Oracle® Solaris 11.3 DTrace (Dynamic Tracing) Guide

ORACLE®

Oracle Solaris 11.3 DTrace (Dynamic Tracing) Guide

Part No: E53395

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Accessibilité de la documentation

Pour plus d'informations sur l'engagement d'Oracle pour l'accessibilité à la documentation, visitez le site Web Oracle Accessibility Program, à l'adresse http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info ou le site http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs si vous êtes malentendant.

Contents

Contents

Using This Documentation

- **Overview** – Describes how to use the Dtrace facility and Dtrace providers in an Oracle Solaris system.
- **Audience** – Oracle Solaris developers and Administrators.
- **Required knowledge** – Advanced C programming skills and basic knowledge about Unix and Oracle Solaris internals.

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

1

About DTrace

DTrace is a comprehensive dynamic tracing facility that is built into Oracle Solaris. Administrators and developers can use DTrace on live production systems to examine the behavior of user programs and operating system. DTrace enables 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. DTrace enables you to create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions that you can formulate using the DTrace D programming language.

DTrace allows all Oracle Solaris users to perform the following activities:

- 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

Familiarity with a programming language such as C or a scripting language such as `awk` or `perl` will enable you to learn DTrace and the D programming language faster.

Getting Started

DTrace helps you understand a software application by enabling 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 programmable sensors scattered all over your Oracle 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 gathers the data from your probes and reports it back to you. If you do not specify any actions for a probe, DTrace will just take note of each time the probe fires.

Every probe in DTrace has an unique integer ID and a probe description. For more information on the fields of a probe description see "Providers and Probes" on page 19.

You can 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(utility's -n option to enable a probe using its string name. Type the following command:

```
# dtrace -n BEGIN
```

The output shows that a probe was enabled and the BEGIN probe was fired. Once you see this output, dtrace remains paused waiting for other probes to fire. Since no other probes are enabled and BEGIN only fires once, press Control-C in your shell to exit 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 displays that the probe named BEGIN fired once and both its name and integer ID, 1, are printed. 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 requests can be constructed using arbitrary numbers of probes and actions. Create a simple request using two probes by adding the END probe to the previous example command. The END probe fires once when tracing is completed. Type the following command, and then again press 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
#
```

Pressing Control-C to exit dtrace triggers the END probe. dtrace reports this probe firing before exiting. In addition to constructing DTrace experiments on the command line, you can also write them in text files using the D programming language. In a text editor, create a file called hello.d and type in your first D program:

EXAMPLE 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 the following command:

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

`dtrace` printed the same output as before followed by the text "hello, world". Unlike the previous example, you did not have to wait and press `Control-C`, either. These changes were the result of the actions you specified for your `BEGIN` probe in `hello.d`.

Each D program consists of a series of `clauses`, each clause 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 (;). Your 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 in Chapter 4, "Actions and Subroutines".

By now, if you are familiar with the C programming language, you have probably realized from the name and examples that D programming language is 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. You will learn more about the D programming features in subsequent chapters. You will understand all of the syntax by the end of this chapter. But first, take a step back from language rules and learn more about how DTrace works.

Providers and Probes

In the preceding examples, you learnt to use two simple probes named `BEGIN` and `END`. DTrace probes come from a set of kernel modules called providers, each of which performs 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 might take some time to display all of the output. To count up all your probes, type the command:

```
# dtrace -l | wc -l
        30122
```

You might observe a different total on your system, 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. In fact, this output is not the complete list because, 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. The output of the `dtrace -l` command displays the probes. Notice that each probe has an integer ID and a probe description. A probe description is composed of four parts shown as separate columns in the `dtrace` output. The four parts of a probe description 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 name 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 description is a name that gives you some idea of the probe's semantic meaning, such as `BEGIN` or `END`.
	This name can be referenced in a D program by using the built-in variable probename.

When writing a probe description, write all four parts of the description separated by colons.

```
provider:module:function:name
```

Notice that some of the probes in the output do not have values for module and function, such as the `BEGIN` and `END` probes 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. A probe that has a module and function as part of its name is known as an *anchored probe*, and one that does not is known as *unanchored*.

By convention, if you do not specify all of the fields of a probe description, 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 you used the probe name BEGIN earlier, you 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 you now know that the true name of the BEGIN probe is dtrace::: BEGIN, which indicates that this probe is provided by the DTrace framework itself and is not anchored to any function. Therefore, the hello.d program could have been written as follows and would produce the same result:

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

2

D Programming Language

D is a systems programming language that allows you to interface with the operating system APIs (application programming interface) and with the hardware. This chapter describes the overall structure of a D program and features for constructing probe descriptions that match more than one probe. You can also use the C preprocessor `cpp` with D programs.

D 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 and definitions of new types.

Probe Clauses and Declarations

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

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

The predicate and list of action statements may be omitted. Any directives found outside probe clauses are referred to as *declarations*. Declarations must be made 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. For more information, see "Variables" on page 52. You can use declarations to define types in D. For more

information, see "Type and Constant Definitions" on page 84. 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. For example, D pragmas are used to set run-time DTrace options. For more information, see Chapter 10, "Options and Tunables".

Probe Descriptions

Every D program clause begins with a list of one or more probe descriptions, each taking the following 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.

You must 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 do not specify the provider, you might obtain unexpected results if multiple providers publish probes with the same name. Similarly, future versions of DTrace might 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 table lists the special characters that are recognized in probe name patterns.

TABLE 1   Probe Name Pattern Matching Characters

Symbol	Description
*	Matches any string, including the null string.
?	Matches any single character.

Symbol	Description
[...]	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 the patterns on the command line with the `dtrace -l` command. For example, the command `dtrace -l -f kmem_*` lists 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);
}
```

You can use the above close to enable probe ID `12345`, as reported by the `dtrace -l -i 12345` command. You must always write D programs using human-readable probe descriptions. Integer probe IDs are not guaranteed to remain consistent as DTrace provider kernel modules are loaded and unloaded or following a reboot.

Predicates

Predicates are expressions enclosed in slashes `/ /` that are evaluated at probe firing time to determine whether the associated actions should be executed. 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.

Actions

Probe actions are described by a list of statements separated by semicolons (;) and enclosed in braces { }. If you only want 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.

Order of Execution

Each clause is represented by its predicate, if any, and the clause's actions. When an enabled probe fires, its actions will execute if the predicate evaluates to true or if no predicate is given. Program order determines the order in which actions are executed. Two or more clauses that enable the same probe will also execute in program order.

Use of the `C` Preprocessor

The C programming language used for defining system interfaces in Oracle Solaris 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` 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 type definitions such as types used in your own C programs. You can also use the preprocessor to perform tasks such as 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. The D compiler cannot parse C header files that include additional program elements like C function source code and will produce an appropriate error message.

Compilation and Instrumentation

When you write traditional programs in Oracle Solaris, you use a compiler to convert your program from source code into object code that you can execute. Running the `dtrace` command invokes the complier for the D language. Dtrace sends the compiled program to the operating system kernel for execution. 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 your system does not experience any kind of performance degradation 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. No effective difference exists 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 enable 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 the commands 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 that 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 Oracle 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 there is a mistake in the program, Dtrace reports the error and disables the instrumentation. You can rectify the program and try again. The DTrace error reporting and debugging features are described later in this book.

The following diagram shows the different components of the DTrace architecture, including providers, probes, the DTrace kernel software, and the dtrace command.

FIGURE 1   Overview of the DTrace Architecture and Components
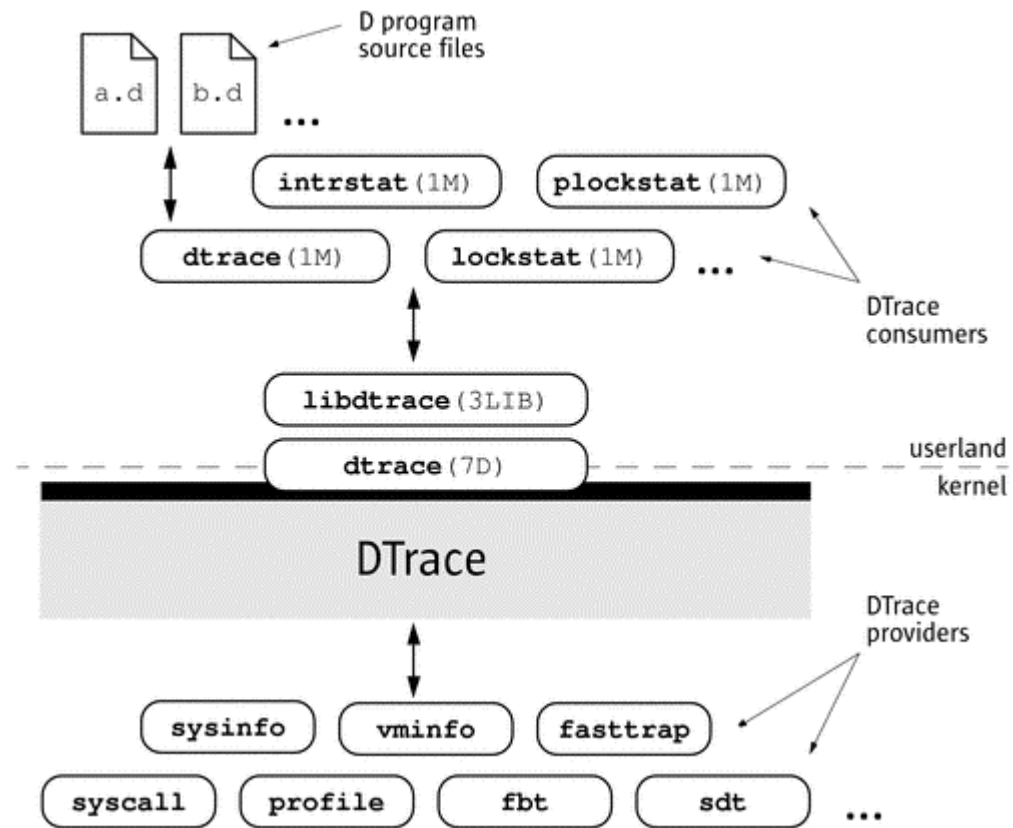
Variables and Arithmetic Expressions

The next example program makes use of the DTrace `profile` provider to implement a simple time-based counter. The `profile` provider is able to create 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 creates a probe that fires every *n* seconds. Type the following source code 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 press 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 the 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 a 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 a 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 the program, so subsequent references must conform to the same type as the initial assignment. In counter.d, the variable i is first assigned the integer constant zero, so its type is set to int. D program 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, described in "Types, Operators, and Expressions" on page 40. The D programming language 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.

The central part of `counter.d` is the probe clause that increments the 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 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; the complete list is found in "Types, Operators, and Expressions" on page 40. 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 you 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.

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.

EXAMPLE 2   countdown.d

Type the following source code 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!
#
```

This example uses the BEGIN probe to initialize an integer i to 10 to begin the countdown. Next, as in the previous example, the program uses the tick-1sec probe to implement a timer that fires once per second. Notice that in countdown.d, the tick-1sec probe description is used 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 the 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 "Types, Operators, and Expressions" on page 40. If i is not yet zero, the script traces i and then decrements it by one using the - operator.

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

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

Similar to the first example, hello.d, countdown.d uses 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, 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 want to write complex programs using predicates, try to first visualize your algorithm in this manner, and then transform each path of your conditional constructs into a separate clause and predicate.

Now combine predicates with a new provider, the syscall provider, and create the first real D tracing program. The syscall provider permits you to enable probes on entry to or return from any Oracle Solaris system call. The next example uses 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 you'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 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 the shell that was printed in response to your echo command.

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

Notice that the body of rw.d's probe clause is left empty because the program is only intended to trace notification of probe firings and not to trace 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 command. As you type, you should see dtrace report probe firings in your first window, similar to the following example:

```
# 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
...
```

The read and write system calls read a character from your terminal window and echo back the result. This example includes many of the concepts described so far and a few new ones as well. First, to instrument read and write in the same manner, the script uses a single probe clause with multiple probe descriptions by separating the descriptions with commas like this:

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

For readability, each probe description appears on its own line. This arrangement is not strictly required, but it makes for a more readable script. Next the script defines a predicate that matches only those system calls that are executed by your shell process:

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

The 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
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 you have 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, you can make one more simple change and turn `rw.d` into a very simple version of a system call tracing tool like `truss`. 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.

Output Formatting

System call tracing is a powerful way to observe the behavior of most user processes. If you have used the Oracle Solaris `truss` utility before as an administrator or developer, you have probably learned that it is a useful tool to keep around for whenever there is a problem. If you

have 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` followed by its one line of output at the end. The following example improves upon the earlier `rw.d` program by formatting its output to look more like `truss` so you can more easily understand the output. Type the following program and save it in a file called `trussrw.d`:

EXAMPLE 3   `trussrw.d`: Trace System Calls with `truss` Output Format

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
        printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}

syscall::read:return,
syscall::write:return
/pid == $1/
{
        printf("tt = %dn", arg1);
}
```

In this example, the constant `12345` is replaced with the label `$1` in each predicate. This label allows you to specify the process of interest as an *argument* to the script: `$1` is replaced by the value of the first argument when the script is compiled. To execute `trussrw.d`, use the `dtrace` options `-q` and `-s`, followed by the process ID of your shell as the final argument. The `-q` option indicates that `dtrace` should be quiet and suppress the header line and the CPU and ID columns shown in the preceding examples. As a result, you will only see the output for the data that you explicitly traced. Type the following command (replacing `12345` with the process ID of a shell process) and then press return a few times in the specified shell:

```
# dtrace -q -s trussrw.d 12345
                        = 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 examine the D program and its output in more detail. First, a clause similar to the earlier program instruments each of the shell's calls to `read` and `write`. But for this example, a new function, `printf`, is used to trace data and print it out in a specific format:

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
        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 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` utility. If you've never seen `printf` before, the formats and options are explained in detail in Chapter 6, "Output Formatting". 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 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, 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`. For more information about probe arguments, see "Variables" on page 52. The first argument to *read(2)* and *write(2)* is a file descriptor, printed in decimal. The second argument is a buffer address, formatted as a hexadecimal value. The final argument is the buffer size, formatted as a decimal value. The format specifier `%4d` is used 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 to align the output.

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

```
syscall::read:return,
syscall::write:return
/pid == $1/
{
        printf("tt = %dn", 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. The return value is formatted 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 "Types, Operators, and Expressions" on page 40.

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*. You can use any type in D that you can in C. For more information about the different kinds of data types, see "Types, Operators, and Expressions" on page 40. 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 use an associative array in a D program. Type the following program and save it in a file named rwtime.d:

EXAMPLE 4   rwtime.d: Time read and write Calls

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
ts[probefunc] = timestamp;
}

syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
printf("%d nsecs", timestamp - ts[probefunc]);
}
```

As with trussrw.d, specify the ID of shell process when you execute rwtime.d. If you type a few shell commands, you can see the amount of time elapsed during each system call. Type the following command and press return a few times in your other shell:

```
# dtrace -s rwtime.d `pgrep -n

bash`
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, you must instrument both the entry to and return from read and write and sample the time at each point. Then, on return from a given system call, you must compute the difference between the first and second timestamp. You could use separate variables for each system call, but this would make the program annoying to extend to additional system calls. Instead, it is 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 == $1/
{
        ts[probefunc] = timestamp;
}
```

This clause defines an array named ts and assigns the appropriate member the value of the DTrace variable timestamp. This variable returns the value of an always-incrementing nanosecond counter, similar to the Oracle Solaris library routine gethrtime(3C). Once the entry timestamp is saved, the corresponding return probe samples timestamp again and reports the difference between the current time and the saved value:

```
syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
        printf("%d nsecs", timestamp - ts[probefunc]);
}
```

The predicate on the return probe requires that DTrace is 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 DTrace first starts. If your shell is already waiting in a read system call for input when you execute dtrace, the read:return probe will fire without a preceding read:entry for this first read and ts[*probefunc*] will evaluate to zero because it has not yet been assigned.

External Symbols and Types

DTrace instrumentation executes inside the Oracle 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 enable advanced DTrace users, administrators, service personnel, and driver developers to examine low-level behavior of the operating system kernel and device drivers.

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 Oracle Solaris kernel contains a C declaration of a system tunable named `kmem_flags` for enabling memory allocator debugging features. For more information about `kmem_flags`, see *Oracle Solaris 11.3 Tunable Parameters Reference Manual*. 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 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.

The following chapters describe the complete set of rules for D and demonstrate how DTrace can make complex performance measurements and functional analysis of the system easy. Later, you will see how to use DTrace to connect user application behavior to system behavior, giving you the capability to analyze your entire software stack.

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. This chapter describes the detailed set of rules for types, operators, and expressions.

Identifier Names and Keywords

D identifier names are composed of upper case 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. You should avoid using such

names in your D programs. By convention, D programmers 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 should not be used for the names of D variables.

TABLE 2   D Keywords

auto*	goto*	sizeof
break*	if*	static*
case*	import*+	string+
char	inline	stringof+
const	int	struct
continue*	long	switch*
counter*+	offsetof+	this+
default*	probe*+	translator+
do*	provider*+	typedef
double	register*	union
else*	restrict*	unsigned
enum	return+	void
extern	self+	volatile
float	short	while*
for*	signed	xlate+

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 "+". D provides the complete set of types and operators found in ANSI-C. The major difference in D programming is the absence of control-flow constructs. Keywords associated with control-flow in ANSI-C are reserved for future use in D.

Data Types and Sizes

D provides fundamental data types for integers and floating-point constants. Arithmetic can be performed only on integers in D programs. Floating-point constants can be used to initialize data structures, but you cannot perform floating-point arithmetic 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 use the `isainfo -b` command to determine the native data model of your system.

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

TABLE 3   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 type aliases listed in the following table.

TABLE 4   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 Table 3, "D Integer Data Types," on page 42 and are appropriately defined for each data model. For example, the type name `uint8_t` is an alias for the type unsigned `char`. For more information about how to define your own type aliases, see "Type and Constant Definitions" on page 84.

Note - The predefined type aliases cannot be used in files included by the preprocessor.

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. You can use the floating-point types listed in the following table.

TABLE 5   `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 "Strings" on page 70.

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 special escape sequences shown in the following table in your character constants. D supports the escape sequences found in ANSI-C.

TABLE 6   `D` Character Escape Sequences

\a	alert	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\0oo	octal value 0_oo_
\t	horizontal tab	\xhh	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 endianness 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 in Table 6, "`D` Character Escape Sequences," on page 44. Similar to ANSI-C, strings are represented as arrays of characters terminated by a null character (`\0`) that 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. For more information, see "Strings" on page 70.

Arithmetic Operators

D provides the following binary arithmetic operators. The arithmetic operators have the same meaning in both D and ANSCI-C.

TABLE 7   `D` Binary Arithmetic Operators

+	integer addition

-	integer subtraction
*	integer multiplication
/	integer division
%	integer modulus

Arithmetic in D can be performed only on integer operands, or on pointers, as discussed in "Pointers and Arrays" on page 62. 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 overflow and underflow can occur.

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. Errors detected by DTrace have no effect on other DTrace users or on the operating system kernel.

In addition to these binary operators, the + and - operators may also be used as unary operators as well; these operators have higher precedence than any of the binary arithmetic operators. The order of precedence and associativity properties for all the D operators is presented in Table 8, "D Relational Operators," on page 45. You can control precedence by grouping expressions in parentheses ( ).

Relational Operators

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

TABLE 8   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
>=	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` 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. You can use enumeration to create named integer constants. For more information, see "Type and Constant Definitions" on page 84.

Logical Operators

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

TABLE 9   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.

Bitwise Operators

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

TABLE 10   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 the compiler 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.

Assignment Operators

D provides the following binary assignment operators for modifying D variables. 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 11   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
*=	multiply 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
>>=	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 shorthand 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.

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 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 code fragments 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 code fragments produce identical results:

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

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. Pointers and pointer arithmetic in D are discussed in "Pointers and Arrays" on page 62.

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 enable 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 whether 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 returns 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 path returns an integer. The second and third expressions also may not invoke a tracing function

such as `trace` or `printf`. If you want to conditionally trace data, use a predicate instead, as discussed in "Predicates " on page 25.

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, the signed type 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 in "Strings" on page 70. 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. Integers are cast to types of lower rank by zeroing the excess high-order bits of the integer.

Because 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.

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 12   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

The following operators are discussed in subsequent chapters:

- `sizeof` – Computes the size of an object
- `offsetof` – Computes the offset of a type member
- `stringof` – Converts the operand to a string
- `xlate` – Translates a data type
- unary `&` – Computes the address of an object
- unary `*` – Dereferences a pointer to an object
- `-> and .` – Accesses a member of a structure or union type

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 right most expression. This operator is provided strictly for compatibility with C and should generally not be used.

The () entry in the table of operator precedence represents a function call; examples of calls to functions such as `printf` and `trace` are presented in Chapter 6, "Output Formatting". 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 the pair of expressions `i` and `i++`, in these contexts.

The [] entry in the table of operator precedence represents an array or associative array reference. Examples of associative arrays are presented in "Associative Arrays" on page 53. A special kind of associative array called an *aggregation* is described in Chapter 3, "Aggregations". The [] operator can also be used to index into fixed-size C arrays as well, as described in "Pointers and Arrays" on page 62.

Variables

D provides two basic types of variables for use in your tracing programs: scalar variables and associative arrays. This chapter explores the rules for D variables in more detail and how variables can be associated with different scopes. A special kind of array variable, called an *aggregation*, is discussed in Chapter 3, "Aggregations".

- Scalar variables and associative arrays have a global scope and are not multi-processor safe (MP-safe). It means that the value of these variables can be changed by more than one processor and thus there are chances that the variable can became corrupt.
- Aggregations are MP-safe even though they have a global scope.

This section covers the following types of variables:

- "Scalar Variables" on page 52
- "Associative Arrays" on page 53
- "Thread-Local Variables" on page 55
- "Clause-Local Variables" on page 57
- "Built-in Variables" on page 59
- "External Variables" on page 61

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 of objects as well as composite structures. DTrace also represents strings as fixed-size scalars by permitting them to grow up to a predefined maximum length. For or e information about control over string length, see "Strings" on page 70.

Scalar variables are created automatically the first time you assign a value to a previously undefined identifier in your D program. 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 following example. 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 by DTrace before you first reference the variable.

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. 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 10, "Options and Tunables".

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 in your D program that you can use to create more complex control flows.

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. Because `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 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. For more information, see "Types, Operators, and Expressions" on page 40.

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, an associative array `x` is defined as follows:

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

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. For more information about arithmetic conversion rules, see "Type Conversions" on page 50.

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. Accessing 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 behavior 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. Always assign zero to associative array elements that are no longer in use. For other techniques to eliminate dynamic variable drops, see Chapter 10, "Options and Tunables".

Thread-Local Variables

In DTrace, you can declare a variable storage that is local to each operating system thread. Thread-local variables are useful in situations where you want to enable a probe and mark every thread that fires the probe with some tag or other data. Creating a program to solve this problem 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` system call and associates a thread-local variable named read with each thread that fires the probe. Similar to global variables, thread-local variables are created 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 zeros. As with associative array elements, underlying storage

is not allocated for a thread-local variable until a non-zero value is assigned to it. Also as with associative array elements, assigning zero to a thread-local variable causes DTrace to deallocate the underlying storage. Always assign zero to thread-local variables that are no longer in use. For other techniques to fine-tune the dynamic variable space, see Chapter 10, "Options and Tunables".

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

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

Like any D variable, you are not required 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. The following example shows how to use thread-local variables.

EXAMPLE   5   rtime.d: Compute Time Spent in read(2)

In a text editor, type in the following program and save it in a file named rtime.d.

```
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);
        /*
         * Done with the thread-local variable; assign zero to it to
         * allow the DTrace runtime to reclaim the underlying storage.
         */
        self->t = 0;
}
```

Go to your shell and type the following command to see a similar output.

```
# 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
#
```

`rtime.d` uses a thread-local variable named to capture a timestamp on entry to `read` by any thread. Then, in the return clause, the program prints out the amount of time spent in `read` by subtracting `self->t` from the current timestamp. The built-in D variables `pid` and `tid` report the process ID and thread ID of the thread performing the `read`. Because `self->t` is no longer needed once this information is reported, it is then assigned 0 to allow DTrace to reuse the underlying storage associated with t for the current thread.

Typically you will see many lines of output without even doing anything because, behind the scenes, server processes and daemons are executing `read` all the time. Try changing the second clause of `rtime.d` to use the `execname` variable to print out the name of the process performing a `read` to learn more:

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

If you find a process that is of particular interest, add a predicate to learn more about its `read` behavior:

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

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 are created 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 zeros. Note that if your program contains multiple clauses for a single probe, any clause-local variables will remain intact as the clauses are executed, as shown in Example 6, "clause.d: Clause-local Variables," on page 58.

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. 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. You cannot rely on the storage location 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. Therefore, 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.

EXAMPLE 6   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
{
       /*
        * 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 program 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
```

Built-in Variables

The following table provides a complete list of D built-in variables. 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 13   DTrace Built-in Variables

`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 index, but each element is defined to be the type corresponding to the given probe argument. For example consider the `start` probe in the `io` provider. In this probe, `args[0]` is of type `bufinfo_t struct`, `args[1]` is of type `devinfo_t struct` and `args[2]` is of type `fileinfo_t struct`.
`uintptr_t caller`	The program counter location that called the current kernel thread, at the time the probe fired.
`uintptr_t ucaller`	The program counter location that called the current user-level thread, at the time the probe fired.
`chipid_t chip`	The CPU chip identifier for the current physical chip. See "sched Provider" on page 266 for more information.
`processorid_t cpu`	The CPU identifier for the current CPU. See "sched Provider" on page 266 for more information.
`cpuinfo_t *curcpu`	The CPU information for the current CPU. See "sched Provider" on page 266 for more information.
`lwpsinfo_t *curlwpsinfo`	The lightweight process (LWP) state of the LWP associated with the current thread. This structure is described in further detail in the `proc` man page.
`psinfo_t *curpsinfo`	The process state of the process associated with the current thread. This structure is described in further detail in the `proc` man page.
`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 variable and other operating system data structures.
`string cwd`	The name of the current working directory of the process associated with the current thread.
`uint_t epid`	The enabled probe ID (EPID) for the current probe. This integer uniquely identifies 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` to execute the current process.
`gid_t gid`	The real group ID of the current process.
`uint_t id`	The probe ID for the current probe. This ID 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 Oracle Solaris operating system kernel.
`lgrp_id_t lgrp`	The latency group ID for the latency group of which the current CPU is a member. See "sched Provider" on page 266 for more information.
`pid_t pid`	The process ID of the current process.
`pid_t ppid`	The parent 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.
`psetid_t pset`	The processor set ID for the processor set containing the current CPU. See "sched Provider" on page 266 for more information.
`string root`	The name of the root directory of the process associated with the current thread.
`string kthreadname`	The kernel thread name associated with the currently executing thread.
`string uthreadname`	The user thread name associated with the currently executing thread.
`uint_t stackdepth`	The current thread's stack frame depth at probe firing time.
`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 *pthread_self(3C)*.
`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.
`uid_t uid`	The real user ID of the current process.
`uint64_t uregs[]`	The current thread's saved user-mode register values at probe firing time. For information about the use of the `uregs[]` array, see Chapter 12, "User Process Tracing".
`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.
`uint64_t walltimestamp`	The current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.
`string zonename`	Name of the zone as specified on creation of the process.
`zoneid_t zoneid`	The zone identifier.

For information about functions built into the D language such as `trace`, see Chapter 4, "Actions and Subroutines".

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. For more information about `kmem_flags`, see the *Oracle Solaris 11.3 Tunable Parameters Reference Manual*. 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 following D notation:

```
`kmem_flags
```

DTrace associates each kernel symbol with the type used for the symbol in the corresponding operating system C code, providing 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 depend on these details might 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. For more information about the stability of your D programs, see Chapter 18, "Stability".

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. Because the Oracle Solaris kernel supports dynamically loaded modules with separate symbol namespaces, the same variable name might be 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.

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. This chapter describes 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. Also discussed are issues relating to the use of pointers in different address spaces.

Note - 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. For information about features and issues specific to DTrace, see "Pointers to DTrace Objects" on page 69 and "Pointers and Addresses" on page 63.

Pointers and Addresses

The Oracle 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 of memory resources is referred to as an *address space*. An address space associates a range of address values with a set of translations that the operating system and hardware use to convert each virtual address to a corresponding physical memory location. The range of address values are[0 ... 0xffffffff] for a 32-bit address space and [0 ... 0xffffffffffffffff] for a 64-bit address space. 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 an asterisk (*) to the type name to indicate you want to declare a pointer type. For example:

```
int *p;
```

The above declaration declares a D global variable named `p` that is a pointer to an integer. This declaration 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. Because 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 -b` command to determine the number of bits used for pointers by the active operating system kernel. For more information about the `isainfo` command, see `isainfo(1)`.

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. 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(`kmem_flags);
trace(*p);
```

The left-hand fragment creates a D global variable pointer p. Because 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.

Pointer Safety

If you are a C or C++ programmer, you know that misuse of pointers in your programs can cause your programs to crash. DTrace is a robust, safe environment for executing your D programs where these mistakes cannot cause program crashes. 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 probably know that the Java language does not support pointers 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 the run-time environment for the Java programming language that detects a programming error and reports an exception back to you.

To see DTrace's error handling and reporting, write a deliberately bad D program using pointers.

EXAMPLE 7   badptr.d: Demonstration of DTrace Error Handling

In an editor, type the following D program and save it in a file named badptr.d.

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

The badptr.d program creates a D pointer named x that is a pointer to int. The program assigns 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. The program uses a cast expression to convert NULL to be a pointer

to an integer. The program then dereferences the pointer using the expression `*x`, and assigns the result to another variable `y`, and then attempts to trace `y`. When the D program is executed, DTrace detects an invalid pointer access when the statement `y = *x` is executed and reports the error.

```
# 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`. Because `int` is 4 bytes and 2 is not a multiple of 4, the expression `*x` results in a DTrace alignment error.

For information about the DTrace error mechanism, see “ERROR Probe” on page 183.

Array Declarations and Storage

D provides support for *scalar arrays* in addition to the dynamic associative arrays described in “Variables” on page 52 . 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*, but these are sometimes needed when accessing existing operating system array data structures that are declared in C. Aggregations are described in Chapter 3, “Aggregations”.

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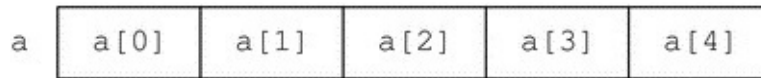
The D expression `a[0]` is used to refer to the first array element, `a[1]` refers to the second, and so on. From a syntactic perspective, scalar arrays and associative arrays are very similar. You can declare an associative array of five integers referenced by an integer key.

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

You can 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 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 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.

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]`. The expression `p[2]` traces the value of the third array element (index 2). Because `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.

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 turn contains the location of the array. This difference is illustrated in the following diagram.

FIGURE 3   Pointer and Array Storage
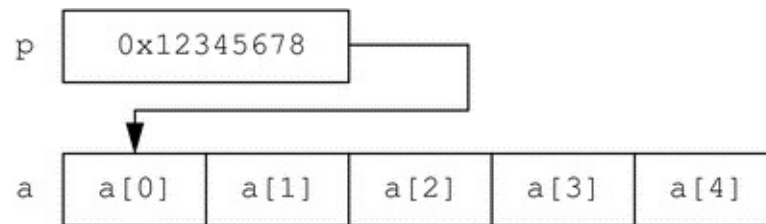
This difference is manifested in the D syntax if you attempt to assign pointers and scalar arrays. If `x` and `y` are pointer variables, the expression `x = y` is legal; it simply copies the pointer address in `y` to the storage location named by `x`. If `x` and `y` are scalar array variables, the expression `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]`.

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.

EXAMPLE 8   illustration of Pointer Arithmetic

```
int *x;
```

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

This fragment creates an integer pointer x and then trace its value, its value incremented by one, and its value incremented by two. If you create and execute this program, DTrace reports 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 , 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. 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);
}
```

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.

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.

Pointers to DTrace Objects

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 can 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 can only call functions defined by the DTrace environment by name. For more information, see Chapter 4, "Actions and Subroutines". Indirect function calls using pointers are not permitted in D.

Pointers and Address Spaces

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 Oracle 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 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 to the user address space pointer. For more information about functions that can be applied to the user address space pointer, see Chapter 4, "Actions and Subroutines".To avoid confusion, ensure that you name and comment variables that store user addresses. You can also store user addresses as uintptr_t so you do not accidentally compile D code that dereferences them. For information about techniques for using DTrace on user processes, see Chapter 12, "User Process Tracing".

Strings

DTrace provides support for tracing and manipulating strings. This chapter describes the complete set of D language features for declaring and manipulating strings. Unlike ANSI-C, strings in D have their own built-in type and operator support so you can easily and unambiguously use them in your tracing programs.

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. For more information about tunable DTrace options, see Chapter 10, "Options and Tunables". 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. In the D statement, `trace(s);`, if `s` is of type `char *`, DTrace will trace the value of the pointer `s` (that is, it will trace an integer address value).

In the D statement, `trace(*s);`, by definition of the `*` operator, the D compiler will dereference the pointer `s` and trace the single character at that location.

You can 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. In the D statement, `trace(s);`, if `s` is of type string, the string type indicates to the D compiler that 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`. For more information, see "String Comparison " on page 73.

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 Table 6, "D Character Escape Sequences," on page 44.

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. You can create a new variable of type `string` by assigning it an expression of type `string`. The D statement, `s = "hello";`, creates 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).

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

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 to 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.

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. 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 DTrace safety features will prevent you from damaging the system or DTrace, but you might end up tracing a sequence of undecipherable characters.

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`. For more information, see "String Assignment" on page 72. All of the relational operators can be used to compare strings:

TABLE 14   `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. The following table lists example D string comparisons and their results.

TABLE 15   String Comparison Examples

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

Structs and Unions

Collections of related variables can be grouped together into composite data objects called *structs* and *unions*. You can define these objects in D by creating new type definitions for them. You can use your new types for any D variables, including associative array values. This chapter explores the syntax and semantics for creating and manipulating these composite types and the D operators that interact with them. The syntax for structs and unions is illustrated using several example programs that demonstrate the use of the DTrace function boundary tracing (`fbt`) and `pid` providers.

Structs

The D keyword `struct,` short for *structure,* is used to introduce a new type composed of a group of other types. The `struct` type can be used as the type for D variables and arrays, enabling 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 D struct as a class, but one with data members only and no methods.

Suppose you want to create a more sophisticated system call tracing program in D that records a number of things about each `read` and `write` system call executed by your shell, such as the elapsed time, number of calls, and the largest byte count passed as an argument. You could write a D clause to record these properties in three separate associative arrays as shown in the following example:

```
int maxbytes;    /* declare maxbytes */

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

However, this clause 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. Use a `struct` type to conserve space and make the program easier to read and maintain. For example, declare a `struct` type at the top of the program source file as follows:

```
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 used to refer back to the 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 (;).

The struct declaration itself simply defines the new type; it does not create any variables or allocate any storage in DTrace. Once declared, you can use `struct callinfo` as a type throughout the remainder of your D program, and each variable of type `struct callinfo` will store a copy of the four variables described by the 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.

You 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

The following example is an improved program using the new structure type.

EXAMPLE 9   `rwinfo.d`: Gather `read(2)` and `write(2)` Statistics

In a text editor type the following program and save as `rwinfo.d`.

```
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 == $1/
{
        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 == $1/
{
        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, run `dtrace -q -s rwinfo.d`, specifying one of your shell processes. Then go type in a few commands in your shell and, 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 `pgrep -n

bash`
^C
       calls  max bytes  elapsed nsecs
------  -----  ---------  -------------
  read     36       1024  3588283144
 write     35         59  14945541
#
```

Pointers to Structs

Referring to structs using pointers is very common in C and D. You can use the operator `->` 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 Oracle Solaris LWP is the kernel's representation of a user thread, upon which the Oracle 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/lwpinfo`. The `/proc` structures are defined in the system header file `<sys/procfs.h>` and are used by observability and debugging tools such as ps, pgrep, and truss. For more information, see ps(1), pgrep(1), and truss(1), and proc(4) man page. The follow tables lists example expressions using `curpsinfo`, their types, and their meanings:

TABLE 16   Example Expressions Using `curpsinfo`

Expression	Type
`curpsinfo->pr_pid`	`pid_t`

Expression	Type
`curpsinfo->pr_fname`	`char []`
`curpsinfo->pr_psargs`	`char []`

You should review the complete structure definition later by examining the `<sys/procfs.h>` header file. For more information, see proc(4). The next example uses the `pr_psargs` member to identify a process of interest by matching command-line arguments.

Structs are used 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 Oracle Solaris operating system kernel and its system interfaces. In addition to using the aforementioned `curpsinfo` struct, the next example examines some kernel structs as well by observing 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. The next example illustrates 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)*.

Use the `strings` 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. For more information, see the strings(1) man page.

In an editor, type in the first clause of the example script 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);
}
```

This first clause uses the expression `curpsinfo->pr_psargs` to access and match the command-line arguments of the *strings(1)* command so that the script selects the correct *read(2)* requests

before tracing 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, the D compiler infers 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` executes, you will see output from DTrace similar to the following example:

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

This example can be extended using a common D programming technique to follow a thread from this initial `read` request deeper into the kernel. Upon entry to the kernel in `syscall::read:entry`, the next script sets a thread-local flag variable indicating this thread is of interest, and clears this flag on `syscall::read:return`. Once the flag is set, it can be used 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 in the file `ksyms.d`:

EXAMPLE 10   ksyms.d: Trace `read(2)` and `uiomove(9F)` Relationship

```
/*
 * When 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 the example uses the thread-local variable `self->watched` to identify when a kernel thread of interest enters the DDI routine uiomove(9F). Once there, the script uses 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. A copy of this pointer is saved for use in a clause in the clause-local variable `this->iov`. In the final statement, the script dereferences `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. These values should 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 similar to the following example:

```
# 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` match the parameters passed to `uiomove` by the `ksyms` driver.

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 Oracle Solaris `kstat` framework defines a struct containing a union that is used in the following example 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. The framework is used to implement utilities such as `mpstat` and `iostat`. 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 */
};
```

The examined declaration is shortened 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.

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` 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. Press Control-C in your shell to abort `mpstat` after a few seconds. To observe counter sampling, enable a probe that fires each time the `mpstat` command calls the `kstat_data_lookup(3KSTAT)` function in `libkstat`. To do so, make use of a new DTrace provider: `pid`. The `pid` provider enables you to dynamically create probes in user processes at C symbol locations such as function entry points. You can ask the `pid` provider to create a probe at a user function entry and return sites by writing probe descriptions of the form:

```
pidprocess-ID:object-name:function-name:entry
```

```
pidprocess-ID:object-name:function-name:return
```

For example, if you wanted to create a probe in process ID 12345 that fires on entry to `kstat_data_lookup(3KSTAT)`, you would write the following 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. The `pid` provider is described in further detail in "pid Provider" on page 233.

To apply your D program to different processes, use macro variables. Macro variables are evaluated at compile time and are replaced with additional `dtrace` command-line arguments. Macro variables are specified 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 more information about macro variables and reusable scripts, see Chapter 9, "Scripting". Now that you know how to instrument user processes using their process ID, return to sampling unions.

EXAMPLE 11   kstat.d: Trace Calls to `kstat_data_lookup(3KSTAT)`

Type the following source code in a text editor and save it as `kstat.d`:

```
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;
}
```

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

```
# dtrace -q -s kstat.d `pgrep

zonestatd`
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 statistics, you should be able to correlate the `dtrace` output with the `mpstat` output. The example program records the counter name pointer on entry to the lookup function, and then performs most of the tracing work on return from kstat_data_lookup(3KSTAT). The D built-in functions `copyinstr` and `copyin` copy the function results from the user process back into DTrace when `arg1` (the return value) is not `NULL`. Once the `kstat` data has been copied, the example reports the `ui64` counter value from the union. This simplified example assumes 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`.

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 the 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)
```

Here *type-name* is the name of any struct or union type or type alias, 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.

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 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 compliers try to pack bits efficiently The compliers do not follow any order while packing the bits. Therefore, 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 by simply specifying its name in combination with the "." or -> operators like any other struct or union member. The bit-field is automatically promoted to the next largest integer type for use in any expressions. Because 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.

Type and Constant Definitions

This chapter describes how to declare type aliases and named constants in D. This chapter also discusses D type and namespace management for program and operating system types and identifiers.

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 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. Type 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`. Oracle Solaris C system headers often use the `suffix _t` to denote a `typedef` alias.

Enumerations

Defining symbolic names for constants in a program eases readability and simplifies 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 the following example:

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

The enumerator `BLUE` is assigned the value 10 by the compiler because 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. D also 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. Kernel enumerators 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) 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 to a value of type `enum uio_rw`, as shown in the following example clause:

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

This example traces calls to the uiomove(9F) function for write requests by comparing `args[2]`, a variable of type `enum uio_rw`, to the enumerator `UIO_WRITE`. Because the left-hand argument is an enumeration type, the D compiler searches the enumeration 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.

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 substitutes 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. 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: 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;
...
```

These inline definitions provide you access to the current set of Oracle 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 automatically so you can use these definitions in any D program.

Type Namespaces

This section discusses D namespaces and namespace issues related to types. In traditional languages such as ANSI-C, type visibility is determined by whether 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 are visible throughout the entire program. Types defined in C header files are typically included in this outer scope. Unlike th C language, D provides access to types from multiple outer scopes.

D is a language that facilitates 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. 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. Therefore, more than one data type of the same name, perhaps with different definitions, might 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, a kernel module named `foo` contains the following C type declaration:

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

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 that use 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) using the `dtrace -C` option will be processed by 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 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, the member types 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, the compiler 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 uses compressed ANSI-C debugging information provided with the core Oracle 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 might 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.

3
♦♦♦ **C H A P T E R   3**

Aggregations

Dtrace enables you to aggregate data gathered by the probes. For example, if you wanted 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. Without aggregation, you would have postprocessed the data gathered at each system call by using a tool such as `awk` or `perl`. For more information about `awk` and `perl`, see `awk(1)` or `perl(1)` man page. In DTrace, aggregation of data is a first-class operation. This chapter describes the DTrace facilities for manipulating aggregations.

Aggregating Functions

An *aggregating function* has the following property:

`f(f(x0) U f(x1) U ... U f(xn)) = f(x0 U x1 U ... U xn)`

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. For example, consider a function `SUM` that yields the summation of a given data 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 the original raw data, if `MEDIAN` is applied 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.

Many common functions for understanding a set of data are aggregating functions. These functions 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. Determining the arithmetic mean of the set can be constructed from the function to count the number of elements in the set and the function to sum the elements in the set.

However, several useful functions are not aggregating functions. These functions 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 as it is traced 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 process reduces the amount of storage required by a factor of the number of data points, which is often quite large.
- Data collection does not induce pathological scalability problems. Aggregating functions enable intermediate results to be kept *per-CPU* instead of in a shared data structure. DTrace then applies the aggregating function to the set consisting of the per-CPU intermediate results to produce the final system-wide result.

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 this name as an alias for the aggregation name `@_`.

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

TABLE 17   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.
stddev	scalar expression	The standard deviation of 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.
llquantize	scalar expression	A frequency distribution table of the values specified in the expression. Increments the value in the highest bucket that is less than the specified expression. The values are grouped logarithmically at first and then grouped linearly.

For example, to count the number of `write` 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 presses Control-C. The following example output shows the result of running this command, waiting for a few seconds, and pressing Control-C.

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

```
  write system calls 179
#
```

You can count system calls per process name using the execname variable as the key to an aggregation:

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

The following example output shows the result of running this command, waiting for a few seconds, and pressing 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, you might want to further examine writes organized by both executable name and file descriptor. The file descriptor is the first argument to *write(2)*, so the following example uses a key consisting of both execname and arg0:

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

Running this command results in a table with both executable name and file descriptor, as shown in the following example:

```
# 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
#
```

The following example displays the average time spent in the write system call, organized by process name. This example uses the avg aggregating function, specifying the expression to average as the argument. The example averages the wall clock 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;
}
```

The following example output shows the result of running this command, waiting for a few seconds, and pressing 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 often does not provide sufficient detail to understand the distribution of data points. To understand the distribution in further detail, use the quantize aggregating function as shown in the following example:

```
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. The following example shows a selection of sample output:

```
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
```

Notice that the rows for the frequency distribution are *always* power-of-two values. Each rows indicates the count of the number of elements *greater than or equal to* the corresponding value, but *less than* the next larger row value. For example, the above output shows that `iropt` had 4,149 writes taking between 8,192 nanoseconds and 16,383 nanoseconds, inclusive.

While `quantize` is useful for getting quick insight into the data, you might want to examine a distribution across linear values instead. To display a linear value distribution, 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 you wanted to look at the distribution of writes by file descriptor, a power-of-two quantization would not be effective. Instead, use a linear quantization with a small range, as shown in the following example:

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

Running this script for several seconds yields a large amount of information. The following example shows a selection of typical output:

```
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
```

You can also use the `lquantize` aggregating function to aggregate on time since some point in the past. This technique allows you to observe a change in behavior over time. The following example displays the change in system call behavior over the lifetime of a process executing the *date(1)* command:

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

syscall:::entry
/self->start/
{
        /*
         * You linearly quantize on the current virtual time minus the
         * process's start time. Divide by 1000 to yield microseconds
         * rather than nanoseconds. The range runs from 0 to 10 milliseconds
         * in steps of 100 microseconds; you can expect that no date 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 preceding script provides greater insight into system call behavior when many *date(1)* processes are executed. To see this result, run `sh -c 'while true; do date >/dev/null; done'` in one window, while executing the D script in another. The script produces 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
```

This output provides a rough idea of the different phases of the date command with respect to the services required of the kernel. To better understand these phases, you might want to understand which system calls are being called when. If so, you could change the D script to aggregate on the variable probefunc instead of a constant string.

You can use the llquantize aggregating function to group data both logarithmically and linearly. The arguments to the llquantize function in order are: the expression, the base, the low exponent, the high exponent, and the number of steps per order of magnitude. The following example displays system call latencies in the microsecond range. Note that timestamp is in nanoseconds and 1000 nanoseconds = 1 microsecond.

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

```
syscall:::return
/ self->ts /
{
        /* The expression "timestamp - self->ts" gives the system call latency in
 nanoseconds */

        @ = llquantize(timestamp - self->ts, 10, 3, 5, 10);
        self->ts = 0;
}
```

In this example the expression `@ = llquantize(timestamp - self->ts, 10, 3, 5, 10);` generates a frequency distribution of the observed system call latencies. The distribution runs from 10^3 nanoseconds through 10^5 nanoseconds (inclusive). For each order of the magnitude, the data is displayed linearly in groups of 10. This example generates an output similar to the following:

```
     value  ------------- Distribution ------------- count
       < 1000 |@@@@@@                                   12899
         1000 |@@@@@@@@@@@@                             26357
         2000 |@                                        3202
         3000 |@                                        1869
         4000 |@                                        2110
         5000 |@@                                       4716
         6000 |@@                                       3998
         7000 |@                                        1617
         8000 |@@                                       4924
         9000 |@                                        2515
        10000 |@@@@@@@                                  15307
        20000 |@                                        2240
        30000 |@                                        1327
        40000 |@                                        1369
        50000 |                                         990
        60000 |                                         1057
        70000 |                                         631
        80000 |                                         453
        90000 |                                         434
       100000 |@                                        1570
       200000 |                                         228
       300000 |                                         45
       400000 |                                         59
       500000 |                                         60
       600000 |                                         52
       700000 |                                         30
       800000 |                                         22
       900000 |                                         17
    >= 1000000 |                                        513
```

You can use the `stddev` aggregating function to characterize the distribution of data points. This example shows the average and standard deviation of the time it takes to exec processes:

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

syscall::exece:return

/ self->ts /
{
        t = timestamp - self->ts;
        @execavg[probefunc] = avg(t);
        @execsd[probefunc] = stddev(t);

        self->ts = 0;
}

END
{
        printf("AVERAGE:");
        printa(@execavg);
        printf("nSTDDEV:");
        printa(@execsd);
}
```

With sample output as follows:

```
# dtrace -s ./stddev.d
dtrace: script './stddev.d' matched 3 probes
^C
CPU     ID                    FUNCTION:NAME
  0      2                             :END AVERAGE:
  exece                                                    7053786
STDDEV:
  exece                                                    9470351
```

Printing Aggregations

By default, multiple aggregations are displayed in the order they are introduced in the D program. You can override this behavior using the printa function to print the aggregations. The printa function also enables you to precisely format the aggregation data using a format string, as described in Chapter 6, "Output Formatting".

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 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 display an integer decimal value corresponding to the aggregated value for each tuple. The default output format for the lquantize and quantize aggregating function displays an ASCII table of the results. Aggregation tuples are printed as if trace had been applied to each tuple element.

Data Normalization

When aggregating data over some period of time, you might want to *normalize* the data with respect to some constant factor. This technique enables you to compare disjoint data more easily. For example, when aggregating system calls, you might want to output system calls as a per-second rate instead of as an absolute value over the course of the run. The DTrace normalize action enables you to normalize data in this way. The parameters to normalize are an aggregation and a normalization factor. The output of the aggregation shows each value divided by the normalization factor.

EXAMPLE 12   normalize.d Normalizing an Aggregation

The following example shows how to aggregate data by 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 it has
        * been running. (There are 1,000,000,000 nanoseconds in one second.)
        */
        normalize(@func, (timestamp - start) / 1000000000);
}
```

Running the above script for a brief period of time results in the following output:

```
# 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 this action does not modify the underlying data. denormalize takes only an aggregation. Adding the denormalize action to the preceding example returns both raw system call counts and per-second rates.

EXAMPLE 13   denorm.d: Denormalize an Aggregation

```
#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 script for a brief period of time produces output similar to the following example:

```
# 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 can also be renormalized. If `normalize` is called more than once for the same aggregation, the normalization factor will be the factor specified in the most recent call. The following example prints per-second rates over time:

EXAMPLE 14   `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);
}
```

Clearing Aggregations

When using DTrace to build simple monitoring scripts, you can periodically clear the values in an aggregation using the `clear` function. This function takes an aggregation as its only parameter. The `clear` function clears only the aggregation's *values*; the aggregation's keys are

retained. Therefore, 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`. To discard both an aggregation's values and its keys, use the `trunc` function. For more information about the `trunc` function, see "Truncating Aggregations" on page 106.

The following example adds the `clear` function to Example 14, "`renormalize.d`: Renormalizing an Aggregation," on page 103:

EXAMPLE 15   Clearing Aggregations

```
#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;
}
```

While Example 14, "`renormalize.d`: Renormalizing an Aggregation," on page 103 shows the system call rate over the lifetime of the `dtrace` invocation, the preceding example shows the system call rate only for the most recent ten-second period.

Sorting Aggregations

When displaying aggregated data, you can sort the output. The output can be sorted based on aggregation keys or aggregation variables. The following options are available to sort the output:

aggsortkey   Sort aggregation by key order with ties broken by value.

aggsortkeypos   Position of the aggregate key on which the output is sorted.

aggsortpos   Position of the argument in the aggregate function on which the output is sorted.

aggsortrev   Sort aggregation in the reverse order.

For example, consider the following `aggsort.d` script that prints aggregation values.

```
/* aggsort.d */
syscall::read:entry
{
@avg[execname, pid] = avg(arg2);
@max[execname, pid] = max(arg2);
@min[execname, pid] = min(arg2);
@cnt[execname, pid] = count();
}
END
{
printf("%20s %10s %10s %10s %10s %10s\n", "EXECNAME", "PID", "COUNT", "MIN", "MAX",
 "AVG");
printa("%20s %10d %@10d %@10d %@10d %@10d\n", @cnt, @min, @max, @avg);
}
```

Running the `aggsort.d` script displays an output similar to the following:

```
# dtrace -q -s ./aggsort.d
^C

EXECNAME                   PID       COUNT        MIN        MAX        AVG
battstat-applet-        100981           2         32         32         32
gnome-settings-d        100853           3         32         64         53
soffice.bin             101382          10         32         32         32
dsdm                    100708          10         32        160         54
xscreensaver            101082          14         32         32         32
gnome-panel             100896          24         12        168         51
firefox-bin             101363          31          1       1024         35
gnome-terminal          101029          40         32       4096        163
nautilus                100906         119         32        480         48
wnck-applet             100961         161          8        128         32
metacity                100893         447          4         96         33
Xorg                    100534         926         64       5104       3263
```

Note - If no sort options are provided, the output is sorted on the value of first aggregation. In the preceeding example, the output is sorted on COUNT.

If you want to sort the output based on the `pid` which is the second aggregation key, you must specify the `aggsortkey` and the `aggsortkeypos` options. Running the `aggsort.d` with these options, displays an output similar to the following:

```
# dtrace -q -s ./aggsort.d -x aggsortkey -x aggsortkeypos=1
^C

EXECNAME                   PID       COUNT        MIN        MAX        AVG
```

```
Xorg                 100534        885       64       4940       3688
dsdm                 100708         23       32        128         40
gnome-settings-d     100853          4       32         64         40
metacity             100893        196        4        128         34
gnome-panel          100896         26       12        168         49
nautilus             100906         79       32        320         40
wnck-applet          100961        161        8         96         32
gnome-terminal       101029         27       32       4096        220
xscreensaver         101082         11       32         64         34
thunderbird-bin      101337          2        1       1024        512
firefox-bin          101363         29        1       1024         36
soffice.bin          101382        525        4       1440         33
```

To sort the output based on any of the aggregation variables, you must use the `aggsortpos` option. Running the following command sorts the output based on `MAX` which is the third aggregation variable.

