* is detected in the *argv* list, the debugger will `OR` bits into the integer referenced by pointer *p*.

`MDB_OPT_CLRBITS` The option clears the specified bits from a flag word. The option is described by these parameters:

```
char c, uint_t type, uint_t bits, uint_t *p
```

If type is `MDB_OPT_SETBITS` and option *c* is detected in the *argv* list, the debugger clears bits from the integer referenced by pointer *p*.

`MDB_OPT_STR` The option accepts a string argument. The option is described by these parameters:

```
char c, uint_t type, const char **p
```

If type is `MDB_OPT_STR` and option *c* is detected in the *argv* list, the debugger stores a pointer to the string argument following *c* in the pointer referenced by *p*.

`MDB_OPT_UINTPTR` The option accepts a `uintptr_t` argument. The option is described by these parameters:

```
char c, uint_t type, uintptr_t *p
```

If type is `MDB_OPT_UINTPTR` and option c is detected in the *argv* list, the debugger stores the integer argument following *c* in the `uintptr_t` referenced by *p*.

`MDB_OPT_UINT64` The option accepts a `uint64_t` argument. The option is described by these parameters:

```
char c, uint_t type, uint64_t *p
```

If type is `MDB_OPT_UINT64` and option *c* is detected in the *argv* list, the debugger stores the integer argument following *c* in the `uint64_t` referenced by *p*.

For example, the following source code:

```
int
dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        uint_t opt_v = FALSE;
        const char *opt_s = NULL;

        if (mdb_getopts(argc, argv,
            'v', MDB_OPT_SETBITS, TRUE, &opt_v,
            's', MDB_OPT_STR, &opt_s, NULL) != argc)
                return (DCMD_USAGE);

        /* ... */
}
```

demonstrates how `mdb_getopts()` might be used in a dcmd to accept a boolean option "–v" that sets the *opt_v* variable to TRUE, and an option "–s" that accepts a string argument that is stored in the *opt_s* variable. The `mdb_getopts()` function also automatically issues warning messages if it detects an invalid option letter or missing option argument before returning to the caller. The storage for argument strings and the *argv* array is automatically garbage-collected by the debugger upon completion of the dcmd.

`mdb_strtoull()`

```
u_longlong_t mdb_strtoull(const char *s);
```

Convert the specified string *s* to an `unsigned long long` representation. This function is intended for use in processing and converting string arguments in situations where `mdb_getopts()` is not appropriate. If the string argument cannot be converted to a valid integer representation, the function fails by printing an appropriate error message and aborting the dcmd. Therefore, error checking code is not required. The string can be prefixed with any of the valid base specifiers (0i, 0I, 0o, 0O, 0t, 0T, 0x, or 0X); otherwise, it is interpreted using the default base. The function will fail and abort the dcmd if any of the characters in *s* are not appropriate for the base, or if integer overflow occurs.

`mdb_alloc()`, `mdb_zalloc()` and `mdb_free()`

```
void *mdb_alloc(size_t size, uint_t flags);
void *mdb_zalloc(size_t size, uint_t flags);
void mdb_free(void *buf, size_t size);
```

`mdb_alloc()` allocates *size* bytes of debugger memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can

hold any C data structure. No greater alignment can be assumed. The *flags* parameter should be the bitwise OR of one or more of the following values:

UM_NOSLEEP	If sufficient memory to fulfill the request is not immediately available, return NULL to indicate failure. The caller must check for NULL and handle this case appropriately.
UM_SLEEP	If sufficient memory to fulfill the request is not immediately available, sleep until such time as the request can be fulfilled. As a result, UM_SLEEP allocations are guaranteed to succeed. The caller need not check for a NULL return value.
UM_GC	Garbage-collect allocation automatically at the end of this debugger command. The caller should not subsequently call mdb_free( ) on this block, as the debugger will take care of de-allocation automatically. All memory allocation from within a dcmd must use UM_GC so that if the dcmd is interrupted by the user, the debugger can garbage-collect the memory.

mdb_zalloc( ) is like mdb_alloc( ), but the allocated memory is filled with zeroes before returning it to the caller. No guarantees are made about the initial contents of memory returned by mdb_alloc( ). mdb_free( ) is used to free previously allocated memory (unless it was allocated UM_GC). The buffer address and size must exactly match the original allocation. It is not legal to free only a portion of an allocation with mdb_free( ). It is not legal to free an allocation more than once. An allocation of zero bytes always returns NULL; freeing a NULL pointer with size zero always succeeds.

