Solaris Dynamic Tracing Guide

Beta

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–3016–01
November 2003

031105@6671

Contents

Tables

Figures

Examples

Preface

DTrace is a comprehensive dynamic tracing framework for the Solaris™ Operating System. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs. The *Solaris Dynamic Tracing Guide* describes how to use DTrace to observe, debug, and tune system behavior. This book also includes a complete reference for bundled DTrace observability tools and the D programming language.

Who Should Use This Book

If you have ever wanted to understand the behavior of your system, DTrace is the tool for you. DTrace is a comprehensive dynamic tracing facility that is built into Solaris that can be used by administrators and developers on live production systems to examine the behavior of both user programs and of the operating system itself. DTrace will allow you to explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. As we'll see, DTrace lets you create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions you can formulate using the DTrace D programming language.

DTrace allows all Solaris users to:

- Dynamically enable and manage thousands of probes
- Dynamically associate logical predicates and actions with probes
- Dynamically manage trace buffers and buffer policies
- Display and examine trace data from the live system or a crash dump

DTrace allows Solaris developers and administrators to:

- Implement custom scripts that use the DTrace facility

- Implement layered tools that use DTrace to retrieve trace data

This guide will teach you everything you need to know about using DTrace. Basic familiarity with a programming language such as C or a scripting language such as `awk`(1) or `perl`(1) will help you learn DTrace and the D programming language faster, but you need not be an expert in any of these areas. If you have never written a program or script before in any language, "Related Books" on page 20 provides references to other documents you might find useful.

How This Book Is Organized

The first chapter provides a tour of the entire DTrace facility and introduces all readers to its key concepts. Next, a series of chapters provide a reference for the D programming language and all of the built-in language functions. Then a series of chapters discuss the details of each DTrace provider and instrumentation utility. The final chapters discuss more advanced topics.

Related Books

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

- Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, 1988. ISBN 0–13–110370–9
- Vahalia, Uresh. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996. ISBN 0-13-101908-2
- Mauro, Jim and McDougall, Richard. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001. ISBN 0-13-022496-0

Note – In this document the term "x86" refers to the Intel 32–bit family of microprocessor chips and compatible microprocessor chips made by AMD.

Accessing Sun Documentation Online

The docs.sun.com[SM] 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`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at http://docs.sun.com.

Typographic Conventions

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:`
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

Introduction

Welcome to Dynamic Tracing in the Solaris™ Operating System! If you have ever wanted to understand the behavior of your system, DTrace is the tool for you. DTrace is a comprehensive dynamic tracing facility that is built into Solaris that can be used by administrators and developers on live production systems to examine the behavior of both user programs and of the operating system itself. DTrace will allow you to explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. As we'll see, DTrace lets you create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions you can formulate using the DTrace D programming language. We'll begin with a quick introduction to DTrace and show you how to write your very first D program. Later, we'll discuss the complete set of rules for programming in D as well as tips and techniques for performing in-depth analysis of your system.

1.1 Getting Started

DTrace helps you understand a software system by permitting you to dynamically modify the operating system kernel and user processes to record additional data that you specify at locations of interest, called *probes*. A probe is a location or activity to which DTrace can bind a request to perform a set of *actions*, like recording a stack trace, a timestamp, or the argument to a function. Probes are like little programmable sensors scattered all over your Solaris system in interesting places. If you want to figure out what's going on, you use DTrace to program the appropriate sensors to record the information that is of interest to you. Then, as each probe *fires*, DTrace will gather the data from your probes and report it back to you. If you don't specify any actions for a probe, DTrace will just take note of each time the probe fires.

Every probe in DTrace has two names: a unique integer ID and a human-readable string name. We're going to start learning DTrace by building some very simple requests using the probe named `BEGIN`, which fires once each time you start a new tracing request. You can use the `dtrace`(1M) utility's `-n` option to enable a probe using its string name. Type in the following command:

```
# dtrace -n BEGIN
```

After a brief pause, you will see DTrace tell you that one probe was enabled and you will see a line of output telling you that the `BEGIN` probe fired. Once you see this, `dtrace` will remain paused waiting for other probes to fire. Since we haven't enabled any others and `BEGIN` only fires once, just type Control-C in your shell to abort `dtrace` and return to your shell prompt:

```
# dtrace -n BEGIN
dtrace: description 'BEGIN' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN
^C
#
```

The output tells you that the probe named `BEGIN` fired once and both its name and integer ID, 1, are printed. Notice that by default, the integer name of the CPU on which this probe fired is displayed. In this example, the CPU column indicates that the `dtrace` command was executing on CPU 0 when the probe fired.

DTrace lets you construct requests using arbitrary numbers of probes and actions. Let's create a simple request using two probes by adding the `END` probe to our command, which fires once when tracing is completed. Type the following command, and then again type Control-C in your shell after you see the line of output for the `BEGIN` probe:

```
# dtrace -n BEGIN -n END
dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN
^C
  0      2                        :END
#
```

As you can see, when you typed Control-C to abort `dtrace`, that triggered the `END` probe and `dtrace` reported this probe firing to you before exiting.

Now that we understand a little bit about naming and enabling probes, we're ready to write the DTrace version of everyone's first program, "Hello, World." In addition to constructing DTrace experiments on the command line, you can also write them in text files using the D programming language. Open your favorite text editor and create a new file called `hello.d` and type in your first D program:

EXAMPLE 1–1 hello.d: Hello, World from the D Programming Language

```
BEGIN
{
    trace("hello, world");
    exit(0);
}
```

After you have saved your program, you can run it using the dtrace `-s` option. Type in the following command:

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0        1                      :BEGIN   hello, world
#
```

As you can see, `dtrace` printed the same output we saw before followed by the text "`hello, world`". Unlike our previous example, we didn't have to wait and type Control-C, either. These changes were the result of the *actions* we specified for our `BEGIN` probe in `hello.d`. Let's explore the structure of our D program in more detail in order to understand what happened.

Each D program consists of a series of *clauses*, each describing one or more probes to enable, and an optional set of actions to perform when the probe fires. The actions are listed as a series of statements enclosed in braces { } following the probe name. Each statement ends with a semicolon (`;`). Our first statement uses the function `trace()` to indicate that DTrace should record the specified argument, the string "`hello, world`", when the `BEGIN` probe fires, and then print it out. The second statement uses the function `exit()` to indicate that DTrace should cease tracing and exit the `dtrace` command. DTrace provides a set of useful functions like `trace()` and `exit()` for you to call in your D programs. To call a function, you specify its name followed by a parenthesized list of arguments. The complete set of D functions is described later in Chapter 10.

By now, if you're familiar with the C programming language, you've probably realized from the name and our examples that DTrace's D programming language is very similar to C. Indeed, D is derived from a large subset of C combined with a special set of functions and variables to help make tracing easy. We'll learn more about these features in subsequent chapters. If you've written a C program before, you will be able to immediately transfer most of your knowledge to building tracing programs in D. If you've never written a C program before, learning D is still very easy: don't be intimidated! We're going to teach you all the syntax shortly. But before we do that, we're going to take a step back from language rules and learn more about how DTrace works, and then we'll return to learning how to build more interesting D programs.

1.2 Providers and Probes

In our earlier examples, we learned to use two simple probes named `BEGIN` and `END`. But where did these probes come from? DTrace probes come from a set of kernel modules called *providers*, each of which knows how to perform a particular kind of instrumentation to create probes. When you use DTrace, each provider is given an opportunity to publish the probes it can provide to the DTrace framework. You can then enable and bind your tracing actions to any of the probes that have been published. To list all of the available probes on your system, type the command:

```
# dtrace -l
  ID   PROVIDER            MODULE          FUNCTION NAME
   1     dtrace                                     BEGIN
   2     dtrace                                     END
   3     dtrace                                     ERROR
   4   lockstat           genunix       mutex_enter adaptive-acquire
   5   lockstat           genunix       mutex_enter adaptive-block
   6   lockstat           genunix       mutex_enter adaptive-spin
   7   lockstat           genunix       mutex_exit  adaptive-release

   ... many lines of output omitted ...

#
```

It may take a while for all the output to scroll by. To count up all your probes you can type the command:

```
# dtrace -l | wc -l
        25499
```

You will get a different total on your machine, as the number of probes varies depending on your operating platform and the software you have installed. As you can see, there are a very large number of probes available to you so you can peer into every previously dark corner of the system. In fact, even this output isn't the complete list because, as we'll see later, some providers offer the ability to create new probes on-the-fly based on your tracing requests, making the actual number of DTrace probes virtually unlimited!

Now look back at the output from **`dtrace -l`** in your terminal window. Notice that each probe has the two names we mentioned earlier, an integer ID and a human-readable name. You should now see that the human readable name is composed of four parts, shown as separate columns in the `dtrace` output. The four parts of a probe name are:

Provider	The name of the DTrace provider that is publishing this probe. The provider name typically corresponds to the name of the DTrace kernel module that performs the instrumentation to enable the probe.
Module	If this probe corresponds to a specific program location, the name of the module in which the probe is located. This is either the name of a kernel module or the name of a user library.
Function	If this probe corresponds to a specific program location, the name of the program function in which the probe is located.
Name	The final component of the probe name is a name that gives you some idea of the probe's semantic meaning, such as `BEGIN` or `END`.

When we write out the full human-readable name of a probe in this book or in our D programs, we will most often write all four parts of the name separated by colons like this:

provider:*module*:*function*:*name*

Notice that some of the probes in the list do not have a module and function, such as the `BEGIN` and `END` probes we used earlier. Some probes leave these two fields blank because these probes do not correspond to any specific instrumented program function or location. Instead, these probes refer to a more abstract concept like the idea of the end of your tracing request. We will refer to a probe that has a module and function as part of its name as an *anchored probe*, and one that does not as *unanchored*.

By convention, if you do not specify all of the fields of a probe name, then DTrace matches your request to *all* of the probes that have matching values in the parts of the name that you do specify. In other words, when we used the probe name `BEGIN` earlier, we were actually telling DTrace to match any probe whose name field is `BEGIN`, regardless of the value of the provider, module, and function fields. As it happens, there is only one probe matching that description, so the result is the same. But we now know that the true name of the `BEGIN` probe is `dtrace:::BEGIN`, which means this is a probe provided by the DTrace framework itself and is not anchored to any function. Therefore we could also have written our `hello.d` program as follows and obtained the same result:

```
dtrace:::BEGIN
{
    trace("hello, world");
    exit(0);
}
```

Now that we understand where probes originate from and how they are named, we're going to learn a little more about what happens when you enable probes and ask DTrace to do something, and then we'll return to our whirlwind tour of D.

1.3 Compilation and Instrumentation

When you write traditional programs in Solaris, you use a compiler to convert your program from source code into object code that you can execute. When you use the `dtrace` command you are invoking the compiler for the D language that we used earlier to write the `hello.d` program. Once your program is compiled, it is sent into the operating system kernel for execution by DTrace. There the probes that are named in your program are enabled and the corresponding provider performs whatever instrumentation is needed to activate them.

All of the instrumentation in DTrace is completely dynamic: probes are enabled discretely only when you are using them. No instrumented code is present for inactive probes, so there is no performance degradation of any kind to your system when you are not using DTrace. Once your experiment is complete and the `dtrace` command exits, all of the probes you used are automatically disabled and their instrumentation is removed, returning your system to its exact original state. There is effectively no difference between a system where DTrace is not active and one where the DTrace software is not installed.

The instrumentation for each probe is performed dynamically on the live running operating system or on user processes you select. The system is not quiesced or paused in any way, and instrumentation code is added only for the probes that you enable. As a result, the probe effect of using DTrace is limited to exactly what you ask DTrace to do: no extraneous data is traced, no one big "tracing switch" is turned on in the system, and all of the DTrace instrumentation is designed to be as efficient as possible. These features allow you to use DTrace in production to solve real problems in real time.

The DTrace framework also provides support for an arbitrary number of virtual clients. You can run as many simultaneous DTrace experiments and commands as you like, limited only by your system's memory capacity, and they all operate independently using the same underlying instrumentation. This same capability also permits any number of distinct users on the system to take advantage of DTrace simultaneously: developers, administrators, and service personnel can all work together or on distinct problems on the same system using DTrace without interfering with one another.

Unlike programs written in C and C++ and similar to programs written in the Java™ programming language, DTrace D programs are compiled into a safe intermediate form which is used for execution when your probes fire. This intermediate form is validated for safety when your program is first examined by the DTrace kernel software. The DTrace execution environment also handles any run-time errors that might occur during your D program's execution, including dividing by zero, dereferencing invalid memory, and so on, and reports them to you. As a result, you can never construct an unsafe program that would cause DTrace to inadvertently damage the Solaris kernel or one of the processes running on your system. These

safety features allow you to use DTrace in a production environment without worrying about crashing or corrupting your system. If you make a programming mistake, DTrace will report your error to you, disable your instrumentation, and you can correct your mistake and try again. We'll learn more about DTrace error reporting and debugging features later in the book.

The diagram below shows the different components of the DTrace architecture we have learned about so far, including providers, probes, the DTrace kernel software, and the `dtrace` command.

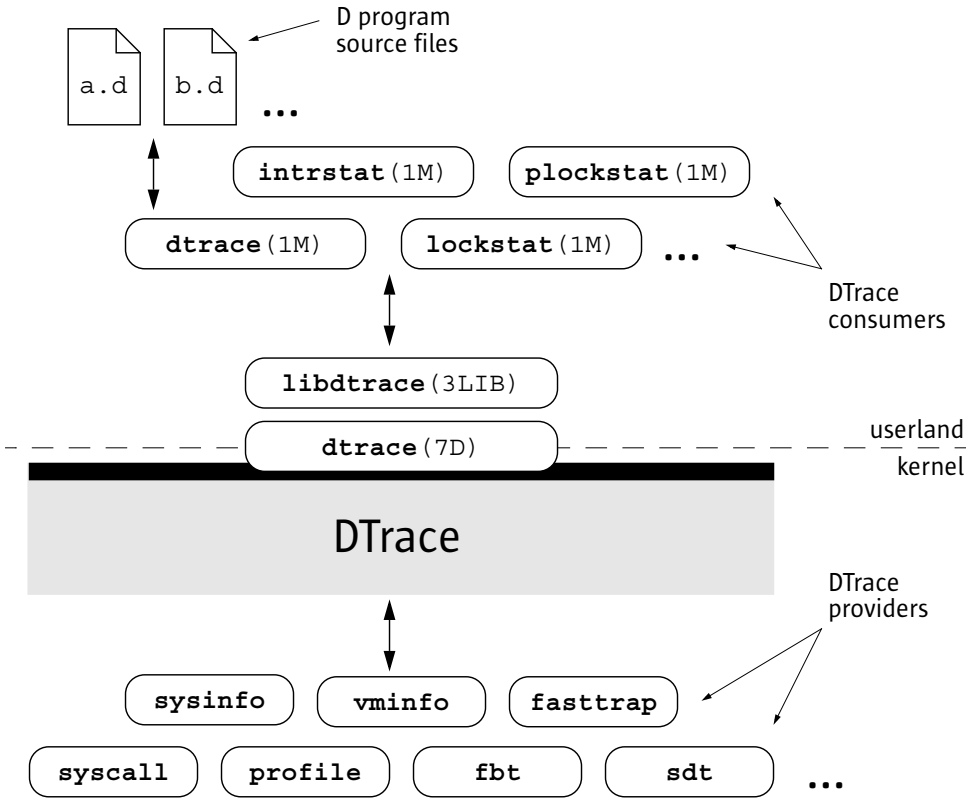

FIGURE 1–1 Overview of the DTrace Architecture and Components

Now that we understand how DTrace works, let's return to our tour of the D programming language and start writing some more interesting programs.

1.4 Variables and Arithmetic Expressions

Our next example program makes use of the DTrace `profile` provider to implement a simple time-based counter. The profile provider is able to create new probes based on the descriptions found in your D program. If you create a probe named `profile:::tick-`*n*`sec` for some integer *n*, the profile provider will create a probe that fires every *n* seconds. Type in the following source code for our second example and save it in a file named `counter.d`:

```
/*
 * Count off and report the number of seconds elapsed
 */
dtrace:::BEGIN
{
    i = 0;
}

profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}

dtrace:::END
{
    trace(i);
}
```

When executed, the program counts off the number of elapsed seconds until you type Control-C, and then prints the total at the end:

```
# dtrace -s counter.d
dtrace: script 'counter.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0  25499                        :tick-1sec         1
  0  25499                        :tick-1sec         2
  0  25499                        :tick-1sec         3
  0  25499                        :tick-1sec         4
  0  25499                        :tick-1sec         5
  0  25499                        :tick-1sec         6
^C
  0      2                             :END         6
#
```

The first three lines of our program are a comment to explain what the program does. Similar to C, C++, and the Java programming language, the D compiler ignores any characters between the `/*` and `*/` symbols. Comments can be used anywhere in your D program, including both inside and outside your probe clauses.

The BEGIN probe clause defines a new variable named i and assigns it the integer value zero using the statement:

```
i = 0;
```

Unlike C, C++, and the Java programming language, D variables can be created by simply using them in a program statement; explicit variable declarations are not required. When a variable is used for the first time in your program, the type of the variable is set based on the type of its first assignment. Each variable has only one type over the lifetime of your program, so subsequent references must conform to the same type as the initial assignment. In counter.d, we first assign i the integer constant zero, so its type is set to int. D provides the same basic integer data types as C, including:

char	Character or single byte integer
int	Default integer
short	Short integer
long	Long integer
long long	Extended long integer

The sizes of these types are dependent on the operating system kernel's data model; we'll review these details later. D also provides built-in friendly names for signed and unsigned integer types of various fixed sizes, as well as thousands of other types that are defined by the operating system. For now we will only use simple types in our examples.

The central part of counter.d is the probe clause that increments our counter i:

```
profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}
```

This clause names the probe profile:::tick-1sec, which tells the profile provider to create a new probe which fires once per second on an available processor. The clause contains two statements, the first assigning i to the previous value plus one, and the second tracing the new value of i. All the usual C arithmetic operators are available in D; we'll review the complete list in Chapter 2. Also as in C, the ++ operator can be used as shorthand for incrementing the corresponding variable by one. The trace() function takes any D expression as its argument, so we could write counter.d more concisely as follows:

```
profile:::tick-1sec
{
```

```
    trace(++i);
}
```

If you want to explicitly control the type of the variable `i`, you can surround the desired type in parentheses when you assign it in order to *cast* the integer zero to a specific type. For example, if you wanted to determine the maximum size of a `char` in D, you could change the `BEGIN` clause as follows:

```
dtrace:::BEGIN
{
    i = (char)0;
}
```

After running `counter.d` for a while, you should see the traced value grow and then wrap around back to zero. If you grow impatient waiting for the value to wrap, try changing the `profile` probe name to `profile:::tick-100msec` to make a counter that increments once every 100 milliseconds, or 10 times per second.

1.5 Predicates

One major difference between D and other programming languages such as C, C++, and the Java programming language is the absence of control-flow constructs such as if-statements and loops. D program clauses are written as single straight-line statement lists that trace an optional, fixed amount of data. D does provide the ability to conditionally trace data and modify control flow using logical expressions called *predicates* that can be used to prefix program clauses. A predicate expression is evaluated at probe firing time prior to executing any of the statements associated with the corresponding clause. If the predicate evaluates to true, represented by any non-zero value, the statement list is executed. If the predicate is false, represented by a zero value, none of the statements are executed and the probe firing is ignored.

Type in the following source code for the next example and save it in a file named `countdown.d`:

```
dtrace:::BEGIN
{
    i = 10;
}

profile:::tick-1sec
/i > 0/
{
    trace(i--);
}
```

```
profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

This D program implements a 10-second countdown timer using predicates. When executed, `countdown.d` counts down from 10 and then prints a message and exits:

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
    0  25499                       :tick-1sec        10
    0  25499                       :tick-1sec         9
    0  25499                       :tick-1sec         8
    0  25499                       :tick-1sec         7
    0  25499                       :tick-1sec         6
    0  25499                       :tick-1sec         5
    0  25499                       :tick-1sec         4
    0  25499                       :tick-1sec         3
    0  25499                       :tick-1sec         2
    0  25499                       :tick-1sec         1
    0  25499                       :tick-1sec   blastoff!
#
```

We first use the dtrace `BEGIN` probe to initialize an integer `i` to 10 to begin our countdown. Next, as in the previous example, we use the `tick-1sec` probe to implement a timer that fires once per second. Notice that in `countdown.d`, we use the same `tick-1sec` probe description in two different clauses, each with a different predicate and action list. The predicate is a logical expression surrounded by enclosing slashes / / that appears after the probe name and before the braces { } that surround the clause statement list.

The first predicate tests whether `i` is greater than zero, indicating that our timer is still running:

```
profile:::tick-1sec
/i > 0/
{
    trace(i--);
}
```

The relational operator > means *greater than* and returns the integer value zero for false and one for true. All of the C relational operators are supported in D; the complete list is found in Chapter 2. If `i` is not yet zero, we trace `i` and then decrement it by one using the `--` operator.

Our second predicate uses the == operator to return true when `i` is exactly equal to zero, indicating that our countdown is complete:

```
profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

Similar to our first example, `hello.d`, we use a sequence of characters enclosed in double quotes, called a *string constant*, to print a final message when the countdown is complete. The `exit()` function is then used to exit `dtrace` and return to the shell prompt.

If you look back at the structure of `countdown.d`, you will see that by creating two clauses with the same probe description but different predicates and actions, we effectively created the logical flow:

```
i = 10
once per second,
   if i is greater than zero
      trace(i--);
   otherwise if i is equal to zero
      trace("blastoff!");
      exit(0);
```

When you wish to write complex programs using predicates, it is sometimes useful to first visualize your algorithm in this manner, and then transform each path of your conditional constructs into a separate clause and predicate.

Now let's combine predicates with a new provider, the `syscall` provider, and create our first real D tracing program. The `syscall` provider permits us to enable probes on entry to or return from any Solaris system call. We're going to use DTrace to observe every time your shell performs a `read`(2) or `write`(2) system call. First, open two terminal windows, one to use for DTrace and the other containing the shell process we're going to watch. In the second window, type the following command to obtain the process ID of this shell:

```
# echo $$
12345
```

Now go back to your first terminal window and type in the following D program and save it in a file named `rw.d`. As you type in the program, replace the integer constant `12345` with the process ID of your shell that was printed in response to your `echo` command.

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{

}
```

Notice that we leave the body of the `rw.d`'s probe clause empty since we are only interested in notification of probe firings and not in tracing any additional data. Once you're done typing in `rw.d`, use `dtrace` to start your experiment and then go to your second shell window and type a few commands, pressing return after each. As you type, you should see `dtrace` report probe firings in your first window like this:

```
# dtrace -s rw.d
dtrace: script 'rw.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
    0     34                      write:entry
    0     32                       read:entry
    0     34                      write:entry
    0     32                       read:entry
    0     34                      write:entry
    0     32                       read:entry
    0     34                      write:entry
    0     32                       read:entry
...
```

You are now watching your shell perform `read`(2) and `write`(2) system calls to read a character from your terminal window and echo back the result! We put together many of the concepts we've learned so far and a few new ones to create this example. First, since we wanted to instrument `read`(2) and `write`(2) in the same manner, we created a single probe clause with multiple probe descriptions by separating the descriptions with commas like this:

```
syscall::read:entry, syscall::write:entry
```

Next we created a predicate that matched only those system calls that were executed by our shell process:

```
/pid == 12345/
```

Our predicate uses the predefined DTrace variable `pid`, which always evaluates to the process ID associated with the thread that fired the corresponding probe. DTrace provides many built-in variable definitions for useful things like the process ID. Here is a list of a few DTrace variables you can use to write your first D programs:

Variable Name	Data Type	Meaning
`errno`	`int`	Current `errno` value for system calls
`execname`	`string`	Name of the current process's executable file
`pid`	`pid_t`	Process ID of the current process
`tid`	`id_t`	Thread ID of the current thread
`probeprov`	`string`	Current probe description's provider field

Variable Name	Data Type	Meaning
probemod	string	Current probe description's module field
probefunc	string	Current probe description's function field
probename	string	Current probe description's name field

Now that we've written a real instrumentation program, try experimenting with it on different processes running on your system by changing the process ID and the system call probes that are instrumented. Then, let's make one more simple change and turn `rw.d` into a very simple version of a system call tracing tool like `truss`(1). An empty probe description field acts as a wildcard, matching any probe, so change your program to the following new source code to trace *any* system call executed by your shell:

```
syscall:::entry
/pid == 12345/
{

}
```

Try typing a few commands in the shell such as `cd`, `ls`, and `date` and see what your DTrace program reports.

1.6 Output Formatting

System call tracing is a very powerful way to observe the behavior of most user processes. If you've used the Solaris `truss`(1) utility before as an administrator or developer, you've probably learned that it's the Solaris equivalent of a trusty pocket knife you always keep around for whenever there is a problem. If you've never used `truss` before, give it a try right now by typing this command into one of your shells:

```
$ truss date
```

You will see a formatted trace of all the system calls executed by `date`(1) followed by its one line of output at the end. For our next example, we're going to improve our earlier `rw.d` program by formatting its output to look more like `truss`(1) so we can more easily understand the output. Type in the following program and save it in a file called `trussrw.d`:

EXAMPLE 1–2 trussrw.d: Trace System Calls with `truss`(1) Output Format

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
```

EXAMPLE 1–2 trussrw.d: Trace System Calls with `truss`(1) Output Format *(Continued)*

```
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}

syscall::read:return, syscall::write:return
/pid == 12345/
{
    printf("\t\t = %d\n", arg1);
}
```

As before, change the process ID from `12345` to the process ID of the shell or other process you are going to examine when you run `dtrace`. To execute `trussrw.d`, we're going to specify the `dtrace` option `-q` along with `-s`. The `-q` option tells `dtrace` to be quiet and suppress its usual printing of a header line and the CPU and ID columns we saw before. This way, we will only see the output for the data that we explicitly traced. Type in the following command and then press return a few times in your shell:

```
# dtrace -q -s trussrw.d
                      = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)^C
#
```

Now let's examine our D program and its output in more detail. First, we created a clause similar to our earlier program to instrument each of the shell's calls to `read`(2) and `write`(2). But for this example, we used a new function, `printf()`, to trace data and print it out in a specific format:

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}
```

The `printf()` function combines the ability to trace data, as if by the `trace()` function we used earlier, with the ability to output the data and other text in a specific format that you describe. The `printf()` function tells DTrace to trace the data associated with each argument after the first argument, and then to format the results using the rules described by the first `printf()` argument, known as a *format string*.

The format string is a regular string that contains any number of format conversions, each beginning with the `%` character, that describe how to format the corresponding argument. The first conversion in the format string corresponds to the second `printf()` argument, the second conversion to the third argument, and so on. All of the text between conversions is printed verbatim. The character following the `%` conversion character describes the format to use for the corresponding argument. Here are the meanings of the three format conversions used in `trussrw.d`:

`%d`	Print the corresponding value as a decimal integer
`%s`	Print the corresponding value as a string
`%x`	Print the corresponding value as a hexadecimal integer

DTrace `printf()` works just like the C `printf`(3C) library routine or the shell `printf`(1) utility. If you've never seen `printf()` before, don't worry: we'll explain all the formats and options in detail in Chapter 12. You should read this chapter carefully even if you're already familiar with `printf()` from another language. In D, `printf()` is provided as a built-in and some new format conversions are available to you designed specifically for DTrace.

To help you write correct programs, the D compiler validates each `printf()` format string for you against its argument list. Try changing `probefunc` in the clause above to the integer `123`. If you run the modified program you will see an error message telling you that the string format conversion `%s` is not appropriate for use with an integer argument:

```
# dtrace -q -s trussrw.d
dtrace: failed to compile script trussrw.d: line 4: printf( )
       argument #2 is incompatible with conversion #1 prototype:
            conversion: %s
             prototype: char [] or string (or use stringof)
              argument: int
#
```

To print the name of the read or write system call and its arguments, we use the `printf()` statement:

```
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
```

to trace the name of the current probe function and the first three integer arguments to the system call, available in the DTrace variables `arg0`, `arg1`, and `arg2`. We'll learn more about probe arguments in Chapter 3. The first argument to `read`(2) and `write`(2) is a file descriptor, which we print in decimal. The second argument is a buffer address, which we format as a hexadecimal value. The final argument is the buffer size, which we again format as a decimal value. We use the format specifier `%4d` for the third argument to indicate that the value should be printed using the `%d` format conversion with a minimum field width of 4 characters. If the integer is less than 4 characters wide, `printf()` will insert extra blanks for us to make the output line up nicely.

To print the result of the system call and complete each line of output, we use the following clause:

```
syscall::read:return, syscall::write:return
/pid == 12345/
{
    printf("\t\t = %d\n", arg1);
}
```

Notice that the `syscall` provider also publishes a probe named `return` for each system call in addition to `entry`. The DTrace variable `arg1` for the syscall `return` probes evaluates to the system call's return value. We format the return value as a decimal integer. The character sequences beginning with backwards slashes in the format string expand to tab (`\t`) and newline (`\n`) respectively. These *escape sequences* help you print or record characters that are difficult to type. D supports the same set of escape sequences as C, C++, and the Java programming language. The complete list of escape sequences is found in Chapter 2.

1.7 Arrays

D permits you to define variables that are integers, as well as other types to represent strings and composite types called *structs* and *unions*. If you are familiar with C programming, you'll be happy to know you can use any type in D that you can in C. If you're not a C expert, don't worry: we'll cover all the different kinds of data types later in Chapter 2. D also supports a special kind of variable called an *associative array*. An associative array is similar to a normal array in that it associates a set of keys with a set of values, but in an associative array the keys are not limited to integers of a fixed range.

D associative arrays can be indexed by a list of one or more values of any type. Together the individual key values form a *tuple* that is used to index into the array and access or modify the value corresponding to that key. Every tuple used with a given associative array must conform to the same type signature; that is, each tuple key

must be of the same length and have the same key types in the same order. The value associated with each element of a given associative array is also of a single fixed type for the entire array. For example, the following D statement defines a new associative array `a` of value type `int` with the tuple signature `[ string, int ]` and stores the integer value 456 in the array:

```
a["hello", 123] = 456;
```

Once an array is defined, its elements can be accessed like any other D variable. For example, the following D statement modifies the array element previously stored in `a` by incrementing the value from 456 to 457:

```
a["hello", 123]++;
```

The values of any array elements you have not yet assigned are set to zero. Now let's use an associative array in a D program. Type in the following program and save it in a file named `rwtime.d`:

EXAMPLE 1–3 rwtime.d: Time `read`(2) and `write`(2) Calls

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
}

syscall::read:return, syscall::write:return
/ts[probefunc] != 0 && pid == 12345/
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

As usual, change the process ID `12345` to the process ID of one of your shells before running the program. When you execute `rwtime.d` and then type a few shell commands, you'll see the amount time elapsed during each system call. Type in the following dtrace command and then press return a few times in your other shell:

```
# dtrace -s rwtime.d
dtrace: script 'rwtime.d' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0     33                      read:return 22644 nsecs
  0     33                      read:return 3382 nsecs
  0     35                     write:return 25952 nsecs
  0     33                      read:return 916875239 nsecs
  0     35                     write:return 27320 nsecs
  0     33                      read:return 9022 nsecs
  0     33                      read:return 3776 nsecs
  0     35                     write:return 17164 nsecs
...
^C
#
```

To trace the elapsed time for each system call, we need to instrument both the entry to and return from `read`(2) and `write`(2) and sample the time at each point. Then, on return from a given system call, we want to compute the difference between our first and second timestamp. We could use separate variables for each system call, but this would make the program annoying to extend to additional system calls. Instead, it's easier to use an associative array indexed by the probe function name. Here is the first probe clause:

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
}
```

We define an array named `ts` and assign the appropriate member the value of the DTrace variable `timestamp`. This variable returns the value of an always-incrementing nanosecond counter, similar to the Solaris library routine `gethrtime`(3). Once we have saved the entry timestamp, we use the corresponding return probe to sample `timestamp` again and report the difference between the current time and the saved value:

```
syscall::read:return, syscall::write:return
/ts[probefunc] != 0 && pid == 12345 /
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

The predicate on the return probe requires that we are tracing the appropriate process and that the corresponding `entry` probe has already fired and assigned `ts[probefunc]` a non-zero value. This trick eliminates invalid output when we first start DTrace. If our shell is already waiting in a `read`(2) system call for input when we execute `dtrace`, the `read:return` probe will fire without a preceding `read:entry` for this first `read`(2) and `ts[probefunc]` will evaluate to zero because it has not yet been assigned.

1.8 External Symbols and Types

DTrace instrumentation executes inside the Solaris operating system kernel, so in addition to accessing special DTrace variables and probe arguments, you can also access kernel data structures, symbols, and types. These capabilities allow advanced DTrace users, administrators, service personnel, and driver developers to examine low-level behavior of the operating system kernel and device drivers. The reading list at the start of this book includes books that can help you learn more about Solaris operating system internals.

D uses the backquote character (`) as a special scoping operator for accessing symbols that are defined in the operating system and not in your D program. For example, the Solaris kernel contains a C declaration of a system tunable named `kmem_flags` for enabling memory allocator debugging features (see the Tunable Parameters Guide for more information about `kmem_flags`). This tunable is declared in C in the kernel source code as follows:

```
int kmem_flags;
```

To trace the value of this variable in a D program, you can write the D statement:

```
trace(`kmem_flags);
```

DTrace associates each kernel symbol with the type used for it in the corresponding operating system C code, providing you easy source-based access to the native operating system data structures. Kernel symbol names are kept in a separate namespace from D variable and function identifiers, so you never need to worry about these names conflicting with your D variables. We'll discuss more about how to access kernel data structures later on in the book.

We have now completed our whirlwind tour of DTrace and you've learned many of the basic DTrace building blocks necessary to build larger and more complex D programs. We're now going to learn the complete set of rules for D and see how DTrace can make complex performance measurements and functional analysis of the system easy. Later, we'll see how to use DTrace to connect user application behavior to system behavior, giving you the capability to analyze your entire software stack.

We've only just begun!

CHAPTER 2

Types, Operators, and Expressions

D provides the ability to access and manipulate a variety of data objects: variables and data structures can be created and modified, data objects defined in the operating system kernel and user processes can be accessed, and integer, floating-point, and string constants can be declared. D provides a superset of the ANSI-C operators that are used to manipulate objects and create complex expressions. In this chapter we will learn the detailed set of rules for types, operators, and expressions.

2.1 Identifier Names and Keywords

D identifier names are composed of upper and lower-case letters, digits, and underscores where the first character must be a letter or underscore. All identifier names beginning with an underscore (_) are reserved for use by the D system libraries, described later in this guide. You should avoid using such names in your D programs. By convention, we typically use mixed-case names for variables and all upper-case names for constants.

D language keywords are special identifiers reserved for use in the programming language syntax itself. These names are always specified in lower-case and may not be used for the names of D variables.

TABLE 2–1 D Keywords

`auto`*	`goto`*	`string`+
`break`*	`if`*	`stringof`+
`case`*	`import`*+	`struct`
`char`	`inline`	`switch`*

TABLE 2–1 D Keywords *(Continued)*

const	int	this+
continue*	long	translator+
counter*+	offsetof+	typedef
default*	register*	union
do*	restrict*	unsigned
double	return*	void
else*	self+	volatile
enum	short	while*
extern	signed	xlate+
float	sizeof	
for*	static*	

D reserves for use as keywords a superset of the ANSI-C keywords. The keywords reserved for future use by the D language are marked with "*". The D compiler will produce a syntax error if you attempt to use a keyword that is reserved for future use. The keywords defined by D but not defined by ANSI-C are marked with "+". As we discussed in Chapter 1, D provides the complete set of types and operators found in ANSI-C. The major difference in D progamming is the absence of control-flow constructs, as we discussed earlier. Keywords associated with control-flow in ANSI-C are reserved for future use in D.

2.2 Data Types and Sizes

D provides fundamental data types for integers and floating-point constants. Arithmetic may only be performed on integers in D programs. Floating-point constants may be used to initialize data structures, but floating-point arithmetic is not permitted in D. D provides a 32-bit and 64-bit data model for use in writing programs. The data model used when executing your program is the native data model associated with the active operating system kernel. You can determine the native data model for your system using `isainfo`(1) `-b`.

The names of the integer types and their sizes in each of the two data models are shown in the table below. Integers are always represented in twos-complement form in the native byte-encoding order of your system.

TABLE 2–2 D Integer Data Types

Type Name	32–bit Size	64–bit Size
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes

Integer types may be prefixed with the `signed` or `unsigned` qualifier. If no sign qualifier is present, the type is assumed to be signed. The D compiler also provides the following type aliases for you as a convenience:

TABLE 2–3 D Integer Type Aliases

Type Name	Description
`int8_t`	1 byte signed integer
`int16_t`	2 byte signed integer
`int32_t`	4 byte signed integer
`int64_t`	8 byte signed integer
`intptr_t`	Signed integer of size equal to a pointer
`uint8_t`	1 byte unsigned integer
`uint16_t`	2 byte unsigned integer
`uint32_t`	4 byte unsigned integer
`uint64_t`	8 byte unsigned integer
`uintptr_t`	Unsigned integer of size equal to a pointer

These type aliases are equivalent to using the name of the corresponding base type in the previous table and are appropriately defined for each data model. For example, the type name `uint8_t` is an alias for the type `unsigned char`. Later, we'll learn how you can define your own type aliases for use in your D programs.

D provides floating-point types for compatibility with ANSI-C declarations and types. Floating-point operators are not supported in D, but floating-point data objects can be traced and formatted using the `printf()` function. The following floating-point types may be used:

TABLE 2–4 D Floating-Point Data Types

Type Name	32–bit Size	64–bit Size
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`long double`	16 bytes	16 bytes

D also provides the special type `string` to represent ASCII strings. Strings are discussed in more detail in Chapter 6.

2.3 Constants

Integer constants can be written in decimal (`12345`), octal (`012345`), or hexadecimal (`0x12345`). Octal (base 8) constants must be prefixed with a leading zero. Hexadecimal (base 16) constants must be prefixed with either `0x` or `0X`. Integer constants are assigned the smallest type among `int`, `long`, and `long long` that can represent their value. If the value is negative, the signed version of the type is used. If the value is positive and too large to fit in the signed type representation, the unsigned type representation is used. You can apply one of the following suffixes to any integer constant to explicitly specify its D type:

`u` or `U`	`unsigned` version of the type selected by the compiler
`l` or `L`	`long`
`ul` or `UL`	`unsigned long`
`ll` or `LL`	`long long`
`ull` or `ULL`	`unsigned long long`

Floating-point constants are always written in decimal and must contain either a decimal point (`12.345`) or an exponent (`123e45`) or both (`123.34e-5`). Floating-point constants are assigned the type `double` by default. You can apply one of the following suffixes to any floating-point constant to explicitly specify its D type:

`f` or `F`	`float`
`l` or `L`	`long double`

Character constants are written as a single character or escape sequence enclosed in a pair of single quotes (`'a'`). Character constants are assigned the type `int` and are equivalent to an integer constant whose value is determined by that character's value in the ASCII character set. You can refer to `ascii`(5) for a list of characters and their values. You can also use any of the following special escape sequences in your character constants. D supports the same escape sequences found in ANSI-C.

TABLE 2–5 D Character Escape Sequences

`\a`	alert	`\\`	backslash
`\b`	backspace	`\?`	question mark
`\f`	formfeed	`\'`	single quote
`\n`	newline	`\"`	double quote
`\r`	carriage return	`\0`*oo*	octal value 0*oo*
`\t`	horizontal tab	`\x`*hh*	hexadecimal value 0x*hh*
`\v`	vertical tab	`\0`	null character

You can include more than one character specifier inside single quotes to create integers whose individual bytes are initialized according to the corresponding character specifiers. The bytes are read left-to-right from your character constant and assigned to the resulting integer in the order corresponding to the native endian-ness of your operating environment. Up to eight character specifiers can be included in a single character constant.

Strings constants of any length can be composed by enclosing them in a pair of double quotes (`"hello"`). A string constant may not contain a literal newline character; to create strings containing newlines use the `\n` escape sequence instead of a literal newline. String constants may contain any of the special character escape sequences shown for character constants above. Similar to ANSI-C, strings are represented as arrays of characters terminated by a null character (`\0`) which is implicitly added to each string constant that you declare. String constants are assigned the special D type `string`. The D compiler provides a set of special features for comparing and tracing character arrays that are declared as strings, as described in Chapter 6.

2.4 Arithmetic Operators

D provides the following binary arithmetic operators for use in your programs. These operators all have the same meaning for integers as they do in ANSI-C.

TABLE 2–6 D Binary Arithmetic Operators

`+`	integer addition
`-`	integer subtraction
`*`	integer multiplication
`/`	integer division
`%`	integer modulus

Arithmetic in D may only be performed on integer operands, or on pointers, as discussed later in Chapter 5. Arithmetic may not be performed on floating-point operands in D programs. The DTrace execution environment does not take any action on integer overflow or underflow; you must check for these conditions yourself in situations where they are applicable.

The DTrace execution environment does automatically check for and report division by zero errors resulting from improper use of the `/` and `%` operators. If a D program executes an invalid division operation, DTrace will automatically disable the affected instrumentation and report the error to you. Errors detected by DTrace will have no effect on other DTrace users or on the operating system kernel, so you don't need to worry about causing any damage if your D program inadvertently contains one of these errors.

In addition to these binary operators, the `+` and `-` operators may also be used as unary operators as well; these have higher precedence than any of the binary arithmetic operators. We will summarize the order of precedence and associativity properties for all the D operators at the end of this chapter. You can control precedence by grouping expressions in parentheses `( )`.

2.5 Relational Operators

D provides the following binary relational operators for use in your programs. These operators all have the same meaning as they do in ANSI-C.

TABLE 2–7 D Relational Operators

`<`	left-hand operand is less than right-operand
`<=`	left-hand operand is less than or equal to right-hand operand
`>`	left-hand operand is greater than right-hand operand

TABLE 2–7 D Relational Operators *(Continued)*

`>=`	left-hand operand is greater than or equal to right-hand operand
`==`	left-hand operand is equal to right-hand operand
`!=`	left-hand operand is not equal to right-hand operand

Relational operators are most frequently used to write D predicates. Each operator evaluates to a value of type `int` which is equal to one if the condition is true, or zero if it is false.

Relational operators may be applied to pairs of integers, pointers, or strings. If pointers are compared, the result is equivalent to an integer comparison of the two pointers interpreted as unsigned integers. If strings are compared, the result is determined as if by performing a `strcmp`(3C) on the two operands. Here are some example D string comparisons and their results:

```
"coffee" < "espresso"          ... returns 1 (true)
"coffee" == "coffee"           ... returns 1 (true)
"coffee" >= "mocha"            ... returns 0 (false)
```

Relational operators may also be used to compare a data object associated with an enumeration type with any of the enumerator tags defined by the enumeration. Enumerations are a facility for creating named integer constants and are described in more detail in Chapter 8.

2.6 Logical Operators

D provides the following binary logical operators for use in your programs. The first two are equivalent to the corresponding ANSI-C operators.

TABLE 2–8 D Logical Operators

`&&`	logical *AND*: true if both operands are true
`||`	logical *OR*: true if one or both operands are true
`^^`	logical *XOR*: true if exactly one operand is true

Logical operators are most frequently used in writing D predicates. The logical AND operator performs short-circuit evaluation: if the left-hand operand is false, the right-hand expression is not evaluated. The logical OR operator also performs short-circuit evaluation: if the left-hand operand is true, the right-hand expression is not evaluated. The logical XOR operator does not short-circuit: both expression operands are always evaluated.

In addition to the binary logical operators, the unary `!` operator may be used to perform a logical negation of a single operand: it converts a zero operand into a one, and a non-zero operand into a zero. By convention, D programmers use `!` when working with integers that are meant to represent boolean values, and `== 0` when working with non-boolean integers, although both expressions are equivalent in meaning.

The logical operators may be applied to operands of integer or pointer types. The logical operators interpret pointer operands as unsigned integer values. As with all logical and relational operators in D, operands are true if they have a non-zero integer value and false if they have a zero integer value.

2.7 Bitwise Operators

D provides the following binary operators for manipulating individual bits inside of integer operands. These operators all have the same meaning as they do in ANSI-C.

TABLE 2–9 D Bitwise Operators

`&`	bitwise AND
`|`	bitwise OR
`^`	bitwise XOR
`<<`	shift the left-hand operand left by the number of bits specified by the right-hand operand
`>>`	shift the left-hand operand right by the number of bits specified by the right-hand operand

The binary `&` operator is used to clear bits from an integer operand. The binary `|` operator is used to set bits in an integer operand. The binary `^` operator returns one in each bit position where exactly one of the corresponding operand bits is set.

The shift operators are used to move bits left or right in a given integer operand. Shifting left fills empty bit positions on the right-hand side of the result with zeroes. Shifting right using an unsigned integer operand fills empty bit positions on the left-hand side of the result with zeroes. Shifting right using a signed integer operand fills empty bit positions on the left-hand side with the value of the sign bit, also known as an *arithmetic shift* operation.

Shifting an integer value by a negative number of bits or by a number of bits larger than the number of bits in the left-hand operand itself produces an undefined result. The D compiler will produce an error message if it can detect this condition when you compile your D program.

In addition to the binary logical operators, the unary ~ operator may be used to perform a bitwise negation of a single operand: it converts each zero bit in the operand into a one bit, and each one bit in the operand into a zero bit.

2.8 Assignment Operators

D provides the following binary assignment operators for modifying D variables. Remember that you can only modify D variables and arrays: kernel data objects and constants may not be modified using the D assignment operators. The assignment operators have the same meaning as they do in ANSI-C.

TABLE 2–10 D Assignment Operators

`=`	set the left-hand operand equal to the right-hand expression value
`+=`	increment the left-hand operand by the right-hand expression value
`-=`	decrement the left-hand operand by the right-hand expression value
`*=`	mulitply the left-hand operand by the right-hand expression value
`/=`	divide the left-hand operand by the right-hand expression value
`%=`	modulo the left-hand operand by the right-hand expression value
`|=`	bitwise OR the left-hand operand with the right-hand expression value
`&=`	bitwise AND the left-hand operand with the right-hand expression value
`^=`	bitwise XOR the left-hand operand with the right-hand expression value
`<<=`	shift the left-hand operand left by the number of bits specified by the right-hand expression value

TABLE 2–10 D Assignment Operators *(Continued)*

`>>=`	shift the left-hand operand right by the number of bits specified by the right-hand expression value

Aside from the assignment operator `=`, the other assignment operators are provided as short-hand for using the `=` operator with one of the other operators described earlier. For example, the expression `x = x + 1` is equivalent to the expression `x += 1`, except that the expression `x` is evaluated once. These assignment operators obey the same rules for operand types as the binary forms described earlier.

The result of any assignment operator is an expression equal to the new value of the left-hand expression. You can use the assignment operators or any of the operators described so far in combination to form expressions of arbitrary complexity. You can use parentheses `( )` to group terms in complex expressions.

2.9 Increment and Decrement Operators

D provides the special unary `++` and `--` operators for incrementing and decrementing pointers and integers. These operators have the same meaning as they do in ANSI-C. These operators can only be applied to variables, and may be applied either before or after the variable name. If the operator appears before the variable name, the variable is first modified and then the resulting expression is equal to the new value of the variable. For example, the following two expressions produce identical results:

```
x += 1;
y = x;
```

```
y = ++x;
```

If the operator appears after the variable name, then the variable is modified after its current value is returned for use in the expression. For example, the following two expressions produce identical results:

```
y = x;
x -= 1;
```

```
y = x--;
```

You can use the increment and decrement operators to create new variables without declaring them. If a variable declaration is omitted and the increment or decrement operator is applied to a variable, the variable is implicitly declared to be of type `int64_t`.

The increment and decrement operators can be applied to integer or pointer variables. When applied to integer variables, the operators increment or decrement the corresponding value by one. When applied to pointer variables, the operators increment or decrement the pointer address by the size of the data type referenced by the pointer. We'll learn more about pointers and pointer arithmetic in D in Chapter 5.

2.10 Conditional Expressions

Although D does not provide support for if-then-else constructs, it does provide support for simple conditional expressions using the `?` and `:` operators. These operators permit a triplet of expressions to be associated where the first expression is used to conditionally evaluate one of the other two. For example, the following D statement could be used to set a variable `x` to one of two strings depending on the value of `i`:

```
x = i == 0 ? "zero" : "non-zero";
```

In this example, the expression `i == 0` is first evaluated to determine if it is true or false. If the first expression is true, the second expression is evaluated and the `?:` expression returns its value. If the first expression is false, the third expression is evaluated and the `?:` expression return its value.

As with any D operator, you can use multiple `?:` operators in a single expression to create more complex expressions. For example, the following expression would take a `char` variable `c` containing one of the characters 0-9, a-z, or A-Z and return the value of this character when interpreted as a digit in a hexadecimal (base 16) integer:

```
hexval = (c >= '0' && c <= '9') ? c - '0' :
    (c >= 'a' && c <= 'z') ? c + 10 - 'a' : c + 10 - 'A';
```

The first expression used with `?:` must be a pointer or integer in order to be evaluated for its truth value. The second and third expressions may be of any compatible types. You may not construct a conditional expression where, for example, one path returns a string and another an integer. The second and third expressions also may not invoke a tracing function such as `trace()` or `printf()`. If you wish to conditionally trace data, you should use a predicate instead, as we discussed earlier in Chapter 1.

2.11 Type Conversions

When expressions are constructed using operands of different but compatible types, type conversions are performed in order to determine the type of the resulting expression. The D rules for type conversions are the same as the arithmetic conversion rules for integers in ANSI-C. These rules are sometimes referred to as the *usual arithmetic conversions*.

A simple way to describe the conversion rules is as follows: each integer type is ranked in the order `char`, `short`, `int`, `long`, `long long`, with the corresponding unsigned types assigned a rank above its signed equivalent but below the next integer type. When you construct an expression using two integer operands such as `x + y` and the operands are of different integer types, the operand type with the highest rank is used as the result type.

If a conversion is required, the operand of lower rank is first *promoted* to the type of higher rank. Promotion does not actually change the value of the operand: it simply extends the value to a larger container according to its sign. If an unsigned operand is promoted, the unused high-order bits of the resulting integer are filled with zeroes. If a signed operand is promoted, the unused high-order bits are filled by performing sign extension. If a signed type is converted to an unsigned type, it is first sign-extended and then assigned the new unsigned type determined by the conversion.

Integers and other types can also be explicitly *cast* from one type to another. In D, pointers and integers can be cast to any integer or pointer types, but not to other types. Rules for casting and promoting strings and character arrays are discussed later in Chapter 6. An integer or pointer cast is formed using an expression such as:

```
y = (int)x;
```

where the destination type is enclosed in parentheses and used to prefix the source expression. Integers are cast to types of higher rank by performing promotion as described above. Integers are cast to types of lower rank by simply zeroing the excess high-order bits of the integer.

Since D does not permit floating-point arithmetic, no floating-point operand conversion or casting is permitted and no rules for implicit floating-point conversion are defined.

2.12 Precedence

The D rules for operator precedence and associativity are described in the following table. These rules are somewhat complex, but are necessary to provide precise compatibility with the ANSI-C operator precedence rules. The table entries are in order from highest precedence to lowest precedence.

TABLE 2–11 D Operator Precedence and Associativity

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * & (`*type*`) sizeof stringof offsetof xlate`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`^^`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

There are several operators in the table that we have not yet discussed; these will be covered in subsequent chapters:

`sizeof`	... computes the size of an object (Chapter 7)
`offsetof`	... computes the offset of a type member (Chapter 7)
`stringof`	... converts the operand to a string (Chapter 6)

`xlate`	... translates a data type (Chapter 33)
unary `&`	... computes the address of an object (Chapter 5)
unary `*`	... dereferences a pointer to an object (Chapter 5)
`->` and `.`	... accesses a member of a structure or union type (Chapter 7)

The comma (`,`) operator listed in the table is for compatibility with the ANSI-C comma operator, which can be used to evaluate a set of expressions in left-to-right order and return the value of the rightmost expression. This operator is provided strictly for compatibility with C and should generally not be used.

The `()` entry in the table above represents a function call; we saw examples of calls to functions such as `printf()` and `trace()` in Chapter 1. A comma is also used in D to list arguments to functions and to form lists of associative array keys. This comma is not the same as the comma operator and does *not* guarantee left-to-right evaluation. The D compiler provides no guarantee as to the order of evaluation of arguments to a function or keys to an associative array. You should be careful of using expressions with interacting side-effects, such as `i` and `i++`, in these contexts.

The `[]` entry in the table above represents an array or associative array reference. We saw examples of associative arrays earlier in Chapter 1. Later, we'll learn about a special kind of associative array called an *aggregation*; see Chapter 9. The `[]` operator can also be used to index into fixed-size C arrays as well; see Chapter 5.

CHAPTER 3

Variables

D provides two basic types of variables for use in your tracing programs: scalar variables and associative arrays. We briefly illustrated the use of these variables in our examples in Chapter 1. In this chapter, we will explore the rules for D variables in more detail and learn about how variables can be associated with different scopes. We will discuss a special kind of array variable, called an *aggregation*, later in Chapter 9.

3.1 Scalar Variables

Scalar variables are used to represent individual fixed-size data objects, such as integers and pointers. Scalar variables can also be used for fixed-size objects that are composed of one or more primitive or composite types. D provides the ability to create both arrays (see Chapter 5) of objects as well as composite structures (see Chapter 7). DTrace also represents strings as fixed-size scalars by permitting them to grow up to a predefined maximum length. Control over string length in your D program is discussed further in Chapter 6.

Scalar variables are created automatically the first time you assign a value to a previously undefined identifier in your D program, as we saw in Chapter 1. For example, to create a scalar variable named `x` of type `int`, you can simply assign it a value of type `int` in any probe clause:

```
BEGIN
{
    x = 123;
}
```

Scalar variables created in this manner are *global* variables: their name and data storage location is defined once and is visible in every clause of your D program. Any time you reference the identifier `x`, you are referring to a single storage location associated with this variable.

Unlike ANSI-C, D does not require explicit variable declarations. If you do want to declare a global variable to assign its name and type explicitly before using it, you can place a declaration outside of the probe clauses in your program as shown in the example below. Explicit variable declarations are not necessary in most D programs, but are sometimes useful when you want to carefully control your variable types or when you want to begin your program with a set of declarations and comments documenting your program's variables and their meanings.

```
int x; /* declare an integer x for later use */

BEGIN
{
    x = 123;
    ...
}
```

Unlike ANSI-C declarations, D variable declarations may not assign initial values; you must use a `BEGIN` probe clause to assign any initial values. All global variable storage is filled with zeroes for you by DTrace before you first reference it.

The D language definition places no limit on the size and number of D variables, but limits are defined by the DTrace implementation and by the memory available on your system and the D compiler will enforce any of the limitations that can be applied at the time you compile your program. You can learn more about how to tune options related to program limits in Chapter 16.

3.2 Associative Arrays

Associative arrays are used to represent collections of data elements that can be retrieved by specifying a name called a *key*. D associative array keys are formed by a list of scalar expression values called a *tuple*. You can think of the array tuple itself as an imaginary parameter list to a function that is called to retrieve the corresponding array value when you reference the array. Each D associative array has a fixed *key signature* consisting of a fixed number of tuple elements where each element has a given, fixed type. You can define different key signatures for each array in your D program.

Associative arrays differ from normal, fixed-size arrays in that they have no predefined limit on the number of elements, the elements can be indexed by any tuple as opposed to just using integers as keys, and the elements are not stored in preallocated consecutive storage locations. Associative arrays are useful in situations where you would use a hash table or other simple dictionary data structure in a C, C++, or Java™ language program. Associative arrays give you the ability to create a

dynamic history of events and state captured so far in your D program that you can use to create more complex control flows. We'll discuss a few more rules for programming with associative arrays, and then construct a D program that uses them later in this chapter.

To define an associative array, you write an assignment expression of the form:

name `[` *key* `]` `=` *expression* `;`

where *name* is any valid D identifier and *key* is a comma-separated list of one or more expressions. For example, the following statement defines an associative array `a` with key signature `[ int, string ]` and stores the integer value 456 in a location named by the tuple `[ 123, "hello" ]`:

```
a[123, "hello"] = 456;
```

The type of each object contained in the array is also fixed for all elements in a given array. Since `a` was first assigned using the integer 456, every subsequent value stored in the array will also be of type `int`. You can use any of the assignment operators defined in Chapter 2 to modify associative array elements, subject to the operand rules defined for each operator. The D compiler will produce an appropriate error message if you attempt an incompatible assignment. You can use any type with an associative array key or value that you can use with a scalar variable. You cannot nest an associative array within another associative array as a key or value.

You can reference an associative array using any tuple that is compatible with the array key signature. The rules for tuple compatibility are similar to those for function calls and variable assignments: the tuple must be of the same length and each type in the list of actual parameters must be compatible with the corresponding type in the formal key signature. For example, if an associative array `x` is defined as follows:

```
x[123ull] = 0;
```

then the key signature is of type `unsigned long long` and the values are of type `int`. This array can also be referenced using the expression `x['a']` because the tuple consisting of the character constant `'a'` of type `int` and length one is compatible with the key signature `unsigned long long` according to the arithmetic conversion rules described in Section 2.11.

If you need to explicitly declare a D associative array before using it, you can create a declaration of the array name and key signature outside of the probe clauses in your program source code:

```
int x[unsigned long long, char];

BEGIN
{
    x[123ull, 'a'] = 456;
}
```

Once an associative array is defined, references to any tuple of a compatible key signature are permitted, even if the tuple in question has not been previously assigned. Loading from such an unassigned associative array element is defined to return a zero-filled object. A consequence of this definition is that underlying storage is not allocated for an associative array element until a non-zero value is assigned to that element. Conversely, assigning an associative array element to zero causes DTrace to deallocate the underlying storage. This is important because the dynamic variable space out of which associative array elements are allocated is finite; if it is exhausted when an allocation is attempted, the allocation will fail and an error message will be generated indicating a dynamic variable drop. In general, one should always assign zero to associative array elements that are no longer in use. See Chapter 16 for other techniques to eliminate dynamic variable drops.

3.3 Thread-Local Variables

DTrace provides the ability to declare variable storage that is local to each operating system thread, as opposed to the global variables we have used so far. Thread-local variables are useful in situations where you wish to enable a probe and mark every thread that happens to fire the probe with some tag or other data. Creating a program to do this is easy in D because thread-local variables share a common name in your D code but refer to separate data storage associated with each thread. Thread-local variables are referenced by applying the `->` operator to the special identifier `self`:

```
syscall::read:entry
{
    self->read = 1;
}
```

This D fragment example enables the probe on the `read`(2) system call and associates a thread-local variable named `read` with each thread that fires the probe. Similar to global variables, thread-local variables spring into existence automatically on their first assignment and assume the type used on the right-hand side of the first assignment statement (in this example, `int`).

Each time the variable `self->read` is referenced in your D program, the data object referenced is the one associated with the operating system thread that was executing when the corresponding DTrace probe fired. You can think of a thread-local variable as an associative array that is implicitly indexed by a tuple that describes the thread's identity in the system. A thread's identity is unique over the lifetime of the system: if the thread exits and the same operating system data structure is used to create a new thread, this thread does *not* reuse the same DTrace thread-local storage identity.

Once you have defined a thread-local variable, you can reference it for any thread in the system even if the variable in question has not been previously assigned for that particular thread. If a thread's copy of the thread-local variable has not yet been

assigned, the data storage for the copy is defined to be filled with zeroes. As with associative array elements, underlying storage is not allocated for a thread-local variable until a non-zero value is assigned to it. And also as with associative array elements, assigning zero to a thread-local variable causes DTrace to deallocate the underlying storage. In general, one should always assign zero to thread-local variables that are no longer in use. See Chapter 16 for other techniques to fine-tune the dynamic variable space from which thread-local variables are allocated.

Thread-local variables of any type can be defined in your D program, including associative arrays. Here are some example thread-local variable definitions:

```
self->x = 123;              /* integer value */
self->s = "hello";              /* string value */
self->a[123, 'a'] = 456;    /* associative array */
```

Like any D variable, you don't need to explicitly declare thread-local variables before using them. If you want to create a declaration anyway, you can place one outside of your program clauses by prepending the keyword `self`:

```
self int x;    /* declare int x as a thread-local variable */

syscall::read:entry
{
    self->x = 123;
}
```

Thread-local variables are kept in a separate namespace from global variables so you can reuse names. Remember that `x` and `self->x` are not the same variable if you overload names in your program! Now that we've learned about thread-local variables let's use one in a real example. Go to your editor and type in the following program and save it in a file named `rtime.d`:

EXAMPLE 3–1 rtime.d: Compute Time Spent in `read`(2)

```
syscall::read:entry
{
    self->t = timestamp;
}

syscall::read:return
/self->t != 0/
{
    printf("%d/%d spent %d nsecs in read(2)\n",
        pid, tid, timestamp - self->t);

    /*
     * We're done with this thread-local variable; assign zero to it to allow
     * the DTrace runtime to reclaim the underlying storage.
     */
    self->t = 0;
}
```

Now go to your shell and start the program running. Wait a few seconds and you should start to see some output. If not, try running a few commands.

```
# dtrace -q -s rtime.d
100480/1 spent 11898 nsecs in read(2)
100441/1 spent 6742 nsecs in read(2)
100480/1 spent 4619 nsecs in read(2)
100452/1 spent 19560 nsecs in read(2)
100452/1 spent 3648 nsecs in read(2)
100441/1 spent 6645 nsecs in read(2)
100452/1 spent 5168 nsecs in read(2)
100452/1 spent 20329 nsecs in read(2)
100452/1 spent 3596 nsecs in read(2)
...
^C
#
```

In `rtime.d`, we use a thread-local variable named `t` to capture a timestamp on entry to `read`(2) by any thread. Then, in our return clause, we print out the amount of time spent in `read`(2) by subtracting `self->t` from the current timestamp. We use the built-in D variables `pid` and `tid` to report the process ID and thread ID of the thread performing the `read`(2). Because we are done using `self->t` once we report this information, we take care to assign it to 0 to allow DTrace to reuse the underlying storage associated with `t` for the curren thread.

Typically you will see many lines of output without even doing anything because, behind the scenes, server processes and daemons are executing `read`(2) all the time even when you aren't doing anything. What are they up to? Well, keep reading! Figuring out what your system is up to is what DTrace is all about. Try changing the second clause of `rtime.d` to use the `execname` variable to print out the name of the process performing a `read`(2) to learn more:

```
printf("%s/%d spent %d nsecs in read(2)\n",
    execname, tid, timestamp - self->t);
```

If you find a process that's of particular interest, add a predicate to learn more about its `read`(2) behavior:

```
syscall::read:entry
/execname == "Xsun"/
{
    self->t = timestamp;
}
```

3.4 Clause-Local Variables

You can also define D variables whose storage is reused for each D program clause. Clause-local variables are similar to automatic variables in a C, C++, or Java language program that are active during each invocation of a function. Like all D program variables, clause-local variables spring into existence on their first assignment. These variables can be referenced and assigned by applying the `->` operator to the special identifier `this`:

```
BEGIN
{
    this->secs = timestamp / 1000000000;
    ...
}
```

If you want to explicitly declare a clause-local variable before using it, you can do so using the `this` keyword:

```
this int x;   /* an integer clause-local variable */
this char c;  /* a character clause-local variable */

BEGIN
{
    this->x = 123;
    this->c = 'D';
}
```

Clause-local variables are only active for the lifetime of a given probe clause. After DTrace performs the actions associated with your clauses for a given probe, the storage for all clause-local variables is reclaimed and reused for the next clause. For this reason, clause-local variables are the only D variables that are not initially filled with zeroes. Note that if your program contains multiple clauses for a single probe, any clause-local variables will remain intact as the clauses are executed. For example:

EXAMPLE 3–2 clause.d: Clause-local variables

```
int me;            /* an integer global variable */
this int foo;        /* an integer clause-local variable */

tick-1sec
{
    /*
     * Set foo to be 10 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 10 : this->foo;
    printf("Clause 1 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
```

EXAMPLE 3–2 clause.d: Clause-local variables *(Continued)*

```
{
    /*
     * Set foo to be 20 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 20 : this->foo;
    printf("Clause 2 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
{
    /*
     * Set foo to be 30 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 30 : this->foo;
    printf("Clause 3 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
```

Because the clauses are *always* executed in program order, and because clause-local variables are persistent across different clauses enabling the same probe, running the above will always produce the same output:

```
# dtrace -q -s clause.d
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
^C
```

While clause-local variables are persistent across clauses enabling the same probe, their values are undefined in the first clause executed for a given probe. You should be sure to assign each clause-local variable an appropriate value before using it, or your program may have unexpected results.

Clause-local variables can be defined using any scalar variable type, but associative arrays may not be defined using clause-local scope. The scope of clause-local variables only applies to the corresponding variable data, not to the name and type identity defined for the variable. Once a clause-local variable is defined, this name and type signature may be used in any subsequent D program clause; the storage location may not be relied upon to be the same across different clauses.

You can use clause-local variables to accumulate intermediate results of calculations or as temporary copies of other variables. Access to a clause-local variable is much faster than access to an associative array, so if you need to reference an associative array value multiple times in the same D program clause it is more efficient to copy it into a clause-local variable first and then reference the local variable repeatedly.

3.5 Built-in Variables

We've already used a number of special built-in D variables in our example programs, including `timestamp`, `pid`, and several others. The complete list of D built-in variables is shown in the table below. All of these variables are scalar global variables; no thread-local or clause-local variables or built-in associative arrays are currently defined by D.

TABLE 3–1 DTrace Built-in Variables

Type and Name	Description
`int64_t arg0, ..., arg9`	The first ten input arguments to a probe represented as raw 64-bit integers. If fewer than ten arguments are passed to the current probe, the remaining variables return zero.
`args[]`	The typed arguments to the current probe, if any. The `args[]` array is accessed using an integer intex, but each element is defined to be the type corresponding to the given probe argument. For example, if `args[]` is referenced by a `read`(2) system call probe, `args[0]` is of type `int`, `args[1]` is of type `void *`, and `args[2]` is of type `size_t`.
`uintptr_t caller`	The program counter location of the current thread just before entering the current probe.
`lwpsinfo_t *curlwpsinfo`	The lightweight process (LWP) state of the LWP associated with the current thread. This structure is described in further detail in `proc`(4).
`psinfo_t *curpsinfo`	The process state of the process associated with the current thread. This structure is described in further detail in `proc`(4).

TABLE 3–1 DTrace Built-in Variables *(Continued)*

Type and Name	Description
`kthread_t *curthread`	The address of the operating system kernel's internal data structure for the current thread, the `kthread_t`. The `kthread_t` is defined in `<sys/thread.h>`. Refer to *Solaris Internals* for more information on this and other operating system data structures.
`epid`	The enabled probe ID (EPID) for the current probe. This integer uniquely identifiers a particular probe that is enabled with a specific predicate and set of actions.
`int errno`	The error value returned by the last system call executed by this thread.
`string execname`	The name that was passed to `exec`(2) to execute the current process.
`uint_t id`	The probe ID for the current probe. This is the system-wide unique identifier for the probe as published by DTrace and listed in the output of `dtrace -l`.
`uint_t ipl`	The interrupt priority level (IPL) on the current CPU at probe firing time. Refer to *Solaris Internals* for more information on interrupt levels and interrupt handling in the Solaris operating system kernel.
`pid_t pid`	The process ID of the current process.
`string probefunc`	The function name portion of the current probe's description.
`string probemod`	The module name portion of the current probe's description.
`string probename`	The name portion of the current probe's description.
`string probeprov`	The provider name portion of the current probe's description.
`uint64_t regs[]`	The current thread's kernel-mode register values at probe firing time.
`uint_t stackdepth`	The current thread's stack frame depth at probe firing time.

TABLE 3–1 DTrace Built-in Variables *(Continued)*

Type and Name	Description
`id_t tid`	The thread ID of the current thread. For threads associated with user processes, this value is equal to the result of a call to `thr_self`(3THREAD).
`uint64_t timestamp`	The current value of a nanosecond timestamp counter. This counter increments from an arbitrary point in the past and should only be used for relative computations.
`uint64_t uregs[]`	The current thread's saved user-mode register values at probe firing time. Use of the `uregs[]` array is discussed in Chapter 27.
`uint64_t vtimestamp`	The current value of a nanosecond timestamp counter that is virtualized to the amount of time that the current thread has been running on a CPU, minus the time spent in DTrace predicates and actions. This counter increments from an arbitrary point in the past and should only be used for relative time computations.

Functions built into the D language such as `trace()` are discussed in Chapter 10.

3.6 External Variables

D uses the backquote character (`) as a special scoping operator for accessing variables that are defined in the operating system and not in your D program. For example, the Solaris kernel contains a C declaration of a system tunable named `kmem_flags` for enabling memory allocator debugging features (see the Tunable Parameters Guide for more information about `kmem_flags`). This tunable is declared as a C variable in the kernel source code as follows:

```
int kmem_flags;
```

To access the value of this variable in a D program, use the D notation:

```
`kmem_flags
```

DTrace associates each kernel symbol with the type used for it in the corresponding operating system C code, providing you easy source-based access to the native operating system data structures. In order to use external operating system variables, you will need access to the corresponding operating system source code.

When you access external variables from a D program, you are accessing the internal implementation details of another program such as the operating system kernel or its device drivers. These implementation details do not form a stable interface upon which you can rely! Any D programs you write that consume these details may cease to work when you next upgrade the corresponding piece of software. For this reason, external variables are typically used by kernel and device driver developers and service personnel in order to debug performance or functionality problems using DTrace. To learn more about the stability of your D programs, refer to Chapter 32.

Kernel symbol names are kept in a separate namespace from D variable and function identifiers, so you never need to worry about these names conflicting with your D variables. When you prefix a variable with a backquote, the D compiler searches the known kernel symbols in order using the list of loaded modules in order to find a matching variable definition. Since the Solaris kernel supports dynamically loaded modules with separate symbol namespaces, it is possible that the same variable name is used more than once in the active operating system kernel. You can resolve these name conflicts by specifying the name of the kernel module whose variable should be accessed prior to the backquote in the symbol name. For example, each loadable kernel module typically provides a `_fini`(9E) function, so to refer to the address of the `_fini` function provided by a kernel module named `foo` you would write:

```
foo`_fini
```

You can apply any of the D operators to external variables, except those that modify values, subject to the usual rules for operand types. When you launch DTrace, the D compiler loads the set of variable names corresponding to the active kernel modules, so declarations of these variables are not required. You may not apply any operator to an external variable that modifies its value, such as = or +=. For safety reasons, DTrace prevents you from damaging or corrupting the state of the software you are observing.

CHAPTER 4

Program Structure

D programs consist of a set of clauses that describe probes to enable and predicates and actions to bind to these probes. D programs can also contain declarations of variables, as we saw in the previous chapter, and definitions of new types, described later in Chapter 8. In this chapter, we will formally describe the overall structure of a D program and features for constructing probe descriptions that match more than one probe. We'll also discuss the use of the C preprocessor, `cpp`, with D programs.

4.1 Probe Clauses and Declarations

As shown in our examples so far, a D program source file consists of one or more probe clauses that describe the instrumentation to be enabled by DTrace. Each probe clause has the general form:

```
probe descriptions
/ predicate /
{
   action statements
}
```

The predicate and list of action statements may each be optionally omitted. Any directives found outside probe clauses are referred to as *declarations*. Declarations may only be used outside of probe clauses; no declarations inside of the enclosing { } are permitted and declarations may not be interspersed between the elements of the probe clause shown above. Whitespace can be used to separate any D program elements and to indent action statements.

Declarations can be used to declare D variables and external C symbols as we discussed in Chapter 3, or to define new types for use in D (see Chapter 8). Special D compiler directives called *pragmas* may also appear anywhere in a D program, including outside of probe clauses. D pragmas are specified on lines beginning with a # character. D pragmas are used, for example, to set run-time DTrace options; see Chapter 16 for details.

4.2 Probe Descriptions

Every D program clause begins with a list of one or more probe descriptions, each taking the usual form:

provider:*module*:*function*:*name*

If one or more fields of the probe description are omitted, the specified fields are interpreted from right to left by the D compiler. For example, the probe description `foo:bar` would match a probe with function `foo` and name `bar` regardless of the value of the probe's provider and module fields. Therefore, a probe description is really more accurately viewed as a *pattern* that can be used to match one or more probes based on their names.

In general you should write your D probe descriptions specifying all four field delimiters so that you can specify the desired *provider* on the left-hand side. If you don't specify the provider, you may obtain unexpected results if multiple providers publish probes with the same name. Similarly, future versions of DTrace may include new providers whose probes unintentionally match your partially-specified probe descriptions. You can specify a provider but match any of its probes by leaving any of the module, function, and name fields blank. For example, the description `syscall:::` can be used to match every probe published by the DTrace `syscall` provider.

Probe descriptions also support a pattern matching syntax similar to the shell *globbing* pattern matching syntax described in `sh`(1). Before matching a probe to a description, DTrace scans each description field for the characters `*`, `?`, and `[`. If one of these characters appears in a probe description field and is not preceded by a `\`, the field is regarded as a pattern. The description pattern must match the entire corresponding field of a given probe. The complete probe description must match on every field in order to successfully match and enable a probe. A probe description field that is not a pattern must exactly match the corresponding field of the probe. A description field that is empty matches any probe.

The following special characters are recognized in probe name patterns:

TABLE 4–1 Probe Name Pattern Matching Characters

`*`	Matches any string, including the null string.
`?`	Matches any single character.
`[ ... ]`	Matches any one of the enclosed characters. A pair of characters separated by `-` matches any character between the pair, inclusive. If the first character after the `[` is `!`, any character not enclosed in the set is matched.
`\`	Interpret the next character as itself, without any special meaning.

Pattern match characters can be used in any or all of the four fields of your probe descriptions. You can also use patterns to list matching probes by using them on the command line with `dtrace -l`. For example, the command `dtrace -l -f kmem_*` would list all DTrace probes in functions whose names begin with the prefix `kmem_`.

If you want to specify the same predicate and actions for more than one probe description or description pattern, you can place the descriptions in a comma-separated list. For example, the following D program would trace a timestamp each time probes associated with entry to system calls containing the words "`lwp`" or "`sock`" fire:

```
syscall::*lwp*:entry, syscall::*sock*:entry
{
    trace(timestamp);
}
```

A probe description may also specify a probe using its integer probe ID. For example, the clause:

```
12345
{
    trace(timestamp);
}
```

could be used to enable probe ID 12345, as reported by `dtrace -l -i 12345`. You should always write your D programs using human-readable probe descriptions, as integer probe IDs are not guaranteed to remain consistent as DTrace provider kernel modules are loaded and unloaded or following a reboot.

4.3 Predicates

Predicates are expressions enclosed in slashes / / that are evaluated at probe firing time to determine whether the associated actions should be executed or not. As we've seen in our examples, predicates are the primary conditional construct used for building more complex control flow in a D program. You can omit the predicate section of the probe clause entirely for any probe, in which case the actions are always executed when the probe fires.

Predicate expressions can use any of the previously described D operators and may refer to any D data objects such as variables and constants. The predicate expression must evaluate to a value of integer or pointer type so that it can be considered as true or false. As with all D expressions, a zero value is interpreted as false and any non-zero value is interpreted as true.

4.4 Actions

Probe actions are described by a list of statements separated by semicolons (`;`) and enclosed in braces { }. If you only wish to note that a particular probe fired on a particular CPU without tracing any data or performing any additional actions, you can specify an empty set of braces with no statements inside.

4.5 Use of the C Preprocessor

The C programming language used for defining Solaris system interfaces includes a *preprocessor* that performs a set of initial steps in C program compilation. The C preprocessor is commonly used to define macro substitutions where one token in a C program is replaced with another predefined set of tokens, or to include copies of system header files. You can use the C preprocessor in conjunction with your D programs by specifying the `dtrace -C` option. This option causes `dtrace` to first execute the `cpp`(1) preprocessor on your program source file and then pass the results to the D compiler. The C preprocessor is described in more detail in *The C Programming Language*.

The D compiler automatically loads the set of C type descriptions associated with the operating system implementation, but you can use the preprocessor to include other type definitions such as types used in your own C programs. You can also use the preprocessor to perform other tricks, like creating macros that expand to chunks of D

code and other program elements. If you use the preprocessor with your D program, you may only include files that contain valid D declarations. Typical C header files include only external declarations of types and symbols, which will be correctly interpreted by the D compiler. C header files that include additional program elements like C function source code will not be able to be parsed by the D compiler and will produce an appropriate error message.

CHAPTER 5

Pointers and Arrays

Pointers are memory addresses of data objects in the operating system kernel or in the address space of a user process. D provides the ability to create and manipulate pointers and store them in variables and associative arrays. In this chapter, we explore the D syntax for pointers, operators that can be applied to create or access pointers, and the relationship between pointers and fixed-size scalar arrays. We also discuss issues relating to the use of pointers in different address spaces. If you are an experienced C or C++ programmer, you can skim most of this chapter as the D pointer syntax is the same as the corresponding ANSI-C syntax, but you should read sections 5.1, 5.2, 5.7, and 5.8 as they describe features and issues specific to DTrace.

5.1 Pointers and Addresses

The Solaris operating system uses a technique called *virtual memory* to provide each user process with its own virtual view of the memory resources on your system. A virtual view on memory resources is referred to as an *address space*, which associates a range of address values (either `[0 ... 0xffffffff]` for a 32-bit address space or `[0 ... 0xffffffffffffffff]` for a 64-bit address space) with a set of translations that the operating system and hardware use to convert each virtual address to a corresponding physical memory location. Pointers in D are data objects that store an integer virtual address value and associate it with a D type that describes the format of the data stored at the corresponding memory location.

You can declare a D variable to be of pointer type by first specifying the type of the referenced data and then appending a `*` to the type name to indicate you want to declare a pointer type. For example, the declaration:

```
int *p;
```

declares a D global variable named `p` that is a pointer to an integer. This means that `p` itself is an integer of size 32 or 64-bits whose value is the address of another integer located somewhere in memory. Since the compiled form of your D code is executed at probe firing time inside the operating system kernel itself, D pointers are typically pointers associated with the kernel's address space. You can use the `isainfo`(1) `-b` command to determine the number of bits used for pointers by the active operating system kernel.

If you want to create a pointer to a data object inside of the kernel, you can compute its address using the `&` operator. For example, the operating system kernel source code declares an `int kmem_flags` tunable that we discussed earlier. You could trace the address of this `int` by tracing the result of applying the `&` operator to the name of that object in D:

```
trace(&`kmem_flags);
```

The `*` operator can be used to refer to the object addressed by the pointer, and acts as the inverse of the `&` operator. For example, the following two D code fragments are equivalent in meaning:

```
p = &`kmem_flags;
trace(*p);
```

```
trace(`kmem_flags);
```

The left-hand fragment creates a D global variable pointer `p`. Since the `kmem_flags` object is of type `int`, the type of the result of `&`kmem_flags` is `int *` (that is, pointer to `int`). The left-hand fragment traces the value of `*p`, which follows the pointer back to the data object `kmem_flags`. This fragment is therefore the same as the right-hand fragment which simply traces the value of the data object directly using its name.

5.2 Pointer Safety

If you are a C or C++ programmer, you may be a bit frightened after reading the previous section because you know that misuse of pointers in your programs can cause your programs to crash. Don't worry! DTrace is designed to be a robust, safe environment for executing your D programs. You may indeed write a buggy D program, but invalid D pointer accesses will not cause DTrace or the operating system kernel to fail or crash in any way. Instead, the DTrace software will detect any invalid pointer accesses, disable your instrumentation, and report the problem back to you for debugging.

If you have programmed in the Java™ programming language, you may know that the Java language does not support pointers at all for precisely the same reasons of safety. Pointers are needed in D because they are an intrinsic part of the operating system's implementation in C, but DTrace implements the same kind of safety

mechanisms found in the Java programming language that prevent buggy programs from damaging themselves or each other. DTrace's error reporting is similar to how the run-time environment for the Java programming language can detect a programming error and report an exception back to you.

We can demonstrate DTrace's error handling and reporting by writing a deliberately bad D program using pointers. Go to your editor and type in the following D program and save it in a file named `badptr.d`:

EXAMPLE 5–1 badptr.d: Demonstrate DTrace Error Handling

```
BEGIN
{
    x = (int *)NULL;
    y = *x;
    trace(y);
}
```

The `badptr.d` program creates a D pointer named `x` that is a pointer to `int`. We assign this pointer the special invalid pointer value `NULL` which is a built-in alias for address 0. By convention, address 0 is always defined to be invalid so that `NULL` can be used as a sentinel value in C and D programs. We use a cast expression to convert `NULL` to be a pointer to an integer. We then dereference our pointer using the expression `*x`, and assign the result to another variable `y`, and then attempt to trace `y`. When we execute our D program, DTrace detects an invalid pointer access when the statement `y = *x` is executed and reports the error to us:

```
# dtrace -s badptr.d
dtrace: script '/dev/stdin' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #2 at DIF offset 4
dtrace: 1 error on CPU 0
^C
#
```

The other problem that can arise from programs that use invalid pointers is an *alignment error*. By architectural convention, fundamental data objects such as integers are aligned in memory according to their size. For example, 2-byte integers are aligned on addresses that are multiples of 2, 4-byte integers on multiples of 4, and so on. If you dereference a pointer to a 4-byte integer and your pointer address is an invalid value that is not a multiple of 4, your access will fail with an alignment error. Alignment errors in D almost always indicate that your pointer has an invalid or corrupt value due to a bug in your D program. You can create an example alignment error by changing the source code of `badptr.d` to use the address `(int *)2` instead of `NULL`. Since `int` is 4 bytes and 2 is not a multiple of 4, the expression `*x` will now result in a DTrace alignment error.

For details about the DTrace error mechanism, see "17.3 The `ERROR` Probe" on page 183.

5.3 Array Declarations and Storage

D provides support for *scalar arrays* in addition to the dynamic associative arrays we learned about in Chapter 3. Scalar arrays are a fixed-length group of consecutive memory locations that each store a value of the same type. Scalar arrays are accessed by referring to each location with an integer starting from zero. Scalar arrays correspond directly in concept and syntax with arrays in C and C++. Scalar arrays are not used as frequently in D as associative arrays and their more advanced counterparts *aggregations* (see Chapter 9), but they are sometimes needed when accessing existing operating system array data structures declared in C.

A D scalar array of 5 integers would be declared by using the type `int` and suffixing the declaration with the number of elements in square brackets as follows:

```
int a[5];
```

The following diagram shows a visual representation of the array storage:

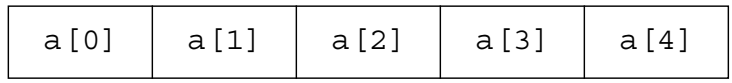

FIGURE 5–1 Scalar Array Representation

The D expression `a[0]` is used to refer to the first array element, `a[1]` refers to the second, and so on. From a syntactic perspective, scalar arrays and associative arrays are very similar. You can declare an associative array of five integers referenced by an integer key as follows:

```
int a[int];
```

and also reference this array using the expression `a[0]`. But from a storage and implementation perspective, the two arrays are very different. The static array `a` consists of five consecutive memory locations numbered from zero and the index refers to an offset in the storage allocated for the array. An associative array, on the other hand, has no predefined size and does not store elements in consecutive memory locations. In addition, associative array keys have no relationship to the corresponding's value storage location: you can access associative array elements `a[0]` and `a[-5]` and only two words of storage will be allocated by DTrace which may or may not be consecutive. Associative array keys are abstract names for the corresponding value that have no relationship to the value storage locations.

If you create an array using an initial assignment and use a single integer expression as the array index (for example, `a[0] = 2`), the D compiler will always create a new associative array, even though in this expression `a` could also be interpreted as an assignment to a scalar array. Scalar arrays must be predeclared in this situation so that the D compiler can see the definition of the array size and infer that the array is a scalar array.

5.4 Pointer and Array Relationship

Pointers and arrays have a special relationship in D, just as they do in ANSI-C. An array is represented by a variable that is associated with the address of its first storage location. A pointer is also the address of a storage location with a defined type, so D permits the use of the array [ ] index notation with both pointer variables and array variables. For example, the following two D fragments are equivalent in meaning:

```
p = &a[0];                  trace(a[2]);
trace(p[2]);
```

In the left-hand fragment, the pointer `p` is assigned to the address of the first array element in `a` by applying the `&` operator to the expression `a[0]`. We then trace the value of the third array element (index 2) using the expression `p[2]`. Since `p` now contains the same address associated with `a`, this expression yields the same value as `a[2]`, shown in the right-hand fragment. One consequence of this equivalence is that C and D permit you to access any index of any pointer or array; array bounds checking is not performed for you by the compiler or DTrace runtime environment. If you access memory beyond the end of an array's predefined value, you will either get an unexpected result or DTrace will report an invalid address error, as shown above. As always, you can't damage DTrace itself or your operating system, but you will need to debug your D program.

The difference between pointers and arrays is that a pointer variable refers to a separate piece of storage that contains the integer address of some other storage. An array variable names the array storage itself, not the location of an integer that in turns contains the location of the array. This difference is illustrated in the diagram below:

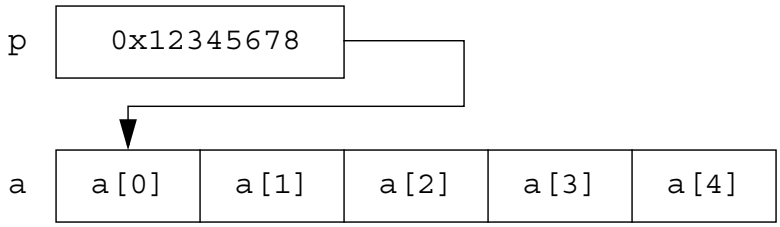

FIGURE 5–2 Pointer and Array Storage

This difference is manifested in the D syntax if you attempt to assign pointers and scalar arrays. If `x` and `y` are pointer variables, the expression `x = y` is legal; it simply copies the pointer address in `y` to the storage location named by `x`. If `x` and `y` are scalar array variables, the expression `x = y` is not legal. Arrays may not be assigned as a whole in D. However, an array variable or symbol name can be used in any context where a pointer is permitted. If `p` is a pointer and `a` is an array, the statement `p = a` is permitted; this statement is equivalent to the statement `p = &a[0]`.

5.5 Pointer Arithmetic

Since pointers are just integers used as addresses of other objects in memory, D provides a set of features for performing arithmetic on pointers. However, pointer arithmetic is not identical to integer arithmetic. Pointer arithmetic implicitly adjusts the underlying address by multiplying or dividing the operands by the size of the type referenced by the pointer. The following D fragment illustrates this property:

```
int *x;

BEGIN
{
    trace(x);
    trace(x + 1);
    trace(x + 2);
}
```

We create an integer pointer `x` and then trace its value, its value incremented by one, and its value incremented by two. If you type in and execute this program, DTrace will report the integer values 0, 4, and 8.

Since `x` is a pointer to an int (size 4 bytes), incrementing `x` adds 4 to the underlying pointer value. This property is useful when using pointers to refer to consecutive storage locations such as arrays. For example, if `x` were assigned to the address of an array `a` like the one shown in our diagram above, the expression `x + 1` would be

equivalent to the expression `&a[1]`. Similarly, the expression `*(x + 1)` would refer to the value `a[1]`. Pointer arithmetic is implemented by the D compiler whenever a pointer value is incremented using the `+=`, `+`, or `++` operators.

Pointer arithmetic is also applied when an integer is subtracted from a pointer on the left-hand side, when a pointer is subtracted from another pointer, or when the `--` operator is applied to a pointer. For example, the following D program would trace the result 2:

```
int *x, *y;
int a[5];

BEGIN
{
    x = &a[0];
    y = &a[2];
    trace(y - x);
}
```

5.6 Generic Pointers

Sometimes it is useful to represent or manipulate a generic pointer address in a D program without specifying the type of data referred to by the pointer. Generic pointers can be specified using the type `void *`, where the keyword `void` represents the absence of specific type information, or using the built-in type alias `uintptr_t` which is aliased to an unsigned integer type of size appropriate for a pointer in the current data model. You may not apply pointer arithmetic to an object of type `void *`, and these pointers cannot be dereferenced without casting them to another type first. You can cast a pointer to the `uintptr_t` type when you need to perform integer arithmetic on the pointer value.

Pointers to `void` may be used in any context where a pointer to another data type is required, such as an associative array tuple expression or the right-hand side of an assignment statement. Similarly, a pointer to any data type may be used in a context where a pointer to `void` is required. To use a pointer to a non-`void` type in place of another non-`void` pointer type, an explicit cast is required. You must always use explicit casts to convert pointers to integer types such as `uintptr_t`, or to convert these integers back to the appropriate pointer type.

5.7 Multi-Dimensional Arrays

Multi-dimensional scalar arrays are used infrequently in D, but are provided for compatibility with ANSI-C and for observing and accessing operating system data structures created using this capability in C. A multi-dimensional array is declared as a consecutive series of scalar array sizes enclosed in square brackets `[ ]` following the base type. For example, to declare a fixed-size two-dimensional rectangular array of integers of dimensions 12 rows by 34 columns, you would write the declaration:

```
int a[12][34];
```

A multi-dimensional scalar array is accessed using similar notation. For example, to access the value stored at row 0 column 1 you would write the D expression:

```
a[0][1]
```

Storage locations for multi-dimensional scalar array values are computed by multiplying the row number by the total number of columns declared, and then adding the column number.

You should be careful not to confuse the multi-dimensional array syntax with the D syntax for associative array accesses (that is, `a[0][1]` is not the same as `a[0, 1]`). If you use an incompatible tuple with an associative array or attempt an associative array access of a scalar array, the D compiler will report an appropriate error message and refuse to compile your program.

5.8 Pointers to DTrace Objects

In general, the D compiler prohibits you from using the `&` operator to obtain pointers to DTrace objects such as associative arrays, built-in functions, and variables. You are prohibited from obtaining the address of these variables so that the DTrace runtime environment is free to relocate them as needed between probe firings in order to more efficiently manage the memory required for your programs. If you create composite structures, it is possible to construct expressions that do retrieve the kernel address of your DTrace object storage. You should avoid creating such expressions in your D programs. If you need to use such an expression, be sure not to cache the address across probe firings.

In ANSI-C, pointers can also be used to perform indirect function calls or to perform assignments, such as placing an expression using the unary `*` dereference operator on the left-hand side of an assignment operator. In D, these types of expressions using

pointers are not permitted. You may only assign values directly to D variables using their name or by applying the array index operator `[]` to a D scalar or associative array. You may only call functions defined by the DTrace environment by name as specified in Chapter 10. Indirect function calls using pointers are not permitted in D.

5.9 Pointers and Address Spaces

As we discussed briefly at the beginning of this chapter, a pointer is an address that provides a translation within some *virtual address space* to a piece of physical memory. DTrace executes your D programs within the address space of the operating system kernel itself. Your entire Solaris system manages many address spaces: one for the operating system kernel, and one for each user process. Since each address space provides the illusion that it can access all of the memory on the system, the same virtual address pointer value can be reused across address spaces but translate to different physical memory. Therefore, when writing D programs that use pointers, you must be aware of the address space corresponding to the pointers you intend to use.

For example, if you use the `syscall` provider to instrument entry to a system call that takes a pointer to an integer or array of integers as an argument (for example, `pipe`(2)), it would not be valid to dereference that pointer or array using the `*` or `[]` operators as described in this chapter because the address in question is an address in the address space of the user process that performed the system call. Applying the `*` or `[]` operators to this address in D would result in a kernel address space access, which would result in an invalid address error or in returning unexpected data to your D program depending upon whether the address happened to match a valid kernel address.

To access user process memory from a DTrace probe, you must apply one of the `copyin()`, `copyinstr()`, or `copyinto()` functions described in Chapter 10 to the user address space pointer. You should take care when writing your D programs to name and comment variables storing user addresses appropriately to avoid confusion. You can also store user addresses as `uintptr_t` so you don't accidentally compile D code that dereferences them. We'll learn more about techniques for using DTrace on user processes in Chapter 27.

CHAPTER 6

Strings

As we've seen in previous chapters, DTrace provides support for tracing and manipulating strings. In this chapter, we describe the complete set of D language features for declaring and manipulating strings. Unlike ANSI-C, strings in D are first-class citizens with their own built-in type and operator support to permit you to easily and unambiguously use them in your tracing programs.

6.1 String Representation

Strings are represented in DTrace as an array of characters terminated by a null byte (that is, a byte whose value is zero, usually written as `'\0'`). The visible part of the string is of variable length, depending on the location of the null byte, but DTrace stores each string in a fixed-size array so that each probe traces a consistent amount of data. Strings may not exceed the length of this predefined string limit, but the limit can be modified in your D program or on the `dtrace` command line by tuning the `strsize` option. Refer to Chapter 16 for more information on tunable DTrace options. The default string limit is 256 bytes.

The D language provides an explicit `string` type rather than using the type `char *` to refer to strings. The `string` type is equivalent to a `char *` in that it is the address of a sequence of characters, but the D compiler and D functions like `trace()` provide enhanced capabilities when applied to expressions of type `string`. For example, the string type removes the ambiguity of the type `char *` when you need to trace the actual bytes of a string. If you write the D statement:

```
trace(s);
```

and `s` is of type `char *`, DTrace will trace the value of the pointer `s` (that is, it will trace an integer address value). If you write the D statement:

```
trace(*s);
```

then by definition of the * operator, the D compiler will dereference the pointer s and trace the single character at that location. These behaviors are consistent with the language definitions we have presented so far, and are essential to permitting you to manipulate character pointers that by design refer to either single characters, or to arrays of byte-sized integers that are not strings and do not end with a null byte. The string type makes it clear to the D compiler that when you write:

```
trace(s);
```

and s is of type string, you want DTrace to trace a null terminated string of characters whose address is stored in the variable s. You can also perform lexical comparison of expressions of type string, as we'll see shortly.

6.2 String Constants

String constants are enclosed in double quotes (") and are automatically assigned the type string by the D compiler. You can define string constants of any length, limited only by the amount of memory DTrace is permitted to consume on your system. The terminating null byte (\0) is added automatically by the D compiler to any string constants that you declare. The size of a string constant object is the number of bytes associated with the string plus one additional byte for the terminating null byte.

A string constant may not contain a literal newline character; to create strings containing newlines use the \n escape sequence instead of a literal newline. String constants may also contain any of the special character escape sequences defined for character constants in Chapter 2, Section 2.3.

6.3 String Assignment

Unlike assignment of char * variables, strings are copied by value, not by reference. String assignment is performed using the = operator and copies the actual bytes of the string from the source operand up to and including the null byte to the variable on the left-hand side, which must be of type string. As usual, you can create a new variable of type string by simply assigning it an expression of type string. For example, the D statement:

```
s = "hello";
```

would create a new variable `s` of type `string` and copy the 6 bytes of the string `"hello"` into it (5 printable characters plus the null byte). String assignment is analogous to the C library function `strcpy`(3C), except that if the source string exceeds the limit of the storage of the destination string, the resulting string is automatically truncated at this limit.

You can also assign a string variable an expression of a type that is compatible with strings. In this case, the D compiler automatically promotes the source expression to the string type and performs a string assignment. The D compiler permits any expression of type `char *` or of type `char[`*n*`]` (that is, a scalar array of `char` of any size), to be promoted to a `string`.

6.4 String Conversion

Expressions of other types may be explicitly converted to type `string` by using a cast expression or by applying the special `stringof` operator, which are equivalent in meaning:

`s = (string)` *expression* `s = stringof (` *expression* `)`

The `stringof` operator binds very tightly to the operand on its right-hand side, but typically parentheses are used to surround the expression for clarity, although they are not strictly necessary.

Any expression that is a scalar type such as a pointer or integer or a scalar array address may be converted to string. Expressions of other types such as `void` may not be converted to `string`. If you erroneously convert an invalid address to a string, the usual DTrace safety features will prevent you from damaging the system or DTrace, but you may end up tracing a sequence of undecipherable characters.

6.5 String Comparison

D overloads the binary relational operators and permits them to be used for string comparisons as well as integer comparisons. The relational operators perform string comparison whenever both operands are of type `string`, or when one operand is of type `string` and the other operand can be promoted to type `string`, as described under String Assignment above. All of the relational operators can be used to compare strings:

TABLE 6–1 D Relational Operators for Strings

`<`	left-hand operand is less than right-operand
`<=`	left-hand operand is less than or equal to right-hand operand
`>`	left-hand operand is greater than right-hand operand
`>=`	left-hand operand is greater than or equal to right-hand operand
`==`	left-hand operand is equal to right-hand operand
`!=`	left-hand operand is not equal to right-hand operand

As with integers, each operator evaluates to a value of type `int` which is equal to one if the condition is true, or zero if it is false.

The relational operators compare the two input strings byte-by-byte, similar to the C library routine `strcmp`(3C). Each byte is compared using its corresponding integer value in the ASCII character set, as shown in `ascii`(5), until a null byte is read or the maximum string length is reached. Here are some example D string comparisons and their results:

```
"coffee" < "espresso"          ... returns 1 (true)
"coffee" == "coffee"           ... returns 1 (true)
"coffee" >= "mocha"            ... returns 0 (false)
```

CHAPTER 7

Structs and Unions

Collections of related variables can be grouped together into composite data objects called *structs* and *unions*. D permits you to define these objects by creating new type definitions for them. You can use your new types for any D variables, including associative array values. In this chapter we explore the syntax and semantics for creating and manipulating these composite types and learn about the D operators that interact with them. After introducing the syntax for structs and unions, we illustrate their use by constructing several new example programs that also demonstrate the use of the DTrace `fbt` and `fasttrap` providers.

7.1 Structs

The D keyword `struct`, short for *structure*, is used to introduce a new type composed of a group of other types. The new struct type can be used as the type for D variables and arrays, in effect permitting you to define groups of related variables under a single name. D structs are the same as the corresponding construct in C and C++. If you have programmed in the Java™ programming language, think of a struct as a class in the Java programming language, but one with data members only and no methods.

Let's suppose we want to create a more sophisticated system call tracing program in D that records a number of things about each `read`(2) and `write`(2) system call executed by our shell, such as the elapsed time, number of calls, and the largest byte count passed as an argument. We could write a D clause like this to record these properties in three separate associative arrays:

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
    calls[probefunc]++;
```

```
    maxbytes[probefunc] = arg2 > maxbytes[probefunc] ?
        arg2 : maxbytes[probefunc];
}
```

but this is inefficient because DTrace must create three separate associative arrays and store separate copies of the identical tuple values corresponding to probefunc for each one. Instead, we can conserve space and make our program easier to read and maintain by using a struct. First, we'll declare a new struct type outside of probe clauses:

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};
```

The `struct` keyword is followed by an optional identifier that we can use to refer back to our new type, which is now known as `struct callinfo`. The struct members are then enclosed in a set of braces { } and the entire declaration is terminated by a semicolon (`;`). Each struct member is defined using the same syntax as a D variable declaration, with the type of the member listed first followed by an identifier naming the member and another semicolon (`;`). If you want to declare multiple members of the same type you can list the identifiers in a comma-separated list like this:

```
struct callinfo {
    uint64_t ts, elapsed;
    ...
};
```

The struct declaration itself simply defines the new type; it does not create any variables or allocate any storage in DTrace. But once declared, we can use `struct callinfo` as a type throughout the remainder of our D program, and each variable of type `struct callinfo` will store a copy of the four variables described by our structure template. The members will be arranged in memory in order according to the member list, with padding space introduced between members as required for data object alignment purposes.

We can use the member identifier names to access the individual member values using the "`.`" operator by writing an expression of the form:

variable-name`.`*member-name*

We can now write an improved program using our new structure type. Go to your editor and type in the following D program and save it in a file named `rwinfo.d`, changing the PID 12345 to the PID of one of your shells:

EXAMPLE 7–1 rwinfo.d: Gather `read`(2) and `write`(2) Statistics

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};

struct callinfo i[string];    /* declare i as an associative array */

syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    i[probefunc].ts = timestamp;
    i[probefunc].calls++;
    i[probefunc].maxbytes = arg2 > i[probefunc].maxbytes ?
        arg2 : i[probefunc].maxbytes;
}

syscall::read:return, syscall::write:return
/i[probefunc].ts != 0 && pid == 12345/
{
    i[probefunc].elapsed += timestamp - i[probefunc].ts;
}

END
{
    printf("        calls  max bytes  elapsed nsecs\n");
    printf("------  -----  ---------  -------------\n");
    printf("  read  %5d  %9d  %d\n",
        i["read"].calls, i["read"].maxbytes, i["read"].elapsed);
    printf(" write  %5d  %9d  %d\n",
        i["write"].calls, i["write"].maxbytes, i["write"].elapsed);
}
```

After you type in the program and change the PID 12345 to the PID of one of your shells, run `dtrace -q -s rwinfo.d` and then go type in a few commands in your shell. When you're done entering your shell commands, type Control-C in the `dtrace` terminal to fire the `END` probe and print the results:

```
# dtrace -q -s rwinfo.d
^C
        calls  max bytes  elapsed nsecs
------  -----  ---------  -------------
  read     36       1024  3588283144
 write     35         59  14945541
#
```

7.2 Pointers to Structs

It is very common in C and D to refer to structs using pointers, so as a convenience the operator `->` is provided to access struct members through a pointer. If a `struct s` has a member `m` and you have a pointer to this struct named `sp` (that is, `sp` is a variable of type `struct s *`), you can either use the `*` operator to first dereference `sp` pointer in order to access the member:

```
struct s *sp;

(*sp).m
```

or you can use the `->` operator as a shorthand for this notation. The following two D fragments are equivalent in meaning if `sp` is a pointer to a struct:

```
(*sp).m                 sp->m
```

DTrace provides several built-in variables which are pointers to structs, including `curpsinfo` and `curlwpsinfo`. These pointers refer to the structs `psinfo` and `lwpsinfo` respectively, and their content provides a snapshot of information about the state of the current process and lightweight process (LWP) associated with the thread that has fired the current probe. A Solaris LWP is the kernel's representation of a user thread, upon which the Solaris threads and POSIX threads interfaces are built. For convenience, DTrace exports this information in the same form as the `/proc` filesystem files `/proc/`*`pid`*`/psinfo` and `/proc/`*`pid`*`/lwps/`*`lwpid`*`/lwpsinfo`. The `/proc` structures are used by observability and debugging tools such as `ps`(1), `pgrep`(1), and `truss`(1), and are defined in the system header file `<sys/procfs.h>` and are described in the `proc`(4) man page. Here are few example expressions using `curpsinfo`, their types, and their meanings:

`curpsinfo->pr_pid`	`pid_t`	current process ID
`curpsinfo->pr_fname`	`pid_t`	executable file name
`curpsinfo->pr_psargs`	`pid_t`	initial command line arguments

You should review the complete structure definition later by examining the `<sys/procfs.h>` header file and the corresponding descriptions in `proc`(4). We're going to use the `pr_psargs` member to identify a process of interest by matching our command line arguments in the next example.

Structs are used very frequently to create complex data structures in C programs, so the ability to describe and reference structs from D also provides a powerful capability for observing the inner workings of the Solaris operating system kernel and its system interfaces. In addition to using the aforementioned `curpsinfo` struct, we're also

going to peek inside some kernel structs as well in our next example by constructing a program to observe the relationship between the ksyms(7D) driver and read(2) requests. The driver makes use of two common structs, known as uio(9S) and iovec(9S), to respond to requests to read from the character device file /dev/ksyms.

The uio struct, accessed using the name struct uio or type alias uio_t, is described in the uio(9S) man page and is used to describe an i/o request that involves copying data between the kernel and a user process. The uio in turn contains an array of one or more iovec(9S) structures which each describe a piece of the requested i/o, in the event that multiple chunks are requested using the readv(2) or writev(2) system calls. One of the kernel device driver interface (DDI) routines that operates on struct uio is the function uiomove(9F), which is one of a family of functions kernel drivers use to respond to user process read(2) requests and copy data back to user processes.

The ksyms driver manages a character device file named /dev/ksyms, which appears to be an ELF file containing information about the kernel's symbol table, but is in fact an illusion created by the driver using the set of modules that are currently loaded into the kernel. The driver uses the uiomove(9F) routine to respond to read(2) requests. For our example, we'd like to illustrate that the arguments and calls to read(2) from /dev/ksyms match the calls by the driver to uiomove(9F) to copy the results back into the user address space at the location specified to read(2).

We can use the strings(1) utility with the -a option to force a bunch of reads from /dev/ksyms. Try running strings -a /dev/ksyms in your shell and see what output it produces. Now go to your editor and type in the first clause of our example and save it in a file named ksyms.d:

```
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
}
```

Here we are using the expression curpsinfo->pr_psargs to access and match on the command-line arguments of our strings(1) command so print out the correct read(2) requests and report the arguments. Notice that by using operator == with a left-hand argument that is an array of char and a right-hand argument that is a string, we are permitting the D compiler to infer that the left-hand argument should be promoted to a string and a string comparison should be performed. Type in and execute the command dtrace -q -s ksyms.d in one shell, and then type in the command strings -a /dev/ksyms in another shell. As strings(1) executes, you will see output from DTrace similar to the following:

```
# dtrace -q -s ksyms.d
read 8192 bytes at user address 80639fc
read 8192 bytes at user address 80639fc
read 8192 bytes at user address 80639fc
read 8192 bytes at user address 80639fc
...
```

```
^C
#
```

We can extend our example using a common D programming technique to follow a thread from this initial `read`(2) request deeper into the kernel. Upon entry to the kernel in `syscall::read:entry`, we will also set a thread-local flag variable indicating this thread is of interest, and clear this flag on `syscall::read:return`. Once the flag is set, we can use it as a predicate on other probes to instrument kernel functions such as `uiomove`(9F). The DTrace function boundary tracing (`fbt`) provider publishes probes for entry and return to functions defined within the kernel, including those in the DDI. Type in the following source code which uses the `fbt` provider to instrument `uiomove`(9F) and again save it again in the file `ksyms.d`:

EXAMPLE 7–2 ksyms.d: Trace `read`(2) and `uiomove`(9F) Relationship

```
/*
 * When our strings(1) invocation starts a read(2), set a watched flag on
 * the current thread.  When the read(2) finishes, clear the watched flag.
 */
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
    self->watched = 1;
}

syscall::read:return
/self->watched/
{
    self->watched = 0;
}

/*
 * Instrument uiomove(9F).  The prototype for this function is as follows:
 * int uiomove(caddr_t addr, size_t nbytes, enum uio_rw rwflag, uio_t *uio);
 */
fbt::uiomove:entry
/self->watched/
{
    this->iov = args[3]->uio_iov;

    printf("uiomove %u bytes to %p in pid %d\n",
        this->iov->iov_len, this->iov->iov_base, pid);
}
```

The final clause of our completed example uses the thread-local variable `self->watched` to identify when a kernel thread of interest enters the DDI routine `uiomove`(9F). Once there, we use the built-in `args` array to access the fourth argument (`args[3]`) to `uiomove()`, which is a pointer to the `struct uio` representing the request. The D compiler automatically associates each member of the `args` array with the type corresponding to the C function prototype for the

instrumented kernel routine. The `uio_iov` member contains a pointer to the `struct iovec` for the request, and we save a copy of this pointer for use in our clause in the clause-local variable `this->iov`. In the final statement, we further dereference `this->iov` to access the `iovec` members `iov_len` and `iov_base`, which represent the length in bytes and destination base address for `uiomove`(9F), respectively. We expect these values to match the input parameters to the `read`(2) system call issued on the driver. Go to your shell and run `dtrace -q -s ksyms.d` and then again enter the command `strings -a /dev/ksyms` in another shell. You should see output like this:

```
# dtrace -q -s ksyms.d
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
...
^C
#
```

The addresses and process IDs will be different in your output, but you should observe that the input arguments to `read`(2) match the parameters passed to `uiomove`(9F) by the ksyms driver.

7.3 Unions

Unions are another kind of composite type supported by ANSI-C and D, and are closely related to structs. A union is a composite type where a set of members of different types are defined and the member objects all occupy the same region of storage. A union is therefore an object of variant type, where only one member is valid at any given time, depending on how the union has been assigned. Typically, some other variable or piece of state is used to indicate which union member is currently valid. The size of a union is the size of its largest member, and the memory alignment used for the union is the maximum alignment required by the union members.

The Solaris `kstat` framework defines a struct containing a union that we can use to illustrate and observe C and D unions. The `kstat` framework is used to export a set of named counters representing kernel statistics such as memory usage and i/o throughput and is used to implement utilities such as `mpstat`(1M) and `iostat`(1M). This framework uses `struct kstat_named` to represent a named counter and its value and is defined as follows:

```
struct kstat_named {
    char name[KSTAT_STRLEN]; /* name of counter */
    uchar_t data_type;    /* data type */
    union {
        char c[16];
        int32_t i32;
        uint32_t ui32;
        long l;
        ulong_t ul;
        ...
    } value;    /* value of counter */
};
```

We have shortened the declaration for illustrative purposes; the complete structure definition can be found in the `<sys/kstat.h>` header file and is described in `kstat_named`(9S). The declaration above is valid in both ANSI-C and D, and defines a struct containing as one of its members a union value with members of various types, depending on the type of the counter. Notice that since the union itself is declared inside of another type, `struct kstat_named`, a formal name for the union type is omitted. This declaration style is known as an *anonymous union*; the member named `value` is of a union type described by the preceding declaration, but this union type itself has no name because it does not need to be used anywhere else. The struct member `data_type` is assigned a value that indicates which union member is valid for each object of type `struct kstat_named`. A set of C preprocessor tokens are defined for the values of `data_type` (for example, the token `KSTAT_DATA_CHAR` is equal to zero and indicates that the member `value.c` is where the value is currently stored).

We're going to demonstrate accessing the `kstat_named.value` union by writing our first D program to trace a user process. The `kstat` counters can be sampled from a user process using the `kstat_data_lookup`(3KSTAT) function, which returns a pointer to a `struct kstat_named`. The `mpstat`(1M) utility calls this function repeatedly as it executes in order to sample the latest counter values. Go to your shell and try running `mpstat 1` and observe the output. Type Control-C in your shell to abort `mpstat` after a few seconds. To observe counter sampling, we would like to enable a probe that fires each time the `mpstat` command calls the `kstat_data_lookup`(3KSTAT) function in `libkstat`. To do so, we're going to make use of a new DTrace provider: `fasttrap`. The `fasttrap` provider permits you to dynamically create probes in user processes at C symbol locations such as function entry points. You can ask the `fasttrap` provider to create a probe at a user function entry and return sites by writing probe descriptions of the form:

`pid`*process-ID*`:`*object-name*`:`*function-name*`:entry`
`pid`*process-ID*`:`*object-name*`:`*function-name*`:return`

For example, if you want to create a probe in process ID 12345 that fires on entry to `kstat_data_lookup`(3KSTAT), you can write the probe description:

```
pid12345:libkstat:kstat_data_lookup:entry
```

The `pid` provider inserts dynamic instrumentation into the specified user process at the program location corresponding to the probe description. The probe implementation forces each user thread that reaches the instrumented program location to trap into the operating system kernel and enter DTrace, firing the corresponding probe. So although the instrumentation location is associated with a user process, the DTrace predicates and actions you specify still execute in the context of the operating system kernel. We'll discuss the `pid` provider in further detail in Chapter 25. For now, we'll use only this simple probe description to complete our `kstat` example.

Instead of having to edit your D program source each time you wish to apply your program to a different process, the D compiler also permits you to insert identifiers called *macro variables* into your program that are evaluated at the time your program is compiled and replaced with the additional `dtrace` command-line arguments. Macro variables are specified in your D program using a dollar sign `$` followed by an identifier or digit. If you execute the command `dtrace -s` *script* `foo bar baz`, the D compiler will automatically define the macro variables `$1`, `$2`, and `$3` to be the tokens `foo`, `bar`, and `baz` respectively. You can use macro variables in D program expressions, or in probe descriptions. For example, the probe description:

```
pid$1:libkstat:kstat_data_lookup:entry
{
    self->ksname = arg1;
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 != NULL/
{
    this->ksp = (kstat_named_t *)copyin(arg1, sizeof (kstat_named_t));
    printf("%s has ui64 value %u\n", copyinstr(self->ksname),
        this->ksp->value.ui64);
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
    self->ksname = NULL;
}
```

can be used to write a D program that instruments whatever process ID is specified as an additional argument to `dtrace`. We discuss macro variables and reusable scripts in further detail in Chapter 15. Now that we know how to instrument user processes using their process ID, let's return to sampling unions. Go to your editor and type in the source code for our complete example and save it in a file named `kstat.d`:

EXAMPLE 7–3 kstat.d: Trace Calls to `kstat_data_lookup`(3KSTAT)

Now go to one of your shells and execute the command `mpstat 1` to start `mpstat`(1M) running in a mode where it samples kstats and reports them once per second. Once `mpstat` is running, execute the command `dtrace -q -s kstat.d 'pgrep mpstat'` in your other shell. You will see output corresponding to the kstats that are being accessed. Type Control-C to abort `dtrace` and return to the shell prompt.

```
# dtrace -q -s kstat.d 'pgrep mpstat'
cpu_ticks_idle has ui64 value 41154176
cpu_ticks_user has ui64 value 1137
cpu_ticks_kernel has ui64 value 12310
cpu_ticks_wait has ui64 value 903
hat_fault has ui64 value 0
as_fault has ui64 value 48053
maj_fault has ui64 value 1144
xcalls has ui64 value 123832170
intr has ui64 value 165264090
intrthread has ui64 value 124094974
pswitch has ui64 value 840625
inv_swtch has ui64 value 1484
cpumigrate has ui64 value 36284
mutex_adenters has ui64 value 35574
rw_rdfails has ui64 value 2
rw_wrfails has ui64 value 2
...
^C
#
```

If you capture the output in each terminal window and subtract each value from the value reported by the previous iteration through the kstats, you should be able to correlate the `dtrace` output with the `mpstat` output. Our program records the counter name pointer on entry to the kstat lookup function, and then performs most of its work on return from `kstat_data_lookup`(3KSTAT). We use the D built-in functions `copyinstr()` and `copyin()` to copy the function results from the user process back into DTrace when `arg1` (the return value) is not `NULL`. Once we have the `kstat` data, we report the `ui64` counter value from the union. Notice that we simplified our example by assuming that `mpstat` samples counters that use the `value.ui64` member. As an exercise, try recoding `kstat.d` to use multiple predicates and print out the union member corresponding to the `data_type` member. You can also try to create a version of `kstat.d` that computes the difference between successive data values and actually produces output similar to `mpstat`.

7.4 Member Sizes and Offsets

You can determine the size in bytes of any D type or expression, including a struct or union, using the `sizeof` operator. The `sizeof` operator can be applied either to an expression or to the name of a type surrounded by parentheses, as illustrated by the following two examples:

`sizeof` *expression* `sizeof ( type-name )`

For example, the expression `sizeof (uint64_t)` would return the value 8, and the expression `sizeof (callinfo.ts)` would also return 8 if inserted into the source code of our example program above. The formal return type of the `sizeof` operator is the type alias `size_t`, which is defined to be an unsigned integer of the same size as a pointer in the current data model, and is used to represent byte counts. When the `sizeof` operator is applied to an expression, the expression is validated by the D compiler but the resulting object size is computed at compile time and no code for the expression is generated. You can use `sizeof` anywhere an integer constant is required.

You can use the companion operator `offsetof` to determine the offset in bytes of a struct or union member from the start of the storage associated with any object of the struct or union type. The `offsetof` operator is used in an expression of the following form:

`offsetof ( type-name , member-name )`

Where *type-name* is the name of any struct or union type or type alias thereof, and *member-name* is the identifier naming a member of that struct or union. Similar to `sizeof`, `offsetof` returns a `size_t` and can be used anywhere in a D program that an integer constant can be used.

7.5 Bit-Fields

D also permits the definition of integer struct and union members of arbitrary numbers of bits, known as *bit-fields*. A bit-field is declared by specifying a signed or unsigned integer base type, a member name, and a suffix indicating the number of bits to be assigned for the field, as shown in the following example:

```
struct s {
    int a : 1;
    int b : 3;
    int c : 12;
```

```
};
```

The bit-field width is an integer constant separated from the member name by a trailing colon. The bit-field width must be positive and must be of a number of bits not larger than the width of the corresponding integer base type. Bit-fields larger than 64 bits may not be declared in D. D bit-fields provide compatibility with and access to the corresponding ANSI-C capability. Bit-fields are typically used in situations when memory storage is at a premium or when a struct layout must match a hardware register layout.

A bit-field is simply a compiler construct that automates the layout of an integer and a set of masks to extract the member values; the same result can be achieved by simply defining the masks yourself and using the `&` operator. C and D compilers try to pack bits as efficiently as possible, but they are free to do so in any order or fashion they desire, so bit-fields are not guaranteed to produce identical bit layouts across differing compilers or architectures. If you require stable bit layout, you should construct the bit masks yourself and extract the values using the `&` operator.

A bit-field member is accessed like any other struct or union member by simply specifying its name in combination with the "`.`" or `->` operators. The bit-field will be automatically promoted to the next largest integer type for use in any expressions. As bit-field storage may not be aligned on a byte boundary or be a round number of bytes in size, you may not apply the `sizeof` or `offsetof` operators to a bit-field member. The D compiler also prohibits you from taking the address of a bit-field member using the `&` operator.

CHAPTER 8

Type and Constant Definitions

In the preceding chapters, we have learned about the primitive D data types as well as how to build more complex types out of these primitives. In this chapter we learn how to declare type aliases and named constants in D, and then use these constructs to motivate a discussion of D type and namespace management for program and operating system types and identifiers.

8.1 Typedef

The `typedef` keyword is used to declare an identifier as an alias for an existing type. Like all D type declarations, the `typedef` keyword is used outside of probe clauses in a declaration of the form:

`typedef` *existing-type new-type* `;`

where *existing-type* is any type declaration and *new-type* is an identifier to be used as the alias for this type. For example, the declaration:

```
typedef unsigned char uint8_t;
```

is used internally by the D compiler to create the `uint8_t` type alias described earlier. Typedef aliases can be used anywhere that a normal type can be used, such as the type of a variable or associative array value or tuple member. You can also combine `typedef` with more elaborate declarations such as the definition of a new `struct`:

```
typedef struct foo {
    int x;
    int y;
} foo_t;
```

In this example, `struct foo` is defined as the same type as its alias, `foo_t`. Solaris C system headers often use the suffix `_t` to denote a `typedef` alias.

8.2 Enumerations

It is useful to define symbolic names for constants in a program to ease readability and simplify the process of maintaining the program in the future. One method is to define an *enumeration*, which associates a set of integers with a set of identifiers called enumerators that the compiler recognizes and replaces with the corresponding integer value. An enumeration is defined using a declaration such as:

```
enum colors {
    RED,
    GREEN,
    BLUE
};
```

The first enumerator in the enumeration, `RED`, is assigned the value zero and each subsequent identifier is assigned the next integer value. You can also specify an explicit integer value for any enumerator by suffixing it with an equal sign and an integer constant, as in:

```
enum colors {
    RED = 7,
    GREEN = 9,
    BLUE
};
```

The enumerator `BLUE` is assigned the value 10 by the compiler since it has no value specified and the previous enumerator is set to 9. Once an enumeration is defined, the enumerators can be used anywhere in a D program that an integer constant can be used. In addition, the enumeration `enum colors` is also defined as a type that is equivalent to an `int`. The D compiler will allow a variable of `enum` type to be used anywhere an `int` can be used, and will allow any integer value to be assigned to a variable of `enum` type. You can also omit the `enum` name in the declaration if the type name is not needed.

Enumerators are visible in all subsequent clauses and declarations in your program, so you cannot define the same enumerator identifier in more than one enumeration. However, you may define more than one enumerator that has the same value in either the same or different enumerations. You may also assign integers that have no corresponding enumerator to a variable of the enumeration type.

The D enumeration syntax is the same as the corresponding syntax in ANSI-C, and D provides access to enumerations defined in the operating system kernel and its loadable modules, but these enumerators are not globally visible in your D program. They are only visible when used as an argument to one of the binary comparison operators when compared to an object of the corresponding enumeration type. For example, the function `uiomove`(9F) we instrumented in Chapter 7 has a parameter of type `enum uio_rw` defined as follows:

```
enum uio_rw { UIO_READ, UIO_WRITE };
```

The enumerators UIO_READ and UIO_WRITE are not normally visible in your D program, but you can promote them to global visibility by comparing one of them to a value of type `enum uio_rw`, as shown in the following example clause:

```
fbt::uiomove:entry
/args[2] == UIO_WRITE/
{
    ...
}
```

In this example, we instrument `uiomove`(9F) for write requests by comparing `args[2]`, a variable of type `enum uio_rw`, to the enumerator `UIO_WRITE`. Since the left-hand argument is an enumeration type, the D compiler knows to search it when attempting to resolve the right-hand identifier. This feature protects your D programs against inadvertent identifier name conflicts with the large collection of enumerations defined in the operating system kernel.

8.3 Inlines

D named constants can also be defined using `inline` directives, which provide a more general means of creating identifiers that are replaced by predefined values or expressions during compilation. Inline directives are a more powerful form of lexical replacement than the `#define` directive provided by the C preprocessor because the replacement is assigned an actual type and is performed using the compiled syntax tree and not simply a set of lexical tokens. An inline directive is specified using a declaration of the form:

`inline` *type name* `=` *expression* `;`

where *type* is a type declaration of an existing type, *name* is any valid D identifier that is not previously defined as an inline or global variable, and *expression* is any valid D expression. Once the inline directive is processed, the D compiler will substitute the compiled form of *expression* for each subsequent instance of *name* in the program source. For example, the following D program would trace the string `"hello"` and integer value `123`:

```
inline string hello = "hello";
inline int number = 100 + 23;

BEGIN
{
    trace(hello);
    trace(number);
}
```

An inline name may be used anywhere a global variable of the corresponding type can be used. If the inline expression can be evaluated to an integer or string constant at compile time, then the inline name can also be used in contexts that require constant expressions, such as scalar array dimensions.

The inline expression is validated for syntax errors as part of evaluating the directive, and the expression result type must be compatible with the type defined by the inline, according to the same rules used for the D assignment operator (=). An inline expression may not reference the inline identifier itself (that is, recursive definitions are not permitted).

The DTrace software packages install a number of D source files in the system directory `/usr/lib/dtrace` that contain inline directives you can use in your D programs. For example, the `signal.d` library includes directives of the form:

```
inline int SIGHUP = 1;
inline int SIGINT = 2;
inline int SIGQUIT = 3;
...
```

to provide you access to the current set of Solaris signal names described in `signal`(3HEAD). Similarly, the `errno.d` library contains inline directives for the C `errno` constants described in `Intro`(2).

By default, the D compiler includes all of the provided D library files for you automatically so you can use these definitions in any D program you write.

8.4 Type Namespaces

We have already discussed a variety of namespaces for D identifiers, including global variables, thread-local variables, and clause-local variables. We now complete our discussion of D namespaces by discussing namespace issues related to types. In traditional languages such as ANSI-C, type visibility is determined by whether or not a type is nested inside of a function or other declaration. Types declared at the outer scope of a C program are associated with a single global namespace and visible throughout the entire program. Types defined in C header files are typically included in this outer scope. Unlike these languages, D provides access to types from multiple outer scopes.

D is designed as a language to facilitate dynamic observability across multiple layers of a software stack, including the operating system kernel, an associated set of loadable kernel modules, and user processes running on the system. Since a single D program may instantiate probes to gather data from multiple kernel modules or other software entities that are compiled into independent binary objects, more than one data type of the same name, perhaps with different definitions, may be present in the universe of types available to DTrace and the D compiler. To manage this situation, the D compiler associates each type with a namespace identified by the containing program object. Types from a particular program object can be accessed by specifying the object name and backquote (`) scoping operator in any type name.

For example, if a kernel module named `foo` contains the following C type declaration:

```
typedef struct bar {
    int x;
} bar_t;
```

then the types `struct bar` and `bar_t` could be accessed from D using the type names:

```
struct foo`bar                foo`bar_t
```

The backquote operator can be used in any context where a type name is appropriate, including when specifying the type for D variable declarations or cast expressions in D probe clauses.

The D compiler also provides two special built-in type namespaces accessed using the names "`C`" and "`D`" respectively. The C type namespace is initially populated with the standard ANSI-C intrinsic types such as `int`. In addition, type definitions acquired using the C preprocessor `cpp`(1) via the `dtrace -C` option will be processed using and added to the C scope. As a result, you can `#include` C header files containing type declarations which are already visible in another type namespace without causing a compilation error.

The D type namespace is initially populated with the D type intrinsics such as `int` and `string` as well as the built-in D type aliases such as `uint32_t`. Any new type declarations that appear in the D program source itself are automatically added to the D type namespace. If you create a complex type such as a `struct` in your D program consisting of member types from other namespaces, they will be copied into the D namespace by the declaration.

When the D compiler encounters a type declaration that does not specify an explicit namespace using the backquote operator, it searches the set of active type namespaces to find a match using the specified type name. The C namespace is always searched first followed by the D namespace. If the type name is not found in either the C or D namespace, the type namespaces of the active kernel modules are searched in ascending order by kernel module ID. This ordering guarantees that the binary objects that form the core kernel are searched before any loadable kernel modules, but does not guarantee any ordering properties among the loadable modules. You should use the scoping operator when accessing types defined in loadable kernel modules to avoid type name conflicts with other kernel modules.

The D compiler consumes compressed ANSI-C debugging information provided with the core Solaris kernel modules in order to automatically access the types associated with the operating system source code without the need for accessing the corresponding C include files. This symbolic debugging information may not be available for all kernel modules on your system. The D compiler will report an error if you attempt to access a type within the namespace of a module that lacks compressed C debugging information intended for use with DTrace.

CHAPTER 9

Aggregations

When instrumenting the system to answer performance-related questions, it is often useful to think not in terms of data gathered by individual probes, but rather how that data can be aggregated to answer a specific question. For example, if you wished to know the number of system calls by user ID, you would not necessarily care about the datum collected at *each* system call — you simply want to see a table of user IDs and system calls. Historically, one would answer this question by gathering data at each system call, and postprocessing the data using a tool like `awk`(1) or `perl`(1). However, in DTrace the aggregating of data is a first-class operation. In this chapter, we explore the DTrace facilities for manipulating *aggregations*.

9.1 Aggregating Functions

We define an *aggregating function* to be one that has the following property:

$$f(f(x_0)\ U\ f(x_1)\ U\ \ldots\ U\ f(x_n)) = f(x_0\ U\ x_1\ U\ \ldots\ U\ x_n)$$

where x_n is a set of arbitrary data. That is, applying an aggregating function to subsets of the whole and then applying it again to the results gives the same result as applying it to the whole itself. This is most easily seen in a comprehensive example. Take a function `SUM` that, given a set of data, yields the summation of that set. If the raw data consists of {2, 1, 2, 5, 4, 3, 6, 4, 2}, the result of applying `SUM` to the entire set is {29}. Similarly, the result of applying `SUM` to the subset consisting of the first three elements is {5}, the result of applying `SUM` to the set consisting of the subsequent three elements is {12}, and the result of of applying `SUM` to the remaining three elements is also {12}. `SUM` is an aggregating function because applying it to the set of these results, {5, 12, 12}, yields the same result — {29} — as applying `SUM` to the original data.

Not all functions are aggregating functions. An example of a non-aggregating function is the function `MEDIAN` that determines the median element of the set. (The median is defined to be that element of a set for which as many elements in the set are greater than it as are less than it.) The `MEDIAN` is derived by sorting the set and selecting the middle element. Returning to our original raw data, if we apply `MEDIAN` to the set consisting of the first three elements, the result is {2}. (The sorted set is {1, 2, 2}; {2} is the set consisting of the middle element.) Likewise, applying `MEDIAN` to the next three elements yields {4} and applying `MEDIAN` to the final three elements yields {4}. Applying `MEDIAN` to each of the subsets thus yields the set {2, 4, 4}; applying `MEDIAN` to this set yields the result {4}. However, sorting the original set yields {1, 2, 2, 2, 3, 4, 4, 5, 6}; applying `MEDIAN` to this set thus yields {3}. Because these results do not match, `MEDIAN` is *not* an aggregating function.

Fortunately, many common functions for understanding a set of data are aggregating functions. These include counting the number of elements in the set, computing the minimum value of the set, computing the maximum value of the set, and summing all elements in the set. (Note that determining the arithmetic mean of the set can be trivially built on functions to count the number of elements in the set and to sum the number the elements in the set.)

However, several other useful functions are not aggregating functions. These include computing the mode (the most common element) of a set, the median value of the set, or the standard deviation of the set.

Applying aggregating functions to data *in situ* has a number of advantages:

- The entire data set need not be stored. Whenever a new element is to be added to the set, the aggregating function is calculated given the set consisting of the current intermediate result and the new element. After the new result is calculated, the new element may be discarded. This reduces the amount of storage required by a factor of the number of data points, which is often quite large.
- A scalable implementation is allowed. One does not wish for data collection to induce pathological scalability problems. Aggregating functions allow for intermediate results to be kept *per-CPU* instead of in a shared data structure. When a system-wide result is desired, the aggregating function may then be applied to the set consisting of the per-CPU intermediate results.

9.2 Aggregations

DTrace stores the results of aggregating functions in objects called *aggregations*. The aggregation results are indexed using a tuple of expressions similar to those used for associative arrays. In D, the syntax for an aggregation is:

@name `[` *keys* `]` `=` *aggfunc* `(` *args* `);`

where *name* is the name of the aggregation, *keys* is a comma-separated list of D expressions, *aggfunc* is one of the DTrace aggregating functions, and *args* is a comma-separated list of arguments appropriate for the aggregating function. The aggregation *name* is a D identifier that is prefixed with the special character `@`. All aggregations named in your D programs are global variables; there are no thread- or clause-local aggregations. The aggregation names are kept in a separate identifier namespace from other D global variables. Remember that `a` and `@a` are not the same variable if you reuse names. The special aggregation name `@` can be used to name an anonymous aggregation in simple D programs: the D compiler treats it as an alias for the aggregation name `@_`.

The DTrace aggregating functions are shown in the table below. Most aggregating functions take just a single argument that represents the new datum.

TABLE 9–1 DTrace Aggregating Functions

Function Name	Arguments	Result
`count`	none	The number of times called.
`sum`	scalar expression	The total value of the specified expressions.
`avg`	scalar expression	The arithmetic average of the specified expressions.
`min`	scalar expression	The smallest value among the specified expressions.
`max`	scalar expression	The largest value among the specified expressions.
`lquantize`	scalar expression, lower bound, upper bound, step value	A linear frequency distribution, sized by the specified range, of the values of the specified expressions. Increments the value in the *highest* bucket that is *less* than the specified expression.
`quantize`	scalar expression	A power-of-two frequency distribution of the values of the specified expressions. Increments the value in the*highest* power-of-two bucket that is *less* than the specified expression.

For example, to count the number of `write`(2) system calls in the system, you could use an informative string as a key and the `count()` aggregating function:

```
syscall::write:entry
{
    @counts["write system calls"] = count();
}
```

The `dtrace` command prints aggregation results by default when the process terminates – either as the result of an explicit `END` action or when the user types Control-C. Running this, waiting a bit, and typing Control-C yields:

```
# dtrace -s writes.d
dtrace: script './writes.d' matched 1 probe
^C
```

```
  write system calls                                              179
#
```

It may be more interesting to count system calls per process name. This can be done by using the `execname` variable as the key to aggregation:

```
syscall::write:entry
{
    @counts[execname] = count();
}
```

Running the above, and again waiting a bit before typing Control-C:

```
# dtrace -s writesbycmd.d
dtrace: script './writesbycmd.d' matched 1 probe
^C

  dtrace                                                            1
  cat                                                               4
  sed                                                               9
  head                                                              9
  grep                                                             14
  find                                                             15
  tail                                                             25
  mountd                                                           28
  expr                                                             72
  sh                                                              291
  tee                                                             814
  def.dir.flp                                                    1996
  make.bin                                                       2010
#
```

Alternatively, one may wish to have writes broken down by both executable name and file descriptor. The file descriptor is the first argument to `write`(2), so we key off both `execname` and `arg0`:

```
syscall::write:entry
{
    @counts[execname, arg0] = count();
}
```

Running this yields a table with both executable name and file descriptor:

```
# dtrace -s writesbycmdfd.d
dtrace: script './writesbycmdfd.d' matched 1 probe
^C

  cat                                                               1       58
  sed                                                               1       60
  grep                                                              1       89
  tee                                                               1      156
```

```
  tee                                                               3     156
  make.bin                                                          5     164
  acomp                                                             1     263
  macrogen                                                          4     286
  cg                                                                1     397
  acomp                                                             3     736
  make.bin                                                          1     880
  iropt                                                             4    1731
#
```

We may wish to know the average time spent in the write system call, by process name. To do this, we use the `avg()` aggregating function, specifying the expression that we wish to average as the argument. In this case, we are averaging the wall time spent in the system call:

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = avg(timestamp - self->ts);
    self->ts = 0;
}
```

Running the above, and waiting a bit before typing Control-C:

```
# dtrace -s writetime.d
dtrace: script './writetime.d' matched 2 probes
^C

  iropt                                                         31315
  acomp                                                         37037
  make.bin                                                      63736
  tee                                                           68702
  date                                                          84020
  sh                                                            91632
  dtrace                                                       159200
  ctfmerge                                                     321560
  install                                                      343300
  mcs                                                          394400
  get                                                          413695
  ctfconvert                                                   594400
  bringover                                                   1332465
  tail                                                        1335260
#
```

The average can be useful, but it may not give one as much of a feel for the distribution as one may like. To do this, we use the `quantize()` aggregating function:

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Because each line of output becomes a frequency distribution diagram, the output of this script is substantially longer than previous ones; here is a snippet:

```
  lint
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |                                         2
           32768 |                                         0
           65536 |@@@@@@@@@@@@@@@@@@@@                     74
          131072 |@@@@@@@@@@@@@@@@                         59
          262144 |@@@                                      14
          524288 |                                         0

  acomp
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@                             840
           16384 |@@@@@@@@@@@                              750
           32768 |@@                                       165
           65536 |@@@@@@                                   460
          131072 |@@@@@@                                   446
          262144 |                                         16
          524288 |                                         0
         1048576 |                                         1
         2097152 |                                         0

  iropt
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@               4149
           16384 |@@@@@@@@@@                               1798
           32768 |@                                        332
           65536 |@                                        325
          131072 |@@                                       431
          262144 |                                         3
          524288 |                                         2
         1048576 |                                         1
         2097152 |                                         0
```

Note that the bucket values for the frequency distribution are *always* power-of-two values. Note, too, that each bucket contains a count of the number of elements *greater than or equal to* the corresponding value, but *less than* the next larger value. For example, from the above output, we know that `iropt` had 4,149 writes taking between 8,192 nanoseconds and 16,383 nanoseconds, inclusive.

`quantize()` is useful for getting quick insight into the data, but you may want instead a distribution across linear values. To do this, use the `lquantize()` aggregating function. The `lquantize()` function takes three arguments in addition to a D expression: a lower bound, an upper bound, and a step. For example, if one wished to look at the distribution of writes by file descriptor, a power-of-two quantization doesn't make much sense; one should instead use a linear quantization with a small range:

```
syscall::write:entry
{
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

Running this for several seconds yields quite a bit of output; here is a snippet:

```
  mountd
           value  ------------- Distribution ------------- count
              11 |                                         0
              12 |@                                        4
              13 |                                         0
              14 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              70
              15 |                                         0
              16 |@@@@@@@@@@@@@                            34
              17 |                                         0

  xemacs-20.4
           value  ------------- Distribution ------------- count
               6 |                                         0
               7 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  521
               8 |                                         0
               9 |                                         1
              10 |                                         0

  make.bin
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  3596
               2 |                                         0
               3 |                                         0
               4 |                                         42
               5 |                                         50
               6 |                                         0

  acomp
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@                                    1156
```

```
               2 |                                         0
               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    6635
               4 |@                                        297
               5 |                                         0

  iropt
           value  ------------- Distribution ------------- count
               2 |                                         0
               3 |                                         299
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  20144
               5 |                                         0
```

Another interesting use of the `lquantize()` aggregating function is to aggregate on time since some point in the past. This allows one to see change in behavior over time. For example, to see the change in system call behavior over the lifetime of a process executing the `date`(1) command, try this D script:

```
syscall::exec:return,
syscall::exece:return
/execname == "date"/
{
    self->start = vtimestamp;
}

syscall:::entry
/self->start/
{
    /*
     * We linearly quantize on the current virtual time minus our
     * process's start time.  We divide by 1000 to yield microseconds
     * rather than nanoseconds.  The range runs from 0 to 10 milliseconds
     * in steps of 100 microseconds; we expect that no date(1) process
     * will take longer than 10 milliseconds to complete.
     */
    @a["system calls over time"] =
        lquantize((vtimestamp - self->start) / 1000, 0, 10000, 100);
}

syscall::rexit:entry
/self->start/
{
    self->start = 0;
}
```

The above script is most interesting to run when many `date`(1) processes are executed. To do this, run `sh -c 'while true; do date >/dev/null; done'` in one window, while executing the D script in another. This yields a profile of the system call behavior of the `date`(1) command:

```
# dtrace -s dateprof.d
dtrace: script './dateprof.d' matched 218 probes
^C
```

```
system calls over time
         value  ------------- Distribution ------------- count
           < 0 |                                         0
             0 |@@                                       20530
           100 |@@@@@@                                   48814
           200 |@@@                                      28119
           300 |@                                        14646
           400 |@@@@@                                    41237
           500 |                                         1259
           600 |                                         218
           700 |                                         116
           800 |@                                        12783
           900 |@@@                                      28133
          1000 |                                         7897
          1100 |@                                        14065
          1200 |@@@                                      27549
          1300 |@@@                                      25715
          1400 |@@@@                                     35011
          1500 |@@                                       16734
          1600 |                                         498
          1700 |                                         256
          1800 |                                         369
          1900 |                                         404
          2000 |                                         320
          2100 |                                         555
          2200 |                                         54
          2300 |                                         17
          2400 |                                         5
          2500 |                                         1
          2600 |                                         7
          2700 |                                         0
```

From this output, one can get a rough idea of the different phases of the `date`(1) command with respect to the services required of the kernel. If we wanted to better understand these phases, it may be useful to understand which system calls are being called when. To do this, we could change the D script to aggregate on the variable `probefunc` instead of a constant string.