```
# dtrace -q -s ./aggsort.d -x aggsortpos=2
```

To sort the aggregation values in a reverse order, use the `aggsortrev` option. For example:

```
# dtrace -q -s ./aggsort.d -x aggsortrev
```

Truncating Aggregations

You can use the `trunc` function to manipulate the aggregation results. For example, you can display only the top results, or discard an entire aggregation result and so on.

The parameters to `trunc` are an aggregation and an optional truncation value. Without the truncation value, `trunc` discards *both* aggregation values *and* aggregation keys for the entire aggregation. When a truncation value *n* is present, `trunc` discards aggregation values and keys *except* for those values and keys associated with the highest *n* values. That is, `trunc(@foo, 10)` truncates the aggregation named `foo` after the top ten values, where `trunc(@foo)` discards the entire aggregation. The entire aggregation is also discarded if `0` is specified as the truncation value.

To see the bottom *n* values instead of the top *n*, specify a negative truncation value to `trunc`. For example, `trunc(@foo, -10)` truncates the aggregation named `foo` after the bottom ten values.

The following example augments the system call example to only display the per-second system call rates of the top ten system-calling applications in a ten-second period.

EXAMPLE 16   Truncating an Aggregation

```
#pragma D option quiet
```

```
BEGIN
{
        last = timestamp;
}

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

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

The following example shows output from running the above script on a lightly loaded laptop:

```
FvwmAuto                                                       7
  telnet                                                      13
  ping                                                        14
  dtrace                                                      27
  xclock                                                      34
  MozillaFirebird-                                            63
  xterm                                                      133
  fvwm2                                                      146
  acroread                                                   168
  Xsun                                                       616

  telnet                                                       4
  FvwmAuto                                                     5
  ping                                                        14
  dtrace                                                      27
  xclock                                                      35
  fvwm2                                                       69
  xterm                                                       70
  acroread                                                   164
  MozillaFirebird-                                           491
  Xsun                                                      1287
```

Minimizing Drops

Because DTrace buffers some aggregation data in the kernel, space might not 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 indicating an aggregation drop. This situation rarely occurs because DTrace keeps 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 occur, you can increase the aggregation buffer size with the `aggsize` option to reduce the likelihood of drops. You can also use this option to minimize the memory footprint of DTrace. As with any size option, `aggsize` may be specified with any size suffix. The resizing policy of this buffer is dictated by the `bufresize` option. For more information about buffering, see Chapter 5, "Buffers and Buffering". For more information about options, see Chapter 10, "Options and Tunables".

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, and 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. For more information about the `aggsize` option, see Chapter 10, "Options and Tunables".

4

Actions and Subroutines

You can use D function calls such as `trace` and `printf` to invoke two different kinds of services provided by DTrace: *actions* that trace data or modify state external to DTrace, and *subroutines* that affect only internal DTrace state. This chapter defines the actions and subroutines and describes their syntax and semantics.

Actions

Actions enable your DTrace programs to interact with the system outside of DTrace. The most common actions record data to a DTrace buffer. Other actions are available, such as stopping the current process, raising a specific signal on the current process, or ceasing tracing altogether. Some of these actions are destructive 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. For more details on the principal buffer and buffer policies, see Chapter 5, "Buffers and Buffering".

Default Action

A clause can contain any number of actions and variable manipulations. If a clause is left empty, the default action is taken. The default action is to trace the enabled probe identifier (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 enable the consumer to make sense of the data. Therefore, the default action is to trace the EPID.

Using the default action allows for simple use of `dtrace`. For more information about `dtrace`, see the `dtrace(1M)` man page. The following example command enables all probes in the `TS` timeshare scheduling module with the default action:

```
# dtrace -m TS
```

The preceding command might produce output similar to the following example:

```
# 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
...
```

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 action may also be used to record data to speculative buffers. For more information about the principal buffer, see Chapter 5, "Buffers and Buffering". For more information about speculative buffers, see Chapter 7, "Speculative Tracing". The descriptions in this section refer only to the *directed buffer,* indicating that data is recorded either to the principal buffer or to a speculative buffer if the action follows a speculate.

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. You can force `trace` to use binary format for ASCII text by using the `rawbytes` option.

EXAMPLE 17   Tracing Action

The following statements are examples of `trace` actions:

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

If the `trace` action is used to record a buffer, the output format depends on the data that is collected. If `dtrace` decides that the data looks like an ASCII string, it prints text and the output is terminated by the first null byte. However, if `dtrace` decides that the data is in a binary form, it prints the output in a hexadecimal format. For example:

```
  0    342                       write:entry
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
        0: c0 de 09 c2 4a e8 27 54 dc f8 9f f1 9a 20 4b d1  ....J.'T..... K.
       10: 9c 7a 7a 85 1b 03 0a fb 3a 81 8a 1b 25 35 b3 9a  .zz.....:...%5..
       20: f1 7d e6 2b 66 6d 1c 11 f8 eb 40 7f 65 9a 25 f8  .}.+fm....@.e.%.
       30: c8 68 87 b2 6f 48 a2 a5 f3 a2 1f 46 ab 3d f9 d2  .h..oH.....F.=..
       40: 3d b8 4c c0 41 3c f7 3c cd 18 ad 0d 0d d3 1a 90  =.L.A<.<........
```

tracemem

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

The `tracemem` action 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 *address* into the directed buffer for the length specified by *nbytes*. If only two arguments are provided, `dtrace` will dump the entire contents of the buffer.

In the second format, the `tracemem` action takes an additional third argument, `dbytes`, which is a D expression that is computed dynamically; the result is used to limit the number of bytes that are displayed to the user. If the result is less than zero or greater than `nbytes`, the result is ignored and `tracemem` will behave as though it is called using the two argument form. Otherwise, `dtrace` will dump only `dbytes` bytes of the directed buffer.

printf

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

Like `trace`, the `printf` action traces D expressions. However, `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. The arguments are later formatted for output by *dtrace(1M)* according to the specified format string. For example, the first two examples of `trace` from "trace" on page 110 could be combined in a single `printf`:

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

For more information on `printf`, see Chapter 6, "Output Formatting".

printa

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

The `printa` action enables you to display and format aggregations. For more information about aggregations, see Chapter 3, "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. For more information about the `printa` format string, see Chapter 6, "Output Formatting".

`printa` only traces a directive that the aggregation should be processed by the DTrace consumer. It does not process the aggregation in the kernel. Therefore, 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. For more information, see Chapter 3, "Aggregations" and Chapter 5, "Buffers and Buffering".

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 specified 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'{@[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

    ...
```

ustack

```
void ustack(int nframes, int strsize)
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 specified 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. If the noresolve option is set, symbol resolution is not performed, and the frames will appear only as an object name and an offset within that object. Applications that contain many symbols can incur an overhead when DTrace resolves addresses to symbol names. This overhead can be reduced with the noresolve option. The object name and offset can be resolved to a symbol name by cross-referencing with the object's symbol table. If *strsize* is specified and non-zero, `ustack` will allocate the specified amount of string space, and use it to perform address-to-symbol translation directly from the kernel. This direct user symbol translation is currently available only for Java virtual machines, version 1.5 and higher. Java address-to-symbol translation annotates user stacks that contain Java frames with the Java class and method name. If such frames cannot be translated, the frames will appear only as hexadecimal addresses.

The following example traces a stack with no string space, and therefore no Java address-to-symbol translation:

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 0);
    exit(0)}' -c "java -version"
dtrace: description 'syscall::write:entry' matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5312 has exited
CPU     ID                    FUNCTION:NAME
  0     35                        write:entry
              libc.so.1`_write+0x15
              libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8
```

```
                 libjvm.so`JVM_Write+0x2f
                 d0c5c946
                 libjava.so`Java_java_io_FileOutputStream_writeBytes+0x2c
                 cb007fcd
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb002a7b
                 cb000152
                 libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_
                          pnMmethodHandle_pnRJavaCallArguments_
                          pnGThread__v_+0x187
                 libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_
                          pnMmethodHandle_pnRJavaCallArguments_
                          pnGThread__v2468_v_+0x14
                 libjvm.so`__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_
                          pnRJavaCallArguments_pnGThread __v_+0x28
                 libjvm.so`__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_
                          pnI_jobject_nLJNICallType_pnK_jmethodID_pnSJNI_
                          ArgumentPusher_pnGThread__v_+0x180
                 libjvm.so`jni_CallStaticVoidMethod+0x10f
                 java`main+0x53d
```

Notice that the C and C++ stack frames from the Java virtual machine are presented symbolically using C++ "mangled" symbol names, and the Java stack frames are presented only as hexadecimal addresses. The following example shows a call to ustack with a non-zero string space:

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 500); exit(0)}'
      -c "java -version"
dtrace: description 'syscall::write:entry' matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5308 has exited
CPU     ID                    FUNCTION:NAME
  0     35                        write:entry
                 libc.so.1`_write+0x15
                 libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8
                 libjvm.so`JVM_Write+0x2f
                 d0c5c946
                 libjava.so`Java_java_io_FileOutputStream_writeBytes+0x2c
```

```
                        java/io/FileOutputStream.writeBytes
                        java/io/FileOutputStream.write
                        java/io/BufferedOutputStream.flushBuffer
                        java/io/BufferedOutputStream.flush
                        java/io/PrintStream.write
                        sun/nio/cs/StreamEncoder$CharsetSE.writeBytes
                        sun/nio/cs/StreamEncoder$CharsetSE.implFlushBuffer
                        sun/nio/cs/StreamEncoder.flushBuffer
                        java/io/OutputStreamWriter.flushBuffer
                        java/io/PrintStream.write
                        java/io/PrintStream.print
                        java/io/PrintStream.println
                        sun/misc/Version.print
                        sun/misc/Version.print
                        StubRoutines (1)
                        libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_
                                pnMmethodHandle_pnRJavaCallArguments_pnGThread
                                __v_+0x187
                        libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_
                                pnMmethodHandle_pnRJavaCallArguments_pnGThread
                                __v2468_v_+0x14
                        libjvm.so`__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle
                                _pnRJavaCallArguments_pnGThread__v_+0x28
                        libjvm.so`__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_pnI
                                _jobject_nLJNICallType_pnK_jmethodID_pnSJNI
                                _ArgumentPusher_pnGThread__v_+0x180
                        libjvm.so`jni_CallStaticVoidMethod+0x10f
                        java`main+0x53d
                        8051b9a
```

The above example output demonstrates symbolic stack frame information for Java stack frames. There are still some hexadecimal frames in this output because some functions are static and do not have entries in the application symbol table. Translation is not possible for these frames.

The `ustack` symbol translation for non-Java frames occurs after the stack data is recorded. Therefore, the corresponding user process might exit before symbol translation can be performed, making stack frame translation impossible. If the user process exits before symbol translation is performed, `dtrace` will emit a warning message, followed by the hexadecimal stack frames, as shown in the following example:

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

Techniques for mitigating this problem are described in Chapter 12, "User Process Tracing".

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 that leaves *strsize* unspecified:

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

To run this example for the Oracle Solaris prstat command use the following command:

```
# dtrace -s brk.d prstat
dtrace: script 'brk.d' matched 1 probe
^C

                libc.so.1`_brk_unlocked+0xa
                libc.so.1`sbrk+0x2c
                libc.so.1`_morecore+0x112
                libc.so.1`_malloc_unlocked+0x182
                libc.so.1`realloc+0x39f
                prstat`Realloc+0x21
                prstat`Zalloc+0x17
                prstat`fd_init+0x2a
                prstat`main+0x55
                prstat`0x4047dc
                  1

                libc.so.1`_brk_unlocked+0xa
                libc.so.1`sbrk+0x2c
                libc.so.1`_morecore+0x112
                libc.so.1`_malloc_unlocked+0x182
                libc.so.1`malloc+0x31
                libc.so.1`calloc+0x57
                libcurses.so.1`setupterm+0xacc
                libcurses.so.1`newscreen+0x87
                libcurses.so.1`initscr32+0x53
                prstat`curses_on+0x1f
                prstat`main+0x7f0
                prstat`0x4047dc
                  1

...
```

jstack

```
void jstack(int nframes, int strsize)
```

```
void jstack(int nframes)
void jstack(void)
```

jstack is an alias for ustack that uses the value of the jstackframes option for the number of stack frames and the value of the jstackstrsize option for the string space size. By default, jstacksize defaults to a non-zero value. As a result, use of jstack will result in a stack with in situ Java frame translation.

clear

When using dtrace to build simple monitoring scripts, you can periodically clear the values in an aggregation using the clear function. For more information, see the "Clearing Aggregations" on page 103.

denormalize

For more information, see the "Data Normalization" on page 100.

normalize

For more information, see the "Data Normalization" on page 100.

trunc

For more information, see the "Truncating Aggregations" on page 106.

mod

The mod action displays the module name of the kernel argument. This action is exactly the same as the sym command.

umod

The umod action displays the module name of the user-land argument. This action is exactly the same as theusym command.

func

The func action displays the function name of the kernel argument.

ufunc

The ufunc action displays the function name of the user-land argument.

freopen

The freopen action changes the standard output file to the file specified by pathname argument. This action is potentially destructive and needs to be activated via -w or destructive option.

```
"dispatch_command":
#!/usr/sbin/dtrace -qw
#pragma D option strsize=1024
dtrace:::BEGIN
{
        freopen("/tmp/sqls");

}
pid$target::*dispatch_command*:entry

{
        printf("%Y-> %s \n",walltimestamp,copyinstr(arg2));
}
```

which gives logs like:

```
2009 Feb  5 08:13:43- > create table fo_bawr  (i INTEGER)
2009 Feb  5 08:13:56- > create table foo_bar  (is INTEGER)
```

ftruncate

The ftruncate action truncates standard output file, but does not truncate standard error file. Perform the following steps to truncate a standard error file:

1. Re-open stdout as stderr.
2. Truncate stderr.
3. Re-open the original stdout.

Following example displays how to truncate a standard error file.

```
# cat > script.d <<EOF
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option destructive

BEGIN
{
 printf("data\n");
 freopen("/dev/stderr");
 ftruncate();
 freopen(""); /* re-open original stdout */
 printf("more data\n");
 exit(0);
}
EOF
# chmod a+x script.d
# echo foo > stdout
# echo foofoo > stderr
# ls -l std*
-rw-r--r--   1 root     root          7 Feb 27 14:21 stderr
-rw-r--r--   1 root     root          4 Feb 27 14:21 stdout
# ./script.d >> stdout 2 >> stderr
# ls -l std*
-rw-r--r--   1 root     root          0 Feb 27 14:26 stderr
-rw-r--r--   1 root     root         19 Feb 27 14:26 stdout
# cat stdout
foo
data
more data
#
```

setopt

setopt action allows dynamic specification of a dtrace option.

```
void setopt(const char *opt_name)
void setopt(const char *opt_name, const char *opt_value)
```

For example:

```
setopt("quiet");
```

```
setopt("bufsize", "20m");
```

```
setopt("aggrate", "96hz");
```

uaddr

```
_usymaddr uaddr(uintptr_t address)
```

uaddr prints the symbol for a specified address, including hexadecimal offset. This enables the
same symbol resolution that ustack provides.

```
# dtrace -c date -n 'pid$target::main:entry{ uaddr(0x8062578); }'
dtrace: description 'pid$target::main:entry' matched 1 probe
Sun Feb  3 20:58:03 PST 2008
dtrace: pid 105537 has exited
CPU     ID                    FUNCTION:NAME
  0  59934                       main:entry   date`clock_val
```

In the above example, a call to uaddr(0x8062578) causes date`clock_val to be printed.

The following example shows how the hexadecimal offsets are printed.

```
demo$ sudo dtrace -n "pid\$target::main:{uaddr(uregs[R_PC])}" -c

date
dtrace: description 'pid$target::main:' matched 203 probes
Saturday, June  9, 2012 07:06:39 AM IST
CPU     ID                    FUNCTION:NAME
  0  67146                       main:entry   date`main
  0  67147                           main:0   date`main
  0  67148                           main:1   date`main+0x1
  0  67149                           main:3   date`main+0x3
  0  67151                           main:4   date`main+0x4
  0  67152                           main:5   date`main+0x5
  0  67153                           main:6   date`main+0x6
  0  67154                           main:9   date`main+0x9
  0  67155                           main:c   date`main+0xc
  0  67156                           main:f   date`main+0xf
  0  67157                          main:14   date`main+0x14
  0  67158                          main:16   date`main+0x16
  0  67159                          main:18   date`main+0x18
  0  67160                          main:1a   date`main+0x1a
  0  67161                          main:1f   date`main+0x1f
  0  67162                          main:22   date`main+0x22
  0  67163                          main:27   date`main+0x27
  0  67164                          main:2c   date`main+0x2c
  0  67165                          main:2f   date`main+0x2f
  0  67166                          main:34   date`main+0x34
  0  67167                          main:37   date`main+0x37
  0  67168                          main:3a   date`main+0x3a
  0  67169                          main:3b   date`main+0x3b
  0  67170                          main:40   date`main+0x40
  0  67171                          main:43   date`main+0x43
  0  67172                          main:46   date`main+0x46
  0  67210                          main:bb   date`main+0xbb
```

```
    0  67211                              main:be    date`main+0xbe
    0  67212                              main:c4    date`main+0xc4
    0  67213                              main:c6    date`main+0xc6
    0  67214                              main:c9    date`main+0xc9
    0  67215                              main:cc    date`main+0xcc
    0  67216                              main:cf    date`main+0xcf
    0  67217                              main:d6    date`main+0xd6
    0  67221                              main:dd    date`main+0xdd
    0  67222                              main:df    date`main+0xdf
    0  67223                              main:e5    date`main+0xe5
    0  67224                              main:e8    date`main+0xe8
    0  67225                              main:ed    date`main+0xed
    0  67226                              main:ee    date`main+0xee
    0  67227                              main:f0    date`main+0xf0
    0  67228                              main:f5    date`main+0xf5
    0  67229                              main:f8    date`main+0xf8
    0  67230                              main:fd    date`main+0xfd
    0  67231                              main:102   date`main+0x102
    0  67232                              main:104   date`main+0x104
    0  67233                              main:10a   date`main+0x10a
    0  67234                              main:10e   date`main+0x10e
    0  67258                              main:156   date`main+0x156
    0  67259                              main:159   date`main+0x159
    0  67260                              main:15b   date`main+0x15b
    0  67261                              main:160   date`main+0x160
    0  67262                              main:163   date`main+0x163
    0  67263                              main:165   date`main+0x165
    0  67264                              main:16c   date`main+0x16c
    0  67265                              main:16e   date`main+0x16e
    0  67266                              main:171   date`main+0x171
    0  67267                              main:176   date`main+0x176
    0  67268                              main:17b   date`main+0x17b
    0  67269                              main:17e   date`main+0x17e
    0  67270                              main:180   date`main+0x180
    0  67281                              main:1a9   date`main+0x1a9
    0  67282                              main:1ac   date`main+0x1ac
    0  67283                              main:1b1   date`main+0x1b1
    0  67284                              main:1b2   date`main+0x1b2
    0  67285                              main:1b7   date`main+0x1b7
    0  67286                              main:1ba   date`main+0x1ba
    0  67287                              main:1bc   date`main+0x1bc
    0  67288                              main:1bf   date`main+0x1bf
    0  67289                              main:1c0   date`main+0x1c0
    0  67290                              main:1c1   date`main+0x1c1
    0  67291                              main:1c6   date`main+0x1c6
    0  67292                              main:1c9   date`main+0x1c9
    0  67293                              main:1cc   date`main+0x1cc
    0  67294                              main:1cd   date`main+0x1cd
    0  67295                              main:1d2   date`main+0x1d2
    0  67296                              main:1d7   date`main+0x1d7
    0  67297                              main:1dc   date`main+0x1dc
```

```
 0  67298                        main:1e1   date`main+0x1e1
 0  67299                        main:1e4   date`main+0x1e4
 0  67300                        main:1e9   date`main+0x1e9
 0  67301                        main:1ee   date`main+0x1ee
 0  67302                        main:1f1   date`main+0x1f1
 0  67303                        main:1f3   date`main+0x1f3
 0  67304                        main:1f6   date`main+0x1f6
 0  67305                        main:1f7   date`main+0x1f7
 0  67306                        main:1f8   date`main+0x1f8
 0  67307                        main:1f9   date`main+0x1f9
 0  67308                        main:1fa   date`main+0x1fa
 0   2513                     main:return   date`main+0x1fa
dtrace: pid 2972 has exited
```

When the noresolve option is specified, symbol resolution is not performed and the address prints as an object name and an offset within that object. Applications that contains many symbols can incur an overhead when DTrace resolves addresses to symbol names. This overhead can be reduced with the noresolve option. The object name and offset can be resolved to a symbol name by cross-referencing with the object's symbol table.

sym

The sym action describes the symbol name of the kernel argument.

usym

```
_usymaddr usym(uintptr_t address)
```

The usym action describes the symbol name of the user-land argument. usym prints the symbol for a specified address. This is analogous to how uaddr works, but without the hexadecimal offsets.

```
uaddr: date`clock_val+0x1
usym: date`clock_val
```

When the noresolve option is specified, symbol resolution is not performed and the address prints as an object name and an offset within that object. Applications that contains many symbols can incur an overhead when DTrace resolves addresses to symbol names. This overhead can be reduced with the noresolve option. The object name and offset can be resolved to a symbol name by cross-referencing with the object's symbol table.

Destructive Actions

Some DTrace actions are destructive in that they change the state of the system in some well-defined way. Destructive actions may not be used unless they have been explicitly enabled. When using dtrace(1M), you can enable destructive actions using 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 the following example:

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

An administrator may choose to disable destructive actions system-wide by setting the kernel tunable dtrace_destructive_disallow to 1. This may be done in a number of ways including rebooting after adding the following line to /etc/system:

```
set dtrace:dtrace_destructive_disallow = 1
```

You can set the kernel tunable dtrace_destructive_disallow parameter by using the mdb utility. For more information about the mdb utility, see mdb(1).

```
# echo "dtrace_destructive_disallow/W 1" | mdb -kw
dtrace_destructive_disallow: 0x0 = 0x1
```

Process Destructive Actions

Some destructive actions are destructive only to a particular process. These actions are available to users with the dtrace_proc or dtrace_user privileges. For more information about DTrace security privileges, see Chapter 14, "Security".

stop

```
void stop(void)
```

The stop action forces the process that fires the enabled probe to stop when it next leaves the kernel, as if stopped by a proc action. You can use the prun utility to resume a process that has been stopped by the stop action. For more information, see the prun(1) man page. The stop action can be used to stop a process at any DTrace probe point. This action can be used to capture a program in a particular state that would be difficult to achieve with a simple breakpoint, and then attach a traditional debugger like mdb to the process. You can also use the gcore utility to save the state of a stopped process in a core file for later analysis. For more information, see the gcore(1) man page.

raise

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

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

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.

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. For more information, see Chapter 10, "Options and Tunables".

system

```
void system(string program, ...)
```

The `system` action causes the program specified by *program* to be executed as if it were given to the shell as input. The *program* string may contain any of the `printf`/`printa` format conversions. Arguments must be specified that match the format conversions. For more information about valid format conversions, see Chapter 6, "Output Formatting".

The following example runs the `date` command once per second:

```
# dtrace -wqn tick-1sec'{system("date")}'
 Tue Jul 20 11:56:26 CDT 2004
 Tue Jul 20 11:56:27 CDT 2004
 Tue Jul 20 11:56:28 CDT 2004
 Tue Jul 20 11:56:29 CDT 2004
```

```
 Tue Jul 20 11:56:30 CDT 2004
```

The following example shows a more elaborate use of the action, using `printf` conversions in the *program* string along with traditional filtering tools like pipes.

```
#pragma D option destructive
#pragma D option quiet

proc:::signal-send
/args[2] == SIGINT/
{
        printf("SIGINT sent to %s by ", args[1]->pr_fname);
        system("getent passwd %d | cut -d: -f5", uid);
}
```

The above script results in the following output.

```
# ./whosend.d
SIGINT sent to MozillaFirebird- by Bryan Cantrill
SIGINT sent to run-mozilla.sh by Bryan Cantrill
^C
SIGINT sent to dtrace by Bryan Cantrill
```

The execution of the specified command does not occur in the context of the firing probe. It occurs when the buffer containing the details of the `system` action are processed at user-level. How and when this processing occurs depends on the buffering policy. For more information, see Chapter 5, "Buffers and Buffering". With the default buffering policy, the buffer processing rate is specified by the `switchrate` option. You can see the delay inherent in `system` if you explicitly tune the `switchrate` higher than its one-second default, as shown in the following example:

```
#pragma D option quiet
#pragma D option destructive
#pragma D option switchrate=5sec

tick-1sec
/n++ < 5/
{
        printf("walltime  : %Y\n", walltimestamp);
        printf("date      : ");
        system("date");
        printf("\n");
}

tick-1sec
/n == 5/
{
        exit(0);
}
```

Running the above script results in the following output.

```
# dtrace -s ./time.d
 walltime  : 2004 Jul 20 13:26:30
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:31
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:32
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:33
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:34
date      : Tue Jul 20 13:26:35 CDT 2004
```

Notice that the walltime values differ, but the date values are identical. This result reflects the fact that the execution of the date command occurred only when the buffer was processed, not when the system action was recorded.

Kernel Destructive Actions

Some destructive actions affect the entire system. These actions must be used carefully, as it can affect every process on the system and any other system implicitly or explicitly depending upon the affected system's network services.

breakpoint

```
void breakpoint(void)
```

The breakpoint action induces a kernel breakpoint, causing the system to stop and transfer control to the kernel debugger. The kernel debugger will emit a string denoting the DTrace probe that triggered the action. For example, type the following command:

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

The following message might appear on the console if you are running Oracle Solaris on a SPARC system:

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

The following message might appear on the console if you are running Oracle Solaris on a x86
system:

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

The address following the probe description is the address of the enabling control block (ECB)
within DTrace. You can use this address to determine more details about the probe enabling that
induced the breakpoint action.

A mistake with the `breakpoint` action may cause it to be called far more often than
intended. This behavior might in turn prevent you from even terminating the DTrace
consumer that is triggering the breakpoint actions. In this situation, set the kernel tunable
`dtrace_destructive_disallow` to `1`. This setting will disallow all destructive actions on the
system.

The method for setting `dtrace_destructive_disallow`, depends on the kernel debugger that
you are using. If you are using the OpenBoot PROM on a SPARC system, use the `w!` option.

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

Use the `w?` option to confirm that the `dtrace_destructive_disallow` variable is set.

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

Type `go` to continue.

```
ok go
```

If you are using [kmdb(1)](#) on x86 or SPARC systems, use the 4–byte write modifier (`W`) with the `/`
formatting `dcmd` to set `dtrace_destructive_disallow`.

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

Use `:c` to continue.

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

To re-enable destructive actions after continuing, explicitly reset
`dtrace_destructive_disallow` back to 0 using [mdb(1)](#).

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

```
#
```

panic

```
void panic(void)
```

The panic action causes a kernel panic when triggered. This action should be used to force a system crash dump at a time of interest. You can use this action together with ring buffering and postmortem analysis to understand a problem. For more information, see Chapter 5, "Buffers and Buffering" and Chapter 16, "Postmortem Tracing" respectively. When the panic action is used, a panic message appears that denotes the probe causing 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 also contains the probe and ECB responsible for the panic action.

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 might be timing related. For example, you can use this action 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 use of chill can induce interrupt latency, scheduling latency, and dispatch latency. Therefore, chill can

cause unexpected systemic effects and it must not be used indiscriminately. Because system activity relies on periodic interrupt handling, DTrace will refuse to execute the `chill` action for more than 500 milliseconds out of each one-second interval on any given CPU. If the maximum `chill` interval is exceeded, DTrace will report an illegal operation error, as shown in the following example:

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

This limit is enforced even if the time is spread across multiple calls to `chill`, or multiple DTrace consumers of a single probe. For example, the same error would be generated by the following command:

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

Special Actions

This section describes actions that are neither data recording actions nor destructive actions.

Speculative Actions

The actions associated with speculative tracing are `speculate`, `commit`, and `discard`. These actions are discussed in Chapter 7, "Speculative Tracing".

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` returns a status to user-level, it is a data recording action, However, unlike other data storing actions, `exit` cannot be speculatively traced. `exit` will cause the DTrace consumer to exit regardless of buffer policy. Because `exit` is a data recording action, it can be dropped.

When `exit` is called, only DTrace actions already in progress on other CPUs will be completed. No new actions will occur on any CPU. The only exception to this rule is the processing of the

END probe, which will be called after the DTrace consumer has processed the exit action and indicated that tracing should stop.

Subroutines

Subroutines differ from actions because they generally only affect internal DTrace state. Therefore, there are no destructive subroutines, and subroutines never trace data into buffers. Many subroutines have analogs in the Section 9F or Section 3C interfaces. For more information, see Intro(9F) and *Intro(3)* man pages.

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, no memory is allocated and an error is generated.

basename

```
string basename(char *str)
```

basename is a D analogue for basename(1). This subroutine creates a string that consists of a copy of the specified string, but without any prefix that ends in /. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, basename does not execute and an error is generated.

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 these conditions are not met, no copying takes place and an error is generated.

`cleanpath`

string cleanpath(char *_str_)

`cleanpath` creates a string that consists of a copy of the path indicated by *str*, but with certain redundant elements eliminated. In particular "`/./`" elements in the path are removed, and "`/../`" elements are collapsed. The collapsing of `/../` elements in the path occurs without regard to symbolic links. Therefore, it is possible that `cleanpath` could take a valid path and return a shorter, invalid one.

For example, if *str* were "`/foo/../bar`" and `/foo` were a symbolic link to `/net/foo/export`, `cleanpath` would return the string "`/bar`" even though `bar` might only be in `/net/foo` not `/`. This limitation is due to the fact that `cleanpath` is called in the context of a firing probe, where full symbolic link resolution o f arbitrary names is not possible. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `cleanpath` does not execute and an error is generated.

`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. For information about techniques to reduce the likelihood of `copyin` errors, see Chapter 12, "User Process Tracing".

`copyinstr`

string copyinstr(uintptr_t _addr_)
string copyinstr(uintptr_t _addr_, size_t maxlength)

`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 *maxlength* parameter, if specified, sets a limit on the number of bytes past *addr* which will be examined (the resulting string will always be null-terminated). The resulting string's length is limited to the value set by the `strsize` option; see Chapter 10, "Options and Tunables" 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 12, "User Process Tracing" for techniques to reduce the likelihood of `copyinstr` errors.

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 12, "User Process Tracing" for techniques to reduce the likelihood of `copyinto` errors.

dirname

```
string dirname(char *str)
```

`dirname` is a D analogue for dirname(1). This subroutine creates a string that consists of all but the last level of the pathname specified by *str*. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `dirname` does not execute and an error is generated.

inet_ntoa

```
string inet_ntoa(ipaddr_t *addr)
```

`inet_ntoa` takes a pointer to an IPv4 address and returns it as a dotted quad decimal string. This is similar to `inet_ntoa()` from libnsl as described in `inet(3SOCKET)`, however this D version takes a pointer to the IPv4 address rather than the address itself. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `inet_ntoa` does not execute and an error is generated.

inet_ntoa6

```
string inet_ntoa6(in6_addr_t *addr)
```

`inet_ntoa6` takes a pointer to an IPv6 address and returns it as an RFC 1884 convention 2 string, with lower case hexadecimal digits. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `inet_ntoa6` does not execute and an error is generated.

inet_ntop

`string inet_ntop(int af, void *addr)`

`inet_ntop` takes a pointer to an IP address and returns a string version depending on the provided address family. This is similar to `inet_ntop()` from libnsl as described in `inet (3SOCKET)`. Supported address families are AF_INET and AF_INET6, both of which have been defined for use in D programs. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `inet_ntop` does not execute and an error is generated.

msgdsize

`size_t msgdsize(mblk_t *mp)`

`msgdsize` returns the number of bytes in the data message pointed to by *mp*. For more information, see msgdsize(9F). `msgdsize` only includes data blocks of type `M_DATA` in the count.

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.

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 returns zero if the specified adaptive mutex is currently unowned.

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. For more information, see mutex_owned(9F).

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 meet one or more of the following conditions:

- The mutex is declared statically
- The mutex is created with an interrupt block cookie of NULL
- The mutex is created with an interrupt block cookie that does not correspond to a high-level interrupt

For more information about mutexes, see mutex_init(9F). The majority of mutexes in the Oracle Solaris kernel are adaptive.

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.

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.

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 held only by readers and no writer is blocked, or if the lock is not held at all, rw_iswriter returns zero. See rw_init(9F).

rw_write_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).

speculation

int speculation(void)

speculation reserves a speculative trace buffer for use with speculate and returns an identifier for this buffer. For information, see Chapter 7, "Speculative Tracing".

strjoin

string strjoin(char *str1, char *str2)

strjoin creates a string that consists of *str1* concatenated with *str2*. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, strjoin does not execute and an error is generated.

strlen

size_t strlen(string *str*)

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

5

Buffers and Buffering

Data buffering and management is an essential service provided by the DTrace framework for its clients, such as dtrace. This chapter explores data buffering in detail and describes options you can use to change DTrace's buffer management policies.

Principal Buffers

The *principal buffer* is present in every DTrace invocation and is the buffer to which tracing actions record their data by default. The following table lists the tracing actions that record their data by default in the principal buffer.

exit	printf	trace	ustack
printa	stack	tracemem	

The principal buffers are *always* allocated on a per-CPU basis. This policy is not tunable, but tracing and buffer allocation can be restricted to a single CPU by using the cpu option.

Principal Buffer Policies

DTrace permits tracing in highly constrained contexts in the kernel. In particular, DTrace permits tracing in contexts in which kernel software may not reliably allocate memory. The consequence of this flexibility of context is that there *always* exists a possibility that DTrace will attempt to trace data when there is no space available. DTrace must have a policy to deal with such situations when they arise, but you might wish to tune the policy based on the needs of a given experiment. Sometimes the appropriate policy might be to discard the new data. Other times it might be desirable to reuse the space containing the oldest recorded data to trace new data. Most often, the desired policy is to minimize 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 10, "Options and Tunables" for more details on setting options.

`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 and the other buffer is inactive. When a DTrace consumer attempts to read a buffer, 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: a buffer is never simultaneously traced to and copied out. This technique also avoids introducing a window in which tracing is paused or otherwise prevented. The rate at which the buffer is switched and read out 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. For more details on `switchrate` and other options, see Chapter 10, "Options and Tunables".

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, `dtrace` displays a message similar to the following example:

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

If a given record is larger than the total buffer size, the record will be dropped regardless of buffer policy. You can reduce or eliminate drops 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.

`fill` Policy

For some problems, you might want to use a single in-kernel buffer. While this approach can be implemented with the `switch` policy and appropriate D constructs by incrementing a variable in D and predicating an `exit` action appropriately, such an implementation does not eliminate the possibility of drops. To request a single, large in-kernel buffer, and continue tracing until one or more of the per-CPU buffers has filled, use the `fill` buffer policy. Under this policy, tracing continues until an enabled probe attempts to trace more data than can fit in the remaining principal buffer space. When insufficient space remains, the buffer is marked as filled and the consumer is notified that at least one of its per-CPU buffers has filled. Once `dtrace` detects a

single filled buffer, tracing is stopped, all buffers are processed and `dtrace` exits. 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, the following command 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
```

`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. If the `fill` policy is selected, the `END` probe may fire on a CPU that has a filled buffer. To accommodate `END` tracing in `fill` buffers, DTrace calculates the amount of space potentially consumed by `END` probes and *subtracts* this space from the size of the principal buffer. If the net size is negative, DTrace will refuse to start, and `dtrace` will output a corresponding error message:

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

The reservation mechanism ensures that a full buffer always has sufficient space for any `END` probes.

`ring` Policy

The DTrace `ring` buffer policy helps you trace the events leading up to a failure. If reproducing the failure takes hours or days, you might want to keep only the most recent data. Once a principal buffer has filled, tracing wraps around to the first entry, thereby overwriting older tracing data. You establish the ring buffer by setting the `bufpolicy` option to the string `ring`:

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

When used to create a ring buffer, `dtrace` will not display any output until the process is terminated. At that time, the ring buffer is consumed and processed. `dtrace` processes each ring buffer in CPU order. Within a CPU's buffer, trace records will be displayed in order from oldest to youngest. Just as with the `switch` buffering policy, no ordering exists between records from different CPUs. If such an ordering is required, you should trace the `timestamp` variable as part of your tracing request.

The following example demonstrates the use of a `#pragma D 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);
}
```

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 one or more *speculative buffers*. For more information about aggregation buffer, see Chapter 3, "Aggregations". For more information about speculative tracing, see Chapter 7, "Speculative Tracing".

Buffer Sizes

The size of each buffer can be tuned on a per-consumer basis. The following table lists separate options that 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. For more information, see Chapter 10, "Options and Tunables". For example, you can use one of the following commands to set the buffer size to one megabyte on the command line to `dtrace`:

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

- `#pragma D option bufsize=1m`

The buffer size you select 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.

Buffer Resizing Policy

Occasionally, the system might not have adequate free kernel memory to allocate a buffer of desired size either because not enough memory is available or because the DTrace consumer has exceeded one of the tunable limits. For more information about the tunable limits, see Chapter 10, "Options and Tunables". You can configure the policy for buffer allocation failure using the `bufresize` option, which defaults to `auto`. Under the `auto` buffer resize policy, the size of a buffer is halved until a successful allocation occurs. `dtrace` generates a message if a buffer allocated is smaller than the requested size:

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

`dtrace` might generate the following error message:

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

Alternatively, you can require manual intervention after buffer allocation failure 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
#
```

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

6

Output Formatting

DTrace provides built-in formatting functions `printf` and `printa` that you can use to format output. The D compiler provides features that are not found in the `printf(3C)`. This chapter also discusses the formatting behavior of the trace function and the default output format used by `dtrace` to display aggregations.

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 the C library `printf`, DTrace `printf` is 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 provides an 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.

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`.

Conversion Specifications

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

- Zero or more *flags* (in any order), that modify the meaning of the conversion specification as described in the next section.
- An optional minimum *field width*. If the converted value has fewer bytes than the field width, the value 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 indicates 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. For more information, see "Overview" on page 236. The size prefixes are not necessary in D and are provided for compatibility with the C `printf` function.
- A *conversion specifier* 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.

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`) is formatted with thousands grouping characters using the non-monetary grouping character. Some locales, including the POSIX C locale, do not provide non-monetary grouping characters for use with this flag.

-	The result of the conversion is left-justified within the field. The conversion is right-justified if this flag is not specified.
+	The result of signed conversion always begins with a sign (+ or -). If this flag is not specified, the conversion begins with a sign only when a negative value is converted.
space	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space is 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 an alternate form is defined for the selected conversion. The alternate formats for conversions are described along with the corresponding 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 is ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag is ignored. If the 0 and ' flags both appear, the grouping characters are inserted before the zero padding.

Width and Precision Specifiers

The minimum field width can be specified as a decimal digit string following any flag specifier, in which case the field width is 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 following order: width, precision, value.

Size Prefixes

Size prefixes are required in ANSI-C programs that use `printf` 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 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 as follows:

`h`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a short or unsigned short.
`l`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a long or unsigned long.
`ll`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a long long or unsigned long long.
`L`	Specifies that a following `e`, `E`, `f`, `g`, or `G` conversion applies to a long double.
`l`	Specifies that a following `c` conversion applies to a `wint_t` argument, and that a following `s` conversion character applies to a pointer to a `wchar_t` argument.

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 issues an appropriate error message. If an undefined conversion format is specified, the D compiler issues an appropriate error message. The conversion character sequences 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.

A
Exactly like `%a`, but for user symbols.

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

C
The `char`, `short`, or `int` argument is printed as an ASCII character if the character is a printable ASCII character. If the character is not a printable character, it is printed using the corresponding escape sequence as shown in Table 6, "`D` Character Escape Sequences," on page 44.

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.ddde+/-dd*, where there is one digit before the radix character and the number of digits after it is equal to the precision. The radix character is non-zero if the argument is non-zero. 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 produces 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 specified, 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`) is 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 are not removed from the result.

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`.

k   The `stack` argument is printed as if by a call to `trace()`. Handles
           both user and kernel-level stacks. Valid only with `printa()` because,
           according to the D compiler, `stack()` and `ustack()` "may not be called
           from a D expression (D program context required)."

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 is 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.

S   The argument must be an array of `char` or a `string`. The argument is
           processed as if by the %s conversion, but any ASCII characters that
           are not printable are replaced by the corresponding escape sequence
           described in Table 6, "D Character Escape Sequences," on page 44.

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 is 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.

Y   The `uint64_t` argument is interpreted to be the number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970, and is printed in the following cftime(3C) form: `"%Y %a %b %e %T %Z."` The current number of nanoseconds since 00:00 UTC, January 1, 1970 is available in the `walltimestamp` variable.

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

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` command takes a consistent snapshot of the aggregation data and produces output equivalent to the default output format used for aggregations, described in Chapter 3, "Aggregations". If the second form of the function is used, the `dtrace` 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, aggregate a count using the following D statements:

```
@a["hello", 123] = count();
@a["goodbye", 456] = count();
```

Add the D statement `printa`(*format-string*, `@a`)to a probe clause, `dtrace` takes a snapshot of the aggregation data and produce output as if you had entered the following statements for each tuple defined in the aggregation.

```
printf(format-string, "hello", 123);
printf(format-string, "goodbye", 456);
```

- 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 keys you want to omit to the end of the tuple and then omit corresponding conversion specifiers for them in the `printa` format string.

- You can use the additional `@` format flag character, which is only valid when used with `printa`, to include the aggregation result in the output. 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. The following table lists the aggregation result types.

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 the 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 you use `dtrace` to execute this program, wait a few seconds, and press `Control-C`, you will see output similar to the following example:

```
# 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
        ...
```

trace Default Format

If the `trace` function is used to capture data rather than `printf`, the `dtrace` command formats the results using a default output format. If the data is 1, 2, 4, or 8 bytes in size, the result is formatted as a decimal integer value. If the data is any other size and is a sequence of printable characters if interpreted as a 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.

7 CHAPTER 7

♦ ♦ ♦

Speculative Tracing

This chapter discusses the DTrace facility for *speculative tracing*, the ability to tentatively trace data and decide whether to *commit* the data to a tracing buffer or *discard* it. In DTrace, the primary mechanism for filtering out uninteresting events is the *predicate* mechanism. For more information, see "D Program Structure" on page 23. Predicates are useful when you know at the time that a probe fires whether or not the probe event is of interest. 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 descriptor of interest. However, in other situations, you might not know whether a given probe event is of interest until some time *after* the probe fires.

For example, if a system call is occasionally failing with a common error code (such as, `EIO` or `EINVAL`), examine the code path leading to the error condition. To capture the code path, you 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 are sporadic or nondeterministic, you would be forced to trace all events that *might* 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.

You can use the speculative tracing facility in these situations to tentatively trace data at one or more probe locations, and then decide to commit the data to the principal buffer at another probe location. As a result, your trace data contains only the output of interest, no postprocessing is required, and the DTrace overhead is minimized.

Speculation Interfaces

The following table describes the DTrace speculation functions.

TABLE 18   DTrace Speculation Functions

Function Name	Args	Description
`speculation`	None	Returns an identifier for a new speculative buffer

Function Name	Args	Description
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

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`. If a call to `speculation` fails, a `dtrace` message similar to the following example is generated:

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

The number of speculative buffers defaults to one, but may be optionally tuned higher. See "Speculation Options and Tuning" on page 160 for more information.

Using a Speculation

To use a speculation, an identifier 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. The D compiler will generate a compile-time error if a call to `speculate` follows data recording actions in a D probe clause. Therefore, clauses may contain speculative tracing or non-speculative tracing requests, 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` results in a compile-time error. A `speculate` may not follow a `speculate`: only one speculation is permitted per clause. A clause that contains *only* a `speculate` will speculatively trace the default action, which is defined to trace only the enabled probe ID. For information about the default action, see Chapter 4, "Actions and Subroutines".

Typically, you assign the result of `speculation` to a thread-local variable and then use that variable as a subsequent predicate to other probes as well as an argument to `speculate`. For example:

```
syscall::openat:entry
{
```

```
        self->spec = speculation();
}

syscall:::
/self->spec/
{
        speculate(self->spec);
        printf("this is speculative");
}
```

Committing a Speculation

You commit speculations 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 is copied and the drop count for the buffer is incremented. If the buffer has been speculatively traced to on more than one CPU, the speculative data on the committing CPU is copied immediately, while speculative data on other CPUs is copied some time after the `commit`. Thus, some time might 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 "Speculation 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` will silently fail. Finally, a clause containing a `commit` cannot contain a data recording action, but a clause may contain multiple `commit` calls to commit disjoint buffers.

Discarding a Speculation

You discard speculations 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, a `dtrace` message similar to the following example is generated:

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

The likelihood of all buffers being unavailable can be reduced by tuning the number of speculation buffers or the cleaning rate.

Speculation Example

One potential use for speculations is to highlight a particular code path. The following example shows the entire code path under the open(2) system call only when the open fails:

EXAMPLE 18   `specopen.d`: Code Flow for Failed open(2)

```
#!/usr/sbin/dtrace -Fs

syscall::openat:entry,
syscall::openat64: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
         * speculation is subsequently commited.
         */
        printf("%s", stringof(copyinstr(arg1)));
}

fbt:::
/self->spec/
{
        /*
         * A speculate() with no other actions speculates the default action:
         * tracing the EPID.
         */
        speculate(self->spec);
}

syscall::openat:return,
syscall::openat64:return
/self->spec/
{
        /*
```

```
         * To balance the output with the -F option, make sure that
         * every entry has a matching return.  Because the open entry was
         * speculated, you must also speculate the open return.
         * This is also a convenient time to trace the errno value.
         */
        speculate(self->spec);
        trace(errno);
}

syscall::openat:return,
syscall::openat64:return
/self->spec && errno != 0/
{
        /*
         * If errno is non-zero, commit the speculation.
         */
        commit(self->spec);
        self->spec = 0;
}

syscall::openat:return,
syscall::openat64:return
/self->spec && errno == 0/
{
        /*
         * If errno is not set, discard the speculation.
         */
        discard(self->spec);
        self->spec = 0;
}
```

Running the above script produces output similar to the following 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
```

Speculation Options and Tuning

If a speculative buffer is full when a speculative tracing action is attempted, no data is stored in the buffer and a drop count is incremented. In this situation, a `dtrace` message similar to the following example is generated:

```
dtrace: 38 speculative drops
```

Speculative drops will *not* prevent the full speculative buffer from being copied into the principal buffer when the buffer is committed. Similarly, speculative drops can occur even if drops were experienced on a speculative buffer that was ultimately discarded. Speculative drops can be reduced by increasing the speculative buffer size, which is tuned using the `specsize` option. The `specsize` option may be specified with any size suffix. The resizing policy of this buffer is dictated by the `bufresize` option.

Speculative buffers might be unavailable when `speculation` is called. If buffers exist that have not yet been committed or discarded, a `dtrace` message similar to the following example is generated:

```
dtrace: 1 failed speculation (no speculative buffer available)
```

You can reduce the likelihood of failed speculations of this nature 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, a `dtrace` message similar to the following example is generated:

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

This message indicates that `speculation` was called after `commit` was called for a speculative buffer, but before that buffer was actually committed on all CPUs. You can reduce the likelihood of failed speculations of this nature by increasing the rate at which CPUs are cleaned with the `cleanrate` option. The value of `cleanrate` defaults to 101.

`dtrace` Utility

The `dtrace` command is 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.

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 15, "Anonymous Tracing")
- Options to generate program stability reports (see Chapter 18, "Stability")
- Options to modify DTrace tracing and buffering behavior and enable additional D compiler features (see Chapter 10, "Options and Tunables")

`dtrace` can also be used to create D scripts by using it in a `#!` declaration to create an interpreter file (see Chapter 9, "Scripting"). 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.

Options

The `dtrace` command accepts the following options:

dtrace[-32| -64][-aACeFGHlqSvVwZ][-b *bufsz*][-c *cmd*][-D *name*[=def]][-I *path*][-L *path*][-o *output*][-p *pid*][-s *script*][-U *name*][-x *arg*[=val]][-X[a| 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 interpretation 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 -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 15, "Anonymous Tracing" 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 and then exits. For more information, see Chapter 15, "Anonymous Tracing". 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 15, "Anonymous Tracing". 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 specified command *cmd* and exit upon its completion. If more than one `-c` option is present on the command line, `dtrace` exits when all commands have exited, reporting the exit status for each child process as it terminates. The process-ID of the first command is made available to any D programs specified on the command line or using the `-s` option through the `$target` macro variable. Refer to Chapter 9, "Scripting" for more information on macro variables.

`-C`

Run the C preprocessor `cpp` 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` which is 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.

`-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 that can be linked into another

program. If the -o option is present, the ELF file is saved using the pathname specified as the argument for this operand. If the -o option is not present and the DTrace program is contained with a file whose name is *filename* `.s`, then the ELF file is saved using the name *file* `.o`; otherwise the ELF file is saved using the name `d.out`.

-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` produces a report of the probes matching the descriptions given using the -P, -m, -f, -n, -i, and -s options. If none of these options are specified, all probes are listed.

-L

Add the specified directory *path* to the search path for DTrace libraries. DTrace libraries are used to contain common definitions that may be used when writing D programs. The specified *path* is added after the default library search path.

-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 , -G, and -l options, or for the traced data. 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

Grab the specified process-ID *pid*, cache its symbol tables, and exit upon its completion. If more than one -p option is present on the command line, dtrace exits when all commands have exited, reporting the exit status for each process as it terminates. The first process-ID is made available to any D programs specified on the command line or using the -s option through the $target macro variable. Refer to Chapter 9, "Scripting" for more information on macro variables.

-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 is compiled but no instrumentation is enabled. If the -l option is present, the program is compiled and the set of probes matched by it is listed, but no instrumentation will be enabled. If neither -e nor -l are present, the instrumentation specified by the D program is enabled and tracing begins.

-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` produces 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 18, "Stability".

-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 20, "Versioning" 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 4, "Actions and Subroutines".

-x

Enable or modify a DTrace runtime option or D compiler option. The options are listed in Chapter 10, "Options and Tunables". 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 mode 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.

Because the -X option affects only how the D compiler invokes the C preprocessor, the -Xa and -Xt options are equivalent from the perspective of D. Both options are provided 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)
- `__sparcv9` (on SPARC systems only when 64–bit programs are compiled)
- `__i386` (on x86 systems only when 32–bit programs are compiled)
- `__amd64` (on x86 systems only when 64–bit programs are compiled)
- `___`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 20, "Versioning" 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.

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 9, "Scripting".

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 that 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.

9

Scripting

You can use the *dtrace(1M)* utility 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.

