Oracle Linux

Using DTrace for System Tracing

F76721-03
May 2025

Oracle Linux Using DTrace for System Tracing,

F76721-03

Contents

4 D Program Syntax Reference

5 DTrace Runtime and Compile-time Options Reference

6 DTrace Stability Reference

7 DTrace Built-in Variable Reference

8 DTrace Function Reference

9 DTrace Provider Reference

Preface

Oracle Linux: Using DTrace for System Tracing describes how to use DTrace, which is a powerful dynamic tracing tool based on eBPF. Most of the information in this document is generic and applies to all releases of Oracle Linux from Oracle Linux 8 onward, and for Unbreakable Enterprise Kernel Release 6 and later.

Documentation License

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners,

we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

Get Started With DTrace

Learn how to get started with DTrace, including how to install DTrace on Oracle Linux.

The topics in this section provide guidance on how to perform particular operations with DTrace and serve as an introduction to installing and using DTrace. By following steps in this guide, you can get started with DTrace immediately. After you have explored these topics, you can either review DTrace Concepts to get a better understanding of how DTrace works and how you can improve the way that you use it, or you can use the various references that are included to find out more about writing D programs that do what you need them to do.

Install DTrace

The following instructions provide steps to install DTrace on different Oracle Linux releases and to verify that the installation was successful.

Install DTrace on Oracle Linux 10

1. Enable the yum repository.

   If running on an x86 platform, enable the `ol10_UEKR8` yum repository for the system.

   ```
   sudo dnf config-manager --enable ol10_UEKR8
   ```

   > **Note:**
   >
   > Oracle releases UEK and DTrace packages in the `baseos` repository for aarch64 platforms. You don't need to enable any other repositories to access the DTrace packages for aarch64 platforms.

2. Install DTrace.

   Install the `dtrace` package.

   ```
   sudo dnf install -y dtrace
   ```

Install DTrace on Oracle Linux 9

1. Enable the yum repository.

   If running on an x86 platform, enable the `ol9_UEKR7` yum repository for the system.

   ```
   sudo dnf config-manager --enable ol9_UEKR7
   ```

> **Note:**
>
> Oracle releases UEK and DTrace packages in the `baseos` repository for aarch64 platforms. You don't need to enable any other repositories to access the DTrace packages for aarch64 platforms.

2. Install DTrace.

   Install the `dtrace` package.

   ```
   sudo dnf install -y dtrace
   ```

Install DTrace on Oracle Linux 8

1. Enable the yum repository.

   If running on an x86 platform, enable either the `ol8_UEKR6` or `ol8_UEKR7` yum repository for the system.

   For example, run:

   ```
   sudo dnf config-manager --enable ol8_UEKR7
   ```

> **Note:**
>
> Oracle releases UEK and DTrace packages in the `baseos` repository for aarch64 platforms. You don't need to enable any other repositories to access the DTrace packages for aarch64 platforms.

2. Install DTrace.

   Install the `dtrace` package.

   ```
   sudo dnf install -y dtrace
   ```

Install DTrace on Oracle Linux 7

> **WARNING:**
>
> Oracle Linux 7 is now in Extended Support. See Oracle Linux Extended Support and Oracle Open Source Support Policies for more information.
>
> Migrate applications and data to Oracle Linux 8, Oracle Linux 9, or Oracle Linux 10, as soon as possible.

1. Enable the `ol7_UEKR6` yum repository.

For example, if you have `yum-utils` installed, run:

```
sudo yum-config-manager --enable ol7_UEKR6
```

2. Install DTrace.

   Install the `dtrace` and `libdtrace-ctf` packages:

   ```
   sudo yum install -y dtrace libdtrace-ctf
   ```

Verify the DTrace Installation

Check that DTrace is installed to the correct location and verify the DTrace version.

1. Confirm DTrace is installed into `/usr/sbin/dtrace`.

   ```
   ls -lah /usr/sbin/dtrace
   ```

2. Display the DTrace version number.

   ```
   dtrace -V
   ```

   The output looks similar to:

   ```
   dtrace: Oracle D 2.0
   ```

List and Enable Probes

DTrace providers publish available probes to DTrace so that you can enable them to perform functions when they fire. You can use the `dtrace` command to list all available probes or to enable a probe.

1. List available probes.

   To list all available probes, run:

   ```
   sudo dtrace -l
   ```

   > **Note:**
   >
   > Most uses of DTrace require `root` privileges. This document assumes that you run commands with the appropriate privileges. Use the sudo command to escalate to root user privileges before you run the commands presented in this document.