mdb_printf( )

```
void mdb_printf(const char *format, ...);
```

Print formatted output using the specified format string and arguments. Module writers should use mdb_printf( ) for all output, except for warning and error messages. This function automatically triggers the built-in output pager when appropriate. The mdb_printf( ) function is similar to printf(3C), with certain exceptions: the %C, %S, and %ws specifiers for wide character strings are not supported, the %f floating-point format is not supported, the %e, %E, %g, and %G specifiers for alternative double formats produce only a single style of output, and precision specifications of the form %.n are not supported. The list of specifiers that are supported follows:

Flag Specifiers

%#	If the # sign is found in the format string, this selects the alternate form of the given format. Not all formats have an alternate form; the alternate form is different depending on the format. Refer to the format descriptions below for details on the alternate format.
%+	When printing signed values, always display the sign (prefix with either '+' or '-'). Without %+, positive values have no sign prefix, and negative values have a '-' prefix prepended to them.
%-	Left-justify the output within the specified field width. If the width of the output is less than the specified field width, the output will be padded with blanks on the right-hand side. Without %-, values are right-justified by default.
%0	Zero-fill the output field if the output is right-justified and the width of the output is less than the specified field width. Without %0, right-justified values are prepended with blanks in order to fill the field.

Field Width Specifiers

%n	Field width is set to the specified decimal value.
%?	Field width is set to the maximum width of a hexadecimal pointer value. This is 8 in an ILP32 environment, and 16 in an LP64 environment.
%*	Field width is set to the value specified at the current position in the argument list. This value is assumed to be an `int`. Note that in the 64-bit compilation environment, it may be necessary to cast `long` values to `int`.

Integer Specifiers

%h	Integer value to be printed is a `short`.
%l	Integer value to be printed is a `long`.
%ll	Integer value to be printed is a `long long`.

Terminal Attribute Specifiers

If standard output for the debugger is a terminal, and terminal attributes can be obtained by the terminfo database, the following terminal escape constructs can be used:

%<n>	Enable the terminal attribute corresponding to *n*. Only a single attribute can be enabled with each instance of %<>.
%</n>	Disable the terminal attribute corresponding to *n*. Note that in the case of reverse video, dim text, and bold text, the terminal codes to disable these attributes might be identical. Therefore, it might not be possible to disable these attributes independently of one another.

If no terminal information is available, each terminal attribute construct is ignored by `mdb_printf()`. For more information on terminal attributes, see `terminfo`(4). The available terminfo attributes are:

a	Alternate character set
b	Bold text
d	Dim text
r	Reverse video
s	Best standout capability
u	Underlining

Format Specifiers

%%	The '%' symbol is printed.
%a	Prints an address in symbolic form. The minimum size of the value associated with %a is a `uintptr_t`; specifying %la is not necessary. If address-to-symbol conversion is on, the debugger will attempt to convert the address to a symbol name followed by an offset in the current output radix and print this string; otherwise, the value is printed in the default output radix. If %#a is used, the alternate format adds a ':' suffix to the output.
%A	This format is identical to %a, except when an address cannot be converted to a symbol name plus an offset, nothing is printed. If

%#A is used, the alternate format prints a '?' when address conversion fails.

%b — Decode and print a bit field in symbolic form. This specifier expects two consecutive arguments: the bit field value (`int` for %b, `long` for %lb, and so forth), and a pointer to an array of `mdb_bitmask_t` structures:

```
typedef struct mdb_bitmask {
  const char *bm_name;       /* String name to print */
  u_longlong_t bm_mask;      /* Mask for bits */
  u_longlong_t bm_bits;      /* Result for value & mask */
} mdb_bitmask_t;
```

The array should be terminated by a structure whose `bm_name` field is set to NULL. When %b is used, the debugger reads the value argument, then iterates through each `mdb_bitmask` structure checking to see if:

```
(value & bitmask->bm_mask) == bitmask->bm_bits
```