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 specified 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 the following command in your shell:

```
# dtrace -s interpreter-file
```

The following example shows how to create and execute a dtrace interpreter file. Type the following D source code and save it in a file named interp.d:

```
#!/usr/sbin/dtrace -s

BEGIN
{
```

```
        trace("hello");
        exit(0);
}
```

Mark the `interp.d` file as executable and execute it as follows:

```
# 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(3C) to process 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 10, "Options and Tunables".

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 19   D Macro Variables

Name	Description	Reference
`$[0-9]+`	macro arguments	See "Macro Arguments" on page 174
`$egid`	effective group-ID	getegid(2)

Name	Description	Reference
`$euid`	effective user-ID	geteuid(2)
`$gid`	real group-ID	getgid(2)
`$pid`	process ID	getpid(2)
`$pgid`	process group ID	getpgid(2)
`$ppid`	parent process ID	getppid(2)
`$projid`	project ID	getprojid(2)
`$sid`	session ID	getsid(2)
`$target`	target process ID	See "Target Process ID" on page 175
`$taskid`	task ID	gettaskid(2)
`$uid`	real user-ID	getuid(2)

Except for the `$[0-9]+` macro arguments and the `$target` macro variable, 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.

Using macro variables in interpreter files enables you to create persistent D programs that do not need to be edited each time you want to use them. For example, to count all system calls except those executed by the `dtrace` command, 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.

Macro variables 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 uses the DTrace `pid` provider to instrument the `dtrace` command:

```
pid$pid:libc.so:printf:entry
{
        ...
}
```

Macro variables are only expanded once within each probe description field; they may not contain probe description delimiters (:).

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 Oracle 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
```

D programs can reference unspecified macro arguments if the `defaultargs` option is set. If `defaultargs` is set, unspecified arguments will have the value 0. See Chapter 10, "Options and Tunables" for more information about D compiler options. 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, surround the argument in an extra pair of single quotes to avoid interpretation of the double quotes and string contents by your shell:

```
# ./foo '"a string argument"'
```

If you want your D macro arguments to be interpreted as string tokens even if they match the form of an integer or identifier, prefix the macro variable or argument name with two leading dollar signs (for example, `$$1`) to force the D compiler to interpret the argument value as if it were a string surrounded by double quotes. All the usual D string escape sequences (see Table 6, "D Character Escape Sequences," on page 44) are expanded inside of any string macro arguments, regardless of whether they are referenced using the *$arg* or *$$arg* form of the macro. If the `defaultargs` option is set, unspecified arguments that are referenced with the *$$arg* form have the value of the empty string (`""`).

Target Process ID

Use the `$target` macro variable to create scripts that can be applied to a particular user process of interest that is selected on the `dtrace` command line using the `-p` option or created using the `-c` option. The D programs specified on the command line or using the `-s` option are compiled after processes are created or grabbed and the `$target` variable expands to the integer process-ID of the first such process. For example, the following D script could be used to determine the distribution of system calls executed by a particular subject process:

```
syscall:::entry
/pid == $target/
{
        @[probefunc] = count();
}
```

To determine the number of system calls executed by the *date(1)* command, save the script in the file `syscall.d` and execute the following command:

```
# dtrace -s syscall.d -c date
dtrace: script 'syscall.d' matched 227 probes
Fri Jul 30 13:46:06 PDT 2004
dtrace: pid 109058 has exited

  gtime                                                             1
  getpid                                                            1
  getrlimit                                                         1
  rexit                                                             1
  ioctl                                                             1
  resolvepath                                                       1
  read                                                              1
  stat                                                              1
  write                                                             1
  munmap                                                            1
  close                                                             2
  fstat64                                                           2
  setcontext                                                        2
  mmap                                                              2
  open                                                              2
  brk                                                               4
```

♦ ♦ ♦ CHAPTER 10

10

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. This chapter describes the DTrace options and tunables and the interfaces you can use to modify them.

Consumer Options

DTrace is tuned by setting or enabling options. The available options are described in the table below. For some options, *dtrace(1M)* provides a corresponding command-line option.

TABLE 20   DTrace Consumer Options

Option Name	Value	*dtrace(1M)* Alias	Description	See Chapter
aggrate	time		Rate of aggregation reading	Chapter 3, "Aggregations"
aggsize	size		Aggregation buffer size	Chapter 3, "Aggregations"
aggsortkey	—		Sort aggregation by key order with ties broken by value	Chapter 3, "Aggregations"
aggsortrev	—		Sort aggregation in the reverse order	Chapter 3, "Aggregations"
aggsortpos	position		Position of the aggregate variable on which the output is sorted	Chapter 3, "Aggregations"
aggsortkeypos	position		Position of the aggregate key on which the output is sorted	Chapter 3, "Aggregations"
bufresize	auto or manual		Buffer resizing policy	Chapter 5, "Buffers and Buffering"
bufsize	size	-b	Principal buffer size	Chapter 5, "Buffers and Buffering"
cleanrate	time		Cleaning rate	Chapter 7, "Speculative Tracing"

Option Name	Value	*dtrace(1M)* Alias	Description	See Chapter
cpu	scalar	-c	CPU on which to enable tracing	Chapter 5, "Buffers and Buffering"
defaultargs	—		Allow references to unspecified macro arguments	Chapter 9, "Scripting"
destructive	—	-w	Allow destructive actions	Chapter 4, "Actions and Subroutines"
dynvarsize	size		Dynamic variable space size	"Variables" on page 52
errexit	scalar		Exit on error with a specified status code. However, if no arguments are specified, exits with the status code 1.	
flowindent	—	-F	Indent function entry and prefix with >; unindent function return and prefix with <	Chapter 8, "`dtrace` Utility"
grabanon	—	-a	Claim anonymous state	Chapter 15, "Anonymous Tracing"
jstackframes	scalar		Number of default stack frames jstack	Chapter 4, "Actions and Subroutines"
jstackstrsize	scalar		Default string space size for jstack	Chapter 4, "Actions and Subroutines"
noresolve	—		Prevent resolution of user address symbols	Chapter 4, "Actions and Subroutines"
nspec	scalar		Number of speculations	Chapter 7, "Speculative Tracing"
quiet	—	-q	Output only explicitly traced data	Chapter 8, "`dtrace` Utility"
rawbytes	—		Always print tracemem output in hexadecimal	Chapter 4, "Actions and Subroutines"
specsize	size		Speculation buffer size	Chapter 7, "Speculative Tracing"
strsize	size		String size	"Strings" on page 70
stackframes	scalar		Number of stack frames	Chapter 4, "Actions and Subroutines"
stackindent	scalar		Number of whitespace characters to use when indenting stack and ustack output	Chapter 4, "Actions and Subroutines"
statusrate	time		Rate of status checking	
switchrate	time		Rate of buffer switching	Chapter 5, "Buffers and Buffering"
ustackframes	scalar		Number of user stack frames	Chapter 4, "Actions and Subroutines"

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.

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. The following examples are all 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` command also 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` indicates that the option name is invalid and exits:

```
# 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, `dtrace` will indicate that the value 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, such as `grabanon`, may only be set. The presence of such an option sets it, and you cannot subsequently unset it.

Options that are set for an anonymous enabling will be honored by the DTrace consumer that claims the anonymous state. See Chapter 15, "Anonymous Tracing" for information about enabling anonymous tracing.

11

Providers

This chapter lists and explains the existing DTrace providers.

`dtrace` **Provider**

The `dtrace` provider provides several probes related to DTrace. You can use these probes to initialize state before tracing begins, process state after tracing has completed, and handle unexpected execution errors in other probes.

`BEGIN` **Probe**

The `BEGIN` probe fires before any other probe. No other probe fires until all `BEGIN` clauses are completed. You can use this probe to initialize any state that is required in other probes. The following example shows how to use the `BEGIN` probe to initialize an associative array to map between `mmap` protection bits and a textual representation. For information about the `mmap` system call, see the mmap(2) man page.

```
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. These values should not be relied upon or interpreted to infer any meaningful information. No arguments are defined for the BEGIN probe.

END Probe

The END probe fires after all other probes. This probe will not fire until all other probe clauses have completed. This probe can be used to process state that has been gathered or to format the output. Therefore, the END probe often uses the printa action. You can use the BEGIN and END probes together to measure the total time spent in tracing.

```
BEGIN
{
        start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
        printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

For information about other uses of the END probe, see "Data Normalization" on page 100 and "printa" on page 112.

No arguments are defined for the END probe. The context in which the END probe fires is arbitrary.

When tracing with the bufpolicy option set to fill, adequate space is reserved to accommodate any records traced in the END probe. For more information, see "fill Policy and END Probes" on page 139.

Note - 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. No probes fire during this delay. 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. For more information, see Chapter 10, "Options and Tunables".

ERROR Probe

The ERROR probe fires when a run-time error occurs in executing a clause for a DTrace probe. For example, if a clause attempts to dereference a NULL pointer, the ERROR probe will fire, as shown in the following example.

EXAMPLE 19   error.d: Record Errors

```
BEGIN
{
*(char *)NULL;
}

ERROR
{
printf("Hit an error!");
}
```

When you run this program, you will see the following output:

```
# 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
```

The output shows that the ERROR probe fired, and also illustrates dtrace reporting the error. dtrace has its own enabling of the ERROR probe to allow it to report errors. Use the ERROR probe to customize error handling.

The arguments to the ERROR probe are as follows:

- arg1 – Enabled probe identifier (EPID) of the probe that caused the error
- arg2 – Index of the action that caused the fault
- arg3 – DIF offset into that action or -1 if not applicable
- arg4 – Fault type
- arg5 – Value particular to the fault type

The following table describes the arg4 fault types and the value and its meaning for the arg5.

TABLE 21   Fault Types and Values

arg4 Value	Description
DTRACEFLT_UNKNOWN	Unknown fault type

arg4 Value	Description
DTRACEFLT_BADADDR	Access to unmapped or invalid address
DTRACEFLT_BADALIGN	Unaligned memory access
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 suffici privileges
DTRACEFLT_UPRIV	Attempt to access a user address or property without sufficien privileges
DTRACEFLT_TUPOFLOW	DTrace internal parameter stack overflow
DTRACEFLT_BADSTACK	Invalid user process stack

If the error actions taken in the ERROR probe cause an error, the error is dropped and the ERROR probe is not recursively invoked.

Stability

The dtrace provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 22   DTrace Stability Mechanism

Element	Name stability
Provider	Stable
Module	Private
Function	Private
Name	Stable
Arguments	Stable

lockstat Provider

The lockstat provider provides probes that can be used to discern lock contention statistics, or to understand virtually any aspect of locking behavior. The lockstat command is actually a DTrace consumer that uses the lockstat provider to gather its raw data. For more information, see lockstat(1M) man page.

Overview

The lockstat provider provides two kinds of probes: contention-event probes and hold-event probes.

Contention-event probes correspond to contention on a synchronization primitive. These probes fire when a thread is forced to wait for a resource to become available. Oracle Solaris is generally optimized for the non-contention case, so prolonged contention is not expected. You must use these probes to understand cases of contention. Because contention is relatively rare, enabling contention-event probes does not substantially affect performance.

Hold-event probes correspond to acquiring, releasing, or otherwise manipulating a synchronization primitive. You can use these probes to answer arbitrary questions about the way synchronization primitives are manipulated. Because Oracle Solaris acquires and releases synchronization primitives very often (on the order of millions of times per second per CPU on a busy system), hold-event probes has a much higher probe effect than 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 provides probes that correspond to the different synchronization primitives in Oracle Solaris.

Adaptive Lock Probes

Adaptive locks enforce mutual exclusion to a critical section, and may be acquired in most contexts in the kernel. Because adaptive locks have few context restrictions, they comprise the vast majority of synchronization primitives in the Oracle 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 following table lists the lockstat probes pertaining to adaptive locks. For each probe, arg0 contains a pointer to the kmutex_t structure that represents the adaptive lock.

TABLE 23   Adaptive Lock Probes

Probe	Description
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 enabled, adaptive-block fires *before* adaptive-acquire. At most one of 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 ad spin fires before adaptive-acquire. At most one of adaptive-spin and contains the *spin time*: the number of nanoseconds that were spent in the s

Probe	Description
adaptive-release	Hold-event probe that fires immediately after an adaptive lock is releas

Spin Lock Probes

Threads cannot block in some contexts in the kernel, such as high-level interrupt context and any context manipulating dispatcher state. 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 following table describes the probes related to spin locks.

TABLE 24   Spin Lock Probes

Probe	Description
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 loc spin-acquire. arg1 for spin-spin contains the *spin time*: the number of nan
spin-release	Hold-event probe that fires immediately after a spin lock is released.

Adaptive locks are much more common than spin locks. The following script displays the total for both lock types to provide data to support this observation.

```
lockstat:::adaptive-acquire
/execname == "date"/
{
        @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
        @locks["spin"] = count();
}
```

Run this script in one window, and the `date` command in another window. When you terminate the DTrace script, you will see output similar to the following example:

```
# dtrace -s ./whatlock.d
dtrace: script './whatlock.d' matched 5 probes
^C
spin                                                  26
adaptive                                            2981
```

As this output indicates, over 99 percent of the locks acquired in running the date command are adaptive locks. It may be surprising that so many locks are acquired in doing something

as simple as a date. The large number of locks is a natural artifact of the fine-grained locking required of an extremely scalable system like the Oracle Solaris kernel.

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. .

`thread-spin` is the 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, multiple `thread-spin` probe firings may correspond to a single `spin-acquire` probe firing.

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. These locks 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 25, "Readers/Writer Lock Probes," on page 187. For each probe, `arg0` contains a pointer to the `krwlock_t` structure that represents the adaptive lock.

TABLE 25   Readers/Writer Lock Probes

Probe	Description
rw-acquire	Hold-event probe that fires immediately after a readers/writer lock is acquired. `arg1` lock was acquired as a writer.
rw-block	Contention-event probe that fires after a thread that has blocked on a held readers/wr nanoseconds) that the current thread had to sleep to acquire the lock. `arg2` contains t acquired as a writer. `arg3` and `arg4` contain more information on the reason for block blocked. `arg4` contains the readers count when the current thread blocked. If both th
rw-upgrade	Hold-event probe that fires after a thread has successfully upgraded a readers/writer 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 because they always succeed without contention.

Probe	Description
`rw-release`	Hold-event probe that fires immediately after a readers/writer lock is released. ar if the released lock was held as a writer. Due to upgrades and downgrades, the lo

Stability

The `lockstat` provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 26   `lockstat` Provider Stability

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

profile **Provider**

The `profile` provider provides probes associated with a time-based interrupt firing every fixed, specified time interval. These *unanchored* probes that are not associated with any particular point of execution, but rather with the asynchronous interrupt event. 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. Using DTrace actions, the `profile` provider can be used to sample practically anything in the system. For example, you could sample the state of the current thread, the state of the CPU, or the current system instruction.

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. Table 27, "Valid time suffixes," on page 189 lists the valid suffixes and the units denote by the suffix.

TABLE 27   Valid time suffixes

Suffix	Time Units
nsec or ns	nanoseconds
usec or us	microseconds
msec or ms	milliseconds
sec or s	seconds
min or m	minutes
hour or h	hours
day or d	days
hz	hertz (frequency per second)

EXAMPLE 20   Sample Profile Probe

The following example creates 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 example for a brief period of time results in output similar to the following example:

```
# 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. The following example 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 example script in action, type the following commands in one window:

```
$ echo $$
494621
$ while true ; do let i=0 ; done
```

In another window, run the D script for a brief period of time:

```
# 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 output shows the 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. To see this result, type Control-C in the spinning shell and run the script again:

```
# dtrace -s ./profpri.d 494621
 dtrace: script './profpri.d' matched 1 probe
```

Now in the shell, type a few characters. When you terminate the DTrace script, output like the following example will appear:

```
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.

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. 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, such as providing some periodic output or taking a periodic action.

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 descriptions imply, if `arg0` is non-zero then `arg1` is zero; if `arg0` is zero then `arg1` is non-zero. Thus, you can use `arg0` and `arg1` to differentiate user-level from kernel level, as in this example:

```
profile-1ms
{
        @ticks[arg0 ? "kernel" : "user"] = count();
}
```

Timer Resolution

The `profile` provider uses arbitrary resolution interval timers in the operating system. On architectures that do not support arbitrary resolution time-based interrupts, the frequency is 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 would 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.

EXAMPLE 21   Sample Timer Resolution for profile Probe

The following example tests a given architecture's resolution:

```
profile-5000
{
        /*
         * Divide by 1,000,000 to convert nanoseconds to milliseconds, and
         * then 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 example script 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 example script will yield an 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. Many variants of the x86 architecture (`i86pc`) also support arbitrary resolution `profile` probes, although some older variants do not.

Probe Creation

Unlike other providers, the `profile` provider creates probes dynamically based on the requirement. Thus, the desired profile probe might not appear in a listing of all probes but the probe will be created when it is explicitly enabled. You can use the `dtrace -l -P profile` to list all the probes.

On architectures that support arbitrary resolution `profile` probes, a time interval that is too short would cause the system to continuously field time-based interrupts, thereby denying service on the system. To prevent this situation, the `profile` provider will silently refuse to create any probe that would result in an interval of less than two hundred microseconds.

Stability

The `profile` provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 28   Stability Mechanism for the `profile` Provider

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

cpc Provider

The cpc provider makes available probes associated with CPU performance counter events. A probe fires when a specified number of events of a given type in a chosen processor mode have occurred. When a probe fires you can sample aspects of the system state and you can make inferences to the system behavior. Accurate inferences are possible when high sampling rates or long sampling times are employed.

Probes

Probes made available by the cpc provider have the following format:

cpc:::*event name*-*mode*[-*attributes*]-*count*

The format of attributes is:

attr1_val1[-*attr2_val2...*]|*value*

The definitions and detail of the components of the probe name are listed in the following table.

TABLE 29   Probe Name Components

Component	Meaning
event name	The platform specific or generic event name. A full list of events can be obtained using the -h option with cpustat.
mode	The privilege mode in which to count events. Valid modes are "user" for user mode events, "kernel" for kernel mode events and "all" for both user mode and kernel mode events.
attributes	This component is optional and accepts one or more event attributes. On some platforms it is possible to specify event attributes to further refine a platform specific event specification. The attributes can only be specified for platform specific events.
	The attributes are specified as name-value pairs or single value in the following format:
	attr1_val1[-*attr2_val2...*]|*value*
	■ The attributes (attr1, attr2, and so on) are the string names of the platform-specific attributes.
	■ The values (val1, val2, and so on) are the hex values for the corresponding attributes.
	■ If only a value without attribute name is specified, it is interpreted as a mask value. The mask value is commonly referred to as a unit mask or event mask.
	■ Available attribute names can be obtained through -h option with the cpustat command.
	■ The nouser and the sys attributes are not accepted and should be specified at the mode component.
count	The number of events that must occur on a CPU for a probe to be fired on that CPU.

A sample usage of the `cpc` provider is as follows:

```
cpc:::BU_fill_req_missed_L2-all-umask_0x7-cmask_0x0-10000
```

In this example, the parameters are set to the following values:

- Event name is set to `BU_fill_req_missed_L2`
- Mode is set to `all`
- Attributes are set to `umask = 0x7` and `cmask = 0x0`
- Count is set to `10000`

The following introductory example fires a probe on a CPU for every 10000 user-mode Level 1 instruction cache misses on a SPARC platform. When the probe fires, record the name of the executable that was on processor at the time the probe fires.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

cpc:::IC_miss-user-10000
{
        @[execname] = count();
}

END
{
        trunc(@, 10);
}

# ./user-l1miss.d
^C

  dirname                                              8
  firefox                                              8
  sed                                                 11
  intrd                                               12
  run-mozilla.sh                                      13
  java                                                64
  sshd                                               135
  gconfd-2                                           569
  thunderbird-bin                                   1666
  firefox-bin                                       2060
```

Note - When working with the `cpc` provider it is important to remember that the state available when a probe fires is valid for the performance counter event that caused the probe to fire and not for all events counted with that probe.The above output shows that the `firefox-bin` application caused the `cpc:::IC_miss-user-10000` probe to fire 2060 times. As this probe fires once for every 10000 level 1 instruction cache misses on a CPU, the `firefox-bin` application could have contributed anywhere from 2060 to 20600000 of these misses.

For more examples, see "Examples of cpc Provider Usage" on page 197.

Arguments

The arguments to cpc probes are listed in table below.

TABLE 30   Probe Arguments

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 descriptions imply, if arg0 is non-zero then arg1 is zero; if arg0 is zero then arg1 is non-zero.

Probe Availability

CPU performance counters are a finite resource and the number of probes that can be enabled depends upon hardware capabilities. Processors that cannot determine which counter has overflowed when multiple counters are programmed, such as AMD and UltraSPARC, are only allowed to have a single enabling at any one time. On such platforms, consumers attempting to enable more than 1 probe will fail as will consumers attempting to enable a probe when a disparate enabling already exists. Processors that can detect which counter has overflowed, such as Niagara2 and Intel P4, are allowed to have as many probes enabled as the hardware will allow. This will be, at most, the number of counters available on a processor. On such configurations, multiple probes can be enabled at any one time.

Probes are enabled by consumers on a first-come, first-served basis. When hardware resources are fully utilized subsequent enablings will fail until resources become available.

Probe Creation

Like the profile provider, the cpc provider creates probes dynamically on an as-needed basis. Thus, the desired cpc probe might not appear in a listing of all probes but the probe will be created when it is explicitly enabled. You can use dtrace -l -P cpc for listing all the cpc probes.

Specifying a small event overflow count for frequently occurring events, such as cycle count and instructions executed, would quickly render the system unusable as a processor would be continuously servicing performance counter overflow interrupts. To prevent this situation, the smallest overflow count that can be specified for any probe is set, by default, at 5000. This can be altered by adjusting the dcpc-min-overflow variable in the /kernel/drv/dcpc.conf configuration file and then unloading and reloading the dcpc driver.

Note - Specify high frequency events such as instructions executed or cycle count. For example, measuring busy cycles on a fully utilized 3GHz processor with a count of 50000 would generate approximately 65000 interrupts/sec. This rate of interrupt delivery could degrade system performance to some degree.

Co-existence With Existing Tools

The provider has priority over per-LWP libcpc usage, that is cputrack, for access to counters. In the same manner as cpustat, enabling probes causes all existing per-LWP counter contexts to be invalidated. As long as enabled probes remain active, the counters will remain unavailable to cputrack-type consumers.

Only one of cpustat and DTrace may use the counter hardware at any one time. Ownership of the counters is given on a first-come, first-served basis.

Examples of cpc Provider Usage

Examples of cpc provider usage follow.

EXAMPLE 22   user-insts.d

The script displays instructions executed by applications on an AMD platform

```
cpc:::FR_retired_x86_instr_w_excp_intr-user-10000
{
        @[execname] = count();
}

# ./user-insts.d
dtrace: script './user-insts.d' matched 1 probe
^C
[chop]
  init                                                            138
  dtrace                                                          175
```

```
    nis_cachemgr                                              179
    automountd                                                183
    intrd                                                     235
    run-mozilla.sh                                            306
    thunderbird                                               316
    Xorg                                                      453
    thunderbird-bin                                          2370
    sshd                                                     8114
```

EXAMPLE 23   kern-cycles.d

The following example shows a kernel profiled by cycle usage on an AMD platform.

```
cpc:::BU_cpu_clk_unhalted-kernel-10000
{
        @[func(arg0)] = count();
}

# ./kern-cycles.d
dtrace: script './kern-cycles.d' matched 1 probe
^C
[chop]
  genunix`vpm_sync_pages                                   478948
  genunix`vpm_unmap_pages                                  496626
  genunix`vpm_map_pages                                    640785
  unix`mutex_delay_default                                 916703
  unix`hat_kpm_page2va                                     988880
  tmpfs`rdtmp                                              991252
  unix`hat_page_setattr                                   1077717
  unix`page_try_reclaim_lock                              1213379
  genunix`free_vpmap                                      1914810
  genunix`get_vpmap                                       2417896
  unix`page_lookup_create                                 3992197
  unix`mutex_enter                                        5595647
  unix`do_copy_fault_nta                                 27803554
```

EXAMPLE 24   brendan-l2miss.d

This example describes user-mode L2 cache misses and the functions that generated them on an AMD platform. The predicate ensures that you only sample function names when the probe was fired by the brendan executable.

```
cpc:::BU_fill_req_missed_L2-all-0x7-10000
/execname == "brendan"/
{
 @[ufunc(arg1)] = count();
}

./brendan-l2miss.d
dtrace: script './brendan-l2miss.d' matched 1 probe
```

```
CPU    ID                    FUNCTION:NAME
^C

  brendan`func_gamma                                           930
  brendan`func_beta                                           1578
  brendan`func_alpha                                          2945
```

You can have the same result with the following probe name format.

```
cpc:::BU_fill_req_missed_L2-all-umask_0x7-10000
/ execname == "brendan" /
{
@[ufunc(arg1)] = count();
}
```

EXAMPLE 25   `brendan-generic-l2miss.d`

This example descibes a generic event `PAPI_l2_dcm` to indicate the interest in L2 data cache misses instead of the platform event.

```
cpc:::PAPI_l2_dcm-all-10000
/execname == "brendan"/
{
        @[ufunc(arg1)] = count();
}

# ./breandan-generic-l2miss.d
dtrace: script './brendan-generic-l2miss.d' matched 1 probe
^C

  brendan`func_gamma                                          1681
  brendan`func_beta                                           2521
  brendan`func_alpha                                          5068
```

EXAMPLE 26   `off_core_event.d`

The following example probes offcore event on an Intel platform:

```
cpc:::off_core_response_0-all-msr_offcore_0x3001-10000
{
@[execname] = count();
}

# ./off_core_event.d
dtrace: script './off_core_event.d' matched 1 probe
^C

fmd                                                              3
fsflush                                                         36
sched                                                          175
```

EXAMPLE   27   l2miss.d

Multiple attributes are allowed using '-' between attributes.

The following example sets two attributes to probe L2 miss event in an AMD platform.

```
cpc:::BU_fill_req_missed_L2-all-umask_0x7-cmask_0x0-10000
{
@[execname] = count();
}

# ./l2miss.d
dtrace: script './l2miss.d' matched 1 probe


automountd                                                 1
dtrace                                                     1
fmd                                                        1
in.routed                                                  1
netcfgd                                                    1
nscd                                                       1
sendmail                                                   1
utmpd                                                      1
kcfd                                                       2
syslogd                                                    2
uname                                                      2
file                                                       3
ls                                                         3
sshd                                                       4
zfs                                                        9
bash                                                      10
ksh93                                                     10
ssh                                                       22
fsflush                                                   34
sched                                                     68
beadm                                                    146
```

Stability

The cpc provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 31   Stability Mechanism for the cpc Provider

Element	Name Stability	Data Stability	Dependency Class
Provider	Evolving	Evolving	Common
Module	Private	Private	Unknown

Element	Name Stability	Data Stability	Dependency Class
Function	Private	Private	Unknown
Name	Evolving	Evolving	CPU
Arguments	Evolving	Evolving	Common

fbt Provider

This chapter describes the Function Boundary Tracing (FBT) provider, which provides probes associated with the entry to and return from most functions in the Oracle Solaris kernel. The function is the fundamental unit of program text. In a well-designed system, each function performs a discrete and well-defined operation on a specified object or series of like objects. Therefore, even on the smallest Oracle Solaris systems, FBT will provide on the order of 20,000 probes.

Similar to other DTrace providers, FBT has no probe effect when it is not explicitly enabled. When enabled, FBT only induces a probe effect in probed functions. While the FBT implementation is highly specific to the instruction set architecture, FBT has been implemented on both SPARC and x86 platforms. For each instruction set, there are a small number of functions that do not call other functions and are highly optimized by the compiler, called leaf functions, that cannot be instrumented by FBT. Probes for these functions are not present in DTrace.

Effective use of FBT probes requires knowledge of the operating system implementation. Therefore, it is recommended that you use FBT only when developing kernel software or when other providers are not sufficient. Other DTrace providers, including syscall, sched, proc, and io, can be used to answer most system analysis questions without requiring operating system implementation knowledge.

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 FBT probes specify a function name and module name.

Probe arguments

The following probe arguments are supported for FBT provider.

`entry` **probes**

The arguments to `entry` probes are the same as the arguments to the corresponding operating system kernel 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.

`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. You are usually interested in either the value that a function returned or the fact that the function returned at all rather than the specific return path taken. FBT therefore collects a function's multiple return sites into a single `return` probe. If you want to know the exact return path, you can examine the `return` probe `args[0]` value, which indicates the *offset* in bytes of the returning instruction in the function text.

If the function has a return value, the return value is stored in `args[1]`. If a function does not have a return value, `args[1]` is not defined.

Examples

You can use FBT to easily explore the kernel's implementation. The following example script records the first `ioctl` from any `xclock` process and then follows the subsequent code path through the kernel.

```
/*
 * To make the output more readable, 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 results in output similar to the following example:

```
# 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
```

The output shows that an xclock process called ioctl on a file descriptor that appears to be associated with a socket.

You can also use FBT when trying to understand kernel drivers. For example, the ssd driver has many code paths by which EIO may be returned. FBT can be easily used to determine the precise code path that resulted in an error condition, as shown in the following example:

```
fbt:ssd::return
/arg1 == EIO/
{
        printf("%s+%x returned EIO.", probefunc, arg0);
}
```

For more information on any one return of `EIO`, you might want to speculatively trace all `fbt` probes, and then `commit` or `discard` based on the return value of a specific function. For more information about speculative tracing, see Chapter 7, "Speculative Tracing".

Alternatively, you can use FBT to understand the functions called within a specified module. The following example lists 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 the function is being called. For example, `putnext` takes a pointer to a queue
structure as its first member. The `q_qinfo` member of the `queue` structure is a pointer to a
`qinit` structure. The `qi_minfo` member of the `qinit` structure has a pointer to a `module_info`
structure, which contains the module name in its `mi_idname` member. For more information, see
putnext(9F), queue(9S), qinit(9S), and module_info(9S) man pages.

The following example puts this information together by using the FBT probe in `putnext` to
track `putnext` calls by module name:

```
fbt::putnext:entry
{
        @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

Running the above script results in output similar to the following example:

```
# 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. The following example shows how to determine the callers of the DDI delaying routines drv_usecwait and delay. For more information, see drv_usecwait(9F) and delay(9F) man pages.

```
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 example script is particularly interesting to run during boot. For information about procedures performing anonymous tracing during system boot, see Chapter 15, "Anonymous Tracing". Upon reboot, you might see output similar to the following example:

```
# 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
```

Tail-call Optimization

When one function ends by calling another function, the compiler can engage in *tail-call optimization*, in which the function being called reuses the caller's stack frame. This procedure is most commonly used in the SPARC architecture, where the compiler reuses the caller's register window in the function being called in order to minimize register window pressure.

The presence of this optimization causes the `return` probe of the calling function to fire *before* the `entry` probe of the called function. This ordering can lead to quite a bit of confusion. For example, if you wanted to record all functions called from a particular function and any functions that this function calls, you might use the following script:

```
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 any functions that it calls, will not be captured. The kernel cannot be dynamically de-optimized, and DTrace does not engage in a lie about how code is structured. Therefore, you must be aware of when tail-call optimization might be used.

Tail-call optimization is likely to be used in source code similar to the following example:

```
 return (bar());
```

Tail-call optimization can also be used in a source code similar to the following example:

```
        (void) bar();
        return;
```

Conversely, function source code that ends like the following example *cannot* have its call to `bar` optimized, because the call to `bar` is not a tail-call:

```
        bar();
        return (rval);
```

You can determine whether a call has been tail-call optimized using the following techniques:

- While running DTrace, trace `arg0` of the `return` probe in question. `arg0` contains the offset of the returning instruction in the function.
- After DTrace has stopped, use `mdb` 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.

Due to the instruction set architecture, tail-call optimization is far more common on SPARC systems than on x86 systems. The following example uses `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 this command is running, run a program that performs a dup(2), such as a `bash` process. The above command should provide output similar to the following example:

```
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
```

The output shows that `dup+0x10` is a call to the `fcntl` function and not a `ret` instruction. Therefore, the call to `fcntl` is an example of tail-call optimization.

Assembly Functions

You might observe functions that seem to enter but never return or vice versa. Such rare functions are generally hand-coded assembly routines that branch to the middle of other hand-coded assembly functions. These functions should not impede analysis: the branched-to function must still return to the caller of the branched-from function. That is, if you enable all FBT probes, you should see the entry to one function and the return from another function at the same stack depth.

Instruction Set Limitations

Some functions cannot be instrumented by FBT. The exact nature of uninstrumentable functions is specific to the instruction set architecture.

x86 Limitations

Functions that do not create a stack frame on x86 systems cannot be instrumented by FBT. Because the register set for x86 is extraordinarily small, most functions must put data on the stack and therefore create a stack frame. However, some x86 functions do not create a stack frame and therefore cannot be instrumented.

SPARC Limitations

Leaf routines hand-coded in assembly language on SPARC systems cannot be instrumented by FBT. The majority of the kernel is written in C, and all functions written in C can be instrumented by FBT.

Breakpoint Interaction

FBT works by dynamically modifying kernel text. Because kernel breakpoints also work by modifying kernel text, 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. In this situation, the breakpoint will trigger first, and then the probe will fire when the debugger resumes the kernel. It is recommended that kernel breakpoints not be used concurrently with DTrace. If breakpoints are required, use the DTrace `breakpoint` action instead.

Module Loading

The Oracle Solaris kernel can dynamically load and unload kernel modules. 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 *no enabled* 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.

Stability

The FBT provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 32   Stability Mechanism for the FBT Provider

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`, which 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.

`syscall` Provider

The `syscall` provider makes available a probe at the entry to and return from every system call in the system. Because 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.

Probes

`syscall` provides a pair of probes for each system call: an `entry` probe that fires before the system call is entered, and a `return` probe 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 `/etc/name_to_sysnum` file. Often, the system call names provided by `syscall` correspond to names in Section 2 of the man pages. However, some probes provided by the `syscall` provider do not directly correspond to any documented system call. The common reasons for this discrepancy are described in this section.

System Call 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` in `/etc/name_to_sysnum` is `rexit`. Similarly, the name of `time` is `gtime`, and the name of `execve` is `exece`.

Subcoded System Calls

Some system calls as presented in man page section 2 are implemented as suboperations of an undocumented system call. For example, the system calls related to System V semaphores, such as `semctl`, `semget`, `semids`, `semop`, and `semtimedop` are 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`

New System Calls

Unlike Oracle Solaris 10 where some of them were not implemented at all and the remainder were implemented as subcodes of a single private system call, Oracle Solaris 11 implements the following system interfaces as individual system calls:

- `faccessat()`
- `fchmodat()`
- `fchownat()`
- `fstatat()`

- linkat()
- mkdirat()
- mknodat()
- openat()
- readlinkat()
- renameat()
- symlinkat()
- unlinkat()

These system calls implement a superset of the functionality of their old non-at-suffixed counterparts. They take an additional first argument that is either an open directory file descriptor, in which case the operation on a relative pathname is taken relative to the specified directory, or is the reserved value AT_FDCWD, in which case the operation takes place relative to the current working directory.

Deleted System Calls

In Oracle Solaris 11, the following old system calls have been removed from the system. The libc interfaces remain, but they are reimplemented not as system calls in their own right, but as calls to the new system calls as indicated:

access(p, m)	faccessat(AT_FDCWD, p, m, 0)
chmod(p, m)	fchmodat(AT_FDCWD, p, m, 0)
chown(p, u, g)	fchownat(AT_FDCWD, p, u, g, 0)
creat(p, m)	openat(AT_FDCWD, p, O_WRONLY | O_CREAT | O_TRUNC, m)
fchmod(fd, m)	fchmodat(fd, NULL, m, 0)
fchown(fd, u, g)	fchownat(fd, NULL, u, g, 0)
fstat(fd, s)	fstatat(fd, NULL, s, 0)
lchown(p, u, g)	fchownat(AT_FDCWD, p, u, g, AT_SYMLINK_NOFOLLOW)
link(p1, p2)	linkat(AT_FDCWD, p1, AT_FDCWD, p2, 0)
lstat(p, s)	fstatat(AT_FDCWD, p, s, AT_SYMLINK_NOFOLLOW)
mkdir(p, m)	mkdirat(AT_FDCWD, p, m)
mknod(p, m. d)	mknodat(AT_FDCWD, p, m, d)
open(p, o, m)	openat(AT_FDCWD, p, o, m)
readlink(p, b, s)	readlinkat(AT_FDCWD, p, b, s)
rename(p1, p2)	renameat(AT_FDCWD, p1, AT_FDCWD, p2)
rmdir(p)	unlinkat(AT_FDCWD, p, AT_REMOVEDIR)
stat(p, s)	fstatat(AT_FDCWD, p, s, 0)
symlink(p1, p2)	symlinkat(p1, AT_FDCWD, p2)

unlink(p)	unlinkat(AT_FDCWD, p, 0)

Large File System Calls

A 32-bit program that supports *large files* that exceed two gigabytes in size must be able to process 64-bit file offsets. Because large files require use of large offsets, these files are manipulated through a parallel set of system interfaces. For more information, see the lf64(5) man page. These interfaces are documented in lf64, but they do not have individual man pages. Each of these large file system call interfaces appears as its own syscall probe as shown in the following table.

TABLE 33   sycall Large File Probes

Large File syscall Probe	System Call
fstatat64	fstatat
fstatvfs64	fstatvfs
getdents64	fgetdents
getrlimit64	getrlimit
mmap64	mmap
openat64	openat
pread64	pread
pwrite64	pwrite
setrlimit64	setrlimit
statvfs64	statvfs

Private System Calls

Some system calls are private implementation details of Oracle Solaris subsystems that span the user-kernel boundary. As such, these system calls do not have man pages in man page 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. For more information, see the fuser(1M) man page.

Arguments

For entry probes, the arguments arg0 .. argn are the arguments to the system call. For return probes, both arg0 and arg1 contain the return value. A non-zero value in the D variable errno indicates system call failure.

Stability

The syscall provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table. For more information about the stability mechanism, refer to Chapter 18, "Stability".

TABLE 34 Stability Mechanism for the syscall Provider

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

sdt Provider

The Statically Defined Tracing (SDT) provider creates probes at sites that a software programmer has formally designated. The SDT mechanism allows programmers to consciously choose locations of interest to users of DTrace and to convey some semantic knowledge about each location through the probe name. The Oracle Solaris kernel has defined a handful of SDT probes. DTrace also provides a mechanism for user application developers to define static probes. For more information, see Chapter 13, "Statically Defined Tracing for User Applications".

Probes

The SDT probes defined by the Oracle Solaris kernel are listed in the following table. The name stability and data stability of these probes are both Private because their description here thus reflects the kernel's implementation and should not be inferred to be an interface commitment. For more information about the DTrace stability mechanism, see Chapter 18, "Stability".

TABLE 35 SDT Probes

Probe name	Description	arg0
callout-start	Fires immediately before executing a callout (see <sys/callo.h>). Callouts are executed by periodic system clock, and represent the implementation for timeout.	Pointer to the callout_t (see <sys/callo.h>) corresponding to the callout to be executed.

Probe name	Description	arg0
callout-end	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	Fires immediately before calling into a device's interrupt handler.	Pointer to the dev_info structure (see <sys/ddi_impldefs.h>) corresponding to the interrupting device.
interrupt-complete	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.

Examples

The following example is a 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 example reveals the frequent users of timeout in the system, as shown in the following output:

```
# dtrace -s ./callout.d
                        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
          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
          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 interface only produces a single timer expiration. Consumers of timeout requiring interval timer functionality typically reinstall their timeout from their timeout handler. The following example shows this behavior:

```
#pragma D option quiet

sdt:::callout-start
{
        self->callout = ((callout_t *)arg0)->c_func;
}

fbt::timeout:entry
/self->callout && arg2 <= 100/
{
        /*
         * This case focusses interval timeout(9F)s that
         * are short.  Therfore, perform 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 script and waiting several seconds before typing Control-C results in output similar to the following example:

```
# 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
```

The output shows that uhci_handle_root_hub_status_change in the uhci driver represents the shortest interval timer on the system: it is called every system clock tick.

The interrupt-start probe can be used to understand interrupt activity. The following example shows how 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 script results in output similar to the following example:

```
# 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
```

Creating SDT Probes

If you are a device driver developer, you might be interested in creating your own SDT probes in your Oracle Solaris driver. The disabled probe effect of SDT is essentially the cost of several

no-operation system instructions. You are therefore encouraged to add SDT probes to your device drivers as needed. Unless these probes negatively affect performance, you can leave them in your shipping code.

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 substitution allows C code to provide macro names that are not valid C identifiers without specifying a string.

DTrace includes the kernel module name and function name as part of the tuple identifying a probe, so you do not need to include this information in the probe name to prevent name space collisions. You can use the command `dtrace -l -P sdt -m` *module* on your driver *module* to list the probes you have installed and the full names that will be seen by users of DTrace.

Probe Arguments

The arguments for each SDT probe are the arguments specified in the corresponding `DTRACE_PROBE`*n* macro reference. 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 not dereferencing pointers and not 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, so DTrace users can request these actions only when they are needed.

Stability

The SDT provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 36   Stability Mechanism for the SDT provider

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

`mib` Provider

The `mib` provider makes available probes that correspond to counters in the Oracle Solaris management information bases (MIBs). MIB counters are used by the simple network management protocol (SNMP) that allow remote monitoring of heterogeneous networking entities. You can also view the counters with the `kstat` and `netstat` commands. For more information, see `kstat(1M)` and `netstat(1M)` man pages. The `mib` provider facilitates quick exploration of aberrant networking behavior that is observed using either remote or local networking monitors.

Probes

The `mib` provider makes available probes for counters from several MIBs. The protocols that export MIBs instrumented by the `mib` provider are listed in Table 37, "`mib` probes," on page 221. The table includes a reference to documentation that specifies some or all of the MIB, the name of the kernel statistic that may be used to access the running counts using the `kstat` `-n` *statistic* option, and a reference to the table that has a complete definition of the probes. All MIB counters are also available using the `-s` option to `netstat`.

TABLE 37   `mib` probes

Protocol	MIB Description	Kernel Statistic	`mib` **Probes Table**
ICMP	RFC 1213	`icmp`	Table 38, "ICMP `mib` Probes," on page 222
IP	RFC 4293	`ip`	Table 39, "IP `mib` Probes," on page 223
IPsec	—	`ip`	Table 40, "IPsec `mib` Probes," on page 224
IPv6	RFC 4293	—	Table 41, "IPv6 `mib` Probes," on page 224
SCTP	SCTP MIB (Internet draft)	`sctp`	Table 43, "SCTP `mib` Probes," on page 227

TCP	RFC 1213	`tcp`	Table 44, "TCP `mib` Probes," on page 228
UDP	RFC 1213	`udp`	Table 45, "UDP `mib` Probes," on page 230

TABLE 38   ICMP `mib` Probes

Name of the Probe	Description
`icmpInAddrMaskReps`	Fires whenever an ICMP Address Mask Reply message is received.
`icmpInAddrMasks`	Fires whenever an ICMP Address Mask Request message is received.
`icmpInBadRedirects`	Fires whenever an ICMP Redirect message is received that is determined to be malformed in s off-link, and the like).
`icmpInCksumErrs`	Fires whenever an ICMP message with a bad checksum is received.
`icmpInDestUnreachs`	Fires whenever an ICMP Destination Unreachable message is received.
`icmpInEchoReps`	Fires whenever an ICMP Echo Reply message is received.
`icmpInEchos`	Fires whenever an ICMP Echo request message is received.
`icmpInErrors`	Fires whenever an ICMP message is received that is determined to have an ICMP-specific err
`icmpInFragNeeded`	Fires whenever an ICMP Destination Unreachable (Fragmentation Needed) message is receive was larger than some MTU and the `Do not Fragment` flag was set.
`icmpInMsgs`	Fires whenever an ICMP message is received. Whenever this probe fires, the `icmpInErrors` p have an ICMP-specific error.
`icmpInOverflows`	Fires whenever an ICMP message is received, but the message is subsequently dropped due to
`icmpInParmProbs`	Fires whenever an ICMP Parameter Problem message is received.
`icmpInRedirects`	Fires whenever an ICMP Redirect message is received.
`icmpInSrcQuenchs`	Fires whenever an ICMP Source Quench message is received.
`icmpInTimeExcds`	Fires whenever an ICMP Time Exceeded message is received.
`icmpInTimestampReps`	Fires whenever an ICMP Timestamp Reply message is received.
`icmpInTimestamps`	Fires whenever an ICMP Timestamp request message is received.
`icmpInUnknowns`	Fires whenever an ICMP message of unknown type is received.
`icmpOutAddrMaskReps`	Fires whenever an ICMP Address Mask Reply message is sent.
`icmpOutDestUnreachs`	Fires whenever an ICMP Destination Unreachable message is sent.
`icmpOutDrops`	Fires whenever an outbound ICMP message is dropped for some reason (such as memory allo destination, and the like).
`icmpOutEchoReps`	Fires whenever an ICMP Echo Reply message is sent.
`icmpOutErrors`	Fires whenever an ICMP message is not sent due to problems discovered within ICMP, such a are discovered outside the ICMP layer, such as the inability of IP to route the resulting datagra
`icmpOutFragNeeded`	Fires whenever an ICMP Destination Unreachable (Fragmentation Needed) message is sent.
`icmpOutMsgs`	Fires whenever an ICMP message is sent. Whenever this probe fires, the `icmpOutErrors` prob have ICMP-specific errors.
`icmpOutParmProbs`	Fires whenever an ICMP Parameter Problem message is sent.
`icmpOutRedirects`	Fires whenever an ICMP Redirect message is sent. For a system, this probe will never fire, be
`icmpOutTimeExcds`	Fires whenever an ICMP Time Exceeded message is sent.

Name of the Probe	Description
Fires whenever an ICMP Timestamp Reply message is sent.	`icmpOutTimestampReps`

TABLE 39   IP `mib` Probes

`ipForwDatagrams`	Fires whenever a datagram is received that does not have this system as its final IP destination, and an attempt is made to find a route to forward the datagram to that final destination. On systems that do not act as IP gateways, this probe will only fire for those packets that are source-routed through this system, and for which the source-route option processing was successful.
`ipForwProhibits`	Fires whenever a datagram is received that does not have this system as its final IP destination, but because the system is not permitted to act as a router, no attempt is made to find a route to forward the datagram to that final destination.
`ipFragCreates`	Fires whenever an IP datagram fragment is generated as a result of fragmentation.
`ipFragFails`	Fires whenever an IP datagram is discarded because it could not be fragmented, for example, because fragmentation was required and the `Do not Fragment` flag was set.
`ipFragOKs`	Fires whenever an IP datagram has been successfully fragmented.
`ipInCksumErrs`	Fires whenever an input datagram is discarded due to a bad IP header checksum.
`ipInDelivers`	Fires whenever an input datagram is successfully delivered to IP user protocols, including ICMP.
`ipInDiscards`	Fires whenever an input IP datagram is discarded for reasons unrelated to the packet, such as for lack of buffer space. This probe does not fire for any datagram discarded while awaiting reassembly.
`ipInHdrErrors`	Fires whenever an input datagram is discarded due to an error in its IP header, including a version number mismatch, a format error, an exceeded time-to-live, an error discovered in processing IP options, and the like.
`ipInIPv6`	Fires whenever an IPv6 packet erroneously arrives on an IPv4 queue.
`ipInReceives`	Fires whenever a datagram is received from an interface, even if that datagram is received in error.
`ipInUnknownProtos`	Fires whenever a locally addressed datagram is received successfully but subsequently discarded because of an unknown or unsupported protocol.
`ipOutDiscards`	Fires whenever an output IP datagram is discarded for reasons unrelated to the packet (for example, for lack of buffer space). This probe will fire for a packet counted in the `ipForwDatagrams` MIB counter if the packet meets such a (discretionary) discard criterion.
`ipOutIPv6`	Fires whenever an IPv6 packet is sent over an IPv4 connection.
`ipOutNoRoutes`	Fires whenever an IP datagram is discarded because no route could be found to transmit it to its destination. This probe will fire for a packet counted in the `ipForwDatagrams` MIB counter if the packet meets this "no-route"; criterion. This probe will also fire for any datagrams which cannot be routed because all default gateways are down.
`ipOutRequests`	Fires whenever an IP datagram is supplied to IP for transmission from local IP user protocols (include ICMP). Note that this probe will not fire for any packet counted in the `ipForwDatagrams` MIB counter.
`ipOutSwitchIPv6`	Fires whenever a connection changes from using IPv4 to using IPv6 as its IP protocol.
`ipReasmDuplicates`	Fires whenever the IP reassembly algorithm determines that an IP fragment contains *only* previously received data.

ipReasmFails	Fires whenever any failure is detected by the IP reassembly algorithm. This probe does not necessarily fire for every discarded IP fragment because some algorithms, notably the algorithm in RFC 815, can lose track of fragments by combining them as they are received.
ipReasmOKs	Fires whenever an IP datagram is successfully reassembled.
ipReasmPartDups	Fires whenever the IP reassembly algorithm determines that an IP fragment contains both some previously received data and some new data.
ipReasmReqds	Fires whenever an IP fragment is received that needs to be reassembled.

TABLE 40   IPsec mib Probes

ipsecInFailed	Fires whenever a received packet is dropped because it fails to match the specified IPsec policy.
ipsecInSucceeded	Fires whenever a received packet matches the specified IPsec policy and processing is allowed to continue.

TABLE 41   IPv6 mib Probes

ipv6ForwProhibits	Fires whenever an IPv6 datagram is received that does not have this system as its final IPv6 destination, but because the system is not permitted to act as a router, no attempt is made to find a route to forward the datagram to that final destination.
ipv6IfIcmpBadHoplimit	Fires whenever an ICMPv6 neighbor discovery protocol message is received that is found to have a Hop Limit less than the defined maximum. Such messages might not have originated from a neighbor, and are therefore discarded.
ipv6IfIcmpInAdminProhibs	Fires whenever an ICMPv6 Destination Unreachable (Communication Administratively Prohibited) message is received.
ipv6IfIcmpInBadNeighborAdvertisements	Fires whenever an ICMPv6 Neighbor Advertisement message is received that is malformed in some way.
ipv6IfIcmpInBadNeighborSolicitations	Fires whenever an ICMPv6 Neighbor Solicit message is received that is malformed in some way.
ipv6IfIcmpInBadRedirects	Fires whenever an ICMPv6 Redirect message is received that is malformed in some way.
ipv6IfIcmpInDestUnreachs	Fires whenever an ICMPv6 Destination Unreachable message is received.
ipv6IfIcmpInEchoReplies	Fires whenever an ICMPv6 Echo Reply message is received.
ipv6IfIcmpInEchos	Fires whenever an ICMPv6 Echo request message is received.
ipv6IfIcmpInErrors	Fires whenever an ICMPv6 message is received that is determined to have an ICMPv6-specific error (such as bad ICMPv6 checksum, bad length, and the like).
ipv6IfIcmpInGroupMembBadQueries	Fires whenever an ICMPv6 Group Membership Query message is received that is malformed in some way.
ipv6IfIcmpInGroupMembBadReports	Fires whenever an ICMPv6 Group Membership Report message is received that is malformed in some way.
ipv6IfIcmpInGroupMembOurReports	Fires whenever an ICMPv6 Group Membership Report message is received.
ipv6IfIcmpInGroupMembQueries	Fires whenever an ICMPv6 Group Membership Query message is received.
ipv6IfIcmpInGroupMembReductions	Fires whenever an ICMPv6 Group Membership Reduction message is received.
ipv6IfIcmpInGroupMembResponses	Fires whenever an ICMPv6 Group Membership Response message is received.
ipv6IfIcmpInGroupMembTotal	Fires whenever an ICMPv6 multicast listener discovery message is received.
ipv6IfIcmpInMsgs	Fires whenever an ICMPv6 message is received. When this probe fires, the ipv6IfIcmpInErrors probe might also fire if the message has an ICMPv6-specific error.

`ipv6IfIcmpInNeighborAdvertisements`	Fires whenever an ICMPv6 Neighbor Advertisement message is received.
`ipv6IfIcmpInNeighborSolicits`	Fires whenever an ICMPv6 Neighbor Solicit message is received.
`ipv6IfIcmpInOverflows`	Fires whenever an ICMPv6 message is received, but that message is subsequently dropped due to lack of buffer space.
`ipv6IfIcmpInParmProblems`	Fires whenever an ICMPv6 Parameter Problem message is received.
`ipv6IfIcmpInRedirects`	Fires whenever an ICMPv6 Redirect message is received.
`ipv6IfIcmpInRouterAdvertisements`	Fires whenever an ICMPv6 Router Advertisement message is received.
`ipv6IfIcmpInRouterSolicits`	Fires whenever an ICMPv6 Router Solicit message is received.
`ipv6IfIcmpInTimeExcds`	Fires whenever an ICMPv6 Time Exceeded message is received.
`ipv6IfIcmpOutAdminProhibs`	Fires whenever an ICMPv6 Destination Unreachable (Communication Administratively Prohibited) message is sent.
`ipv6IfIcmpOutDestUnreachs`	Fires whenever an ICMPv6 Destination Unreachable message is sent.
`ipv6IfIcmpOutEchoReplies`	Fires whenever an ICMPv6 Echo Reply message is sent.
`ipv6IfIcmpOutEchos`	Fires whenever an ICMPv6 Echo message is sent.
`ipv6IfIcmpOutErrors`	Fires whenever an ICMPv6 message is not sent due to problems discovered within ICMPv6, such as a lack of buffers. This probe will not fire if errors are discovered outside the ICMPv6 layer, such as the inability of IPv6 to route the resulting datagram.
`ipv6IfIcmpOutGroupMembQueries`	Fires whenever an ICMPv6 Group Membership Query message is sent.
`ipv6IfIcmpOutGroupMembReductions`	Fires whenever an ICMPv6 Group Membership Reduction message is sent.
`ipv6IfIcmpOutGroupMembResponses`	Fires whenever an ICMPv6 Group Membership Response message is sent.
`ipv6IfIcmpOutMsgs`	Fires whenever an ICMPv6 message is sent. When this probe fires, the `ipv6IfIcmpOutErrors` probe might also fire if the message has ICMPv6-specific errors.
`ipv6IfIcmpOutNeighborAdvertisements`	Fires whenever an ICMPv6 Neighbor Advertisement message is sent.
`ipv6IfIcmpOutNeighborSolicits`	Fires whenever an ICMPv6 Neighbor Solicitation message is sent.
`ipv6IfIcmpOutParmProblems`	Fires whenever an ICMPv6 Parameter Problem message is sent.
`ipv6IfIcmpOutPktTooBigs`	Fires whenever an ICMPv6 Packet Too Big message is sent.
`ipv6IfIcmpOutRedirects`	Fires whenever an ICMPv6 Redirect message is sent. For a system, this probe will never fire, because systems do not send redirects.
`ipv6IfIcmpOutRouterAdvertisements`	Fires whenever an ICMPv6 Router Advertisement message is sent.
`ipv6IfIcmpOutRouterSolicits`	Fires whenever an ICMPv6 Router Solicit message is sent.
`ipv6IfIcmpOutTimeExcds`	Fires whenever an ICMPv6 Time Exceeded message is sent.
`ipv6InAddrErrors`	Fires whenever an input datagram is discarded because the IPv6 address in their IPv6 header's destination field is not a valid address to be received by this entity. This probe will fire for invalid addresses (for example, ::0) and for unsupported addresses (for example, addresses with unallocated prefixes). For machines that are not configured to act as IPv6 routers and therefore do not forward datagrams, this probe will fire for datagrams discarded because the destination address was not a local address.
`ipv6InDelivers`	Fires whenever an input datagram is successfully delivered to IPv6 user-protocols (including ICMPv6).
`ipv6InDiscards`	Fires whenever an input IPv6 datagram is discarded for reasons unrelated to the packet (for example, for lack of buffer space). This probe does not fire for any datagram discarded while awaiting reassembly.

`ipv6InHdrErrors`	Fires whenever an input datagram is discarded due to an error in its IPv6 header, including a version number mismatch, a format error, an exceeded hop count, an error discovered in processing IPv6 options, and the like.
`ipv6InIPv4`	Fires whenever an IPv4 packet erroneously arrives on an IPv6 queue.
`ipv6InMcastPkts`	Fires whenever a multicast IPv6 packet is received.
`ipv6InNoRoutes`	Fires whenever a routed IPv6 datagram is discarded because no route could be found to transmit it to its destination. This probe will *only* fire for packets that have originated externally.
`ipv6InReceives`	Fires whenever an IPv6 datagram is received from an interface, even if that datagram is received in error.
`ipv6InTooBigErrors`	Fires whenever a fragment is received that is larger than the maximum fragment size.
`ipv6InTruncatedPkts`	Fires whenever an input datagram is discarded because the datagram frame did not carry enough data.
`ipv6InUnknownProtos`	Fires whenever a locally-addressed IPv6 datagram is received successfully but subsequently discarded because of an unknown or unsupported protocol.
`ipv6OutDiscards`	Fires whenever an output IPv6 datagram is discarded for reasons unrelated to the packet (for example, for lack of buffer space). This probe will fire for a packet counted in the `ipv6OutForwDatagrams` MIB counter if the packet meets such a (discretionary) discard criterion.
`ipv6OutForwDatagrams`	Fires whenever a datagram is received that does not have this system as its final IPv6 destination, and an attempt is made to find a route to forward the datagram to that final destination. On a system that does not act as an IPv6 router, this probe will only fire for those packets that are source-routed through the system, and for which the source-route option processing was successful.
`ipv6OutFragCreates`	Fires whenever an IPv6 datagram fragment is generated as a result of fragmentation.
`ipv6OutFragFails`	Fires whenever an IPv6 datagram is discarded because it could not be fragmented, for example, because its `Do not Fragment` flag was set.
`ipv6OutFragOKs`	Fires whenever an IPv6 datagrams has been successfully fragmented.
`ipv6OutIPv4`	Fires whenever an IPv6 packet is sent over an IPv4 connection.
`ipv6OutMcastPkts`	Fires whenever a multicast packet is sent.
`ipv6OutNoRoutes`	Fires whenever an IPv6 datagram is discarded because no route could be found to transmit it to its destination. This probe will *not* fire for packets that have originated externally.
`ipv6OutRequests`	Fires whenever an IPv6 datagram is supplied to IPv6 for transmission from local IPv6 user protocols (including ICMPv6). This probe will not fire for any packet counted in the `ipv6ForwDatagrams` MIB counter.
`ipv6OutSwitchIPv4`	Fires whenever a connection changes from using IPv6 to using IPv4 as its IP protocol.
`ipv6ReasmDuplicates`	Fires whenever the IPv6 reassembly algorithm determines that an IPv6 fragment contains *only* previously received data.
`ipv6ReasmFails`	Fires whenever a failure is detected by the IPv6 reassembly algorithm. This probe does not necessarily fire for every discarded IPv6 fragment since some algorithms can lose track of fragments by combining them as they are received.
`ipv6ReasmOKs`	Fires whenever an IPv6 datagram is successfully reassembled.
`ipv6ReasmPartDups`	Fires whenever the IPv6 reassembly algorithm determines that an IPv6 fragment contains both some previously received data and some new data.
`ipv6ReasmReqds`	Fires whenever an IPv6 fragment is received that needs to be reassembled.

TABLE 42   Raw IP mib Probes

rawipInCksumErrs	Fires whenever a raw IP packet is received that has a bad IP checksum.
rawipInDatagrams	Fires whenever a raw IP packet is received.
rawipInErrors	Fires whenever a raw IP packet is received that is malformed in some way.
rawipInOverflows	Fires whenever a raw IP packet is received, but that packet is subsequently dropped due to lack of buffer space.
rawipOutDatagrams	Fires whenever a raw IP packet is sent.
rawipOutErrors	Fires whenever a raw IP packet is not sent due to some error condition, typically because the raw IP packet was malformed in some way.

TABLE 43   SCTP mib Probes

sctpAborted	Fires whenever an SCTP association has made a direct transition to the CLOSED state from any state using the ABORT primitive, denoting ungraceful termination of the association.
sctpActiveEstab	Fires whenever an SCTP association has made a direct transition to the ESTABLISHED state from the COOKIE-ECHOED state, denoting that the upper layer has initiated the association attempt.
sctpChecksumError	Fires whenever an SCTP packet is received from peers with an invalid checksum.
sctpCurrEstab	Fires whenever an SCTP association is tallied as a part of reading the sctpCurrEstab MIB counter. An SCTP association is tallied if its current state is ESTABLISHED, SHUTDOWN-RECEIVED, or SHUTDOWN-PENDING.
sctpFragUsrMsgs	Fires whenever a user message has to be fragmented because of the MTU.
sctpInClosed	Fires whenever data is received on a closed SCTP association.
sctpInCtrlChunks	Fires whenever the sctpInCtrlChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpInDupAck	Fires whenever a duplicate ACK is received.
sctpInInvalidCookie	Fires whenever an invalid cookie is received.
sctpInOrderChunks	Fires whenever the sctpInOrderChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpInSCTPPkts	Fires whenever the sctpInSCTPPkts MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpInUnorderChunks	Fires whenever the sctpInUnorderChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpListenDrop	Fires whenever an incoming connection is dropped for any reason.
sctpOutAck	Fires whenever a selective acknowledgement is sent.
sctpOutAckDelayed	Fires whenever delayed acknowledgement processing is performed for an SCTP association. Any acknowledgements sent as a part of delayed acknowledgement processing will cause the sctpOutAck probe to fire.
sctpOutCtrlChunks	Fires whenever the sctpOutCtrlChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].

sctpOutOfBlue	Fires whenever an otherwise correct SCTP packet is received for which the receiver is not able to identify the association to which the packet belongs.
sctpOutOrderChunks	Fires whenever the sctpOutOrderChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in aRgs[0].
sctpOutSCTPPkts	Fires whenever the sctpOutSCTPPkts MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpOutUnorderChunks	Fires whenever the sctpOutUnorderChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpOutWinProbe	Fires whenever a window probe is sent.
sctpOutWinUpdate	Fires whenever a window update is sent.
sctpPassiveEstab	Fires whenever SCTP associations have made a direct transition to the ESTABLISHED state from the CLOSED state. The remote endpoint has initiated the association attempt.
sctpReasmUsrMsgs	Fires whenever the sctpReasmUsrMsgs MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpRetransChunks	Fires whenever the sctpRetransChunks MIB counter is updated, either because the MIB counter is explicitly queried or because an SCTP connection is closed. The value by which the MIB counter is to be increased is in args[0].
sctpShutdowns	Fires whenever an SCTP association makes the direct transition to the CLOSED state from either the SHUTDOWN-SENT state or the SHUTDOWN-ACK-SENT state, denoting graceful termination of the association.
sctpTimHeartBeatDrop	Fires whenever an SCTP association is aborted due to failure to receive a heartbeat acknowledgement.
sctpTimHeartBeatProbe	Fires whenever an SCTP heartbeat is sent.
sctpTimRetrans	Fires whenever timer-based retransmit processing is performed on an association.
sctpTimRetransDrop	Fires whenever prolonged failure to perform timer-based retransmission results in the association being aborted.

TABLE 44   TCP mib Probes

Name of the Probe	Description
tcpActiveOpens	Fires whenever a TCP connection makes a direct transition from the CLOSED state to the SY
tcpAttemptFails	Fires whenever a TCP connection makes a direct transition to the CLOSED state from either t state and whenever a TCP connection makes a direct transition to the LISTEN state from the S
tcpCurrEstab	Fires whenever a TCP connection is tallied as a part of reading the tcpCurrEstab MIB counte state is either ESTABLISHED or CLOSE_WAIT.
tcpEstabResets	Fires whenever a TCP connection makes the direct transition to the CLOSED state from either CLOSE_WAIT state.
tcpHalfOpenDrop	Fires whenever a connection is dropped due to a full queue of connections in the SYN_RCVD
tcpInAckBytes	Fires whenever an ACK is received for previously sent data. The number of bytes acknowledg
tcpInAckSegs	Fires whenever an ACK is received for a previously sent segment.
tcpInAckUnsent	Fires whenever an ACK is received for an unsent segment.

Name of the Probe	Description
tcpInClosed	Fires whenever data was received for a connection in a closing state.
tcpInDataDupBytes	Fires whenever a segment is received such that all data in the segment has been previously receiv... duplicated segment is passed in `args[0]`.
tcpInDataDupSegs	Fires whenever a segment is received such that all data in the segment has been previously receiv... duplicated segment is passed in `args[0]`.
tcpInDataInorderBytes	Fires whenever data is received such that *all* data prior to the new data's sequence number has be... bytes received in-order is passed in `args[0]`.
tcpInDataInorderSegs	Fires whenever a segment is received such that *all* data prior to the new segment's sequence numl...
tcpInDataPartDupBytes	Fires whenever a segment is received such that some of the data in the segment has been previous... segment is new. The number of duplicate bytes is passed in `args[0]`.
tcpInDataPartDupSegs	Fires whenever a segment is received such that some of the data in the segment has been previous... segment is new. The number of duplicate bytes is passed in `args[0]`.
tcpInDataPastWinBytes	Fires whenever data is received that lies past the current receive window. The number of bytes is ...
tcpInDataPastWinSegs	Fires whenever a segment is received that lies past the current receive window.
tcpInDataUnorderBytes	Fires whenever data is received such that some data prior to the new data's sequence number is m... unordered is passed in `args[0]`.
tcpInDataUnorderSegs	Fires whenever a segment is received such that some data prior to the new data's sequence numbe...
tcpInDupAck	Fires whenever a duplicate ACK is received.
tcpInErrs	Fires whenever a TCP error (for example, a bad TCP checksum) is found on a received segment.
tcpInSegs	Fires whenever a segment is received, even if that segment is later found to have an error that pre...
tcpInWinProbe	Fires whenever a window probe is received.
tcpInWinUpdate	Fires whenever a window update is received.
tcpListenDrop	Fires whenever an incoming connection is dropped due to a full listen queue.
tcpListenDropQ0	Fires whenever a connection is dropped due to a full queue of connections in the SYN_RCVD sta...
tcpOutAck	Fires whenever an ACK is sent.
tcpOutAckDelayed	Fires whenever an ACK is sent after having been initially delayed.
tcpOutControl	Fires whenever a SYN, FIN, or RST is sent.
tcpOutDataBytes	Fires whenever data is sent. The number of bytes sent is in `args[0]`.
tcpOutDataSegs	Fires whenever a segment is sent.
tcpOutFastRetrans	Fires whenever a segment is retransmitted as part of the fast retransmit algorithm.
tcpOutRsts	Fires whenever a segment is sent with the RST flag set.
tcpOutSackRetransSegs	Fires whenever a segment is retransmitted on a connection that has selective acknowledgement er...
tcpOutSegs	Fires whenever a segment is sent that contains at least one non-retransmitted byte.
tcpOutUrg	Fires whenever a segment is sent with the URG flag set, and with a valid urgent pointer.
tcpOutWinProbe	Fires whenever a window probe is sent.
tcpOutWinUpdate	Fires whenever a window update is sent.
tcpPassiveOpens	Fires whenever a TCP connections have made a direct transition to the SYN_RCVD state from th...
tcpRetransBytes	Fires whenever data is retransmitted. The number of bytes retransmitted is in `args[0]`.
tcpRetransSegs	Fires whenever a segment is sent that contains one or more retransmitted bytes.
tcpRttNoUpdate	Fires whenever data was received, but there was no timestamp information available with which t...

Name of the Probe	Description
tcpRttUpdate	Fires whenever data was received containing the timestamp information necessary to update t
tcpTimKeepalive	Fires whenever timer-based keep-alive processing is performed on a connection.
tcpTimKeepaliveDrop	Fires whenever keep-alive processing results in termination of a connection.
tcpTimKeepaliveProbe	Fires whenever a keep-alive probe is sent out as a part of keep-alive processing.
tcpTimRetrans	Fires whenever timer-based retransmit processing is performed on a connection.
tcpTimRetransDrop	Fires whenever prolonged failure to perform timer-based retransmission results in termination

TABLE 45   UDP `mib` Probes

udpInCksumErrs	Fires whenever a datagram is discarded due to a bad UDP checksum.
udpInDatagrams	Fires whenever a UDP datagram is received.
udpInErrors	Fires whenever a UDP datagram is received, but is discarded due to either a malformed packet header or the failure to allocate an internal buffer.
udpInOverflows	Fires whenever a UDP datagram is received, but subsequently dropped due to lack of buffer space.
udpNoPorts	Fires whenever a UDP datagram is received on a port to which no socket is bound.
udpOutDatagrams	Fires whenever a UDP datagram is sent.
udpOutErrors	Fires whenever a UDP datagram is not sent due to some error condition, typically because the datagram was malformed in some way.

Arguments

The sole argument for each `mib` probe has the same semantics: `args[0]` contains the value with which the counter is to be incremented. For most `mib` probes, `args[0]` always contains the value `1`, but for some probes `args[0]` may take arbitrary positive values. For these probes, the meaning of `args[0]` is noted in the probe description.

Stability

The `mib` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 46   Stability Mechanism for the `mib` Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA

Element	Name stability	Data stability	Dependency class
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

`fpuinfo` Provider

The `fpuinfo` provider includes probes that correspond to the simulation of floating-point instructions on SPARC microprocessors. While most floating-point instructions are executed in hardware, some floating-point operations trap into the operating system for simulation. The conditions under which floating-point operations require operating system simulation are specific to a microprocessor implementation. The operations that require simulation are rare. However, if an application uses one of these operations frequently, the effect on performance could be severe. The `fpuinfo` provider enables rapid investigation of floating-point simulation seen by using the `kstat` and the `fpu_info` kernel statistic or `trapstat` and the `fp-xcp-other` trap. For more information, see `trapstat(1M)`.

Probes

The `fpuinfo` provider makes available a probe for each type of floating-point instruction that can be simulated. The `fpuinfo` provider has a Name Stability of `CPU`; the names of the probes are specific to a microprocessor implementation, and might not be available on different microprocessors within the same family. For example, some of the probes listed might only be available on UltraSPARC-III and not UltraSPARC-III+ or vice versa.

The `fpuinfo` probes are described in Table 47, "fpuinfo Probes," on page 231.

TABLE 47   `fpuinfo` Probes

`fpu_sim_fitoq`	Fires whenever an `fitoq` instruction is simulated by the kernel.
`fpu_sim_fitod`	Fires whenever an `fitod` instruction is simulated by the kernel.
`fpu_sim_fitos`	Fires whenever an `fitos` instruction is simulated by the kernel.
`fpu_sim_fxtoq`	Fires whenever an `fxtoq` instruction is simulated by the kernel.
`fpu_sim_fxtod`	Fires whenever an `fxtod` instruction is simulated by the kernel.
`fpu_sim_fxtos`	Fires whenever an `fxtos` instruction is simulated by the kernel.
`fpu_sim_fqtox`	Fires whenever an `fqtox` instruction is simulated by the kernel.
`fpu_sim_fdtox`	Fires whenever an `fdtox` instruction is simulated by the kernel.
`fpu_sim_fstox`	Fires whenever an `fstox` instruction is simulated by the kernel.

fpu_sim_fqtoi	Fires whenever an fqtoi instruction is simulated by the kernel.
fpu_sim_fdtoi	Fires whenever an fdtoi instruction is simulated by the kernel.
fpu_sim_fstoi	Fires whenever an fstoi instruction is simulated by the kernel.
fpu_sim_fsqrtq	Fires whenever an fsqrtq instruction is simulated by the kernel.
fpu_sim_fsqrtd	Fires whenever an fsqrtd instruction is simulated by the kernel.
fpu_sim_fsqrts	Fires whenever an fsqrts instruction is simulated by the kernel.
fpu_sim_fcmpeq	Fires whenever an fcmpeq instruction is simulated by the kernel.
fpu_sim_fcmped	Fires whenever an fcmped instruction is simulated by the kernel.
fpu_sim_fcmpes	Fires whenever an fcmpes instruction is simulated by the kernel.
fpu_sim_fcmpq	Fires whenever an fcmpq instruction is simulated by the kernel.
fpu_sim_fcmpd	Fires whenever an fcmpd instruction is simulated by the kernel.
fpu_sim_fcmps	Fires whenever an fcmps instruction is simulated by the kernel.
fpu_sim_fdivq	Fires whenever an fdivq instruction is simulated by the kernel.
fpu_sim_fdivd	Fires whenever an fdivd instruction is simulated by the kernel.
fpu_sim_fdivs	Fires whenever an fdivs instruction is simulated by the kernel.
fpu_sim_fdmulx	Fires whenever an fdmulx instruction is simulated by the kernel.
fpu_sim_fsmuld	Fires whenever an fsmuld instruction is simulated by the kernel.
fpu_sim_fmulq	Fires whenever an fmulq instruction is simulated by the kernel.
fpu_sim_fmuld	Fires whenever an fmuld instruction is simulated by the kernel.
fpu_sim_fmuls	Fires whenever an fmuls instruction is simulated by the kernel.
fpu_sim_fsubq	Fires whenever an fsubq instruction is simulated by the kernel.
fpu_sim_fsubd	Fires whenever an fsubd instruction is simulated by the kernel.
fpu_sim_fsubs	Fires whenever an fsubs instruction is simulated by the kernel.
fpu_sim_faddq	Fires whenever an faddq instruction is simulated by the kernel.
fpu_sim_faddd	Fires whenever an faddd instruction is simulated by the kernel.
fpu_sim_fadds	Fires whenever an fadds instruction is simulated by the kernel.
fpu_sim_fnegd	Fires whenever an fnegd instruction is simulated by the kernel.
fpu_sim_fnegq	Fires whenever an fneqq instruction is simulated by the kernel.
fpu_sim_fnegs	Fires whenever an fnegs instruction is simulated by the kernel.
fpu_sim_fabsd	Fires whenever an fabsd instruction is simulated by the kernel.
fpu_sim_fabsq	Fires whenever an fabsq instruction is simulated by the kernel.
fpu_sim_fabss	Fires whenever an fabss instruction is simulated by the kernel.
fpu_sim_fmovd	Fires whenever an fmovd instruction is simulated by the kernel.
fpu_sim_fmovq	Fires whenever an fmovq instruction is simulated by the kernel.
fpu_sim_fmovs	Fires whenever an fmovs instruction is simulated by the kernel.
fpu_sim_fmovr	Fires whenever an fmovr instruction is simulated by the kernel.
fpu_sim_fmovcc	Fires whenever an fmovcc instruction is simulated by the kernel.

Arguments

There are no arguments to `fpuinfo` probes.

Stability

The `fpuinfo` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 48   Stability Mechanism for the `fpuinfo` Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	CPU
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	CPU
Arguments	Evolving	Evolving	CPU

`pid` Provider

The `pid` provider enables 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. When probes are enabled, the probes only induce probe effect on those processes that are traced.

Note - When the compiler inlines a function, the `pid` provider's probe does not fire. Use one of the following methods to compile a particular C function so that it will not be inlined.

- Sun Studio: #pragma no_inline (funcname[, funcname])
- gcc: funcname __attribute__ ((noinline))

  Consult your compiler documentation for updates.

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. The following example uses `mdb` to display a list of objects:

```
$ 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 can also omit the `.1` or `so.1` suffix. All of the following examples name the same probe:

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

The first example is the actual name of the probe. The other examples are convenient aliases that are replaced with the full load object name internally.

For the load object of the executable, you can use the alias `a.out`. The following 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 application binary might have several names for the same function. For example, `mutex_lock` might be an alternate name for the function `pthread_mutex_lock` in `libc.so.1`. DTrace chooses one canonical name for such functions and uses 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 means that the names of the probes you enable may be slightly different than those actually enabled. The canonical name will always be consistent between runs of DTrace on systems running the same Oracle Solaris release.

For examples about how to use the `pid` provider effectively, see Chapter 12, "User Process Tracing"..

Function Boundary Probes

The `pid` provider enables you to 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 manual that use the FBT provider to trace kernel function calls can be modified slightly to apply to user processes.

`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.

`return` Probes

A `return` probe 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.

In 32-bit processes, for functions that return either a signed `long long` or an unsigned `long long` type, `arg1` contains only half of the total bits in the return value. The other half of the total bits returned is available in `arg2`. On SPARC systems, `arg1` contains the upper 32-bits of the return value and `arg2` contains the lower 32-bits of the return value. On x86 systems, `arg1` contains the lower 32-bits of the return value and `arg2` contains the upper 32-bits of the return value.

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 the following command:

```
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. The uregs[] array will help you to examine process state at these probe sites. For more information, see "uregs[] Array" on page 382.

Stability

The pid provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 49   Stability Mechanisms for the pid Provider

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

plockstat Provider

The plockstat provider makes available probes that can be used to observe the behavior of user-level synchronization primitives including lock contention and hold times. The plockstat command is a DTrace consumer that uses the plockstat provider to gather data on user-level locking events. For more information, see the plockstat(1M) man page.

Overview

The plockstat provider makes available probes for the following types of events:

- Contention Events – These probes correspond to contention on a user-level synchronization primitive, and fire when a thread is forced to wait for a resource to become available. Oracle 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 rare, enabling contention-event probes generally does not have a serious probe effect; they can be enabled without concern for substantially affecting performance.

- Hold Events – These probes correspond to acquiring, releasing or otherwise manipulating a user-level synchronization primitive. As such, these probes can be used to answer arbitrary questions about the way user-level synchronization primitives are manipulated. Because applications typically acquire and release synchronization primitives very often, enabling hold-event probes can have a greater probe effect than 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 applications.
- Error Events – These probes correspond to any kind of anomalous behavior encountered when acquiring or releasing a user-level synchronization primitive. These events can be used to detect errors encountered while a thread is blocking on a user-level synchronization primitive. Error events should be extremely uncommon so enabling them would not induce a serious probe effect.

Mutex Probes

Mutexes enforce mutual exclusion to critical sections. When a thread attempts to acquire a mutex held by another thread using `mutex_lock` or `pthread_mutex_lock`, it will determine if the owning thread is running on a different CPU. If it is, the acquiring thread will *spin* for a short while waiting for the mutex to become available. If the owner is not executing on another CPU, the acquiring thread will *block*.

The four `plockstat` probes pertaining to mutexes are listed in the following table . For each probe, `arg0` contains a pointer to the `mutex_t` or `pthread_mutex_t` structure (these are identical types) that represents the mutex.

TABLE 50   Mutex Probes

Name of the Probe	Description
mutex-acquire	Hold event probe that fires immediately after a mutex is acquired. `arg1` contains a boolean value that indic recursive on a recursive mutex. `arg2` indicates the number of iterations that the acquiring thread spent spin zero only if the `mutex-spin` probe fired on this mutex acquisition.
mutex-block	Contention event probe that fires before a thread blocks on a held mutex. Both `mutex-block` and `mutex-sp acquisition.
mutex-spin	Contention event probe that fires before a thread begins spinning on a held mutex. Both `mutex-block` and lock acquisition.
mutex-release	Hold event probe that fires immediately after an mutex is released. `arg1` contains a boolean value that indic to a recursive release on a recursive mutex.
mutex-error	Error event probe that fires when an error is encountered on a mutex operation. `arg1` is the `errno` value for

Reader/Writer Lock Probes

Reader/writer locks enable multiple readers *or* a single writer, but not both, to be in a critical section at one time. These locks are typically used for structures that are searched more frequently than they are modified, or when threads spend substantial time in a critical section. Users interact with reader/writer locks using the Oracle Solaris `rwlock` or POSIX `pthread_rwlock_init` interfaces.

The probes pertaining to readers/writer locks are described in the following table. For each probe, `arg0` contains a pointer to the `rwlock_t` or `pthread_rwlock_t` structure that represents the adaptive lock. `arg1` contains a boolean value that indicates whether the operation was as a writer.

TABLE 51   Readers/Writer Lock Probes

Name o the Probe	Description
rw-acquire	Hold event probe that fires immediately after a readers/writer lock is acquired.
rw-block	Contention event probe that fires before a thread blocks while attempting to acquire a lock. If enabled, the probe or the `rw-error` probe will fire after `rw-block`.
rw-release	Hold event probe that fires immediately after a reader/writer lock is released.
rw-error	Error event probe that fires when an error is encountered during a reader/writer lock operation. `arg1` is the the error encountered.

Stability

The `plockstat` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 52   Stability Mechanism for the `ploclstat` Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

fasttrap Provider

The fasttrap provider allows for tracing at specific, preprogrammed user process locations. Unlike most other DTrace providers, the fasttrap provider is not designed for tracing system activity. Rather, this provider is meant as a way for DTrace consumers to inject information into the DTrace framework by activating the fasttrap probe.

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 available.

Stability

The fasttrap provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 53   Stability Mechanism for the fasttrap Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

sysinfo Provider

The sysinfo provider include probes that correspond to kernel statistics classified by the name sys. Because these statistics provide the input for system monitoring utilities like mpstat, the sysinfo provider enables quick exploration of observed aberrant behavior.

Probes

The sysinfo provider makes available probes that correspond to the fields in the sys named kernel statistic: a probe provided by sysinfo fires immediately before the corresponding sys value is incremented. The following example shows how to display both the names and the current values of the sys named kernel statistic using the kstat command. For more information about the kstat command, see the kstat(1M) man page.

```
$ 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 the following table.

TABLE 54   sysinfo Probes

Name of the Probe	Description
bawrite	Probe that fires whenever a buffer is about to be asynchronously written out to a device.
bread	Fires whenever a buffer is physically read from a device. bread fires *after* the buffer has been requested fro completion.
bwrite	Fires whenever a buffer is about to be written out to a device, whether synchronously or asynchronously.
cpu_ticks_idle	Fires when the periodic system clock has made the determination that a CPU is *idle*. Note that this probe f fires on the CPU running the system clock. The cpu_t argument (arg2) indicates the CPU that has been de ” on page 242.
cpu_ticks_kernel	Fires when the periodic system clock has made the determination that a CPU is executing in the *kernel*. Th and therefore fires on the CPU running the system clock. The cpu_t argument (arg2) indicates the CPU th For more information, see "Arguments " on page 242.
cpu_ticks_user	Fires when the periodic system clock has made the determination that a CPU is executing in *user mode*. Th and therefore fires on the CPU running the system clock. The cpu_t argument (arg2) indicates the CPU th For more information, see "Arguments " on page 242.
cpu_ticks_wait	Fires when the periodic system clock has made the determination that a CPU is otherwise idle, but some th fires in the context of the system clock and therefore fires on the CPU running the system clock. The cpu_ deemed waiting on I/O. For more information, see "Arguments " on page 242.
idlethread	Fires whenever a CPU enters the idle loop.
intrblk	Fires whenever an interrupt thread blocks.
inv_swtch	Fires whenever a running thread is forced to involuntarily give up the CPU.
lread	Fires whenever a buffer is logically read from a device.
lwrite	Fires whenever a buffer is logically written to a device.

Name of the Probe	Description
modload	Fires whenever a kernel module is loaded.
modunload	Fires whenever a kernel module is unloaded.
msg	Fires whenever a msgsnd or msgrcv system call is made, but before the message queue operations have been p
mutex_adenters	Fires whenever an attempt is made to acquire an owned adaptive lock. If this probe fires, one of the lockstat probes will also fire. For more information, see "Stability" on page 188.
namei	Fires whenever a name lookup is attempted in the filesystem.
nthreads	Fires whenever a thread is created.
phread	Fires whenever a raw I/O read is about to be performed.
phwrite	Fires whenever a raw I/O write is about to be performed.
procovf	Fires whenever a new process cannot be created because the system is out of process table entries.
pswitch	Fires whenever a CPU switches from executing one thread to executing another.
readch	Fires after each successful read, but before control is returned to the thread performing the read. A read may c calls. arg0 contains the number of bytes that were successfully read.
rw_rdfails	Fires whenever an attempt is made to read-lock a readers/writer when the lock is either held by a writer, or de lockstat provider's rw-block probe will also fire. For more information, see "Stability" on page 188.
rw_wrfails	Fires whenever an attempt is made to write-lock a readers/writer lock when the lock is held either by some nu probe fires, the lockstat provider's rw-block probe will also fire. For more information, see "Stability" on p
sema	Fires whenever a semop system call is made, but before any semaphore operations have been performed.
sysexec	Fires whenever an exec system call is made.
sysfork	Fires whenever a fork system call is made.
sysread	Fires whenever a read, readv, or pread system call is made.
sysvfork	Fires whenever a vfork system call is made.
syswrite	Fires whenever a write, writev, or pwrite system call is made.
trap	Fires whenever a processor trap occurs. Note that some processors, in particular UltraSPARC variants, handle that does not cause this probe to fire.
ufsdirblk	Fires whenever a directory block is read from the UFS file system. For more information, see the ufs(7FS) m
ufsiget	Fires whenever an inode is retrieved.
ufsinopage.	Fires after an in-core inode without any associated data pages has been made available for reuse.
ufsipage	Fires after an in-core inode with associated data pages has been made available for reuse. This probe fires afte disk.
wait_ticks_io	Fires when the periodic system clock has made the determination that a CPU is otherwise idle but some threa fires in the context of the system clock and therefore fires on the CPU running the system clock. The cpu_t ar described as waiting for I/O. For more information about arg2, see "Arguments " on page 242. No semantic d cpu_ticks_wait; wait_ticks_io exists solely for historical reasons.
writech	Fires after each successful write, but before control is returned to the thread performing the write. A write ma system calls. arg0 contains the number of bytes that were successfully written.
xcalls	Fires whenever a cross-call is about to be made. A cross-call is the operating system's mechanism for one CPU

Arguments

The arguments to sysinfo probes are as follows:

TABLE 55   sysinfo Probe Arguments

arg0	The value by which the statistic is to be incremented. For most probes, this argument is always 1, but for some probes this argument 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 enables 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.

The value of arg0 is 1 for most sysinfo probes. However, the readch and writech probes set arg0 to the number of bytes read or written, respectively. This features permits you to determine the size of reads by executable name, as shown in the following example:

```
# 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
```

The sysinfo provider sets arg2 to be 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 probes do not. The exceptional probes include cpu_ticks_idle, cpu_ticks_kernel, cpu_ticks_user and cpu_ticks_wait, which always fire on the CPU executing the system clock. Use the cpu_id member of the cpu_t structure to determine the CPU of interest. The following D script 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 script results in output similar to the following example:

```
# 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
```

Example

Examine the following output from mpstat:

```
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
```

From the above output, you might conclude that the xcal field seems too high, especially given the relative idleness of the system. mpstat determines the value in the xcal field by examining the xcalls field of thesys kernel statistic. This aberration can therefore be explored easily by enabling the xcalls sysinfo probe, as shown in the following example:

```
# 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
```

The output shows where to look for the source of the cross-calls. Some number of find
processes are causing the majority of the cross-calls. The following D script can be used to
understand the problem in further detail:

```
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, such as those resulting from page faults, might not emanate from system
calls. The script prints <none> in these cases. Running the script results in output similar to the
following example:

```
# dtrace -s ./find.d
 dtrace: script './find.d' matched 444 probes
^C
  <none>                                                     2
  lstat64                                                 2433
  getdents64                                             14873
```

This output indicates that the majority of cross-calls induced by find are in turn induced by getdents system calls. Further exploration would depend on the direction you want to explore. If you want to understand why find processes are making calls to getdents, you could write a D script to aggregate on ustack when find induces a cross-call. If you want to understand why calls to getdents are inducing cross-calls, you could write a D script to aggregate on stack when find induces a cross-call. Whatever your next step, the presence of the xcalls probe has enabled you to quickly discover the root cause of the unusual monitoring output.

Stability

The sysinfo provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 56   Stability Mechanism for the sysinfo Provider

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

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, the vminfo provider enables quick exploration of observed aberrant behavior.

Probes

The vminfo provider makes available probes that correspond to the fields in the vm named
kernel statistic: 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 kernel
statistic, use the kstat command, as shown in the following 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 57   vminfo Probes

Name of the Probe	Description
anonfree	Fires whenever an unmodified anonymous page is freed as part of paging activity. Anonymous p associated with a file. Memory containing such pages includes heap memory, stack memory, or mapping zero.
anonpgin	Fires whenever an anonymous page is paged in from a swap device.
anonpgout	Fires whenever a modified anonymous page is paged out to a swap device.
as_fault	Fires whenever a fault is taken on a page and the fault is neither a protection fault nor a copy-on
cow_fault	Fires whenever a copy-on-write fault is taken on a page. arg0 contains the number of pages that copy-on-write.
dfree	Fires whenever a page is freed as a result of paging activity. Whenever dfree fires, exactly one fsfree will also subsequently fire.
execfree	Fires whenever an unmodified executable page is freed as a result of paging activity.
execpgin	Fires whenever an executable page is paged in from the backing store.
execpgout	Fires whenever a modified executable page is paged out to the backing store. Most paging of ex of execfree. execpgout can only fire if an executable page is modified in memory, an uncomm
fsfree	Fires whenever an unmodified file system data page is freed as part of paging activity.
fspgin	Fires whenever a file system page is paged in from the backing store.
fspgout	Fires whenever a modified file system page is paged out to the backing store.
kernel_asflt	Fires whenever a page fault is taken by the kernel on a page in its own address space. Whenever immediately preceded by a firing of the as_fault probe.

Name of the Probe	Description
maj_fault	Fires whenever a page fault is taken that results in I/O from a backing store or swap device. Whene be immediately preceded by a firing of the pgin probe.
pgfrec	Fires whenever a page is reclaimed off of the free page list.
pgin	Fires whenever a page is paged in from the backing store or from a swap device. This probe differs maj_fault only fires when a page is paged in as a result of a page fault. pgin fires every time a pag the reason.
pgout	Fires whenever a page is paged out to the backing store or to a swap device.
pgpgin	Fires whenever a page is paged in from the backing store or from a swap device. The only differenc is that pgpgin contains the number of pages paged in as arg0. pgin always contains 1 in arg0.
pgpgout	Fires whenever a page is paged out to the backing store or to a swap device. The only difference be that pgpgout contains the number of pages paged out as arg0. pgout always contains 1 in arg0.)
pgrec	Fires whenever a page is reclaimed.
pgrrun	Fires whenever the pager is scheduled.
pgswapin	Fires whenever pages from a swapped-out process are swapped in. The number of pages swapped i
pgswapout	Fires whenever pages are swapped out as part of swapping out a process. The number of pages swa arg0.
prot_fault	Fires whenever a page fault is taken due to a protection violation.
rev	Fires whenever the page daemon begins a new revolution through all pages.
scan	Fires whenever the page daemon examines a page.
softlock	Fires whenever a page is faulted as a part of placing a software lock on the page.
swapin	Fires whenever a swapped-out process is swapped back in.
swapout	Fires whenever a process is swapped out.
zfod	Fires whenever a zero-filled page is created on demand.

Arguments

The following table describes the vminfo provider arguments.

TABLE 58   vminfo Provider Arguments

Argument	Description
arg0	The value by which the statistic is to be incremented. For most probes, this argument is alwa 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 quanti that will be incremented by the value in arg0. Dereferencing this pointer allows consumers t determine the current count of the statistic corresponding to the probe.

Example

Examine the following output from the vmstat command:

```
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 enables you to learn more about the source of these page-ins, as shown in the following example:

```
# 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
```

The output shows that a process associated with the StarOffice™ software, soffice.bin, is responsible for most of the page-ins. To get a better picture of soffice.bin in terms of virtual memory behavior, you could enable all vminfo probes. The following example runs dtrace while launching the StarOffice software:

```
# 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
```

The following example script provides more information about the virtual memory behavior of the StarOffice software during its startup:

```
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
         * the 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 the initial timestamp.  (There are 1,000,000,000 nanoseconds
         * in a second.)  Assume that the script will be terminated before
         * 60 seconds elapses.
         */
        @[probename] =
            lquantize((timestamp - start) / 1000000000, 0, 60);
}
```

Run the script while again starting the StarOffice software. Then, create a new drawing, create a new presentation, and then close all files and quit the application. Press Control-C in the shell running the D script. The results provide a view of some virtual memory 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
```

The output shows some StarOffice behavior with respect to the virtual memory system. For example, the maj_fault probe did not fire until a new instance of the application was started. As you would hope, a "warm start" of StarOffice did not result in new major faults. The as_fault output shows an initial burst of activity, 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. The zfod output shows 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 direction you want to explore. If you want to understand the source of the demand for zero-filled pages, you could aggregate on ustack in a zfod enabling. You might want to establish a threshold for zero-filled pages and use the stop destructive action to stop the offending process when the threshold is exceeded. This approach would enable you to use more traditional debugging tools like truss or mdb. The vminfo provider enables you to associate statistics seen in the output of conventional tools like vmstat with the applications that are inducing the systemic behavior.

Stability

The vminfo provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 59   Stability Mechanism for the vminfo Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown

Element	Name stability	Data stability	Dependency class
Name	Evolving	Evolving	ISA
Arguments	Private	Private	ISA

`proc` **Provider**

The `proc` provider include probes pertaining to the following activities: process creation and termination, LWP creation and termination, executing new program images, and sending and handling signals.

Probes

The `proc` probes are described in the following table.

TABLE 60   `proc` Probes

Probe	Description
create	Fires when a process is created using `fork`, `forkall`, `fork1`, or `vfork`. `psinfo_t` corresponding to the new child process is pointed to by `args[0]`. You can distinguish `vfork` from the other fork variants by checking for `PR_VFORKP` in the `pr_flag` member of the forking thread's `lwpsinfo_t`. You can distinguish `fork1` from `forkall` by examining the `pr_nlwp` members of both the parent process's `psinfo_t` (`curpsinfo`) and the child process's `psinfo_t` (`args[0]`). Because the create probe only fires after the process has been successfully created, and because LWP creation is part of creating a process, `lwp-create` will fire for any LWPs created at process creation time before the create probe fires for the new process.
exec	Fires whenever a process loads a new process image with a variant of the `exec` system call: `exec`, `execle`, `execlp`, `execv`, `execve`, and `execvp`. The `exec` probe fires before the process image is loaded. Process variables like `execname` and `curpsinfo` therefore contain the process state before the image is loaded. Some time after the `exec` probe fires, either the `exec-failure` probe or the exec-success probe will subsequently fire in the same thread. The path of the new process image is pointed to by `args[0]`.
exec-failure	Fires when an `exec` variant has failed. The `exec-failure` probe fires only after the exec probe has fired in the same thread. The `errno` value is provided in `args[0]`.
exec-success	Fires when an `exec` variant has succeeded. Like the exec-failure probe, the exec-success probe fires only after the exec probe has fired in the same thread. By the time the exec-success probe fires, process variables like `execname` and `curpsinfo` contain the process state after the new process image has been loaded.
exit	Fires when the current process is exiting. The reason for exit, which is expressed as one of the `SIGCHLD` `siginfo.h(3HEAD)` codes, is contained in `args[0]`.
fault	Fires when a thread experiences a system fault. The fault code, as defined in `proc`, is in `args[0]`. The `siginfo` structure corresponding to the fault is pointed to by `args[1]`. Only those faults that induce a signal can trigger the `fault` probe.

Probe	Description
lwp-create	Fires when an LWP is created, typically as a result of thr_create. The lwpsinfo_t corresponding to the new thread is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1].
lwp-exit	Fires when an LWP is exiting, due either to a signal or to an explicit call to thr_exit.
lwp-start	Fires within the context of a newly created LWP. The lwp-start probe will fire before any user-level instructions are executed. If the LWP is the first LWP in the process, the start probe will fire, followed by lwp-start.
signal-clear	Probes that fires when a pending signal is cleared because the target thread was waiting for the signal in sigwait, sigwaitinfo, or sigtimedwait. Under these conditions, the pending signal is cleared and the signal number is returned to the caller. The signal number is in args[0]. signal-clear fires in the context of the formerly waiting thread.
signal-discard	Fires when a signal is sent to a single-threaded process, and the signal is both unblocked and ignored by the process. Under these conditions, the signal is discarded on generation. The lwpsinfo_t and psinfo_t of the target process and thread are in args[0] and args[1], respectively. The signal number is in args[2].
signal-handle	Fires immediately before a thread handles a signal. The signal-handle probe fires in the context of the thread that will handle the signal. The signal number is in args[0]. A pointer to the siginfo_t structure that corresponds to the signal is in args[1]. The address of the signal handler in the process is in args[2].
signal-send	Fires when a signal is sent to a thread or process. The signal-send probe fires in the context of the sending process and thread. The lwpsinfo_t and psinfo_t of the receiving process and thread are in args[0] and args[1], respectively. The signal number is in args[2]. signal-send is always followed by signal-handle or signal-clear in the receiving process and thread.
start	Fires in the context of a newly created process. The start probe will fire before any user-level instructions are executed in the process.

Arguments

The argument types for the proc probes are listed in Table 61, "proc Probe Arguments," on page 255. The arguments are described in Table 60, "proc Probes," on page 254.

TABLE 61   proc Probe Arguments

Probe	args[0]	args[1]	args[2]
create	psinfo_t *	—	—
exec	char *	—	—
exec-failure	int	—	—
exec-success	—	—	—
exit	int	—	—
fault	int	siginfo_t *	—
lwp-create	lwpsinfo_t *	psinfo_t *	—

Probe	args[0]	args[1]	args[2]
lwp-exit	—	—	—
lwp-start	—	—	—
signal-clear	int	—	—
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-handle	int	siginfo_t *	void (*)(void)
signal-send	lwpsinfo_t *	psinfo_t *	int
start	—	—	—

lwpsinfo_t

Several proc probes have arguments of type lwpsinfo_t, a structure that is documented in proc. The definition of the lwpsinfo_t structure as available to DTrace consumers is as follows:

```
typedef struct lwpsinfo {
        int pr_flag;              /* flags; see below */
        id_t pr_lwpid;            /* LWP id */
        uintptr_t pr_addr;        /* internal address of thread */
        uintptr_t pr_wchan;       /* wait addr for sleeping thread */
        char pr_stype;            /* synchronization event type */
        char pr_state;            /* numeric thread state */
        char pr_sname;            /* printable character for pr_state */
        char pr_nice;             /* nice for cpu usage */
        short pr_syscall;         /* system call number (if in syscall) */
        int pr_pri;               /* priority, high value = high priority */
        char pr_clname[PRCLSZ];   /* scheduling class name */
        processorid_t pr_onpro;   /* processor which last ran this thread */
        processorid_t pr_bindpro; /* processor to which thread is bound */
        psetid_t pr_bindpset;     /* processor set to which thread is bound */
} lwpsinfo_t;
```

The pr_flag field is a bit-mask holding flags describing the process. These flags and their meanings are described in .

TABLE 62   pr_flag Values

Flags	Descriptions
PR_ISSYS	The process is a system process.
PR_VFORKP	The process is the parent of a vfork child.
PR_FORK	The process has its inherit-on-fork mode set.
PR_RLC	The process has its run-on-last-close mode set.
PR_KLC	The process has its kill-on-last-close mode set.

Flags	Descriptions
PR_ASYNC	The process has its asynchronous-stop mode set.
PR_MSACCT	The process has microstate accounting enabled.
PR_MSFORK	The process microstate accounting is inherited on fork.
PR_BPTADJ	The process has its breakpoint adjustment mode set.
PR_PTRACE	The process has its ptrace compatibility mode set.
PR_STOPPED	The thread is an LWP that is stopped.
PR_ISTOP	The thread is an LWP stopped on an event of interest.
PR_DSTOP	The thread is an LWP that has a stop directive in effect.
PR_STEP	The thread is an LWP that has a single-step directive in effect.
PR_ASLEEP	The thread is an LWP in an interruptible sleep within a system call.
PR_DETACH	The thread is a detached LWP. See pthread_create(3C) and pthread_join(3C).
PR_DAEMON	The thread is a daemon LWP. See pthread_create(3C).
PR_AGENT	The thread is the agent LWP for the process.
PR_IDLE	The thread is the idle thread for a CPU. Idle threads only run on a CPU when the run queues

The pr_addr field is the address of a private, in-kernel data structure representing the thread. While the data structure is private, the pr_addr field may be used as a token unique to a thread for the thread's lifetime.

The pr_wchan field is set when the thread is sleeping on a synchronization object. The meaning of the pr_wchan field is private to the kernel implementation, but the field may be used as a token unique to the synchronization object.

The pr_stype field is set when the thread is sleeping on a synchronization object. The possible values for the pr_stype field are in Table 63, "pr_stype Values," on page 257.

TABLE 63   pr_stype Values

SOBJ_MUTEX	Kernel mutex synchronization object. Used to serialize access to shared data regions in the kernel. See"Stability" on page 188 and *mutex_init* for details on kernel mutex synchronization objects.
SOBJ_RWLOCK	Kernel readers/writer synchronization object. Used to synchronize access to shared objects in the kernel that can allow multiple concurrent readers or a single writer. See "Stability" on page 188 and *rwlock(9F)* for details on kernel readers/writer synchronization objects.
SOBJ_CV	Condition variable synchronization object. A condition variable is designed to wait indefinitely until some condition becomes true. Condition variables are typically used to synchronize for reasons other than access to a shared data region, and are the mechanism generally used when a process performs a program-directed indefinite wait. For example, blocking in poll, pause, wait, and the like.
SOBJ_SEMA	Semaphore synchronization object. A general-purpose synchronization object that — like condition variable objects — does not track a notion of ownership. Because ownership is required to implement priority inheritance

	in the Oracle Solaris kernel, the lack of ownership inherent in semaphore objects inhibits their widespread use. See *semaphore(9F)* for details.
SOBJ_USER	A user-level synchronization object. All blocking on user-level synchronization objects is handled with SOBJ_USER synchronization objects. User-level synchronization objects include those created with mutex_init (3C), sema_init(3C), rwlock_init(3C), cond_init(3C) and their POSIX equivalents.
SOBJ_USER_PI	A user-level synchronization object that implements priority inheritance. Some user-level synchronization objects that track ownership additionally allow for priority inheritance. For example, mutex objects created with pthread_mutex_init may be made to inherit priority using pthread_mutexattr_setprotocol.
SOBJ_SHUTTLE	A shuttle synchronization object. Shuttle objects are used to implement doors. See door_create(3DOOR) for more information.

The `pr_state` field is set to one of the values in Table 64, "`pr_state` Values," on page 258. The `pr_sname` field is set to a corresponding character shown in parentheses in the same table.

TABLE 64   `pr_state` Values

SSLEEP (S)	The thread is sleeping. The sched:::sleep probe will fire immediately before a thread's state is transitioned to SSLEEP.
SRUN (R)	The thread is runnable, but is not currently running. The sched::: enqueue probe will fire immediately before a thread's state is transitioned to SRUN.
SZOMB (Z)	The thread is a zombie LWP.
SSTOP (T)	The thread is stopped, either due to an explicit proc directive or some other stopping mechanism.
SIDL (I)	The thread is an intermediate state during process creation.
SONPROC (O)	The thread is running on a CPU. The sched:::on-cpu probe will fire in the context of the SONPROC thread a short time after the thread's state is transitioned to SONPROC.
SWAIT (W)	The thread is waiting on wait queue. The sched:::cpucaps-sleep probe will fire immediately before a thread state is transitioned to SWAIT.

`psinfo_t`

Several `proc` probes have an argument of type `psinfo_t`, a structure that is documented in `proc`. The definition of the `psinfo_t` structure as available to DTrace consumers is as follows:

```
typedef struct psinfo {
        int    pr_nlwp;         /* number of active lwps in the process */
        pid_t  pr_pid;          /* unique process id */
        pid_t  pr_ppid;         /* process id of parent */
        pid_t  pr_pgid;         /* pid of process group leader */
        pid_t  pr_sid;          /* session id */
```

```
        uid_t   pr_uid;            /* real user id */
        uid_t   pr_euid;           /* effective user id */
        gid_t   pr_gid;            /* real group id */
        gid_t   pr_egid;           /* effective group id */
        uintptr_t pr_addr;         /* address of process */
        dev_t   pr_ttydev;         /* controlling tty device (or PRNODEV) */
        timestruc_t pr_start;      /* process start time, from the epoch */
        char    pr_fname[PRFNSZ];  /* name of execed file */
        char    pr_psargs[PRARGSZ]; /* initial characters of arg list */
        int     pr_argc;           /* initial argument count */
        uintptr_t pr_argv;         /* address of initial argument vector */
        uintptr_t pr_envp;         /* address of initial environment vector */
        char    pr_dmodel;         /* data model of the process */
        taskid_t pr_taskid;        /* task id */
        projid_t pr_projid;        /* project id */
        poolid_t pr_poolid;        /* pool id */
        zoneid_t pr_zoneid;        /* zone id */
} psinfo_t;
```

The pr_dmodel field is set to either PR_MODEL_ILP32, denoting a 32–bit process, or
PR_MODEL_LP64, denoting a 64–bit process.

Examples

exec

You can use the exec probe to determine which programs are being executed, and by whom, as
shown in the following example:

```
#pragma D option quiet

proc:::exec
{
        self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
        @[self->parent, execname] = count();
        self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
        self->parent = NULL;
```

```
}

END
{
        printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
        printa("%-20s %-20s %@d\n", @);
}
```

Running the example script for a short period of time on a build machine results in output similar to the following example:

```
# dtrace -s ./whoexec.d
^C
WHO                  WHAT                 COUNT
make.bin             yacc                 1
tcsh                 make                 1
make.bin             spec2map             1
sh                   grep                 1
lint                 lint2                1
sh                   lint                 1
sh                   ln                   1
cc                   ld                   1
make.bin             cc                   1
lint                 lint1                1
sh                   lex                  1
make.bin             mv                   2
sh                   sh                   3
sh                   make                 3
sh                   sed                  4
sh                   tr                   4
make                 make.bin             4
sh                   install.bin          5
sh                   rm                   6
cc                   ir2hf                33
cc                   ube                  33
sh                   date                 34
sh                   mcs                  34
cc                   acomp                34
sh                   cc                   34
sh                   basename             34
basename             expr                 34
make.bin             sh                   87
```

start and exit

If you want to know how long programs are running from creation to termination, you can enable the start and exit probes, as shown in the following example:

```
proc:::start
{
```

```
        self->start = timestamp;
}

proc:::exit
/self->start/
{
        @[execname] = quantize(timestamp - self->start);
        self->start = 0;
}
```

Running the example script on the build server for several seconds results in output similar to the following example:

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

  ir2hf
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@                                        1
        16777216 |@@@@@@@@@@@@@@                           14
        33554432 |@@@@@@@@@@                               9
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |@@@@                                     4
       536870912 |@                                        1
      1073741824 |                                         0

  ube
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@@@@@@@                                  6
        67108864 |@@@                                      3
       134217728 |@@                                       2
       268435456 |@@@@                                     4
       536870912 |@@@@@@@@@@@@                             10
      1073741824 |@@@@@@@                                  6
      2147483648 |@@                                       2
      4294967296 |                                         0

  acomp
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@                                       2
        33554432 |                                         0
        67108864 |@                                        1
       134217728 |@@@                                      3
       268435456 |                                         0
       536870912 |@@@@@                                    5
      1073741824 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@              22
      2147483648 |@                                        1
```

```
     4294967296 |                                                  0

  cc
          value  ------------- Distribution ------------- count
       33554432 |                                                  0
       67108864 |@@@                                               3
      134217728 |@                                                 1
      268435456 |                                                  0
      536870912 |@@@@                                              4
     1073741824 |@@@@@@@@@@@@@                                     13
     2147483648 |@@@@@@@@@@@                                       11
     4294967296 |@@@                                               3
     8589934592 |                                                  0

  sh
          value  ------------- Distribution ------------- count
         262144 |                                                  0
         524288 |@                                                 5
        1048576 |@@@@@@                                            29
        2097152 |                                                  0
        4194304 |                                                  0
        8388608 |@@@                                               12
       16777216 |@@                                                9
       33554432 |@@                                                9
       67108864 |@@                                                8
      134217728 |@                                                 7
      268435456 |@@@@@                                             20
      536870912 |@@@@@@                                            26
     1073741824 |@@@                                               14
     2147483648 |@@                                                11
     4294967296 |                                                  3
     8589934592 |                                                  1
    17179869184 |                                                  0

  make.bin
          value  ------------- Distribution ------------- count
       16777216 |                                                  0
       33554432 |@                                                 1
       67108864 |@                                                 1
      134217728 |@@                                                2
      268435456 |                                                  0
      536870912 |@@                                                2
     1073741824 |@@@@@@@@@                                         9
     2147483648 |@@@@@@@@@@@@@@                                    14
     4294967296 |@@@@@@                                            6
     8589934592 |@@                                                2
    17179869184 |                                                  0
```

`lwp-start` and `lwp-exit`

Instead of knowing the amount of time that a particular process takes to run, you might want to know how long individual threads take to run. The following example shows how to use the `lwp-start` and `lwp-exit` probes for this purpose:

```
proc:::lwp-start
/tid != 1/
{
        self->start = timestamp;
}

proc:::lwp-exit
/self->start/
{
        @[execname] = quantize(timestamp - self->start);
        self->start = 0;
}
```

Running the example script on an NFS and calendar server results in output similar to the following example:

```
# dtrace -s ./lwptime.d
dtrace: script './lwptime.d' matched 3 probes
^C

  nscd
         value  ------------- Distribution ------------- count
        131072 |                                         0
        262144 |@                                        18
        524288 |@@                                       24
       1048576 |@@@@@@@                                  75
       2097152 |@@@@@@@@@@@@@@@@@@@@@@@                   245
       4194304 |@@                                       22
       8388608 |@@                                       24
      16777216 |                                         6
      33554432 |                                         3
      67108864 |                                         1
     134217728 |                                         1
     268435456 |                                         0

  mountd
         value  ------------- Distribution ------------- count
        524288 |                                         0
       1048576 |@                                        15
       2097152 |@                                        24
       4194304 |@@@                                      51
       8388608 |@                                        17
      16777216 |@                                        24
      33554432 |@                                        15
      67108864 |@@@@                                     57
```

```
     134217728 |@                                         28
     268435456 |@                                         26
     536870912 |@@                                        39
    1073741824 |@@@                                       45
    2147483648 |@@@@                                      72
    4294967296 |@@@@                                      77
    8589934592 |@@@                                       55
   17179869184 |                                          14
   34359738368 |                                          2
   68719476736 |                                          0

  automountd
         value  ------------- Distribution ------------- count
       1048576 |                                          0
       2097152 |                                          3
       4194304 |@@@@                                      146
       8388608 |                                          6
      16777216 |                                          6
      33554432 |                                          9
      67108864 |@@@@@                                     203
     134217728 |@@                                        87
     268435456 |@@@@@@@@@@@@@@                            534
     536870912 |@@@@@@                                    223
    1073741824 |@                                         45
    2147483648 |                                          20
    4294967296 |                                          26
    8589934592 |                                          20
   17179869184 |                                          19
   34359738368 |                                          7
   68719476736 |                                          2
  137438953472 |                                          0

  iCald
         value  ------------- Distribution ------------- count
       8388608 |                                          0
      16777216 |@@@@@@@                                   20
      33554432 |@@@                                       9
      67108864 |@@                                        8
     134217728 |@@@@@                                      16
     268435456 |@@@@                                       11
     536870912 |@@@@                                       11
    1073741824 |@                                          4
    2147483648 |                                           2
    4294967296 |                                           0
    8589934592 |@@                                         8
   17179869184 |@                                          5
   34359738368 |@                                          4
   68719476736 |@@                                         6
  137438953472 |@                                          4
  274877906944 |                                           2
  549755813888 |                                           0
```

signal-send

You can use the signal-send probe to determine the sending and receiving process associated
with any signal, as shown in the following example:

```
#pragma D option quiet

proc:::signal-send
{
        @[execname, stringof(args[1]->pr_fname), args[2]] = count();
}

END
{
        printf("%20s %20s %12s %s\n",
            "SENDER", "RECIPIENT", "SIG", "COUNT");
        printa("%20s %20s %12d %@d\n", @);
}
```

Running this script results in output similar to the following example:

```
# dtrace -s ./sig.d
^C
            SENDER            RECIPIENT         SIG COUNT
             xterm               dtrace           2 1
             xterm           soffice.bin           2 1
                tr                 init          18 1
             sched                 test          18 1
             sched                fvwm2          18 1
              bash                 bash          20 1
               sed                 init          18 2
             sched                  ksh          18 15
             sched                 Xsun          22 471
```

Stability

The proc provider uses DTrace's stability mechanism to describe its stabilities, as shown in
the following table. For more information about the stability mechanism, see Chapter 18,
"Stability".