9.3 Output

If multiple aggregations are present in a D program, they will be displayed in the order that they are introduced in the program. This behavior may be overridden by using the `printa()` function to print the aggregations. The `printa()` function also permits you to precisely format the aggregation data using a format string, as described in Chapter 12.

If an aggregation is not formatted with a `printa()` statement in your D program, the `dtrace` command will snapshot the aggregation data and print the results once after tracing has completed using the default aggregation format. If a given aggregation is formatted using a `printa()` statement, the default behavior is disabled. You can achieve equivalent results by adding the statement `printa(`*`@aggregation-name`*`)` to a `dtrace:::END` probe clause in your program. The default output format for the `avg()`, `count()`, `min()`, `max()`, and `sum()` aggregating functions is to display an integer decimal value corresponding to the aggregated value for each tuple. The default output format for the `lquantize()` and `quantize()` aggregating functions is to display an ASCII table of the results. Aggregation tuples are printed as if `trace()` had been applied to each tuple element.

9.4 Normalization

When aggregating data over some period of time, it may be desirable to *normalize* the data with respect to some constant factor. This may allow disjoint data to be more readily compared. For example, if aggregating system calls, you may wish to output system calls as a per-second rate instead of as an absolute value over the course of the run. To allow for this, DTrace provides the `normalize()` action. The parameters to `normalize()` are an aggregation and a normalization factor; the output of the aggregation will be each value divided by the normaliztion factor.

For example, to aggregate based on system call:

```
#pragma D option quiet

BEGIN
{
    /*
     * Get the start time, in nanoseconds.
     */
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

END
{
    /*
     * Normalize the aggregation based on the number of seconds we have
     * been running.  (There are 1,000,000,000 nanoseconds in one second.)
     */
    normalize(@func, (timestamp - start) / 1000000000);
}
```

Running the above for a little while yields the following output on a desktop machine:

```
# dtrace -s ./normalize.d
 ^C
  syslogd                                                     0
  rpc.rusersd                                                 0
  utmpd                                                       0
  xbiff                                                       0
  in.routed                                                   1
  sendmail                                                    2
  echo                                                        2
  FvwmAuto                                                    2
  stty                                                        2
  cut                                                         2
  init                                                        2
  pt_chmod                                                    3
  picld                                                       3
  utmp_update                                                 3
  httpd                                                       4
  xclock                                                      5
  basename                                                    6
  tput                                                        6
  sh                                                          7
  tr                                                          7
  arch                                                        9
  expr                                                       10
  uname                                                      11
  mibiisa                                                    15
  dirname                                                    18
  dtrace                                                     40
  ksh                                                        48
  java                                                       58
  xterm                                                     100
  nscd                                                      120
  fvwm2                                                     154
  prstat                                                    180
  perfbar                                                   188
  Xsun                                                     1309
  .netscape.bin                                            3005
```

`normalize()` sets the normalization factor for the specified aggregation, but it does not modify the underlying data. This allows the data to be *denormalized* with the `denormalize()` function. `denormalize()` takes only an aggregation. Modifying the early example, we can provide both raw system call counts and per-second rates:

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

syscall:::entry
{
```

```
    @func[execname] = count();
}

END
{
    this->seconds = (timestamp - start) / 1000000000;
    printf("Ran for %d seconds.\n", this->seconds);

    printf("Per-second rate:\n");
    normalize(@func, this->seconds);
    printa(@func);

    printf("\nRaw counts:\n");
    denormalize(@func);
    printa(@func);
}
```

Running the above for a little while:

```
# dtrace -s ./denorm.d
^C
Ran for 14 seconds.
Per-second rate:

  syslogd                                                   0
  in.routed                                                 0
  xbiff                                                     1
  sendmail                                                  2
  elm                                                       2
  picld                                                     3
  httpd                                                     4
  xclock                                                    6
  FvwmAuto                                                  7
  mibiisa                                                  22
  dtrace                                                   42
  java                                                     55
  xterm                                                    75
  adeptedit                                               118
  nscd                                                    127
  prstat                                                  179
  perfbar                                                 184
  fvwm2                                                   296
  Xsun                                                    829

Raw counts:

  syslogd                                                   1
  in.routed                                                 4
  xbiff                                                    21
  sendmail                                                 30
  elm                                                      36
  picld                                                    43
  httpd                                                    56
  xclock                                                   91
```

```
  FvwmAuto                                                      104
  mibiisa                                                       314
  dtrace                                                        592
  java                                                          774
  xterm                                                        1062
  adeptedit                                                    1665
  nscd                                                         1781
  prstat                                                       2506
  perfbar                                                      2581
  fvwm2                                                        4156
  Xsun                                                        11616
```

Aggregations may also be renormalized: if `normalize()` is called more than once for the same aggregation, the normalization factor will be that specified in the most recent call. This may be useful to print per-second rates over time:

EXAMPLE 9–1 renormalize.d: Renormalizing an aggregation

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - start) / 1000000000);
    printa(@func);
}
```

9.5 Clearing aggregations

If using DTrace to build simple monitoring scripts, it may be useful to periodically clear the values in an aggregation. An aggregation's values are cleared using the `clear()` function, which takes an aggregation as its only parameter. Note that the `clear()` function clear only the aggregation's *values*; the aggregation's keys are retained. This is by design: the presence of a key in an aggregation that has an associated value of zero indicates that the key *had* a non-zero value that was subsequently set to zero as part of a `clear()`.

Recasting Example 9–1 using `clear()`:

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall::::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

Unlike Example 9–1 — which will show the system call rate ove the lifetime of the `dtrace` invocation — this will show the system call rate only for the most recent ten second period.

9.6 Minimizing drops

DTrace buffers some aggregation data in the kernel. As a result, it is possible for no space to be available when a new key is added to an aggregation. In this case, the data will be dropped, a counter will be incremented, and `dtrace` will generate a message indicate an aggregation drop. While this is in principle possible, the implementation of DTrace is designed to practically eliminate such drops by keeping long-running state (consisting of the aggregation's key and intermediate result) at user-level where space may grow dynamically. In the unlikely event that aggregation drops are seen, they may be eliminated by increasing the aggregation buffer with the `aggsize` option. (This option may also be useful to minimize the memory footprint of DTrace.) As with any size option, `aggsize` may be specified with any size suffix. As with any buffer option, the resizing policy of this buffer is dictated by the `bufresize` option. More details on buffering can be found in Chapter 11; more details on options can be found in Chapter 16.

An alternative method to eliminate aggregation drops is to increase the rate at which aggregation data is consumed at user-level. (This rate defaults to once per second.) This rate may be explicitly tuned with the `aggrate` option. As with any rate option, `aggrate` may be specified with any time suffix, but defaults to rate-per-second. As with `aggsize`, more details can be found in Chapter 16.

CHAPTER 10

Actions and Subroutines

We've already seen examples of function calls in our D program examples such as `trace()` and `printf()`. D function calls permit you to invoke two different kinds of services provided by DTrace: *actions* that trace data or modify state external to DTrace, and *subroutines* that only affect internal DTrace state. This chapter formally defines the set of actions and subroutines along with their syntax and semantics.

10.1 Actions

In Chapter 9, we introduced the DTrace mechanism for aggregating data from multiple probe firings. While this is useful for many kinds of problems, you may wish for your programs to simply record data instead of aggregating it. Actions are what allow your DTrace programs to interact with the system outside of DTrace. While they commonly act to record data to a DTrace buffer, there are other actions as well — like stopping the current process, raising a specific signal on the current process, or ceasing tracing altogether. Some of these actions are *destructive* in that they change the system — albeit in a well-defined way. These actions may only be used if destructive actions have been explicitly enabled. By default, data recording actions record data to the *principal buffer*. More details on the principal buffer will be forthcoming in Chapter 11. To understand the default behavior of most actions, you need only know that the principal buffer is the fundamental DTrace buffer, present in every DTrace invocation.

10.2 Default Action

A clause need not contain an action — it may instead consist simply of manipulation of variable state, or of any combination of actions and manipulations of variable state. If a clause contains no actions and no D manipulation (that is, if a clause is empty), the *default action* is taken. The default action is to trace the enabled probe identifer (EPID) to the principal buffer. The EPID identifies a particular enabling of a particular probe with a particular predicate and actions. From the EPID, DTrace consumers can determine the probe that induced the action. Indeed, whenever any data is traced, it must be accompanied by the EPID to allow the consumer to make sense of the data — hence the default action is to trace the EPID and nothing else.

Using the default action allows for simple use of `dtrace`(1M). For example, the command:

```
# dtrace -m TS
```

enables all probes in the `TS` module with the default action. (The `TS` module implements the timesharing scheduling class; see `dispadmin`(1M) for more information.) The above command may have output like the following:

```
# dtrace -m TS
dtrace: description 'TS' matched 80 probes
CPU     ID                    FUNCTION:NAME
  0  12077                 ts_trapret:entry
  0  12078                ts_trapret:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12081                  ts_wakeup:entry
  0  12082                 ts_wakeup:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12023                  ts_update:entry
  0  12079             ts_update_list:entry
  0  12080            ts_update_list:return
  0  12079             ts_update_list:entry
...
```

10.3 Data Recording Actions

The data recording actions comprise the core DTrace actions. Each of these actions records data to the principal buffer by default, but each may also record data to speculative buffers. (See Chapter 11 for more details on the principal buffer and Chapter 13 for more details on speculative buffers.) In the below descriptions, we refer only to the *directed buffer*; by default, this is the principal buffer, but may also be a speculative buffer if the action follows a `speculate()`.

10.3.1 `trace()`

`void trace(`*expression*`)`

The most basic action is the `trace()` action, which takes a D expression as its argument and traces the result to the directed buffer. All of the following are valid `trace()` actions:

```
trace(execname);
trace(curlwpsinfo->pr_pri);
trace(timestamp / 1000);
trace(`lbolt);
trace("somehow managed to get here");
```

10.3.2 `tracemem()`

`void tracemem(`*address*`, size_t` *nbytes*`)`

A cousin to `trace()` is the `tracemem()` action, which takes a D expression as its first argument, *address*, and a constant as its second argument, *nbytes*. `tracemem()` copies the memory from the address specified by *addr* into the directed buffer for the length specified by *nbytes*.

10.3.3 `printf()`

`void printf(string` *format*`, ...)`

Like `trace()`, the `printf()` action traces D expressions — but `printf()` allows for elaborate `printf`(3C)-style formatting. Like `printf`(3C), the parameters consists of a *format* string followed by a variable number of arguments. By default, the arguments are traced to the directed buffer. When the arguments are later processed for output by `dtrace`(1M), and later formatted using the specified format string. For example, the first two examples of `trace()` from "10.3.1 `trace()`" on page 123 could be combined in a single `printf()`:

```
printf("execname is %s; priority is %d", execname, curlwpsinfo->pr_pri);
```

A more detailed description of `printf()` is beyond the scope of this chapter; see Chapter 12 for more details.

10.3.4 `printa()`

`void printa(`*aggregation*`)`
`void printa(string` *format*`,` *aggregation*`)`

The `printa()` action allows for displaying and formatting aggregations. (See Chapter 9 for more detail on aggregations.) If a *format* is not provided, `printa()` only traces a directive to the DTrace consumer that the specified aggregation should be processed and displayed using the default format. If a *format* is provided, the aggregation will be formatted as specified; see Chapter 12 for a more detailed description of the `printa()` format string.

Note that `printa()` only traces a *directive* that the aggregation should be processed by the DTrace consumer — it does not process the aggregation in the kernel. This means that the time between the tracing of the `printa()` directive and the actual processing of the directive depends on the factors that affect buffer processing. These factors include the aggregation rate, the buffering policy and — if the buffering policy is `switching` — the rate at which buffers are switched. See Chapter 9 and Chapter 11 for detailed descriptions of these factors.

10.3.5 `stack()`

`void stack(int` *nframes*`)`
`void stack(void)`

The `stack()` action records a kernel stack trace to the directed buffer. The kernel stack will be *nframes* in depth. If *nframes* is not provided, the number of stack frames recorded is the number specifed by the `stackframes` option. For example:

```
# dtrace -n uiomove:entry'{stack()}'
  CPU     ID                    FUNCTION:NAME
    0   9153                    uiomove:entry
                genunix`fop_write+0x1b
                namefs`nm_write+0x1d
                genunix`fop_write+0x1b
                genunix`write+0x1f7

    0   9153                    uiomove:entry
                genunix`fop_read+0x1b
                genunix`read+0x1d4

    0   9153                    uiomove:entry
```

```
              genunix`strread+0x394
              specfs`spec_read+0x65
              genunix`fop_read+0x1b
              genunix`read+0x1d4
  ...
```

The `stack()` action is a little different from other actions in that it may also be used as the key to an aggregation:

```
# dtrace -n kmem_alloc:entry'{@a[stack()] = count()}'
dtrace: description 'kmem_alloc:entry' matched 1 probe
^C

              rpcmod`endpnt_get+0x47c
              rpcmod`clnt_clts_kcallit_addr+0x26f
              rpcmod`clnt_clts_kcallit+0x22
              nfs`rfscall+0x350
              nfs`rfs2call+0x60
              nfs`nfs_getattr_otw+0x9e
              nfs`nfsgetattr+0x26
              nfs`nfs_getattr+0xb8
              genunix`fop_getattr+0x18
              genunix`cstat64+0x30
              genunix`cstatat64+0x4a
              genunix`lstat64+0x1c
                1

              genunix`vfs_rlock_wait+0xc
              genunix`lookuppnvp+0x19d
              genunix`lookuppnat+0xe7
              genunix`lookupnameat+0x87
              genunix`lookupname+0x19
              genunix`chdir+0x18
                1

              rpcmod`endpnt_get+0x6b1
              rpcmod`clnt_clts_kcallit_addr+0x26f
              rpcmod`clnt_clts_kcallit+0x22
              nfs`rfscall+0x350
              nfs`rfs2call+0x60
              nfs`nfs_getattr_otw+0x9e
              nfs`nfsgetattr+0x26
              nfs`nfs_getattr+0xb8
              genunix`fop_getattr+0x18
              genunix`cstat64+0x30
              genunix`cstatat64+0x4a
              genunix`lstat64+0x1c
                1
  ...
```

10.3.6 `ustack()`

```
void ustack(int nframes)
void ustack(void)
```

The `ustack()` action records a *user* stack trace to the directed buffer. The user stack will be *nframes* in depth. If *nframes* is not provided, the number of stack frames recorded is the number specifed by the `ustackframes` option. While `ustack()` is able to determine the address of the calling frames when the probe fires, the stack frames will not be translated into symbols until the `ustack()` action is processed at user-level by the DTrace consumer. Note that some functions are static and therefore do not have entries in the symbol table; call sites in these functions will be displayed with their hexadecimal address. Also, because `ustack()` symbol translation does not occur until *after* the data was recorded, there exists a possibility that the process in question has exited — making stack frame translation impossible. In this case, `dtrace` will emit a warning, followed by the hexadecimal stack frames. For example:

```
  dtrace: failed to grab process 100941: no such process
                c7b834d4
                c7bca85d
                c7bca1a4
                c7bd4374
                c7bc2628
                8047efc
```

More details on this phenomenon — along with techniques for mitigating it — can be found in Chapter 27.

Finally, because the postmortem DTrace debugger commands cannot perform the frame translation, using `ustack()` with a `ring` buffer policy always results in raw `ustack()` data.

The following D program shows an example of `ustack()`:

```
syscall::brk:entry
/execname == $1/
{
    @a[ustack(40)] = count();
}
```

Now run it specifying a process for the Netscape web browser, `.netscape.bin` in default Solaris installations. (The double quotes must be specified to DTrace — the single quoting below is to prevent the shell from stripping the double quotes.)

```
# dtrace -s brk.d '".netscape.bin"'
dtrace: description 'syscall::brk:entry' matched 1 probe
^C
                libc.so.1`_brk_unlocked+0xc
                88143f6
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
```

```
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                   1

                libc.so.1`_brk_unlocked+0xc
                libc.so.1`sbrk+0x29
                88143df
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                  1
...
```

10.4 Destructive Actions

Some actions are destructive in that they change the state of the system. Each of these changes the system in a well-defined way, but they change it nonetheless. Destructive actions may not be used unless they have been explicitly enabled. In `dtrace`(1M), destructive actions are enabled with the `-w` option. If an attempt is made to enable destructive actions in `dtrace`(1M) without explicitly enabling them, `dtrace` will fail with a message similar to:

```
dtrace: failed to enable 'syscall': destructive actions not allowed
```

10.4.1 Process Destructive Actions

Some destructive actions are destructive only to a process — the system itself remains intact. These actions are available to those with the `dtrace_proc` or `dtrace_user` privileges; see Chapter 28 for details.

10.4.1.1 `stop()`

```
void stop(void)
```

The `stop()` action forces the process that hit the enabled probe to stop when it next leaves the kernel, as if stopped by a `proc`(4) action. The `prun`(1) utility may be used to resume a process that has been stopped by the `stop()` action. The `stop()` action can be used to stop a process at any DTrace probe point; this can be used to capture a program in a very particular state (that would be difficult to achieve with a simple breakpoint), and then attach a traditional debugger (like `mdb`(1)) to examine the program's state, or use the `gcore`(1) utility to capture that state in a core file for later analysis.

10.4.1.2 `raise()`

```
void raise(int signal)
```

The `raise()` action sends the specified signal to the currently running process. This is similar to using the `kill`(1) command to send a process a signal, except the `raise()` action can be used to send a signal at a precise point in a process's execution.

10.4.1.3 `copyout()`

```
void copyout(void *buf, uintptr_t addr, size_t nbytes)
```

The `copyout()` action copies *nbytes* from the buffer specified by *buf* to the address specified by *addr* in the address space of the process associated with the current thread. If the user-space address does not correspond to a valid, faulted-in page in the current address space, an error will be generated.

10.4.1.4 `copyoutstr()`

```
void copyoutstr(string str, uintptr_t addr, size_t maxlen)
```

The `copyoutstr()` action copies the string specified by *str* to the address specified by *addr* in the address space of the process associated with the current thread. If the user-space address does not correspond to a valid, faulted-in page in the current address space, an error will be generated. The string length is limited to the value set by the `strsize` option; see Chapter 16 for details.

10.4.2 Kernel Destructive Actions

Some destructive actions are destructive to the entire system. These must obviously be used extremely carefully, as they will affect any process on the system (and any other system implicitly or explicitly depending upon your network services).

10.4.2.1 `breakpoint()`

```
void breakpoint(void)
```

The `breakpoint()` action induces a kernel breakpoint — causing the system to stop and control to transfer to the kernel debugger. The kernel debugger will emit a string denoting the DTrace probe that triggered the action. For example, if one were to do the following:

```
# dtrace -w -n clock:entry'{breakpoint()}'
dtrace: allowing destructive actions
dtrace: description 'clock:entry' matched 1 probe
```

On Solaris running on SPARC, one may see the following on the console:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb 30002765700)
Type  'go' to resume
ok
```

On Solaris running on x86, one may see the following on the console:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb d2b97060)
stopped at      int20+0xb:      ret
kadb[0]:
```

The address following the probe description is the address of the enabling control block (ECB) within DTrace. It may be used to determine more details about the probe enabling that induced the breakpoint action.

Note that a mistake with the `breakpoint()` action may cause it to be called far more often than intended. This may in turn prevent you from even terminating the DTrace consumer that is inducing the breakpoint actions. If you find yourself in this situation, set the kernel integer variable `dtrace_destructive_disallow` to 0. This will disallow *all* destructive actions on the machine — it is recommended that this be used *only* if you find yourself in this particular situation.

The exact method for setting dtrace_destructive_disallow will depend on the kernel debugger that you are using. If using the OpenBoot PROM on SPARC, use w!:

```
ok 1 dtrace_destructive_disallow w!
ok
```

Confirm that this has been set using w?:

```
ok dtrace_destructive_disallow w?
1
ok
```

Continue using go:

```
ok go
```

If using kadb(1M) on x86, use the 4 byte write modifer (W) with the / formatting dcmd:

```
kadb[0]: dtrace_destructive_disallow/W 1
dtrace_destructive_disallow:    0x0             =       0x1
kadb[0]:
```

Continue using :c:

```
kadb[0]: :c
```

If you wish to reenable destructive actions after continuing, you will need to explicitly reset dtrace_destructive_disallow back to 0. This can be done using mdb(1):

```
# echo "dtrace_destructive_disallow/W 0" | mdb -kw
dtrace_destructive_disallow:    0x1             =       0x0
#
```

10.4.2.2 panic()

```
void panic(void)
```

The panic() action induces a kernel panic when triggered. This should be used to force a system crash dump at a time of interest, and may be used together with ring buffering and postmortem analysis to understand a problem. (See Chapter 11 and Chapter 30 respectively.) When the panic action is used, one will see a panic message that denotes the probe inducing the panic. For example:

```
panic[cpu0]/thread=30001830b80: dtrace: panic action at probe
syscall::mmap:entry (ecb 300000acfc8)

000002a10050b840 dtrace:dtrace_probe+518 (fffe, 0, 1830f88, 1830f88,
  30002fb8040, 300000acfc8)
```

```
    %l0-3: 0000000000000000 00000300030e4d80 0000030003418000 00000300018c0800
    %l4-7: 000002a10050b980 0000000000000500 0000000000000000 0000000000000502
  000002a10050ba30 genunix:dtrace_systrace_syscall32+44 (0, 2000, 5,
    80000002, 3, 1898400)
    %l0-3: 00000300030de730 0000000002200008 00000000000000e0 000000000184d928
    %l4-7: 00000300030de000 0000000000000730 0000000000000073 0000000000000010

  syncing file systems... 2 done
  dumping to /dev/dsk/c0t0d0s1, offset 214827008, content: kernel
  100% done: 11837 pages dumped, compression ratio 4.66, dump
  succeeded
  rebooting...
```

`syslogd`(1M) will also emit a message upon reboot:

```
Jun 10 16:56:31 machine1 savecore: [ID 570001 auth.error] reboot after panic:
dtrace: panic action at probe syscall::mmap:entry (ecb 300000acfc8)
```

The message buffer of the crash dump will also contain the probe and ECB responsible for the `panic()` action.

10.4.2.3 `chill()`

`void chill(int` *nanoseconds*`)`

The `chill()` action causes DTrace to spin for the specified number of nanoseconds. `chill()` is primarily useful for exploring problems that may be timing related. For example, it may be used to open race condition windows, or to bring periodic events into or out of phase with one another. Because interrupts are disabled while in DTrace probe context, any `chill()`ing will induce interrupt latency, scheduling latency, dispatch latency, etc. It is therefore possible to cause very strange systemic effects with `chill()`; it should not used indiscriminately. Moreover, because the liveness of the system relies on being able to periodically handle interrupts, DTrace will refuse to `chill()` for a total of longer than 500 milliseconds and instead report an illegal operation error:

```
# dtrace -w -n syscall::open:entry'{chill(500000001)}'
dtrace: allowing destructive actions
dtrace: description 'syscall::open:entry' matched 1 probe
dtrace: 57 errors
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 14: syscall::open:entry): \
  illegal operation in action #1
```

Note that this cap is enforced even if the time is spread across multiple calls to `chill()()`. That is, the same error would be generated with:

```
# dtrace -w -n syscall::open:entry'{chill(250000000); chill(250000001);}'
```

The cap is also enforced if the time is spread across multiple DTrace consumers for a single probe.

10.5 Special Actions

Some actions don't fall neatly into either the data recording action or destructive action categories. This section describes these other special actions.

10.5.1 Speculative Actions

There are a number of actions associated with speculative tracing: `speculate()`, `commit()` and `discard()`. The discussion of these actions is outside the scope of this chapter; see Chapter 13.

10.5.2 `exit()`

`void exit(int` *`status`*`)`

The `exit()` action is used to immediately stop tracing, and to inform the DTrace consumer that it should cease tracing, perform any final processing, and call `exit`(3C) with the status specified. Because `exit()` *does* return a status to user-level, it is a data storing action —— but unlike other data storing actions, it cannot be speculatively traced. `exit()` will cause the DTrace consumer to exit regardless of buffer policy. Note that the data storing nature of `exit()` means that it *can* be dropped.

When `exit()` is called, only DTrace actions already underway on other CPUs will be taken; no subsequent actions will be taken on any CPU. The only exception to this is the `END` probe, which will be called after the DTrace consumer has processed the `exit()` action and indicated that tracing should stop.

10.6 Subroutines

Subroutines differ from actions in that they generally only affect internal DTrace state. There is therefore no such thing as a destructive subroutine, and subroutines never trace data into buffers. Many subroutines have analogs in Section 9F or Section 3C; see `Intro`(9F) and `Intro`(3), respectively.

10.6.1 alloca()

```
void *alloca(size_t size)
```

`alloca()` allocates *size* bytes out of scratch space, and returns a pointer to the allocated memory. The returned pointer is guaranteed to have 8–byte alignment. Scratch space is only valid for the duration of a clause; memory allocated with `alloca()` will be deallocated when the clause completes. If insufficient scratch space is available, `alloca()` no memory is allocated and an error is generated.

10.6.2 bcopy()

```
void bcopy(void *src, void *dest, size_t size)
```

`bcopy()` copies *size* bytes from the memory pointed to by *src* to the memory pointed to by *dest*. All of the source memory must lie outside of scratch memory and all of the destination memory must lie within it; if this is not the case, no copying takes place and an error is generated.

10.6.3 copyin()

```
void *copyin(uintptr_t addr, size_t size)
```

`copyin()` copies the specified size in bytes from the specified user address into a DTrace scratch buffer, and returns the address of this buffer. The user address is interpreted as an address in the space of the process associated with the current thread. The resulting buffer pointer is guaranteed to have 8-byte alignment. The address in question *must* correspond to a faulted-in page in the current process. If the address does not correspond to a faulted-in page, or if insufficient scratch space is available, `NULL` is returned, and an error is generated. See Chapter 27 for techniques to reduce the likelihood of `copyin` errors.

10.6.4 copyinstr()

```
string copyinstr(uintptr_t addr)
```

`copyinstr()` copies a null-terminated C string from the specified user address into a DTrace scratch buffer, and returns the address of this buffer. The user address is interpreted as an address in the space of the process associated with the current thread. The string length is limited to the value set by the `strsize` option; see Chapter 16 for details. As with `copyin`, the specified address *must* correspond to a faulted-in page in the current process. If the address does not correspond to a faulted-in page, or if insufficient scratch space is available, `NULL` is returned, and an error is generated. See Chapter 27 for techniques to reduce the likelihood of `copyinstr` errors.

10.6.5 copyinto()

```
void copyinto(uintptr_t addr, size_t size, void *dest)
```

`copyinto()` copies the specified size in bytes from the specified user address into the DTrace scratch buffer specified by *dest*. The user address is interpreted as an address in the space of the process associated with the current thread. The address in question *must* correspond to a faulted-in page in the current process. If the address does not correspond to a faulted-in page, or if any of the destination memory lies outside scratch space, no copying takes place, and an error is generated. See Chapter 27 for techniques to reduce the likelihood of `copyinto` errors.

10.6.6 msgdsize()

```
size_t msgdsize(mblk_t *mp)
```

`msgdsize()` returns the number of bytes in the data message pointed to by *mp*. See `msgdsize`(9F) for details. Note that `msgdsize()` only includes data blocks of type `M_DATA` in the count.

10.6.7 msgsize()

```
size_t msgsize(mblk_t *mp)
```

`msgsize()` returns the number of bytes in the message pointed to by *mp*. Unlike `msgdsize()` — which returns only the number of *data* bytes — `msgsize()` returns the *total* number of bytes in the message.

10.6.8 mutex_owned()

```
int mutex_owned(kmutex_t *mutex)
```

`mutex_owned()` is an implementation of `mutex_owned`(9F). `mutex_owned()` returns non-zero if the calling thread currently holds the specified kernel mutex, or zero if the specified adaptive mutex is currently unowned.

10.6.9 mutex_owner()

```
kthread_t *mutex_owner(kmutex_t *mutex)
```

`mutex_owner()` returns the thread pointer of the current owner of the specified adaptive kernel mutex. `mutex_owner()` returns `NULL` if the specified adaptive mutex is currently unowned, or if the specified mutex is a spin mutex. See `mutex_owned`(9F).

10.6.10 mutex_type_adaptive()

```
int mutex_type_adaptive(kmutex_t *mutex)
```

`mutex_type_adaptive()` returns non-zero if the specified kernel mutex is of type `MUTEX_ADAPTIVE`, or zero if it is not. Mutexes are adaptive if they are:

- declared statically,
- created with an interrupt block cookie of NULL, or
- created with an interrupt block cookie that doesn't correspond to a high-level interrupt

See `mutex_init`(9F) for more details on mutexes. The vast majority of mutexes in the Solaris kernel are adaptive.

10.6.11 progenyof()

```
int progenyof(pid_t pid)
```

`progenyof()` returns non-zero if the calling process (the process associated with the thread that is currently triggering the matched probe) is among the progeny of the specified process ID.

10.6.12 rand()

```
int rand(void)
```

`rand()` returns a pseudo-random integer. The number returned is a weak pseudo-random number, and should not be used for any cryptographic application.

10.6.13 rw_iswriter()

```
int rw_iswriter(krwlock_t *rwlock)
```

`rw_iswriter()` returns non-zero if the specified reader-writer lock is either held or desired by a writer. If the lock is neither held nor desired by any writers (that is, it is held only by readers and no writer is blocked, or it is not held at all), `rw_iswriter()` returns zero. See `rw_init`(9F).

10.6.14 rw_read_held()

```
int rw_write_held(krwlock_t *rwlock)
```

`rw_write_held()` returns non-zero if the specified reader-writer lock is currently held by a writer. If the lock is held only by readers or not held at all, `rw_write_held()` returns zero. See `rw_init`(9F).

10.6.15 `speculation()`

```
int speculation(void)
```

`speculation()` reserves a speculative trace buffer for use with `speculate()` and returns an identifier for this buffer. See Chapter 13 for details.

10.6.16 `strlen()`

```
size_t strlen(string str)
```

`strlen()` returns the length of the specified string in bytes, excluding the terminating null byte.

CHAPTER 11

Buffers and Buffering

Data buffering and management is an essential service provided by the DTrace framework for its clients. In the preceding chapters, we have used DTrace without discussing the details of how data that is traced is transported from the DTrace framework to clients such as `dtrace`(1M). In this chapter, we explore data buffering in detail and learn about options you can tune to change DTrace's buffer management policies.

11.1 Principal Buffers

The buffer most fundamental to DTrace operation is the *principal buffer*. The principal buffer is present in every DTrace invocation and is the buffer to which tracing actions record their data by default. These actions include:

`exit()`	`printf()`	`trace()`	`ustack()`
`printa()`	`stack()`	`tracemem()`	

The principal buffers are *always* allocated on a per-CPU basis. This is not tunable, though tracing (and thus buffer allocation) may be restricted to a single CPU by using the `cpu` option.

11.2 Principal Buffer Policies

DTrace allows for tracing in highly constrained contexts in the kernel. In particular, DTrace allows for tracing in contexts in which one may not reliably allocate memory. The consequence of this flexibility of context is that there *always* exists a possibility that one will wish to trace data when there isn't space available. Given this, DTrace must have a policy to deal with such situations when they arise — but the policy that one desires will be dictated by the specifics of how DTrace is being used: sometimes it may be best to discard the new data, while other times it may be desirable to reuse the space containing the oldest recorded data to trace the new data. Most often, however, the desired policy is the one that simply minimizes the likelihood of running out of available space in the first place. To accommodate these varying demands, DTrace supports several different buffer policies. This support is implemented with the `bufpolicy` option, and can be set on a per-consumer basis. (See Chapter 16 for more details on setting options.)

11.2.1 `switch` Policy

By default, the principal buffer has a `switch` buffer policy. Under this policy, per-CPU buffers are allocated in pairs: one buffer is active, the other is inactive. When a DTrace consumer asks to read its buffer out of the kernel, the kernel firsts *switches* the inactive and active buffers. Buffer switching is done in such a manner that there is no window in which tracing data may be lost. Once the buffers are switched, the newly inactive buffer is copied out to the DTrace consumer. This policy assures that the consumer always sees a self-consistent buffer (that is, a buffer is never simultaneously traced to and copied out), while not introducing a window in which tracing is paused or otherwise prevented. The rate at which the buffer is read out (and thus switched) is controlled by the consumer with the `switchrate` option. As with any rate option, `switchrate` may be specified with any time suffix, but defaults to rate-per-second. More details on `switchrate` and other options may be found in Chapter 16.

Under the `switch` policy, if a given enabled probe would trace more data than there is space available in the active principal buffer, the data is *dropped* and a per-CPU drop count is incremented. In the event of one or more drops, one will see this message or similar from `dtrace`(1M):

```
dtrace: 11 drops on CPU 0
```

Drops may be reduced or eliminated by either increasing the size of the principal buffer with the `bufsize` option or by increasing the switching rate with the `switchrate` option.

Under the `switch` policy, scratch space for `copyin()`, `copyinstr()`, and `alloca()` is allocated out of the active buffer.

11.2.2 `fill` Policy

For some problems, it may be desirable to have a single in-kernel buffer. While this can be implemented with a `switch` policy and appropriate D constructs (for example, incrementing a variable in D and predicating an `exit()` action appropriately), one may wish to avoid any D overhead and/or completely eliminate the possibility of drops. In such situations, it may be desirable to have a single, large in-kernel buffer, and continue tracing until one or more of the per-CPU buffers has filled. DTrace implements this with the `fill` buffer policy. Under this policy, tracing continues until an enabled probe would trace more data than there is space in the principal buffer. At this time, the buffer is marked as filled and the consumer is notified that at least one of its per-CPU buffers has filled. Once `dtrace`(1M) detects a single filled buffer, tracing is stopped, all buffers are processed and `dtrace` exits. Note that no further data will be traced to a filled buffer — even if the data would fit in the buffer.

To use the `fill` policy, set the `bufpolicy` option to `fill`. For example, this invocation of DTrace traces every system call entry into a per-CPU 2K buffer with the buffer policy set to `fill`:

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

11.2.2.1 `fill` Policy and `END` Probes

`END` probes normally do not fire until tracing has been explicitly stopped by the DTrace consumer. `END` probes are guaranteed to only fire on one CPU, but the CPU on which the probe fires is undefined. With `fill` buffers, tracing is explicitly stopped when at least one of the per-CPU principal buffers has been marked as filled. As described, `END` probes for `fill` buffers would be problematic — the `END` probe may attempt to fire on a CPU that has a filled buffer. To allow for `END` tracing in `fill` buffers, DTrace will a priori calculate the amount of space potentially consumed by `END` probes and *subtract* this from the size of the principal buffer. If the net size is negative, DTrace will refuse to start, and `dtrace`(1M) will output a corresponding error message:

```
dtrace: END enablings exceed size of principal buffer
```

Reserving space a priori assures that a full buffer always has sufficient space for any and all `END` probes.

11.2.3 `ring` Policy

When using DTrace to help diagnose failure (as opposed to understanding non-failing behavior), one often wishes to track the events leading up to failure. Moreover, in cases where reproducing failure can take hours or days, one may wish to only keep the most recent data. To be applicable to such problems, DTrace provides a `ring`

buffer policy. Under this policy, once a principal buffer has filled, tracing wraps around to the first entry, thereby overwriting older tracing data. A ring buffer is established by setting the `bufpolicy` option to the string `ring`:

```
# dtrace -s foo.d -x bufpolicy=ring
```

When used to create a ring buffer, `dtrace`(1M) will not display any output until the process is terminated — at which time the ring buffer will be consumed and processed. When processing a ring buffer, `dtrace` will process each buffer in CPU order, and within a CPU, records will be displayed in order from oldest to youngest. Just as with the `switch` buffering policy, there is no ordering between records on different CPUs per se — if an ordering is required, `timestamp` should be traced.

Note that if a given record cannot fit at *all* in the buffer (that is, if the record is larger than the buffer size), the record will be dropped regardless of buffer policy. The following example program demonstrates the use of a `#pragma option` directive to enable ring buffering:

```
#pragma D option bufpolicy=ring
#pragma D option bufsize=16k

syscall:::entry
/execname == $1/
{
    trace(timestamp);
}

syscall::rexit:entry
{
    exit(0);
}
```

11.3 Other Buffers

Principal buffers exist in every DTrace enabling. Beyond principal buffers, some DTrace consumers may have additional in-kernel data buffers: an *aggregation buffer* and/or some number of *speculative buffers*. A full discussion of the use of these buffers is beyond the scope of this chapter; details can be found in Chapter 9 and Chapter 13, respectively.

11.4 Buffer Sizes

The size of each buffer can be tuned on a per-consumer basis. Separate options are provided to tune each buffer size:

Buffer	Size Option
Principal	`bufsize`
Speculative	`specsize`
Aggregation	`aggsize`

Each of these options is set with a value that denotes the size. As with any size option, the value may have an optional size suffix; see Chapter 16 for more details. For example, to set the buffer size to one megabyte on the command line to `dtrace`, one could use `-x` to set the option:

```
# dtrace -P syscall -x bufsize=1m
```

One may alternatively use the `-b` option to `dtrace`:

```
# dtrace -P syscall -b 1m
```

And as with any option, in a D script, one may set `bufresize` using `#pragma D option`:

```
#pragma D option bufsize=1m
```

Note that setting the buffer size denotes the size of the buffer on *each* CPU. Moreover, for the `switch` buffer policy, `bufsize` denotes the size of *each* buffer on each CPU. The buffer size defaults to four megabytes.

11.5 Buffer Resizing Policy

One may find that there is not adequate free kernel memory to allocate a buffer of desired size. This may be because there is simply not enough memory available, or it may be because the DTrace consumer has exceeded a tunable limit (see Chapter 16 for details). DTrace allows for a configurable policy when a buffer cannot be allocated.