If this expression is true, the `bm_name` string is printed. Each string printed is separated by a comma. The following example shows how %b can be used to decode the `t_flag` field in a `kthread_t`:

```
const mdb_bitmask_t t_flag_bits[] = {
  { "T_INTR_THREAD", T_INTR_THREAD, T_INTR_THREAD },
  { "T_WAKEABLE", T_WAKEABLE, T_WAKEABLE },
  { "T_TOMASK", T_TOMASK, T_TOMASK },
  { "T_TALLOCSTK", T_TALLOCSTK, T_TALLOCSTK },
   ...
  { NULL, 0, 0 }
};

void
thr_dump(kthread_t *t)
{
  mdb_printf("t_flag = <%hb>\n", t->t_flag, t_flag_bits);

  ...
}
```

If `t_flag` was set to 0x000a, the function would print:

```
t_flag = <T_WAKEABLE,T_TALLOCSTK>
```

%c — Print the specified integer as an ASCII character.

%d — Print the specified integer as a signed decimal value. Same as %i.

%e — Print the specified double in the floating-point format `[+/-]d.ddddddde[+/-]dd`, where there is one digit before the

radix character, seven digits of precision, and at least two digits following the exponent.

`%E` Print the specified double using the same rules as `%e`, except that the exponent character will be '`E`' instead of '`e`'.

`%g` Print the specified double in the same floating-point format as `%e`, but with sixteen digits of precision. If `%llg` is specified, the argument is expected to be of type `long double` (quad-precision floating-point value).

`%G` Print the specified double using the same rules as `%g`, except that the exponent character will be '`E`' instead of '`e`'.

`%i` Print the specified integer as a signed decimal value. Same as `%d`.

`%I` Print the specified 32-bit unsigned integer as an Internet IPv4 address in dotted-decimal format (for example, the hexadecimal value `0xffffffff` would print as `255.255.255.255`).

`%m` Print a margin of whitespace. If no field is specified, the default output margin width is used; otherwise, the field width determines the number of characters of white space that are printed.

`%o` Print the specified integer as an unsigned octal value. If `%#o` is used, the alternate format prefixes the output with '`0`'.

`%p` Print the specified pointer (`void *`) as a hexadecimal value.

`%q` Print the specifierd integer as a signed octal value. If `%#o` is used, the alternate format prefixes the output with '`0`'.

`%r` Print the specified integer as an unsigned value in the current output radix. The user can change the output radix using the `$d` dcmd. If `%#r` is specified, the alternate format prefixes the value with the appropriate base prefix: '`0i`' for binary, '`0o`' for octal, '`0t`' for decimal, or '`0x`' for hexadecimal.

`%R` Print the specified integer as a signed value in the current output radix. If `%#R` is specified, the alternate format prefixes the value with the appropriate base prefix.

`%s` Print the specified string (`char *`). If the string pointer is NULL, the string '`<NULL>`' is printed.

%t	Advance one or more tab stops. If no width is specified, output advances to the next tab stop; otherwise the field width determines how many tab stops are advanced.
%T	Advance the output column to the next multiple of the field width. If no field width is specified, no action is taken. If the current output column is not a multiple of the field width, white space is added to advance the output column.
%u	Print the specified integer as an unsigned decimal value.
%x	Print the specified integer as a hexadecimal value. The characters a-f are used as the digits for the values 10-15. If %#X is specified, the alternate format prefixes the value with '0x'.
%X	Print the specified integer as a hexadecimal value. The characters A-F are used as the digits for the values 10-15. If %#x is specified, the alternate format prefixes the value with '0X'.
%Y	The specified time_t is printed as the string 'year month day HH:MM:SS'.

mdb_snprintf( )

```
size_t mdb_snprintf(char *buf, size_t len, const char *format, ...);
```

Construct a formatted string based on the specified format string and arguments, and store the resulting string into the specified *buf*. The mdb_snprintf( ) function accepts the same format specifiers and arguments as the mdb_printf( ) function. The *len* parameter specifies the size of *buf* in bytes. No more than *len* - 1 formatted bytes is placed in *buf*; mdb_snprintf( ) always terminates *buf* with a null byte. The function returns the number of bytes required for the complete formatted string, not including the terminating null byte. If the *buf* parameter is NULL and *len* is set to zero, the function will not store any characters to *buf* and returns the number of bytes required for the complete formatted string; this technique can be used to determine the appropriate size of a buffer for dynamic memory allocation.

mdb_warn( )

```
void mdb_warn(const char *format, ...);
```

Print an error or warning message to standard error. The mdb_warn( ) function accepts a format string and variable argument list that can contain any of the specifiers documented for mdb_printf( ). However, the output of mdb_warn( ) is

sent to standard error, which is not buffered and is not sent through the output pager or processed as part of a dcmd pipeline. All error messages are automatically prefixed with the string "mdb: ".