TABLE 65   Stability Mechanism for the proc Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA

Element	Name stability	Data stability	Dependency class
Arguments	Evolving	Evolving	ISA

sched **Provider**

The sched provider makes available probes related to CPU scheduling. Because CPUs are the one resource that all threads must consume, the sched provider is useful for understanding systemic behavior. For example, you can use the sched provider to understand when and why threads sleep, run, change priority, or wake other threads.

Probes

The sched probes are described in the following table.

TABLE 66   sched Probes

Probe	Description
change-pri	Fires whenever a thread's priority is about to be changed. The lwpsinfo_t of the thread is pointed to by args[0]. The thread's current priority is in the pr_pri field of this structure. The psinfo_t of the process containing the thread is pointed to by args[1]. The thread's new priority is contained in args[2].
cpucaps-sleep	Fires immediately before the current thread is placed on a wait queue. The lwpsinfo_t of the waiting thread is pointed to by args[0]. The psinfo_t of the process containing the waiting thread is pointed to by args[1].
cpucaps-wakeup	Fires immediately after a thread is removed from a wait queue. The lwpsinfo_t of the waiting thread is pointed to by args[0]. The psinfo_t of the process containing the waiting thread is pointed to by args[1].
dequeue	Fires immediately before a runnable thread is dequeued from a run queue. The lwpsinfo_t of the thread being dequeued is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1]. The cpuinfo_t of the CPU from which the thread is being dequeued is pointed to by args[2]. If the thread is being dequeued from a run queue that is not associated with a particular CPU, the cpu_id member of this structure will be -1.
enqueue	Fires immediately before a runnable thread is enqueued to a run queue. The lwpsinfo_t of the thread being enqueued is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1]. The cpuinfo_t of the CPU to which the thread is being enqueued is pointed to by args[2]. If the thread is being enqueued from a run queue that is not associated with a particular CPU, the cpu_id member of this structure will be -1. The value in args[3] is a boolean indicating whether the thread will be enqueued to the front of the run queue. The value is non-zero if the thread

Probe	Description
	will be enqueued at the front of the run queue, and zero if the thread will be enqueued at the back of the run queue.
off-cpu	Fires when the current CPU is about to end execution of a thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is ending execution. The curpsinfo variable describes the process containing the current thread. The lwpsinfo_t structure of the thread that the current CPU will next execute is pointed to by args[0]. The psinfo_t of the process containing the next thread is pointed to by args[1].
on-cpu	Fires when a CPU has just begun execution of a thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is beginning execution. The curpsinfo variable describes the process containing the current thread.
preempt	Fires immediately before the current thread is preempted. After this probe fires, the current thread will select a thread to run and the off-cpu probe will fire for the current thread. In some cases, a thread on one CPU will be preempted, but the preempting thread will run on another CPU in the meantime. In this situation, the preempt probe will fire, but the dispatcher will be unable to find a higher priority thread to run and the remain-cpu probe will fire instead of the off-cpu probe.
remain-cpu	Fires when a scheduling decision has been made, but the dispatcher has elected to continue to run the current thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is beginning execution. The curpsinfo variable describes the process containing the current thread.
schedctl-nopreempt	Fires when a thread is preempted and then re-enqueued at the front of the run queue due to a preemption control request. See *schedctl_init(3C)* for details on preemption control. As with preempt, either off-cpu or remain-cpu will fire after schedctl-nopreempt. Because schedctl-nopreempt denotes a re-enqueuing of the current thread at the front of the run queue, remain-cpu is more likely to fire after schedctl-nopreempt than off-cpu. The lwpsinfo_t of the thread being preempted is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1].
schedctl-preempt	Fires when a thread that is using preemption control is nonetheless preempted and re-enqueued at the back of the run queue. See *schedctl_init(3C)* for details on preemption control. As with preempt, either off-cpu or remain-cpu will fire after schedctl-preempt. Like preempt (and unlike schedctl-nopreempt), schedctl-preempt denotes a re-enqueuing of the current thread at the back of the run queue. As a result, off-cpu is more likely to fire after schedctl-preempt than remain-cpu. The lwpsinfo_t of the thread being preempted is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1].
schedctl-yield	Fires when a thread that had preemption control enabled and its time slice artificially extended executed code to yield the CPU to other threads.
sleep	Fires immediately before the current thread sleeps on a synchronization object. The type of the synchronization object is contained in the pr_stype member of the lwpsinfo_t pointed to by curlwpsinfo. The address of the synchronization object is contained in the pr_wchan member of the lwpsinfo_t pointed to by curlwpsinfo. The meaning of this address is a private implementation detail, but the address value may be treated as a token unique to the synchronization object.

Probe	Description
surrender	Fires when a CPU has been instructed by another CPU to make a scheduling decision — often because a higher-priority thread has become runnable.
tick	Fires as a part of clock tick-based accounting. In clock tick-based accounting, CPU accounting is performed by examining which threads and processes are running when a fixed-interval interrupt fires. The `lwpsinfo_t` that corresponds to the thread that is being assigned CPU time is pointed to by `args[0]`. The `psinfo_t` that corresponds to the process that contains the thread is pointed to by `args[1]`.
wakeup	Fires immediately before the current thread wakes a thread sleeping on a synchronization object. The `lwpsinfo_t` of the sleeping thread is pointed to by `args[0]`. The `psinfo_t` of the process containing the sleeping thread is pointed to by `args[1]`. The type of the synchronization object is contained in the `pr_stype` member of the `lwpsinfo_t` of the sleeping thread. The address of the synchronization object is contained in the `pr_wchan` member of the `lwpsinfo_t` of the sleeping thread. The meaning of this address is a private implementation detail, but the address value may be treated as a token unique to the synchronization object.

Arguments

The argument types for the `sched` probes are listed in the following table; the arguments are described in .

TABLE 67   sched Probe Arguments

Probe	args[0]	args[1]	args[2]	args[3]
change-pri	`lwpsinfo_t *`	`psinfo_t *`	`pri_t`	—
cpucaps-sleep	`lwpsinfo_t *`	`psinfo_t *`	—	—
cpucaps-wakeup-	`lwpsinfo_t *`	`psinfo_t *`	—	—
dequeue	`lwpsinfo_t *`	`psinfo_t *`	`cpuinfo_t *`	—
enqueue	`lwpsinfo_t *`	`psinfo_t *`	`cpuinfo_t *`	`int`
off-cpu	`lwpsinfo_t *`	`psinfo_t *`	—	—
on-cpu	—	—	—	—
preempt	—	—	—	—
remain-cpu	—	—	—	—
schedctl-nopreempt	`lwpsinfo_t *`	`psinfo_t *`	—	—
schedctl-preempt	`lwpsinfo_t *`	`psinfo_t *`	—	—
schedctl-yield	`lwpsinfo_t *`	`psinfo_t *`	—	—
sleep	—	—	—	—
surrender	`lwpsinfo_t *`	`psinfo_t *`	—	—
tick	`lwpsinfo_t *`	`psinfo_t *`	—	—

Probe	args[0]	args[1]	args[2]	args[3]
wakeup	lwpsinfo_t *	psinfo_t *	—	—

Many sched probes have arguments consisting of a pointer to an lwpsinfo_t and a pointer to a psinfo_t, indicating a thread and the process containing the thread, respectively. These structures are described in detail in lwpsinfo_t and psinfo_t, respectively. For more information, see Table 67, "sched Probe Arguments," on page 268.

cpuinfo_t

The cpuinfo_t structure defines a CPU. Arguments to both the enqueue and dequeue probes include a pointer to a cpuinfo_t. Additionally, the cpuinfo_t corresponding to the current CPU is pointed to by the curcpu variable. The definition of the cpuinfo_t structure is as follows:

```
typedef struct cpuinfo {
        processorid_t cpu_id;          /* CPU identifier */
        psetid_t cpu_pset;             /* processor set identifier */
        chipid_t cpu_chip;             /* chip identifier */
        lgrp_id_t cpu_lgrp;            /* locality group identifer */
        processor_info_t cpu_info;     /* CPU information */
} cpuinfo_t;
```

The cpu_id member is the processor identifier, as returned by psrinfo and p_online. For more information, see psrinfo(1M) and p_online(2) man pages.

The cpu_pset member is the processor set that contains the CPU, if any. See psrset(1M) for more details on processor sets.

The cpu_chip member is the identifier of the physical chip. Physical chips may contain several CPUs. For more information, see the psrinfo(1M) man page.

The cpu_lgrp member is the identifier of the latency group associated with the CPU. See *liblgrp(3LIB)* for details on latency groups.

The cpu_info member is the processor_info_t structure associated with the CPU, as returned by processor_info. For more information, see the processor_info(2) man page.

Examples

This seciton describes sched provider examples.

`on-cpu` and `off-cpu`

One common question you might want answered is which CPUs are running threads and for how long. You can use the on‑cpu and off‑cpu probes to easily answer this question on a system-wide basis as shown in the following example:

```
sched:::on-cpu
{
        self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
        @[cpu] = quantize(timestamp - self->ts);
        self->ts = 0;
}
```

Running the above script results in output similar to the following example:

```
# dtrace -s ./where.d
dtrace: script './where.d' matched 5 probes
^C

        0
          value  ------------- Distribution ------------- count
           2048 |                                         0
           4096 |@@                                       37
           8192 |@@@@@@@@@@@@                             212
          16384 |@                                        30
          32768 |                                         10
          65536 |@                                        17
         131072 |                                         12
         262144 |                                         9
         524288 |                                         6
        1048576 |                                         5
        2097152 |                                         1
        4194304 |                                         3
        8388608 |@@@@                                     75
       16777216 |@@@@@@@@@@@@                             201
       33554432 |                                         6
       67108864 |                                         0

        1
          value  ------------- Distribution ------------- count
           2048 |                                         0
           4096 |@                                        6
           8192 |@@@@                                     23
          16384 |@@@                                      18
          32768 |@@@@                                     22
          65536 |@@@@                                     22
```

```
     131072 |@                                              7
     262144 |                                               5
     524288 |                                               2
    1048576 |                                               3
    2097152 |@                                              9
    4194304 |                                               4
    8388608 |@@@                                            18
   16777216 |@@@                                            19
   33554432 |@@@                                            16
   67108864 |@@@@                                           21
  134217728 |@@                                             14
  268435456 |                                               0
```

The above output shows that on CPU 1 threads tend to run for less than 100 microseconds at a stretch, or for approximately 10 milliseconds. A noticable gap between the two clusters of data is shown in the histogram. You also might be interested in knowing which CPUs are running a particular process. You can use the on-cpu and off-cpu probes for answering this question as well. The following script displays which CPUs run a specified application over a period of ten seconds:

```
#pragma D option quiet

dtrace:::BEGIN
{
        start = timestamp;
}

sched:::on-cpu
/execname == $$1/
{
        self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
        @[cpu] = sum(timestamp - self->ts);
        self->ts = 0;
}

profile:::tick-1sec
/++x == 10/
{
        exit(0);
}

dtrace:::END
{
        printf("CPU distribution over %d seconds:\n\n",
            (timestamp - start) / 1000000000);
        printf("CPU microseconds\n--- ------------\n");
```

```
            normalize(@, 1000);
            printa("%3d %@d\n", @);
}
```

Running the above script on a large mail server and specifying the IMAP daemon results in output similar to the following example:

```
# dtrace -s ./whererun.d imapd
CPU distribution of imapd over 10 seconds:

CPU microseconds
--- ------------
 15 10102
 12 16377
 21 25317
 19 25504
 17 35653
 13 41539
 14 46669
 20 57753
 22 70088
 16 115860
 23 127775
 18 160517
```

Oracle Solaris takes into account the amount of time that a thread has been sleeping when selecting a CPU on which to run the thread: a thread that has been sleeping for less time tends not to migrate. You can use the off-cpu and on-cpu probes to observe this behavior:

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
        self->cpu = cpu;
        self->ts = timestamp;
}

sched:::on-cpu
/self->ts/
{
        @[self->cpu == cpu ?
            "sleep time, no CPU migration" : "sleep time, CPU migration"] =
            lquantize((timestamp - self->ts) / 1000000, 0, 500, 25);
        self->ts = 0;
        self->cpu = 0;
}
```

Running the above script for approximately 30 seconds results in output similar to the following example:

```
# dtrace -s ./howlong.d
dtrace: script './howlong.d' matched 5 probes
^C
```

```
sleep time, CPU migration
        value  -------------- Distribution ------------ count
          < 0 |                                          0
            0 |@@@@@@@                                    6838
           25 |@@@@@                                     4714
           50 |@@@                                       3108
           75 |@                                         1304
          100 |@                                         1557
          125 |@                                         1425
          150 |                                          894
          175 |@                                         1526
          200 |@@                                        2010
          225 |@@                                        1933
          250 |@@                                        1982
          275 |@@                                        2051
          300 |@@                                        2021
          325 |@                                         1708
          350 |@                                         1113
          375 |                                          502
          400 |                                          220
          425 |                                          106
          450 |                                          54
          475 |                                          40
        >= 500 |@                                         1716

sleep time, no CPU migration
        value  -------------- Distribution ------------ count
          < 0 |                                          0
            0 |@@@@@@@@@@@@                               58413
           25 |@@@                                       14793
           50 |@@                                        10050
           75 |                                          3858
          100 |@                                         6242
          125 |@                                         6555
          150 |                                          3980
          175 |@                                         5987
          200 |@                                         9024
          225 |@                                         9070
          250 |@@                                        10745
          275 |@@                                        11898
          300 |@@                                        11704
          325 |@@                                        10846
          350 |@                                         6962
          375 |                                          3292
          400 |                                          1713
          425 |                                          585
          450 |                                          201
          475 |                                          96
        >= 500 |                                          3946
```

The example output shows that there are many more occurrences of non-migration than migration. Also, when sleep times are longer, migrations are more likely. The distributions are noticeably different in the sub 100 millisecond range, but look very similar as the sleep times get longer. This result would seem to indicate that sleep time is not factored into the scheduling decision once a certain threshold is exceeded.

The final example using off-cpu and on-cpu shows how to use these probes along with the pr_stype field to determine why threads sleep and for how long:

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
        /*
         * In sleeping state.  Track the sobj type.
         */
        self->sobj = curlwpsinfo->pr_stype;
        self->bedtime = timestamp;
}

sched:::off-cpu
/curlwpsinfo->pr_state == SRUN/
{
        self->bedtime = timestamp;
}

sched:::on-cpu
/self->bedtime && !self->sobj/
{
        @["preempted"] = quantize(timestamp - self->bedtime);
        self->bedtime = 0;
}

sched:::on-cpu
/self->sobj/
{
        @[self->sobj == SOBJ_MUTEX ? "kernel-level lock" :
            self->sobj == SOBJ_RWLOCK ? "rwlock" :
            self->sobj == SOBJ_CV ? "condition variable" :
            self->sobj == SOBJ_SEMA ? "semaphore" :
            self->sobj == SOBJ_USER ? "user-level lock" :
            self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
            self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
            quantize(timestamp - self->bedtime);
        self->sobj = 0;
        self->bedtime = 0;
}
```

Running the above script for several seconds results in output similar to the following example:

```
# dtrace -s ./whatfor.d
dtrace: script './whatfor.d' matched 12 probes
```

```
^C
 kernel-level lock
           value  ------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@                                 3
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            11
          131072 |@@                                       1
          262144 |                                         0

  preempted
           value  ------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         4
           65536 |@@@@@@@@                                 408
          131072 |@@@@@@@@@@@@@@@@@@@@@@@                   1031
          262144 |@@@                                      156
          524288 |@@                                       116
         1048576 |@                                        51
         2097152 |                                         42
         4194304 |                                         16
         8388608 |                                         15
        16777216 |                                         4
        33554432 |                                         8
        67108864 |                                         0

  semaphore
           value  ------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@                                       61
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@                553
          262144 |@@                                       63
          524288 |@                                        36
         1048576 |                                         7
         2097152 |                                         22
         4194304 |@                                        44
         8388608 |@@@                                      84
        16777216 |@                                        36
        33554432 |                                         3
        67108864 |                                         6
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         0
      1073741824 |                                         0
      2147483648 |                                         0
      4294967296 |                                         0
      8589934592 |                                         0
     17179869184 |                                         1
     34359738368 |                                         0

  shuttle
           value  ------------- Distribution ------------ count
```

```
              32768 |                                                0
              65536 |@@@@@                                           2
             131072 |@@@@@@@@@@@@@@                                  6
             262144 |@@@@@                                           2
             524288 |                                                0
            1048576 |                                                0
            2097152 |                                                0
            4194304 |@@@@@                                           2
            8388608 |                                                0
           16777216 |                                                0
           33554432 |                                                0
           67108864 |                                                0
          134217728 |                                                0
          268435456 |                                                0
          536870912 |                                                0
         1073741824 |                                                0
         2147483648 |                                                0
         4294967296 |@@@@@                                           2
         8589934592 |                                                0
        17179869184 |@@                                              1
        34359738368 |                                                0

    condition variable
              value ------------- Distribution ------------ count
              32768 |                                                0
              65536 |                                                122
             131072 |@@@@@                                           1579
             262144 |@                                               340
             524288 |                                                268
            1048576 |@@@                                             1028
            2097152 |@@@                                             1007
            4194304 |@@@                                             1176
            8388608 |@@@@                                            1257
           16777216 |@@@@@@@@@@@@@                                   4385
           33554432 |                                                295
           67108864 |                                                157
          134217728 |                                                96
          268435456 |                                                48
          536870912 |                                                144
         1073741824 |                                                10
         2147483648 |                                                22
         4294967296 |                                                18
         8589934592 |                                                5
        17179869184 |                                                6
        34359738368 |                                                4
        68719476736 |                                                0
```

enqueue and dequeue

When a CPU becomes idle, the dispatcher looks for work enqueued on other non-idle CPUs. The following example uses the dequeue probe to understand how often applications are transferred and by which CPU:

```
#pragma D option quiet

sched:::dequeue
/args[2]->cpu_id != -1 && cpu != args[2]->cpu_id &&
    (curlwpsinfo->pr_flag & PR_IDLE)/
{
        @[stringof(args[1]->pr_fname), args[2]->cpu_id] =
            lquantize(cpu, 0, 100);
}

END
{
        printa("%s stolen from CPU %d by:\n%@d\n", @);
}
```

The tail of the output from running the above script on a four CPU system results in output similar to the following example:

```
# dtrace -s ./whosteal.d
^C
...
 nscd stolen from CPU 1 by:

          value  -------------- Distribution ------------ count
              1 |                                         0
              2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 28
              3 |                                         0

snmpd stolen from CPU 1 by:

          value  -------------- Distribution ------------ count
            < 0 |                                         0
              0 |@                                        1
              1 |                                         0
              2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   31
              3 |@@                                       2
              4 |                                         0

sched stolen from CPU 1 by:

          value  -------------- Distribution ------------ count
            < 0 |                                         0
              0 |@@                                       3
              1 |                                         0
              2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     36
```

```
          3 |@@@@                                          5
          4 |                                              0
```

Instead of knowing which CPUs took which work, you might want to know the CPUs on which processes and threads are waiting to run. You can use the enqueue and dequeue probes together to answer this question:

```
sched:::enqueue
{
     a[args[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id] =
         timestamp;
}

sched:::dequeue
/a[args[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id]/
{
     @[args[2]->cpu_id] = quantize(timestamp -
         a[args[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id]);
     a[args[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id] = 0;
}
```

Running the above script for several seconds results in output similar to the following example:

```
# dtrace -s ./qtime.d
dtrace: script './qtime.d' matched 5 probes
^C

        1
          value  ------------- Distribution ------------- count
           1024 |                                         0
           2048 |                                         10
           4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            4316
           8192 |@@@@@@@@                                 1115
          16384 |@@@                                      549
          32768 |@@                                       337
          65536 |@@                                       330
         131072 |                                         13
         262144 |                                         6
         524288 |                                         4
        1048576 |                                         2
        2097152 |                                         1
        4194304 |                                         0

        0
          value  ------------- Distribution ------------- count
           1024 |                                         0
           2048 |                                         22
           4096 |@@@@@@@@@@@@@@@@@                        2747
           8192 |@@@@@@@                                  1205
          16384 |@@@@@@@@@@@@                             1942
          32768 |@@@                                      469
```

```
      65536 |@@                                              304
     131072 |                                               28
     262144 |                                               16
     524288 |                                               5
    1048576 |                                               1
    2097152 |                                               2
    4194304 |                                               1
    8388608 |                                               0

#
```

Instead of looking at wait times, you might want to examine the length of the run queue over time. Using the enqueue and dequeue probes, you can set up an associative array to track the queue length:

```
sched:::enqueue
{
        this->len = qlen[args[2]->cpu_id]++;
        @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
        qlen[args[2]->cpu_id]--;
}
```

Running the above script for approximately 30 seconds on a largely idle uniprocessor system results in output similar to the following example:

```
# dtrace -s ./qlen.d
dtrace: script './qlen.d' matched 5 probes
^C
        0
          value  -------------- Distribution ----------- count
            < 0 |                                         0
              0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              110626
              1 |@@@@@@@@@                                41142
              2 |@@                                       12655
              3 |@                                        5074
              4 |                                         1722
              5 |                                         701
              6 |                                         302
              7 |                                         63
              8 |                                         23
              9 |                                         12
             10 |                                         24
             11 |                                         58
             12 |                                         14
             13 |                                         3
             14 |                                         0
```

The output is roughly what you would expect for an idle system: the majority of the time that a runnable thread is enqueued, the run queue was very short, three or fewer threads in length. However, given that the system was largely idle, the exceptional data points at the bottom of the table might be unexpected. For example, why was the run queue as long as 13 runnable threads? To explore this question, you could write a D script that displays the contents of the run queue when the length of the run queue is long. This problem is complicated because D enablings cannot iterate over data structures, and therefore cannot simply iterate over the entire run queue. Even if D enablings could do so, you should avoid dependencies on the kernel's internal data structures.

For this type of script, you would enable the `enqueue` and `dequeue` probes and use both speculations and associative arrays. Whenever a thread is enqueued, the script increments the length of the queue and records the timestamp in an associative array keyed by the thread. You cannot use a thread-local variable in this case because a thread might be enqueued by another thread. The script then checks to see if the queue length exceeds the maximum. If it does, the script starts a new speculation, and records the timestamp and the new maximum. Then, when a thread is dequeued, the script compares the enqueue timestamp to the timestamp of the longest length: if the thread was enqueued before the timestamp of the longest length, the thread was in the queue when the longest length was recorded. In this case, the script speculatively traces the thread's information. Once the kernel dequeues the last thread that was enqueued at the timestamp of the longest length, the script commits the speculation data. This script is shown below:

```
#pragma D option quiet
#pragma D option nspec=4
#pragma D option specsize=100k

int maxlen;
int spec[int];

sched:::enqueue
{
        this->len = ++qlen[this->cpu = args[2]->cpu_id];
        in[args[0]->pr_addr] = timestamp;
}

sched:::enqueue
/this->len > maxlen && spec[this->cpu]/
{
        /*
         * There is already a speculation for this CPU.  As a new record
         * is set, discard the old one.
         */
        discard(spec[this->cpu]);
}

sched:::enqueue
/this->len > maxlen/
{
```

```
        /*
         * You have a winner.  Set the new maximum length and set the timestamp
         * of the longest length.
         */
        maxlen = this->len;
        longtime[this->cpu] = timestamp;
        /*
         * Now start a new speculation, and speculatively trace the length.
         */
        this->spec = spec[this->cpu] = speculation();
        speculate(this->spec);
        printf("Run queue of length %d:\n", this->len);
}

sched:::dequeue
/(this->in = in[args[0]->pr_addr]) &&
    this->in <= longtime[this->cpu = args[2]->cpu_id]/
{
        speculate(spec[this->cpu]);
        printf("  %d/%d (%s)\n",
            args[1]->pr_pid, args[0]->pr_lwpid,
            stringof(args[1]->pr_fname));
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
        in[args[0]->pr_addr] = 0;
        this->len = --qlen[args[2]->cpu_id];
}

sched:::dequeue
/this->len == 0 && spec[this->cpu]/
{
        /*
         * As you just processed the last thread that was enqueued at the time
         * of longest length; commit the speculation, which by now contains
         * each thread that was enqueued when the queue was longest.
         */
        commit(spec[this->cpu]);
        spec[this->cpu] = 0;
}
```

Running the above script on the same uniprocessor results in output similar to the following example:

```
# dtrace -s ./whoqueue.d
Run queue of length 3:
 0/0 (sched)
  0/0 (sched)
  101170/1 (dtrace)
Run queue of length 4:
```

```
     0/0 (sched)
     100356/1 (Xsun)
     100420/1 (xterm)
     101170/1 (dtrace)
Run queue of length 5:
     0/0 (sched)
     0/0 (sched)
     100356/1 (Xsun)
     100420/1 (xterm)
     101170/1 (dtrace)
Run queue of length 7:
     0/0 (sched)
     100221/18 (nscd)
     100221/17 (nscd)
     100221/16 (nscd)
     100221/13 (nscd)
     100221/14 (nscd)
     100221/15 (nscd)
Run queue of length 16:
     100821/1 (xterm)
     100768/1 (xterm)
     100365/1 (fvwm2)
     101118/1 (xterm)
     100577/1 (xterm)
     101170/1 (dtrace)
     101020/1 (xterm)
     101089/1 (xterm)
     100795/1 (xterm)
     100741/1 (xterm)
     100710/1 (xterm)
     101048/1 (xterm)
     100697/1 (MozillaFirebird-)
     100420/1 (xterm)
     100394/1 (xterm)
     100368/1 (xterm)
^C
```

The output reveals that the long run queues are due to many runnable xterm processes. This experiment coincided with a change in virtual desktop, and therefore the results are probably due to some sort of X event processing.

sleep and wakeup

In enqueue and dequeue, the final example demonstrated that a burst in run queue length was due to runnable xterm processes. One hypothesis is that the observations resulted from a change in virtual desktop. You can use the wakeup probe to explore this hypothesis by determining who is waking the xterm processes, and when, as shown in the following example:

```
#pragma D option quiet
```

```
dtrace:::BEGIN
{
        start = timestamp;
}

sched:::wakeup
/stringof(args[1]->pr_fname) == "xterm"/
{
        @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);
}

profile:::tick-1sec
/++x == 10/
{
        exit(0);
}
```

To investigate the hypothesis, run the above script, waiting roughly five seconds, and switch your virtual desktop exactly once. If the burst of runnable xterm processes is due to switching the virtual desktop, the output should show a burst of wakeup activity at the five second mark.

```
# dtrace -s ./xterm.d

  Xsun

          value  -------------- Distribution ------------ count
              4 |                                         0
              5 |@                                        1
              6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 32
              7 |                                         0
```

The output does show that the X server is waking xterm processes, clustered around the time that you switched virtual desktops. If you wanted to understand the interaction between the X server and the xterm processes, you could aggregate on user stack traces when the X server fires the wakeup probe.

Understanding the performance of client/server systems like the X windowing system requires understanding the clients on whose behalf the server is doing work. This kind of question is difficult to answer with conventional performance analysis tools. However, if you have a model where a client sends a message to the server and sleeps pending the server's processing, you can use the wakeup probe to determine the client for whom the request is being performed, as shown in the following example:

```
self int last;

sched:::wakeup
/self->last && args[0]->pr_stype == SOBJ_CV/
{
        @[stringof(args[1]->pr_fname)] = sum(vtimestamp - self->last);
```

```
        self->last = 0;
}

sched:::wakeup
/execname == "Xsun" && self->last == 0/
{
        self->last = vtimestamp;
}
```

Running the above script results in output similar to the following example:

```
dtrace -s ./xwork.d
dtrace: script './xwork.d' matched 14 probes
^C
  xterm                                            9522510
  soffice.bin                                      9912594
  fvwm2                                          100423123
  MozillaFirebird                               312227077
  acroread                                      345901577
```

This output reveals that much Xsun work is being done on behalf of the processes acroread, MozillaFirebird and, to a lesser degree, fvwm2. Notice that the script only examined wakeups from condition variable synchronization objects SOBJ_CV. As described in Table 63, "pr_stype Values," on page 257, condition variables are the type of synchronization object typically used to synchronize for reasons other than access to a shared data region. In the case of the X server, a client will wait for data in a pipe by sleeping on a condition variable.

You can additionally use the sleep probe along with the wakeup probe to understand which applications are blocking on which applications, and for how long, as shown in the following example:

```
#pragma D option quiet

sched:::sleep
/!(curlwpsinfo->pr_flag & PR_ISSYS) && curlwpsinfo->pr_stype == SOBJ_CV/
{
        bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
        @[stringof(args[1]->pr_fname), execname] =
            quantize(timestamp - bedtime[args[0]->pr_addr]);
        bedtime[args[0]->pr_addr] = 0;
}

END
{
        printa("%s sleeping on %s:\n%@d\n", @);
```

```
}
```

The tail of the output from running the example script for several seconds on a system resembles the following example:

```
# dtrace -s ./whofor.d
^C
...
 xterm sleeping on Xsun:

           value  -------------- Distribution ----------- count
          131072 |                                         0
          262144 |                                         12
          524288 |                                         2
         1048576 |                                         0
         2097152 |                                         5
         4194304 |@@@                                      45
         8388608 |                                         1
        16777216 |                                         9
        33554432 |@@@@@                                    83
        67108864 |@@@@@@@@@@                               164
       134217728 |@@@@@@@@@@                               147
       268435456 |@@@@                                     56
       536870912 |@                                        17
      1073741824 |                                         9
      2147483648 |                                         1
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0

fvwm2 sleeping on Xsun:

           value  -------------- Distribution ----------- count
           32768 |                                         0
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@                  67
          131072 |@@@@@                                    16
          262144 |@@                                       6
          524288 |@                                        3
         1048576 |@@@@@                                    15
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         1
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         1
      1073741824 |                                         1
      2147483648 |                                         2
      4294967296 |                                         2
      8589934592 |                                         2
```

```
       17179869184 |                                                0
       34359738368 |                                                2
       68719476736 |                                                0

syslogd sleeping on syslogd:

             value  -------------- Distribution ------------ count
       17179869184 |                                                0
       34359738368 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
       68719476736 |                                                0

MozillaFirebird sleeping on MozillaFirebird:

             value  -------------- Distribution ------------ count
             65536 |                                                0
            131072 |                                                3
            262144 |@@                                              14
            524288 |                                                0
           1048576 |@@@                                             18
           2097152 |                                                0
           4194304 |                                                0
           8388608 |                                                1
          16777216 |                                                0
          33554432 |                                                1
          67108864 |                                                3
         134217728 |@                                               7
         268435456 |@@@@@@@@@@                                      53
         536870912 |@@@@@@@@@@@@@@                                  78
        1073741824 |@@@@                                            25
        2147483648 |                                                0
        4294967296 |                                                0
        8589934592 |@                                               7
       17179869184 |                                                0
```

You might want to understand how and why `MozillaFirebird` is blocking on itself. You could modify the above script as shown in the following example to answer this question:

```
#pragma D option quiet

sched:::sleep
/execname == "MozillaFirebird" && curlwpsinfo->pr_stype == SOBJ_CV/
{
        bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "MozillaFirebird" && bedtime[args[0]->pr_addr]/
{
        @[args[1]->pr_pid, args[0]->pr_lwpid, pid, curlwpsinfo->pr_lwpid] =
            quantize(timestamp - bedtime[args[0]->pr_addr]);
        bedtime[args[0]->pr_addr] = 0;
```

```
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
        bedtime[args[0]->pr_addr] = 0;
}

END
{
        printa("%d/%d sleeping on %d/%d:\n%@d\n", @);
}
```

Running the modified script for several seconds results in output similar to the following example:

```
# dtrace -s ./firebird.d
^C

 100459/1 sleeping on 100459/13:

         value  -------------- Distribution ------------ count
        262144 |                                         0
        524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
       1048576 |                                         0

100459/13 sleeping on 100459/1:

         value  -------------- Distribution ------------ count
      16777216 |                                         0
      33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
      67108864 |                                         0

100459/1 sleeping on 100459/2:

         value  -------------- Distribution ------------ count
         16384 |                                         0
         32768 |@@@@                                     5
         65536 |@                                        2
        131072 |@@@@@                                    6
        262144 |                                         1
        524288 |@                                        2
       1048576 |                                         0
       2097152 |@@                                       3
       4194304 |@@@@                                     5
       8388608 |@@@@@@@                                  9
      16777216 |@@@@@                                    6
      33554432 |@@                                       3
      67108864 |                                         0

100459/1 sleeping on 100459/5:
```

```
              value  -------------- Distribution ------------ count
              16384 |                                          0
              32768 |@@@@@                                     12
              65536 |@@                                        5
             131072 |@@@@@@                                    15
             262144 |                                          1
             524288 |                                          1
            1048576 |                                          2
            2097152 |@                                         4
            4194304 |@@@@@                                     13
            8388608 |@@@                                       8
           16777216 |@@@@@                                     13
           33554432 |@@                                        6
           67108864 |@@                                        5
          134217728 |@                                         4
          268435456 |                                          0
          536870912 |                                          1
         1073741824 |                                          0


100459/2 sleeping on 100459/1:

              value  -------------- Distribution ------------ count
              16384 |                                          0
              32768 |@@@@@@@@@@@@@@                            11
              65536 |                                          0
             131072 |@@                                        2
             262144 |                                          0
             524288 |                                          0
            1048576 |@@@@                                      3
            2097152 |@                                         1
            4194304 |@@                                        2
            8388608 |@@                                        2
           16777216 |@                                         1
           33554432 |@@@@@@                                    5
           67108864 |                                          0
          134217728 |                                          0
          268435456 |                                          0
          536870912 |@                                         1
         1073741824 |@                                         1
         2147483648 |@                                         1
         4294967296 |                                          0


100459/5 sleeping on 100459/1:

              value  -------------- Distribution ------------ count
              16384 |                                          0
              32768 |                                          1
              65536 |                                          2
             131072 |                                          4
             262144 |                                          7
             524288 |                                          1
```

```
        1048576 |                                          5
        2097152 |                                          10
        4194304 |@@@@@@                                    77
        8388608 |@@@@@@@@@@@@@@@@@@@@@@                    270
       16777216 |@@@                                       43
       33554432 |@                                         20
       67108864 |@                                         14
      134217728 |                                          5
      268435456 |                                          2
      536870912 |                                          1
     1073741824 |                                          0
```

You can also use the sleep and wakeup probes to understand the performance of door servers
such as the name service cache daemon, as shown in the following example:

```
sched:::sleep
/curlwpsinfo->pr_stype == SOBJ_SHUTTLE/
{
        bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "nscd" && bedtime[args[0]->pr_addr]/
{
        @[stringof(curpsinfo->pr_fname), stringof(args[1]->pr_fname)] =
            quantize(timestamp - bedtime[args[0]->pr_addr]);
        bedtime[args[0]->pr_addr] = 0;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
        bedtime[args[0]->pr_addr] = 0;
}
```

The tail of the output from running the above script on a large mail server resembles the
following example:

```
imapd
          value  ------------- Distribution ------------ count
          16384 |                                         0
          32768 |                                         2
          65536 |@@@@@@@@@@@@@@@@@@@                       57
         131072 |@@@@@@@@@@@                               37
         262144 |                                         3
         524288 |@@@                                       11
        1048576 |@@@                                       10
        2097152 |@@                                        9
        4194304 |                                         1
        8388608 |                                         0
```

```
mountd
         value  -------------- Distribution ------------ count
         65536 |                                         0
        131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@             49
        262144 |@@@                                      6
        524288 |                                         1
       1048576 |                                         0
       2097152 |                                         0
       4194304 |@@@@                                     7
       8388608 |@                                        3
      16777216 |                                         0

sendmail
         value  -------------- Distribution ------------ count
         16384 |                                         0
         32768 |@                                        18
         65536 |@@@@@@@@@@@@@@@@@                        205
        131072 |@@@@@@@@@@@@@                            154
        262144 |@                                        23
        524288 |                                         5
       1048576 |@@@@                                     50
       2097152 |                                         7
       4194304 |                                         5
       8388608 |                                         2
      16777216 |                                         0

automountd
         value  -------------- Distribution ------------ count
         32768 |                                         0
         65536 |@@@@@@@@@@                               22
        131072 |@@@@@@@@@@@@@@@@@@@@@@@@                  51
        262144 |@@                                       6
        524288 |                                         1
       1048576 |                                         0
       2097152 |                                         2
       4194304 |                                         2
       8388608 |                                         1
      16777216 |                                         1
      33554432 |                                         1
      67108864 |                                         0
     134217728 |                                         0
     268435456 |                                         1
     536870912 |                                         0
```

You might be interested in the unusual data points for automountd or the persistent data point at over one millisecond for sendmail. You can add additional predicates to the above script to hone in on the causes of any exceptional or anomalous

preempt and remain-cpu

Because Oracle Solaris is a preemptive system, higher priority threads preempt lower priority ones. Preemption can induce a significant latency bubble in the lower priority thread, so you might want to know which threads are being preempted by which other threads. The following example shows how to use the preempt and remain-cpu probes to display this information:

```
#pragma D option quiet

sched:::preempt
{
        self->preempt = 1;
}

sched:::remain-cpu
/self->preempt/
{
        self->preempt = 0;
}

sched:::off-cpu
/self->preempt/
{
        /*
         * If you want to preempt yourself, see who you ended up giving
         * the CPU to.
         */
        @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname,
            curlwpsinfo->pr_pri] = count();
        self->preempt = 0;
}

END
{
        printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",
            "PREEMPTED", "PRI", "#");
        printa("%30s %3d %30s %3d %5@d\n", @);
}
```

Running the above script for several seconds on a system results in output similar to the following example:

```
# dtrace -s ./whopreempt.d
^C
                     PREEMPTOR PRI                      PREEMPTED PRI      #
                         sched  60                           Xsun  53      1
                         xterm  59                           Xsun  53      1
                MozillaFirebird  57                          Xsun  53      1
                        mpstat 100                          fvwm2  59      1
                         sched  99                 MozillaFirebird  57      1
                         sched  60                         dtrace  30      1
```

```
                      mpstat 100                          Xsun  59     2
                        sched  60                         Xsun  54     2
                        sched  99                        sched  60     2
                        fvwm2  59                         Xsun  44     2
                        sched  99                         Xsun  44     2
                        sched  60                        xterm  59     2
                        sched  99                         Xsun  53     2
                        sched  99                         Xsun  54     3
                        sched  60                        fvwm2  59     3
                        sched  60                         Xsun  59     3
                        sched  99                         Xsun  59     4
                        fvwm2  59                         Xsun  54     8
                        fvwm2  59                         Xsun  53     9
                         Xsun  59            MozillaFirebird  57    10
                        sched  60            MozillaFirebird  57    14
              MozillaFirebird  57                         Xsun  44    16
              MozillaFirebird  57                         Xsun  54    18
```

change-pri

Preemption is based on priorities, so you might want to observe changes in priority over time.
The following example uses the change-pri probe to display this information:

```
sched:::change-pri
{
        @[stringof(args[0]->pr_clname)] =
            lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);
}
```

The example script captures the degree to which priority is raised or lowered, and aggregates by
scheduling class. Running the above script results in output similar to the following example:

```
# dtrace -s ./pri.d
dtrace: script './pri.d' matched 10 probes
^C
 IA
           value  -------------- Distribution ------------ count
           < -50 |                                         20
             -50 |@                                        38
             -45 |                                         4
             -40 |                                         13
             -35 |                                         12
             -30 |                                         18
             -25 |                                         18
             -20 |                                         23
             -15 |                                         6
             -10 |@@@@@@@@                                 201
              -5 |@@@@@@                                   160
               0 |@@@@@                                    138
               5 |@                                        47
```

```
          10 |@@                                    66
          15 |@                                     36
          20 |@                                     26
          25 |@                                     28
          30 |                                      18
          35 |                                      22
          40 |                                      8
          45 |                                      11
       >= 50 |@                                     34

  TS
           value  ------------- Distribution ------------ count
            -15 |                                       0
            -10 |@                                      1
             -5 |@@@@@@@@@@@@                           7
              0 |@@@@@@@@@@@@@@@@@@@@@@                  12
              5 |                                       0
             10 |@@@@@                                  3
             15 |                                       0
```

The output shows the priority manipulation of the Interactive (IA) scheduling class. Instead of seeing priority manipulation, you might want to see the priority values of a particular process and thread over time. The following script uses the change-pri probe to display this information:

```
#pragma D option quiet

BEGIN
{
        start = timestamp;
}

sched:::change-pri
/args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
{
        printf("%d %d\n", timestamp - start, args[2]);
}

tick-1sec
/++n == 5/
{
        exit(0);
}
```

To see the change in priorities over time, type the following command in one window:

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

In another window, run the script and redirect the output to a file:

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out
#
```

You can use the file /tmp/pritime.out that is generated above as input to plotting software to graphically display priority over time. gnuplot is a freely available plotting package that is included in the Oracle Solaris Freeware Companion CD. By default, gnuplot is installed in /opt/sfw/bin.

tick

Oracle Solaris uses *tick-based CPU accounting,* in which a system clock interrupt fires at a fixed interval and attributes CPU utilization to the threads and processes running at the time of the tick. The following example shows how to use the tick probe to observe this attribution:

```
# dtrace -n sched:::tick'{@[stringof(args[1]->pr_fname)] = count()}'
^C
  arch                                                          1
  sh                                                            1
  sed                                                           1
  echo                                                          1
  ls                                                            1
  FvwmAuto                                                      1
  pwd                                                           1
  awk                                                           2
  basename                                                      2
  expr                                                          2
  resize                                                        2
  tput                                                          2
  uname                                                         2
  fsflush                                                       2
  dirname                                                       4
  vim                                                           9
  fvwm2                                                        10
  ksh                                                          19
  xterm                                                        21
  Xsun                                                         93
  MozillaFirebird                                             260
```

The system clock frequency varies from operating system to operating system, but generally ranges from 25 hertz to 1024 hertz. The Oracle Solaris system clock frequency is adjustable, but defaults to 100 hertz.

The tick probe only fires if the system clock detects a runnable thread. To use the tick probe to observe the system clock's frequency, you must have a thread that is always runnable. In one window, create a looping shell as shown in the following example:

```
$ while true ; do let i=0 ; done
```

n another window, run the following script:

```
uint64_t last[int];

sched:::tick
/last[cpu]/
{
        @[cpu] = min(timestamp - last[cpu]);
}

sched:::tick
{
        last[cpu] = timestamp;
}

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

  0            9883789
```

The minimum interval is 9.8 millisecond, which indicates that the default clock tick frequency is 10 milliseconds (100 hertz). The observed minimum is somewhat less than 10 milliseconds due to jitter.

One deficiency of tick-based accounting is that the system clock that performs accounting is often also responsible for dispatching any time-related scheduling activity. As a result, if a thread is to perform some amount of work every clock tick, that is every 10 milliseconds, the system will either over-account for the thread or under-account for the thread, depending on whether the accounting is done before or after time-related dispatching scheduling activity. In Oracle Solaris, accounting is performed before time-related dispatching. As a result, the system will under-account for threads running at regular interval. If such threads run for less than the clock tick interval, they can effectively hide behind the clock tick. The following example shows the degree to which the system has such threads:

```
sched:::tick,
sched:::enqueue
{
        @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

The output of the example script is two distributions of the millisecond offset within a ten millisecond interval, one for the tick probe and another for enqueue:

```
# dtrace -s ./tick.d
dtrace: script './tick.d' matched 4 probes
^C
  tick
          value  -------------- Distribution ------------ count
              6 |                                         0
              7 |@                                        3
              8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   79
              9 |                                         0
```

```
enqueue
        value  ------------- Distribution ------------ count
          < 0 |                                         0
            0 |@@                                       267
            1 |@@                                       300
            2 |@@                                       259
            3 |@@                                       291
            4 |@@@                                      360
            5 |@@                                       305
            6 |@@                                       295
            7 |@@@@                                     522
            8 |@@@@@@@@@@@                              1315
            9 |@@@                                      337
```

The output histogram named `tick` shows that the clock tick is firing at an 8 millisecond offset. If scheduling were not at all associated with the clock tick, the output for enqueue would be evenly spread across the ten millisecond interval. However, the output shows a spike at the same 8 millisecond offset, indicating that at least some threads in the system are being scheduled on a time basis.

cpucaps-sleep and **cpucaps-wakeup**

You can use `cpucaps-sleep` and `cpucaps-wakeup` probes to understand the impact CPU Caps have on specific processes and threads. The following example shows how much various processes spend on wait queues:

```
sched:::cpucaps-sleep
{
  sleep[args[1]->pr_pid] = timestamp;
}

sched:::cpucaps-wakeup
/sleep[args[1]->pr_pid]/
{
  @sleeps[args[1]->pr_fname] = quantize(timestamp - sleep[args[1]->pr_pid]);
  sleep[args[1]->pr_pid] = 0;
}
```

Running the above script results in output similar to the following example:

```
# dtrace -s

./capswait.d
dtrace: script './capswait.d' matched 2 probes
^C

  exmh
        value  ------------- Distribution ------------- count
```

```
        8388608 |                                            0
       16777216 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
       33554432 |                                            0

  scan
         value  ------------- Distribution ------------- count
       16777216 |                                            0
       33554432 |@@@@@@@@@@@@@@@@@@@@                         1
       67108864 |                                            0
      134217728 |@@@@@@@@@@@@@@@@@@@@                         1
      268435456 |                                            0

  firefox-bin
         value  ------------- Distribution ------------- count
        4194304 |                                            0
        8388608 |@@                                          1
       16777216 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     19
       33554432 |@@@@                                        2
       67108864 |                                            0
```

Stability

The `sched` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 68   Stability Mechanism for the `sched` Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

`io` Provider

The `io` provider makes available probes related to disk input and output. The `io` provider enables quick exploration of behavior observed through I/O monitoring tools such as `iostat`. For example, using the `io` provider, you can understand I/O by device, by I/O type, by I/O size, by process, by application name, by file name, or by file offset.

Probes

The `io` probes are described in the following table.