The policy is set with the `bufresize` option, and defaults to `auto`. Under the `auto` buffer resize policy, the size of a buffer is halved until a successful allocation occurs. `dtrace`(1M) will emit a message if a buffer as allocated is smaller than the requested size:

```
# dtrace -P syscall -b 4g
dtrace: description 'syscall' matched 430 probes
dtrace: buffer size lowered to 128m
...
```

or:

```
# dtrace -P syscall'{@a[probefunc] = count()}' -x aggsize=1g
dtrace: description 'syscall' matched 430 probes
dtrace: aggregation size lowered to 128m
...
```

Alternatively, one may set the buffer resize policy to be manual by setting `bufresize` to `manual`. Under this policy, a failure to allocate will cause DTrace to fail to start:

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual
dtrace: description 'syscall' matched 430 probes
dtrace: could not enable tracing: Not enough space
#
```

Note that the buffer resizing policy of *all* buffers — principal, speculative and aggregation — is dictated by the `bufresize` option.

CHAPTER 12

Output Formatting

DTrace provides built-in formatting functions `printf()` and `printa()` that you can use from your D programs to format output. The D compiler provides features not found in the `printf`(3C) library routine, so you should read this chapter even if you are already familiar with `printf()`. This chapter also provides information about the formatting behavior of the `trace()` function and the default output format used by `dtrace`(1M) to display aggregations.

12.1 `printf()`

The `printf()` function combines the ability to trace data, as if by the `trace()` function, with the ability to output the data and other text in a specific format that you describe. The `printf()` function tells DTrace to trace the data associated with each argument after the first argument, and then to format the results using the rules described by the first `printf()` argument, known as a *format string*.

The format string is a regular string that contains any number of format conversions, each beginning with the `%` character, that describe how to format the corresponding argument. The first conversion in the format string corresponds to the second `printf()` argument, the second conversion to the third argument, and so on. All of the text between conversions is printed verbatim. The character following the `%` conversion character describes the format to use for the corresponding argument.

Unlike `printf`(3C), DTrace `printf()` is implemented as a built-in function that is recognized by the D compiler. The D compiler provides several useful services for DTrace `printf()` that are not found in the C library `printf()`:

- The D compiler compares the arguments to the conversions in the format string. If an argument's type is incompatible with the format conversion, the D compiler will produce a helpful error message explaining the problem.

- The D compiler does not require the use of size prefixes with `printf()` format conversions. The C `printf()` routine requires that you indicate the size of arguments by adding prefixes such as `%ld` for `long` or `%lld` for `long long`. The D compiler knows the size and type of your arguments, so these prefixes are not required in your D `printf()` statements.
- DTrace provides additional format characters that are useful for debugging and observability; for example, the `%a` format conversion can be used to print a pointer as a symbol name and offset.

In order to implement these features, the format string in the DTrace `printf()` function must be specified as a string constant in your D program; format strings may not be dynamic variables of type `string`.

12.1.1 Conversion Specifications

Each conversion specification in the format string is introduced by the `%` character, after which the following appear in sequence:

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification as described below.
- An optional minimum *field width*. If the converted value has fewer bytes than the field width, it will be padded with spaces on the left by default, or on the right if the left-adjustment flag (`-`) is specified. The field width can also be specified as an asterisk (`*`), in which case the field width is set dynamically based on the value of an additional argument of type `int`.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions (the field is padded with leading zeroes); the number of digits to appear after the radix character for the `e`, `E`, and `f` conversions, the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of bytes to be printed from a string by the `s` conversion. The precision takes the form of a period (`.`) followed by either an asterisk (`*`), described below, or a decimal digit string.
- An optional sequence of *size prefixes* that indicate the size of the corresponding argument, described below. The size prefixes are not necessary in D and are provided solely for compatibility with the C `printf()` function.
- A *conversion specifier*, described below, that indicates the type of conversion to be applied to the argument.

The `printf`(3C) function also supports conversion specifications of the form `%`*n*`$` where *n* is a decimal integer; DTrace `printf()` does not support this type of conversion specification.

12.1.2 Flag Specifiers

The `printf()` conversion flags are enabled by specifying one or more of the following characters, which may appear in any order:

`'`	The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%g`, or `%G`) will be formatted with thousands grouping characters using the non-monetary grouping character. Not all locales, including the POSIX C locale, provide non-monetary grouping characters for use with this flag.
`-`	The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.
`+`	The result of signed conversion will always begin with a sign (`+` or `-`). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
`space`	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space will be placed before the result. If the `space` and `+` flags both appear, the space flag is ignored.
`#`	The value is converted to an alternate form if one is defined for the selected conversion. The alternate formats for conversions are described below in the text corresponding to each conversion.
`0`	For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversions, leading zeroes (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. If the `0` and `'` flags both appear, the grouping characters are inserted before the zero padding.

12.1.3 Width and Precision Specifiers

The minimum field width can be specified as a decimal digit string following any flag specifier, as described above, in which case the field width will be set to the specified number of columns. The field width can also be specified as asterisk (`*`) in which case an additional argument of type `int` is accessed to determine the field width. For example, to print an integer `x` in a field width determined by the value of the `int` variable `w`, you would write the D statement:

```
printf("%*d", w, x);
```

The field width can also be specified using a `?` character to indicate that the field width should be set based on the number of characters required to format an address in hexadecimal in the data model of the operating system kernel. The width is set to 8 if the kernel is using the 32–bit data model, or to 16 if the kernel is using the 64–bit data model.

The precision for the conversion can be specified as a decimal digit string following a period (`.`) or by an asterisk (`*`) following a period. If an asterisk is used to specify the precision, an additional argument of type `int` prior to the conversion argument is accessed to determine the precision. If both width and precision are specified as asterisks, the order of arguments to `printf()` for the conversion should appear in the order: width, precision, value.

12.1.4 Size Prefixes

Size prefixes are required in ANSI-C programs that use `printf`(3C) in order to indicate the size and type of the conversion argument. The D compiler performs this processing for your `printf()` calls automatically, so size prefixes are not required. Although size prefixes are provided for C compatibility, their use is explicitly discouraged in D programs because they also tend to bind your code to a particular data model when using derived types. For example, if a `typedef` is redefined to different integer base types depending on the data model, it is not possible to use a single C conversion that works in both data models without explicitly knowing the two underlying types and including a cast expression, or defining multiple format strings. The D compiler solves this problem automatically by allowing you to omit size prefixes and automatically determining the argument size.

The size prefixes can be placed just prior to the format conversion name and after any flags, widths, and precision specifiers. The size prefixes are:

- An optional `h` specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a `short` or `unsigned short`.
- An optional `l` specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a `long` or `unsigned long`.
- An optional `ll` specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a `long long` or `unsigned long long`.
- An optional `L` specifies that a following `e`, `E`, `f`, `g`, or `G` conversion applies to a `long double`.
- An optional `l` specifies that a following `c` conversion applies to a `wint_t` argument; an optional `l` specifies that a following `s` conversion character applies to a pointer to `awchar_t` argument.

12.1.5 Conversion Formats

Each conversion character sequence results in fetching zero or more arguments. If insufficient arguments are provided for the format string, or if the format string is exhausted and arguments remain, the D compiler will issue an appropriate error message. If an undefined conversion format is specified, the D compiler will issue an appropriate error message. The conversion character sequences and their meanings are:

`a` The pointer or `uintptr_t` argument is printed as a kernel symbol name in the form *module*`` `*symbol-name* `` plus an optional hexadecimal byte offset. If the value does not fall within the range defined by a known kernel symbol, the value is printed as a hexadecimal integer.

`c` The `char`, `short`, or `int` argument is printed as an ASCII character.

`d` The `char`, `short`, `int`, `long`, or `long long` argument is printed as a decimal (base 10) integer. If the argument is `signed`, it will be printed as a signed value. If the argument is `unsigned`, it will be printed as an unsigned value. This conversion has the same meaning as `i`.

`e, E` The `float`, `double`, or `long double` argument is converted to the style `[-]`*d*`.`*ddd*`e±`*dd*, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision. If the precision is not specified, the default precision value is 6. If the precision is 0 and the # flag is not specified, no radix character appears. The `E` conversion format will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. The value is rounded up to the appropriate number of digits.

`f` The `float`, `double`, or `long double` argument is converted to the style `[-]`*ddd*`.`*ddd*, where the number of digits after the radix character is equal to the precision specification. If the precision is not specifed, the default precision value is 6. If the precision is 0 and the # flag is not specified, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded up to the appropriate number of digits.

`g, G` The `float`, `double`, or `long double` argument is printed in the style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted: style `e` (or `E`) will be used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeroes are removed from the fractional part of the result. A radix character appears only if it is followed by a digit. If the # flag is specified, trailing zeroes will not be removed from the result as they normally are.

`i` The `char`, `short`, `int`, `long`, or `long long` argument is printed as a decimal (base 10) integer. If the argument is `signed`, it will be printed as a

signed value. If the argument is `unsigned`, it will be printed as an unsigned value. This conversion has the same meaning as `d`.

`o` The `char`, `short`, `int`, `long`, or `long long` argument is printed as an unsigned octal (base 8) integer. Arguments that are `signed` or `unsigned` may be used with this conversion. If the # flag is specified, the precision of the result will be increased if necessary to force the first digit of the result to be a zero.

`p` The pointer or `uintptr_t` argument is printed as a hexadecimal (base 16) integer. D accepts pointer arguments of any type. If the # flag is specified, a non-zero result will have `0x` prepended to it.

`s` The argument must be an array of `char` or a `string`. Bytes from the array or `string` are read up to a terminating null character or the end of the data and interpreted and printed as ASCII characters. If the precision is not specified, it is taken to be infinite, so all characters up to the first null character are printed. If the precision is specified, only that portion of the character array that will display in the corresponding number of screen columns will be printed. If an argument of type `char *` is to be formatted, it should be cast to `string` or prefixed with the D `stringof` operator to indicate that DTrace should trace the bytes of the string and format them.

`u` The `char`, `short`, `int`, `long`, or `long long` argument is printed as an unsigned decimal (base 10) integer. Arguments that are `signed` or `unsigned` may be used with this conversion, and the result is always formatted as `unsigned`.

`wc` The `int` argument is converted to a wide character (`wchar_t`) and the resulting wide character is printed.

`ws` The argument must be an array of `wchar_t`. Bytes from the array are read up to a terminating null character or the end of the data and interpreted and printed as wide characters. If the precision is not specified, it is taken to be infinite, so all wide characters up to the first null character are printed. If the precision is specified, only that portion of the wide character array that will display in the corresponding number of screen columns will be printed.

`x, X` The `char`, `short`, `int`, `long`, or `long long` argument is printed as an unsigned hexadecimal (base 16) integer. Arguments that are `signed` or `unsigned` may be used with this conversion. If the `x` form of the conversion is used, the letter digits `abcdef` are used. If the `X` form of the conversion is used, the letter digits `ABCDEF` are used. If the # flag is specified, a non-zero result will have `0x` (for `%x`) or `0X` (for `%X`) prepended to it.

`%` Print a literal `%` character; no argument is converted. The entire conversion specification must be `%%`.

12.2 printa()

The printa() function is used to format the results of aggregations in a D program. The function is invoked using one of two forms:

```
printa(@aggregation-name);
printa(format-string, @aggregation-name);
```

If the first form of the function is used, the dtrace(1M) command takes a consistent snapshot of the aggregation data and produces output equivalent to the default output format used for aggregations, described in Chapter 9.

If the second form of the function is used, the dtrace(1M) command takes a consistent snapshot of the aggregation data and produces output according to the conversions specified in the *format string*, according to the following rules:

- The format conversions must match the tuple signature used to create the aggregation. Each tuple element may only appear once. For example, if you aggregate a count using the following D statements:

  ```
  @a["hello", 123] = count();
  @a["goodbye", 456] = count();
  ```

  and then add the D statement printa(*format-string*, @a) to a probe clause, dtrace will snapshot the aggregation data and produce output as if you had entered the statements:

  ```
  printf(format-string, "hello", 123);
  printf(format-string, "goodbye", 456);
  ```

  and so on for each tuple defined in the aggregation.
- Unlike printf(), the format string you use for printa() need not include all elements of the tuple (that is, you can have a tuple of length 3 and only one format conversion). Therefore you can omit any tuple keys from your printa() output by changing your aggregation declaration to move the ones you wish to omit to the end of the tuple and then omitting corresponding conversion specifiers for them from the printa() format string.
- The aggregation result itself can be included in the output by using the additional @ format flag character, which is only valid when used with printa(). The @ flag can be combined with any appropriate format conversion specifier, and may appear more than once in a format string so that your tuple result can appear anywhere in the output and can appear more than once. The set of conversion specifiers that can be used with each aggregating function are implied by the aggregating function's result type, listed below:

avg()	uint64_t
count()	uint64_t
lquantize()	int64_t
max()	uint64_t
min()	uint64_t
quantize()	int64_t
sum()	uint64_t

For example, to format the results of `avg()`, you can apply the `%d`, `%i`, `%o`, `%u`, or `%x` format conversions. The `quantize()` and `lquantize()` functions format their results as an ASCII table rather than as a single value.

The following D program shows a complete example of `printa()`, using the `profile` provider to sample the value of `caller` and then formatting the results as a simple table:

```
profile:::profile-997
{
    @a[caller] = count();
}

END
{
    printa("%@8u %a\n", @a);
}
```

If we use `dtrace` to execute this program and wait a few seconds and type Control-C, we will see output similar to the following:

```
# dtrace -s printa.d
^C
CPU     ID                    FUNCTION:NAME
  1      2                             :END        1 0x1
       1 ohci`ohci_handle_root_hub_status_change+0x148
       1 specfs`spec_write+0xe0
       1 0xff14f950
       1 genunix`cyclic_softint+0x588
       1 0xfef2280c
       1 genunix`getf+0xdc
       1 ufs`ufs_icheck+0x50
       1 genunix`infpollinfo+0x80
       1 genunix`kmem_log_enter+0x1e8
       ...
```

12.3 `trace()` Default Format

If the `trace()` function is used to capture data rather than `printf()`, the `dtrace` command will format the results using a default output format. If the data is 1, 2, 4, or 8 bytes in size, the result will be formatted as a decimal integer value. If the data is any other size and is a sequence of printable characters if interpreted as sequence of bytes, it will be printed as an ASCII string. If the data is any other size and is not a sequence of printable characters, it will be printed as a series of byte values formatted as hexadecimal integers.

CHAPTER 13

Speculative Tracing

In a tracing framework that offers coverage as comprehensive as that of DTrace, the challenge for the user quickly becomes figuring out what *not* to trace. In DTrace, the primary mechanism for filtering out uninteresting events is the *predicate* mechanism as discussed in Chapter 4. Predicates are useful when you know at the time that a probe fires whether or not the probe event is interesting. For example, if you are only interested in activity associated with a certain process or a certain file descriptor, you know when the probe fires if it is associated with the process or file decriptor of interest. However, there are some situations for which one may not know whether or not a given probe event is interesting at the time that the probe fires. It may only be some time *after* the probe fires that one can make the determination that the probe event is interesting.

For example, if a system call is failing some amount of the time with a common error code (for example, `EIO` or `EINVAL`), one may wish to better understand the code path that is leading to the error condition. To capture the code path, one could enable every probe — but only if the failing call can be isolated in such a way that a meaningful predicate can be constructed. If the failures were sporadic or nondeterministic, one would be forced to trace all events that *may* be interesting, and later postprocess the data to filter out the ones that were not associated with the failing code path. In this case, even though the number of interesting events may be reasonably small, the number of events that must be traced is very large — making postprocessing difficult if not impossible.

To address this and similar situations, DTrace has a facility for *speculative tracing*. Using this facility, one may tentatively trace data; later, one may decide that the traced data is interesting and *commit* it to the principal buffer, or one may decide that the traced data is uninteresting, and *discard* it.

13.1 Speculation Interfaces

The following D functions comprise the DTrace speculation facility:

TABLE 13–1 DTrace Speculation Functions

Function Name	Args	Description
`speculation`	None	Returns an identifier for a new speculative buffer.
`speculate`	ID	Denotes that the remainder of the clause should be traced to the speculative buffer specified by ID.
`commit`	ID	Commits the speculative buffer associated with ID.
`discard`	ID	Discards the speculative buffer associated with ID.

13.2 Creating a Speculation

The `speculation()` function allocates a speculative buffer, and returns a speculation identifier. The speculation identifier should be used in subsequent calls to the `speculate()` function. Speculative buffers are a finite resource; if no speculative buffer is available when `speculation()` is called, an ID of zero is returned (and a corresponding DTrace error counter is incremented). An ID of zero is always invalid, but may be passed to `speculate()`, `commit()` or `discard()` without ill effect. If a call to `speculation()` fails, one will see a `dtrace` message like the following:

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

The number of speculative buffers defaults to one, but may be optionally tuned higher; see Options and Tuning, below.

13.3 Using a Speculation

To use a speculation, an identifer returned from `speculation()` must be passed to the `speculate()` function in a clause *before* any data-recording actions. *All* subsequent data-recording actions in a clause containing a `speculate()` will be speculatively traced. As such, `speculate()` may not follow data recording actions;

any attempt to do so will yield a compile-time error. As this implies, clauses may contain speculative tracing or non-speculative tracing — but not both. Aggregating actions, destructive actions, and the `exit` action may never be speculative; any attempt to take one of these actions in a clause containing a `speculate()` will result in a compile-time error. Moreover, a `speculate()` may not follow a `speculate()` — only one speculation is permitted per clause. Note that a clause that contains *only* a `speculate()` will speculatively trace the default action — which is defined to be the enabled probe ID. (See Chapter 10 for a description of the default action.)

Typically, one will assign the result of `speculation()` to a variable (often thread-local), and use that variable as a subsequent predicate to other probes as well as an argument to `speculate()`. For example:

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

13.4 Committing a Speculation

Speculations are committed using the `commit()` function. When a speculative buffer is committed, its data is copied into the principal buffer. If there is more data in the specified speculative buffer than there is available space in the principal buffer, no data will be copied and the drop count for the buffer will be increased. If the speculation is active on more than one CPU (that is, if the buffer has been speculatively traced to on more than one CPU), the speculative data on the committing CPU will be copied immediately, while speculative data on other CPUs will be copied some time after the `commit()`. Thus, some time may elapse between a `commit()` beginning on one CPU and the data being copied from speculative buffers to principal buffers on all CPUs. This time is guaranteed to be no longer than the time dictated by the cleaning rate; see “13.7 Options and Tuning” on page 160 for more details.

A committing speculative buffer will not be made available to subsequent `speculation()` calls until each per-CPU speculative buffer has been completely copied into its corresponding per-CPU principal buffer. Similarly, subsequent calls to `speculate()` to the committing buffer will be silently discarded, and subsequent calls to `commit()` or `discard()` it will silently fail. Finally, there may be no other data recording action in a clause containing a `commit()`, but a clause may contain multiple `commit()` calls to commit disjoint buffers.

13.5 Discarding a Speculation

Speculations are discarded using the `discard()` function. When a speculative buffer is discarded, its contents are thrown away. If the speculation has only been active on the CPU calling `discard()`, the buffer is immediately available for subsequent calls to `speculation()`. If the speculation has been active on more than one CPU, the discarded buffer will be available for subsequent `speculation()` some time after the call to `discard()`. The time between a `discard()` on one CPU and the buffer being made available for subsequent speculations is guaranteed to be no longer than the time dictated by the cleaning rate. If, at the time `speculation()` is called, no buffer is available because *all* speculative buffers are currently being discarded or committed, one will see a `dtrace` message like:

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

The likelihood of this can be reduced by tuning the number of speculation buffers and/or the cleaning rate; see "13.7 Options and Tuning" on page 160, for details.

13.6 Speculation Example

One potential use for speculations is to highlight a particular code path. For example, here is a D script to show the entire codepath under the `open`(2) system call — but only when the `open()` fails:

EXAMPLE 13–1 specopen.d: Code Flow for Failed `open`(2)

```
#!/usr/sbin/dtrace -Fs

syscall::open:entry,
syscall::open64:entry
{
    /*
     * The call to speculation() creates a new speculation.  If this fails,
     * dtrace(1M) will generate an error message indicating the reason for
     * the failed speculation(), but subsequent speculative tracing will be
     * silently discarded.
     */
    self->spec = speculation();
    speculate(self->spec);

    /*
     * Because this printf() follows the speculate(), it is being
     * speculatively traced; it will only appear in the data buffer if the
```

EXAMPLE 13–1 specopen.d: Code Flow for Failed open(2) *(Continued)*

```
     * speculation is subsequently commited.
     */
    printf("%s", stringof(copyinstr(arg0)));
}

fbt:::
/self->spec/
{
    /*
     * A speculate() with no other actions speculates the default action:
     * tracing the EPID.
     */
    speculate(self->spec);
}

syscall::open:return,
syscall::open64:return
/self->spec/
{
    /*
     * To balance the output with the -F option, we want to be sure that
     * every entry has a matching return.  Because we speculated the
     * open entry above, we want to also speculate the open return.
     * This is also a convenient time to trace the errno value.
     */
    speculate(self->spec);
    trace(errno);
}

syscall::open:return,
syscall::open64:return
/self->spec && errno != 0/
{
    /*
     * If errno is non-zero, we want to commit the speculation.
     */
    commit(self->spec);
    self->spec = 0;
}

syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
     */
    discard(self->spec);
    self->spec = 0;
}
```

Running this yields, for example:

```
# ./specopen.d
dtrace: script './specopen.d' matched 24282 probes
CPU FUNCTION
  1  => open                                    /var/ld/ld.config
  1    -> open
  1      -> copen
  1        -> falloc
  1          -> ufalloc
  1            -> fd_find
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_find
  1            -> fd_reserve
  1              -> mutex_owned
  1              <- mutex_owned
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_reserve
  1          <- ufalloc
  1          -> kmem_cache_alloc
  1            -> kmem_cache_alloc_debug
  1              -> verify_and_copy_pattern
  1              <- verify_and_copy_pattern
  1              -> file_cache_constructor
  1                -> mutex_init
  1                <- mutex_init
  1              <- file_cache_constructor
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1            <- kmem_cache_alloc_debug
  1          <- kmem_cache_alloc
  1          -> crhold
  1          <- crhold
  1        <- falloc
  1        -> vn_openat
  1          -> lookupnameat
  1            -> copyinstr
  1            <- copyinstr
  1            -> lookuppnat
  1              -> lookuppnvp
  1                -> pn_fixslash
  1                <- pn_fixslash
  1                -> pn_getcomponent
  1                <- pn_getcomponent
  1                -> ufs_lookup
  1                  -> dnlc_lookup
  1                    -> bcmp
  1                    <- bcmp
  1                  <- dnlc_lookup
  1                  -> ufs_iaccess
  1                    -> crgetuid
```

```
1                   <- crgetuid
1                   -> groupmember
1                     -> supgroupmember
1                     <- supgroupmember
1                   <- groupmember
1                 <- ufs_iaccess
1               <- ufs_lookup
1               -> vn_rele
1               <- vn_rele
1               -> pn_getcomponent
1               <- pn_getcomponent
1               -> ufs_lookup
1                 -> dnlc_lookup
1                   -> bcmp
1                   <- bcmp
1                 <- dnlc_lookup
1                 -> ufs_iaccess
1                   -> crgetuid
1                   <- crgetuid
1                 <- ufs_iaccess
1               <- ufs_lookup
1               -> vn_rele
1               <- vn_rele
1               -> pn_getcomponent
1               <- pn_getcomponent
1               -> ufs_lookup
1                 -> dnlc_lookup
1                   -> bcmp
1                   <- bcmp
1                 <- dnlc_lookup
1                 -> ufs_iaccess
1                   -> crgetuid
1                   <- crgetuid
1                 <- ufs_iaccess
1                 -> vn_rele
1                 <- vn_rele
1               <- ufs_lookup
1               -> vn_rele
1               <- vn_rele
1             <- lookuppnvp
1           <- lookuppnat
1         <- lookupnameat
1       <- vn_openat
1       -> setf
1         -> fd_reserve
1           -> mutex_owned
1           <- mutex_owned
1           -> mutex_owned
1           <- mutex_owned
1         <- fd_reserve
1         -> cv_broadcast
1         <- cv_broadcast
1       <- setf
1       -> unfalloc
1         -> mutex_owned
```

```
1          <- mutex_owned
1          -> crfree
1          <- crfree
1          -> kmem_cache_free
1            -> kmem_cache_free_debug
1              -> kmem_log_enter
1              <- kmem_log_enter
1              -> tsc_gethrtime
1              <- tsc_gethrtime
1              -> getpcstack
1              <- getpcstack
1              -> kmem_log_enter
1              <- kmem_log_enter
1              -> file_cache_destructor
1                -> mutex_destroy
1                <- mutex_destroy
1              <- file_cache_destructor
1              -> copy_pattern
1              <- copy_pattern
1            <- kmem_cache_free_debug
1          <- kmem_cache_free
1        <- unfalloc
1        -> set_errno
1        <- set_errno
1      <- copen
1    <- open
1  <= open                                          2
```

13.7 Options and Tuning

As with normal trace operation, if a speculative buffer is full when a speculative tracing action is attempted, no data will be stored and a drop count will be increased. If this occurs, one will see a `dtrace` message like:

```
dtrace: 38 speculative drops
```

Note that speculative drops will *not* prevent the full speculative buffer from being copied into the principal buffer when it is committed. Similarly, one will see speculative drops even if drops were experienced on a speculative buffer that was ultimately discarded. As with principal buffers and aggregation buffers, speculative drops can be reduced by increasing the buffer size. In the case of speculative buffers, the buffer size may be increased with the `specsize` option. As with any size option, `specsize` may be specified with any size suffix. As with any buffer option, the resizing policy of this buffer is dictated by the `bufresize` option.

As mentioned, speculative buffers may be unavailable when `speculation()` is called. If this is because all buffers are simply outstanding (that is, if there exist buffers that have not been committed or discarded), one will see this message or similar from `dtrace`:

```
dtrace: 1 failed speculation (no speculative buffer available)
```

Failed speculations of this nature may be reduced by increasing the number of speculative buffers with the `nspec` option. The value of `nspec` defaults to one.

Alternatively, `speculation()` may fail because all speculative buffers are busy. In this case, one will see this message or similar from `dtrace`:

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

This denotes that `speculation()` was called after `commit()` was called for a speculative buffer, but before that buffer was actually committed on all CPUs. Failed speculations of this nature may be reduced by increasing the rate at which CPUs are cleaned with the `cleanrate` option. The value of `cleanrate` defaults to 101.

CHAPTER 14

dtrace(1M) Utility

The dtrace(1M) command is provided as a generic front-end to the DTrace facility. The command implements a simple interface to invoke the D language compiler, the ability to retrieve buffered trace data from the DTrace kernel facility, and a set of basic routines to format and print traced data. This chapter provides a complete reference for the dtrace command.

14.1 Description

The dtrace command provides a generic interface to all of the essential services provided by the DTrace facility, including:

- Options to list the set of probes and providers currently published by DTrace
- Options to enable probes directly using any of the probe description specifiers (provider, module, function, name)
- Options to run the D compiler and compile one or more D program files or programs written directly on the command-line
- Options to generate anonymous tracing programs (see Chapter 29)
- Options to generate program stability reports (see Chapter 32)
- Options to modify DTrace tracing and buffering behavior and enable additional D compiler features (see Chapter 16)

dtrace can also be used to create D scripts by using it in a #! declaration to create an interpreter file (see Chapter 15). Finally, you can use dtrace to attempt to compile D programs and determine their properties without actually enabling any tracing using the -e option, described below.

14.2 Options

The `dtrace` command accepts the following options:

`dtrace` `[-32 | -64] [-aACeEFGHlqSvVwz] [-b` *bufsz*`] [-D` *name* `[=`*def*`]] [-I` *path*`] [-o` *output*`] [-s` *script*`] [-U` *name*`] [-x` *arg* `[=`*val*`]] [-Xa | c | s | t] [-P` *provider* `[ [`*predicate*`]`*action*`]] [-m [ [`*provider:*`]`*module* `[ [`*predicate*`]`*action*`]]] [-f [ [`*provider:*`]`*module:*`]`*func* `[ [`*predicate*`]`*action*`]] [-n [ [ [`*provider:*`]`*module:*`]`*func:*`]`*name* `[ [`*predicate*`]`*action*`]] [-i` *probe-id* `[ [`*predicate*`]`*action*`]]`

where *predicate* is any D predicate enclosed in slashes `/ /` and *action* is any D statement list enclosed in braces { } according to the previously described D language syntax. If D program code is provided as an argument to the `-P`, `-m`, `-f`, `-n`, or `-i` options this text must be appropriately quoted to avoid intepretation by the shell. The options are as follows:

`-32`, `-64` The D compiler produces programs using the native data model of the operating system kernel. You can use the `isainfo`(1) `-b` command to determine the current operating system data model. If the `-32` option is specified, `dtrace` will force the D compiler to compile a D program using the 32-bit data model. If the `-64` option is specified, `dtrace` will force the D compiler to compile a D program using the 64-bit data model. These options are typically not required as `dtrace` selects the native data model as the default. The data model affects the sizes of integer types and other language properties. D programs compiled for either data model may be executed on both 32-bit and 64-bit kernels. The `-32` and `-64` options also determine the ELF file format (ELF32 or ELF64) produced by the `-G` option.

`-a` Claim anonymous tracing state and display the traced data. You can combine the `-a` option with the `-e` option to force `dtrace` to exit immediately after consuming the anonymous tracing state rather than continuing to wait for new data. See Chapter 29 for more information about anonymous tracing.

`-A` Generate `driver.conf`(4) directives for anonymous tracing. If the `-A` option is specified, `dtrace` compiles any D programs specified using the `-s` option or on the command-line and constructs a set of `dtrace`(7D) configuration file directives to enable the specified probes for anonymous tracing (see Chapter 29) and then exits. By default, `dtrace` attempts to store the directives to the file `/kernel/drv/dtrace.conf`; this behavior can be modified using the `-o` option to specify an alternate output file.

`-b` Set principal trace buffer size. The trace buffer size can include any of the size suffixes `k`, `m`, `g`, or `t` as described in Chapter 29. If the buffer space cannot be allocated, `dtrace` attempts to reduce the buffer size or exit depending on the setting of the `bufresize` property.

`-C` Run the C preprocessor `cpp`(1) over D programs before compiling them. Options can be passed to the C preprocessor using the `-D`, `-U`, `-I`, and `-H` options. The degree of C standard conformance can be selected using the `-X` option. Refer to the description of the `-X` option for a description of the set of tokens defined by the D compiler when invoking the C preprocessor.

`-D` Define the specified *name* when invoking `cpp`(1) (enabled using the `-C` option). If an equals sign (`=`) and additional *value* are specified, the name is assigned the corresponding value. This option passes the `-D` option to each `cpp` invocation.

`-e` Exit after compiling any requests and consuming anonymous tracing state (`-a` option) but prior to enabling any probes. This option can be combined with the `-a` option to print anonymous tracing data and exit, or it can be combined with D compiler options to verify that the programs compile without actually executing them and enabling the corresponding instrumentation.

`-E` Exit after compiling any requests and enabling probes but prior to tracing any data. This option can be used to compile a set of probes and enable them without attempting to consume any buffer data from DTrace.

`-f` Specify function name to trace or list (`-l` option). The corresponding argument can include any of the probe description forms *provider:module:function*, *module:function*, or *function*. Unspecified probe description fields are left blank and match any probes regardless of the values in those fields. If no qualifiers other than *function* are specified in the description, all probes with the corresponding *function* are matched. The `-f` argument can be suffixed with an optional D probe clause. More than one `-f` option may be specified on the command-line at a time.

`-F` Coalesce trace output by identifying function entry and return. Function entry probe reports are indented and their output is prefixed with `->`. Function return probe reports are unindented and their output is prefixed with `<-`.

`-G` Generate an ELF file containing an embedded DTrace program. The DTrace probes specified in the program are saved inside of a relocatable ELF object which can be linked into another program.

`-H` Print the pathnames of included files when invoking `cpp`(1) (enabled using the `-C` option). This option passes the `-H` option to each `cpp` invocation, causing it to display the list of pathnames, one per line, to `stderr`.

`-i` Specify probe identifier to trace or list (`-l` option). Probe IDs are specified using decimal integers as shown by dtrace `-l`. The `-i` argument can be suffixed with an optional D probe clause. More than one `-i` option may be specified on the command-line at a time.

`-I` Add the specified directory *path* to the search path for `#include` files when invoking `cpp`(1) (enabled using the `-C` option). This option passes

the `-I` option to each `cpp` invocation. The specified directory is inserted into the search path ahead of the default directory list.

`-l` List probes instead of enabling them. If the `-l` option is specified, `dtrace` will produce a report of the probes matching the descriptions given using the `-P`, `-m`, `-f`, `-n`, and `-i` options. If none of these options are specified, all probes are listed.

`-m` Specify module name to trace or list (`-l` option). The corresponding argument can include any of the probe description forms *provider:module* or *module*. Unspecified probe description fields are left blank and match any probes regardless of the values in those fields. If no qualifiers other than *module* are specified in the description, all probes with a corresponding *module* are matched. The `-m` argument can be suffixed with an optional D probe clause. More than one `-m` option may be specified on the command-line at a time.

`-n` Specify probe name to trace or list (`-l` option). The corresponding argument can include any of the probe description forms *provider:module:function:name*, *module:function:name*, *function:name*, or *name*. Unspecified probe description fields are left blank and match any probes regardless of the values in those fields. If no qualifiers other than *name* are specified in the description, all probes with a corresponding *name* are matched. The `-n` argument can be suffixed with an optional D probe clause. More than one `-n` option may be specified on the command-line at a time.

`-o` Specify the *output* file for the `-A` and `-G` options. If the `-A` option is present and `-o` is not present, the default output file is `/kernel/drv/dtrace.conf`. If the `-G` option is present and the `-s` option's argument is of the form *filename*`.d` and `-o` is not present, the default output file is *filename*`.o`; otherwise the default output file is `d.out`.

`-P` Specify provider name to trace or list (`-l` option). The remaining probe description fields module, function, and name are left blank and match any probes regardless of the values in those fields. The `-P` argument can be suffixed with an optional D probe clause. More than one `-P` option may be specified on the command-line at a time.

`-q` Set quiet mode. `dtrace` will suppress messages such as the number of probes matched by the specified options and D programs and will not print column headers, the CPU ID, the probe ID, or insert newlines into the output. Only data traced and formatted by D program statements such as `trace()` and `printf()` will be displayed to `stdout`.

`-s` Compile the specified D program source file. If the `-e` option is present, the program will be compiled but no instrumentation will be enabled. If the `-l` option is present, the program will be compiled and the set of probes matched by it will be listed, but no instrumentation will be enabled. If neither `-e` nor `-l` are present, the instrumentation specified by the D program will be enabled and tracing will begin.

`-S` Show D compiler intermediate code. The D compiler will produce a report of the intermediate code generated for each D program to `stderr`.

`-U` Undefine the specified *name* when invoking `cpp`(1) (enabled using the `-C` option). This option passes the `-U` option to each `cpp` invocation.

`-v` Set verbose mode. If the `-v` option is specified, `dtrace` will produce a program stability report showing the minimum interface stability and dependency level for the specified D programs. DTrace stability levels are explained in further detail in Chapter 32.

`-V` Report the highest D programming interface version supported by `dtrace`. The version information is printed to `stdout` and the `dtrace` command exits. See Chapter 34 for more information about DTrace versioning features.

`-w` Permit destructive actions in D programs specified using the `-s`, `-P`, `-m`, `-f`, `-n`, or `-i` options. If the `-w` option is not specified, `dtrace` will not permit the compilation or enabling of a D program that contains destructive actions. Destructive actions are described in further detail in Chapter 10.

`-x` Enable or modify a DTrace runtime option or D compiler option. The list of options is found in Chapter 16. Boolean options are enabled by specifying their name. Options with values are set by separating the option name and value with an equals sign (`=`).

`-X` Specify the degree of conformance to the ISO C standard that should be selected when invoking `cpp`(1) (enabled using the `-C` option). The `-X` option argument affects the value and presence of the `__STDC__` macro depending upon the value of the argument letter:

`a` (default)	ISO C plus K&R compatibility extensions, with semantic changes required by ISO C. This is the default mode if `-X` is not specified. The predefined macro `__STDC__` has a value of 0 when `cpp` is invoked in conjunction with the `-Xa` option.
`c` (conformance)	Strictly conformant ISO C, without K&R C compatibility extensions. The predefined macro `__STDC__` has a value of 1 when `cpp` is invoked in conjunction with the `-Xc` option.
`s` (K&R C)	K&R C only. The macro `__STDC__` is not defined when `cpp` is invoked in conjunction with the `-Xs` option.
`t` (transition)	ISO C plus K&R C compatibility extensions, without semantic changes required by ISO C. The predefined macro `__STDC__` has a value of 0 when `cpp` is invoked in conjunction with the `-Xt` option.

As the `-X` option only affects how the D compiler invokes the C preprocessor, the `-Xa` and `-Xt` options are equivalent from the perspective of D and both are provided only to ease re-use of settings from a C build environment.

Regardless of the `-X` mode, the following additional C preprocessor definitions are always specified and valid in all modes:

- `__sun`
- `__unix`
- `__SVR4`
- `__sparc` (on SPARC™ systems only)
- `__i386` (on x86 systems only)
- `__`*‘uname -s‘*`_`*‘uname -r‘* (for example, `__SunOS_5_10`)
- `__SUNW_D=1`
- `__SUNW_D_VERSION=0x`*MMmmmuuu* (where *MM* is the Major release value in hexadecimal, *mmm* is the Minor release value in hexadecimal, and *uuu* is the Micro release value in hexadecimal; see Chapter 34 for more information about DTrace versioning)

`-z` Permit probe descriptions that match zero probes. If the `-z` option is not specified, `dtrace` will report an error and exit if any probe descriptions specified in D program files (`-s` option) or on the command-line (`-P`, `-m`, `-f`, `-n`, or `-i` options) contain descriptions that do not match any known probes.

14.3 Operands

Zero or more additional arguments may be specified on the `dtrace` command line to define a set of macro variables (`$1`, `$2`, and so on) to be used in any D programs specified using the `-s` option or on the command-line. The use of macro variables is described further in Chapter 15.

14.4 Exit Status

The following exit values are returned by the `dtrace` utility:

0 The specified requests were completed successfully. For D program requests, the 0 exit status indicates programs were successfully compiled,

probes were successfully enabled, or anonymous state was successfully retrieved. `dtrace` returns 0 even if the specified tracing requests encounted errors or drops.

1 A fatal error occurred. For D program requests, the 1 exit status indicates that program compilation failed or that the specified request could not be satisfied.

2 Invalid command-line options or arguments were specified.

CHAPTER 15

Scripting

The `dtrace`(1M) utility can be used to create interpreter files out of D programs similar to shell scripts that you can install as reusable interactive DTrace tools. The D compiler and `dtrace` command provide a set of *macro variables* that are expanded by the D compiler that make it easy to create DTrace scripts. This chapter provides a reference for the macro variable facility and tips for creating persistent scripts.

15.1 Interpreter Files

Similar to your shell and utilities such as `awk`(1) and `perl`(1), `dtrace`(1M) can be used to create executable interpreter files. An interpreter file begins with a line of the form:

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter and *arg* is a single optional argument. When an interpreter file is executed, the system invokes the specifier interpreter. If *arg* was specified in the interpreter file, it is passed as an argument to the interpreter. The path to the interpreter file itself and any additional arguments specified when it was executed are then appended to the interpreter argument list. Therefore, you will always need to create DTrace interpreter files with at least these arguments:

```
#!/usr/sbin/dtrace -s
```

When your interpreter file is executed, the argument to the `-s` option will therefore be the pathname of the interpreter file itself. `dtrace` will then read, compile, and execute this file as if you had typed:

`# dtrace -s` *interpreter-file*

in your shell. Here is a simple example of how to create and execute a `dtrace` interpreter file. Type in the following D source code and save it in a file named `interp.d`:

```
#!/usr/sbin/dtrace -s
BEGIN
{
    trace("hello");
    exit(0);
}
```

You can now mark the `interp.d` file as executable and execute it:

```
# chmod a+rx interp.d
# ./interp.d
dtrace: script './interp.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello
#
```

Remember that the `#!` directive must comprise the first two characters of your file with no intervening or preceding whitespace. The D compiler knows to automatically ignore this line when it processes the interpreter file.

`dtrace` uses `getopt`(3) to process its command-line options, so you can combine multiple options in your single interpreter argument. For example, to add the `-q` option to the preceding example you could change the interpreter directive to:

```
#!/usr/sbin/dtrace -qs
```

If you specify multiple option letters, the `-s` option must always end the list of boolean options so that the next argument (the interpreter file name) is processed as the argument corresponding to the `-s` option.

If you need to specify more than one option that requires an argument in your interpreter file, you will not be able to fit all your options and arguments into the single interpreter argument. Instead, use the `#pragma D option` directive syntax to set your options. All of the `dtrace` command-line options have `#pragma` equivalents that you can use, as shown in Chapter 16.

15.2 Macro Variables

The D compiler defines a set of built-in macro variables that you can use when writing D programs or interpreter files. Macro variables are identifiers that are prefixed with a dollar sign ($) and are expanded once by the D compiler when processing your input file. The D compiler provides the following macro variables:

TABLE 15–1 D Macro Variables

Name	Description	Reference
`$[0-9]+`	macro arguments	See "15.3 Macro Arguments" on page 174, below
`$egid`	effective group-ID	`getegid`(2)
`$euid`	effective user-ID	`geteuid`(2)
`$gid`	real group-ID	`getgid`(2)
`$pid`	process ID	`getpid`(2)
`$pgid`	parent group ID	`getpgid`(2)
`$ppid`	parent process ID	`getppid`(2)
`$projid`	project ID	`getprojid`(2)
`$sid`	session ID	`getsid`(2)
`$taskid`	task ID	`gettaskid`(2)
`$uid`	real user-ID	`getuid`(2)

Except for the `$[0-9]+` macro arguments (described below), the macro variables all expand to integers corresponding to system attributes such as the process ID and user ID. The variables expand to the attribute value associated with the current `dtrace` process itself, or whatever process is running the D compiler.

Macro variables are useful in interpreter files because they allow you to create persistent D programs that do not need to be edited each time you want to use them, as we did for some of our earliest examples in Chapter 1. For example, to count all system calls except those executed by the `dtrace` command itself, you can use the following D program clause containing `$pid`:

```
syscall:::entry
/pid != $pid/
{
    @calls = count();
}
```

This clause always produces the desired result, even though each invocation of the `dtrace` command will have a different process ID.

Macro variables can be used anywhere an integer, identifier, or string can be used in a D program, but are expanded only once (that is, not recursively) when the input file is parsed. Each macro variable is expanded to form a separate input token, and cannot be concatenated with other text to yield a single token. For example, if `$pid` expands to the value 456, the D code:

```
123$pid
```

would expand to the two adjacent tokens 123 and 456 (resulting in a syntax error), rather than the single integer token 123456.

Macro variables *are* expanded and concatenated with adjacent text inside of D probe descriptions at the start of your program clauses. For example, the following clause utilizes the DTrace `pid` provider to instrument the `dtrace` command itself:

```
pid$pid:libc.so:printf:entry
{
    ...
}
```

Macro variables are only expanded once within each probe description field; they may not contain probe description delimeters (`:`).

15.3 Macro Arguments

The D compiler also provides a set of macro variables corresponding to any additional argument operands specified as part of the `dtrace` command invocation. These *macro arguments* are accessed using the built-in names `$0` for name of the D program file or `dtrace` command, `$1` for the first additional operand, `$2` for the second operand, and so on. If you use the `dtrace -s` option, `$0` expands to the value of the name of the input file used with this option. For D programs specified on the command-line, `$0` expands to the value of `argv[0]` used to exec `dtrace` itself.

Macro arguments can expand to integers, identifiers, or strings, depending on the form of the corresponding text. As with all macro variables, macro arguments can be used anywhere integer, identifier, and string tokens can be used in a D program. All of the following examples could form valid D expressions assuming appropriate macro argument values:

```
execname == $1    /* with a string macro argument */
x += $1           /* with an integer macro argument */
trace(x->$1)      /* with an identifier macro argument */
```

Macro arguments can be used to create `dtrace` interpreter files that act like real Solaris commands and use information specified by a user or by another tool to modify their behavior. For example, the following D interpreter file traces `write`(2) system calls executed by a particular process ID:

```
#!/usr/sbin/dtrace -s

syscall::write:entry
```

```
/pid == $1/
{
}
```

If you make this interpreter file executable, you can specify the value of `$1` using an additional command-line argument to your interpreter file:

```
# chmod a+rx ./tracewrite
# ./tracewrite 12345
```

The resulting command invocation counts each `write`(2) system call executed by process ID 12345.

If your D program references a macro argument that is not provided on the command-line, an appropriate error message will be printed and your program will fail to compile:

```
# ./tracewrite
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

The D compiler will also produce an error message if additional arguments are specified on the command line that are not referenced by your D program.

The macro argument values must match the form of an integer, identifier, or string. If the argument does not match any of these forms, the D compiler will report an appropriate error message. When specifying string macro arguments to a DTrace interpreter file, it is typically necessary to surround the argument in an extra pair of single quotes to avoid interpretation of the double quotes and string contents by the shell:

```
# ./foo '"a string argument"'
```

If you want your D macro arguments to always be interpreted as string tokens even if they match the form of an integer or identifier, you can prefix the macro variable or argument name with two leading dollar signs (for example, `$$1`), to force the D compiler to interpret the argument value as if it were a string surrounded by double quotes. All the usual D string escape sequences (see Table 2–5) are expanded inside of any string macro arguments, regardless of whether they are referenced using the *$arg* or *$$arg* form of the macro.

CHAPTER 16

Options and Tunables

To allow for customization, DTrace affords its consumers several important degrees of freedom. To minimize the likelihood of requiring specific tuning, DTrace is implemented using reasonable default values and flexible default policies. However, situations may arise that require tuning the behavior of DTrace on a consumer-by-consumer basis. In this chapter, we describe the DTrace options and tunables and the interfaces you can use to modify them.

16.1 Consumer Options

DTrace is tuned by setting or enabling options; the available options (along with the units of their corresponding values) are listed in the table below. For some options, `dtrace`(1M) provides a corresponding command-line option; these are listed in the third column. Details on the specifics of each option may be found in the chapter listed in the rightmost column.

TABLE 16–1 DTrace Consumer Options

Option Name	Value	`dtrace`(1M) Alias	Description	See Chapter
`aggrate`	*time*		Rate of aggregation reading	Chapter 9
`aggsize`	*size*		Aggregation buffer size	Chapter 9
`bufresize`	`auto` or `manual`		Buffer resizing policy	Chapter 11

TABLE 16–1 DTrace Consumer Options *(Continued)*

Option Name	Value	`dtrace`(1M) Alias	Description	See Chapter
`bufsize`	*size*	`-b`	Principal buffer size	Chapter 11
`cleanrate`	*time*		Cleaning rate	Chapter 13
`cpu`	*scalar*	`-c`	CPU on which to enable tracing	Chapter 11
`dynvarsize`	*size*		Dynamic variable space size	Chapter 3
`flowindent`	—	`-F`	Indent function entry and prefix with `->`; unindent function return and prefix with `<-`	Chapter 14
`grabanon`	—	`-a`	Claim anonymous state	Chapter 29
`nspec`	*scalar*		Number of speculations	Chapter 13
`quiet`	—	`-q`	Only output explicitly traced data	Chapter 14
`specsize`	*size*		Speculation buffer size	Chapter 13
`strsize`	*size*		String size	Chapter 6
`stackframes`	*scalar*		Number of stack frames	Chapter 10
`statusrate`	*time*		Rate of status checking	
`switchrate`	*time*		Rate of buffer switching	Chapter 11
`ustackframes`	*scalar*		Number of user stack frames	Chapter 10

Values that denote sizes may be given an optional suffix of `k`, `m`, `g`, or `t` to denote kilobytes, megabytes, gigabytes and terabytes respectively. Values that denote times may be given an optional suffix of `ns`, `us`, `ms`, `s` or `hz` to denote nanoseconds, microseconds, milliseconds, seconds, and number-per-second, respectively.

16.2 Modifying Options

Options may be set in a D script by using `#pragma D` followed by the string `option` and the option name. If the option takes a value, the option name should be followed by an equals sign (=) and the option value. For example, all of the following are valid option settings:

```
#pragma D option nspec=4
#pragma D option grabanon
#pragma D option bufsize=2g
#pragma D option switchrate=10hz
#pragma D option aggrate=100us
#pragma D option bufresize=manual
```

The `dtrace`(1M) command also permits accepts option settings on the command-line as an argument to the `-x` option. For example:

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \
    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

If an invalid option is specified, `dtrace` will indicate that the option name is invalid and abort:

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
#
```

Similarly, if an option value is not valid for the given option (due to an illegal suffix or otherwise illegal value), `dtrace` will indicate that the value itself is invalid:

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
#
```

If an option is set more than once, subsequent settings overwrite earlier settings. Some options (like `grabanon`) may *only* be set; the presence of the option sets it, and there is no way to subsequently unset it.

If options are set for an anonymous enabling, those options will be honored by the DTrace consumer that claims the anonymous state. See Chapter 29 for details on enabling anonymous tracing.

CHAPTER 17

dtrace Provider

The dtrace provider provides several probes related to DTrace itself. These probes can be used for initializing state before tracing begins, processing state after tracing has completed, and handling unexpected execution errors in the probes themselves.

17.1 The BEGIN Probe

The BEGIN probe fires before any other — no other probe will fire until all BEGIN clauses have completed. This probe can be used to initialize any state that is needed in other probes. For example, we can use the BEGIN probe to initialize an associative array to map between mmap(2) protection bits and a textual representation:

```
BEGIN
{
    prot[0] = "---";
    prot[1] = "r--";
    prot[2] = "-w-";
    prot[3] = "rw-";
    prot[4] = "--x";
    prot[5] = "r-x";
    prot[6] = "-wx";
    prot[7] = "rwx";
}

syscall::mmap:entry
{
    printf("mmap with prot = %s", prot[arg2 & 0x7]);
}
```

The BEGIN probe fires in an unspecified context. This means that the output of stack() or ustack(), and the value of context-specific variables (for example, execname), are all arbitrary, and are not values that should be relied upon or interpreted to infer any meaningful information. There are no arguments defined for the BEGIN probe.