In addition, if the *format* parameter does not contain a newline (\n) character, the format string is implicitly suffixed with the string ": %s\n", where %s is replaced by the error message string corresponding to the last error recorded by a module API function. For example, the following source code:

```
if (mdb_lookup_by_name("no_such_symbol", &sym) == -1)
    mdb_warn("lookup_by_name failed");
```

produces this output:

```
mdb: lookup_by_name failed: unknown symbol name
```

mdb_flush()

```
void mdb_flush(void);
```

Flush all currently buffered output. Normally, mdb's standard output is line-buffered; output generated using mdb_printf() is not flushed to the terminal (or other standard output destination) until a newline is encountered, or at the end of the current dcmd. However, in some situations you might want to explicitly flush standard output prior to printing a newline; mdb_flush() can be used for this purpose.

mdb_one_bit()

```
const char *mdb_one_bit(int width, int bit, int on);
```

The mdb_one_bit() function can be used to print a graphical representation of a bit field in which a single bit of interest is turned on or off. This function is useful for creating verbose displays of bit fields similar to the output from snoop(1M) -v. For example, the follow source code:

```
#define FLAG_BUSY       0x1

uint_t flags;

/* ... */

mdb_printf("%s = BUSY\n", mdb_one_bit(8, 0, flags & FLAG_BUSY));
```

produces this output:

```
.... ...1 = BUSY
```

Each bit in the bit field is printed as a period (.), with each 4-bit sequence separated by a white space. The bit of interest is printed as 1 or 0, depending on the setting of

the *on* parameter. The total *width* of the bit field in bits is specified by the width parameter, and the bit position of the bit of interest is specified by the *bit* parameter. Bits are numbered starting from zero. The function returns a pointer to an appropriately sized, null-terminated string containing the formatted bit representation. The string is automatically garbage-collected upon completion of the current dcmd.

mdb_inval_bits()

```
const char *mdb_inval_bits(int width, int start, int stop);
```

The `mdb_inval_bits()` function is used, along with `mdb_one_bit()`, to print a graphical representation of a bit field. This function marks a sequence of bits as invalid or reserved by displaying an 'x' at the appropriate bit location. Each bit in the bit field is represented as a period (.), except for those bits in the range of bit positions specified by the start and stop parameters. Bits are numbered starting from zero. For example, the following source code:

```
mdb_printf("%s = reserved\n", mdb_inval_bits(8, 7, 7));
```

produces this output:

```
x... .... = reserved
```

The function returns a pointer to an appropriately sized, null-terminated string containing the formatted bit representation. The string is automatically garbage-collected upon completion of the current dcmd.

mdb_inc_indent() and mdb_dec_indent()

```
ulong_t mdb_inc_indent(ulong_t n);
ulong_t mdb_dec_indent(ulong_t n);
```

These functions increment and decrement the numbers of columns that MDB will auto-indent with white space before printing a line of output. The size of the delta is specified by *n*, a number of columns. Each function returns the previous absolute value of the indent. Attempts to decrement the indent below zero have no effect. Following a call to either function, subsequent calls to `mdb_printf()` are indented appropriately. If the dcmd completes or is forcibly terminated by the user, the indent is restored automatically to its default setting by the debugger.

mdb_eval()

```
int mdb_eval(const char *s);
```

Evaluate and execute the specified command string *s*, as if it had been read from standard input by the debugger. This function returns 0 for success, or -1 for error. `mdb_eval()` fails if the command string contains a syntax error, or if the command string executed by `mdb_eval()` is forcibly aborted by the user using the pager or by issuing an interrupt.

`mdb_set_dot()` and `mdb_get_dot()`

```
void mdb_set_dot(uintmax_t dot);
uintmax_t mdb_get_dot(void);
```

Set or get the current value of dot (the "`.`" variable). Module developers might want to reposition dot so that, for example, it remains referring to the address following the last address read by the dcmd.