TABLE 69   `io` Probes

Probe	Description
start	Fires when an I/O request is about to be made either to a peripheral device or to an NFS server. The `bufinfo_t` corresponding to the I/O request is pointed to by `args[0]`. The `devinfo_t` of the device to which the I/O is being issued is pointed to by `args[1]`. The `fileinfo_t` of the file that corresponds to the I/O request is pointed to by `args[2]`. Note that file information availability depends on the filesystem making the I/O request. See `fileinfo_t` for more information.
done	Fires after an I/O request has been fulfilled. The `bufinfo_t` corresponding to the I/O request is pointed to by `args[0]`. The done probe fires after the I/O completes, but before completion processing has been performed on the buffer. As a result `B_DONE` is not set in `b_flags` at the time the done probe fires. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file that corresponds to the I/O request is pointed to by `args[2]`.
wait-start	Fires immediately before a thread begins to wait pending completion of a given I/O request. The `bufinfo_t` structure corresponding to the I/O request for which the thread will wait is pointed to by `args[0]`. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file that corresponds to the I/O request is pointed to by `args[2]`. Some time after the wait-start probe fires, the wait-done probe will fire in the same thread.
wait-done	Fires when a thread is done waiting for the completion of a given I/O request. The `bufinfo_t` corresponding to the I/O request for which the thread will wait is pointed to by `args[0]`. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file that corresponds to the I/O request is pointed to by `args[2]`. The wait-done probe fires only after the `wait-start` probe has fired in the same thread.

Note that the `io` probes fire for all I/O requests to peripheral devices, and for all file read and file write requests to an NFS server. Requests for metadata from an NFS server, for example, do not trigger `io` probes due to a readdir(3C) request.

Arguments

The argument types for the `io` probes are listed in . The arguments are described in .

TABLE 70   io Probe Arguments

Probe	args[0]	args[1]	args[2]
start	bufinfo_t *	devinfo_t *	fileinfo_t *
done	bufinfo_t *	devinfo_t *	fileinfo_t *
wait-start	bufinfo_t *	devinfo_t *	fileinfo_t *
wait-done	bufinfo_t *	devinfo_t *	fileinfo_t *

Each io probe has arguments consisting of a pointer to a bufinto_t structure, a pointer to a devinfo_t, and a pointer to a fileinfo_t. These structures are described in greater detail in this section.

bufinfo_t structure

The bufinfo_t structure is the abstraction that describes an I/O request. The buffer corresponding to an I/O request is pointed to by args[0] in the start, done, wait-start, and wait-done probes. The bufinfo_t structure definition is as follows:

```
typedef struct bufinfo {
        int b_flags;                    /* flags */
        size_t b_bcount;                /* number of bytes */
        caddr_t b_addr;                 /* buffer address */
        uint64_t b_blkno;               /* expanded block # on device */
        uint64_t b_lblkno;              /* block # on device */
        size_t b_resid;                 /* # of bytes not transferred */
        size_t b_bufsize;               /* size of allocated buffer */
        caddr_t b_iodone;               /* I/O completion routine */
        int b_error;                    /* expanded error field */
        dev_t b_edev;                   /* extended device */
 } bufinfo_t;
```

The b_flags member indicates the state of the I/O buffer, and consists of a bitwise-or of different state values. The valid state values are in Table 71, "b_flags Values," on page 299.

TABLE 71   b_flags Values

B_DONE	Indicates that the data transfer has completed.
B_ERROR	Indicates an I/O transfer error. It is set in conjunction with the b_error field.
B_PAGEIO	Indicates that the buffer is being used in a paged I/O request. See the description of the b_addr field for more information.
B_PHYS	Indicates that the buffer is being used for physical (direct) I/O to a user data area.

B_READ	Indicates that data is to be read from the peripheral device into main memory.
B_WRITE	Indicates that the data is to be transferred from main memory to the peripheral device.
B_ASYNC	The I/O request is asynchronous, and will not be waited upon. The wait-start and wait-done probes do not fire for asynchronous I/O requests. Note that some I/Os directed to be asynchronous might not have B_ASYNC set: the asynchronous I/O subsystem might implement the asynchronous request by having a separate worker thread perform a synchronous I/O operation.

The b_bcount field is the number of bytes to be transferred as part of the I/O request.

The b_addr field is the virtual address of the I/O request, unless B_PAGEIO is set. The address is a kernel virtual address unless B_PHYS is set, in which case it is a user virtual address. If B_PAGEIO is set, the b_addr field contains kernel private data. Exactly one of B_PHYS and B_PAGEIO can be set, or neither flag will be set.

The b_lblkno field identifies which logical block on the device is to be accessed. The mapping from a logical block to a physical block (such as the cylinder, track, and so on) is defined by the device.

The b_resid field is set to the number of bytes not transferred because of an error.

The b_bufsize field contains the size of the allocated buffer.

The b_iodone field identifies a specific routine in the kernel that is called when the I/O is complete.

The b_error field may hold an error code returned from the driver in the event of an I/O error. b_error is set in conjunction with the B_ERROR bit set in the b_flags member.

The b_edev field contains the major and minor device numbers of the device accessed. Consumers may use the D subroutines getmajor and getminor to extract the major and minor device numbers from the b_edev field.

devinfo_t

The devinfo_t structure provides information about a device. The devinfo_t structure corresponding to the destination device of an I/O is pointed to by args[1] in the start, done, wait-start, and wait-done probes. The members of devinfo_t are as follows:

```
typedef struct devinfo {
        int dev_major;                  /* major number */
        int dev_minor;                  /* minor number */
        int dev_instance;               /* instance number */
```

```
        string dev_name;                /* name of device */
        string dev_statname;            /* name of device + instance/minor */
        string dev_pathname;            /* pathname of device */
} devinfo_t;
```

The dev_major field is the major number of the device. See *getmajor(9F)* for more information.

The dev_minor field is the minor number of the device. See *getminor(9F)* for more information.

The dev_instance field is the instance number of the device. The instance of a device is different from the minor number. The minor number is an abstraction managed by the device driver. The instance number is a property of the device node. You can display device node instance numbers with *prtconf(1M)*.

The dev_name field is the name of the device driver that manages the device. You can display device driver names with the -D option to *prtconf(1M)*.

The dev_statname field is the name of the device as reported by iostat. This name also corresponds to the name of a kernel statistic as reported by *kstat(1M)*. This field is provided so that aberrant iostat or kstat output can be quickly correlated to actual I/O activity.

The dev_pathname field is the full path of the device. This path may be specified as an argument to prtconf to obtain detailed device information. The path specified by dev_pathname includes components expressing the device node, the instance number, and the minor node. However, all three of these elements are not necessarily expressed in the statistics name. For some devices, the statistics name consists of the device name and the instance number. For other devices, the name consists of the device name and the number of the minor node. As a result, two devices that have the same dev_statname may differ in dev_pathname.

fileinfo_t

The fileinfo_t structure provides information about a file. The file to which an I/O corresponds is pointed to by args[2] in the start, done, wait-start, and wait-done probes. The presence of file information is contingent upon the filesystem providing this information when dispatching I/O requests. Some filesystems, especially third-party filesystems, might not provide this information. Also, I/O requests might emanate from a filesystem for which no file information exists. For example, any I/O to filesystem metadata will not be associated with any one file. Finally, some highly optimized filesystems might aggregate I/O from disjoint files into a single I/O request. In this case, the filesystem might provide the file information either for the file that represents the majority of the I/O or for the file that represents some of the I/O. Alternately, the filesystem might provide no file information at all in this case.

The definition of the fileinfo_t structure is as follows:

```
typedef struct fileinfo {
        string fi_name;                 /* name (basename of fi_pathname) */
```

```
               string fi_dirname;              /* directory (dirname of fi_pathname) */
               string fi_pathname;             /* full pathname */
               offset_t fi_offset;             /* offset within file */
               string fi_fs;                   /* filesystem */
               string fi_mount;                /* mount point of file system */
       } fileinfo_t;
```

The fi_name field contains the name of the file but does not include any directory components. If no file information is associated with an I/O, the fi_name field will be set to the string <none>. In some rare cases, the pathname associated with a file might be unknown. In this case, the fi_name field will be set to the string <unknown>.

The fi_dirname field contains only the directory component of the file name. As with fi_name, this string may be set to <none> if no file information is present, or <unknown> if the pathname associated with the file is not known.

The fi_pathname field contains the full pathname to the file. As with fi_name, this string may be set to <none> if no file information is present, or <unknown> if the pathname associated with the file is not known.

The fi_offset field contains the offset within the file , or -1 if either file information is not present or if the offset is otherwise unspecified by the filesystem.

Examples

The following example script displays pertinent information for every I/O as it's issued:

```
#pragma D option quiet

BEGIN
{
        printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}

io:::start
{
        printf("%10s %58s %2s\n", args[1]->dev_statname,
            args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W");
}
```

The output of the example when cold-starting Acrobat Reader on an x86 system resembles the following example:

```
# dtrace -s ./iosnoop.d
    DEVICE                                                     FILE RW
     cmdk0                               /opt/Acrobat4/bin/acroread  R
     cmdk0                               /opt/Acrobat4/bin/acroread  R
```

```
cmdk0                                            <unknown>  R
cmdk0                     /opt/Acrobat4/Reader/AcroVersion  R
cmdk0                                            <unknown>  R
cmdk0                                            <unknown>  R
cmdk0                                               <none>  R
cmdk0                                            <unknown>  R
cmdk0                                               <none>  R
cmdk0           /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0           /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0           /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                                               <none>  R
cmdk0                                            <unknown>  R
cmdk0                                            <unknown>  R
cmdk0                                            <unknown>  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                               <none>  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                            <unknown>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0                                               <none>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0            /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0            /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                            <unknown>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0                                               <none>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
  ...
```

The <none> entries in the output indicate that the I/O does not correspond to the data in any particular file: these I/Os are due to metadata of one form or another. The <unknown> entries in the output indicate that the pathname for the file is not known. This situation is relatively rare.

You could make the example script slightly more sophisticated by using an associative array to track the time spent on each I/O, as shown in the following example:

```
#pragma D option quiet

BEGIN
{
        printf("%10s %58s %2s %7s\n", "DEVICE", "FILE", "RW", "MS");
}

io:::start
{
        start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
        this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
        printf("%10s %58s %2s %3d.%03d\n", args[1]->dev_statname,
            args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W",
            this->elapsed / 10000000, (this->elapsed / 1000) % 1000);
        start[args[0]->b_edev, args[0]->b_blkno] = 0;
}
```

The output of the above example while hot-plugging a USB storage device into an otherwise idle x86 system is shown in the following example:

```
# dtrace -s ./iotime.d
    DEVICE                                                     FILE RW      MS
      cmdk0                                  /kernel/drv/scsa2usb  R  24.781
      cmdk0                                  /kernel/drv/scsa2usb  R  25.208
      cmdk0                                      /var/adm/messages  W  25.981
      cmdk0                                  /kernel/drv/scsa2usb  R   5.448
      cmdk0                                                <none>  W   4.172
      cmdk0                                  /kernel/drv/scsa2usb  R   2.620
      cmdk0                                      /var/adm/messages  W   0.252
      cmdk0                                             <unknown>  R   3.213
      cmdk0                                                <none>  W   3.011
      cmdk0                                             <unknown>  R   2.197
      cmdk0                                      /var/adm/messages  W   2.680
      cmdk0                                                <none>  W   0.436
      cmdk0                                      /var/adm/messages  W   0.542
      cmdk0                                                <none>  W   0.339
      cmdk0                                      /var/adm/messages  W   0.414
      cmdk0                                                <none>  W   0.344
      cmdk0                                      /var/adm/messages  W   0.361
      cmdk0                                                <none>  W   0.315
      cmdk0                                      /var/adm/messages  W   0.421
      cmdk0                                                <none>  W   0.349
      cmdk0                                                <none>  R   1.524
      cmdk0                                             <unknown>  R   3.648
      cmdk0                                    /usr/lib/librcm.so.1  R   2.553
      cmdk0                                    /usr/lib/librcm.so.1  R   1.332
      cmdk0                                    /usr/lib/librcm.so.1  R   0.222
```

```
    cmdk0                                 /usr/lib/librcm.so.1  R   0.228
    cmdk0                                 /usr/lib/librcm.so.1  R   0.927
    cmdk0                                               <none>  R   1.189
      ...
    cmdk0                         /usr/lib/devfsadm/linkmod  R   1.110
    cmdk0       /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   1.763
    cmdk0       /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   0.161
    cmdk0         /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.819
    cmdk0         /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.168
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.886
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.185
    cmdk0      /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.778
    cmdk0      /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.166
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   1.634
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   0.163
    cmdk0          /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.477
    cmdk0          /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.161
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.198
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.168
    cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.247
    cmdk0   /usr/lib/devfsadm/linkmod/SUNW_misc_link_i386.so  R   1.735
      ...
```

You can make several observations about the mechanics of the system based on this output. First, note the long time to perform the first several I/Os, which took about 25 milliseconds each. This time might have been due to the `cmdk0` device having been power managed on the laptop. Second, observe the I/O due to the `scsa2usb` driver loading to deal with USB Mass Storage device. Third, note the writes to `/var/adm/messages` as the device is reported. Finally, observe the reading of the device link generators (the files ending in `link.so`), which presumably deal with the new device.

The `io` provider enables in-depth understanding of `iostat` output. Assume you observe `iostat` output similar to the following example:

```
extended device statistics
device     r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
cmdk0      8.0    0.0  399.8    0.0  0.0  0.0    0.8   0   1
sd0        0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd2        0.0  109.0    0.0  435.9  0.0  1.0    8.9   0  97
nfs1       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
nfs2       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
```

You can use the `iotime.d` script to see these I/Os as they happen, as shown in the following example:

```
 DEVICE                                       FILE RW      MS
     sd2                            /mnt/archives.tar  W   0.856
     sd2                            /mnt/archives.tar  W   0.729
     sd2                            /mnt/archives.tar  W   0.890
     sd2                            /mnt/archives.tar  W   0.759
```

```
        sd2                         /mnt/archives.tar  W   0.884
        sd2                         /mnt/archives.tar  W   0.746
        sd2                         /mnt/archives.tar  W   0.891
        sd2                         /mnt/archives.tar  W   0.760
        sd2                         /mnt/archives.tar  W   0.889
      cmdk0            /export/archives/archives.tar  R   0.827
        sd2                         /mnt/archives.tar  W   0.537
        sd2                         /mnt/archives.tar  W   0.887
        sd2                         /mnt/archives.tar  W   0.763
        sd2                         /mnt/archives.tar  W   0.878
        sd2                         /mnt/archives.tar  W   0.751
        sd2                         /mnt/archives.tar  W   0.884
        sd2                         /mnt/archives.tar  W   0.760
        sd2                         /mnt/archives.tar  W   3.994
        sd2                         /mnt/archives.tar  W   0.653
        sd2                         /mnt/archives.tar  W   0.896
        sd2                         /mnt/archives.tar  W   0.975
        sd2                         /mnt/archives.tar  W   1.405
        sd2                         /mnt/archives.tar  W   0.724
        sd2                         /mnt/archives.tar  W   1.841
      cmdk0            /export/archives/archives.tar  R   0.549
        sd2                         /mnt/archives.tar  W   0.543
        sd2                         /mnt/archives.tar  W   0.863
        sd2                         /mnt/archives.tar  W   0.734
        sd2                         /mnt/archives.tar  W   0.859
        sd2                         /mnt/archives.tar  W   0.754
        sd2                         /mnt/archives.tar  W   0.914
        sd2                         /mnt/archives.tar  W   0.751
        sd2                         /mnt/archives.tar  W   0.902
        sd2                         /mnt/archives.tar  W   0.735
        sd2                         /mnt/archives.tar  W   0.908
        sd2                         /mnt/archives.tar  W   0.753
```

This output appears to show that the file archives.tar is being read from cmdk0 in /export/
archives, and being written to device sd2 in /mnt. This existence of two files named
archives.tar that are being operated on separately in parallel seems unlikely. To investigate
further, you can aggregate on device, application, process ID and bytes transferred, as shown in
the following example:

```
#pragma D option quiet

io:::start
{
        @[args[1]->dev_statname, execname, pid] = sum(args[0]->b_bcount);
}

END
{
        printf("%10s %20s %10s %15s\n", "DEVICE", "APP", "PID", "BYTES");
        printa("%10s %20s %10d %15@d\n", @);
}
```

Running this script for a few seconds results in output similar to the following example:

```
# dtrace -s ./whoio.d
^C
    DEVICE              APP      PID        BYTES
    cmdk0                cp      790      1515520
      sd2                cp      790      1527808
```

This output shows that this activity is a copy of the file archives.tar from one device to another. This conclusion leads to another natural question: is one of these devices faster than the other? Which device acts as the limiter on the copy? To answer these questions, you need to know the effective throughput of each device rather than the number of bytes per second each device is transferring. You can determine the throughput with the following example script:

```
#pragma D option quiet

io:::start
{
        start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
        /*
         * You want to get an idea of the throughput to this device in KB/sec.
         * What you have, however, is nanoseconds and bytes.  That is what you want
         * to calculate:
         *
         *                        bytes / 1024
         *              ------------------------
         *                  nanoseconds / 1000000000
         *
         * But you cannot calculate this using integer arithmetic without losing
         * precision (the denomenator, for one, is between 0 and 1 for nearly
         * all I/Os).  So restate the fraction, and cancel:
         *
         *      bytes       1000000000        bytes         976562
         *    --------- * -------------  =  --------- * -------------
         *      1024        nanoseconds        1          nanoseconds
         *
         * This is easy to calculate using integer arithmetic; this is what
         * you do below.
         */
        this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
        @[args[1]->dev_statname, args[1]->dev_pathname] =
            quantize((args[0]->b_bcount * 976562) / this->elapsed);
        start[args[0]->b_edev, args[0]->b_blkno] = 0;
}

END
```

```
{
        printa("  %s (%s)\n%@d\n", @);
}
```

Running the example script for several seconds yields the following output:

```
sd2 (/devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0:r)

          value  ------------- Distribution ------------- count
             32 |                                         0
             64 |                                         3
            128 |                                         1
            256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2257
            512 |                                         1
           1024 |                                         0


cmdk0 (/devices/pci@0,0/pci-ide@1f,1/ide@0/cmdk@0,0:a)

          value  ------------- Distribution ------------- count
            128 |                                         0
            256 |                                         1
            512 |                                         0
           1024 |                                         2
           2048 |                                         0
           4096 |                                         2
           8192 |@@@@@@@@@@@@@@@@@@                        172
          16384 |@@@@@                                    52
          32768 |@@@@@@@@@@@                              108
          65536 |@@@                                      34
         131072 |                                         0
```

The output shows that sd2 is clearly the limiting device. The sd2 throughput is between 256K/sec and 512K/sec, while cmdk0 is delivering I/O at anywhere from 8 MB/second to over 64 MB/second. The script prints out both the name as seen in iostat, and the full path of the device. To find out more about the device, you could specify the device path to prtconf, as shown in the following example:

```
# prtconf -v /devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0
disk, instance #2 (driver name: sd)
    Driver properties:
        name='lba-access-ok' type=boolean dev=(29,128)
        name='removable-media' type=boolean dev=none
        name='pm-components' type=string items=3 dev=none
            value='NAME=spindle-motor' + '0=off' + '1=on'
        name='pm-hardware-state' type=string items=1 dev=none
            value='needs-suspend-resume'
        name='ddi-failfast-supported' type=boolean dev=none
        name='ddi-kernel-ioctl' type=boolean dev=none
    Hardware properties:
        name='inquiry-revision-id' type=string items=1
            value='1.04'
```

```
             name='inquiry-product-id' type=string items=1
                 value='STORAGE DEVICE'
             name='inquiry-vendor-id' type=string items=1
                 value='Generic'
             name='inquiry-device-type' type=int items=1
                 value=00000000
             name='usb' type=boolean
             name='compatible' type=string items=1
                 value='sd'
             name='lun' type=int items=1
                 value=00000000
             name='target' type=int items=1
                 value=00000000
```

As the emphasized terms indicate, this device is a removable USB storage device.

The examples in this section have explored all I/O requests. However, you might only be interested in one type of request. The following example tracks the directories in which writes are occurring, along with the applications performing the writes:

```
#pragma D option quiet

io:::start
/args[0]->b_flags & B_WRITE/
{
        @[execname, args[2]->fi_dirname] = count();
}

END
{
        printf("%20s %51s %5s\n", "WHO", "WHERE", "COUNT");
        printa("%20s %51s %5@d\n", @);
}
```

Running this example script on a desktop workload for a period of time yields some interesting results, as shown in the following example output:

```
# dtrace -s ./whowrite.d
^C
             WHO                                                 WHERE COUNT
              su                                              /var/adm     1
         fsflush                                                  /etc     1
         fsflush                                                     /     1
         fsflush                                              /var/log     1
         fsflush                                      /export/bmc/lisa     1
             esd   /export/bmc/.phoenix/default/78cxczuy.slt/Cache     1
         fsflush                              /export/bmc/.phoenix     1
             esd           /export/bmc/.phoenix/default/78cxczuy.slt     1
              vi                                              /var/tmp     2
              vi                                                  /etc     2
             cat                                                <none>     2
            bash                                                     /     2
```

```
            vi                                          <none>     3
         xterm                                        /var/adm     3
       fsflush                                     /export/bmc     7
 MozillaFirebird                                      <none>     8
           vim                                     /export/bmc     9
 MozillaFirebird                                   /export/bmc    10
       fsflush                                       /var/adm    11
      devfsadm                                           /dev    14
           ksh                                          <none>    71
           ksh                                     /export/bmc    71
       fsflush         /export/bmc/.phoenix/default/78cxczuy.slt   119
 MozillaFirebird         /export/bmc/.phoenix/default/78cxczuy.slt   119
       fsflush                                          <none>   211
 MozillaFirebird   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   591
       fsflush   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   666
         sched                                          <none>  2385
```

As the output indicates, virtually all writes are associated with the Mozilla Firebird cache.
The writes labeled <none> are likely due to writes associated with the UFS log, writes that are
themselves induced by other writes in the filesystem. See the ufs(7FS) man page for details on
logging. This example shows how to use the io provider to discover a problem at a much higher
layer of software. In this case, the script has revealed a configuration problem: the web browser
would induce much less I/O (and quite likely none at all) if its cache were in a directory in a
tmpfs(7FS) filesystem.

The previous examples have used only the start and done probes. You can use the wait-start
and wait-done probes to understand why applications block for I/O — and for how long. The
following example script uses both io probes and sched probes to derive CPU time compared
to I/O wait time for the StarOffice software.

```
#pragma D option quiet

sched:::on-cpu
/execname == "soffice.bin"/
{
        self->on = vtimestamp;
}

sched:::off-cpu
/self->on/
{
        @time["<on cpu>"] = sum(vtimestamp - self->on);
        self->on = 0;
}

io:::wait-start
/execname == "soffice.bin"/
{
        self->wait = timestamp;
}
```

```
io:::wait-done
/self->wait/
{
        @io[args[2]->fi_name] = sum(timestamp - self->wait);
        @time["<I/O wait>"] = sum(timestamp - self->wait);
        self->wait = 0;
}

END
{
        printf("Time breakdown (milliseconds):\n");
        normalize(@time, 1000000);
        printa("  %-50s %15@d\n", @time);

        printf("\nI/O wait breakdown (milliseconds):\n");
        normalize(@io, 1000000);
        printa("  %-50s %15@d\n", @io);
}
```

Running the example script during a cold start of the StarOffice software yields the following output:

```
Time breakdown (milliseconds):
  <on cpu>                                                      3634
  <I/O wait>                                                   13114

I/O wait breakdown (milliseconds):
  soffice.tmp                                                      0
  Office                                                           0
  unorc                                                            0
  sbasic.cfg                                                       0
  en                                                               0
  smath.cfg                                                        0
  toolboxlayout.xml                                                0
  sdraw.cfg                                                        0
  swriter.cfg                                                      0
  Linguistic.dat                                                   0
  scalc.cfg                                                        0
  Views.dat                                                        0
  Store.dat                                                        0
  META-INF                                                         0
  Common.xml.tmp                                                   0
  afm                                                              0
  libsimreg.so                                                     1
  xiiimp.so.2                                                      3
  outline                                                          4
  Inet.dat                                                         6
  fontmetric                                                       6
  ...
  libucb1.so                                                      44
  libj641si_g.so                                                  46
```

```
libX11.so.4                                                  46
liblng641si.so                                               48
swriter.db                                                   53
libwrp641si.so                                               53
liblocaledata_ascii.so                                       56
libi18npool641si.so                                          65
libdbtools2.so                                               69
ofa64101.res                                                 74
libxcr641si.so                                               82
libucpchelp1.so                                              83
libsot641si.so                                               86
libcppuhelper3C52.so                                         98
libfwl641si.so                                              100
libsb641si.so                                              104
libcomphelp2.so                                            105
libxo641si.so                                              106
libucpfile1.so                                             110
libcppu.so.3                                               111
sw64101.res                                                114
libdb-3.2.so                                               119
libtk641si.so                                              126
libdtransX11641si.so                                       127
libgo641si.so                                              132
libfwe641si.so                                             150
libi18n641si.so                                            152
libfwi641si.so                                             154
libso641si.so                                              173
libpsp641si.so                                             186
libtl641si.so                                              189
<unknown>                                                  189
libucbhelper1C52.so                                        195
libutl641si.so                                             213
libofa641si.so                                             216
libfwk641si.so                                             229
libsvl641si.so                                             261
libcfgmgr2.so                                              368
libsvt641si.so                                             373
libvcl641si.so                                             741
libsvx641si.so                                             885
libsfx641si.so                                             993
<none>                                                    1096
libsw641si.so                                             1365
applicat.rdb                                              1580
```

As this output shows, much of the cold StarOffice start time is due to waiting for I/O. 13.1 seconds waiting for I/O as opposed to 3.6 seconds on CPU. Running the script on a warm start of the StarOffice software reveals that page caching has eliminated the I/O time , as shown in the following example output:

```
Time breakdown (milliseconds):
  <I/O wait>                                                   0
```

```
  <on cpu>                                                    2860

I/O wait breakdown (milliseconds):
  temp                                                          0
  soffice.tmp                                                   0
  <unknown>                                                     0
  Office                                                        0
```

The cold start output shows that the file applicat.rdb accounts for more I/O wait time than any other file. This result is presumably due to many I/Os to the file. To explore the I/Os performed to this file, you can use the following D script:

```
io:::start
/execname == "soffice.bin" && args[2]->fi_name == "applicat.rdb"/
{
        @ = lquantize(args[2]->fi_offset != -1 ?
            args[2]->fi_offset / (1000 * 1024) : -1, 0, 1000);
}
```

This script uses the fi_offset field of the fileinfo_t structure to understand which parts of the file are being accessed, at the granularity of a megabyte. Running this script during a cold start of the StarOffice software results in output similar to the following example:

```
# dtrace -s ./applicat.d
dtrace: script './applicat.d' matched 4 probes
^C

            value  ------------- Distribution ------------  count
              < 0 |                                          0
                0 |@@@                                       28
                1 |@@                                        17
                2 |@@@@                                      35
                3 |@@@@@@@@@                                 72
                4 |@@@@@@@@@@                                78
                5 |@@@@@@@@                                  65
                6 |                                          0
```

This output indicates that only the first six megabytes of the file are accessed, perhaps because the file is six megabytes in size. The output also indicates that the entire file is not accessed. If you wanted to improve the cold start time of StarOffice, you might want to understand the access pattern of the file. If the needed sections of the file could be largely contiguous, one way to improve StarOffice cold start time might be to have a scout thread run ahead of the application, inducing the I/O to the file before it's needed. This approach is straightforward if the file is accessed using mmap. However, the approximatet 1.6 seconds that this strategy would gain in cold start time does not merit the additional complexity and maintenance burden in the application. Either way, the data gathered with the io provider allows a precise understanding of the benefit that such work could ultimately deliver.

Stability

The `io` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

Protocols

This section lists all the protocol providers.

`ip` Provider

The `ip` provider provides probes for tracing both IPv4 and IPv6 protocols.

Probes

The `ip` probes are described in the table below.

TABLE 72   `ip` Providers

Probe	Description
send	Fires whenever the kernel network stack sends an `ip` packet.
receive	Fires whenever the kernel network stack receives an `ip` packet.

These probes trace packets on physical interfaces and also packets on loopback interfaces that are processed by `ip`. An IP packet must have a full IP header to be visible by these probes.

Note - Loopback `tcp` packets on Oracle Solaris may be processed by `tcp` fusion, a performance feature that by-passes the `ip` layer. These fused packets will not be visible using the `ip:::send` and `ip:::receive` probes. They are typically all loopback `tcp` packets after the `tcp` handshake.

Arguments

The argument types for the `ip` probes are listed in the table below. The arguments are described in the following section.

TABLE 73   `ip` Probe Arguments

Probe	args[0]	args[1]	args[2]	args[3]	args[4]	args[5]
send	pktinfo_t *	csinfo_t *	ipinfo_t *	ifinfo_t *	ipv4info_t *	ipv6info_t *
receive	pktinfo_t *	csinfo_t *	ipinfo_t *	ifinfo_t *	ipv4info_t *	ipv6info_t *

`args[0] - pktinfo_t` **Structure**

The `pktinfo_t` structure is where packet ID info can be made available for deeper analysis if packet IDs become supported by the kernel in the future.

The `pkt_addr` member is currently always `NULL`.

```
typedef struct pktinfo {
        uintptr_t pkt_addr;            /* currently always NULL */
} pktinfo_t;
```

`args[1] - csinfo_t` **Structure**

The `csinfo_t` structure is where connection state info can be made available if connection IDs become supported by the kernel in the future.

The `cs_addr` member is currently always `NULL`.

```
typedef struct csinfo {
        uintptr_t cs_addr;            /* currently always NULL */
} csinfo_t;
```

`args[2] - ipinfo_t` **Structure**

The `ipinfo_t` structure contains common IP info for both IPv4 and IPv6.

```
typedef struct ipinfo {
        uint8_t ip_ver;                /* IP version (4, 6) */
        uint16_t ip_plength;           /* payload length */
        string ip_saddr;               /* source address */
        string ip_daddr;               /* destination address */
} ipinfo_t;
```

TABLE 74   `ipinfo_t` Members

`ip_ver`	IP version number. Currently either 4 or 6.
`ip_plength`	Payload length in bytes. This is the length of the packet at the time of tracing, excluding the IP header.
`ip_saddr.`	Source IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.
`ip_daddr`	Destination IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.

args[3] - ifinfo_t Structure

The `ifinfo_t` structure contains network interface information.

```
typedef struct ifinfo {
        string if_name;                /* interface name */
        int8_t if_local;               /* is delivered locally */
        netstackid_t if_ipstack;       /* ipstack ID */
        uintptr_t if_addr;             /* pointer to raw ill_t */
} ifinfo_t;
```

TABLE 75   `ifinfo_t` Members

`if_name`	Interface name as a string. For example, "eri0", "lo0", "ip.tun0", "<unknown>".
`if_local`	Is-local status. 1: is a local interface, 0: is not a local interface, -1: is unknown.
`if_ipstack`	`ipstack` ID, for associating `ip` stack instances, or NULL.
`if_addr`	Pointer to raw kernel structure for advanced debugging only.

The `ifinfo_t` details are provided for debugging convenience in the `ip` layer, if that information is available. There may be some types of traffic where some or all of that information is not available during the `ip` layer, for which the members may be: "`<null>`", `-1`, `NULL`, `NULL`.

args[4] - ipv4info_t Structure

The `ipv4info_t` structure is a DTrace translated version of the IPv4 header.

```
typedef struct ipv4info {
        uint8_t ipv4_ver;               /* IP version (4) */
        uint8_t ipv4_ihl;               /* header length, bytes */
        uint8_t ipv4_tos;               /* type of service field */
        uint16_t ipv4_length;           /* length (header + payload) */
        uint16_t ipv4_ident;            /* identification */
        uint8_t ipv4_flags;             /* IP flags */
        uint16_t ipv4_offset;           /* fragment offset */
        uint8_t ipv4_ttl;               /* time to live */
        uint8_t ipv4_protocol;          /* next level protocol */
        string ipv4_protostr;           /* next level protocol, as a string */
        uint16_t ipv4_checksum;         /* header checksum */
        ipaddr_t ipv4_src;              /* source address */
        ipaddr_t ipv4_dst;              /* destination address */
        string ipv4_saddr;              /* source address, string */
        string ipv4_daddr;              /* destination address, string */
        ipha_t *ipv4_hdr;               /* pointer to raw header */
} ipv4info_t;
```

TABLE 76   `ipv4info_t` Members

`ipv4_ver`	IP version 4.
`ipv4_ihl`	IPv4 header length, in bytes.
`ipv4_tos`	Contents of IPv4 type of service field.
`ipv4_length`	IPv4 packet length (header + payload) at time of tracing, in bytes.
`ipv4_ident`	IPv4 identification field.
`ipv4_flags`	IPv4 flags. See the `ipv4_flags` table below for bitwise values.
`ipv4_offset`	IPv4 fragment offset, in bytes.
`ipv4_ttl`	IPv4 time to live.
`ipv4_protocol`	IPv4 encapsulated protocol number. See `/usr/include/netinet/in.h` for the protocol list (IPPROTO_*).
`ipv4_protostr`	IPv4 encapsulated protocol, as a string. For example, "TCP".
`ipv4_checksum`	IPv4 header checksum, if available at time of tracing.
`ipv4_src`	IPv4 source address, as an `ipaddr_t`.
`ipv4_dst`	IPv4 destination address, as an `ipaddr_t`.
`ipv4_saddr`	IPv4 source address, as a dotted decimal quad string.
`ipv4_daddr`	IPv4 destination address, as a dotted decimal quad string.
`ipv4_hdr`	Pointer to raw IPv4 header at the time of tracing.

See RFC-791 for a detailed explanation for these IPv4 header fields. If the packet is IPv6, these members are either "`<null>`", `0`, or `NULL` depending on type.

TABLE 77   `ipv4_flags` Values

`IPH_DF`	Do not fragment

`IPH_MF`	More fragments

args[5] - `ipv6info_t` Structure

The `ipv6info_t` structure is a DTrace translated version of the IPv6 header.

```
typedef struct ipv6info {
        uint8_t ipv6_ver;               /* IP version (6) */
        uint8_t ipv6_tclass;            /* traffic class */
        uint32_t ipv6_flow;             /* flow label */
        uint16_t ipv6_plen;             /* payload length */
        uint8_t ipv6_nexthdr;           /* next header protocol */
        string ipv6_nextstr;            /* next header protocol, as a string*/
        uint8_t ipv6_hlim;              /* hop limit */
        in6_addr_t *ipv6_src;           /* source address */
        in6_addr_t *ipv6_dst;           /* destination address */
        string ipv6_saddr;              /* source address, string */
        string ipv6_daddr;              /* destination address, string */
        ip6_t *ipv6_hdr;                /* pointer to raw header */
} ipv6info_t;
```

TABLE 78   `ipv6info_t` Members

`ipv6_ver`	IP version (6).
`ipv6_tclass`	IPv6 traffic class.
`ipv6_plen`	IPv6 payload length at time of tracing, in bytes.
`ipv6_nexthdr`	IPv6 next header protocol number. See `/usr/include/netinet/in.h` for the protocol list (`IPPROTO_*`).
`ipv6_nextstr`	IPv6 next header protocol, as a string. For example, "`TCP`".
`ipv6_hlim`	IPv6 hop limit.
`ipv6_src`	IPv6 source address, as an `in6_addr_t`.
`ipv6_dst`	IPv6 destination address, as an `in6_addr_t`.
`ipv6_saddr`	IPv6 source address, as an RFC-1884 convention 2 string with lower case hexadecimal digits.
`ipv6_daddr`	IPv6 destination address, as an RFC-1884 convention 2 string with lower case hexadecimal digits.
`ipv6_hdr`	Pointer to raw IPv6 header at the time of tracing.

See RFC-2460 for a detailed explanation for these IPv6 header fields. If the packet is IPv4, these members are either "`<null>`", `0`, or NULL depending on type.

Examples

Some simple examples of `ip` provider usage follow.

Packets by host address

This DTrace one-liner counts received packets by host address:

```
# dtrace -n 'ip:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace: description 'ip:::receive ' matched 4 probes
^C

  192.168.1.5                                                      1
  192.168.1.185                                                    4
  fe80::214:4fff:fe3b:76c8                                         9
  127.0.0.1                                                       14
  192.168.1.109                                                   28
```

The output above shows that 28 IP packets were received from 192.168.1.109, 14 IP packets from 127.0.0.1, and so on.

Sent size distribution

This DTrace one-liner prints distribution plots of sent payload size by destination:

```
# dtrace -n 'ip:::send { @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
dtrace: description 'ip:::send ' matched 11 probes
^C

  192.168.2.27
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           7
              32 |@@@@                                      1
              64 |@@@@                                      1
             128 |                                         0

  192.168.1.109
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@                                     5
              32 |@@@                                       3
              64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@             24
             128 |@                                         1
             256 |@                                         1
             512 |@@                                        2
            1024 |@                                         1
            2048 |                                         0
```

ipio.d

The following DTrace script traces IP packets and prints various details:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf(" %3s %10s %15s     %15s %8s %6s\n", "CPU", "DELTA(us)",
            "SOURCE", "DEST", "INT", "BYTES");
        last = timestamp;
}

ip:::send
{
        this->elapsed = (timestamp - last) / 1000;
        printf(" %3d %10d %15s -> %15s %8s %6d\n", cpu, this->elapsed,
            args[2]->ip_saddr, args[2]->ip_daddr, args[3]->if_name,
            args[2]->ip_plength);
        last = timestamp;
}

ip:::receive
{
        this->elapsed = (timestamp - last) / 1000;
        printf(" %3d %10d %15s <- %15s %8s %6d\n", cpu, this->elapsed,
            args[2]->ip_daddr, args[2]->ip_saddr, args[3]->if_name,
            args[2]->ip_plength);
        last = timestamp;
}
```

This example output shows tracing packets as they pass in and out of tunnels:

```
# ./ipio.d
 CPU  DELTA(us)            SOURCE                   DEST      INT  BYTES
   1     598913     10.1.100.123 ->   192.168.10.75  ip.tun0     68
   1         73    192.168.1.108 ->     192.168.5.1     nge0    140
   1      18325    192.168.1.108 <-     192.168.5.1     nge0    140
   1         69     10.1.100.123 <-   192.168.10.75  ip.tun0     68
   0     102921     10.1.100.123 ->   192.168.10.75  ip.tun0     20
   0         79    192.168.1.108 ->     192.168.5.1     nge0     92
```

The fields printed are:

Field	Description
CPU	CPU id that event occurred on
DELTA(us)	elapsed time since previous event
SOURCE	source IP address
DEST	destination IP address
INT	interface name

Field	Description
BYTES	payload bytes

Note - The output may be shuffled slightly on multi-CPU servers due to DTrace per-CPU buffering; keep an eye on changes in the CPU column, or add a timestamp column and post sort.

---

ipproto.d

This DTrace script provides a neat summary for both send and receive IP traffic, including the next level protocol:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

ip:::send,
ip:::receive
{
        this->protostr = args[2]->ip_ver == 4 ?
            args[4]->ipv4_protostr : args[5]->ipv6_nextstr;
        @num[args[2]->ip_saddr, args[2]->ip_daddr, this->protostr] = count();
}

dtrace:::END
{
        printf("   %-28s %-28s %6s %8s\n", "SADDR", "DADDR", "PROTO", "COUNT");
        printa("   %-28s %-28s %6s %@8d\n", @num);
}
```

This script was run on a system with both IPv4 and IPv6 interfaces for several seconds:

```
# ./ipproto.d
Tracing... Hit Ctrl-C to end.
^C
   SADDR                        DADDR                          PROTO    COUNT
   192.168.1.108                192.168.155.32                   UDP        1
   192.168.1.108                192.168.17.55                    UDP        1
   192.168.1.108                192.168.228.54                   UDP        1
   192.168.1.108                192.168.1.5                      UDP        1
   192.168.1.108                192.168.2.27                    ICMP        1
   192.168.1.200                192.168.3.255                    UDP        1
   192.168.1.5                  192.168.1.108                    UDP        1
```

```
192.168.2.27           192.168.1.108            ICMP     1
fe80::214:4fff:fe3b:76c8  ff02::1                ICMPV6   1
fe80::2e0:81ff:fe5e:8308  fe80::214:4fff:fe3b:76c8  ICMPV6   1
fe80::2e0:81ff:fe5e:8308  ff02::1:2              UDP      1
192.168.1.185          192.168.1.255            UDP      2
192.168.1.211          192.168.1.255            UDP      3
192.168.1.109          192.168.1.108            TCP      428
192.168.1.108          192.168.1.109            TCP      789
```

The following table describes the fields that are printed.

Field	Description
SADDR	source IP address
DADDR	destination IP address
PROTO	IP next level protocol
COUNT	number of packets

The example output above provides a quick summary of network activity with host address details; you can see that both 192.168.1.109 and 192.168.1.108 are swapping many packets via TCP.

Stability

The ip provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 79   Stabililty Mechanism for ip Provider

Element	Name stability	Data stability	Dependency class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

iscsi Provider

The iscsi provider provides probes for tracing iSCSI target activity.

This is a kernel provider built into the COMSTAR iSCSI target port provider. The COMSTAR iSCSI target and the user-land iSCSI target `/usr/sbin/iscsitgtd` are mutually exclusive. Only one of the targets can be enabled at a time. The COMSTAR iSCSI target DTrace provider provides all the probes that are provided by the user-land iSCSI provider, so that any DTrace script written for the userland provider built into the iSCSI target daemon (`iscsitgtd`) will work with the COMSTAR iSCSI target port provider as well without any modification. Since this provider instruments the iSCSI target activity, DTrace commands and scripts must be run on the iSCSI target server to observe these probes.

Probes

SCSI Event	Probes
SCSI command/response	`iscsi:::scsi-commandiscsi:::scsi-response`
Data out/in/request (`rtt`)	`iscsi:::data-sendiscsi:::data-receiveiscsi:::data-request`
Login and logout command/response	`iscsi:::login-commandiscsi:::login-responseiscsi:::logout-commandiscsi:::logout-response`
NOP out/in (pings)	`iscsi:::nop-receiveiscsi:::nop-send`
Text and task command/response	`iscsi:::task-commandiscsi:::task-responseiscsi:::text-commandiscsi:::text-response`
Asynchronous message from target	`iscsi:::async-send`
Buffer dispatch and completion (not available with the USDT provider)	`iscsi:::xfer-startiscsi:::xfer-done`

Arguments

TABLE 80   `iscsi` Provider Arguments

Probes	Variable	Type	Description
`*`	`args[0]`	`conninfo_t *`	connection info
`*`	`args[1]`	`iscsiinfo_t *`	common iSCSI properties
`scsi-command`	`args[2]`	`scsicmd_t *`	SCSI command block (`cdb`)
`xfer-startxfer-done`	`args[2]`	`xferinfo_t *`	Buffer info

Types

All COMSTAR iSCSI target probes have the first and second argument in common:

args[0] conninfo_t * connection information

```
conninfo_t

typedef struct conninfo {
        string ci_local;        /* local host IP address */
        string ci_remote;       /* remote host IP address */
        string ci_protocol;     /* protocol ("ipv4", "ipv6") */
} conninfo_t;
The conninfo_t structure is used by NFSv4 provider,
Fibre Channel provider and is intended for use by all
application protocol providers as the first argument
to indicate some basic information about the connection.
```

args[1] iscsiinfo_t * common iSCSI properties

```
iscsiinfo_t

typedef struct iscsiinfo {
 string ii_target; /* target iqn */
 string ii_initiator; /* initiator iqn */
        string ii_isid;  /* initiator session identifier */
 string ii_tsih;  /* target session identifying handle */
 string ii_transport; /* transport type ("iser-ib", "sockets") */

 uint64_t ii_lun; /* target logical unit number */

 uint32_t ii_itt; /* initiator task tag */
 uint32_t ii_ttt; /* target transfer tag */

 uint32_t ii_cmdsn; /* command sequence number */
 uint32_t ii_statsn; /* status sequence number */
 uint32_t ii_datasn; /* data sequence number */

 uint32_t ii_datalen; /* length of data payload */
 uint32_t ii_flags; /* probe-specific flags */
} iscsiinfo_t;
The iscsiinfo_t structure is used to provide identifying
information about the target and the initiator and
also some PDU level information such as lun, data length and sequence numbers.
```

The third argument is only used for the SCSI command probe or the data transfer probe

args[2] scsicmd_t * SCSI command block (cdb)

```
scsicmd_t

typedef struct scsicmd {
        uint64_t ic_len;        /* CDB length */
        uint8_t *ic_cdb;        /* CDB data */
} scsicmd_t;
The scsicmd_t structure is used by the SCSI command probe
```

```
and it contains information about the SCSI command
blocks and is intended for use by all the application
protocols that deal with SCSI data.
```

Although the transport layer is transparent to the user, the COMSTAR iSCSI target also supports iSCSI over Remote DMA (RDMA), also known as iSER. Since the data transfer phases are mapped to RDMA operations in iSER, the data-send, data-receive and data-request probes cannot be used with iSER. Instead the xfer-start and xfer-done probes can be used to trace the data transfer irrespective of the transport used. The data-receive, data-request, and data-send probes can be used when a user wants to track the SCSI Data-IN and Data-OUT PDUs specifically.

```
args[2] xferinfo_t * data transfer information
```

```
xferinfo_t
```

```
typedef struct xferinfo {
 uintptr_t xfer_laddr; /* local buffer address */
 uint32_t xfer_loffset; /* offset within the local buffer */
 uint32_t xfer_lkey; /* access control to local memory */
 uintptr_t xfer_raddr; /* remote virtual address */
 uint32_t xfer_roffset; /* offset from the remote address */
 uint32_t xfer_rkey; /* access control to remote virtual address */
 uint32_t xfer_len; /* transfer length */
 uint32_t xfer_type; /* Read or Write */
} xferinfo_t;
The xferinfo_t structure is used by the xfer-start
and the xfer-done probes and contain information about the
data transfer. When the transport type is iSER,
the remote buffer information is given by the xfer_raddr,
xfer_rkey and xfer_roffset fields. It is set to 0 when the transport type is sockets.
```

Examples

One-liners

Frequency of iSCSI command types:

```
# dtrace -n 'iscsi*::: { @[probename] = count(); }'
```

Frequency of iSCSI client IP addresses:

```
# dtrace -n 'iscsi*::: { @[args[0]->ci_remote] = count(); }'
```

Payload bytes by iSCSI command type:

```
# dtrace -n 'iscsi*::: { @[probename] = sum(args[1]->ii_datalen); }'
```

Payload byte distribution by iSCSI command type:

```
# dtrace -n 'iscsi*::: { @[probename] = quantize(args[1]->ii_datalen); }'
```

iscsiwho.d

This is a simple script to produce a report of the remote IP addresses and a count of iSCSI events. This is intended to provide a quick summary of iSCSI activity when run on the iSCSI target server:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

iscsi*:::
{
        @events[args[0]->ci_remote, probename] = count();
}

dtrace:::END
{
        printf("  %-26s %14s %8s\n", "REMOTE IP", "iSCSI EVENT", "COUNT");
        printa("  %-26s %14s %@8d\n", @events);
}
```

This output shows the host and the number of iSCSI operations:

```
# ./iscsiwho.d
Tracing... Hit Ctrl-C to end.
^C
  REMOTE IP                  iSCSI EVENT    COUNT
  192.168.1.109              nop-receive        1
  192.168.1.109                 nop-send        1
  192.168.1.109            scsi-response      126
  192.168.1.109             scsi-command      208
  192.168.1.109             data-request     1207
  192.168.1.109             data-receive     1207
```

The following table describes the ouput fields of the iscsiwho.d script:

Field	Description
REMOTE IP	IP address of the client
iSCSI EVENT	iSCSI event type
COUNT	Number of events traced

The example output shows normal traffic. For this simple script, these event names are not translated beyond their iSCSI provider probe names, and require some thought to comprehend as they are from the perspective of the iSCSI target server.

iscsixfer.d

Although the transport layer is transparent to the user, the COMSTAR iSCSI target also supports iSCSI over RDMA, also known as iSER. An iSER initiator should be able to read and write data from an iSER target at high data rates with relatively low CPU utilization compared to iSCSI using TCP/IP. In order to see the transport layer in use, display the ii_transport field from the iscsiinfo_t structure.

Since the data transfer phases are mapped to RDMA operations in iSER, the data-send, data-receive and data-request probes cannot be used with iSER. Instead here is a simple script to print an aggregation of all the data transferred between two points using the xfer-start probe. This can be used for iSCSI using TCP/IP and iSCSI over Remote DMA.

The data-receive, data-request and data-send probes can be used when a user wants to track the SCSI Data-IN and Data-OUT PDUs specifically. For example, if the PDUs are received out of order, you might want to trace the ii_ttt, ii_datasn, and ii_statsn. To just get a trace of IO activity, the xfer-start/xfer-done probes should suffice.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

iscsi:::xfer-start
{
    @[args[0]->ci_remote, args[2]->xfer_type] = sum(args[2]->xfer_len);
}

END
{
    printf("%26s %10s %8s\n",  "REMOTE IP", "READ/WRITE", "BYTES");
    printa("%26s %10s %15@d\n", @);
}
```

This output shows the transfer of bytes:

```
# ./iscsixfer.d
Tracing... Hit Ctrl-C to end.
^C

  REMOTE IP                 READ/WRITE  BYTES
  192.168.1.109             write        464
  192.168.1.109             read        1024
```

The outpu fields are described in the following table:

Field	Description
REMOTE IP	IP address of the client
READ/WRITE	Read or write
BYTES	Number of bytes transferred

You can use the following script to see the data read or write as it happens.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

BEGIN
{
 printf(" %-26s %8s %10s\n", "REMOTE IP", "BYTES", "READ/WRITE");

}

iscsi:::xfer-start
{
  printf("%26s %%8d %108s\n", args[0]->ci_remote,
args[2]->xfer_len, args[2]->xfer_type);
}
```

The following table provides the interpretation for some of these events.

iSCSI event	Interpretation
scsi-command	A SCSI command was issued, such as a read or a write. Use other scripts for a breakdown on the SCSI command type.
data-send	Data was sent from the iSCSI target server to the client; the client is performing a read.
data-receive	sData was received on the iSCSI target server from the client. The client is performing a write.

nfsv3 Server Provider

The nfsv3 provider provides probes for tracing NFS version 3 server activity.

Arguments

All NFS operation probes have the first argument in common:

```
args[0]  conninfo_t *  socket connection information
```

The `conninfo_t` structure is already used by the iSCSI target provider (`iscsi`) and the NFS v4 provider (`nfsv4`), and is intended for use by all provider which are providing some higher level protocol, such as `iscsi`, `nfs`, `http`, and `ftp`.