   The command returns output similar to the following:

```
DTrace 2.0.0 [Pre-Release with limited functionality]
    ID   PROVIDER            MODULE                          FUNCTION NAME
     1     dtrace                                                     BEGIN
     2     dtrace                                                     END
```

```
     3     dtrace                                                     ERROR
     4        fbt           vmlinux        __traceiter_initcall_level entry
     5        fbt           vmlinux        __traceiter_initcall_level
return
     6        fbt           vmlinux        __traceiter_initcall_start entry
     7        fbt           vmlinux        __traceiter_initcall_start
return
     8        fbt           vmlinux       __traceiter_initcall_finish entry
     9        fbt           vmlinux       __traceiter_initcall_finish
return
...
144917        sdt               rtc
rtc_set_time
144918        sdt               i2c
i2c_result
144919        sdt               i2c
i2c_reply
144920        sdt               i2c
i2c_read
144921        sdt               i2c
i2c_write
144922        sdt             smbus
smbus_result
144923        sdt             smbus
smbus_reply
144924        sdt             smbus
smbus_read
144925        sdt             smbus
smbus_write
144926        sdt             hwmon
hwmon_attr_show_string
144927        sdt             hwmon
hwmon_attr_store
144928        sdt             hwmon
hwmon_attr_show
144929        sdt           thermal
thermal_zone_trip
144930        sdt           thermal
cdev_update
144931        sdt           thermal
thermal_temperature
144932        sdt            bcache
...
145763    syscall           vmlinux                          listen entry
145764    syscall           vmlinux                            bind
return
145765    syscall           vmlinux                            bind entry
145766    syscall           vmlinux                      socketpair
return
145767    syscall           vmlinux                      socketpair entry
145768    syscall           vmlinux                          socket
return
145769    syscall           vmlinux                          socket entry
```

> **Tip:**
>
> You can get a unique list of providers available for DTrace by running:
>
> ```
> sudo dtrace -l|tail -n +3|awk '{print $2}'|uniq
> ```

You can limit the list of probes to a particular provider by using the `-P` option. You can also limit to a particular module by using the `-m` option. For example:

```
sudo dtrace -l -P sdt
sudo dtrace -l -m thermal
```

2. Run `dtrace -n` to enable a named probe using the command line utility.

   You can enable any probe matching a name. Although you can specify only the name part for a probe's full name, using the full name helps to avoid unpredictable behavior:

   ```
   sudo dtrace -n dtrace:::BEGIN
   ```

   Output similar to the following is displayed:

   ```
   dtrace: description 'dtrace:::BEGIN' matched 1 probe
   CPU     ID                    FUNCTION:NAME
     2      1                           :BEGIN
   ```

   The `dtrace:::BEGIN` probe fires once when you start a new tracing request. Tabulated output shows the CPU where the probe fired, and the ID, function, and name for the probe. DTrace continues to run, waiting for other probes to fire. To exit, press `Ctrl+C`.

3. Enable several probes by chaining them together in a request.

   You can construct DTrace requests by using arbitrary numbers of probes and functions. For example, create a request using two probes by adding the `BEGIN` and `END` probes.

   Type the following command, and then press `Ctrl+C` in the shell again, after you see the line of output for the `BEGIN` probe:

   ```
   sudo dtrace -n dtrace:::BEGIN -n dtrace:::END
   ```

   The output looks similar to:

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

   The `dtrace:::BEGIN` probe fires when the tracing request starts. DTrace waits for further probes to activate until you press `Ctrl+C` to exit. The `dtrace:::END` probe activates once when tracing completes. The `dtrace` command reports the probe firing before exiting.

4. Enable all probes for a function by using the -f option, or use the -m option to enable all probes for a module.

   You can match and enable probes for functions or for whole modules. For example, to enable both the entry and return probes for the `syscall:vmlinux:socket` function, run:

   ```
   sudo dtrace -f syscall:vmlinux:socket
   ```

   You can also enable probes for an entire module. For example, to enable all probes for the `sdt:tcp` module, run:

   ```
   sudo dtrace -m sdt:tcp
   ```

Create a DTrace Script

Learn how to create a DTrace script to develop understanding of the D Programming language.

Ensure that DTrace is installed on the system and that you can list and enable probes. See Install DTrace and List and Enable Probes.