`mdb_get_pipe()`

```
void mdb_get_pipe(mdb_pipe_t *p);
```

Retrieve the contents of the pipeline input buffer for the current dcmd. The `mdb_get_pipe()` function is intended to be used by dcmds that want to consume the complete set of pipe input and execute only once, instead of being invoked repeatedly by the debugger for each pipe input element. Once `mdb_get_pipe()` is invoked, the dcmd will not be invoked again by the debugger as part of the current command. This can be used, for example, to construct a dcmd that sorts a set of input values.

The pipe contents are placed in an array that is garbage-collected upon termination of the dcmd, and the array pointer is stored in `p->pipe_data`. The length of the array is placed in `p->pipe_len`. If the dcmd was not executed on the right-hand side of a pipeline (that is, the `DCMD_PIPE` flag was not set in its flags parameter), `p->pipe_data` is set to NULL and `p->pipe_len` is set to zero.

`mdb_set_pipe()`

```
void mdb_set_pipe(const mdb_pipe_t *p);
```

Set the pipeline output buffer to the contents described by the pipe structure *p*. The pipe values are placed in the array `p->pipe_data`, and the length of the array is stored in `p->pipe_len`. The debugger makes its own copy of this information, so the caller must remember to free `p->pipe_data` if necessary. If the pipeline output buffer was previously non-empty, its contents are replaced by the new array. If the

dcmd was not executed on the left side of a pipeline (that is, the DCMD_PIPE_OUT flag was not set in its flags parameter), this function has no effect.

mdb_get_xdata()

```
ssize_t mdb_get_xdata(const char *name, void *buf, size_t nbytes);
```

Read the contents of the target external data buffer specified by name into the buffer specified by *buf*. The size of *buf* is specified by the *nbytes* parameter; no more than *nbytes* will be copied to the caller's buffer. The total number of bytes read will be returned upon success; -1 will be returned upon error. If the caller wants to determine the size of a particular named buffer, *buf* should be specified as NULL and *nbytes* should be specified as zero. In this case, mdb_get_xdata() will return the total size of the buffer in bytes but no data will be read. External data buffers provide module writers access to target data that is not otherwise accessible through the module API. The set of named buffers exported by the current target can be viewed using the ::xdata built-in dcmd.

Additional Functions

Additionally, module writers can use the following string(3C) and bstring(3C) functions. They are guaranteed to have the same semantics as the functions described in the corresponding Solaris man page.

```
strcat()        strcpy()          strncpy()
strchr()        strrchr()         strcmp()
strncmp()       strcasecmp()      strncasecmp()
strlen()        bcmp()            bcopy()
bzero()         bsearch()         qsort()
```

APPENDIX **A**

Options

This appendix provides a reference for MDB command-line options.

Summary of Command-line Options

The following options are supported:

–A	Disables automatic loading of `mdb` modules. By default, `mdb` attempts to load debugger modules corresponding to the active shared libraries in a user process or core file, or to the loaded kernel modules in the live operating system or an operating system crash dump.
–F	Forcibly takes over the specified user process, if necessary. By default, `mdb` refuses to attach to a user process that is already under the control of another debugging tool, such as `truss`(1). With the –F option, `mdb` attaches to these processes anyway. This can produce unexpected interactions between `mdb` and the other tools attempting to control the process.
–I	Sets default path for locating macro files. Macro files are read using the `$<` or `$<<` dcmds. The path is a sequence of directory names delimited by colon (`:`) characters. The –I `include` path and –L `library` path (see below) can also contain any of the following tokens:

%i Expands to the current instruction set architecture (ISA) name ('sparc', 'sparcv9', or 'i386')

%o Expands to the old value of the path being modified. This is useful for appending or prepending directories to an existing path.

%p Expands to the current platform string (either `uname -i` or the platform string stored in the process core file or crash dump)

%r Expands to the path name of the root directory. An alternate root directory can be specified using the `-R` option. If no `-R` option is present, the root directory is derived dynamically from the path to the `mdb` executable itself. For example, if `/bin/mdb` is executed, the root directory is `/`. If `/net/hostname/bin/mdb` were executed, the root directory would be derived as `/net/hostname`.

%t Expands to the name of the current target. This is either the literal string '`proc`' (a user process or user process core file), or '`kvm`' (a kernel crash dump or the live operating system).