```
typedef struct conninfo {
 string ci_local;       /* local host address */
 string ci_remote;      /* remote host address */
 string ci_protocol;    /* protocol (ipv4, ipv6, and so on) */
} conninfo_t;
```

Operation probes have their second argument in common:

```
args[1]  nfsv3opinfo_t *  NFS v3 operation properties

typedef struct nfsv3opinfo {
 string noi_curpath; /* current file handle path (if any) */
 cred_t *noi_cred; /* credentials */
 uint64_t noi_xid; /* transaction ID */
} nfsv4opinfo_t;
```

Probes

The following table lists the probes along with the specific argument for each whose type is defined by the NFS v3 specification:

Probe Name	args[2]
nfsv3:::op-access-start	ACCESS3args *
nfsv3:::op-access-done	ACCESS3res *
nfsv3:::op-commit-start	COMMIT3args *
nfsv3:::op-commit-done	COMMIT3res *
nfsv3:::op-create-start	CREATE3args *
nfsv3:::op-create-done	CREATE3res *
nfsv3:::op-fsinfo-start	FSINFO3args *
nfsv3:::op-fsinfo-done	FSINFO3res *
nfsv3:::op-fsstat-start	FSSTAT3args *
nfsv3:::op-fsstat-done	FSSTAT3res *
nfsv3:::op-getattr-start	GETATTR3args *
nfsv3:::op-getattr-done	GETATTR3res *
nfsv3:::op-lookup-start	LOOKUP3args *
nfsv3:::op-lookup-done	LOOKUP3res *

Probe Name	args[2]
nfsv3:::op-link-start	LINK3args *
nfsv3:::op-link-done	LINK3res *
nfsv3:::op-mkdir-start	MKDIR3args *
nfsv3:::op-mkdir-done	MKDIR3res *
nfsv3:::op-mknod-start	MKNOD3args *
nfsv3:::op-mknod-done-	MKNOD3res *
nfsv3:::op-null-start	-
nfsv3:::op-null-done	-
nfsv3:::op-pathconf-start	PATHCONF3args *
nfsv3:::op-pathconf-done	PATHCONF3res *
nfsv3:::op-read-start	READ3args *
nfsv3:::op-read-done	READ3res *
nfsv3:::op-readdir-start	READDIR3args *
nfsv3:::op-readdir-done	READDIR3res *
nfsv3:::op-readdirplus-start	READDIRPLUS3args *
nfsv3:::op-readdirplus-done	READDIRPLUS3res *
nfsv3:::op-readlink-start	READLINK3args *
nfsv3:::op-readlink-done	READLINK3res *
nfsv3:::op-remove-start	REMOVE3args *
nfsv3:::op-remove-done	REMOVE3res *
nfsv3:::op-renamestart	RENAME3args *
nfsv3:::op-rename-done	RENAME3res *
nfsv3:::op-rmdir-start	RMDIR3args *
nfsv3:::op-rmdir-done	RMDIR3res *
nfsv3:::op-setattr-start	SETATTR3args *
nfsv3:::op-setattr-done	SETATTR3res *
nfsv3:::op-symlink-start	SYMLINK3args *
nfsv3:::op-symlink-done	SYMLINK3res *
nfsv3:::op-write-start	WRITE3args *
nfsv3:::op-write-done	WRITE3res *

Note - The op-null-* probes have an undefined args[2].

Examples

Some examples of nfsv3 provider usage are as follows.

nfsv3rwsnoop.d

This DTrace scripts traces NFSv3 read and write requests, showing details of each operation.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-8s %6s %s\n", "TIME(us)",
            "CLIENT", "OP", "OFFSET", "BYTES", "PATHNAME");
}

nfsv3:::op-read-start
{
        printf("%-16d %-18s %2s %-8d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "R", args[2]->offset / 1024, args[2]->count,
            args[1]->noi_curpath);
}

nfsv3:::op-write-start
{
        printf("%-16d %-18s %2s %-8d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "W", args[2]->offset / 1024,
            args[2]->data.data_len, args[1]->noi_curpath);
}
```

The following output shows a read of /export/stuff/bin/ghex2, then a read of /export/stuff/bin/gksu, and finally a write of /export/stuff/words12:

```
# ./nfsv3iosnoop.d
TIME(us)          CLIENT             OP OFFSET     BYTES PATHNAME
4299383207        192.168.17.75      R 0            4096 /export/stuff/bin/ghex2
4299391813        192.168.17.75      R 4           28672 /export/stuff/bin/ghex2
4299395700        192.168.17.75      R 32          32768 /export/stuff/bin/ghex2
4299396038        192.168.17.75      R 96          32768 /export/stuff/bin/ghex2
4299396462        192.168.17.75      R 128          8192 /export/stuff/bin/ghex2
4299396550        192.168.17.75      R 64          32768 /export/stuff/bin/ghex2
4320233417        192.168.17.75      R 0            4096 /export/stuff/bin/gksu
4320240902        192.168.17.75      R 4           28672 /export/stuff/bin/gksu
4320242434        192.168.17.75      R 32          32768 /export/stuff/bin/gksu
4320242730        192.168.17.75      R 64          24576 /export/stuff/bin/gksu
4333460565        192.168.17.75      W 0           32768 /export/stuff/words12
4333461477        192.168.17.75      W 32          32768 /export/stuff/words12
4333463264        192.168.17.75      W 64          32768 /export/stuff/words12
4333463567        192.168.17.75      W 96          32768 /export/stuff/words12
4333463893        192.168.17.75      W 128         32768 /export/stuff/words12
4333464202        192.168.17.75      W 160         32768 /export/stuff/words12
4333464451        192.168.17.75      W 192         10055 /export/stuff/words12
```

The fields printed are:

Field	Description
TIME(us)	Time of event in microseconds
CLIENT	Remote client IP address
OP	R == read, W == write
OFFSET	File offset of I/O, in Kbytes
BYTES	TBytes of I/O
PATHNAME	Pathname of file, if known

Note - The output may be shuffled slightly on multi-CPU servers due to DTrace per-CPU buffering; post sort the TIME column if needed.

nfsv3ops.d

This DTrace script counts NFSv3 operations by client, printing a summary every 5 seconds:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... Interval 5 secs.\n");
}

nfsv3:::op-*
{
        @ops[args[0]->ci_remote, probename] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n   %-32s %-28s %8s\n", "Client", "Operation", "Count");
        printa("   %-32s %-28s %@8d\n", @ops);
        trunc(@ops);
}
```

The following output shows which client is sending which NFSv3 operations:

```
# ./nfsv3ops.d
Tracing... Interval 5 secs.

   Client                           Operation                          Count
```

```
192.168.17.75                   op-commit-done               1
192.168.17.75                   op-commit-start              1
192.168.17.75                   op-create-done               1
192.168.17.75                   op-create-start              1
192.168.17.75                   op-access-done               6
192.168.17.75                   op-access-start              6
192.168.17.75                   op-read-done                 6
192.168.17.75                   op-read-start                6
192.168.17.75                   op-write-done                7
192.168.17.75                   op-write-start               7
192.168.17.75                   op-lookup-done               8
192.168.17.75                   op-lookup-start              8
192.168.17.75                   op-getattr-done             10
192.168.17.75                   op-getattr-start            10

Client                          Operation                Count

Client                          Operation                Count
192.168.17.75                   op-getattr-done              1
192.168.17.75                   op-getattr-start             1
```

The following table list the fields that are printed by the `nfsv3ops.d` script.

Field	Description
Client	Remote client IP address
Operation	NFSv3 operation, described using the `nfsv3` provider probename
Count	Operations during this interval

nfsv3fileio.d

This DTrace script prints a summary of file read and write bytes:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... Hit Ctrl-C to end.\n");
}

nfsv3:::op-read-done
{
        @readbytes[args[1]->noi_curpath] = sum(args[2]->res_u.ok.data.data_len);
}


nfsv3:::op-write-done
```

```
{
        @writebytes[args[1]->noi_curpath] = sum(args[2]->res_u.ok.count);
}

dtrace:::END
{
        printf("\n%12s %12s  %s\n", "Rbytes", "Wbytes", "Pathname");
        printa("%@12d %@12d  %s\n", @readbytes, @writebytes);
}
```

This output shows a few files were read, and one was written:

```
# ./nfsv3fileio.d
Tracing... Hit Ctrl-C to end.
^C

      Rbytes       Wbytes  Pathname
           0       206663  /export/stuff/words10
        8624            0  /export/stuff/bin/echo-client-2
       13228            0  /export/stuff/bin/echo
      496292            0  /export/stuff/bin/ecpg
```

The following table describes the fields that are printed by the `nfsv3fileio.d` script.

Field	Description
Rbytes	Bytes read for this pathname
Wbytes	Bytes written to this pathname
Pathname	Pathname of NFS file

nfsv3rwtime.d

This DTrace script prints a summary NFSv3 read and write elapsed times , along with other details.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

inline int TOP_FILES = 10;

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

nfsv3:::op-read-start,
nfsv3:::op-write-start
```

```
{
        start[args[1]->noi_xid] = timestamp;
}

nfsv3:::op-read-done,
nfsv3:::op-write-done
/start[args[1]->noi_xid] != 0/
{
        this->elapsed = timestamp - start[args[1]->noi_xid];
        @rw[probename == "op-read-done" ? "read" : "write"] =
            quantize(this->elapsed / 1000);
        @host[args[0]->ci_remote] = sum(this->elapsed);
        @file[args[1]->noi_curpath] = sum(this->elapsed);
        start[args[1]->noi_xid] = 0;
}

dtrace:::END
{
        printf("NFSv3 read/write distributions (us):\n");
        printa(@rw);

        printf("\nNFSv3 read/write by host (total us):\n");
        normalize(@host, 1000);
        printa(@host);

        printf("\nNFSv3 read/write top %d files (total us):\n", TOP_FILES);
        normalize(@file, 1000);
        trunc(@file, TOP_FILES);
        printa(@file);
}
```

This output below shows a clear peak in the read time distribution plot in the 64 to 127 microsecond range, and a second smaller peak between 4 and 16 milliseconds:

```
# ./nfsv3rwtime.d
Tracing... Hit Ctrl-C to end.
^C
NFSv3 read/write distributions (us):

  write
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              64 |                                         0

  read
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@                                        1
              32 |@                                        1
              64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         36
             128 |@                                        1
```

```
            256 |@                                                  1
            512 |                                                   0
           1024 |                                                   0
           2048 |@                                                  1
           4096 |@@@                                                3
           8192 |@@@                                                4
          16384 |                                                   0


NFSv3 read/write by host (total us):

  192.168.17.75                                            81674

NFSv4 read/write top 10 files (total us):

  /export/stuff/motd                                          63
  /export/stuff/bin/daps                                    5876
  /export/stuff/bin/date                                    8150
  /export/stuff/bin/dasher                                 67584
```

Other details are printed, such as total read/write latency by host, and the top 10 files by latency.

The next example also shows a pair of peaks, the first around a fraction of a millisecond, the second at around 4 milliseconds:

```
# ./nfsv3rwtime.d
Tracing... Hit Ctrl-C to end.
^C
NFSv3 read/write distributions (us):

  read
          value  ------------- Distribution ------------- count
              8 |                                          0
             16 |@                                         4
             32 |@                                         5
             64 |@@@@@@                                    22
            128 |@@@@                                      13
            256 |@@@@@@@@                                  30
            512 |@@                                        7
           1024 |@                                         3
           2048 |@@@                                       12
           4096 |@@@@@@@                                   26
           8192 |@@@@                                      15
          16384 |@                                         2
          32768 |                                          0


NFSv3 read/write by host (total us):

  192.168.17.75                                           414458

NFSv3 read/write top 10 files (total us):
```

```
/export/stuff/bin/cal                                          11225
/export/stuff/bin/cjpeg                                        11947
/export/stuff/bin/charmap                                      12347
/export/stuff/bin/cdda2wav.bin                                 13449
/export/stuff/bin/chkey                                        13963
/export/stuff/bin/cputrack                                     14533
/export/stuff/bin/catman                                       15535
/export/stuff/bin/csslint-0.6                                  18302
/export/stuff/bin/col                                          19926
/export/stuff/bin/cdrecord.bin                                 40622
```

The first peak is likely to be NFS operations hitting the memory cache, and the second those that missed and went to disk. Further use of DTrace can confirm this theory.

The fields from the distribution plot are:

Field	Description
value	Minimum elapsed time for this event in microseconds
count	Number of events in this time range

nfsv3io.d

This is a simple DTrace script to provide basic I/O details by host every 5 seconds:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        interval = 5;
        printf("Tracing... Interval %d secs.\n", interval);
        tick = interval;
}

nfsv3:::op-*
{
        @ops[args[0]->ci_remote] = count();
}

nfsv3:::op-read-done
{
        @reads[args[0]->ci_remote] = count();
        @readbytes[args[0]->ci_remote] = sum(args[2]->res_u.ok.data.data_len);
}
```

```
nfsv3:::op-write-done
{
        @writes[args[0]->ci_remote] = count();
        @writebytes[args[0]->ci_remote] = sum(args[2]->res_u.ok.count);
}

profile:::tick-1sec
/tick-- == 0/
{
        normalize(@ops, interval);
        normalize(@reads, interval);
        normalize(@writes, interval);
        normalize(@writebytes, 1024 * interval);
        normalize(@readbytes, 1024 * interval);
        printf("\n   %-32s %6s %6s %6s %6s %8s\n", "Client", "r/s", "w/s",
            "kr/s", "kw/s", "ops/s");
        printa("   %-32s %@6d %@6d %@6d %@6d %@8d\n", @reads, @writes,
            @readbytes, @writebytes, @ops);
        trunc(@ops);
        trunc(@reads);
        trunc(@writes);
        trunc(@readbytes);
        trunc(@writebytes);
        tick = interval;
}
```

This output shows 192.168.17.75 calling NFSv3 reads and writes:

```
# ./nfsv3io.d
Tracing... Interval 5 secs.

   Client                              r/s    w/s   kr/s   kw/s    ops/s
   192.168.17.75                        27      1    686     40      100

   Client                              r/s    w/s   kr/s   kw/s    ops/s
   192.168.17.75                         0      0      0      0        8

   Client                              r/s    w/s   kr/s   kw/s    ops/s
   0.0.0.0                               0      0      0      0        0
   192.168.17.75                         2      0     28      0       18
^C
```

Other details can be calculated from the output, such as average read and write size, for example, 686(kr/s) / 27(r/s) = 25.4 average kr. These could also be added to the script to be printed as columns.

The fields printed are:

Field	Description
Client	Remote client IP address

Field	Description
r/s	reads per second
w/s	writes per second
kr/s	kilobytes read per second
kw/s	kilobytes written per second
ops/s	Total NFSv3 operations per second (including the reads and writes)

`nfsv4` Provider

The nfsv4 provider provides probes for tracing NFS v4 server activity.

Arguments

All NFS operation probes have the first argument in common:

```
args[0]  conninfo_t *  socket connection information
```

The conninfo_t structure is already used by the iSCSI target provider (iscsi), and is intended for use by all provider which are providing some higher level protocol, such as iscsi, nfs, http, and ftp.

```
typedef struct conninfo {
 string ci_local;       /* local host address */
 string ci_remote;      /* remote host address */
 string ci_protocol;    /* protocol (ipv4, ipv6, and so on) */
} conninfo_t;
```

Operation probes have their second argument in common:

```
args[1]  nfsv4opinfo_t *  NFS v4 operation properties
```

```
typedef struct nfsv4opinfo {
 string noi_curpath; /* current file handle path (if any) */
 cred_t *noi_cred; /* credentials */
 uint64_t noi_xid; /* transaction ID */
} nfsv4opinfo_t;
```

Callback operation probes have their second argument in common:

```
args[1]  nfsv4cbinfo_t *  NFS v4 callback properties
```

```
typedef struct nfsv4cbinfo {
 string nci_curpath; /* file handle path (if any) */
} nfsv4cbinfo_t;
```

Probes

The following table lists the top level operation probes along with the specific argument for each whose type is defined by the NFS v4 specification.

Probe Name	args[2]
nfsv4:::compound-op-start	COMPOUND4args *
nfsv4:::compound-op-done	COMPOUND4res *

The following table lists the operation probes along with the specific argument for each whose type is defined by the NFS v4 specification.

Probe Name	args[2]
nfsv4:::op-access-start	ACCESS4args *
nfsv4:::op-access-done	ACCESS4res *
nfsv4:::op-close-start	CLOSE4args *
nfsv4:::op-close-done	CLOSE4res *
nfsv4:::op-commit-start	COMMIT4args *
nfsv4:::op-commit-done	COMMIT4res *
nfsv4:::op-create-start	CREATE4args *
nfsv4:::op-create-done	CREATE4res *
nfsv4:::op-delegpurge-start	DELEGPURGE4args *
nfsv4:::op-delegpurge-done	DELEGPURGE4res *
nfsv4:::op-delegreturn-start	DELEGRETURN4args *
nfsv4:::op-delegreturn-done	DELEGRETURN4res *
nfsv4:::op-getattr-start	GETATTR4args *
nfsv4:::op-getattr-done	GETATTR4res *
nfsv4:::op-getfh-start	GETFH4args *
nfsv4:::op-getfh-done	GETFH4res *
nfsv4:::op-link-start	LINK4args *
nfsv4:::op-link-done	LINK4res *
nfsv4:::op-lock-start	LOCK4args *
nfsv4:::op-lock-done	LOCK4res *
nfsv4:::op-lockt-start	LOCKT4args *
nfsv4:::op-lockt-done	LOCKT4res *
nfsv4:::op-locku-start	LOCKU4args *
nfsv4:::op-locku-done	LOCKU4res *
nfsv4:::op-lookup-start	LOOKUP4args *

Probe Name	args[2]
nfsv4:::op-lookup-done	LOOKUP4res *
nfsv4:::op-lookupp-start	LOOKUPP4args *
nfsv4:::op-lookupp-done	LOOKUPP4res *
nfsv4:::op-nverify-start	NVERIFY4args *
nfsv4:::op-nverify-done	NVERIFY4res *
nfsv4:::op-open-start	OPEN4args *
nfsv4:::op-open-done	OPEN4res *
nfsv4:::op-open-confirm-start	OPEN_CONFIRM4args *
nfsv4:::op-open-confirm-done	OPEN_CONFIRM4res *
nfsv4:::op-open-downgrade-start	OPEN_DOWNGRADE4args *
nfsv4:::op-open-downgrade-done	OPEN_DOWNGRADE4args *
nfsv4:::op-openattr-start	OPENATTR4args *
nfsv4:::op-openattr-done	OPENATTR4res *
nfsv4:::op-putfh-start	PUTFH4args *
nfsv4:::op-putfh-done	PUTFH4res *
nfsv4:::op-putpubfh-start	PUTPUBFH4args *
nfsv4:::op-putpubfh-done	PUTPUBFH4res *
nfsv4:::op-putrootfh-start	PUTROOTFH4args *
nfsv4:::op-putrootfh-done	PUTROOTFH4res *
nfsv4:::op-read-start	READ4args *
nfsv4:::op-read-done	READ4res *
nfsv4:::op-readdir-start	READDIR4args *
nfsv4:::op-readdir-done	READDIR4res *
nfsv4:::op-readlink-start	READLINK4args *
nfsv4:::op-readlink-done	READLINK4res *
nfsv4:::op-release-lockowner-start	RELEASE_LOCKOWNER4args *
nfsv4:::op-release-lockowner-done	RELEASE_LOCKOWNER4res *
nfsv4:::op-remove-start	REMOVE4args *
nfsv4:::op-remove-don	REMOVE4res *
nfsv4:::op-rename-start	RENAME4args *
nfsv4:::op-rename-done	RENAME4res *
nfsv4:::op-renew-start	RENEW4args *
nfsv4:::op-renew-done	RENEW4res *
nfsv4:::op-restorefh-start	<none>
nfsv4:::op-restorefh-done	<none>
nfsv4:::op-savefh-start	SAVEFH4args *
nfsv4:::op-savefh-done	SAVEFH4res *
nfsv4:::op-secinfo-start	SECINFO4args *

Probe Name	args[2]
nfsv4:::op-secinfo-done	SECINFO4res *
nfsv4:::op-setattr-start	SETATTR4args *
nfsv4:::op-setattr-done	SETATTR4res *
nfsv4:::op-setclientid-start	SETCLIENTID4args *
nfsv4:::op-setclientid-done	SETCLIENTID4res *
nfsv4:::op-setclientid-confirm-start	SETCLIENTID_CONFIRM4args *
nfsv4:::op-setclientid-confirm-done	SETCLIENTID_CONFIRM4res *
nfsv4:::op-verify-start	VERIFY4args *
nfsv4:::op-verify-done	VERIFY4res *
nfsv4:::op-write-start	WRITE4args *
nfsv4:::op-write-done	WRITE4res *

Callback compound probes have an undefined second argument; this slot is reserved for future use.

The following table lists the top level callback probes along with the specific argument for each whose type is defined by the NFS v4 specification.

Probe Name	args[2]
nfsv4:::compound-cb-start	CB_COMPOUND4args *
nfsv4:::compound-cb-done	CB_COMPOUND4res *

The following table lists the callback probes along with the specific argument for each whose type is defined by the NFS v4 specification:

Probe Name	args[2]
nfsv4:::cb-getattr-start	CB_GETATTR4args*
nfsv4:::cb-getattr-done	CB_GETATTR4res *
nfsv4:::cb-recall-start	CB_RECALL4args *
nfsv4:::cb-recall-done	CB_RECALL4res *

Note - Since the Oracle Solaris NFS v4 implementation does not yet use the 'getattr' callback, the probe will not be implemented; it is noted here in anticipation of a future implementation.

Examples

Examples of nfsv4 provider usage are as follows.

nfsv4rwsnoop.d

This DTrace script traces NFSv4 reads and writes:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-8s %6s %s\n", "TIME(us)",
            "CLIENT", "OP", "OFFSET", "BYTES", "PATHNAME");
}

nfsv4:::op-read-start
{
        printf("%-16d %-18s %2s %-8d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "R", args[2]->offset / 1024, args[2]->count,
            args[1]->noi_curpath);
}

nfsv4:::op-write-start
{
        printf("%-16d %-18s %2s %-8d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "W", args[2]->offset / 1024, args[2]->data_len,
            args[1]->noi_curpath);
}
```

This output shows a few files were read, and one was written:

```
# ./nfsv4rwsnoop.d
TIME(us)          CLIENT             OP OFFSET    BYTES PATHNAME
156889725960      192.168.1.109      R 0           4096 /export/share/bin/nawk
156889735515      192.168.1.109      R 4          28672 /export/share/bin/nawk
156889736298      192.168.1.109      R 32         32768 /export/share/bin/nawk
156889736544      192.168.1.109      R 96         32768 /export/share/bin/nawk
156889736902      192.168.1.109      R 64         32768 /export/share/bin/nawk
156916061653      192.168.1.109      R 0           4096 /export/share/bin/ssh
156916069375      192.168.1.109      R 4          28672 /export/share/bin/ssh
156916070098      192.168.1.109      R 32         32768 /export/share/bin/ssh
156916070435      192.168.1.109      R 96         32768 /export/share/bin/ssh
156916070758      192.168.1.109      R 64         32768 /export/share/bin/ssh
156916071036      192.168.1.109      R 128        32768 /export/share/bin/ssh
156916071352      192.168.1.109      R 160        32768 /export/share/bin/ssh
```

```
156916071582    192.168.1.109       R 192      32768 /export/share/bin/ssh
156916071696    192.168.1.109       R 72        4096 /export/share/bin/ssh
156916080508    192.168.1.109       R 224       4096 /export/share/bin/ssh
156916080844    192.168.1.109       R 228      28672 /export/share/bin/ssh
156916081566    192.168.1.109       R 256      32768 /export/share/bin/ssh
156916081833    192.168.1.109       R 288      32768 /export/share/bin/ssh
156916082237    192.168.1.109       R 320      20480 /export/share/bin/ssh
156933373074    192.168.1.109       W 0        32768 /export/share/words
156933373351    192.168.1.109       W 32       32768 /export/share/words
156933373855    192.168.1.109       W 64       32768 /export/share/words
156933374185    192.168.1.109       W 96       32768 /export/share/words
156933375442    192.168.1.109       W 128      32768 /export/share/words
156933375864    192.168.1.109       W 160      32768 /export/share/words
156933376105    192.168.1.109       W 192      10055 /export/share/words
```

The fields printed are:

Field	Description
TIME(us)	Time of event in microseconds
CLIENT	Remote client IP address
OP	R == read, W == write
OFFSET	File offset of I/O, in Kbytes
BYTES	Bytes of I/O
PATHNAME	Pathname of file, if known

nfsv4ops.d

This DTrace script counts NFSv4 operations by client, printing a summary every 5 seconds:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... Interval 5 secs.\n");
}

nfsv4:::op-*
{
        @ops[args[0]->ci_remote, probename] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n   %-32s %-28s %8s\n", "Client", "Operation", "Count");
```

```
        printa("   %-32s %-28s %@8d\n", @ops);
        trunc(@ops);
}
```

The following output shows which client is sending which NFSv4 operations:

```
# ./nfsv4ops.d
Tracing... Interval 5 secs.

  Client                        Operation                   Count
  192.168.1.109                 op-getattr-done                 1
  192.168.1.109                 op-getattr-start                1
  192.168.1.109                 op-putfh-done                   1
  192.168.1.109                 op-putfh-start                  1

  Client                        Operation                   Count
  192.168.1.109                 op-access-done                  1
  192.168.1.109                 op-access-start                 1
  192.168.1.109                 op-close-done                   1
  192.168.1.109                 op-close-start                  1
  192.168.1.109                 op-getfh-done                   1
  192.168.1.109                 op-getfh-start                  1
  192.168.1.109                 op-open-done                    1
  192.168.1.109                 op-open-start                   1
  192.168.1.109                 op-getattr-done                 3
  192.168.1.109                 op-getattr-start                3
  192.168.1.109                 op-read-done                    9
  192.168.1.109                 op-read-start                   9
  192.168.1.109                 op-putfh-done                  12
  192.168.1.109                 op-putfh-start                 12
^C

  Client                        Operation                   Count
```

The fields printed are:

Field	Description
Client	Remote client IP address
Operation	NFSv4 operation, described using the nfsv4 provider probename
Count	Operations during this interval

nfsv4fileio.d

This DTrace script prints a summary of file read and write bytes:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
```

```
dtrace:::BEGIN
{
        trace("Tracing... Hit Ctrl-C to end.\n");
}

nfsv4:::op-read-done
{
        @readbytes[args[1]->noi_curpath] = sum(args[2]->data_len);
}


nfsv4:::op-write-done
{
        @writebytes[args[1]->noi_curpath] = sum(args[2]->count);
}

dtrace:::END
{
        printf("\n%12s %12s  %s\n", "Rbytes", "Wbytes", "Pathname");
        printa("%@12d %@12d  %s\n", @readbytes, @writebytes);
}
```

This output shows a few files were read, and one was written:

```
# ./nfsv4fileio.d
Tracing... Hit Ctrl-C to end.
^C

      Rbytes      Wbytes  Pathname
           0      206663  /export/share/words1
       24528           0  /export/share/bin/xargs
       44864           0  /export/share/bin/ed
      232476           0  /export/share/bin/vi
```

The fields printed are:

Field	Description
Rbytes	Bytes read for this pathname
Wbytes	Bytes written to this pathname
Pathname	Pathname of NFS file

nfsv4rwtime.d

This DTrace script prints a summary NFSv4 read and write elapsed times, along with other details:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
```

```
inline int TOP_FILES = 10;

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

nfsv4:::op-read-start,
nfsv4:::op-write-start
{
        start[args[1]->noi_xid] = timestamp;
}

nfsv4:::op-read-done,
nfsv4:::op-write-done
{
        this->elapsed = timestamp - start[args[1]->noi_xid];
        @rw[probename == "op-read-done" ? "read" : "write"] =
            quantize(this->elapsed / 1000);
        @host[args[0]->ci_remote] = sum(this->elapsed);
        @file[args[1]->noi_curpath] = sum(this->elapsed);
        start[args[1]->noi_xid] = 0;
}

dtrace:::END
{
        printf("NFSv4 read/write distributions (us):\n");
        printa(@rw);

        printf("\nNFSv4 read/write by host (total us):\n");
        normalize(@host, 1000);
        printa(@host);

        printf("\nNFSv4 read/write top %d files (total us):\n", TOP_FILES);
        normalize(@file, 1000);
        trunc(@file, TOP_FILES);
        printa(@file);
}
```

This output below shows a peak in the read time distribution plot in the 64 to 127 microsecond range, and a second peak between 2 and 8 milliseconds:

```
# ./nfsv4rwtime.d
Tracing... Hit Ctrl-C to end.
^C
NFSv4 read/write distributions (us):

  write
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |@@@@@@                                   1
```
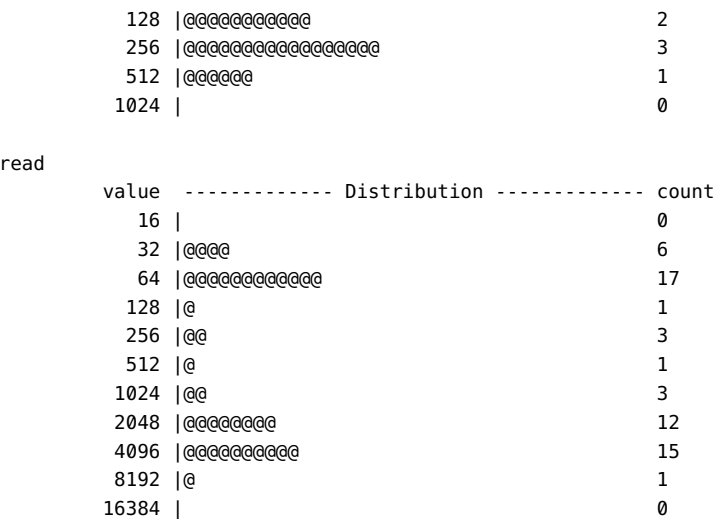
```
             128 |@@@@@@@@@@                              2
             256 |@@@@@@@@@@@@@@@@                        3
             512 |@@@@@@                                  1
            1024 |                                        0

    read
           value ------------- Distribution ------------- count
              16 |                                        0
              32 |@@@@                                    6
              64 |@@@@@@@@@@@@                            17
             128 |@                                       1
             256 |@@                                      3
             512 |@                                       1
            1024 |@@                                      3
            2048 |@@@@@@@@                                12
            4096 |@@@@@@@@@@                               15
            8192 |@                                       1
           16384 |                                        0


NFSv4 read/write by host (total us):

  192.168.1.109                                        148215

NFSv4 read/write top 10 files (total us):

  /export/share/bin/man                                  5020
  /export/share/bin/makeuuid                             5132
  /export/share/bin/mc68030                              5836
  /export/share/bin/m4                                   6446
  /export/share/bin/msgfmt                               6669
  /export/share/bin/mkmsgs                               6674
  /export/share/bin/mailstats                            6935
  /export/share/bin/mkdir                                7009
  /export/share/bin/mac                                  7693
  /export/share/bin/make                                27903
```

Other details are printed, such as total read/write latency by host, and the top 10 files by latency.

The first peak in the read distribution is likely to be NFS operations hitting the memory cache, and the second those that missed and read from disk. The writes were all fast as they are likely to written to the memory cache and returned asynchronously. Further use of DTrace can confirm these theories.

The fields from the distribution plot are:

Field	Description
value	Minimum elapsed time for this event in microseconds
count	Number of events in this time range

nfsv4io.d

This is a simple DTrace script to provide basic I/O details by host every 5 seconds:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        interval = 5;
        printf("Tracing... Interval %d secs.\n", interval);
        tick = interval;
}

nfsv4:::op-*
{
        @ops[args[0]->ci_remote] = count();
}

nfsv4:::op-read-done
{
        @reads[args[0]->ci_remote] = count();
        @readbytes[args[0]->ci_remote] = sum(args[2]->data_len);
}


nfsv4:::op-write-done
{
        @writes[args[0]->ci_remote] = count();
        @writebytes[args[0]->ci_remote] = sum(args[2]->count);
}

profile:::tick-1sec
/tick-- == 0/
{
        normalize(@ops, interval);
        normalize(@reads, interval);
        normalize(@writes, interval);
        normalize(@writebytes, 1024 * interval);
        normalize(@readbytes, 1024 * interval);
        printf("\n   %-32s %6s %6s %6s %6s %8s\n", "Client", "r/s", "w/s",
            "kr/s", "kw/s", "ops/s");
        printa("   %-32s %@6d %@6d %@6d %@6d %@8d\n", @reads, @writes,
            @readbytes, @writebytes, @ops);
        trunc(@ops);
        trunc(@reads);
        trunc(@writes);
        trunc(@readbytes);
        trunc(@writebytes);
        tick = interval;
```

```
}
```

This output shows `192.168.1.109` calling NFSv4 reads and writes:

```
# ./nfsv4io.d
Tracing... Interval 5 secs.

  Client                           r/s   w/s   kr/s  kw/s   ops/s
  192.168.1.109                     17    1    331    40     290

  Client                           r/s   w/s   kr/s  kw/s   ops/s
  192.168.1.109                      9    0    197     0     152

  Client                           r/s   w/s   kr/s  kw/s   ops/s
  192.168.1.109                     16    0    269     0     363

  Client                           r/s   w/s   kr/s  kw/s   ops/s
^C
```

Other details can be calculated from the output, such as average read and write size, for example 331(kr/s) / 17(r/s) = 19.5 average kr. These could also be added to the script to be printed as columns.

The fields printed are:

Field	Description
Client	Remote client IP address
r/s	Reads per second
w/s	Writes per second
kr/s	Kilobytes read per second
kw/s	Kilobytes written per second
ops/s	Total NFSv4 operations per second (including the reads and writes)

`srp` Provider

The `srp` provider provides probes for tracing `srp` port provider activity.

This is a kernel provider built into the COMSTAR srp target port provider.

Probes

Probes Overview

Header	Header
Service up/down	`srp:::service-up`, `srp:::service-down`
Remote Port login/logout	`srp:::login-command`, `srp:::login-response`, `srp:::logout-command`
SRP command/response	`srp:::task-command`, `srp:::task-response`
SCSI command/response	`srp:::scsi-command`, `srp:::scsi-response`
Data transfer	`srp:::xfer-start`, `srp:::xfer-done`

For all of the providers below, string fields which are not known contain the string <unknown>. Integer fields which are not known contain `0`.

Service up/down Event Probes

`srp:::service-up` and `srp:::service-down` trace SRP target online and offline events. Remote port information `ci_remote` is unavailable for both probes.

Probes	Variable	Type	Description

`srp:::service-up`
 	
`srp:::service-down`
	args[0]	conninfo_t *	Connection information

`srp:::service-up`
 	
`srp:::service-down`
	args[1]	srp_portinfo_t *	Local and remote port information

Remote Port Login/Logout Event Probes

Probes	Variable	Type	Description

`srp:::login-command`
 	
`srp:::login-response`
 	
`srp:::logout-command`
	args[0]	conninfo_t *	Connection information

`srp:::login-command`	args[1]	srp_portinfo_t *	Local and remote port information

Probes	Variable	Type	Description
srp:::login-response			
srp:::logout-command			
srp:::login-command	args[2]	srp_logininfo_t *	Login command/response information
srp:::login-response			

SRP Command Event Probes

Probes	Variable	Type	Description
srp:::task-command	args[0]	conninfo_t *	Connection information
srp:::task-response			
srp:::task-command	args[1]	srp_portinfo_t *	Local and remote port information
srp:::task-response			
srp:::scsi-response	args[2]	srp_taskinfo_t *	srp task information
srp:::scsi-command			

SCSI Command Event Probes

Probes	Variable	Type	Description
srp:::scsi-command	args[0]	conninfo_t *	Connection information
srp:::scsi-response			
srp:::scsi-command	args[1]	srp_portinfo_t *	Local and remote port information
srp:::scsi-response			
srp:::scsi-command	args[2]	scsicmd_t *	SCSI command block (cdb)
srp:::scsi-response	args[2]	srp_taskinfo_t *	srp task information
srp:::scsi-command	args[3]	srp_taskinfo_t *	srp task information

Data Transfer Probes

Probes	Variable	Type	Description
srp:::xfer-start	args[0]	conninfo_t *	Connection information

Probes	Variable	Type	Description
`srp:::xfer-done`			
`srp:::xfer-start`	`args[1]`	`fc_port_info_t *`	Local port information
`srp:::xfer-done`			
`srp:::xfer-start`	`args[2]`	`xferinfo_t *`	RDMA transfer information
`srp:::xfer-done`			
`srp:::xfer-start`	`args[3]`	`srp_taskinfo_t *`	srp task information
`srp:::xfer-done`			

Types

`scsicmd_t`, `conninfo_t` and `xferinfo_t` are common types which are used by other providers.

scsicmd_t

```
typedef struct scsicmd {
        uint64_t ic_len;        /* CDB length */
        uint8_t  *ic_cdb;       /* CDB data */
} scsicmd_t;
```

conninfo_t

```
typedef struct conninfo {
        string ci_local;        /* GID of the local HCA */
        string ci_remote;       /* GID of the remote HCA */
        string ci_protocol;     /* protocol ("ib") */
} conninfo_t;
```

srp_portinfo_t

```
typedef struct srp_portinfo {
        /* initiator */
        string  pi_initiator;   /* Initiator: eui.xxxxxxxxxxxxxxx */
        string  pi_i_sid;       /* Initiator seiion id */

        /* target */
        string  pi_target;      /* Target: eui.xxxxxxxxxxxxxxx */
        string  pi_t_sid;       /* Target session id */

        uintptr_t pi_chan_id;   /* Channel identifier */
} srp_portinfo_t;
```

srp_logininfo_t

```
typedef struct srp_logininfo {
        uint64_t li_task_tag;      /* SRP task tag */
        uint32_t li_max_it_iu_len; /* Maxium iu length that initiator can
        send to target */
        uint32_t li_max_ti_iu_len; /* Maxium iu length that target can
        send to initiator */
        uint32_t li_request_limit; /* Maximun number of SRP requests
        that initiator can send on a channel */
        uint32_t reason_code;      /* Reason code */
} srp_logininfo_t;
```

srp_taskinfo_t

```
typedef struct srp_taskinfo {
        uint64_t ti_task_tag;   /* SRP task tag */
        uint64_t ti_lun;        /* Target logical unit number */
        uint8_t  ti_function;   /* Task management function */
        uint32_t ti_req_limit_delta; /* Increment of channel's request limit */
        uint8_t  ti_flag;          /* bit 2:DOOVER 3:DOUNDER 4:DIOVER 5:DIUNDER */
        uint32_t ti_do_resid_cnt;   /* Data-out residual count */
        uint32_t ti_di_resid_cnt;   /* Data-in residual count */
        uint8_t  ti_status;     /* Status of this task */
} srp_taskinfo_t;
```

xferinfo_t

```
typedef struct xferinfo {
        uintptr_t xfer_laddr;       /* Local buffer address */
        uint32_t  xfer_loffset;     /* Relative offset from local buffer */
        uint32_t  xfer_lkey;        /* Access control to local memory */
        uintptr_t xfer_raddr;       /* Remote virtual address */
        uint32_t  xfer_roffset;     /* Offset from the remote address */
        uint32_t  xfer_rkey;        /* Access control to remote address */
        uint32_t  xfer_len;         /* Transfer length */
        uint8_t   xfer_type;        /* 0: read; 1: write; */
} xferinfo_t;
```

Examples

service.d

This is a script to produce a report of target online or offline events.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n\n");
        printf("%-14s   %-35s %-20s\n", "SRP EVENT",

\
"LOCAL PORT", "EUI NAME");
};

srp:::service-up
{
        printf("%-14s   %-35s %-20s\n", probename,

\
args[0]->ci_local, args[1]->pi_target);
}

srp:::service-down
{
        printf("%-14s   %-35s %-20s\n", probename,

\
args[0]->ci_local, args[1]->pi_target);
}
```

This output shows the host and the number of iSCSI operations:

```
thump1## dtrace -s ~/src/service.d
Tracing... Hit Ctrl-C to end.
^C
SRP EVENT       LOCAL PORT                           EUI NAME
service-down    fe80000000000000:0003ba0001004d31    eui.0003BA0001004D30
service-down    fe80000000000000:0003ba0001004d32    eui.0003BA0001004D30
service-up      fe80000000000000:0003ba0001004d31    eui.0003BA0001004D30
service-up      fe80000000000000:0003ba0001004d32    eui.0003BA0001004D30
thump1##
```

The fields are:

Field	Description
SRP EVENT	srp event type
LOCAL PORT	GID of the local port
EUI NAME	EUI name of the local port

srpwho.d

This is a simple script to produce a report of the remote HCA port and a count of srp events.
This is intended to provide a quick summary of srp activity when run on the SRP target server:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

srp:::login-command,
srp:::login-response,
srp:::task-command,
srp:::task-response,
srp:::scsi-command,
srp:::scsi-response,
srp:::xfer-start,
srp:::xfer-done
{
        @events[args[0]->ci_remote, probename] = count();
}

dtrace:::END
{
        printf("   %-33s %14s %8s\n", "REMOTE GID", "iSCSI EVENT", "COUNT");
        printa("   %-33s %14s %@8d\n", @events);
}
```

This output shows the host and the number of iSCSI operations:

```
thump1## dtrace -s ./srpwho.d
Tracing... Hit Ctrl-C to end.
^C
   REMOTE GID                        iSCSI EVENT    COUNT
   fe80000000000000:0003ba000100386d  login-command       1
   fe80000000000000:0003ba000100386d login-response       1
   fe80000000000000:0003ba0001003851  login-command       2
   fe80000000000000:0003ba0001003851 login-response       2
   fe80000000000000:0003ba0001003852  login-command       2
   fe80000000000000:0003ba0001003852 login-response       2
   fe80000000000000:0003ba0001004d32     xfer-done       9
   fe80000000000000:0003ba0001004d32     xfer-start       9
   fe80000000000000:0003ba0001004d31     xfer-done      18
   fe80000000000000:0003ba0001004d31     xfer-start      18
   fe80000000000000:0003ba0001004d32   scsi-command      22
   fe80000000000000:0003ba0001004d32   scsi-response      22
   fe80000000000000:0003ba0001004d32    task-command      22
```

```
fe80000000000000:0003ba0001004d32  task-response        22
fe80000000000000:0003ba0001004d31   scsi-command        42
fe80000000000000:0003ba0001004d31  scsi-response        42
fe80000000000000:0003ba0001004d31   task-command        42
fe80000000000000:0003ba0001004d31  task-response        42
```

The fields are:

Field	Description
REMOTE GID	GID of the client HCA port
SRP EVENT	srp event type
COUNT	Number of events traced

srpsnoop.d

This is a simple script to snoop srp events when run on a srp target server.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10

dtrace:::BEGIN
{
        printf("%17s %3s  %-40s %-14s %6s %10s  %6s\n", "TIMESTAMP",
            "CPU", "REMOTE GID", "EVENT", "BYTES", "TAG", "SCSIOP");

        /*
         * SCSI opcode to string translation hash. This is from
         * /usrp/include/sys/scsi/generic/commands.h. If you would
         * rather all hex, comment this out.
         */
        scsiop[0x08] = "read";
        scsiop[0x0a] = "write";
        scsiop[0x0b] = "seek";
        scsiop[0x28] = "read(10)";
        scsiop[0x2a] = "write(10)";
        scsiop[0x2b] = "seek(10)";
}

srp:::login-*
{
        printf("%17d %3d  %-40s %-14s %17d  -\n", timestamp, cpu,
        args[0]->ci_remote,
            probename, args[2]->li_task_tag);
}
srp:::task-command,
```

```
srp:::task-response,
srp:::scsi-response
{
        printf("%17d %3d  %-40s %-14s %6d %10d  -\n", timestamp, cpu,
    args[0]->ci_remote,
            probename, 0, args[2]->ti_task_tag);
}

srp:::scsi-command
/scsiop[args[2]->ic_cdb[0]] != NULL/
{
        printf("%17d %3d  %-40s %-14s %6d %10d  %s\n", timestamp, cpu,
     args[0]->ci_remote,
            probename, 0, args[3]->ti_task_tag, scsiop[args[2]->ic_cdb[0]]);
}

srp:::scsi-command
/scsiop[args[2]->ic_cdb[0]] == NULL/
{
        printf("%17d %3d  %-40s %-14s %6d %10d  0x%x\n", timestamp, cpu,
    args[0]->ci_remote,
            probename, 0, args[3]->ti_task_tag, args[2]->ic_cdb[0]);
}

srp:::xfer-start,
srp:::xfer-done
{
        printf("%17d %3d  %-40s %-14s %6d %10d   %s\n", timestamp,
    cpu,  args[0]->ci_remote,
            probename,args[2]->xfer_len, args[3]->ti_task_tag,
            args[2]->xfer_type > 0 ? "READ" : "WRITE");
}
```

This output shows the snoop on dd commands executed by the initiator.

```
thump1## dtrace -s ./srpsnoop.d
TIMESTAMP         CPU  REMOTE GID                              EVENT          BYTES   TAG
 SCSIOP
22644410404019   3  fe80000000000000:0003ba0001004d31    task-command       0   26  -
22644410493068   3  fe80000000000000:0003ba0001004d31    scsi-command       0   26  read
(10)
22644410511422   3  fe80000000000000:0003ba0001004d31    task-command       0   30  -
22644410541494   3  fe80000000000000:0003ba0001004d31    scsi-command       0   30  read
(10)
22644410621049   0  fe80000000000000:0003ba0001004d31    xfer-start      2048   26
 READ
22644410720486   1  fe80000000000000:0003ba0001004d31    xfer-start     49152   30
 READ
22644410681390   3  fe80000000000000:0003ba0001004d31    xfer-done       2048   26
 READ
22644410694719   3  fe80000000000000:0003ba0001004d31    scsi-response      0   26  -
22644410703358   3  fe80000000000000:0003ba0001004d31    task-response      0   26  -
```

```
22644410895424   3  fe80000000000000:0003ba0001004d31   xfer-done       49152   30
 READ
22644410901576   3  fe80000000000000:0003ba0001004d31   scsi-response       0   30  -
22644410905717   3  fe80000000000000:0003ba0001004d31   task-response       0   30  -
22727363721107   3  fe80000000000000:0003ba0001004d31   task-command        0   59  -
22727363919179   0  fe80000000000000:0003ba0001004d31   xfer-start      10240   59
 WRITE
22727364095164   0  fe80000000000000:0003ba0001004d31   scsi-response       0   59  -
22727364105406   0  fe80000000000000:0003ba0001004d31   task-response       0   59  -
22727363812953   3  fe80000000000000:0003ba0001004d31   scsi-command        0   59
 write(10)
22727363986185   3  fe80000000000000:0003ba0001004d31   xfer-done       10240   59
 WRITE
```

The fields are:

Field	Description
CPU	CPU event occurred on
REMOTE GID	GID of the client HCA port
EVENT	srp event type
BYTES	Data bytes
TAG	Initiator task tag
SCSIOP	SCSI opcode as a description, as hex, or '-'

`tcp` Provider

he `tcp` provider provides probes for tracing the TCP protocol.

Probes

The `tcp` probes are described in the table below.

TABLE 81   `tcp` Probes

Probe	Description
state-change	Fires a TCP session changes its TCP state. Previous state is noted in the `tcplsinfo_t` * probe argument. The `tcpinfo_t` * and `ipinfo_t` * arguments are NULL.
send	Fires whenever TCP sends a segment (either control or data).
receive	Fires whenever TCP receives a segment (either control or data).

Probe	Description
connect-request	Fires when a TCP active open is initiated by sending an initial SYN segment. The tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers associated with the initial SYN segment sent.
connect-established	Fires when either of the following occurs: either a TCP active OPEN succeeds - the initial SYN has been sent and a valid SYN,ACK segment has been received in response. TCP enters the ESTABLISHED state, and the tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers associated with the SYN,ACK segment received; or a simultaneous active OPEN succeeds and a final ACK is received from the peer TCP. TCP has entered the ESTABLISHED state and the tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers of the final ACK received. The common thread in these cases is that an active-OPEN connection is established at this point, in contrast with tcp:::accept-established which fires on passive connection establishment. In both cases above, the TCP segment that is presented via the tcpinfo_t * is the segment that triggers the transition to ESTABLISHED - the received SYN,ACK in the first case and the final ACK segment in the second.
connect-refused	A TCP active OPEN connection attempt was refused by the peer - a RST segment was received in acknowledgment of the initial SYN. The tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers associated with the RST,ACK segment received.
accept-established	A passive open has succeeded - an initial active OPEN initiation SYN has been received, TCP responded with a SYN,ACK and a final ACK has been received. TCP has entered the ESTABLISHED state. The tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers associated with the final ACK segment received.
accept-refused	An incoming SYN has arrived for a destination port with no listening connection, so the connection initiation request is rejected by sending a RST segment ACKing the SYN. The tcpinfo_t * and ipinfo_t * probe arguments represent the TCP and IP headers associated with the RST segment sent.

The send and receive probes trace packets on physical interfaces and also packets on loopback interfaces that are processed by tcp. On Oracle Solaris, loopback TCP connections can bypass the TCP layer when transferring data packets - this is a performance feature called *tcp fusion*; these packets are also traced by the tcp provider.

Arguments

The argument types for the tcp probes are listed in the table below. The arguments are described in the following section. All probes expect state-change have five arguments, state-change has six arguments..

Probe	args[0]	args[1]	args[2]	args[3]	args[4]	args[5]
state-change	null	csinfo_t *	null	tcpsinfo_t *	null	tcplsinfo_t *

Probe	args[0]	args[1]	args[2]	args[3]	args[4]	args[5]
send	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
receive	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
connect-request	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
connect-established	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
connect-refused	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
accept-established	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
accept-refused	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	

`pktinfo_t` Structure

The pktinfo_t structure is where packet ID info can be made available for deeper analysis if packet IDs become supported by the kernel in the future.

The pkt_addr member is currently always NULL.

```
typedef struct pktinfo {
        uintptr_t pkt_addr;                /* currently always NULL */
} pktinfo_t;
```

`csinfo_t` Structure

The csinfo_t structure is where connection state info is made available. It contains a unique system wide connection ID, and the process ID and zone ID associated with the connection.

```
typedef struct csinfo {
        uintptr_t cs_addr;
 uint64_t cs_cid;
 pid_t cs_pid;
 zoneid_t cs_zoneid;
 } csinfo_t;
```

cs_addr	Address of translated ip_xmit_attr_t *.
cs_cid	Connection id. A unique per-connection identifier which identifies the connection during its lifetime.
cs_pid	Process ID associated with the connection.

cs_zoneid	Zone ID associated with the connection.

`ipinfo_t` Structure

The `ipinfo_t` structure contains common IP info for both IPv4 and IPv6.

```
typedef struct ipinfo {
        uint8_t ip_ver;                 /* IP version (4, 6) */
        uint16_t ip_plength;            /* payload length */
        string ip_saddr;                /* source address */
        string ip_daddr;                /* destination address */
} ipinfo_t;
```

These values are read at the time the probe fired in TCP, and so `ip_plength` is the expected IP payload length. However, the IP layer may add headers such as AH and ESP, which will increase the actual payload length. To examine this, also trace packets using the `ip` provider.

TABLE 82   `ipinfo_t` Members

ip_ver	IP version number. Currently either 4 or 6.
ip_plength	Payload length in bytes. This is the length of the packet at the time of tracing, excluding the IP header.
ip_saddr	Source IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.
ip_daddr	Destination IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.

`tcpsinfo_t` Structure

The `tcpsinfo_t` structure contains `tcp` state info.

```
typedef struct tcpsinfo {
        uintptr tcps_addr;
        int tcps_local;        /* is delivered locally, boolean */
        int tcps_active;        /* active open (from here), boolean */
        uint16_t tcps_lport;      /* local port */
        uint16_t tcps_rport;      /* remote port */
string tcps_laddr;  /* local address, as a string */
string tcps_raddr;  /* remote address, as a string */
int32_t tcps_state;/* TCP state. Use inline tcp_state_string[]to convert to string */
        uint32_t tcps_iss;     /* initial sequence # sent */
        uint32_t tcps_suna;     /* sequence # sent but unacked */
        uint32_t tcps_snxt;     /* next sequence # to send */
        uint32_t tcps_rack;     /* sequence # acked */
        uint32_t tcps_rnxt;     /* next sequence # expected */
        uint32_t tcps_swnd;     /* send window size */
        uint32_t tcps_snd_ws;   /* send window scaling */
```

```
       uint32_t tcps_rwnd;     /* receive window size */
       uint32_t tcps_rcv_ws;   /* receive window scaling */
 uint32_t tcps_cwnd;  /* congestion window */
 uint32_t tcps_cwnd_ssthresh; /* threshold for congestion avoidance */
 uint32_t tcps_sack_fack; /* SACK sequence # acked */
 uint32_t tcps_sack_snxt; /* next SACK seq # for retransmission */
       uint32_t tcps_rto;              /* round-trip timeout, msec */
 uint32_t tcps_mss;  /* max segment size */
       int tcps_retransmit;            /* retransmit send event, boolean */
} tcpsinfo_t;
```

It may seem redundant to supply the local and remote ports and addresses here as well as in the
`tcpinfo_t` below, but the `tcp:::state-change` probes do not have associated `tcpinfo_t` data,
so in order to map the state change to a specific port, you require this data here.

TABLE 83   `tcpsinfo_t` Members

tcps_addr	Address of translated `tcp_t *`.
tcps_local	Local, boolean. 0: is not delivered locally and uses a physical network interface, 1: is delivered locally including loopback interfaces, such as `lo0`.
tcps_active	Active open, boolean. 0: TCP connection was created from a remote host, 1: TCP connection was created from this host.
tcps_lport	Local port associated with the TCP connection.
tcps_rport	Remote port associated with the TCP connection.
tcps_laddr	Local address associated with the TCP connection, as a string.
tcps_raddr	Remote address associated with the TCP connection, as a string.
tcps_state	TCP state. Inline definitions are provided for the various TCP states: `TCP_STATE_CLOSED`, `TCP_STATE_SYN_SENT`, and so on. Use inline `tcp_state_string[]` to convert state to a string.
tcps_iss	Initial sequence number sent.
tcps_suna	Lowest sequence number for which you have sent data but not received acknowledgement.
tcps_snxt	Next sequence number to send. `tcps_snxt - tcps_suna` gives the number of bytes pending acknowledgement for the TCP connection.
tcps_rack	Highest sequence number for which you have received and sent acknowledgement.
tcps_rnxt	Next sequence number expected on receive side. `tcps_rnxt - tcps_rack` gives the number of bytes you have received but not yet acknowledged for the TCP connection.
tcps_swnd	TCP send window size.
tcps_snd_ws	TCP send window scale. `tcps_swnd << tcp_snd_ws` gives the scaled window size if window scaling options are in use.
tcps_rwnd	TCP receive window size.
tcps_rcv_ws	TCP receive window scale. `tcps_rwnd << tcp_rcv_ws` gives the scaled window size if window scaling options are in use.
tcps_cwnd	TCP congestion window size. `tcps_cwnd_ssthresh` TCP congestion window threshold. When the congestion window is greater than `ssthresh`, congestion avoidance begins.

tcps_cwnd_ssthresh	TCP congestion window threshold. When the congestion window is greater than ssthresh, congestion avoidance begins.
tcps_sack_fack	Highest SACK-acked sequence number.
tcps_sack_snxt	Next sequence num to be retransmitted using SACK.
tcps_rto	Round-trip timeout. If you do not receive acknowledgement of data sent tcps_rto msec ago, retransmit is required.
tcps_mss	Maximum segment size.
tcps_retransmit	Send is a retransmit, boolean. 1 for tcp:::send events that are retransmissions, 0 for tcp events that are not send events, and for send events that are not retransmissions.

tcplsinfo_t Structure

The tcplsinfo_t structure contains the previous tcp state during a state change.

```
typedef struct tcplsinfo {
        int32_t tcps_state;              /* TCP state */
} tcplsinfo_t;
```

TABLE 84   tcplsinfo_t Members

tcps_state	Previous TCP state. Inline definitions are provided for the various TCP states: TCP_STATE_CLOSED, TCP_STATE_SYN_SENT, and so on. Use inline tcp_state_string[] to convert state to a string.

tcpinfo_t Structure

The tcpinfo_t structure is a DTrace translated version of the TCP header.

```
typedef struct tcpinfo {
        uint16_t tcp_sport;              /* source port */
        uint16_t tcp_dport;              /* destination port */
        uint32_t tcp_seq;                /* sequence number */
        uint32_t tcp_ack;                /* acknowledgment number */
        uint8_t tcp_offset;              /* data offset, in bytes */
        uint8_t tcp_flags;               /* flags */
        uint16_t tcp_window;             /* window size */
        uint16_t tcp_checksum;           /* checksum */
        uint16_t tcp_urgent;             /* urgent data pointer */
        tcph_t *tcp_hdr;                 /* raw TCP header */
} tcpinfo_t;
```

TABLE 85   tcpinfo_t Members

tcp_sport	TCP source port.