This tutorial provides successive steps toward developing a DTrace script that you can use on a system to gather useful information. You can use this tutorial as a framework to create other scripts for DTrace, in future.

1. In a text editor, create a file named `hello.d` and write a DTrace clause to fire for the `dtrace:::BEGIN` probe.

   Enter the following text into the editor:

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

   Save the file.

2. Run the `hello.d` program by using the `dtrace -s` command.

   ```
   sudo dtrace -s hello.d
   ```

   Output similar to the following is displayed:

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

   Note that you didn't have to press `Ctrl+C` to exit because you specified the `exit` function for the `BEGIN` probe in the program.

3. Open `hello.d` in the text editor and add an interpreter line to the beginning of the script.

Edit the file and add the following line of text to the top of the file:

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

The complete script follows:

```
#!/usr/sbin/dtrace -s
dtrace:::BEGIN
{
  trace("hello, world");
  exit(0);
}
```

Save the file.

4. Change the permissions on the `hello.d` file to make it executable.

   Run the `chmod` command to update the file permissions:

   ```
   chmod a+rx hello.d
   ```

5. Run the new executable script file.

   Use the `sudo` command so that the DTrace script still runs with root privileges so that it can access all DTrace features:

   ```
   sudo ./hello.d
   ```

   Note that by including an interpreter line at the beginning of the program, you can run the script without even specifying the `dtrace` command.

6. Change the script to use an external macro variable.

   Edit the file to greet a person by name, when you specify a name as an argument to the script:

   ```
   #!/usr/sbin/dtrace -s
   dtrace:::BEGIN
   {
     printf("hello, %s", $$1);
     exit(0);
   }
   ```

   Notice how the `trace` function is now replaced with the `printf()` function, which lets you insert the macro variable `$1` into the string by using variable substitution. The `$$` syntax is used when referencing the macro variable, to express it as a string value.

7. Run the script to see how the modification has altered behavior.

   Run the script as before, using the command:

   ```
   sudo ./hello.d
   ```

An error similar to the following is generated.

```
dtrace: failed to compile script ./hello.d: line 4: macro argument $$1 is
not defined
```

The error is generated because the script now expects you to provide another argument when you run it. Try to run the script again, this time specifying a name:

```
sudo ./hello.d bob
```

The script returns output similar to the following:

```
dtrace: script './hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  3      1                           :BEGIN hello, bob
```

8. Change the script to use a pragma statement.

   To reduce how verbose the script is and to limit output to only what's functionally returned by the clause, add a pragma statement to set the runtime `quiet` option. Edit the script to add the pragma statement, as follows:

```
#!/usr/sbin/dtrace -s
 #pragma D option quiet
 dtrace:::BEGIN
 {
   printf("hello, %s", $$1);
   exit(0);
 }
```

9. Run the script to see how the modification has altered behavior.

   Run the script as before, using the command:

```
sudo ./hello.d sally
```

   The script output is reduced to only what's returned by the `printf()` function.

10. Change the script to use a predicate to control when to process the clause.

    You can use a predicate to control the script so that it only runs when a certain condition is true. Edit the script to add a predicate line to evaluate whether the string value of the macro variable is equal to '*bob*', as follows:

```
#!/usr/sbin/dtrace -s
 #pragma D option quiet

 dtrace:::BEGIN
 /$$1=="bob"/
 {
   printf("hello, %s", $$1);
   exit(0);
 }
```

11. Run the script to see how the modification has altered behavior.

Run the script as before, using the command:

```
sudo ./hello.d sally
```

The script doesn't exit and you need to press `Ctrl+C` to force quit the process. This is because the `exit()` function is part of the clause that evaluates whether the first argument of the script is equal to '*bob*'. Try running the script again, using *bob* as the argument.

```
sudo ./hello.d bob
```

The script runs as before, illustrating that the predicate is working.

Use Predicates For Control Flow

For runtime safety, 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 change control flow using logical expressions called *predicates*. This tutorial shows how to use predicates to control D programs.

To illustrate predicates at work, you can create a D program that implements a 10-second countdown timer. When the program runs, it counts down from 10 and then prints a message and exits. The program uses a variable and predicates to evaluate how much time has passed and what to print.

1. Design a logical flow for the program.

   Consider designing the logical flow for a program before trying to write the program itself. When the flow is clearly defined, it's possible to transform conditional constructs into separate clauses and predicates. The logical flow for the program might look as follows:

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

   By creating two clauses with the same probe description but different predicates and functions it's possible to achieve the required logical flow for this program.