17.2 The END Probe

The END probe fires after all others — it will not fire until all other probe clauses have completed. This can be used to process state that has been gathered or to format the output — the printa() action is often used in the END probe. The BEGIN and END probes can be used together to measure the total time spent tracing:

```
BEGIN
{
    start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
    printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

See "9.4 Normalization" on page 116 and "12.2 printa()" on page 149 for other common uses of the END probe.

As with the BEGIN probe, there are no arguments to the END probe and the context in which it fires is arbitrary and should not be depended upon.

When tracing with the bufpolicy option set to fill, adequate space is reserved to accomodate any records traced in the END probe. See "11.2.2.1 fill Policy and END Probes" on page 139 for details.

17.2.1 The END Probe and the exit() Action

The exit() action causes tracing to stop and the END probe to fire, however there is some delay between the invocation of the exit() action and the END probe firing (during this delay, no probes will fire). After a probe invokes the exit() action, the END probe is not fired until the DTrace consumer determines that exit() has been called and stops tracing; the rate at which the exit status is checked can be set using statusrate option. (See Chapter 16).

17.3 The ERROR Probe

The ERROR probe fires when there is a run-time error in executing a clause for a DTrace probe. For example, if a clause attempts to dereference a NULL pointer, the ERROR probe will fire:

EXAMPLE 17–1 error.d: Record Errors

```
BEGIN
{
    *(char *)NULL;
}

ERROR
{
    printf("Hit an error!");
}
```

When we run this program, we'll see output like this:

```
# dtrace -s ./error.d
dtrace: script './error.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  2      3                             :ERROR Hit an error!
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #1 at DIF offset 12
dtrace: 1 error on CPU 2
```

We can see that our ERROR probe fired, but we also see output from dtrace(1M) reporting the error. This is because dtrace has its own enabling of the ERROR probe to allow it to report errors! Using the ERROR probe, you can create your own custom error handling.

The arguments to the ERROR probe are as follows:

arg1	The enabled probe identifier (EPID) of the probe that caused the error
arg2	The index of the action that caused the fault
arg3	The DIF offset into that action or -1 if not applicable
arg4	The fault type (see below)
arg5	Value particular to the fault type (see below)

The table below describes the various fault types and the value that `arg5` will have for each:

`arg4` Value	Description	`arg5` Meaning
`DTRACEFLT_UNKNOWN`	Unknown fault type	—
`DTRACEFLT_BADADDR`	Access to unmapped or invalid address	Address accessed
`DTRACEFLT_BADALIGN`	Unaligned memory access	Address accessed
`DTRACEFLT_ILLOP`	Illegal or invalid operation	—
`DTRACEFLT_DIVZERO`	Integer divide by zero	—
`DTRACEFLT_NOSCRATCH`	Insufficient scratch space to satisfy scratch allocation	—
`DTRACEFLT_KPRIV`	Attempt to access a kernel address or property without sufficient privileges	Address accessed or `0` if not applicable
`DTRACEFLT_UPRIV`	Attempt to access a user address or property without sufficient privileges	Address accessed or `0` if not applicable
`DTRACEFLT_TUPOFLOW`	DTrace internal parameter stack overflow	—

If the actions taken in the `ERROR` probe itself cause an error, that error is silently dropped — the `ERROR` probe will not be recursively invoked.

17.4 Stability

The `dtrace` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Stable	Stable	Common
Module	Private	Private	Unknown
Function	Private	Private	Unknown

Element	Name stability	Data stability	Dependency class
Name	Stable	Stable	Common
Arguments	Stable	Stable	Common

CHAPTER 18

lockstat Provider

The lockstat provider makes available probes that can be used to discern lock contention statistics, or to understand virtually any aspect of locking behavior. Indeed, the lockstat(1M) command is simply a DTrace consumer that uses the lockstat provider to gather its raw data.

18.1 Overview

The lockstat provider makes available two kinds of probes:

Contention-event probes correspond to contention on a synchronization primitive, and fire when a thread is forced to wait for a resource to become available. Solaris is generally optimized for the non-contention case, so prolonged contention is not expected; these probes should be used to understand those cases where contention does arise. Because contention is designed to be (relatively) rare, enabling contention-event probes generally doesn't have a serious probe effect; they can be enabled without concern for substantially affecting performance.

Hold-event probes correspond to acquiring, releasing or otherwise manipulating a synchronization primitive. As such, these probes can be used to answer arbitrary questions about the way synchronization primitives are manipulated. Because Solaris acquires and releases synchronization primitives very often (on the order of millions of times per second per CPU on a busy system), enabling hold-event probes has a much higher probe effect than does enabling contention-event probes. While the probe effect induced by enabling them can be substantial, it is not pathological; they may still be enabled with confidence on production systems.

The lockstat provider makes available probes that correspond to the different synchronization primitives in Solaris; these primitives — and the probes that correspond to them — are discussed in the sections that follow.

18.2 Adaptive Lock Probes

Adaptive locks enforce mutual exclusion to a critical section, and may be acquired in most contexts in the kernel. Because they have few context restrictions, adaptive locks comprise the vast majority of synchronization primitives in the Solaris kernel. These locks are adaptive in their behavior with respect to contention: when a thread attempts to acquire a held adaptive lock, it will determine if the owning thread is currently running on a CPU. If the owner is running on another CPU, the acquiring thread will *spin*; if the owner is not running, the acquiring thread will *block*.

The four lockstat probes pertaining to adaptive locks are in Table 18–1. For each probe, `arg0` contains a pointer to the `kmutex_t` structure that represents the adaptive lock.

TABLE 18–1 Adaptive Lock Probes

`adaptive-acquire`	Hold-event probe that fires immediately after an adaptive lock is acquired.
`adaptive-block`	Contention-event probe that fires after a thread that has blocked on a held adaptive mutex has reawakened and has acquired the mutex. If both are enabled, `adaptive-block` fires *before* `adaptive-acquire`. At most one of `adaptive-block` and `adaptive-spin` will fire for a single lock acquisition. `arg1` for `adaptive-block` contains the sleep time in nanoseconds.
`adaptive-spin`	Contention-event probe that fires after a thread that has spun on a held adaptive mutex has successfully acquired the mutex. If both are enabled, `adaptive-spin` fires *before* `adaptive-acquire`. At most one of `adaptive-spin` and `adaptive-block` will fire for a single lock acquisition. `arg1` for `adaptive-spin` contains the *spin count*: the number of iterations that were taken through the spin loop before the lock was acquired. The spin count has little meaning on its own, but can be used to compare spin times.
`adaptive-release`	Hold-event probe that fires immediately after an adaptive lock is released.

18.3 Spin Lock Probes

There are some contexts in the kernel — notably high-level interrupt context and any context manipulating dispatcher state — in which one may not block. In these contexts, this restriction prevents the use of adaptive locks. *Spin locks* are instead used

to effect mutual exclusion to critical sections in these contexts. As the name implies, the behavior of these locks in the presence of contention is to spin until the lock is released by the owning thread. The three probes pertaining to spin locks are in Table 18–2.

TABLE 18–2 Spin Lock Probes

`spin-acquire`	Hold-event probe that fires immediately after a spin lock is acquired.
`spin-spin`	Contention-event probe that fires after a thread that has spun on a held spin lock has successfully acquired the spin lock. If both are enabled, `spin-spin` fires *before* `spin-acquire`. `arg1` for `spin-spin` contains the *spin count*: the number of iterations that were taken through the spin loop before the lock was acquired. The spin count has little meaning on its own, but can be used to compare spin times.
`spin-release`	Hold-event probe that fires immediately after a spin lock is released.

Adaptive locks are much more common than spin locks; you can use a simple DTrace script to understand the degree to which this is true:

```
lockstat:::adaptive-acquire
/execname == "date"/
{
    @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
    @locks["spin"] = count();
}
```

Run this script in one window, and a `date`(1) command in another. When you terminate the DTrace script you'll see something like:

```
# dtrace -s ./whatlock.d
dtrace: script './whatlock.d' matched 5 probes
^C
spin                                                             26
adaptive                                                       2981
```

As this indicates, over 99 percent of the locks acquired in running `date` are adaptive locks. (It may be surprising that *so* many locks are acquired in doing something as simple as a `date`. This is only startling to the uninitiated: the large number of locks is a natural artifact of the fine-grained locking required of an extremely scalable system like the Solaris kernel.)

18.4 Thread Locks

Thread locks are a special kind of spin lock that are used to lock a thread for purposes of changing thread state. Thread lock hold events are available as spin lock hold-event probes (that is, `spin-acquire` and `spin-release`), but contention events have their own probe specific to thread locks. The thread lock hold-event probe is in Table 18–3.

TABLE 18–3 Thread Lock Probe

`thread-spin`	Contention-event probe that fires after a thread has spun on a thread lock. Like other contention-event probes, if both the contention-event probe and the hold-event probe are enabled, `thread-spin` will fire before `spin-acquire`. Unlike other contention-event probes, however, thread-spin fires *before* the lock is actually acquired. As a result, there may be multiple `thread-spin` probe firings corresponding to a single `spin-acquire` probe firing.

18.5 Readers/Writer Lock Probes

Readers/writer locks enforce a policy of allowing multiple readers *or* a single writer — but not both — to be in a critical section. They are typically used for structures that are searched more frequently than they are modified, and for which there is substantial time in the critical section. (If critical section times are short, readers/writer locks will implicitly serialize over the shared memory used to implement the lock, giving them no advantage over adaptive locks.) See `rwlock`(9F) for more details on readers/writer locks.

The probes pertaining to readers/writer locks are in Table 18–4. For each probe, `arg0` contains a pointer to the `krwlock_t` structure that represents the adaptive lock.

TABLE 18–4 Readers/Writer Lock Probes

`rw-acquire`	Hold-event probe that fires immediately after a readers/writer lock is acquired. `arg1` contains the constant `RW_READER` if the lock was acquired as a reader, and `RW_WRITER` if the lock was acquired as a writer.

TABLE 18–4 Readers/Writer Lock Probes *(Continued)*

`rw-block`	Contention-event probe that fires after a thread that has blocked on a held readers/writer lock has reawakened and has acquired the lock. `arg1` contains the length of time (in nanoseconds) that the current thread had to sleep to acquire the lock. `arg2` contains the constant `RW_READER` if the lock was acquired as a reader, and `RW_WRITER` if the lock was acquired as a writer. `arg3` and `arg4` contain more information on the reason for blocking. `arg3` is non-zero if and only if the lock was held as a writer when the current thread blocked and `arg4` contains the readers count when the current thread blocked. If both are enabled, `rw-block` fires *before* `rw-acquire`.
`rw-upgrade`	Hold-event probe that fires after a thread has successfully upgraded a readers/writer lock from a reader to a writer. Upgrades do not have an associated contention event because they are only possible through a non-blocking interface, `rw_tryupgrade`(9F).
`rw-downgrade`	Hold-event probe that fires after a thread had downgraded its ownership of a readers/writer lock from writer to reader. Downgrades do not have an associated contention event because — by definition — they always succeed without contention.
`rw-release`	Hold-event probe that fires immediately after a readers/writer lock is released. `arg1` contains the constant `RW_READER` if the released lock was held as a reader, and `RW_WRITER` if the released lock was held as a writer. Note that due to upgrades and downgrades, the lock may *not* have been released as it was acquired.

18.6 Stability

The `lockstat` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	Common
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	Common
Arguments	Evolving	Evolving	Common

CHAPTER 19

`profile` Provider

The `profile` provider provides *unanchored* probes — probes that are not associated with any particular point of execution, but rather with some asynchronous event source. In the case of the `profile` provider, the event source is a time-based interrupt firing every fixed, specified time interval. These probes can be used to sample some aspect of system state every unit time — and the samples can then be used to infer system behavior. If the sampling rate is high, or the sampling time is long, an accurate inference is possible. Thanks to the arbitrary actions of DTrace, the `profile` provider can be used to sample practically anything in the system. For example, one could sample the state of the current thread, the state of the CPU, or the current machine instruction.

19.1 `profile-`*n* probes

A `profile-`*n* probe fires every fixed interval on every CPU at high interrupt level. The probe's firing interval is denoted by the value of *n*: the interrupt source will fire *n* times per second. *n* may also have an optional time suffix, in which case *n* is interpreted to be in the units denoted by the suffix. Valid suffixes — and the units they denote — are in Table 19–1.

TABLE 19–1 Valid time suffixes

Suffix	Time units
`nsec` or `ns`	nanoseconds
`usec` or `us`	microseconds
`msec` or `ms`	milliseconds

TABLE 19–1 Valid time suffixes *(Continued)*

Suffix	Time units
`sec` or `s`	seconds
`min` or `m`	minutes
`hour` or `h`	hours
`day` or `d`	days
`hz`	hertz (frequency per second)

For example, we can create a probe to fire at 97 hertz to sample the currently running process:

```
#pragma D option quiet

profile-97
/pid != 0/
{
    @proc[pid, execname] = count();
}

END
{
    printf("%-8s %-40s %s\n", "PID", "CMD", "COUNT");
    printa("%-8d %-40s %@d\n", @proc);
}
```

Running the above for a little while:

```
# dtrace -s ./prof.d
^C
PID      CMD                                      COUNT
223887   sh                                       1
100360   httpd                                    1
100409   mibiisa                                  1
223887   uname                                    1
218848   sh                                       2
218984   adeptedit                                2
100224   nscd                                     3
3        fsflush                                  4
2        pageout                                  6
100372   java                                     7
115279   xterm                                    7
100460   Xsun                                     7
100475   perfbar                                  9
223888   prstat                                   15
```

You can also use the `profile-`*n* provider to sample information about the running process. For example, this D script uses a 1,001 hertz profile probe to sample the current priority of a specified process:

```
profile-1001
/pid == $1/
{
    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);
}
```

To see this in action, type the following in one window:

```
$ echo $$
494621
$ while true ; do let i=0 ; done
```

In another window, run the D script for a little while:

```
# dtrace -s ./profpri.d 494621
 dtrace: script './profpri.d' matched 1 probe
^C
ksh
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 7443
              10 |@@@@@@@                                  2235
              20 |@@@@                                     1679
              30 |@@@                                      1119
              40 |@                                        560
              50 |@                                        554
              60 |                                         0
```

This shows you the clear bias of the timesharing scheduling class: because the shell process is spinning on the CPU, its priority is constantly being lowered by the system. If the shell process were running less frequently, its priority would be higher. Indeed, as an experiment, `^C` the spinning shell, and rerun the script:

```
# dtrace -s ./profpri.d 494621
 dtrace: script './profpri.d' matched 1 probe
```

Now go back to the shell, and type a few characters. When you terminate the DTrace script, you'll see output like the following:

```
ksh
           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

Because the shell process was sleeping awaiting user-input instead of spinning on the CPU, when it *did* run it was run at a much higher priority.

19.2 `tick-`*n* probes

Like `profile-`*n* probes, `tick-`*n* probes fire every fixed interval at high interrupt level. However, unlike `profile-`*n* probes — which fire on *every* CPU — `tick-`*n* probes fire on only *one* CPU per interval. The actual CPU may change over time, but `tick-`*n* probes are guaranteed to only fire on one CPU at a time. As with `profile-`*n* probes, *n* defaults to rate-per-second but may also have an optional time suffix. `tick-`*n* probes have several uses; one use may be to provide some periodic output, or to take a periodic action.

19.3 Arguments

The arguments to `profile` probes are as follows:

`arg0`	The program counter (PC) in the kernel at the time that the probe fired, or 0 if the current process was not executing in the kernel at the time that the probe fired
`arg1`	The PC in the user-level process at the time that the probe fired, or 0 if the current process was executing at the kernel at the time that the probe fired

As the above implies, `arg0` is non-zero and `arg1` is zero, or `arg0` is zero and `arg1` is non-zero. (That is, the logical exclusive-or of `arg0` and `arg1` is always true.) Thus, `arg0` and `arg1` may be used to differentiate user-level from kernel level, as in this simple example:

```
profile-1ms
{
    @ticks[arg0 ? "kernel" : "user"] = count();
}
```

19.4 Resolution

The `profile` provider makes use of arbitrary resolution interval timers in the operating system. While all hardware architectures support this facility in some form, not all have support for truly arbitrary resolution time-based interrupts. On architectures that do not support a truly arbitrary resolution, the frequency will be

limited by the system clock frequency, which is specified by the `hz` kernel variable. Probes of higher frequency than `hz` on such architectures will fire some number of times every `1/hz` seconds. For example, a 1000 hertz `profile` probe on such an architecture with `hz` set to 100 will fire ten times in rapid succession every ten milliseconds. On platforms that support arbitrary resolution, a 1000 hertz `profile` probe would fire exactly every one millisecond.

To test a given architecture's resolution, use this D script:

```
profile-5000
{
    /*
     * We divide by 1,000,000 to convert nanoseconds to milliseconds, and
     * then we take the value mod 10 to get the current millisecond within
     * a 10 millisecond window.  On platforms that do not support truly
     * arbitrary resolution profile probes, all of the profile-5000 probes
     * will fire on roughly the same millisecond.  On platforms that
     * support a truly arbitrary resolution, the probe firings will be
     * evenly distributed across the milliseconds.
     */
    @ms = lquantize((timestamp / 1000000) % 10, 0, 10, 1);
}

tick-1sec
/i++ >= 10/
{
    exit(0);
}
```

On an architecture that supports arbitrary resolution `profile` probes, running the above will yield an even distribution:

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  33631                      :tick-1sec


           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      10760
               1 |@@@@                                     10842
               2 |@@@@                                     10861
               3 |@@@                                      10820
               4 |@@@                                      10819
               5 |@@@                                      10817
               6 |@@@@                                     10826
               7 |@@@@                                     10847
               8 |@@@@                                     10830
               9 |@@@@                                     10830
```

On an architecture that does not support arbitrary resolution `profile` probes, running the above will yield a distinctly uneven distribution:

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
 CPU     ID                    FUNCTION:NAME
  0  28321                      :tick-1sec


           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  107864
               6 |                                         424
               7 |                                         255
               8 |                                         496
               9 |                                         0
```

On these architectures, `hz` may be manually tuned in `/etc/system` to improve the effective profile resolution.

Currently, all variants of UltraSPARC ("`sun4u`") support arbitrary resolution `profile` probes; all variants of the x86 architecture ("`i86pc`") do not.

19.5 Probe creation

Unlike other providers, the `profile` provider creates probes on-the-fly on an as-needed basis. Thus, one may not see the desired profile probe in a listing of all probes (for example, by using `dtrace -l -P profile`); the probe will be created when it is explicitly enabled.

On architectures that support arbitrary resolution `profile` probes, a time interval that is too short would cause the machine to do nothing but field time-based interrupts — thereby denying service on the machine. To prevent this, the `profile` provider will silently refuse to create any probe that would result in an interval of less than two hundred microseconds.

19.6 Stability

The `profile` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	Common
Module	Unstable	Unstable	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	Common
Arguments	Evolving	Evolving	Common

CHAPTER 20

`fbt` Provider

The function is the fundamental unit of program text. In a well-designed system, the function performs a discrete and well-defined operation on a specified object or series of like objects. Most functions themselves call functions on encapsulated objects, but some functions — so-called *leaf functions* — are implemented without making any further function calls. The Function Boundary Tracing (FBT) provider contains a mechanism for instrumenting the vast majority of functions in the Solaris kernel. As a result, FBT provides the lion's share of the probes on a typical system — even on the smallest systems, FBT will provide on the order of 20,000 probes.

As with most other DTrace providers, FBT has zero probe effect when it is not explicitly enabled, and when enabled only induces a probe effect in probed functions. While the mechanism used for the implementation of FBT is highly specific to the instruction set architecture, FBT has been implemented on both SPARC™ and x86.

Because mastery of FBT requires both knowledge of the underlying instruction set architecture and understanding of the operating system implementation, it is recommended that one use FBT only when other providers are deemed insufficient. (In particular, one should take a close look at the `syscall` provider before considering use of FBT.)

20.1 Probes

FBT provides a probe at the *boundary* of most functions in the kernel. The boundary of a function is crossed by entering the function and by returning from the function; FBT thus provides two functions for every function in the kernel: one upon entry to the function, and one upon return from the function. These probes are named `entry` and `return`, respectively. The function name, and module name are specified as part of the probe. All of FBT's probes are anchored: all FBT probes specify a function name and module name.

20.2 Probe arguments

20.2.1 `entry` probes

The arguments to `entry` probes are exactly the arguments to the function. These arguments may be accessed in a typed fashion by using the `args[]` array. These arguments may be accessed as `int64_t`'s by using the `arg0` .. `arg`*n* variables.

20.2.2 `return` probes

While a given function only has a single point of entry, it may have many different points where it returns to its caller. Generally, one is interested in either the value that a function returned or the fact that the function returned at all; one is typically less interested in the specific return path taken. FBT reflects this by collecting a function's multiple return sites into a single `return` probe. Still, there may arise conditions when the exact return path *is* relevant to the understanding of a problem. Thus, for `return` probes FBT provides the *offset* (in bytes) of the returning instruction in the function text as argument zero.

If the function has a return value, it is stored in argument one. If a function does not have a return value, argument one is not defined.

20.3 Examples

FBT can be used to easily explore kernel implementation. For example, here is a script that picks up the first `ioctl`(2) that it sees from any `xclock`(1) process and follows it through the kernel:

```
/*
 * To make the output more readable, we want to indent every function entry
 * (and unindent every function return).  This is done by setting the
 * "flowindent" option.
 */
#pragma D option flowindent

syscall::ioctl:entry
/execname == "xclock" && guard++ == 0/
{
```

```
    self->traceme = 1;
    printf("fd: %d", arg0);
}

fbt:::
/self->traceme/
{}

syscall::ioctl:return
/self->traceme/
{
    self->traceme = 0;
    exit(0);
}
```

Running this script:

```
# dtrace -s ./xioctl.d
dtrace: script './xioctl.d' matched 26254 probes
CPU FUNCTION
  0  => ioctl                                    fd: 3
  0    -> ioctl
  0      -> getf
  0        -> set_active_fd
  0        <- set_active_fd
  0      <- getf
  0      -> fop_ioctl
  0        -> sock_ioctl
  0          -> strioctl
  0            -> job_control_type
  0            <- job_control_type
  0            -> strcopyout
  0              -> copyout
  0              <- copyout
  0            <- strcopyout
  0          <- strioctl
  0        <- sock_ioctl
  0      <- fop_ioctl
  0      -> releasef
  0        -> clear_active_fd
  0        <- clear_active_fd
  0        -> cv_broadcast
  0        <- cv_broadcast
  0      <- releasef
  0    <- ioctl
  0  <= ioctl
```

In this case, we can see that an `xclock` process called `ioctl` on a file descriptor that appears to be associated with a socket.

FBT may also be useful when trying to understand kernel drivers. For example, the `ssd`(7D) driver has many code paths by which `EIO` may be returned. FBT can be easily used to hone in on any of these paths that may be being taken:

```
fbt:ssd::return
/arg1 == EIO/
{
    printf("%s+%x returned EIO.", probefunc, arg0);
}
```

For more information on any one return of `EIO`, one may wish to speculatively trace all `fbt` probes, and then `commit()` (or `discard()`) based on the return value of a specific function. See Chapter 13 for details on speculative tracing.

Alternatively, one may use FBT to understand the functions called within a specified module. For example, to see all of the functions called in UFS:

```
# dtrace -n fbt:ufs::entry'{@a[probefunc] = count()}'
dtrace: description 'fbt:ufs::entry' matched 353 probes
^C
  ufs_ioctl                                                         1
  ufs_statvfs                                                       1
  ufs_readlink                                                      1
  ufs_trans_touch                                                   1
  wrip                                                              1
  ufs_dirlook                                                       1
  bmap_write                                                        1
  ufs_fsync                                                         1
  ufs_iget                                                          1
  ufs_trans_push_inode                                              1
  ufs_putpages                                                      1
  ufs_putpage                                                       1
  ufs_syncip                                                        1
  ufs_write                                                         1
  ufs_trans_write_resv                                              1
  ufs_log_amt                                                       1
  ufs_getpage_miss                                                  1
  ufs_trans_syncip                                                  1
  getinoquota                                                       1
  ufs_inode_cache_constructor                                       1
  ufs_alloc_inode                                                   1
  ufs_iget_alloced                                                  1
  ufs_iget_internal                                                 2
  ufs_reset_vnode                                                   2
  ufs_notclean                                                      2
  ufs_iupdat                                                        2
  blkatoff                                                          3
  ufs_close                                                         5
  ufs_open                                                          5
  ufs_access                                                        6
  ufs_map                                                           8
  ufs_seek                                                         11
  ufs_addmap                                                       15
  rdip                                                             15
  ufs_read                                                         15
  ufs_rwunlock                                                     16
  ufs_rwlock                                                       16
  ufs_delmap                                                       18
```

```
  ufs_getattr                                                      19
  ufs_getpage_ra                                                   24
  bmap_read                                                        25
  findextent                                                       25
  ufs_lockfs_begin                                                 27
  ufs_lookup                                                       46
  ufs_iaccess                                                      51
  ufs_imark                                                        92
  ufs_lockfs_begin_getpage                                        102
  bmap_has_holes                                                  102
  ufs_getpage                                                     102
  ufs_itimes_nolock                                               107
  ufs_lockfs_end                                                  125
  dirmangled                                                      498
  dirbadname                                                      498
```

If you know the purpose or arguments of a kernel function, you can use FBT to understand how or why it is being called. For example,`putnext`(9F) takes a pointer to a `queue`(9S) structure as its first member. The `q_qinfo` member of the `queue` structure is a pointer to a `qinit`(9S) structure. The `qi_minfo` member of the `qinit` structure has a pointer to a `module_info`(9S) structure, which contains the module name in its `mi_idname` member. We can put all of this together with the FBT probe in `putnext` to track `putnext`(9F) calls by module name:

```
fbt::putnext:entry
{
    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

Running the above:

```
# dtrace -s ./putnext.d
^C

  iprb                                                             1
  rpcmod                                                           1
  pfmod                                                            1
  timod                                                            2
  vpnmod                                                           2
  pts                                                             40
  conskbd                                                         42
  kb8042                                                          42
  tl                                                              58
  arp                                                            108
  tcp                                                            126
  ptm                                                            249
  ip                                                             313
  ptem                                                           340
  vuid2ps2                                                       361
  ttcompat                                                       412
  ldterm                                                         413
  udp                                                            569
  strwhead                                                       624
```

```
  mouse8042                                                    726
```

You can also use FBT to determine the time spent in a particular function. For example, you may be interested in callers of the DDI delaying routines: `drv_usecwait`(9F) and `delay`(9F). Here is a script to see who is calling these routines — and how long they end up delaying:

```
fbt::delay:entry,
fbt::drv_usecwait:entry
{
    self->in = timestamp
}

fbt::delay:return,
fbt::drv_usecwait:return
/self->in/
{
    @snoozers[stack()] = quantize(timestamp - self->in);
    self->in = 0;
}
```

This script is particularly interesting to run during boot. To do this, follow the procedure outlined Chapter 29. Upon reboot, one may see output like this:

```
# dtrace -ae

              ata`ata_wait+0x34
              ata`ata_id_common+0xf5
              ata`ata_disk_id+0x20
              ata`ata_drive_type+0x9a
              ata`ata_init_drive+0xa2
              ata`ata_attach+0x50
              genunix`devi_attach+0x75
              genunix`attach_node+0xb2
              genunix`i_ndi_config_node+0x97
              genunix`i_ddi_attachchild+0x4b
              genunix`devi_attach_node+0x3d
              genunix`devi_config_one+0x1d0
              genunix`ndi_devi_config_one+0xb0
              devfs`dv_find+0x125
              devfs`devfs_lookup+0x40
              genunix`fop_lookup+0x21
              genunix`lookuppnvp+0x236
              genunix`lookuppnat+0xe7
              genunix`lookupnameat+0x87
              genunix`cstatat_getvp+0x134

           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@                 4105
            8192 |@@@@                                     783
           16384 |@@@@@@@@@@@@@@@                          2793
           32768 |                                         16
```

```
    65536 |                                         0


       kb8042`kb8042_wait_poweron+0x29
       kb8042`kb8042_init+0x22
       kb8042`kb8042_attach+0xd6
       genunix`devi_attach+0x75
       genunix`attach_node+0xb2
       genunix`i_ndi_config_node+0x97
       genunix`i_ddi_attachchild+0x4b
       genunix`devi_attach_node+0x3d
       genunix`devi_config_one+0x1d0
       genunix`ndi_devi_config_one+0xb0
       genunix`resolve_pathname+0xa5
       genunix`ddi_pathname_to_dev_t+0x16
       consconfig_dacf`consconfig_load_drivers+0x14
       consconfig_dacf`dynamic_console_config+0x6c
       consconfig`consconfig+0x8
       unix`stubs_common_code+0x3b

     value  ------------- Distribution ------------- count
    262144 |                                         0
    524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      221
   1048576 |@@@@                                     29
   2097152 |                                         0


       usba`hubd_enable_all_port_power+0xed
       usba`hubd_check_ports+0x8e
       usba`usba_hubdi_attach+0x275
       usba`usba_hubdi_bind_root_hub+0x168
       uhci`uhci_attach+0x191
       genunix`devi_attach+0x75
       genunix`attach_node+0xb2
       genunix`i_ndi_config_node+0x97
       genunix`i_ddi_attachchild+0x4b
       genunix`i_ddi_attach_node_hierarchy+0x49
       genunix`attach_driver_nodes+0x49
       genunix`ddi_hold_installed_driver+0xe3
       genunix`attach_drivers+0x28

     value  ------------- Distribution ------------- count
  33554432 |                                         0
  67108864 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
 134217728 |                                         0
```

20.4 Tail-call optimization

When one function ends by calling another, the compiler can engage in *tail-call optimization* whereby the callee reuses the caller's stack frame. This is most commonly used on SPARC, where the compiler reuses the caller's register window in the callee — thereby minimizing register window pressure.

The presence of this optimization induces the `return` probe of the calling function to fire *before* the `entry` probe of the called function. This can lead to quite a bit of confusion. For example, if one wished to capture all functions called from a particular function and its callees, one may well write the following:

```
fbt::foo:entry
{
    self->traceme = 1;
}

fbt:::entry
/self->traceme/
{
    printf("called %s", probefunc);
}

fbt::foo:return
/self->traceme/
{
    self->traceme = 0;
}
```

However, if `foo()` ends in an optimized tail-call, the tail-called function (and therefore its callees) will not be captured.

Because the kernel cannot be dynamically deoptimized on the fly — and because DTrace does not wish to engage in a lie about how code is structured — tail-call optimization must be thought of as a complicating fact of life. As such, the best way to deal with tail-call optimization is to become aware of when it may used.

Tail-call optimization is likely to be used in calls like this:

```
    return (bar());
```

Or calls like this:

```
    (void) bar();
    return;
```

Conversely, a function that ends this way *cannot* have its call to `bar()` optimized, as it is not a tail-call:

```
bar();
return (rval);
```

If one suspects that a call has been tail-call optimized, check it this way:

- While running DTrace, trace `arg0` of the `return` probe in question. As discussed above, `arg0` contains the offset of the returning instruction in the function.
- After DTrace has stopped, use `mdb`(1) to look at the function. If the traced offset contains a call to another function instead of an instruction to return from the function, the call has been tail-call optimized.

For reasons of instruction set architecture, tail-call optimization is far more common on SPARC than it is on x86. Here is an example of using `mdb` to discover tail-call optimization in the kernel's `dup()` function:

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

While the above is running, run a program that performs a `dup`(2) (for example, start a `bash`(1) process). The above invocation should provide some output:

```
dup+0x10
^C
```

Now examine the function with `mdb`:

```
# echo "dup::dis" | mdb -k
dup:                            sra       %o0, 0, %o0
dup+4:                          mov       %o7, %g1
dup+8:                          clr       %o2
dup+0xc:                        clr       %o1
dup+0x10:                       call      -0x1278       <fcntl>
dup+0x14:                       mov       %g1, %o7
```

We can see that `dup+0x10` is a call to the `fcntl()` function and not a `ret` instruction. We conclude that this is an example of tail-call optimization.

20.5 Unsporting functions

Rarely, one may run into functions that seem to enter but never return or vice versa. These are generally hand-coded assembly routines that branch to the middle of another (hand-coded assembly) function to complete their tasks. While these functions are ungentlemanly, they should be reasonably harmless to analysis: the branched-to

function must still return to the caller of the branched-from function. That is, if one enables all FBT probes, one should see the entry as one function name and the return at the same stack depth as another function name. In general, this construct is regarded as inappropriate, and functions that exhibit this behavior are being eliminated as they are found — but some may persist.

20.6 Uninstrumentable functions

Some functions cannot be instrumented by FBT. The exact nature of uninstrumentable functions is specific to the instruction set architecture.

20.6.1 x86

Functions that do not create a stack frame cannot be instrumented by FBT. Because the register set for x86 is extraordinarily small, most functions — even functions that do not call other functions — must put data on the stack and must therefore create a stack frame. Still, there exists a non-trivial number of functions that do not create a stack frame (and are therefore uninstrumentable). Actual numbers vary, but typically fewer than five percent of functions are uninstrumentable on x86.

20.6.2 SPARC

On SPARC, only leaf routines hand-coded in assembly cannot be instrumented by FBT. The vast majority of the kernel is written in C, and is thus instrumentable by FBT. Actual numbers vary, but typically fewer than five percent of functions are uninstrumentable on SPARC.

20.7 Breakpoints

FBT works by dynamically modifying kernel text. Kernel breakpoints also work by modifying kernel text, giving rise to a natural conflict. If a kernel breakpoint is placed at an entry or return site *before* loading DTrace, FBT will refuse to provide a probe for the function — even if the kernel breakpoint is subsequently removed. If the kernel breakpoint is placed *after* loading DTrace, both the kernel breakpoint *and* the DTrace probe will correspond to the same point in text. (That is, the breakpoint will be hit,

and — when the debugger is continued — the probe will fire.) Because these interactions may be non-intuitive, it is recommended that kernel breakpoints not be used concurrently with DTrace — or least that they not be used concurrently for the same point in text. If breakpoints are required, consider using the DTrace `breakpoint()` action instead; see "10.4.2.1 `breakpoint()`" on page 129 for details.

20.8 Module loading

The Solaris kernel has a facility for dynamic module loading and unloading. When FBT is loaded and a module is dynamically loaded, FBT automatically provides new probes associated with the new module. If a loaded module has *unenabled* FBT probes, the module may be unloaded; the corresponding probes will be destroyed as the module is unloaded. If a loaded module has *enabled* FBT probes, the module is considered busy, and cannot be unloaded.

20.9 Stability

The FBT provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Private	Private	ISA

As FBT exposes the kernel implementation, nothing about it is Stable — and the Module and Function name and data stability are explicitly Private. The data stability for Provider and Name are Evolving, but all other data stabilities are Private: they are artifacts of the current implementation. The dependency class for FBT is ISA: while FBT is available on all current instruction set architectures, there is no guarantee that FBT will be available on arbitrary future instruction set architectures.

CHAPTER 21

syscall Provider

The syscall provider makes available a probe at the entry to and return from every system call in the system. As system calls are the primary interface between user-level applications and the operating system kernel, the syscall provider can offer tremendous insight into application behavior with respect to the system.

21.1 Probes

syscall provides a pair of probes for each system call: one named "entry" that fires before the system call is entered, and another named "return" that fires after the system call has completed but before control has transferred back to user-level. For all syscall probes, the function name is set to be the name of the instrumented system call and the module name is undefined.

The names of the system calls as provided by the syscall provider may be found in the file "/etc/name_to_sysnum." Often, the system call names provided by syscall correspond exactly to names in Section 2 of the manual (see Intro(2)). However, some probes provided by the syscall provider do not directly correspond to any documented system call. There are several possible reasons for this, the most common of which are outlined below.

21.1.1 Anachronisms

In some cases, the name of the system call as provided by the syscall provider is actually a reflection of an ancient implementation detail. For example, for reasons dating back to UNIX™ antiquity, the name of exit(2) in /etc/name_to_sysnum is "rexit." Similarly, the name of time(2) is "gtime," and the name of both execle(2) and execve(2) is "exece."

21.1.2 Subcoded System Calls

Some system calls as presented in Section 2 are actually implemented as suboperations of an undocumented system call. For example, the system calls related to System V semaphores — `semctl`(2), `semget`(2), `semids`(2), `semop`(2), and `semtimedop`(2) — are actually implemented as suboperations of a single system call, "`semsys`." The `semsys` system call takes as its first argument an implementation-specific *subcode* denoting the specific system call required: `SEMCTL`, `SEMGET`, `SEMIDS`, `SEMOP` or `SEMTIMEDOP`, respectively. As a result of overloading a single system call to implement multiple system calls, there is only a single pair of syscall probes for System V semaphores: `syscall::semsys:entry` and `syscall::semsys:return`.

21.1.3 Large File System Calls

In order for a 32-bit program to support file offsets larger than four gigabytes, it must be able to process *large files*. Because large files require use of large offsets, large files are manipulated through a parallel set of system interfaces, as described in `lf64`(5). While virtually all of these interfaces are documented in `lf64`, they do not have their own manual entry. Each of these large file system call interfaces appears as its own `syscall` probe; these interfaces are listed in Table 21–1.

TABLE 21–1 `sycall` Large File Probes

Large file `syscall` probe	System call
`creat64`	`creat`(2)
`fstat64`	`fstat`(2)
`fstatvfs64`	`fstatvfs`(2)
`getdents64`	`getdents`(2)
`getrlimit64`	`getrlimit`(2)
`lstat64`	`lstat`(2)
`mmap64`	`mmap`(2)
`open64`	`open`(2)
`pread64`	`pread`(2)
`pwrite64`	`pwrite`(2)
`setrlimit64`	`setrlimit`(2)
`stat64`	`stat`(2)
`statvfs64`	`statvfs`(2)

21.1.4 Implementation Details

Some system calls exist purely as implementation details of Solaris subsystems that span the user-kernel boundary. As such, these system calls do not have their own manual entry in Section 2. Examples of system calls in this category include the "`signotify`" system call, which is used as part of the implementation of POSIX.4 message queues, and the "`utssys`" system call, which is used to implement `fuser`(1M).

21.2 Arguments

For "`entry`" probes, the arguments (`arg0` .. `arg`*n*) are the arguments to the system call. For "`return`" probes, both `arg0` and `arg1` contain the return value. (That is, they contain the same value.) To check for system call failure in "`return`" probes, one should check for a non-zero value in the D variable "`errno`."

21.3 Stability

The `syscall` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	Common
Module	Private	Private	Unknown
Function	Unstable	Unstable	ISA
Name	Evolving	Evolving	Common
Arguments	Unstable	Unstable	ISA

CHAPTER 22

sdt Provider

Some DTrace providers — like FBT — use instrumentation methodologies that allow them to create probes where the original implementor never envisioned them. While these providers are able to offer unparalleled probe coverage, they provide little (if anything) in terms of semantic information of the probes they create. To allow for the explicit, premeditated creation of probes, DTrace has a statically defined tracing (SDT) provider. This provider creates probes at sites that the original implementor has formally designated, allowing the implementor to consciously choose the points in their code that are desired probe points, and to convey some semantic knowledge about that point with the choice of probe name.

22.1 Probes

The Solaris kernel has defined a handful of SDT probes, and will likely add more over time. Some of the current SDT probes are listed in Table 22–1. As mentioned in “22.4 Stability” on page 223, the name stability and data stability of these probes are both Private — their description here thus reflects the kernel’s implementation and should not be inferred to be an interface commitment. Still, as a practical matter, these probes are likely to continue to exist largely as described.

TABLE 22–1 SDT Probes

Probe name	Description	arg0
callout-start	Probe that fires immediately before executing a callout (see <sys/callo.h>). Callouts are executed by periodic system clock, and represent the implementation for timeout(9F)	Pointer to the callout_t (see <sys/callo.h>) corresponding to the callout to be executed.
callout-end	Probe that fires immediately after executing a callout (see <sys/callo.h>).	Pointer to the callout_t (see <sys/callo.h>) corresponding to the callout just executed.
interrupt-start	Probe that fires immediately before calling into a device's interrupt handler.	Pointer to thedev_info structure (see <sys/ddi_impldefs.h>) corresponding to the interrupting device.
interrupt-complete	Probe that fires immediately after returning from a device's interrupt handler.	Pointer to dev_info structure (see <sys/ddi_impldefs.h>) corresponding to the interrupting device.

22.2 Examples

Here is an example of a simple monitoring script to observe callout behavior on a per-second basis:

```
#pragma D option quiet

sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

Running this reveals the frequent users of timeout(9F) in the system:

```
# dtrace -s ./callout.d
                                      FUNC       COUNT
                               TS`ts_update           1
                uhci`uhci_cmd_timeout_hdlr           3
```

```
                          genunix`setrun          5
                      genunix`schedpaging         5
                          ata`ghd_timeout        10
 uhci`uhci_handle_root_hub_status_change        309

                                     FUNC      COUNT
               ip`tcp_time_wait_collector          1
                             TS`ts_update          1
               uhci`uhci_cmd_timeout_hdlr          3
                      genunix`schedpaging          4
                           genunix`setrun          8
                          ata`ghd_timeout         10
 uhci`uhci_handle_root_hub_status_change        300

                                     FUNC      COUNT
               ip`tcp_time_wait_collector          0
                        iprb`mii_portmon          1
                             TS`ts_update          1
               uhci`uhci_cmd_timeout_hdlr          3
                      genunix`schedpaging          4
                           genunix`setrun          7
                          ata`ghd_timeout         10
 uhci`uhci_handle_root_hub_status_change        300
```

The `timeout`(9F) interface does not allow for interval timers; consumers of `timeout()` requiring interval timer functionality typically reinstall their timeout from their `timeout()` handler. We can see this behavior with the following D script:

```
#pragma D option quiet

sdt:::callout-start
{
    self->callout = ((callout_t *)arg0)->c_func;
}

fbt::timeout:entry
/self->callout && arg2 <= 100/
{
    /*
     * In this case, we are most interested in interval timeout(9F)s that
     * are short.  We therefore do a linear quantization from 0 ticks to
     * 100 ticks.  The system clock's frequency — set by the variable
     * "hz" — defaults to 100, so 100 system clock ticks is one second.
     */
    @callout[self->callout] = lquantize(arg2, 0, 100);
}

sdt:::callout-end
{
    self->callout = NULL;
}

END
{
```

```
    printa("%a\n%@d\n\n", @callout);
}
```

Running this and waiting several seconds before typing ^C:

```
# dtrace -s ./interval.d
^C
genunix`schedpaging

           value  ------------- Distribution ------------- count
              24 |                                         0
              25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 20
              26 |                                         0


ata`ghd_timeout

           value  ------------- Distribution ------------- count
               9 |                                         0
              10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 51
              11 |                                         0


uhci`uhci_handle_root_hub_status_change

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1515
               2 |                                         0
```

From the above, `uhci_handle_root_hub_status_change()` in the `uhci`(7D) driver represents (by far) the shortest interval timer on the system – it is called every system clock tick. This may or may not be desirable, depending on the nature of the timeout. In any case, this view provides a better understanding of the system, and avenues for further exploration.

The `interrupt-start` probe can be used to understand interrupt activity. For example, we may want to quantize the time spent executing an interrupt handler by driver name:

```
interrupt-start
{
    self->ts = vtimestamp;
}

interrupt-complete
/self->ts/
{
    this->devi = (struct dev_info *)arg0;
    @[stringof(`devnamesp[this->devi->devi_major].dn_name),
        this->devi->devi_instance] = quantize(vtimestamp - self->ts);
}
```

Running this:

```
# dtrace -s ./intr.d
dtrace: script './intr.d' matched 2 probes
^C
 isp                                                     0
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
           32768 |                                         0

  pcf8584                                                 0
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |                                         2
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     157
             512 |@@@@@@@                                  31
            1024 |                                         3
            2048 |                                         0

  pcf8584                                                 1
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  154
            8192 |@@@@@@@@                                 37
           16384 |                                         2
           32768 |                                         0

  qlc                                                     0
           value  ------------- Distribution ------------- count
           16384 |                                         0
           32768 |@@                                       9
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  126
          131072 |@                                        5
          262144 |                                         2
          524288 |                                         0

  hme                                                     0
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |                                         6
            4096 |                                         2
            8192 |@@@@                                     89
           16384 |@@@@@@@@@@@@@@@                          262
           32768 |@                                        37
           65536 |@@@@@@@@                                 139
          131072 |@@@@@@@@@                                161
          262144 |@@@                                      73
          524288 |                                         4
         1048576 |                                         0
         2097152 |                                         1
         4194304 |                                         0

  ohci                                                    0
           value  ------------- Distribution ------------- count
```

```
         8192 |                                         0
        16384 |                                         3
        32768 |                                         1
        65536 |@@@                                      143
       131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1368
       262144 |                                         0
```

22.3 Creating SDT Probes

If you are a device driver developer, you may be interested in creating your own SDT probes in your Solaris driver. The disabled probe effect of SDT is very slight – essentially the cost of several no-operation machine instructions. You are therefore encouraged to add SDT probes liberally to your device driver, and — barring compelling performance data to the contrary – to leave those probes in your shipping code.

22.3.1 Declaring Probes

SDT probes are declared using the `DTRACE_PROBE`, `DTRACE_PROBE1`, `DTRACE_PROBE2`, `DTRACE_PROBE3` and `DTRACE_PROBE4` macros from `<sys/sdt.h>`. The module name and function name of an SDT-based probe corresponds to the kernel module and function of the probe. The name of the probe depends on the name given in the `DTRACE_PROBE`*n* macro: if the name contains no two consecutive underbars ("__"), the name of the probe is as-written in the macro. If the name contains any two consecutive underbars, the probe name converts the consecutive underbars to a single dash ("-"). For example, if a `DTRACE_PROBE` macro specifies `transaction__start`, the SDT probe will be named `transaction-start`. This allows C code to provide macro names that are not valid C identifiers without specifying a string.

When naming your SDT probes, you need not worry about name space collisions: because the module name and function name are part of the tuple identifying a probe, DTrace users will always be able to precisely specify an SDT probe of interest. This also means that you need not worry about somehow incorporating your driver name and function name in your probe name – DTrace takes care of this for you.

22.3.2 Probe Arguments

The arguments are those specified in the `DTRACE_PROBE`*n* macro. The number of arguments depends on which macro was used to create the probe: `DTRACE_PROBE1` specifies one argument, `DTRACE_PROBE2` specifies two arguments, and so on. When

declaring your SDT probes, you can minimize their disabled probe effect by neither dereferencing pointers nor loading from global variables in the probe arguments. (Both pointer dereferencing and global variable loading may be done safely in D actions that enable probes – allowing the cost of these actions to be only borne when they are explicitly required.)