`tcp_dport`	TCP destination port.
`tcp_seq`	TCP sequence number.
`tcp_ack`	TCP acknowledgment number.
`tcp_offset`	Payload data offset, in bytes not 32-bit words.
`tcp_flags`	TCP flags. See the `tcp_flags` table below for available macros.
`tcp_window`	TCP window size, bytes.
`tcp_checksum`	Checksum of TCP header and payload.
`tcp_urgent`	TCP urgent data pointer, bytes.
`tcp_hdr`	Pointer to raw TCP header at time of tracing.

TABLE 86   `tcp_flags` Values

`TH_FIN`	No more data from sender (finish).
`TH_SYN`	Synchronize sequence numbers (connect).
`TH_RST`	Reset the connection.
`TH_PUSH`	TCP push function.
`TH_ACK`	Acknowledgment field is set.
`TH_URG`	Urgent pointer field is set.
`TH_ECE`	Explicit congestion notification echo. For more information, see RFC-3168.
`TH_CWR`	Congestion window reduction.

See RFC-793 for a detailed explanation of the standard TCP header fields and flags.

Examples

Some simple examples of `tcp` provider usage follow.

Connections by Host Address

This DTrace one-liner counts inbound TCP connections by source IP address:

```
# dtrace -n 'tcp:::accept-established { @[args[3]->tcps_raddr] = count(); }'
dtrace: description 'tcp:::state-change' matched 1 probes
^C

  127.0.0.1                                                   1
  192.168.2.88                                                1
  fe80::214:4fff:fe8d:59aa                                    1
  192.168.1.109                                               3
```

The output above shows there were 3 TCP connections from `192.168.1.109`, a single TCP connection from the IPv6 host `fe80::214:4fff:fe8d:59aa`.

Connections by TCP Port

This DTrace one-liner counts inbound TCP connections by local TCP port:

```
# dtrace -n 'tcp:::accept-established { @[args[3]->tcps_lport] = count(); }'
dtrace: description 'tcp:::state-change' matched 1 probes
^C

 40648              1
    22              3
```

The output above shows there were 3 TCP connections for port 22 `ssh`, a single TCP connection for port 40648 (an RPC port).

Who is Connecting to What

Combining the previous two examples produces a useful one liner, to quickly identify who is connecting to what:

```
# dtrace -n
'tcp:::accept-established { @[args[3]->tcps_raddr, args[3]->tcps_lport] = count(); }'
dtrace: description 'tcp:::state-change' matched 1 probes
^C

  192.168.2.88                                    40648              1
  fe80::214:4fff:fe8d:59aa                           22              1
  192.168.1.109                                      22              3
```

The output above shows there were 3 TCP connections from `192.168.1.109` to port 22 (`ssh`).

Who is not Connecting to What

It may be useful when troubleshooting connection issues to see who is failing to connect to their requested ports. This is equivalent to seeing where incoming SYNs arrive when no listener is present, as per RFC793:

```
# dtrace -n 'tcp:::accept-refused
{ @[args[2]->ip_daddr, args[4]->tcp_sport] = count(); }'
dtrace: description 'tcp:::receive ' matched 1 probes
^C

  192.168.1.109                                      23              2
```

Here you traced two failed attempts by host `192.168.1.109` to connect to port 23 (`telnet`).

Packets by Host Address

This DTrace one-liner counts TCP received packets by host address:

```
# dtrace -n 'tcp:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace: description 'tcp:::receive ' matched 5 probes
^C

  127.0.0.1                                                       7
  fe80::214:4fff:fe8d:59aa                                       14
  192.168.2.30                                                   43
  192.168.1.109                                                  44
  192.168.2.88                                                 3722
```

The output above shows that 7 TCP packets were received from `127.0.0.1`, 14 TCP packets from the IPv6 host `fe80::214:4fff:fe8d:59aa`.

Packets by Local Port

This DTrace one-liner counts TCP received packets by the local TCP port:

```
# dtrace -n 'tcp:::receive { @[args[4]->tcp_dport] = count(); }'
dtrace: description 'tcp:::receive ' matched 5 probes
^C

 42303               3
 42634               3
  2049              27
 40648              36
    22             162
```

The output above shows that 162 packets were received for port 22 (`ssh`), 36 packets were received for port 40648 (an RPC port), 27 packets for 2049 (NFS), and a few packets to high numbered client ports.

Sent Size Distribution

This DTrace one-liner prints distribution plots of IP payload size by destination, for TCP sends:

```
# dtrace -n 'tcp:::send { @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
dtrace: description 'tcp:::send ' matched 3 probes
^C

  192.168.1.109
           value  ------------ Distribution ------------- count
```

```
        32 |                                            0
        64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    14
       128 |@@@                                          1
       256 |                                            0

 192.168.2.30
           value  ------------- Distribution ------------- count
        16 |                                            0
        32 |@@@@@@@@@@@@@@@@@@@@@                         7
        64 |@@@@@@@@@                                     3
       128 |@@@                                          1
       256 |@@@@@@                                       2
       512 |@@@                                          1
      1024 |                                            0
```

tcpstate.d

This DTrace script demonstrates the capability to trace TCP state changes:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10

int last[int];

dtrace:::BEGIN
{
        printf(" %3s %12s  %-20s     %-20s\n", "CPU", "DELTA(us)", "OLD", "NEW");
}

tcp:::state-change
/ last[args[1]->cs_cid] /
{
        this->elapsed = (timestamp - last[args[1]->cs_cid]) / 1000;
        printf(" %3d %12d  %-20s -> %-20s\n", cpu, this->elapsed,
            tcp_state_string[args[5]->tcps_state],
        tcp_state_string[args[3]->tcps_state]);
        last[args[1]->cs_cid] = timestamp;
}

tcp:::state-change
/ last[args[1]->cs_cid] == 0 /
{
        printf(" %3d %12s  %-20s -> %-20s\n", cpu, "-",
            tcp_state_string[args[5]->tcps_state],
            tcp_state_string[args[3]->tcps_state]);
        last[args[1]->cs_cid] = timestamp;
```

This script was run on a system for a couple of minutes:

```
# ./tcpstate.d

 CPU    DELTA(us)  OLD                     NEW
   0           -   state-listen        -> state-syn-received
   0         613   state-syn-received  -> state-established
   0           -   state-idle          -> state-bound
   0          63   state-bound         -> state-syn-sent
   0         685   state-syn-sent      -> state-bound
   0          22   state-bound         -> state-idle
   0         114   state-idle          -> state-closed
```

In the above example output, an inbound connection is traced, It takes 613 us to go from syn-received to established. An outbound connection attempt is also made to a closed port. It takes 63us to go from bound to syn-sent, 685 us to go from syn-sent to bound .

The fields printed are:

Field	Description
CPU	CPU id for the event
DELTA(us)	time since previous event for that connection, microseconds
OLD	old TCP state
NEW	new TCP state

tcpio.d

The following DTrace script traces TCP packets and prints various details:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf(" %3s %15s:%-5s       %15s:%-5s %6s  %s\n", "CPU",
            "LADDR", "LPORT", "RADDR", "RPORT", "BYTES", "FLAGS");
}

tcp:::send
{
        this->length = args[2]->ip_plength - args[4]->tcp_offset;
        printf(" %3d %16s:%-5d -> %16s:%-5d %6d  (", cpu,
            args[2]->ip_saddr, args[4]->tcp_sport,
            args[2]->ip_daddr, args[4]->tcp_dport, this->length);
}

tcp:::receive
```

```
{
        this->length = args[2]->ip_plength - args[4]->tcp_offset;
        printf(" %3d %16s:%-5d <- %16s:%-5d %6d  (", cpu,
            args[2]->ip_daddr, args[4]->tcp_dport,
            args[2]->ip_saddr, args[4]->tcp_sport, this->length);
}

tcp:::send,
tcp:::receive
{
        printf("%s", args[4]->tcp_flags & TH_FIN ? "FIN|" : "");
        printf("%s", args[4]->tcp_flags & TH_SYN ? "SYN|" : "");
        printf("%s", args[4]->tcp_flags & TH_RST ? "RST|" : "");
        printf("%s", args[4]->tcp_flags & TH_PUSH ? "PUSH|" : "");
        printf("%s", args[4]->tcp_flags & TH_ACK ? "ACK|" : "");
        printf("%s", args[4]->tcp_flags & TH_URG ? "URG|" : "");
        printf("%s", args[4]->tcp_flags & TH_ECE ? "ECE|" : "");
        printf("%s", args[4]->tcp_flags & TH_CWR ? "CWR|" : "");
        printf("%s", args[4]->tcp_flags == 0 ? "null " : "");
        printf("\b)\n");
}
```

This example output has captured a TCP handshake:

```
# ./tcpio.d
 CPU           LADDR:LPORT                  RADDR:RPORT  BYTES  FLAGS
   1     192.168.2.80:22      ->    192.168.1.109:60337    464  (PUSH|ACK)
   1     192.168.2.80:22      ->    192.168.1.109:60337     48  (PUSH|ACK)
   2     192.168.2.80:22      ->    192.168.1.109:60337     20  (PUSH|ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337      0  (SYN)
   3     192.168.2.80:22      ->    192.168.1.109:60337      0  (SYN|ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337      0  (ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337      0  (ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337     20  (PUSH|ACK)
   3     192.168.2.80:22      ->    192.168.1.109:60337      0  (ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337      0  (ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337    376  (PUSH|ACK)
   3     192.168.2.80:22      ->    192.168.1.109:60337      0  (ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337     24  (PUSH|ACK)
   2     192.168.2.80:22      ->    192.168.1.109:60337    736  (PUSH|ACK)
   3     192.168.2.80:22      <-    192.168.1.109:60337      0  (ACK)
```

The fields printed are:

Field	Description
CPU	CPU id that event occurred on
LADDR	local IP address
LPORT	local TCP port
RADDR	remote IP address

Field	Description
RPORT	remote TCP port
BYTES	TCP payload bytes
FLAGS	TCP flags

Note - The output may be shuffled slightly on multi-CPU servers due to DTrace per-CPU buffering, and events such as the TCP handshake can be printed out of order. Keep an eye on changes in the CPU column, or add a timestamp column to this script and post sort.

Stability

The tcp provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 87   Stability Mechanism for tcp Provider

Element	Name Stability	Data Stability	Dependency Class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

udp Provider

The udp provider provides probes for tracing the UDP protocol.

Probes

The udp probes are described in the table below.

TABLE 88   udp Probes

Probe	Description
send	Probe that fires whenever UDP sends a datagram.

Probe	Description
receive	Probe that fires whenever UDP receives a datagram.

The send and receive probes trace datagrams on physical interfaces and also packets on loopback interfaces that are processed by udp.

Arguments

The argument types for the udp probes are listed in the table below. The arguments are described in the following section.

TABLE 89   udp Probe Arguments

Probe	args[0]	args[1]	args[2]	args[3]	args[4]
send	pktinfo_t *	csinfo_t *	ipinfo_t *	udpinfo_t *	udpinfo_t *
receive	pktinfo_t *	csinfo_t *	ipinfo_t *	udpinfo_t *	udpinfo_t *

pktinfo_t Structure

The pktinfo_t structure is where packet ID info can be made available for deeper analysis if packet IDs become supported by the kernel in the future.

The pkt_addr member is currently always NULL.

```
typedef struct pktinfo {
        uintptr_t pkt_addr;              /* currently always NULL */
} pktinfo_t;
```

csinfo_t Structure

The csinfo_t structure is where connection state info is made available. It contains a unique (system-wide) connection ID, and the process ID and zone ID associated with the connection.

```
typedef struct csinfo {
        uintptr_t cs_addr;
 uint64_t cs_cid;
 pid_t cs_pid;
 zoneid_t cs_zoneid;
 } csinfo_t;
```

TABLE 90   csinfo_t Members

cs_addr	Address of translated ip_xmit_attr_t *.

cs_cid	Connection id. A unique per-connection identifier which identifies the connection during its lifetime.
cs_pid	Process ID associated with the connection.
cs_zoneid	Zone ID associated with the connection.

`ipinfo_t` Structure

The `ipinfo_t` structure contains common IP info for both IPv4 and IPv6.

```
typedef struct ipinfo {
        uint8_t ip_ver;                 /* IP version (4, 6) */
        uint16_t ip_plength;            /* payload length */
        string ip_saddr;                /* source address */
        string ip_daddr;                /* destination address */
} ipinfo_t;
```

These values are read at the time the probe fired in UDP, and so `ip_plength` is the expected IP payload length. However, the IP layer may add headers such as AH and ESP, which will increase the actual payload length. To examine this, also trace packets using the `ip` provider.

TABLE 91   `ipinfo_t` Members

ip_ver	IP version number. Currently either 4 or 6.
ip_plength	Payload length in bytes. This is the length of the packet at the time of tracing, excluding the IP header.
ip_saddr	Source IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.
ip_daddr	Destination IP address, as a string. For IPv4 this is a dotted decimal quad, IPv6 follows RFC-1884 convention 2 with lower case hexadecimal digits.

`udpsinfo_t` Structure

The `udpsinfo_t` structure contains udp state info.

```
typedef struct udpsinfo {
        uintptr_t       udps_addr;
 uint16_t udps_lport; /* local port */
 uint16_t udps_fport; /* remote port */
 string          udps_laddr; /* local address, as a string */
 string          udps_faddr; /* remote address, as a string */
} udpsinfo_t;
```

TABLE 92   `udpsinfo_t` Members

udps_addr	Address of translated `udp_t *`.

udps_lport	local port associated with the UDP connection.
udps_fport	remote port associated with the UDP connection.
udps_laddr	local address associated with the UDP connection, as a string
udps_fport	remote address associated with the UDP connection, as a string

`udpsinfo_t` Structure

The `udpinfo_t` structure is a DTrace translated version of the UDP header.

```
typedef struct udpinfo {
 uint16_t udp_sport;            /* source port */
 uint16_t udp_dport;            /* destination port */
 uint16_t udp_length;           /* total length */
 uint16_t udp_checksum;         /* headers + data checksum */
 udpha_t *udp_hdr;              /* raw UDP header */
} udpinfo_t;
```

TABLE 93   `udpinfo_t` Members

udp_sport	UDP source port.
udp_dport	UDP destination port.
udp_length	Payload length in bytes.
udp_checksum	Checksum of UDP header and payload.
udp_hdr	Pointer to raw UDP header at time of tracing.

See RFC-768 for a detailed explanation of the standard UDP header fields and flags.

Examples

Some simple examples of udp provider usage follow.

Packets by Host Address

This DTrace one-liner counts UDP received packets by host address:

```
# dtrace -n 'udp:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace: description 'udp:::receive ' matched 5 probes
^C
  127.0.0.1                                                7
```

```
 fe80::214:4fff:fe8d:59aa                                        14
 192.168.2.30                                                    43
 192.168.1.109                                                   44
 192.168.2.88                                                  3722
```

The output above shows that 7 UDP packets were received from 127.0.0.1, 14 UDP packets from the IPv6 host fe80::214:4fff:fe8d:59aa.

Packets by Local Port

This DTrace one-liner counts UDP received packets by the local UDP port:

```
# dtrace -n 'udp:::receive { @[args[4]->udp_dport] = count(); }'
dtrace: description 'udp:::receive ' matched 1 probe
^C

 33294               1
 33822               1
 38961               1
 44433               1
 46258               1
 46317               1
 47511               1
 50581               1
 54685               1
 56491               1
 59056               1
 62171               1
 62769               1
 64231               1
```

The output above shows that 1 packet was received for port 33294, 1 packet was received for port 33822, and so on.

Sent Size Distribution

This DTrace one-liner prints distribution plots of IP payload size by destination, for UDP sends:

```
# dtrace -n 'udp:::send { @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
dtrace: description 'udp:::send ' matched 6 probes
^C

 129.156.86.11
          value  ------------- Distribution ------------- count
             16 |                                         0
             32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
             64 |                                         0
```

Stability

The udp provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table. For more information about the stability mechanism, see Chapter 18, "Stability".

TABLE 94   Stability Mechanism for the udp Provider

Element	Name Stability	Data Stability	Dependency Class
Provider	Evolving	Evolving	ISA
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Arguments	Evolving	Evolving	ISA

User Process Tracing

DTrace is an extremely powerful tool for understanding the behavior of user processes. DTrace can be invaluable when 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.

`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, 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 Oracle 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 following `write` 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` system call:

```
syscall::write:entry
{
        printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

If you try to run this script, DTrace will produce error messages similar to the following example:

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

The `arg1` variable, containing the value of the *buf* parameter, is an address that refers to memory in the process executing the system call. To read the string at that address, use the `copyinstr` subroutine and record its result with the `printf` action:

```
syscall::write:entry
{
        printf("%s", copyinstr(arg1)); /* correct use of arg1 */
}
```

The output of this script shows all of the strings being passed to the `write` system call. Occasionally, however, you might see irregular output similar to the following example:

```
 0    37                      write:entry mada&^%**&
```

The `copyinstr` subroutine acts on an input argument that is the user address of a null-terminated ASCII string. However, buffers passed to the `write` system call might refer to binary data rather than ASCII strings or to ASCII strings which do not include a terminating null byte. To print only as much of the string as the caller intended, use the two parameter version of the `copyinstr` subroutine which includes the size of the targeted string buffer:

```
syscall::write:entry
{
        printf("%s", copyinstr(arg1, arg2));
}
```

An alternate way to accomplish the same end would be to use the `copyin` subroutine which takes an address and size:

```
syscall::write:entry
{
        printf("%s", stringof(copyin(arg1, arg2)));
}
```

Notice that the `stringof` operator is necessary so that DTrace properly converts the user data retrieved using `copyin` to a string. The use of `stringof` is not necessary when using `copyinstr` because this function always returns type `string`.

Avoiding Errors

The `copyin` and `copyinstr` subroutines cannot read from user addresses which have not yet been touched so even a valid address may cause an error if the page containing that address has not yet been faulted in by being accessed. Consider the following example:

```
# dtrace -n syscall::openat:entry'{ trace(copyinstr(arg1)); }'
dtrace: description 'syscall::openat:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 2 (ID 50: syscall::openat:entry): invalid address
(0x9af1b) in action #1 at DIF offset 52
```

In the above example output, the application was functioning properly, and the address in `arg0` was valid, but it referred to a page that had not yet been accessed by the corresponding process. To resolve this issue, wait for kernel or application to use the data before tracing it.

For example, you might wait until the system call returns to apply `copyinstr`, as shown in the following example:

```
# dtrace -n syscall::openat:entry'{ self->file = arg1; }' \
-n syscall::openat:return'{ trace(copyinstr(self->file)); self->file = 0; }'
dtrace: description 'syscall::openat:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2     51                      open:return   /dev/null
```

Eliminating `dtrace` Interference

If you trace every call to the `write` system call, you will cause a cascade of output. Each call to `write` causes the `dtrace` command 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. You 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 the script does not trace any events related to the running of this script itself.

`syscall` Provider

The `syscall` provider enables you to trace every system call entry and return. System calls can be a good starting point for understanding a process's behavior, especially if the process seems to be spending a large amount of time executing or blocked in the kernel. You can use the `prstat` 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
```

This example shows 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. You can use a simple D program specified on the command-line to see which system calls 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 shows which system calls are being called most often, in this case, the write system call. You can use the syscall provider to further examine the source of all the write system calls:

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes = quantize(arg2); }'
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
```

The output shows that the process is executing many write system calls with a relatively small amount of data. This ratio could be the source of the performance problem for this particular process. This example illustrates a general methodology for investigating system call behavior.

ustack Action

Tracing a process thread's stack at the time a particular probe is activated is often useful for examining a problem in more detail. The ustack action traces the user thread's stack. If, for example, a process that opens many files occasionally fails in the open system call, you can use the ustack action to discover the code path that executes the failed open:

```
syscall::openat:entry
/pid == $1/
{
        self->path = copyinstr(arg1);
}

syscall::openat:return
/self->path != NULL && 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 command-line:

```
# dtrace -s ./badopen.d 31337
dtrace: script './badopen.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
```

The ustack action records program counter (PC) values for the stack and dtrace 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 integer.

If a process exits or is killed before the ustack data is formatted for output, dtrace might be unable to convert the PC values in the stack trace to symbol names, and will be forced to display them as hexadecimal integers. To work around this limitation, specify a process of interest with the -c or -p option to dtrace. For information about the dtrace command options, see Chapter 8, "dtrace Utility". If the process ID or command is not known in advance, the following example D program can be used 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::openat:entry
{
        ustack();
        stop_pids[pid] = 1;
}
```

```
syscall::rexit:entry
/stop_pids[pid] != 0/
{
        printf("stopping pid %d", pid);
        stop();
        stop_pids[pid] = 0;
}
```

The above script stops a process just before it exits if the ustack action has been applied to a thread in that process. This technique 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. Remember to set stopped processes running again using the prun command or your system will accumulate many stopped processes. For more information, see .

uregs[] Array

The uregs[] array enables you to access individual user registers. The following tables list indices into the uregs[] array corresponding to each supported Oracle Solaris system architecture.

TABLE 95   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 96   x86 uregs[] Constants

Constant	Register
R_CS	%cs
R_GS	%gs
R_ES	%es

Constant	Register
R_DS	%ds
R_EDI	%edi
R_ESI	%esi
R_EBP	%ebp
R_EAX	%eax
R_ESP	%esp
R_EAX	%eax
R_EBX	%ebx
R_ECX	%ecx
R_EDX	%edx
R_TRAPNO	%trapno
R_ERR	%err
R_EIP	%eip
R_CS	%cs
R_ERR	%err
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

On AMD64 platforms, the `uregs` array has the same content as it does on x86 platforms, plus the additional elements listed in the following table:

TABLE 97   amd64 `uregs[]` Constants

Constant	Register
R_RSP	%rsp
R_RFL	%rfl
R_RIP	%rip
R_RAX	%rax
R_RCX	%rcx
R_RDX	%rdx
R_RBX	%rbx
R_RBP	%rbp
R_RSI	%rsi
R_RDI	%rdi
R_R8	%r8
R_R9	%r9
R_R10	%r10
R_R11	%r11

Constant	Register
R_R12	%r12
R_R13	%r13
R_R14	%r14
R_R15	%r15

The aliases listed in the following table can be used on all platforms:

Constant	Register
R_PC	program counter register
R_SP	stack pointer register
R_R0	first return code
R_R1	second return code

pid Provider

The pid provider enables you to trace any instruction in a process. Unlike most other providers, pid probes are created on demand based on the probe descriptions found in your D programs. As a result, no pid probes are listed in the output of dtrace -l until you have enabled the probes.

User Function Boundary Tracing

The simplest mode of operation for the pid provider is as the user space analogue to the fbt provider. The following example program traces all function entries and returns that are made from a single function. The $1 macro variable, that is the first operand on the command line, is the process ID for the process to trace. The $2 macro variable, that is the second operand on the command line, is the name of the function from which to trace all function calls.

EXAMPLE 28   userfunc.d: Trace User Function Entry and Return

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent

pid$1::$2:entry
{
```

```
        self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
        self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

Type in the above example script and save it in a file named `userfunc.d`, and then use the `chmod` command to set the execute permisison. This script produces output similar to the following example:

```
# ./userfunc.d 15032 execute
dtrace: script './userfunc.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 `$target` macro variable and the `dtrace -c` and `-p` options to create and grab processes of interest and instrument them using DTrace. For more information about macro variables, see see Chapter 9, "Scripting". For example, the following D script can be used to determine the distribution of function calls made to `libc` by a particular subject process:

```
pid$target:libc.so::entry
{
        @[probefunc] = count();
```

```
}
```

To determine the distribution of such calls made by the `date` command, save the script in a file named `libc.d` and execute the following command:

```
# dtrace -s libc.d -c date
dtrace: script 'libc.d' matched 2476 probes
Fri Jul 30 14:08:54 PDT 2004
dtrace: pid 109196 has exited

  pthread_rwlock_unlock                                            1
  _fflush_u                                                        1
  rwlock_lock                                                      1
  rw_write_held                                                    1
  strftime                                                         1
  _close                                                           1
  _read                                                            1
  __open                                                           1
  _open                                                            1
  strstr                                                           1
  load_zoneinfo                                                    1

...
  _ti_bind_guard                                                  47
  _ti_bind_clear                                                  94
```

Tracing Arbitrary Instructions

You can use the `pid` provider 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
```

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 command demonstrates an extremely powerful technique for debugging and analyzing user applications. Infrequent errors can be difficult to debug because they can be difficult to reproduce. Often, you can identify a problem after the failure has occurred, too late to reconstruct the code path. The following example demonstrates how to combine the `pid` provider with speculative tracing to solve this problem by tracing every instruction in a function. For more information about speculative tracing, see Chapter 7, "Speculative Tracing".

EXAMPLE 29   `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;
}
```

Executing `errorpath.d` results in output similar to the following example:

```
# ./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
```

13

Statically Defined Tracing for User Applications

DTrace provides a facility for user application developers to define customized probes in application code to augment the capabilities of the `pid` provider. These static probes impose little to no overhead when disabled and are dynamically enabled like all other DTrace probes. You can use static probes to describe application semantics to users of DTrace without exposing or requiring implementation knowledge of your applications. This chapter describes how to define static probes in user applications and how to use DTrace to enable such probes in user processes.

Choosing the Probe Points

DTrace allows developers to embed static probe points in application code, including both complete applications and shared libraries. These probes can be enabled wherever the application or library is running, either in development or in production. You should define probes that have a semantic meaning that is readily understood by your DTrace user community. For example, you could define `query-receive` and `query-respond` probes for a web server that correspond to a client submitting a request and the web server responding to that request. These example probes are easily understood by most DTrace users and correspond to the highest level abstractions for the application, rather than lower level implementation details. DTrace users might use these probes to understand the time distribution of requests. If your `query-receive` probe presented the URL request strings as an argument, a DTrace user could determine which requests were generating the most disk I/O by combining this probe with the `io` provider.

You should also consider the stability of the abstractions you describe when choosing probe names and locations. Will this probe persist in future releases of the application, even if the implementation changes? Does the probe make sense on all system architectures or is it specific to a particular instruction set? This chapter will discuss the details of how these decisions guide your static tracing definitions.

Adding Probes to an Application

DTrace probes for libraries and executables are defined in an ELF section in the corresponding application binary. This section describes how to define your probes, add them to your application source code, and augment your application's build process to include the DTrace probe definitions.

Defining Providers and Probes

You define DTrace probes in a `.d` source file which is then used when compiling and linking your application. First, select an appropriate name for your user application provider. The provider name you choose will be appended with the process identifier for each process that is executing your application code. For example, if you chose the provider name `myserv` for a web server that was executing as process ID 1203, the DTrace provider name corresponding to this process would be `myserv1203`. In your `.d` source file, add a provider definition similar to the following example:

```
provider myserv {
        ...
};
```

Next, add a definition for each probe and the corresponding arguments. The following example defines the two probes: `probe_query_receive` and `probe query__respond`. The first probe has two arguments, both of type `string`, and the second probe has no arguments. For more information about the probes in this example, see "Choosing the Probe Points" on page 389. The D compiler converts two consecutive underscores (`--`) in any probe name to a hyphen (`-`).

```
provider myserv {
        probe query__receive(string, string);
        probe query__respond();
};
```

You should add stability attributes to your provider definition so that consumers of your probes understand the likelihood of change in future versions of your application. For more information on the DTrace stability attributes, see Chapter 18, "Stability". Stability attributes are defined as shown in the following example:

EXAMPLE 30   `myserv.d`: Statically Defined Application Probes

```
#pragma D attributes Evolving/Evolving/Common provider myserv provider
#pragma D attributes Private/Private/Unknown provider myserv module
#pragma D attributes Private/Private/Unknown provider myserv function
```

```
#pragma D attributes Evolving/Evolving/Common provider myserv name
#pragma D attributes Evolving/Evolving/Common provider myserv args

provider myserv {
        probe query__receive(string, string);
        probe query__respond();
};
```

Adding Probes to Application Code

Now that you have defined your probes in a `.d` file, you need to augment your source code to indicate the locations that should trigger your probes. Consider the following example C application source code:

```
void
main_look(void)
{
        ...
        query = wait_for_new_query();
        process_query(query)
        ...
}
```

To add a probe site, add a reference to the `DTRACE_PROBE()` macro defined in `<sys/sdt.h>` as shown in the following example:

```
...
void
main_look(void)
{
        ...
        query = wait_for_new_query();
        DTRACE_PROBE2(myserv, query__receive, query->clientname, query->msg);
        process_query(query)
        ...
}
```

The suffix 2 in the macro name `DTRACE_PROBE2` refers the number of arguments that are passed to the probe. The first two arguments to the probe macro are the provider name and probe name and must correspond to your D provider and probe definitions. The remaining macro arguments are the arguments assigned to the DTrace `arg0..9` variables (or the `args[]` array) when the probes fires . Your application source code can contain multiple references to the same provider and probe name. If multiple references to the same probe are present in your source code, any of the macro references will cause the probe to fire.

Building Applications with Probes

You must augment the build process for your application to include the DTrace provider and probe definitions. A typical build process takes each source file and compiles it to create a corresponding object file. The compiled object files are then linked together to create the finished application binary, as shown in the following example:

```
cc -c src1.c
cc -c src2.c
...
cc -o myserv src1.o src2.o ...
```

To include DTrace probe definitions in your application, add appropriate Makefile rules to your build process to execute the `dtrace` command as shown in the following example:

```
cc -c src1.c
cc -c src2.c
...
dtrace -G -32 -s myserv.d src1.o src2.o ...
cc -o myserv myserv.o src1.o src2.o ...
```

The `dtrace` command shown above post-processes the object files generated by the preceding compiler commands and generates the object file `myserv.o` from `myserv.d` and the other object files. The `dtrace -G` option is used to link provider and probe definitions with a user application. The -32 option is used to build 32–bit application binaries. The -64 option is used to build 64–bit application binaries.

If a user-land object contains a very large number of statically-defined probes then there may be a measurable delay when the object is first executed or loaded.In this case, you can use the `lazyload` option when building the object. For example:

```
dtrace -x lazyload -G ...
```

Objects built in this way do not advertise their probes to the kernel until required to do so.

```
dtrace -l | fgrep myserv
```

The above command might not show any results until after running. For example:

```
dtrace -n myserv1203:::query-receive ...
```

You can also type the following command:

```
dtrace -l -n myserv1203:::
```

14

Security

This chapter describes the privileges that system administrators can use to grant access to DTrace to particular users or processes. DTrace enables visibility into all aspects of the system including user-level functions, system calls, kernel functions, and more. It allows for powerful actions some of which can modify a program's state. Just as it would be inappropriate to allow a user access to another user's private files, a system administrator must not grant every user full access to all the facilities that DTrace offers. By default, only the super-user can use DTrace. The Least Privilege facility can be used to allow other users controlled use of DTrace.

Privileges

The Oracle Solaris Least Privilege facility enables administrators to grant specific privileges to specific Oracle Solaris users. To give a user a privilege on login, use the `usermod -K` command. For example:

```
$ usermod -K defaultpriv='basic,proc_clock_highres' jdoe
```

For more information about assigning privileges directly to a user, see "Expanding Users' Rights" in *Securing Users and Processes in Oracle Solaris 11.3* guide. For a list of all the privileges that can be assigned to a user, see the `privileges(5)` man page.

To give a running process an additional privilege, use the `ppriv(1)` command:

```
# ppriv -s A+privilege process-ID
```

The three privileges that control a user's access to DTrace features are `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.

Privileged Use of DTrace

Users with any of the three DTrace privileges can enable probes provided by the `dtrace` provider. You can 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		

For more information, see "dtrace Provider" on page 181.

`dtrace_proc` Privilege

The `dtrace_proc` privilege enables use of the `pid` and `fasttrap` providers for process-level tracing. The following table list actions and variables that you can use.

Providers	pid		
Actions	copyin	copyout	stop
	copyinstr	raise	ustack
Variables	execname	pid	uregs
Address Spaces	User		

This privilege does not grant any visibility to Oracle Solaris kernel data structures or to processes for 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. The `dtrace_proc` privilege is intended for users interested in the debugging or performance analysis of user processes. This privilege is ideal for a developer working on a new application or an engineer trying to improve an application's performance in a production environment.

Note - 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_proc privilege allows access to DTrace that can impose a performance 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.

dtrace_user Privilege

The dtrace_user privilege enables use of the profile and syscall providers with some caveats. The following table describes the action and variables that you can use.

Providers	profile	syscall	fasttrap
Actions	copyin	copyout	stop
Variables	execname	pid	uregs
	copyinstr	raise	ustack
Address Spaces	User		

The dtrace_user privilege provides visibility only 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, you can enable the profile provider, but the enabled probes will only activate in processes to which the user has permission. The profile provider cannot be acivated in the Oracle Solaris kernel.

This privilege enables the use of instrumentation that, while only allowing visibility into particular processes, can affect overall system performance. The syscall provider has some small performance impact on every system call for every process. The profile provider affects overall system performance by executing every time interval, similar to a real-time timer. Neither of these performance degradations severely limit the system's progress, but system administrators should consider the implications of granting a user this privilege. For information about the performance impact of the syscall and profile providers, see "syscall Provider" on page 211 and "profile Provider" on page 188.

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 that are not owned by the user. This privilege also permits the use of all actions and variables except for kernel destructive actions, such as breakpoint, panic, and chill. This privilege permits complete visibility into kernel and user state. 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

Note - All functionalities of DTrace are not available if you are running DTrace in a non-global zone. For more information, see "Running DTrace in a Non-Global Zone" in *Creating and Using Oracle Solaris Zones*.

Super User Privileges

A user with all privileges may use every provider 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

15

Anonymous Tracing

This chapter describes *anonymous* tracing, tracing that is not associated with any DTrace consumer. Anonymous tracing is used in situations when no DTrace consumer processes can run. The most common use of anonymous tracing is to permit device driver developers to debug and trace activity that occurs during system boot. Any tracing that you can perform interactively can also be done anonymously. However, only the super user may create an anonymous enabling, and only one anonymous enabling can exist at any time.

Anonymous Enablings

To create an anonymous enabling, use the -A option with a dtrace invocation that specifies the desired probes, predicates, actions and options. dtrace will add a series of driver properties representing your request to the dtrace driver's configuration file, typically /kernel/drv/ dtrace.conf. These properties will be read by the dtrace driver when it is loaded. 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 driver is loaded on-demand, similar to any drivers that act as DTrace providers. To allow tracing during boot, the dtrace driver must be loaded as early as possible. dtrace adds the necessary forceload statements to /etc/system for each required DTrace provider and for dtrace itself.

Thereafter, when the system boots, a message is emitted by dtrace to indicate that the configuration file has been successfully processed.

All options may be set with an anonymous enabling, including buffer size, dynamic variable size, speculation size, number of speculations, and so on.

To remove an anonymous enabling, specify -A with dtrace without any probe descriptions.

Claiming Anonymous State

Once the system has completely booted, any anonymous state may be claimed by specifying the
--a option with dtrace. By default, -a claims the anonymous state, processes the existing data,
and continues to run. To consume the anonymous state and then exit, add the -e option.

Once anonymous state has been consumed from the kernel, it cannot be replaced as the
in-kernel buffers that contained the anonymous state are reused. If you attempt to claim
anonymous tracing state where none exists, dtrace generates a message similar to the following
example:

```
dtrace: could not enable tracing: 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.

Anonymous Tracing Examples

EXAMPLE 31   Enabling Anonymous Tracing for the iprb Module

The following example shows an anonymous DTrace enabling for every probe in the iprb
module:

```
# dtrace -A -m iprb
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

After rebooting, dtrace 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 system has rebooted, the anonymous state may be consumed by specifying the -a
option with 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
  ...
```

EXAMPLE 32   Enabling Anonymous Tracing for Functions Called From `iprbattach`

The following example focuses only on those functions called from `iprbattach`. In an editor,
type the following script and save it in a file named `iprb.d`.

```
fbt::iprbattach:entry
{
        self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
        self->trace = 0;
}
```

Run 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
dtrace: cleaned up old anonymous enabling in /kernel/drv/dtrace.conf
dtrace: cleaned up forceload directives in /etc/system
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
```

```
# reboot
```

After rebooting, `dtrace` 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 system has completely booted, run the `dtrace` with the -a option and the -e option to
consume the anonymous data and then exit.

```
# 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
...
```

16

Postmortem Tracing

This chapter describes the DTrace facilities for *postmortem* extraction and processing of the in-kernel data of DTrace consumers. 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. DTrace data may be extracted and processed from the system crash dump to aid you in understanding fatal system failures. By coupling these postmortem capabilities of DTrace with its ring buffering buffer policy, DTrace can be used as an operating system analog to the *black box* flight data recorder present on commercial aircraft. For more information about buffering policies, see Chapter 5, "Buffers and Buffering".

To extract DTrace data from a specific crash dump, you should begin by running the Oracle Solaris Modular Debugger, `mdb`, on the crash dump of interest. The MDB module containing the DTrace functionality will be loaded automatically. To learn more about MDB, see *Oracle Solaris Modular Debugger Guide*.

Displaying DTrace Consumers

To extract DTrace data from a DTrace consumer, you must first determine the DTrace consumer of interest by running 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 command displays a table of DTrace state structures. Each row of the table consists of the following information:

- The address of the state structure
- The minor number associated with the `dtrace` 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` device

To obtain further information about a specific DTrace consumer, specify the address of its process structure to the `::ps` dcmd:

```
> d71377f0::ps
S    PID  PPID  PGID   SID    UID     FLAGS     ADDR NAME
R 100647 100642 100647 100638     0 0x00004008 d71377f0 dtrace
```

Displaying Trace Data

Once you determine the consumer of interest, you can retrieve the data corresponding to any unconsumed buffers by specifying the address of the state structure to the `::dtrace` dcmd. The following example shows 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 as `dtrace`. If drops, errors, speculative drops, or the like are encountered, `::dtrace` will emit a message corresponding to the `dtrace` message.

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, trace the timestamp variable.

You can 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
...
```

Notice that ::dtrace only processes *in-kernel* DTrace data. Data that has been consumed from the kernel and processed through dtrace 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, use a ring buffer buffering policy. For more information on buffer policies, see Chapter 5, "Buffers and Buffering".

The following example creates a 16K ring buffer and records all system calls and the process making the ring buffer:

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

Looking at a crash dump taken when the above command was running results in output similar to the following example:

```
> ::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 system calls by an mdb -kw
process. This result is likely related to the reason for the system failure because a user can
modify running kernel data or text with mdb 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.

17

Performance Considerations

Because DTrace causes additional work in the system, enabling DTrace always 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.

Limit Enabled Probes

Dynamic instrumentation techniques enable DTrace to provide unparalleled tracing coverage of the kernel and of arbitrary user processes. While this coverage allows revolutionary new insight into system behavior, it also can cause enormous probe effect. If tens of thousands or hundreds of thousands of probes are enabled, the effect on the system can easily be substantial. Therefore, you should only enable as many probes as you need to solve a problem. You should not, for example, enable all FBT probes if a more concise enabling will answer your question. For example, your question might allow you to concentrate on a specific module of interest or a specific function.

When using the `pid` provider, you should be especially careful. Because the `pid` provider can instrument every *instruction*, you could enable millions of probes in an application, and therefore slow the target process to a crawl.

DTrace can also be used in situations where large numbers of probes *must* be enabled for a question to be answered. Enabling a large number of probes might slow down the system quite a bit, but it will never induce fatal failure on the system. You should therefore not hesitate to enable many probes if required.

Use Aggregations

DTrace's aggregations enable for a scalable way of aggregating data. Associative arrays might appear to offer similar functionality to aggregations. However, by nature of being global, general-purpose variables, they cannot offer the linear scalability of aggregations. You

should therefore prefer to use aggregations over associative arrays when possible. For more information, see Chapter 3, "Aggregations".

The following example is not recommended:

```
syscall:::entry
{
        totals[execname]++;
}

syscall::rexit:entry
{
        printf("%40s %d\n", execname, totals[execname]);
        totals[execname] = 0;
}
```

The following example is preferable:

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

END
{
        printa("%40s %@d\n", @totals);
}
```

Use Cacheable Predicates

DTrace predicates are used to filter unwanted data from the experiment by tracing data only if a specified condition is found to be true. When enabling many probes, you generally use predicates of a form that identifies a specific thread or threads of interest, such as `/self->traceme/` or `/pid == 12345/`. Although many of these predicates evaluate to a false value for most threads in most probes, the evaluation itself can become costly when done for many thousands of probes. To reduce this cost, DTrace caches the evaluation of a predicate if it includes only thread-local variables, for example `/self->traceme/`, or immutable variables, for example `/pid == 12345/`. 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, it does imply some guidelines for constructing optimal predicates, as shown in the following table.

Cacheable	Uncacheable
`self->mumble`	`mumble`*curthread*, `mumble`*pid, tid*

Cacheable	Uncacheable
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*

The following example is not recommended:

```
syscall::read:entry
{
        follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
        follow[pid, tid] = 0;
}
```

The following example using thread-local variables is preferable:

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

fbt:::
/self->follow/
{}

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

A predicate must consist *exclusively* of cacheable expressions in order to be cacheable. The following predicates are all cacheable:

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

The following examples, which use global variables, are not cacheable:

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

18 CHAPTER 18

♦♦♦ **C H A P T E R   1 8**

Stability

Developers are provided 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. Application and interface stability levels are documented 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.

No one stability attribute appropriately describes the arbitrary set of entities and services that can be accessed from a D program. DTrace and the D compiler therefore include features to dynamically compute and describe the stability levels of D programs you create. This chapter discusses the DTrace features for determining program stability to help you 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.

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 Oracle 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 appear in the following list in order from lowest to highest stability. 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 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, the following 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:

- Internal – The interface is private to DTrace and represents an implementation detail of DTrace. Internal interfaces might change in minor or micro releases.
- Private – The interface is private to Oracle and represents an interface developed for use by other Oracle products that is not yet publicly documented for use by customers and ISVs. Private interfaces might change in minor or micro releases.
- Obsolete – The interface is supported in the current release but is scheduled to be removed, most likely in a future minor release. The D compiler might produce warning messages if you attempt to use an Obsolete interface.
- External – The interface is controlled by an entity other than Oracle. Oracle 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 that is essential for observing or debugging system behavior for which a more stable solution is anticipated in the future. Oracle makes no claims about either source or binary compatibility for Unstable interfaces from one minor release to another.
- Evolving – The interface might eventually become Standard or Stable but is still in transition. When non-upward compatible changes become necessary, they will occur in minor and major releases. These 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, migration aids will be provided for binary compatibility and continued D program development.
- Stable – The interface is a mature interface.
- Standard – The interface complies with an industry standard. The corresponding documentation for the interface 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 to 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 versions of a standard; support for later versions is not guaranteed.

Dependency Classes

Since Oracle 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 Oracle 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 Oracle Solaris systems. The DTrace dependency classes are described in the following list 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 might 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. You can use the `psrinfo` utility's `-v` option to display the current CPU model and implementation names. Interfaces with CPU model dependencies might 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 might 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`. You can display the current platform name using the `uname -i` option. The interface might 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`. You can display the current platform group name using the `uname -m` option. The interface is available on other platforms in the platform group, but might 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. You can display the native instruction sets supported by the system using the `isainfo` utility. The interface might not be supported on systems that do not export any of the same instruction sets. For example, an ISA-dependent interface on a Oracle Solaris SPARC system might not be supported on a Oracle Solaris x86 system.

- Common – The interface is common to all Oracle 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 Oracle Solaris systems with the same Oracle Solaris and DTrace revisions. The majority of DTrace interfaces are Common, so you can use them wherever you use Oracle Solaris.

Interface Attributes

DTrace describes interfaces using a triplet of attributes consisting of two stability levels and a dependency class. By convention, the interface attributes are written 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. For example, the `execname` D variable is a `Stable` name.

The *data stability* of an interface is distinct from the stability associated with the interface name. This stability level describes the commitment to maintain the data formats used by the interface and any associated data semantics.

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.

DTrace and the D compiler track the stability attributes for all of the DTrace interface entities, including providers, probe descriptions, D variables, D functions, types, and program statements themselves. 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 Oracle Solaris operating platforms, but this variable provides access to a Private data format that is an artifact of the Oracle Solaris kernel implementation. Most D variables are provided with `Stable/Stable/Common` attributes, as are the variables you define.

USDT Interfaces

USDT providers require that stabilities be defined for the following five interface groups that fully specify a DTrace probe:

- Provider
- Modules
- Functions
- Probes
- Arguments

This will generally be done in the D program file where the provider is declared with pragmas using the triplets discussed in "Interface Attributes" on page 414. For example:

```
#pragma D attributes Evolving/Evolving/Common provider providername provider
```

```
#pragma D attributes Private/Private/Common provider providername module
#pragma D attributes Private/Private/Common provider providername function
#pragma D attributes Evolving/Evolving/Common provider providername name
#pragma D attributes Evolving/Evolving/Common provider providername args
```

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. The following example uses 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
#
```

This example references the D variable curthread, which has a Stable name, but Private data semantics, which is accessing Private implementation details of the kernel, and this status 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:::
```

This example probe indicates that DTrace should trace entry and return from all kernel functions as this has greater stability than the probe description:

```
fbt:foo:bar:entry
```

This probe names a specific internal function `bar` in the 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 varies by provider.

If the provider field is not specified in a probe description, then the description is assigned the stability attributes `Unstable/Unstable/Common` because the description might end up matching probes of providers that do not yet exist when used on a future Oracle Solaris version. As such, Oracle is not able to provide guarantees about the future stability and behavior of this program. You must always explicitly specify the provider when writing the D program clauses. In addition, any probe description fields that contain pattern matching characters are treated as if they are unspecified because these description patterns might expand to match providers or probes released in future versions of DTrace and the Oracle Solaris OS. For more information about pattern matching characters, see "D Program Structure" on page 23) or macro variables such as $1 (see Chapter 9, "Scripting".

Stability attributes are computed for most D language statements by taking the minimum stability and class of the entities in the statement. For example, the following D language entities have the following attributes:

Entity	Attributes
D built-in variable `curthread`	`Stable/Private/Common`
D user-defined variable `x`	`Stable/Stable/Common`

For example, write the following D program statement:

```
x += curthread->t_pri;
```

The resulting attributes of the statement are `Stable/Private/Common`, the minimum attributes associated with the operands `curthread` and `x`. 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 namespace, 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. Therefore, you might be misled about the 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.

Stability Enforcement

When developing a DTrace script or layered tool, you might want 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
#
```

♦♦♦ **C H A P T E R 1 9**

Translators

DTrace computes and reports program stability attributes. Ideally, you might like to construct DTrace programs by consuming only `Stable` or `Evolving` interfaces. However, when debugging a low-level problem or measuring system performance, you might 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 Oracle 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.

Translator Declarations

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. To understand the need for and use of translators, consider the ANSI-C standard library routines defined in `stdio.h` as an example. 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` function to obtain the descriptor rather than dereferencing a member of the FILE struct directly. The Oracle 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 suppose that `fileno` is implemented in C as follows:

```
int
fileno(FILE *fp)
{
        struct file_impl *ip = (struct file_impl *)fp;
```

```
        return (ip->fd);
}
```

The 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. By abstracting the details of the current `libc` implementation away from client programs, it is possible to maintain a commitment to strong binary compatibility while continuing to evolve and change the internal implementation details of `libc`. In this example, the `fd` member could change size or position within `struct file_impl`, even in a patch, and existing binaries calling `fileno` would not be affected by this change because they do not depend on these artifacts.

However, observability software such as DTrace must peer inside the implementation in order to provide useful results, and does not have the luxury of calling arbitrary C functions defined in Oracle Solaris libraries or in the kernel. You could declare a copy of `struct file_impl` in the D program in order to instrument the routines declared in `stdio.h`, but then the 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, you would 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 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, 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` can 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 the example translator, the input expression is of the 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` shown above. A similar translation is performed to obtain the value of the EOF indicator.

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 assign 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:

1. The compiler looks for a translation from the exact input expression type to the exact output type.
2. 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.
3. 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.

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` structures `psinfo` and `lwpsinfo`. These structures are also used in the Oracle Solaris `/proc` file system 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` command. For description of the struct members and semantics, see [proc(4)](#).

TABLE 98 `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

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, you must 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.

20

Versioning

You can use the DTrace features for determining the stability attributes of D programs that you create. After you create a D program with the appropriate stability attributes, you might also want 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 must establish version bindings for any D programs that you want to install as persistent scripts or use in layered tools. For more information about persistent scripts, see Chapter 9, "Scripting".

Versions and Releases

The D compiler labels set 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" for a major release, "x.y" for a minor release, or "x.y.z" for 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 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 the standard nomenclature for interface versions. For information about nomenclature for interface versions, see `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 99   DTrace Release Versions

Release	Version	Significance
Major	*x*.0	A Major release might contain major feature additions, adhere to different or possibly incompatible Standard revisions, and change, drop, or replace

		Standard or Stable interfaces. For more information about stability, see Chapter 18, "Stability". 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. For more information, see Chapter 18, "Stability".
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.

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: Oracle D 1.11.2
$
```

If you wamt 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, you can set the version option either on the command-line using `dtrace -x`:

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

For more information about options, see Chapter 10, "Options and Tunables".

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 "Identifier Names and Keywords" on page 40 for tips on how to pick identifier names that are unlikely to conflict with interfaces offered by future versions of DTrace.

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 Oracle 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 Oracle Solaris and are described in the corresponding version of the Oracle Solaris Dynamic Tracing Guide. You can use the `dtrace -l` option to explore the set of providers and probes available on your Oracle Solaris system. Providers label their interfaces using the DTrace stability attributes, and you can use the DTrace stability reporting features to determine whether the provider interfaces used by your D program are likely to change or be offered in future Oracle Solaris releases. For more information, see Chapter 18, "Stability".

Index

Numbers and Symbols

$ (dollar sign), 79
$$, 174

A

actions, 26, 109, 109
   data recording, 110
   default, 109
   destructive , 124
   kernel destructive, 127
   process destructive, 124
   special, 130
   speculative, 130
Aggregation, 89
   clearing, 103
   drops, 107
   functions, 89
   normalization, 100
   printing, 99
   sorting, 104
   truncating, 106
aggregations, 90
anonymous enabling, 397
anonymous tracing, 397
   examples, 398
arguments
   `profile` provider, 191
arithmetic
   expressions, 28
array
   declaration, 65
   multi-dimensional, 69
   storage, 65
arrays, 37, 62
   associative, 53

assembly
   functions, 209
avoiding errors, 378

B

bit-fields, 83
breakpoint interaction, 210
buffer
   other, 140
   policies, 137
   principal, 137
   resizing, 141
   sizes, 140
buffer policy
   `fill`, 138, 139
   `ring`, 139
   `switch`, 138
buffering, 137
buffers, 137
   principal, 137
`bufinfo_t`
   structure, 299

C

`*curlwpsinfo`, 59
`*curpsinfo`, 59
`*curthread`, 59
C preprocessor
   use, 26
cacheable predicates, 408
claiming anonymous state, 398
`clear` function, 103
compilation, 27
constant