The default include path for 32-bit `mdb` is:

```
%r/usr/platform/%p/lib/adb:%r/usr/lib/adb
```

The default include path for 64-bit `mdb` is:

```
%r/usr/platform/%p/lib/adb/%i:%r/usr/lib/adb/%i
```

`-k` Forces kernel debugging mode. By default, `mdb` attempts to infer whether the object and core file operands refer to a user executable and core dump, or to a pair of operating system crash dump files. The `-k` option forces `mdb` to assume these files are operating system crash dump files. If no object or core operand is specified, but the `-k` option is specified, `mdb` defaults to an object file of `/dev/ksyms` and a core file of `/dev/kmem`. Access to `/dev/kmem` is restricted to group sys.

`-L` Sets default path for locating debugger modules. Modules are loaded automatically on startup or by using the

`::load` dcmd. The path is a sequence of directory names delimited by colon (`:`) characters. The `–L` library path can also contain any of the tokens shown for `–I` above.

`–m`

Disables demand-loading of kernel module symbols. By default, `mdb` processes the list of loaded kernel modules and performs demand loading of per-module symbol tables. If the `–m` option is specified, `mdb` does not attempt to process the kernel module list or provide per-module symbol tables. As a result, `mdb` modules corresponding to active kernel modules are not loaded on startup.

`–M`

Preloads all kernel module symbols. By default, `mdb` performs demand-loading for kernel module symbols: the complete symbol table for a module is read when an address is that module's text or data section is referenced. With the `–M` option, `mdb` loads the complete symbol table of all kernel modules during startup.

`–o` ***option***

Enables the specified debugger option. If the `+o` form of the option is used, the specified option is disabled. Unless noted below, each option is off by default. `mdb` recognizes the following option arguments:

adb — Enable stricter `adb`(1) compatibility. The prompt is set to the empty string and many `mdb` features, such as the output pager, are disabled.

follow_child — The debugger follows the child process if a `fork`(2) system call occurs. By default, the debugger remains attached to the original target process (the parent).

ignoreeof — The debugger does not exit when an `EOF` sequence (`^D`) is entered at the terminal. The `::quit` dcmd must be used to quit.

pager — The output pager is enabled (default).

repeatlast — If a `NEWLINE` is entered as the complete command at the terminal, `mdb` repeats the previous command with the current value of dot. This option is implied by `–o adb`.

–p ***pid***	Attach to and stop the specified process id. `mdb` uses the `/proc/`*pid*`/object/a.out` file as the executable file path name.
–P	Sets the command prompt. The default prompt is '> '.
–R	Sets root directory for path name expansion. By default, the root directory is derived from the path name of the `mdb` executable itself. The root directory is substituted in place of the `%r` token during path name expansion.
–s ***distance***	Sets the symbol matching distance for address-to-symbol-name conversions to the specified *distance*. By default, `mdb` sets the distance to zero, which enables a smart-matching mode. Each ELF symbol table entry includes a value V and size S, representing the size of the function or data object in bytes. In smart mode, `mdb` matches an address A with the given symbol if A is in the range [V, V + S). If any non-zero distance is specified, the same algorithm is used, but S in the given expression is always the specified absolute distance and the symbol size is ignored.
–S	Suppresses processing of the user's `~/.mdbrc` file. By default, `mdb` reads and processes the macro file `.mdbrc` if one is present in the user's home directory, as defined by $HOME. If the –S option is present, this file is not read.
–u	Forces user debugging mode. By default, `mdb` attempts to infer whether the object and core file operands refer to a user executable and core dump, or to a pair of operating system crash dump files. The –u option forces `mdb` to assume these files are not operating system crash dump files.
–V	Sets disassembler version. By default, `mdb` attempts to infer the appropriate disassembler version for the debug target. The disassembler can be set explicitly using the –V option. The `::disasms` dcmd lists the available disassembler versions.
–w	Opens the specified object and core files for writing.
–y	Sends explicit terminal initialization sequences for tty mode. Some terminals, such as `cmdtool`(1), require explicit initialization sequences to switch into a tty mode.

Without this initialization sequence, terminal features such as standout mode might not be available to `mdb`.

APPENDIX **B**

Transition From `crash`

The transition from using the legacy `crash`(1M) utility to using `mdb`(1) is relatively simple: MDB provides most of the “canned” crash commands. The additional extensibility and interactive features of MDB allow the programmer to explore aspects of the system not examined by the current set of commands.

This appendix briefly discusses several features of `crash`(1M) and provides pointers to equivalent MDB functionality.