22.4 Stability

The SDT provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Private	Private	ISA
Arguments	Private	Private	ISA

CHAPTER 23

sysinfo Provider

The sysinfo provider makes available probes that correspond to the "sys" kernel statistics. Because these statistics provide the input for system monitoring utilities like mpstat(1M), the sysinfo provider can allow for quick exploration of observed aberrant behavior.

23.1 Probes

The sysinfo provider makes available probes that correspond to the fields in the "sys" named kstat: a probe provided by sysinfo fires immediately before the corresponding sys value is incremented. To display both the names and the current values of the sys named kstat, one may use the kstat(1M) command. For example:

```
$ kstat -n sys
module: cpu                             instance: 0
name:   sys                             class:    misc
    bawrite                         123
    bread                           2899
    bwrite                          17995
    cpu_ticks_idle                  73743866
    cpu_ticks_kernel                2096277
    cpu_ticks_user                  1010122
    cpu_ticks_wait                  46413
...
```

The sysinfo probes are described in Table 23–1.

TABLE 23–1 `sysinfo` Probes

`bawrite`	Probe that fires whenever a buffer is about to be asynchronously written out to a device.
`bread`	Probe that fires whenever a buffer is physically read from a device. `bread` fires *after* the buffer has been requested from the device, but *before* blocking pending its completion.
`bwrite`	Probe that fires whenever a buffer is about to be written out to a device – synchronously *or* asynchronously.
`cpu_ticks_idle`	Probe that fires when the periodic system clock has made the determination that a CPU is *idle*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock; one must examine the `cpu_t` argument (`arg2`) to determine the CPU that has been deemed idle. See "23.2 Arguments" on page 228 for details.
`cpu_ticks_kernel`	Probe that fires when the periodic system clock has made the determination that a CPU is executing in the *kernel*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock; one must examine the `cpu_t` argument (`arg2`) to determine the CPU that has been deemed to be executing in the kernel. See "23.2 Arguments" on page 228 for details.
`cpu_ticks_user`	Probe that fires when the periodic system clock has made the determination that a CPU is executing in*user mode*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock; one must examine the `cpu_t` argument (`arg2`) to determine the CPU that has been deemed to be running in user-mode. See "23.2 Arguments" on page 228 for details.
`cpu_ticks_wait`	Probe that fires when the periodic system clock has made the determination that a CPU is otherwise idle, but on which some threads are *waiting for I/O*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock; one must examine the `cpu_t` argument (`arg2`) to determine the CPU that has been deemed waiting on I/O. See "23.2 Arguments" on page 228 for details.
`idlethread`	Probe that fires whenever a CPU enters the idle loop.
`intrblk`	Probe that fires whenever an interrupt thread blocks.
`inv_swtch`	Probe that fires whenever a running thread is forced to involuntarily give up the CPU.
`lread`	Probe that fires whenever a buffer is logically read from a device.
`lwrite`	Probe that fires whenever a buffer is logically written to a device
`modload`	Probe that fires whenever a kernel module is loaded.
`modunload`	Probe that fires whenever a kernel module is unloaded.

TABLE 23–1 `sysinfo` Probes *(Continued)*

`msg`	Probe that fires whenever a `msgsnd`(2) or `msgrcv`(2) system call is made, but before the message queue operations have been performed.
`mutex_adenters`	Probe that fires whenever an attempt is made to acquire an owned adaptive lock. If this probe fires, one of the `lockstat` provider's `adaptive-block` or `adaptive-spin` will also fire. See Chapter 18 for details.
`namei`	Probe that fires whenever a name lookup is attempted in the filesystem.
`nthreads`	Probe that fires whenever a thread is created.
`phread`	Probe that fires whenever a raw I/O read is about to be perfomed.
`phwrite`	Probe that fires whenever a raw I/O write is about to be performed.
`procovf`	Probe that fires whenever a new process cannot be created because the system is out of process table entries.
`pswitch`	Probe that fires whenever a CPU switches from executing one thread to executing another.
`readch`	Probe that fires after each successful read, but before control is returned to the thread performing the read. A read may occur through the `read`(2), `readv`(2) or `pread`(2) system calls. `arg0` contains the number of bytes that were successfully read.
`rw_rdfails`	Probe that fires whenever an attempt is made to read-lock a readers/writer when the lock is either held by a writer, or desired by a writer. If this probe fires, the `lockstat` provider's `rw-block` probe will also fire. See Chapter 18 for details.
`rw_wrfails`	Probe that fires whenever an attempt is made to write-lock a readers/writer lock when the lock is held – either by some number of readers or by another writer. If this probe fires, the `lockstat` provider's `rw-block` probe will also fire. See Chapter 18 for details.
`sema`	Probe that fires whenever a `semop`(2) system call is made, but before any semaphore operations have been performed.
`sysexec`	Probe that fires whenever an `exec`(2) system call is made.
`sysfork`	Probe that fires whenever a `fork`(2) system call is made.
`sysread`	Probe that fires whenever a `read`(2), `readv`(2) or `pread`(2) system call is made.
`sysvfork`	Probe that fires whenever a `vfork`(2) system call is made.
`syswrite`	Probe that fires whenever a `write`(2), `writev`(2) or `pwrite`(2) system call is made.
`trap`	Probe that fires whenever a processor trap occurs. Note that some processors (in particular, UltraSPARC variants) handle some light-weight traps through a mechanism that does not cause this probe to fire.

TABLE 23–1 `sysinfo` Probes *(Continued)*

`ufsdirblk`	Probe that fires whenever a directory block is read from the UFS file system. See `fs_ufs`(4) for details on UFS.
`ufsiget`	Probe that fires whenever an inode is retrieved. See `fs_ufs`(4)for details on UFS.
`ufsinopage`	Probe that fires after an in-core inode *without* any associated data pages has been made available for reuse. See `fs_ufs`(4) for details on UFS.
`ufsipage`	Probe that fires after an in-core inode *with* associated data pages has been made available for reuse – and therefore after the associated data pages have been flushed to disk. See `fs_ufs`(4) for details on UFS.
`wait_ticks_io`	Probe that fires when the periodic system clock has made the determination that a CPU is otherwise idle, but on which some threads are *waiting for I/O*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock; one must examine the `cpu_t` argument (`arg2`) to determine the CPU that has been deemed waiting on I/O. See "23.2 Arguments" on page 228 for details on `arg2`. Note that there is no semantic difference between `wait_ticks_io` and `cpu_ticks_io`; `wait_ticks_io` exists purely for historical reasons.
`writech`	Probe that fires after each successful write, but before control is returned to the thread performing the write. A write may occur through the `write`(2), `writev`(2) or `pwrite`(2) system calls. `arg0` contains the number of bytes that were successfully written.
`xcalls`	Probe that fires whenever a cross-call is about to be made. A cross-call is the operating system's mechanism for one CPU to request immediate work of another.

23.2 Arguments

The arguments to `sysinfo` probes are as follows:

`arg0`	The value by which the statistic is to be incremented. For most probes, this argument is always 1, but for some it may take other values.
`arg1`	A pointer to the current value of the statistic to be incremented. This value is a 64–bit quantity that will be incremented by the value in `arg0`. Dereferencing this pointer allows consumers to determine the current count of the statistic corresponding to the probe.

arg2	A pointer to the cpu_t structure that corresponds to the CPU on which the statistic is to be incremented. This structure is defined in <sys/cpuvar.h>, but it is part of the kernel implementation and should be considered Private.

As the above mentions, arg0 is always 1 for most sysinfo probes. Two exceptions to this are the readch and writech probes, for which arg0 is set to the number of bytes read or written, respectively. This allows, for example, an easy glance at the size of reads by executable name:

```
# dtrace -n readch'{@[execname] = quantize(arg0)}'
dtrace: description 'readch' matched 4 probes
^C
  xclock
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              64 |                                         0

  acroread
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
              64 |                                         0

  FvwmAuto
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@@@@@@@@                           13
               8 |@@@@@@@@@@@@@@@@@@@@@@                   21
              16 |@@@@@                                    5
              32 |                                         0

  xterm
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@                19
              64 |@@@@@@@@@@                               7
             128 |@@@@@@                                   5
             256 |                                         0

  fvwm2
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@                               186
               1 |                                         0
               2 |                                         0
               4 |@@                                       51
               8 |                                         17
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              503
              64 |                                         9
             128 |                                         0
```

```
  Xsun
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@                             269
               1 |                                         0
               2 |                                         0
               4 |                                         2
               8 |@                                        31
              16 |@@@@@                                    128
              32 |@@@@@@@                                  171
              64 |@                                        33
             128 |@@@                                      85
             256 |@                                        24
             512 |                                         8
            1024 |                                         21
            2048 |@                                        26
            4096 |                                         21
            8192 |@@@@                                     94
           16384 |                                         0
```

As described in the table above, `arg2` is a pointer to a `cpu_t`, a structure internal to the kernel implementation. Most `sysinfo` probes fire on the CPU on which the statistic is being incremented, but some (notably the `cpu_ticks_idle`, `cpu_ticks_kernel`, `cpu_ticks_user` and `cpu_ticks_wait` probes) do not. These probes always fire on the CPU executing the system clock; to determine the CPU of interest, one may examine the `cpu_id` member of the `cpu_t` structure. For example, here is a D script that runs for about ten seconds and gives a quick snapshot of relative CPU behavior on a statistic-by-statistic basis:

```
cpu_ticks_*
{
    @[probename] = lquantize(((cpu_t *)arg2)->cpu_id, 0, 1024, 1);
}

tick-1sec
/x++ >= 10/
{
    exit(0);
}
```

Running the above:

```
# dtrace -s ./tick.d
dtrace: script ’./tick.d’ matched 5 probes
CPU     ID                    FUNCTION:NAME
 22  37588                       :tick-1sec

  cpu_ticks_user
           value  ------------- Distribution ------------- count
              11 |                                         0
              12 |@@@@@@@@                                 14
              13 |@@@@                                     7
```

```
           14 |@                                        3
           15 |@                                        2
           16 |@@                                       4
           17 |@@@@@@                                   10
           18 |                                         0
           19 |@                                        2
           20 |@@@                                      6
           21 |@@@                                      5
           22 |                                         1
           23 |@@@@@@                                   10
           24 |                                         0

cpu_ticks_wait
         value  ------------- Distribution ------------- count
           11 |                                         0
           12 |@@@@@@@@@@@@@@                           241
           13 |@@@@@@@@@@@@@@                           236
           14 |                                         16
           15 |@@@@@@@@                                 132
           16 |                                         11
           17 |                                         10
           18 |                                         7
           19 |@                                        18
           20 |                                         4
           21 |                                         16
           22 |                                         13
           23 |                                         10
           24 |                                         0

cpu_ticks_kernel
         value  ------------- Distribution ------------- count
           11 |                                         0
           12 |@@@@@@@@@                                234
           13 |@@@@@                                    159
           14 |@@@                                      104
           15 |@@@@                                     131
           16 |@@                                       66
           17 |@                                        40
           18 |@                                        51
           19 |@                                        36
           20 |@@                                       56
           21 |@                                        42
           22 |@@@                                      96
           23 |@@                                       57
           24 |                                         0

cpu_ticks_idle
         value  ------------- Distribution ------------- count
           11 |                                         0
           12 |@@                                       534
           13 |@@                                       621
           14 |@@@                                      900
           15 |@@                                       758
           16 |@@@                                      942
           17 |@@@                                      963
```

```
  18 |@@@                                      965
  19 |@@@                                      967
  20 |@@@                                      957
  21 |@@@                                      960
  22 |@@@                                      913
  23 |@@@                                      946
  24 |                                         0
```

23.3 Example

Suppose one were to see the following output from `mpstat`(1M):

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 12   90  22 5760   422  299  435   26   71  116   11  1372    5  19  17  60
 13   46  18 4585   193  162  431   25   69  117   12  1039    3  17  14  66
 14   33  13 3186   405  381  397   21   58  105   10   770    2  17  11  70
 15   34  19 4769   109   78  417   23   57  115   13   962    3  14  14  69
 16   74  16 4421   437  406  448   29   77  111    8  1020    4  23  14  59
 17   51  15 4493   139  110  378   23   62  109    9   928    4  18  14  65
 18   41  14 4204   494  468  360   23   56  102    9   849    4  17  12  68
 19   37  14 4229   115   87  363   22   50  106   10   845    3  15  14  67
 20   78  17 5170   200  169  456   26   69  108    9  1119    5  21  25  49
 21   53  16 4817    78   51  394   22   56  106    9   978    4  17  22  57
 22   32  13 3474   486  463  347   22   48  106    9   769    3  17  17  63
 23   43  15 4572    59   34  361   21   46  102   10   947    4  15  22  59
```

In the above, the `xcal` field may seem aberrantly high, especially given the relative idleness of the system. `mpstat` determines the value in the `xcal` field by examining the `xcalls` field of the `sys` kstat. This aberration can thus be easily explored by enabling the `xcalls` `sysinfo` probe:

```
# dtrace -n xcalls'{@[execname] = count()}'
dtrace: description 'xcalls' matched 4 probes
^C
  dtterm                                                            1
  nsrd                                                              1
  in.mpathd                                                         2
  top                                                               3
  lockd                                                             4
  java_vm                                                          10
  ksh                                                              19
  iCald.pl6+RPATH                                                  28
  nwadmin                                                          30
  fsflush                                                          34
  nsrindexd                                                        45
  in.rlogind                                                       56
  in.routed                                                       100
  dtrace                                                          153
```

```
  rpc.rstatd                                                      246
  imapd                                                           377
  sched                                                           431
  nfsd                                                           1227
  find                                                           3767
```

We now know where to look for the source of the cross-calls: some number of `find`(1) processes are inducing the majority of them. To better understand how this is happening, we can write the following D script:

```
syscall:::entry
/execname == "find"/
{
    self->syscall = probefunc;
    self->insys = 1;
}

sysinfo:::xcalls
/execname == "find"/
{
    @[self->insys ? self->syscall : "<none>"] = count();
}

syscall:::return
/self->insys/
{
    self->insys = 0;
    self->syscall = NULL;
}
```

This script uses the `syscall` provider to attribute cross-calls from `find` to a particular system call. Some cross-calls — for example, those from page faults — may not emanate from system calls; the script prints "`<none>`" in these cases. Running the script:

```
# dtrace -s ./find.d
 dtrace: script './find.d' matched 444 probes
^C
  <none>                                                            2
  lstat64                                                        2433
  getdents64                                                    14873
```

This indicates that the vast majority of cross-calls induced by `find` are in turn induced by `getdents`(2) system calls. Further exploration would depend on what avenue we wish to explore: we may want to understand why `find` processes are making calls to `getdents` (in which case we might write a D script to aggregate on `ustack()` when `find` induces a cross-call) or we may want to understand why calls to `getdents` are inducing cross-calls (in which case we might write a D script to aggregate on `stack()` when `find` induces a cross-call). Either way, the presence of the `xcalls` probe has allowed up to quickly drill down from strange monitoring output to its root-cause.

23.4 Stability

The `sysinfo` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Private	Private	ISA

CHAPTER 24

vminfo Provider

The vminfo provider makes available probes that correspond to the "vm" kernel statistics. Because these statistics provide the input for system monitoring utilities like vmstat(1M), the vminfo provider can allow for quick exploration of observed aberrant behavior.

24.1 Probes

The vminfo provider makes available probes that correspond to the fields in the "vm" named kstat: a probe provided by vminfo fires immediately before the corresponding vm value is incremented. To display both the names and the current values of the vm named kstat, one may use the kstat(1M) command. For example:

```
$ kstat -n vm
module: cpu                             instance: 0
name:   vm                              class:    misc
        anonfree                        13
        anonpgin                        2620
        anonpgout                       13
        as_fault                        12528831
        cow_fault                       2278711
        crtime                          202.10625712
        dfree                           1328740
        execfree                        0
        execpgin                        5541
        ...
```

The vminfo probes are described in Table 24–1.

TABLE 24–1 `vminfo` Probes

`anonfree`	Probe that fires whenever an unmodified anonymous page is freed as part of paging activity. Anonymous pages are those that are not associated with a file; memory containing such pages include heap memory, stack memory or memory obtained by explicitly mapping `zero`(7D).
`anonpgin`	Probe that fires whenever an anonymous page is paged in from a swap device.
`anonpgout`	Probe that fires whenever a modified anonymous page is paged out to a swap device.
`as_fault`	Probe that fires whenever a fault is taken on a page and the fault is neither a protection fault nor a copy-on-write fault.
`cow_fault`	Probe that fires whenever a copy-on-write fault is taken on a page. `arg0` contains the number of pages that are created as a result of the copy-on-write.
`dfree`	Probe that fires whenever a page is freed as a result of paging activity. Whenever `dfree` fires, exactly one of `anonfree`, `execfree` or `fsfree` will also subsequently fire.
`execfree`	Probe that fires whenever an unmodified executable page is freed as a result of paging activity.
`execpgin`	Probe that fires whenever an executable page is paged in from the backing store.
`execpgout`	Probe that fires whenever a modified executable page is paged out to the backing store. Such as it occurs at all, most paging of executable pages will occur in terms of `execfree`; `execpgout` can only fire if an executable page is modified in memory — an uncommon occurence in most systems.
`fsfree`	Probe that fires whenever an unmodified file system data page is freed as part of paging activity.
`fspgin`	Probe that fires whenever a file system page is paged in from the backing store.
`fspgout`	Probe that fires whenever a modified file system page is paged out to the backing store.
`kernel_asflt`	Probe that fires whenever a page fault is taken by the kernel on a page in its own address space. Whenever `kernel_asflt` fires, it will be immediately preceeded by a firing of the `as_fault` probe.
`maj_fault`	Probe that fires whenever a page fault is taken that results in I/O from a backing store or swap device. Whenever `maj_fault` fires, it will be immediately preceeded by a firing of the `pgin` probe.
`pgfrec`	Probe that fires whenever a page is reclaimed off of the free page list.

TABLE 24–1 `vminfo` Probes *(Continued)*

`pgin`	Probe that fires whenever a page is paged in from the backing store or from a swap device. This differs from `maj_fault` in that `maj_fault` only fires when a page is paged in as a result of a page fault; `pgin` fires *whenever* a page is paged in — regardless of the reason.
`pgout`	Probe that fires whenever a page is paged out to the backing store or to a swap device.
`pgpgin`	Probe that fires whenever a page is paged in from the backing store or from a swap device. The only difference between `pgpgin` and `pgin` is that `pgpgin` contains the number of pages paged in as `arg0`. (`pgin` always contains 1 in `arg0`.)
`pgpgout`	Probe that fires whenever a page is paged out to the backing store or to a swap device. The only difference between `pgpgout` and `pgout` is that `pgpgout` contains the number of pages paged out as `arg0`. (`pgout` always contains 1 in `arg0`.)
`pgrec`	Probe that fires whenever a page is reclaimed.
`pgrrun`	Probe that fires whenever the pager is scheduled.
`pgswapin`	Probe that fires whenever a process is swapped in.
`pgswapout`	Probe that fires whenever a process is swapped out.
`prot_fault`	Probe that fires whenever a page fault is taken due to a protection violation.
`rev`	Probe that fires whenever the page deamon begins a new revolution through all pages.
`scan`	Probe that fires whenever the page daemon examines a page.
`softlock`	Probe that fires whenever a page is faulted as a part of placing a software lock on the page.
`swapin`	Probe that fires whenever a swapped-out process is swapped back in.
`swapout`	Probe that fires whenever a process is swapped out.
`zfod`	Probe that fires whenever a zero-filled page is created on demand.

24.2 Arguments

`arg0`	The value by which the statistic is to be incremented. For most probes, this argument is always 1, but for some it may take other values; these probes are noted in Table 24–1.

`arg1`	A pointer to the current value of the statistic to be incremented. This value is a 64–bit quantity that will be incremented by the value in `arg0`. Dereferencing this pointer allows consumers to determine the current count of the statistic corresponding to the probe.

24.3 Example

Suppose one were to see the following output from `vmstat`(1M):

```
kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 -- --   in   sy   cs us sy id
 0 1 0 1341844 836720 26 311 1644 0 0 0  0 216 0  0  0  797  817  697  9 10 81
 0 1 0 1341344 835300 238 934 1576 0 0 0 0 194 0  0  0  750 2795  791  7 14 79
 0 1 0 1340764 833668 24 165 1149 0 0 0  0 133 0  0  0  637  813  547  5  4 91
 0 1 0 1340420 833024 24 394 1002 0 0 0  0 130 0  0  0  621 2284  653 14  7 79
 0 1 0 1340068 831520 14 202 380 0 0  0  0 59  0  0  0  482 5688 1434 25  7 68
```

The "`pi`" column in the above output denotes the number of pages paged in. The `vminfo` provider makes it easy to learn more about the source of these page-ins:

```
dtrace -n pgin'{@[execname] = count()}'
dtrace: description 'pgin' matched 1 probe
^C
  xterm                                                             1
  ksh                                                               1
  ls                                                                2
  lpstat                                                            7
  sh                                                               17
  soffice                                                          39
  javaldx                                                         103
  soffice.bin                                                    3065
```

From the above, we can see that a process associated with the StarOffice™ Office Suite, "`soffice.bin`", is reponsible for most of the page-ins. To get a better picture of `soffice.bin` in terms of VM behavior, we may wish to enable all `vminfo` probes. In the following example, we run `dtrace`(1M) while launching StarOffice:

```
dtrace -P vminfo'/execname == "soffice.bin"/{@[probename] = count()}'
dtrace: description 'vminfo' matched 42 probes
^C

  kernel_asflt                                                      1
  fspgin                                                           10
  pgout                                                            16
  execfree                                                         16
  execpgout                                                        16
  fsfree                                                           16
```

```
  fspgout                                                        16
  anonfree                                                       16
  anonpgout                                                      16
  pgpgout                                                        16
  dfree                                                          16
  execpgin                                                       80
  prot_fault                                                     85
  maj_fault                                                      88
  pgin                                                           90
  pgpgin                                                         90
  cow_fault                                                     859
  zfod                                                         1619
  pgfrec                                                       8811
  pgrec                                                        8827
  as_fault                                                     9495
```

To further drill down on some of the VM behavior of StarOffice during startup, we could write the following D script:

```
vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin" && start == 0/
{
    /*
     * This is the first time that a vminfo probe has been hit; record
     * out initial timestamp.
     */
    start = timestamp;
}

vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin"/
{
    /*
     * Aggregate on the probename, and lquantize() the number of seconds
     * since our initial timestamp.  (There are 1,000,000,000 nanoseconds
     * in a second.)  We assume that the script will be terminated before
     * 60 seconds elapses.
     */
    @[probename] =
        lquantize((timestamp - start) / 1000000000, 0, 60);
}
```

We run the above while again starting StarOffice. This time, we create a new drawing, create a new presentation, and then close everything, exit StarOffice, and ^C the D script. The results are a view of some VM behavior over time:

```
# dtrace -s ./soffice.d
dtrace: script './soffice.d' matched 10 probes
^C
```

```
maj_fault
           value  ------------- Distribution ------------- count
               7 |                                         0
               8 |@@@@@@@@@@                               88
               9 |@@@@@@@@@@@@@@@@@@@@@@@@                 194
              10 |@                                        18
              11 |                                         0
              12 |                                         0
              13 |                                         2
              14 |                                         0
              15 |                                         1
              16 |@@@@@@@@@                                82
              17 |                                         0
              18 |                                         0
              19 |                                         2
              20 |                                         0

 zfod
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@                                 525
               1 |@@@@@@@@@                                605
               2 |@@                                       208
               3 |@@@                                      280
               4 |                                         4
               5 |                                         0
               6 |                                         0
               7 |                                         0
               8 |                                         44
               9 |@@                                       161
              10 |                                         2
              11 |                                         0
              12 |                                         0
              13 |                                         4
              14 |                                         0
              15 |                                         29
              16 |@@@@@@@@@@@@@@@                          1048
              17 |                                         24
              18 |                                         0
              19 |                                         0
              20 |                                         1
              21 |                                         0
              22 |                                         3
              23 |                                         0

 as_fault
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@                          4139
               1 |@@@@@@@@                                 2249
               2 |@@@@@@@@                                 2402
               3 |@                                        594
               4 |                                         56
               5 |                                         0
```

```
  6 |                                         0
  7 |                                         0
  8 |                                         189
  9 |@@                                       929
 10 |                                         39
 11 |                                         0
 12 |                                         0
 13 |                                         6
 14 |                                         0
 15 |                                         297
 16 |@@@@                                     1349
 17 |                                         24
 18 |                                         0
 19 |                                         21
 20 |                                         1
 21 |                                         0
 22 |                                         92
 23 |                                         0
```

In the above output, we can see some StarOffice behavior with respect to the VM system. For example, we see that `maj_fault` didn't fire until we started up a new application — as we would expect and hope, a warm start of StarOffice did not result in new major faults. In the `as_fault` output, we can very clearly see the initial burst of activity, the latency while the user located the menu to create a new drawing, another period of idleness, and a final burst of activity when the user clicked on a new presentation. From the `zfod` output, we can see that creating the new presentation induced significant pressure for zero-filled pages, but only for a short period of time.

The next iteration of DTrace investigation in this example would depend on the avenue we might choose for further inquiry. If we wished to understand the source of the demand for zero-filled pages, we might aggregate on `ustack()` in a `zfod` enabling. Or perhaps we would want to have some threshold for zero-filled page pressure, and when the pressure is exceeded, we could use the `stop()` destructive action to stop the offending process. This would allow us to use more traditional debugging tools like `truss`(1) or `mdb`(1) The `vminfo` provider allows for a quick mechanism to associate statistics seen in the output of conventional tools like `vmstat`(1M) with the application or applications that are inducing the systemic behavior.

24.4 Stability

The `vminfo` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Private	Private	ISA

CHAPTER 25

pid Provider

The pid provider allows for tracing of the entry and return of any function in a user process as well as any instruction as specified by an absolute address or function offset. The pid provider has no probe effect when probes are not enabled, and when enabled it only induces probe effect on those processes that are traced.

25.1 Naming pid Probes

The pid provider actually defines a *class* of providers -- each process can potentially have its own associated pid provider. A process with ID 123, for example, would be traced by using the pid123 provider. For probes from one of these providers, the module portion of the probe description refers to an object loaded in the corresponding process's address space. We can see a list of objects using mdb(1):

```
$ mdb -p 1234
Loading modules: [ ld.so.1 libc.so.1 ]
> ::objects
    BASE    LIMIT     SIZE NAME
   10000    34000    24000 /usr/bin/csh
ff3c0000 ff3e8000    28000 /lib/ld.so.1
ff350000 ff37a000    2a000 /lib/libcurses.so.1
ff200000 ff2be000    be000 /lib/libc.so.1
ff3a0000 ff3a2000     2000 /lib/libdl.so.1
ff320000 ff324000     4000 /platform/sun4u/lib/libc_psr.so.1
```

In the probe description you name the object by the name of the file, not its full path name. You may also omit the ".1" or "so.1" suffix. All of the following name the same probe:

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

The first is the actual name of the probe, the others are convenient aliases that are replaced with the full load object name internally.

For the load object of the executable itself, you can use the alias "`a.out`"; these two probe descriptions name the same probe:

```
pid123:csh:main:return
pid123:a.out:main:return
```

As with all anchored DTrace probes, the function field of the probe description names a function in the module field. A user-land binary may have several names for the same function (for example, `mutex_lock` may just be an alternate name for the function `pthread_mutex_lock` in `libc.so.1`). DTrace chooses one canonical name for those muliply named functions and will use that name internally. The following example shows how DTrace internally remaps module and function names to a canonical form:

```
# dtrace -q -n pid101267:libc:mutex_lock:entry'{ \
    printf("%s:%s:%s:%s\n", probeprov, probemod, probefunc, probename); }'
pid101267:libc.so.1:pthread_mutex_lock:entry
^C
```

This automatic renaming is not something that should overly concern you, but be aware that the names of the probes you enable may be slighly different than those actually enabled. The canonical name will always be consistent between runs of DTrace on systems running the same release of Solaris.

See Chapter 27 for examples of how to use the `pid` provider effectively.

25.2 Function Boundary Probes

The `pid` provider lets you trace function entry and return in user programs just as the FBT provider provides that capability for the kernel. Most of the examples in this guide that use the FBT provider to trace kernel function calls can be modified slightly to apply to user processes.

25.2.1 Entry Probes

An `entry` probe fires when the traced function is invoked. The arguments to entry probes are the values of the arguments to the traced function.

25.2.2 Return Probes

A `return` probes fires when the traced function returns or makes a tail call to another function. The value for `arg0` is the offset in the function of the return instruction; `arg1` holds the return value.

25.3 Function Offset Probes

The `pid` provider lets you trace any instruction in a function. For example to trace the instruction 4 bytes into a function `main()`, you can use something like this:

```
pid123:a.out:main:4
```

Every time the program executes the instruction at address `main+4`, this probe will be activated. The arguments for offset probes are undefined. To examine process state in a useful way at these probe sites, the `uregs[]` array is a good place to start. (See "27.5 `uregs[]` Array" on page 254.)

25.4 Stability

The `pid` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Private	Private	Unknown

CHAPTER 26

`fasttrap` Provider

The `fasttrap` provider allows for tracing at specific, pre-programmed user process locations. Unlike most other DTrace providers, the `fasttrap` provider is not designed for tracing system activity; rather, it is meant as a way for DTrace consumers to inject information into the DTrace framework by activating the `fasttrap` probe.

26.1 Probes

The `fasttrap` provider makes available a single probe, "`fasttrap:::fasttrap`," that fires whenever a user-level process makes a certain DTrace call into the kernel. The DTrace call to activate the probe is not publicly available; when it becomes available, you will be able to activate the `fasttrap:::fasttrap` probe from your programs using a documented function.

26.2 Stability

The `fasttrap` provider uses DTrace's stability mechanism (see Chapter 32) to describe its stabilities as follows:

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown

Element	Name stability	Data stability	Dependency class
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

CHAPTER 27

User Process Tracing

DTrace is an extremely powerful tool for understanding the behavior of user processes; it is an invaluable tool for debugging, analyzing performance problems, or simply understanding the behavior of a complex application. This chapter focuses on the DTrace facilities relevant for tracing user process activity and provides examples to illustrate their use.

27.1 `copyin()` and `copyinstr()` Subroutines

DTrace's interaction with processes is a little different than most traditional debuggers or observability tools. Many such tools appear to execute within the scope of the process itself, letting users dereference pointers to program variables directly. Rather than appearing to execute within or as part of the process itself, DTrace probes execute in the Solaris kernel. To access process data, a probe needs to use the `copyin()` or `copyinstr()` subroutines to copy user process data into the address space of the kernel.

For example, consider the `write`(2) system call:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

The following D program illustrates an *incorrect* attempt to print the contents of a string passed to the `write`(2) system call:

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

If we try to run this script, DTrace will produce error messages of the form:

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

The `arg1` variable (the value of the *buf* parameter) holds an address that refers to memory in the process executing the system call. To read the string at that address we need to use the `copyinstr()` subroutine and record its result with the `printf()` action:

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

If we run this script, we'll see what we expect: all of the strings being passed to the `write`(2) system call. Occassionally, however, we may see strange output like this:

```
  0     37                      write:entry madaï¿½ï¿½ï¿½
```

The `copyinstr()` subroutine acts on an input argument that is the user address of a null-terminated ASCII string, which need not be the case for buffers passed to the `write`(2) system call. To print only as much of the string as the caller intended, we need to use the `copyin()` subroutine which takes a size as its second argument:

```
syscall::write:entry
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Notice that we need to use the `stringof` operator so that DTrace properly converts the user data we retrieved using `copyin()` to a string (this is not necessary when using `copyinstr()` because this function always returns type `string`).

27.2 Eliminating `dtrace`(1M) Interference

If we let the previous example run for a little while, we can see that even a single call to the `write`(2) system call can cause a cascade of output. Each call to `write()` causes the `dtrace`(1M) command itself to call `write()` as it displays the output, and so on. This feedback loop is a good example of how the `dtrace` command can interfere with the desired data. We can use a simple predicate to prevent these unwanted data from being traced:

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

The `$pid` macro variable expands to the process identifier of the process that enabled the probes. The `pid` variable contains the process identifier of the process whose thread was running on the CPU where the probe was fired. Therefore the predicate `/pid != $pid/` ensures that we don't trace any events related to the running of this script itself.

27.3 `syscall` Provider

The `syscall` provider lets you trace every system call entry and return; we've seen examples of its use throughout this guide. System calls can be a good starting point for understanding a process's behavior especially if it seems to be spending a large amount of time executing or blocked in the kernel. We can use the `prstat`(1M) command to see where processes are spending time:

```
$ prstat -m -p 31337
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
 13499 user1     53  44 0.0 0.0 0.0 0.0 2.5 0.0  4K  24  9K   0 mystery/6
```

In this example, we can see that the process is consuming a large amount of system time. One possible explanation for this behavior is that the process is executing a large number of system calls. We can use a simple D program specified on the command-line to see which syscalls are happening most often:

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }'
dtrace: description 'syscall:::entry' matched 215 probes
^C

  open                                                              1
  lwp_park                                                          2
  times                                                             4
  fcntl                                                             5
  close                                                             6
  sigaction                                                         6
  read                                                             10
  ioctl                                                            14
  sigprocmask                                                     106
  write                                                          1092
```

This report gives a sense of which system calls are being called most often, in this case, the `write`(2) system call. We can use the `syscall` provider to further hone in on the source of all the `write()` system calls:

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes[arg2] = quantize(); }'
dtrace: description 'syscall::write:entry' matched 1 probe
^C

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   1037
               2 |@                                        3
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@                                        3
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |@                                        5
            2048 |                                         0
```

We can see from this report that the process is executing many `write()` system calls with a relatively small amount of data, and this could potentially be the source of the performance problem for this particular process. This example is not applicable to every situation, but it illustrates a general methodology for investigating system call behavior.

27.4 `ustack()` Action

As we try to hone in on a problem, it's often useful to see the process thread's stack at the time a particular probe is activated. The `ustack()` action traces the user thread's stack. This can be useful if, for example, a process that opens many files occassionally fails in the `open`(2) system call, we can use the `ustack()` action to discover the code path that executes the failed `open()`:

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}

syscall::open:return
/self->path && arg1 == -1/
{
    printf("open for '%s' failed", self->path);
```

```
    ustack();
}
```

This script also illustrates the use of the `$1` macro variable which takes the value of the first operand specified on the `dtrace`(1M) command-line:

```
# dtrace -s failed_open.d 31337
dtrace: script './failed_open.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0     40                      open:return open for '/usr/lib/foo' failed
              libc.so.1`__open+0x4
              libc.so.1`open+0x6c
              420b0
              tcsh`dosource+0xe0
              tcsh`execute+0x978
              tcsh`execute+0xba0
              tcsh`process+0x50c
              tcsh`main+0x1d54
              tcsh`_start+0xdc
```

Note that the `ustack()` action records program counter (PC) values for the stack and `dtrace`(1M) resolves those PC values to symbol names by looking though the process's symbol tables. If `dtrace` can't resolve the PC value to a symbol, it will print out the value as a hexadecimal (base 16) integer.

The `ustack()` action has one potential limitation that you may encounter: if a process exits or is killed before your tracing experiment is complete, `dtrace` may be unable to resolve the PC values in the stack trace to symbol names, and will be forced to display them as hexadecimal integers. Here is a D program that demonstrates how to work around the limitation:

```
/*
 * This example uses the open(2) system call probe, but this technique
 * is applicable to any script using the ustack() action where the stack
 * being traced is in a process that may exit soon.
 */
syscall::open:entry
{
    ustack();
    stop_pids[pid] = 1;
}

syscall::rexit:entry
/stop_pids[pid] != 0/
{
    printf("stopping pid %d", pid);
    stop();
    stop_pids[pid] = 0;
}
```

This script stops a process just before it exits if the `ustack()` action has been applied to a thread in that process. This ensures that the `dtrace` command will be able to resolve the PC values to symbolic names. Notice that the value of `stop_pids[pid]` is set to 0 after it has been used to clear the dynamic variable. It's important to remember to set those stopped processes running again by using the `prun`(1) command or your system will accumulate many stopped processes.

27.5 `uregs[]` Array

The `uregs[]` array allows you to access individual user registers. The indices into the `uregs[]` array come from one of the tables below depending on the architecture on which you're running DTrace.

TABLE 27–1 SPARC `uregs[]` Constants

Constant	Register
`R_G0..R_G7`	`%g0..%g7` global registers
`R_O0..R_O7`	`%o0..%o7` out registers
`R_L0..R_L7`	`%l0..%l7` local registers
`R_I0..R_I7`	`%i0..%i7` in registers
`R_CCR`	`%ccr` condition code register
`R_PC`	`%pc` program counter
`R_NPC`	`%npc` next program counter
`R_Y`	`%y` multiply/divide register
`R_ASI`	`%asi` address space identifier register
`R_FPRS`	`%fprs`floating-point registers state

TABLE 27–2 IA `uregs[]` Constants

Constant	Register
`R_GS`	`%gs`
`R_GS`	`%gs`
`R_ES`	`%es`
`R_DS`	`%ds`

TABLE 27–2 IA `uregs[]` Constants *(Continued)*

Constant	Register
R_EDI	%edi
R_ESI	%esi
R_EBP	%ebp
R_ESP	%esp
R_EBX	%ebx
R_EDX	%edx
R_ECX	%ecx
R_EAX	%eax
R_TRAPNO	%trapno
R_ERR	%err
R_EIP	%eip
R_CS	%cs
R_ERR	%cs
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

In addition, the following aliases can be used on both platforms:

TABLE 27–3 Common `uregs[]` Constants

Constant	Register
R_PC	program counter register
R_SP	stack pointer register
R_R0	first return code
R_R1	second return code

27.6 pid Provider

The pid provider allows you to trace any instruction in a process. Unlike most other providers, probes are created on demand based on the probe descriptions found in your D programs. As a result, you will not see any pid probes listed in the output of dtrace -l until you have enabled them yourself.

27.6.1 User Function Boundary Tracing

The simplest mode of operation for the pid provider is as the user-land analogue to the fbt provider (in fact, most examples using the fbt provider can be adapted for user-land by using the pid provider's entry and return probes).

The following example program traces all function entries and returns that are made from a single function. The $1 macro variable (the first operand on the command line) is the process ID for the process we want to trace, and the $2 macro variable (the second operand on the command line) is the name of the function from which we want to trace all function calls.

EXAMPLE 27–1 userfunc.d: Trace User Function Entry and Return

```
pid$1::$2:entry
{
    self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

If we save this example program in a file named userfunc.d and then chmod it to be executable, we can run it as follows:

```
# ./userfunc.d 15032 execute
dtrace: script 'from_function.d' matched 11594 probes
  0  -> execute
  0    -> execute
  0      -> Dfix
  0      <- Dfix
```

```
  0      -> s_strsave
  0        -> malloc
  0        <- malloc
  0      <- s_strsave
  0      -> set
  0        -> malloc
  0        <- malloc
  0      <- set
  0      -> set1
  0        -> tglob
  0        <- tglob
  0      <- set1
  0      -> setq
  0        -> s_strcmp
  0        <- s_strcmp
...
```

The `pid` provider can only be used on processes that are already running. You can use the `truss`(1) utility to start a new process and stop it immediately, then run your DTrace script and restart the process:

```
$ truss -f -t\!all -U a.out:main command args ...
198343/1:       -> main(0x1, 0xffbff77c, 0xffbff784, 0x22000)
```

Now in another shell start your DTrace script or commands. Then resume the stopped process:

```
$ prun 198343
```

27.6.2 Tracing Arbitrary Instructions

The `pid` provider can be used to trace any instruction in any user function. Upon demand, the `pid` provider will create a probe for every instruction in a function. The name of each probe is the offset of its corresponding instruction in the function expressed as a hexadecimal integer. For example, to enable a probe associated with the instruction at offset 0x1c in function `foo` of module `bar.so` in the process with PID 123 you can use the following command:

```
# dtrace -n pid123:bar.so:foo:1c
```

And to enable all of the probes in the function `foo`, including the probe for each instruction, you can use the command:

```
# dtrace -n pid123:bar.so:foo:
```

This is an extremely powerful facility for debugging and analyzing user-land applications. Infrequent errors can be difficult to debug because they can be difficult to reproduce. Often it's easy to identify a problem after the failure has occurred, but at that point, it's too late to reconstruct the code path that was taken along the way to

that failure condition making debugging the problem extremely difficult. The `pid` provider combined with speculative tracing (see Chapter 13) offers a convenient solution to this common problem and illustrates how to trace every instruction in a function.

EXAMPLE 27–2 errorpath.d: Trace User Function Call Error Path

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
    self->spec = 0;
}

pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

If you type in the preceding example source and save it in a file named `errorpath.d` and `chmod` it to be executable, you can run it as follows:

```
# ./errorpath.d 100461 _chdir
dtrace: script './errorpath.d' matched 19 probes
CPU     ID                    FUNCTION:NAME
  0  25253                     _chdir:entry 81e08 6d140 ffbfcb20 656c73 0
  0  25253                     _chdir:entry
  0  25269                         _chdir:0
  0  25270                         _chdir:4
  0  25271                         _chdir:8
  0  25272                         _chdir:c
  0  25273                        _chdir:10
  0  25274                        _chdir:14
  0  25275                        _chdir:18
  0  25276                        _chdir:1c
  0  25277                        _chdir:20
  0  25278                        _chdir:24
  0  25279                        _chdir:28
  0  25280                        _chdir:2c
```

```
  0  25268                     _chdir:return
```

CHAPTER 28

Security

DTrace enables visibility into all aspects of the system including user-level functions, system calls, kernel functions, and more, and it allows for powerful actions some of which can modify a program's state. Just as it would be inapproriate to allow a user access to another user's private files, a system administrator should be loathe to afford every user full access to all the facilities that DTrace offers. By default, only the super-user can use DTrace, but an administrator can allow other users controlled use of DTrace. The Least Privilege facility allows Solaris system administrators to give particular users or processes specific privileges to allow them access to individual DTrace capabilities.

28.1 Privileges

The Solaris Least Privilege facility permits administrators to grant specific privileges to specific Solaris users. To give a user a privilege on login, insert a line into `/etc/user_attr` of the form:

user-name`::::defaultpriv=basic,`*privilege*

To give a running process an additional privilege, use the `ppriv`(1) command:

`# ppriv -s A+`*privilege process-ID*

There are three privileges that control a user's access to DTrace features: `dtrace_proc`, `dtrace_user`, and `dtrace_kernel`. Each privilege permits the use of a certain set of DTrace providers, actions, and variables, and each corresponds to a particular type of use of DTrace. The privilege modes are described in detail in the

following sections; system administrators should carefully weigh each user's need against the visibility and performance impact of the different privilege modes. Users need at least one of the three DTrace privileges in order to use any of the DTrace functionality.

28.2 Privileged Use of DTrace

Users with any of the three DTrace privileges may enable probes provided by the `dtrace` provider (see Chapter 17), and may use the following actions and variables:

Providers	`dtrace`		
Actions	`exit`	`printf`	`tracemem`
	`discard`	`speculate`	
	`printa`	`trace`	
Variables	`args`	`probemod`	`this`
	`epid`	`probename`	`timestamp`
	`id`	`probeprov`	`vtimestamp`
	`probefunc`	`self`	
Address Spaces	None		

28.3 `dtrace_proc` Privilege

The `dtrace_proc` privilege permits use of the `pid` and `fasttrap` providers for process-level tracing. It also allows the use of the following actions and variables:

Providers	`pid`		
Actions	`copyin`	`copyout`	`stop`
	`copyinstr`	`raise`	`ustack`
Variables	`execname`	`pid`	`uregs`

Address Spaces	User

This privilege does not grant any visibility to Solaris kernel data structures or to processes to which the user does not have permission.

Users with this privilege may create and enable probes in processes that they own; if the user also has the `proc_owner` privilege, probes may be created and enabled in any process.[1] The `dtrace_proc` privilege is intended for users interesting in the debugging or performance analysis of user processes. It is ideal for a developer working on a new application or an engineer trying to improve an application's performance in a production environment.

The `dtrace_proc` privilege allows access to DTrace that can impose a perfomance penalty only on those processes to which the user has permission. The instrumented processes will impose more of a load on the system resources, and as such it may have some small impact on the overall system performance. Aside from this increase in overall load, this privilege does not allow any instrumentation that impacts performance for any processes other than those being traced. As this privilege grants users no additional visibility into other processes or the kernel itself, it is recommended that this privilege be granted to all users that may need to better understand the inner-workings of their own processes.

28.4 `dtrace_user` Privilege

The `dtrace_user` privilege permits use of the `profile` and `syscall` providers with some caveats, and the use of the following actions and variables:

Providers	`profile`	`syscall`	`fasttrap`
Actions	`copyin`	`copyout`•	`stop`•
	`copyinstr`	`raise`•	`ustack`
Variables	`execname`	`pid`	`uregs`
Address Spaces	User		

As with the `dtrace_proc` privilege, the `dtrace_user` privilege permits no visibility to Solaris kernel data structures or to processes to which the user does not have permission.

1 Users with the `dtrace_proc` and `proc_owner` privileges may *enable* any `pid` probe from any process, but can only create probes in processes whose privilege set is a subset of their own privilege set. Refer to the Least Privilege documentation for complete details.

The `dtrace_user` privilege provides only visibility to those processes to which the user already has permission; it does not allow any visibility into kernel state or activity. With this privilege, users may enable the `syscall` provider, but the enabled probes will only activate in processes to which the user has permission. Similarly, the `profile` provider may be enabled, but the enabled probes will only activate in processes to which the user has permission, never in the Solaris kernel.

This privilege permits the use of instrumentation that, while only allowing visibility into particular processes, can impact overall system performance. The `syscall` provider has some small performance impact on every system call for every process. The profile impacts overall system performance by executing every time interval (not unlike a real-time timer). Neither of these performance degradations is so great as to severely limit the system's progress, but system administrators should consider of the implications of granting a user this privilege. Refer to Chapter 21 and Chapter 19 for a discussion of the performance impact of the `syscall` and `profile` providers.

28.5 `dtrace_kernel` Privilege

The `dtrace_kernel` privilege permits the use of every provider except for the use of the `pid` and `fasttrap` providers on processes not owned by the user, and permits the use of all actions and variables except for kernel destructive actions (`breakpoint()`, `panic()`, `chill()`). This privilege allows for complete visibility into kernel and user state. Notice that the facilities enabled by the `dtrace_user` privilege are a strict subset of those enabled by `dtrace_kernel`.

Providers	All with above restrictions
Actions	All but destructive actions
Variables	All
Address Spaces	User Kernel

28.6 Super-user Privileges

A user with all privileges may use every providers and every action including the kernel destructive actions unavailable to every other class of user.

Providers	All	
Actions	All including destructive actions	
Variables	All	
Address Spaces	User	Kernel

CHAPTER 29

Anonymous Tracing

DTrace consumers are processes that have open file descriptors corresponding to the in-kernel DTrace framework. In general, having DTrace consumers be processes is a natural fit: the process is a well-understood entity for accounting, security, diagnosis, etc. However, there are some spaces into which processes cannot fit. Specifically, if tracing consumers can only be processes, tracing may only occur when processes can run — which is to say, tracing may not occur during boot. To most users of DTrace, this does not pose much of a limitation — most users do not care about boot once a machine has booted. To device driver developers, however, boot problems are particularly difficult to debug.

To allow for tracing during boot, DTrace provides *anonymous* tracing — tracing that is not associated with any consumer per se. Any tracing that one can do interactively one may do anonymously. However, only the super-user may create an anonymous enabling, and there may only be one anonymous enabling at any time.

29.1 Creating Anonymous State

To create an anonymous enabling, use the `-A` option to a `dtrace`(1M) invocation that specifies the desired probes, predicates, actions and options. `dtrace` will add a series of driver properties representing your request to the `dtrace`(7D) driver's configuration file — typically `/kernel/drv/dtrace.conf`. These properties will be read by the `dtrace`(7D) driver when it is loaded, and the driver will enable the specified probes with the specified actions, and create an *anonymous state* to associate with the new enabling. Normally, the `dtrace`(7D) driver is loaded on-demand (as are any drivers that act as DTrace providers). However, if one wishes to perform tracing during boot, one will want the `dtrace`(7D) driver to be loaded as early as possible. Due to this, add a `forceload` statement to `/etc/system` (see `system`(4)) for each required DTrace provider and for `dtrace`(7D) itself. For example, if one is enabling probes in `fbt`(7D), one would add the following two lines to `/etc/system`:

```
forceload: drv/fbt
forceload: drv/dtrace
```

Once these lines have been added to `/etc/system`, the system should be rebooted. When the system boots, a message will be emitted by `dtrace`(7D) to indicate that the configuration file has been succesfully processed.

Note that all options may be set with an anonymous enabling, including buffer size, dynamic variable size, speculation size, number of speculations, and so on.

29.2 Claiming Anonymous State

Once the machine has completely booted, any anonymous state may be claimed by specifying the `-a` option to `dtrace`. By default, `-a` claims the anonymous state, processes the existing data, and continues to run. If one wishes to grab the anonymous state and then exit, one should add the `-e` option.

Note that once anonymous state has been grabbed, it cannot be replaced; as with all DTrace data, once the data has been consumed from the kernel, the in-kernel buffers that contained it are reused. If one attempts to claim anonymous tracing state where none exists, `dtrace` will generate a message similar to:

```
dtrace: no anonymous tracing state
```

If drops or errors have occurred, `dtrace` will generate the appropriate messages when the anonymous state is claimed. The messages for drops and errors are the same for both anonymous and non-anonymous state.

29.3 Anonymous Tracing Examples

Here is an anonymous DTrace enabling for every probe in the `iprb`(7D) module:

```
# dtrace -A -m iprb
# echo 'forceload: drv/fbt' >>/etc/system
# echo 'forceload: drv/dtrace' >>/etc/system
# reboot
```

After rebooting, `dtrace`(7D) prints a message on the console to indicate that it is enabling the specified probes:

```
...
Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
NOTICE: enabling probe 0 (:iprb::)
NOTICE: enabling probe 1 (dtrace:::ERROR)
configuring IPv4 interfaces: iprb0.
...
```

When the machine has rebooted, the anonymous state may be grabbed by specifying the `-a` option to `dtrace`:

```
# dtrace -a
  CPU     ID                    FUNCTION:NAME
    0  22954                      _init:entry
    0  22955                     _init:return
    0  22800                  iprbprobe:entry
    0  22934           iprb_get_dev_type:entry
    0  22935          iprb_get_dev_type:return
    0  22801                  iprbprobe:return
    0  22802                  iprbattach:entry
    0  22874               iprb_getprop:entry
    0  22875              iprb_getprop:return
    0  22934           iprb_get_dev_type:entry
    0  22935          iprb_get_dev_type:return
    0  22870              iprb_self_test:entry
    0  22871             iprb_self_test:return
    0  22958             iprb_hard_reset:entry
    0  22959            iprb_hard_reset:return
    0  22862       iprb_get_eeprom_size:entry
    0  22826              iprb_shiftout:entry
    0  22828             iprb_raiseclock:entry
    0  22829            iprb_raiseclock:return
  ...
```

Here is a more interesting enabling focusing only on those functions called from `iprbattach()`:

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

Save this example in a file named `iprb.d` and then execute the following commands to clear the previous settings from the driver configuration file, install the new anonymous tracing request, and reboot:

```
# dtrace -AFs iprb.d
# reboot
```

After rebooting, `dtrace`(7D) prints a different message on the console to indicate the slightly different enabling:

```
...
Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
NOTICE: enabling probe 0 (fbt::iprbattach:entry)
NOTICE: enabling probe 1 (fbt:::)
NOTICE: enabling probe 2 (fbt::iprbattach:return)
NOTICE: enabling probe 3 (dtrace:::ERROR)
configuring IPv4 interfaces: iprb0.
...
```

After the machine has completely booted, claim the anonymous state with the `-a` option. We add the `-e` option to denote that `dtrace` should exit after processing the anonymous data:

```
# dtrace -ae
  CPU FUNCTION
    0  -> iprbattach
    0    -> gld_mac_alloc
    0      -> kmem_zalloc
    0        -> kmem_cache_alloc
    0          -> kmem_cache_alloc_debug
    0            -> verify_and_copy_pattern
    0            <- verify_and_copy_pattern
    0            -> tsc_gethrtime
    0            <- tsc_gethrtime
    0            -> getpcstack
    0            <- getpcstack
    0            -> kmem_log_enter
    0            <- kmem_log_enter
    0          <- kmem_cache_alloc_debug
    0        <- kmem_cache_alloc
    0      <- kmem_zalloc
    0    <- gld_mac_alloc
    0    -> kmem_zalloc
    0      -> kmem_alloc
    0        -> vmem_alloc
    0          -> highbit
    0          <- highbit
    0          -> lowbit
    0          <- lowbit
    0          -> vmem_xalloc
    0            -> highbit
    0            <- highbit
```

```
  0            -> lowbit
  0            <- lowbit
  0            -> segkmem_alloc
  0              -> segkmem_xalloc
  0                -> vmem_alloc
  0                  -> highbit
  0                  <- highbit
  0                  -> lowbit
  0                  <- lowbit
  0                  -> vmem_seg_alloc
  0                    -> highbit
  0                    <- highbit
  0                    -> highbit
  0                    <- highbit
  0                    -> vmem_seg_create
...
```

CHAPTER 30

Postmortem Tracing

DTrace allows for understanding complicated problems in a running system. A natural extension to DTrace is to allow it to be used to better understand problems that induce fatal system failure. To enable this, DTrace provides facilities for *postmortem* extraction and processing of the in-kernel data of DTrace consumers. That is, DTrace data may be extracted and processed from the system crash dump that typically results from fatal system failure.

By coupling these postmortem capabilities of DTrace with its ring buffering buffer policy (see Chapter 11), DTrace can be used as an operating system analog to the flight data recorder present on commercial aircraft (the so-called *black box* — even though it is bright orange). In the event of a system crash, the information that has been recorded with DTrace may provide the crucial clues to root-cause the system failure.

Recovering DTrace data from a system crash dump is straightforward, and makes use of the Solaris Modular Debugger, `mdb`(1). To extract DTrace data from a specific crash dump, one should begin by running `mdb`(1) on the dump of interest. The MDB module containing the DTrace functionality will be loaded automatically. To learn more about MDB, refer to the *Solaris Modular Debugger Guide*.

30.1 Displaying DTrace Consumers

To extract DTrace data from a DTrace consumer, one must first determine the DTrace consumer of interest. To do this, run the `::dtrace_state` MDB dcmd:

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
ccaba400     2        - <anonymous>             -
ccab9d80     3 d1d6d7e0 intrstat         cda37078
cbfb56c0     4 d71377f0 dtrace           ceb51bd0
ccabb100     5 d713b0c0 lockstat         ceb51b60
```

```
d7ac97c0     6 d713b7e8 dtrace           ceb51ab8
```

This dcmd provides as its output a table of DTrace state structures. Each row of the table consists of the following:

- The address of the state structure.
- The minor number associated with the `dtrace`(7D) device.
- The address of the process structure that corresponds to the DTrace consumer.
- The name of the DTrace consumer (or `<anonymous>` for anonymous consumers).
- The name of the file structure that corresponds to the open `dtrace`(7D) device.

If further information about a DTrace consumer is desired, the address of its process structure may be specified to the `::ps` dcmd:

```
> d71377f0::ps
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

30.2 Displaying Trace Data

Once the consumer of interest has been determined, the data corresponding to the unconsumed buffers may be processed by specifying the address of the state structure to the `::dtrace` dcmd. For example, here is the output of the `::dtrace` dcmd on an anonymous enabling of `syscall:::entry` with the action `trace(execname)`:

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cbfb7a40     2        - <anonymous>             -

> cbfb7a40::dtrace
CPU     ID                    FUNCTION:NAME
  0    344               resolvepath:entry   init
  0     16                     close:entry   init
  0    202                     xstat:entry   init
  0    202                     xstat:entry   init
  0     14                      open:entry   init
  0    206                    fxstat:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    190                    munmap:entry   init
  0    344               resolvepath:entry   init
  0    216                   memcntl:entry   init
  0     16                     close:entry   init
  0    202                     xstat:entry   init
  0     14                      open:entry   init
  0    206                    fxstat:entry   init
  0    186                      mmap:entry   init
```

```
  0    186                       mmap:entry   init
  0    186                       mmap:entry   init
  0    190                     munmap:entry   init
...
```

The `::dtrace` dcmd handles errors in the same way that `dtrace`(1M) does; if drops, errors, speculative drops, or the like were encountered, `::dtrace` will emit a message corresponding to that of `dtrace`(1M).

As with `dtrace`(1M), the order of events as displayed by `::dtrace` is always oldest to youngest within a given CPU. The CPU buffers themselves are displayed in numerical order; if an ordering is required for events on different CPUs, the `timestamp` variable should be traced.

One may display only the data for a specific CPU by specifying the `-c` option to `::dtrace`:

```
> cbfb7a40::dtrace -c 1
CPU     ID                    FUNCTION:NAME
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
  1    344                  resolvepath:entry   init
  1     16                      close:entry   init
  1    202                      xstat:entry   init
  1    202                      xstat:entry   init
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
...
```

Note that `::dtrace` only processes *in-kernel* DTrace data. Data that has been consumed from the kernel and processed (via `dtrace`(1M) or other means) will not be available to be processed with `::dtrace`. To assure that the most amount of data possible is available at the time of failure, a ring buffer buffering policy should be used. (See Chapter 11 for more information on buffer policies.)

Here is an example that creates a very small (16K) ring buffer recording all system calls and the process making them:

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

Looking at a crash dump when the above was enabled:

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cdccd400     3 d15e80a0 dtrace           ced065f0

> cdccd400::dtrace
CPU     ID                    FUNCTION:NAME
```

```
  0    139                  getmsg:return   mibiisa -r -p 25216
  0    138                   getmsg:entry   mibiisa -r -p 25216
  0    139                  getmsg:return   mibiisa -r -p 25216
  0    138                   getmsg:entry   mibiisa -r -p 25216
  0    139                  getmsg:return   mibiisa -r -p 25216
  0    138                   getmsg:entry   mibiisa -r -p 25216
  0    139                  getmsg:return   mibiisa -r -p 25216
  0    138                   getmsg:entry   mibiisa -r -p 25216
  0    139                  getmsg:return   mibiisa -r -p 25216
  0    138                   getmsg:entry   mibiisa -r -p 25216
  0     17                   close:return   mibiisa -r -p 25216
...
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     16                   close:entry   mibiisa -r -p 25216
  0     17                   close:return   mibiisa -r -p 25216
  0    124                lwp_park:entry   mibiisa -r -p 25216
  1     68                  access:entry   mdb -kw
  1     69                  access:return   mdb -kw
  1    202                   xstat:entry   mdb -kw
  1    203                   xstat:return   mdb -kw
  1     14                    open:entry   mdb -kw
  1     15                    open:return   mdb -kw
  1    206                   fxstat:entry   mdb -kw
  1    207                   fxstat:return   mdb -kw
  1    186                    mmap:entry   mdb -kw
...
  1     13                   write:return   mdb -kw
  1     10                    read:entry   mdb -kw
  1     11                    read:return   mdb -kw
  1     12                   write:entry   mdb -kw
  1     13                   write:return   mdb -kw
  1     96                    ioctl:entry   mdb -kw
  1     97                   ioctl:return   mdb -kw
  1    364                  pread64:entry   mdb -kw
  1    365                  pread64:return   mdb -kw
  1    366                  pwrite64:entry   mdb -kw
  1    367                  pwrite64:return   mdb -kw
  1    364                  pread64:entry   mdb -kw
  1    365                  pread64:return   mdb -kw
  1     38                     brk:entry   mdb -kw
  1     39                     brk:return   mdb -kw
>
```

Note that CPU 1's youngest records include a series of `write`(2) system calls by an `mdb -kw` process. This is very suspicious, as one could modify running kernel data or text with `mdb`(1) when run with the `-k` and `-w` options. In this case, the DTrace data provides at least an interesting avenue of investigation, if not the root-cause of the failure.

CHAPTER 31

Performance Considerations

As it induces additional work in the system, enabling DTrace in just about any fashion affects system performance in some way. Often, this effect is negligible – but it can become substantial if many probes are enabled with costly enablings. This chapter describes techniques for minimizing the performance effect of DTrace.

31.1 Limit Enabled Probes

Thanks to dynamic instrumentation techniques, DTrace allows unparalleled tracing coverage of both the kernel and of arbitrary user processes. While this coverage allows revolutionary new insight into system behavior, it also also allows for enormous probe effect: if tens of thousands or hundreds of thousands of probes are enabled, the effect on the system can easily be substantial. In general, one should only enable as many probes as one needs to solve a problem. One should not, for example, enable all FBT probes if a more concise enabling will answer the question. For example, is there a specific module of interest? A specific function?

If using the `pid` provider, one should be especially careful: because the `pid` provider can instrument every *instruction*, it is possible to quite literally enable millions of probes in an application – and to therefore slow the target process to a crawl.

Still, there are plenty of conditions in which very many probes *must* be enabled for a question to be answered. DTrace has been designed for this in mind; enabling a large number of probes may slow down the system quite a bit, but it will never induce fatal failure on the machine. One should therefore not hesitate to enable many probes if so required.

31.2 Use Aggregations

As discussed in Chapter 9, DTrace's aggregations allow for a scalable way of aggregating data. Associative arrays may appear to offer similar functionality to aggregations. While associative arrays are a powerful (and essential) part of the DTrace facility, they cannot – by nature of being global, general-purpose variables – offer the linear scalability of aggregations. One should therefore always prefer to use aggregations over associative arrays when possible. For example, one should generally *not* do the following:

```
syscall:::entry
{
    totals[execname]++;
}

syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

One should instead prefer the following:

```
syscall:::entry
{
    @totals[execname] = count();
}

END
{
    printa("%40s %@d\n", @totals);
}
```

31.3 Use Cacheable Predicates

In a tracing framework that offers comprehensive coverage, the framework must provide mechanism to allow events *not* to be traced — lest the user be flooded with unwanted data. As discussed in "4.3 Predicates" on page 72, DTrace does this with predicates, whereby data is only traced if a specified condition is found to be true. When enabling many probes, one tends to use predicates of a form that identifies a specific thread or threads of interest, for example "`/self->traceme/`" or "`/pid == 12345/`." Many of these predicates evaluate to the same (false) value for most threads in most probes, but the evaluation itself can become costly when done for, say, every

function entry and return point in the kernel. To reduce this cost, DTrace caches the evaluation of a predicate if it includes only thread-local variables (as in the first example) and/or immutable variables (as in the second). The cost of evaluating a cached predicate is much smaller than the cost of evaluating a non-cached predicate – especially if the predicate involves thread-local variables, string comparisons, or other relatively costly operations. While predicate caching is transparent to the user (cache coherency is maintained by DTrace), it does imply some guidelines for constructing optimal predicates:

Cacheable	Uncacheable
`self->mumble`	`mumble[curthread]`, `mumble[pid, tid]`
`execname`	`curpsinfo->pr_fname`, `curthread->t_procp->p_user.u_comm`
`pid`	`curpsinfo->pr_pid`, `curthread->t_procp->p_pipd->pid_id`
`tid`	`curlwpsinfo->pr_lwpid`, `curthread->t_tid`
`curthread`	`curthread->`*any member*, `curlwpsinfo->`*any member*, `curpsinfo->`*any member*

For example, one should generally *not* do the following:

```
syscall::read:entry
{
    follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

One should instead prefer to use thread-local variables, as in the following:

```
syscall::read:entry
{
    self->follow = 1;
}

fbt:::
/self->follow/
{}
```

```
syscall::read:return
/self->follow/
{
    self->follow = 0;
}
```

Finally, note that a predicate must consist *exclusively* of cacheable expressions in order to be cacheable. Thus, the following predicates are all cacheable:

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

But – due to their use of global variables – these predicates are all *not* cacheable:

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

CHAPTER 32

Stability

Sun often provides developers with early access to new technologies as well as observability tools that allow users to peer into the internal implementation details of user and kernel software. Unfortunately, new technologies and internal implementation details are both prone to changes as interfaces and implementations evolve and mature when software is upgraded or patched. Sun documents application and interface stability levels using a set of labels described in the `attributes`(5) man page to help set user expectations for what kinds of changes might occur in different kinds of future releases.

DTrace provides a unique challenge for Sun in this area, as there is not one stability attribute that appropriately describes the arbitrary set of entities and services that can be accessed from a D program. To meet this challenge, DTrace and the D compiler include features to dynamically compute and describe the stability levels of D programs you create. In this chapter, we discuss the DTrace features for determining program stability and help you understand how to design stable D programs. You can use the DTrace stability features to inform you of the stability attributes of your D programs, or to produce compile-time errors when your program has undesirable interface dependencies.

32.1 Stability Levels

DTrace provides two types of stability attributes for entities such as built-in variables, functions, and probes: a *stability level* and an architectural *dependency class*. The DTrace stability level assists you in making risk assessments when developing scripts and tools based on DTrace by indicating how likely an interface or DTrace entity is to change in a future release or patch. The DTrace dependency class tells you whether an interface is common to all Solaris platforms and processors, or whether the interface is associated with a particular architecture such as SPARC processors only. The two types of attributes used to describe interfaces can vary independently.

The stability values used by DTrace are listed below in order from lowest to highest stability. The more stable interfaces can be used by all D programs and layered applications because Sun will endeavor to ensure that these continue to work in future minor releases. Applications that depend only on Stable interfaces should reliably continue to function correctly on future minor releases and will not be broken by interim patches. The less stable interfaces allow experimentation, prototyping, tuning, and debugging on your current system, but should be used with the understanding that they might change incompatibly or even be dropped or replaced with alternatives in future minor releases. The Solaris Dynamic Tracing Guide will document changes to DTrace interfaces in future releases.

The DTrace stability values also help you understand the stability of the software entities you are observing, in addition to the stability of the DTrace interfaces themselves. Therefore, DTrace stability values also tell you how likely your D programs and layered tools are to require corresponding changes when you upgrade or change the software stack you are observing. Sun's versioning model and release taxonomy is described further in the `attributes`(5) man page.

Internal	The interface is private to DTrace itself and represents an implementation detail of DTrace. Internal interfaces may change in minor or micro releases.
Private	The interface is private to Sun and represents an interface developed for use by other Sun products that is not yet publicly documented for use by customers and ISVs. Private interfaces may change in minor or micro releases.
Obsolete	The interface is supported in the current release, but it scheduled to be removed, most likely in a future minor release. When support of an interface is to be discontinued, Sun will attempt to provide notification before discontinuing the interface. The D compiler may produce warning messages if you attempt to use an Obsolete interface.
External	The interface is controlled by an entity other than Sun. At Sun's discretion, Sun can deliver as part of any release updated and possibly incompatible versions of such interfaces, subject to their availability from the controlling entity. Sun makes no claims regarding either source or binary compatibility for External interfaces between any two releases. Applications based on these interfaces might not work in future releases, including patches that contain External interfaces.
Unstable	The interface is provided to give developers early access to new or rapidly changing technology or to an implementation artifact which is essential for observing or debugging system behavior for which a more stable solution is anticipated in the future. Sun makes no claims about either source or binary compatibility for Unstable interfaces from one minor release to another.

Evolving	The interface may eventually become Standard or Stable but is still in transition. Sun will make reasonable efforts to ensure compatibility with previous releases as it evolves. When non-upwards compatible changes become necessary, they will occur in minor and major releases; such changes will be avoided in micro releases whenever possible. If such a change is necessary, it will be documented in the release notes for the affected release, and when feasible, Sun will provide migration aids for binary compatibility and continued D program development.
Stable	The interface is a mature interface under Sun's control. Sun will try to avoid non-upwards-compatible changes to these interfaces, especially in minor or micro releases. If support of a Stable interface must be discontinued, Sun will attempt to provide notification and the stability level changes to Obsolete, as described above.
Standard	The interface complies with an industry standard. The corresponding documentation for the interface in the Solaris Dynamic Tracing Guide will describe the standard to which the interface conforms. Standards are typically controlled by a standards development organization, and changes can be made to the interface in accordance with approved changes the standard. This stability level can also apply to interfaces that have been adopted (without a formal standard) by an industry convention. Support is provided for only the specified version(s) of a standard; support for later versions is not guaranteed. If the standards development organization approves a non-upward-compatible change to a Standard interface that Sun decides to support, Sun will announce a compatibility and migration strategy.

32.2 Dependency Classes

Since Solaris and DTrace support a variety of operating platforms and processors, DTrace also labels interfaces with a *dependency class* that tells you whether an interface is common to all Solaris platforms and processors, or whether the interface is associated with a particular system architecture. The dependency class is orthogonal to the stability levels described earlier. For example, a DTrace interface can be Stable but only supported on SPARC microprocessors, or it can be Unstable but common to all Solaris systems. The DTrace dependency classes are described below in order from least common (that is, most specific to a particular architecture) to most common (that is, common to all architectures).

Unknown	The interface has an unknown set of architectural dependencies. DTrace does not necessarily know the architectural dependencies of all entities, such as data types defined in the operating system implementation. The Unknown label is typically applied to interfaces of very low stability for which dependencies cannot be computed. The interface may not be available when using DTrace on *any* architecture other than the one you are currently using.
CPU	The interface is specific to the CPU model of the current system. The `psrinfo`(1M) utility's `-v` option can be used to display the current CPU model and implementation name(s). Interfaces with CPU model dependencies may not be available on other CPU implementations, even if those CPUs export the same instruction set architecture (ISA). For example, a CPU-dependent interface on an UltraSPARC-III+ microprocessor may not be available on an UltraSPARC-II microprocessor, even though both processors support the SPARC instruction set.
Platform	The interface is specific to the hardware platform of the current system. A platform typically associates a set of system components and architectural characteristics such as a set of supported CPU models with a system name such as SUNW,Ultra-Enterprise-10000. The current platform name can be displayed by using the `uname`(1) `-i` option. The interface may not be available on other hardware platforms.
Group	The interface is specific to the hardware platform group of the current system. A platform group typically associates a set of platforms with related characteristics together under a single name, such as `sun4u`. The current platform group name can be displayed using the `uname`(1) `-m` option. The interface is available on other platforms in the platform group, but may not be available on hardware platforms that are not members of the group.
ISA	The interface is specific to the instruction set architecture (ISA) supported by the microprocessors on this system. The ISA describes a specification for software that can be executed on the microprocessor, including details such as assembly language instructions and registers. The native instruction sets supported by the system can be displayed using the `isainfo`(1) utility. The interface may not be supported on systems that do not export any of the same instruction sets. For example, an ISA-dependent interface on a Solaris SPARC system may not be supported on a Solaris x86 system.
Common	The interface is common to all Solaris systems, regardless of the underlying hardware. DTrace programs and layered applications that depend only on Common interfaces can be executed and deployed on other Solaris systems with the same Solaris and

DTrace revisions. The vast majority of DTrace interfaces are designed to be Common, so you can use them wherever you use Solaris.

32.3 Interface Attributes

DTrace describes interfaces using a triplet of attributes consisting of two stability levels and a dependency class. By convention, we write down the interface attributes in the following order, separated by slashes:

name-stability / data-stability / dependency-class

The *name stability* of an interface describes the stability level associated with its name as it appears in your D program or on the dtrace command-line. For example, the `execname` D variable is a Stable name: Sun guarantees this identifier will continue to be supported in your D programs according to the rules described for Stable interfaces above.

The *data stability* of an interface is distinct from the stability associated with the interface name. This stability level describes Sun's commitment to maintaining the data format(s) used by the interface and any associated data semantics. For example, the `pid` D variable is a Stable interface: process IDs are a Stable concept in Solaris, and Sun guarantees that the `pid` variable will be of type `pid_t` with the semantic that it is set to the process ID corresponding to the thread that fired a given probe in accordance with the rules described for Stable interfaces above.

The *dependency class* of an interface is distinct from its name and data stability, and describes whether the interface is specific to the current operating platform or microprocessor, as described above.

DTrace and the D compiler track the stability attributes for all of the DTrace interface entities we have seen so far, including providers, probe descriptions, D variables, D functions, types, and even your program statements themselves, as we'll see shortly. Notice that all three values can vary independently. For example, the `curthread` D variable has Stable/Private/Common attributes: the variable name is Stable and is Common to all Solaris operating platforms, but this variable provides access to a Private data format which is an artifact of the Solaris kernel implementation. Most D variables we have today are provided with Stable/Stable/Common attributes, as are the variables you define yourself.

32.4 Stability Computations and Reports

The D compiler performs stability computations for each of the probe descriptions and action statements in your D programs. You can use the `dtrace -v` option to display a report of your program's stability. Here is a simple example using a program written on the command-line:

```
# dtrace -v -n dtrace:::BEGIN'{exit(0);}'
dtrace: description 'dtrace:::BEGIN' matched 1 probe
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Stable
                Dependency Class: Common
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
```

You may also wish to combine the `dtrace -v` option with the `-e` option, which tells dtrace to compile but not execute your D program, so that you can determine program stability without having to enable any probes and execute your program. Here is another example stability report:

```
# dtrace -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Private
                Dependency Class: Common
#
```

Notice that in our new program, we have referenced the D variable `curthread`, which has a Stable name, but Private data semantics (that is, if you look at it, you are accessing Private implementation details of the kernel), and this is now reflected in the program's stability report. Stability attributes in the program report are computed by selecting the minimum stability level and class out of the corresponding values for each interface attributes triplet.

Stability attributes are computed for a probe description by taking the minimum stability attributes of all *specified* probe description fields according to the attributes published by the provider. The attributes of the available DTrace providers are shown in the chapter corresponding to each provider. DTrace providers export a stability attributes triplet for each of the four description fields for all probes published by that provider. Therefore, a provider's name may have a greater stability than the individual probes it exports. For example, the probe description:

```
fbt:::
```

indicating that DTrace should trace entry and return from all kernel functions, has greater stability than the probe description:

```
fbt:foo:bar:entry
```

which names an specific internal function `bar()` in kernel module `foo`. For simplicity, most providers use a single set of attributes for all of the individual *module*`:`*function*`:`*name* values that they publish. Providers also specify attributes for the `args[]` array, as the stability of any probe arguments vary by provider.

If the provider field itself is not specified in a probe description, then the description is assigned the stability attributes Unstable/Unstable/Common because the description may end up matching probes of providers that do not even exist yet when used on a future version of Solaris. As such, Sun is not able to provide guarantees about the future stability and behavior of this program. In general, you should always explicitly specify the provider when writing your D program clauses. In addition, any probe description fields that contain pattern matching characters (see Chapter 4) or macro variables such as `$1` (see Chapter 15) are treated as if they are unspecified because these description patterns may expand to match providers or probes released by Sun in future versions of DTrace and Solaris.

Stability attributes are computed for most D language statements by taking the minimum stability and class of the entities in the statement. For example, if we take the following attributes of D language entities:

Entity	Attributes
D built-in variable `curthread`	Stable/Private/Common
D user-defined variable `x`	Stable/Stable/Common

and we write the following D program statement:

```
x += curthread->t_pri;
```

then the resulting attributes of the statement are Stable/Private/Common, the minimum attributes associated with the operands `curthread` and `x`. In general, the stability of an expression is computed by taking the minimum stability attributes of each of the operands.

Any D variables you define in your program are automatically assigned the attributes Stable/Stable/Common. In addition, the D language grammar and D operators are implicitly assigned the attributes Stable/Stable/Common. References to kernel symbols using the backquote (`) operator are always assigned the attributes Private/Private/Unknown because they reflect implementation artifacts. Types that you define in your D program source code (specifically those that are associated with the C and D type namespaces) are assigned the attributes Stable/Stable/Common. Types that are defined in the operating system implementation and provided by other type namespaces are assigned the attributes Private/Private/Unknown. The D type cast operator yields an expression whose stability attributes are the minimum of the input expression's attributes and the attributes of the cast output type.

If you use the C preprocessor to include C system header files, these types will be associated with the C type namespace and will be assigned the attributes Stable/Stable/Common as the D compiler has no choice but to assume that you are taking responsibility for these declarations. It is therefore possible to mislead yourself about your program's stability if you use the C preprocessor to include a header file containing implementation artifacts. You should always consult the documentation corresponding to the header files you are including in order to determine the correct stability levels.

32.5 Stability Enforcement

When developing a DTrace script or layered tool, you may wish to identify the specific source of stability issues or ensure that your program has a desired set of stability attributes. You can use the `dtrace -x amin=`*attributes* option to force the D compiler to produce an error when any attributes computation results in a triplet of attributes less than the minimum values you specify on the command-line. The following example demonstrates the use of `-x amin` using a snippet of D program source. Notice that attributes are specified using three labels delimited by / in the usual order.

```
# dtrace -x amin=Evolving/Evolving/Common \
    -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
dtrace: invalid probe specifier dtrace:::BEGIN{trace(curthread->t_procp);}: \
    in action list: attributes for scalar curthread (Stable/Private/Common) \
    are less than predefined minimum
#
```

CHAPTER 33

Translators

In Chapter 32, we learned about how DTrace computes and reports program stability attributes. Ideally, we would like to construct our DTrace programs by consuming only Stable or Evolving interfaces. Unfortunately, when debugging a low-level problem or measuring system performance, you may need to enable probes that are associated with internal operating system routines such as functions in the kernel, rather than probes associated with more stable interfaces such as system calls. The data available at probe locations deep within the software stack is often a collection of implementation artifacts rather than more stable data structures such as those associated with the Solaris system call interfaces. In order to aid you in writing stable D programs, DTrace provides a facility to translate implementation artifacts into stable data structures accessible from your D program statements.

33.1 Translator Declarations

To understand the need for and use of translators, we'll consider as an example the ANSI-C standard library routines defined in `stdio.h`. These routines operate on a data structure named `FILE` whose implementation artifacts are abstracted away from C programmers. A standard technique for creating a data structure abstraction is to provide only a forward declaration of a data structure in public header files, while keeping the corresponding struct definition in a separate private header file.

If you are writing a C program and wish to know the file descriptor corresponding to a `FILE` struct, you can use the `fileno`(3C) function to obtain the descriptor rather than dereferencing a member of the FILE struct directly. The Solaris header files enforce this rule by defining FILE as an opaque forward declaration tag so it cannot be dereferenced directly by C programs that include `<stdio.h>`. Inside the `libc.so.1` library, you can imagine that `fileno()` is implemented in C something like this:

```
int
fileno(FILE *fp)
```

```
{
    struct file_impl *ip = (struct file_impl *)fp;

    return (ip->fd);
}
```

Our hypothetical `fileno()` takes a `FILE` pointer as an argument and casts it to a pointer to a corresponding internal `libc` structure, `struct file_impl`, and then returns the value of the `fd` member of the implementation structure. Why does Solaris implement interfaces like this? By abstracting the details of the current `libc` implementation away from client programs, Sun is able to maintain a commitment to strong binary compatibility while continuing to evolve and change the internal implementation details of `libc`. In our example, the `fd` member could change size or position within `struct file_impl`, even in a patch, and existing binaries calling `fileno`(3C) would not be affected by this change because they do not depend on these artifacts.

Unfortunately, observability software such as DTrace has the need to peer inside the implementation in order to provide useful results, and does not have the luxury of calling arbitrary C functions defined in Solaris libraries or in the kernel. You could declare a copy of `struct file_impl` in your D program in order to instrument the routines declared in `stdio.h`, but then your D program would rely on Private implementation artifacts of the library that might break in a future micro or minor release, or even in a patch. Ideally, we want to provide a construct for use in D programs that is bound to the implementation of the library and is updated accordingly, but still provides an additional layer of abstraction associated with greater stability.

A *translator* is a collection of D assignment statements provided by the supplier of an interface that can be used to translate an input expression into an object of struct type. A new translator is created using a declaration of the form:

```
translator output-type < input-type input-identifier > {
    member-name = expression ;
    member-name = expression ;
    ...
};
```

The *output-type* names a struct that will be the result type for the translation. The *input-type* specifies the type of the input expression, and is surrounded in angle brackets < > and followed by an *input-identifier* that can be used in the translator expressions as an alias for the input expression. The body of the translator is surrounded in braces { } and terminated with a semicolon (;), and consists of a list of *member-name* and identifiers corresponding translation expressions. Each member declaration must name a unique member of the *output-type* and must be assigned an expression of a type compatible with the member type, according to the rules for the D assignment (=) operator.

For example, we could define a struct of stable information about `stdio` files based on some of the available `libc` interfaces:

```
struct file_info {
    int file_fd;   /* file descriptor from fileno(3C) */
    int file_eof;  /* eof flag from feof(3C) */
};
```

A hypothetical D translator from `FILE` to `file_info` could then be declared in D as follows:

```
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

In our hypothetical translator, the input expression is of type `FILE *` and is assigned the *input-identifier* `F`. The identifier `F` can then be used in the translator member expressions as a variable of type `FILE *` that is only visible within the body of the translator declaration. To determine the value of the output `file_fd` member, the translator performs a cast and dereference similar to the hypothetical implementation of `fileno`(3C) shown above. A similar translation is performed to obtain the value of the EOF indicator.

Sun provides a set of translators for use with Solaris interfaces that you can invoke from your D programs, and promises to maintain these translators according to the rules for interface stability defined earlier as the implementation of the corresponding interface changes. We'll learn about these translators later in the chapter, after we learn how to invoke translators from D. The translator facility itself is also provided for use by application and library developers who wish to offer their own translators that D programmers can use to observe the state of their software packages.

33.2 Translate Operator

The D operator `xlate` is used to perform a translation from an input expression to one of the defined translation output structures. The `xlate` operator is used in an expression of the form:

`xlate <` *output-type* `> (` *input-expression* `)`

For example, to invoke the hypothetical translator for FILE structs defined above and access the `file_fd` member, you would write the expression:

```
xlate <struct file_info *>(f)->file_fd;
```

where `f` is a D variable of type `FILE *`. The `xlate` expression itself is assigned the type defined by the *output-type*. Once a translator is defined, it can be used to translate input expressions to either the translator output struct type, or to a pointer to that struct.

If you translate an input expression to a struct, you can either dereference a particular member of the output immediately using the " . " operator, or you can assign the entire translated struct to another D variable to make a copy of the values of all the members. If you dereference a single member, the D compiler will only generate code corresponding to the expression for that member. You may not apply the `&` operator to a translated struct to obtain its address, as the data object itself does not exist until it is copied or one of its members is referenced.

If you translate an input expression to a pointer to a struct, you can either dereference a particular member of the output immediately using the `->` operator, or you can dereference the pointer using the unary `*` operator, in which case the result behaves as if you translated the expression to a struct. If you dereference a single member, the D compiler will only generate code corresponding to the expression for that member. You may not assigned a translated pointer to another D variable as the data object itself does not exist until it is copied or one of its members is referenced, and therefore cannot be addressed.

A translator declaration may omit expressions for one or more members of the output type. If an `xlate` expression is used to access a member for which no translation expression is defined, the D compiler will produce an appropriate error message and abort the program compilation. If the entire output type is copied by means of a structure assignment, any members for which no translation expressions are defined will be filled with zeroes.

In order to find a matching translator for an `xlate` operation, the D compiler examines the set of available translators in the following order:

- First, the compiler looks for a translation from the exact input expression type to the exact output type.
- Second, the compiler *resolves* the input and output types by following any typedef aliases to the underlying type names, and then looks for a translation from the resolved input type to the resolved output type.
- Third, the compiler looks for a translation from a compatible input type to the resolved output type. The compiler uses the same rules as it does for determining compatibility of function call arguments with function prototypes in order to determine if an input expression type is compatible with a translator's input type.

If no matching translator can be found according to these rules, the D compiler produces an appropriate error message and program compilation fails.

33.3 Process Model Translators

The DTrace library file `/usr/lib/dtrace/procfs.d` provides a set of translators for use in your D programs to translate from the operating system kernel implementation structures for processes and threads to the stable `proc`(4) structures `psinfo` and `lwpsinfo`. These structures are also used in the Solaris `/proc` filesystem files `/proc/`*pid*`/psinfo` and `/proc/`*pid*`/lwps/`*lwpid*`/lwpsinfo`, and are defined in the system header file `/usr/include/sys/procfs.h`. These structures define useful Stable information about processes and threads such as the process ID, LWP ID, initial arguments, and other data displayed by the `ps`(1) command. Refer to `proc`(4) for a complete description of the struct members and semantics.

TABLE 33–1 `procfs.d` Translators

Input Type	Input Type Attributes	Output Type	Output Type Attributes
`proc_t *`	Private/Private/Common	`psinfo_t *`	Stable/Stable/Common
`kthread_t *`	Private/Private/Common	`lwpsinfo_t *`	Stable/Stable/Common

33.4 Stable Translations

While a translator provides the ability to convert information into a stable data structure, it does not necessarily resolve all stability issues that can arise in translating data. For example, if the input expression for an xlate operation itself references Unstable data, the resulting D program is also Unstable because program stability is always computed as the minimum stability of the accumulated D program statements and expressions. Therefore, it is sometimes necessary to define a specific stable input expression for a translator in order to permit stable programs to be constructed. The D inline mechanism can be used to facilitate such *stable translations*.

The DTrace `procfs.d` library provides the `curlwpsinfo` and `curpsinfo` variables described earlier as stable translations. For example, the `curlwpsinfo` variable is actually an `inline` declared as follows:

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

The `curlwpsinfo` variable is defined as an inlined translation from the `curthread` variable, a pointer to the kernel's Private data structure representing a thread, to the Stable `lwpsinfo_t` type. The D compiler processes this library file and caches the `inline` declaration, making `curlwpsinfo` appear as any other D variable. The

`#pragma` statement following the declaration is used to explicitly reset the attributes of the `curlwpsinfo` identifier to Stable/Stable/Common, masking the reference to `curthread` in the inlined expression. This combination of D features permits D programmers to use curthread as the source of a translation in a safe fashion that can be updated by Sun coincident to corresponding changes in the Solaris implementation.

CHAPTER 34

Versioning

In Chapter 32, we learned about the DTrace features for determining the stability attributes of D programs that you create. Once you have created a D program with the appropriate stability attributes, you may also wish to bind this program to a particular *version* of the D programming interface. The D interface version is a label applied to a particular set of types, variables, functions, constants, and translators made available to you by the D compiler. If you specify a binding to a specific version of the D programming interface, you ensure that you can recompile your program on future versions of DTrace without encountering conflicts between program identifiers that you define and identifiers defined in future versions of the D programming interface. You should establish version bindings for any D programs that you wish to install as persistent scripts (see Chapter 15) or use in layered tools.

34.1 Versions and Releases

The D compiler labels sets of types, variables, functions, constants, and translators corresponding to a particular software release using a *version string*. A version string is a period-delimited sequence of decimal integers of the form "*x*" (a Major release), "*x.y*" (a Minor release), or "*x.y.z*" (a Micro release). Versions are compared by comparing the integers from left to right. If the leftmost integers are not equal, the string with the greater integer is the greater (and therefore more recent) version. If the leftmost integers are equal, the comparison proceeds to the next integer in order from left to right to determine the result. All unspecified integers in a version string are interpreted as having the value zero during a version comparison.

The DTrace version strings correspond to Sun's standard nomenclature for interface versions, as described in `attributes`(5). A change in the D programming interface is accompanied by a new version string. The following table summarizes the version strings used by DTrace and the likely significance of the corresponding DTrace software release.

TABLE 34–1 DTrace Release Versions

Release	Version	Significance
Major	*x*.0	A Major release is likely to contain major feature additions; adhere to different, possibly incompatible Standard revisions; and though unlikely, could change, drop, or replace Standard or Stable interfaces (see Chapter 32). The initial version of the D programming interface is labeled as version 1.0.
Minor	*x.y*	Compared to an *x*.0 or earlier version (where y is not equal to zero), a new Minor release is likely to contain minor feature additions, compatible Standard and Stable interfaces, possibly incompatible Evolving interfaces, or likely incompatible Unstable interfaces. These changes may include new built-in D types, variables, functions, constants, and translators. In addition, a Minor release may remove support for interfaces previously labeled as Obsolete (see Chapter 32).
Micro	*x.y.z*	Micro releases are intended to be interface compatible with the previous release (where z is not equal to zero), but are likely to include bug fixes, performance enhancements, and support for additional hardware.

In general, each new version of the D programming interface will provide a superset of the capabilities offered by the previous version, with the exception of any Obsolete interfaces that have been removed.

34.2 Versioning Options

By default, any D programs you compile using `dtrace -s` or specify using the `dtrace -P`, `-m`, `-f`, `-n`, or `-i` command-line options are bound to the most recent D programming interface version offered by the D compiler. You can determine the current D programming interface version using the `dtrace -V` option:

```
$ dtrace -V
dtrace: Sun D 1.0
$
```

If you wish to establish a binding to a specific version of the D programming interface, you can set the `version` option to an appropriate version string. Similar to other DTrace options (see Chapter 16), you can set the version option either on the command-line using `dtrace -x`:

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

or you can use the `#pragma D option` syntax to set the option in your D program source file:

```
#pragma D option version=1.0

BEGIN
{
    trace("hello");
}
```

If you use the `#pragma D option` syntax to request a version binding, you must place this directive at the top of your D program file prior to any other declarations and probe clauses. If the version binding argument is not a valid version string or refers to a version not offered by the D compiler, an appropriate error message will be produced and compilation will fail. You can therefore also use the version binding facility to cause execution of a D script on an *older* version of DTrace to fail with an obvious error message.

Prior to compiling your program declarations and clauses, the D compiler loads the set of D types, functions, constants, and translators for the appropriate interface version into the compiler namespaces. Therefore, any version binding options you specify simply control the set of identifiers, types, and translators that are visible to your program in addition to the variables, types, and translators that your program defines. Version binding prevents the D compiler from loading newer interfaces that may define identifiers or translators that conflict with declarations in your program source code and would therefore cause a compilation error. See "2.1 Identifier Names and Keywords" on page 43 for tips on how to pick identifier names that are unlikely to conflict with interfaces offered by future versions of DTrace.

34.3 Provider Versioning

Unlike interfaces offered by the D compiler, interfaces offered by DTrace providers (that is, probes and probe arguments) are not affected by or associated with the D programming interface or the previously described version binding options. The available provider interfaces are established as part of loading your compiled instrumentation into the DTrace software in the operating system kernel and vary depending on your instruction set architecture, operating platform, processor, the software installed on your Solaris system, and your current security privileges. The D compiler and DTrace runtime examine the probes described in your D program clauses and report appropriate error messages when probes requested by your D program are not available. These features are orthogonal to the D programming interface version because DTrace providers do not export interfaces that can conflict with definitions in your D programs; that is, you can only enable probes in D, you cannot define them, and probe names are kept in a separate namespace from other D program identifiers.

DTrace providers are delivered with a particular release of Solaris and are described in the corresponding version of the Solaris Dynamic Tracing Guide. The chapter of this guide corresponding to each provider will also describe any relevant changes to or new features offered by a given provider. You can use the `dtrace -l` option to explore the set of providers and probes available on your Solaris system. Providers label their interfaces using the DTrace stability attributes, and you can use the DTrace stability reporting features (see Chapter 32) to determine whether the provider interfaces used by your D program are likely to change or be offered in future Solaris releases.

Glossary

DTrace A dynamic tracing facility that provides concise answers to arbitrary questions.