Command-line Options

The `crash` –`d`, –`n`, and –`w` command-line options are not supported by `mdb`. The `crash` dump file and name list (symbol table file) are specified as arguments to `mdb` in the order of name list, crash dump file. To examine the live kernel, the `mdb` –`k` option should be specified with no additional arguments. Users who want to redirect the output of `mdb` to a file or other output destination, should either employ the appropriate shell redirection operator following the `mdb` invocation on the command line, or use the `::log` built-in dcmd.

Input in MDB

In general, input in MDB is similar to `crash`, except that function names (in MDB, dcmd names) are prefixed with "`::`". Some MDB dcmds accept a leading expression argument that precedes the dcmd name. Like `crash`, string options can follow the dcmd name. If a ! character follows a function invocation, MDB will also create a

pipeline to the specified shell pipeline. All immediate values specified in MDB are interpreted in hexadecimal by default. The radix specifiers for immediate values are different in `crash` and MDB as shown in Table B–1:

TABLE B–1 Radix Specifiers

crash	mdb	Radix
0x	0x	hexadecimal (base 16)
0d	0t	decimal (base 10)
0b	0i	binary (base 2)

Many `crash` commands accepted slot numbers or slot ranges as input arguments. The Solaris operating environment is no longer structured in terms of slots, so MDB dcmds do not provide support for slot-number processing.

Functions

crash function	mdb dcmd	comments
?	::dcmds	List available functions.
!command	!command	Escape to the shell and execute commmand.
base	=	In mdb, the = format character can be used to convert the left-hand expression value to any of the known formats. Formats for octal, decimal, and hexadecimal are provided.
callout	::callout	Print the callout table.
class	::class	Print scheduling classes.
cpu	::cpuinfo	Print information about the threads dispatched on the system CPUs. If the contents of a particular CPU structure are needed, the user should apply the `$<cpu` macro to the CPU address in mdb.

crash function	mdb dcmd	comments
help	::help	Print a description of the named dcmd, or general help information.
kfp	::regs	The mdb ::regs dcmd displays the complete kernel register set, including the current stack frame pointer. The $C dcmd can be used to display a stack backtrace including frame pointers.
kmalog	::kmalog	Display events in kernel memory allocator transaction log.
kmastat	::kmastat	Print kernel memeory allocator transaction log.
kmausers	::kmausers	Print information about the medium and large users of the kernel memory allocator that have current memory allocations.
mount	::fsinfo	Print information about mounted file systems.
nm	::nm	Print symbol type and value information.
od	::dump	Print a formatted memory dump of a given region. In mdb, ::dump displays a mixed ASCII and hexadecimal display of the region.
proc	::ps	Print a table of the active processes.
quit	::quit	Quit the debugger.
rd	::dump	Print a formatted memory dump of a given region. In mdb, ::dump displays a mixed ASCII and hexadecimal display of the region.
redirect	::log	In mdb, output for input and output can be globally redirected to a log file using ::log.
search	::kgrep	In mdb, the ::kgrep dcmd can be used to search the kernel's address space for a particular value. The pattern match built-in dcmds can also be used to search the physical, virtual, or object files address spaces for patterns.
stack	::stack	The current stack trace can be obtained using ::stack. The stack trace of a particular kernel thread can be determined using the ::findstack dcmd. A memory dump of the current stack can be obtained using the / or ::dump dcmds and the current stack pointer. The `$<stackregs` macro can be applied to a stack pointer to obtain the per-frame saved register values.

crash function	mdb dcmd	comments
status	::status	Display status information about the system or dump being examined by the debugger.
stream	::stream	The mdb ::stream dcmd can be used to format and display the structure of a particular kernel STREAM. If the list of active STREAM strucures is needed, the user should execute ::walk stream_head_cache in mdb and pipe the resulting addresses to an appropriate formatting dcmd or macro.
strstat	::kmastat	The ::kmastat dcmd displays a superset of the information reported by the `strstat()` function.
trace	::stack	The current stack trace can be obtained using ::stack. The stack trace of a particular kernel thread can be determined using the ::findstack dcmd. A memory dump of the current stack can be obtained using the / or ::dump dcmds and the current stack pointer. The `$<stackregs` macro can be applied to a stack pointer to obtain the per-frame saved register values.
var	$<v	Print the tunable system parameters in the global `var` structure.
vfs	::fsinfo	Print information about mounted file systems.
vtop	::vtop	Print the physical address translation of the given virtual address.