2. Write the program code using predicates to decide whether the functions for the specified probe description are permitted to run or not when the probe fires.

   The program source code follows. Copy this 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);
}
```

3. Run the program.

```
sudo dtrace -s countdown.d
```

Output similar to the following is displayed:

```
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0    638                       :tick-1sec        10
  0    638                       :tick-1sec         9
  0    638                       :tick-1sec         8
  0    638                       :tick-1sec         7
  0    638                       :tick-1sec         6
  0    638                       :tick-1sec         5
  0    638                       :tick-1sec         4
  0    638                       :tick-1sec         3
  0    638                       :tick-1sec         2
  0    638                       :tick-1sec         1
  0    638                       :tick-1sec   blastoff!
#
```

This tutorial uses the `BEGIN` probe to initialize a variable integer `i` to 10 to begin the countdown. Next, the program uses the `tick-1sec` probe to implement a timer that fires once every second. Notice that in `countdown.d`, the `tick-1sec` probe description is used in two different clauses, each with a different predicate and function 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. If `i` isn't 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);
}
```

The second clause uses the `trace` function on a sequence of characters inside double quotes, called a *string constant*, to print a final message when the countdown is complete. The `exit` function is then used to end all tracing and to perform any remaining tasks such as consuming the final data, printing aggregations (as needed), and performing cleanup before returning to the shell prompt.

Example 1-1 How to use a predicate to monitor system calls for a process ID

You can create a D Program to trace system calls for a process ID, by using a predicate to limit the default tracing function to match the process ID that you want to trace.

```
syscall:::entry
/pid == 2860/
{
}
```

Note that in this example, the built-in variable `pid` is evaluated to match a particular ID, `2860` in this example. You could further change this script to take advantage of shell macro variables, so that it becomes more extensible and can be run for any process ID at runtime. Edit the script as follows and save it to a file called `strace.ds`:

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

syscall:::entry
/pid == $1/
{
}
```

Change the file mode to make it executable:

```
sudo chmod +x strace.ds
```

Now you can use this script to monitor all the system calls made by any process on the system. For example, you could run the script to monitor system calls made by the cron daemon:

```
sudo ./strace.ds $(pidof /usr/sbin/crond)
```

2
DTrace Concepts

Explore DTrace at a conceptual level and understand DTrace components and terminology.

The topics in this section are general and can help you to understand what DTrace is and how it works.

About DTrace

DTrace is a powerful tracing tool that's available in Oracle Linux for use with the Unbreakable Enterprise Kernel (UEK). DTrace has low overhead and is safe to use on production systems to analyze what a system is doing in real time.

DTrace lets you examine the behavior of user programs and the OS, to understand how the system works, to track down performance problems, and to find the causes of aberrant behavior. DTrace can collect or print stack traces, function arguments, timestamps, and statistical aggregates by using probes that can be runtime events or source-code locations.

Unlike many tracing tools, DTrace is fully programmable. You can collect data for one event and store it for use when another event is triggered. You can select what information you want to gather and how to report it. DTrace programs have a familiar syntax that draws on the C programming language.

This implementation of DTrace uses existing Linux kernel tracing facilities, such as eBPF, which didn't exist when DTrace was first ported to Linux. The new implementation removes DTrace dependencies on specialized kernel patches, but retains syntax compatibility with earlier implementations of DTrace to deliver a mature tracing tool based on modern technology. Furthermore, this implementation also maintains functional compatibility with earlier implementations of DTrace, so that you can perform the same actions using either version of DTrace.

This implementation is a user space application and is available on:

- Unbreakable Enterprise Kernel 8 (UEK 8) and later kernels on Oracle Linux 10.
- Unbreakable Enterprise Kernel Release 7 (UEK R7) and later kernels on Oracle Linux 9.
- Unbreakable Enterprise Kernel Release 6 (UEK R6) and later kernels on Oracle Linux 8.

DTrace is also available on Unbreakable Enterprise Kernel Release 6 (UEK R6) and later kernels on Oracle Linux 7, and requires the `libdtrace-ctf` library to run. The functionality of the `libdtrace-ctf` library is integrated into the Oracle Linux GNU tool chain for later Oracle Linux releases. Oracle Linux 7 is in Extended Support. Migrate applications and data to Oracle Linux 8, Oracle Linux 9, or Oracle Linux 10, as soon as possible.

DTrace is developed as an open source project available under the Universal Permissive License (UPL), Version 1.0. You can access source code and more information at https://github.com/oracle/dtrace-utils.

DTrace Components and Terminology

Learn about the different components and the terms used to describe them within the DTrace framework.

DTrace is a framework that dynamically traces data into buffers that are read by the `dtrace` command line utility. The `dtrace` command line utility can run programs that can implement certain functions by compiling D programs to generate eBPF code that's loaded into the kernel. In practice, all interaction with DTrace is performed by using the `dtrace` command line utility. See Install DTrace for information on how to install the command line utility.

Probes

DTrace works by using *probes* that identify particular instrumentation in the kernel or within a user space application, or which can be used to identify interval counters or performance event counters. Events such as when particular code is run or when a specific counter is incremented cause a probe to fire and DTrace can perform functions that are bound to the event in a program or script. For example, a probe can fire when a particular file is opened and a DTrace program can print information related to the event that can be useful for debugging or resolving an issue. Equally, at the moment that DTrace starts or ends any tracing activity, the `BEGIN` and `END` probes dedicated to these actions always fire.

You can list all the available probes on a system by typing the following command:

```
sudo dtrace -l
```

Output is displayed to show each of the different values that are used to reference a probe correctly:

```
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4    syscall          vmlinux                               read entry
    5    syscall          vmlinux                               read return
    6    syscall          vmlinux                              write entry
    7    syscall          vmlinux                              write return
   ...
```

See List and Enable Probes for more information on how to list and enable specific probes.

Probes are made available by *providers*, which group particular kinds of instrumentation together. If a provider is related to source code, its probes might also include information about the piece of code that the probe relates to in a *module* and a *function* identifier. Therefore, a probe is identified by a *probe description*, grouped into four fields:

provider
The name of the DTrace provider that the probe belongs to.

module
If the probe corresponds to a specific program location, the name of the kernel module, library, or user-space program in which the probe is found. Some probes might be associated with a module name that isn't tied to a particular source location in cases where they relate to more abstract tracepoints.

function
If the probe corresponds to a specific program location, the name of the program function in which the probe is found.

name
The name that provides some idea of the probe's semantic meaning, such as `BEGIN` or `END`.

When referencing a probe, write all four parts of the probe description separated by colons:

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

Some probes don't have a module or function identifier when they're listed. When providing the complete probe description for these probes, you must still include the empty fields:

```
dtrace:::BEGIN
```

Probes aren't required to have a module and function. The dtrace `BEGIN`, `END` and `ERROR` probes are good examples of this because these probes don't correspond to any specific instrumented program function or location. Instead, these probes are used for more abstract concepts, such as the idea of the end a tracing request. Other probes, such as those made available by the Profile Provider or the CPC Provider, also don't include module or function identifiers in their descriptions.

D Programs

You can bind a set of processing instructions called statements to one or more DTrace probes, so that when a probe fires, the specified statements are run to perform some required functionality. The set of enabled probes, the statements, and any conditions that might be evaluated when the probe fires, can all be collated into a *D program*.

A program can consist of several probe descriptions that decide which probes can trigger some functionality within the D program. Probe descriptions are followed by a set of processing instructions, called a *clause*, that describes what to do when the selected probe fires. Conditional expressions, called *predicates*, can be inserted between the probe descriptions and the clause to control the conditions under which the actions within the clause are run. For example, a program might be designed to fire for all system calls and to count these for a particular application. The program would consist of a probe description for the `syscall:::entry` probe, a predicate to limit processing to match either a process ID or the name of an executable, and a clause that performed the `count()` function to gather information about each system call function. The resulting D program might be:

```
/* Probe descriptions */
syscall:::entry
/* Predicate */
/execname=='date'/
/* Clause */
{
@reads[probefunc]=count();
}
```

When the script is run it shows each system call that's made by the `date` command and provides the count value for each, as follows:

```
dtrace: description 'syscall:::entry ' matched 344 probes
Wed 22 Feb 11:54:51 GMT 2023

  exit_group                                                    1
  lseek                                                         1
```

```
mmap                                                  1
write                                                 1
openat                                                2
read                                                  2
brk                                                   3
close                                                 4
newfstat                                              4
```

The program probe description matches all system call functions at the entry point. The program predicate evaluates a *built-in variable*, `execname`, against a string using an operator. The clause includes an *aggregation*, `@reads`, that's used to gather data about the firing probe. In this case, the aggregation stores a counter that increments every time the probe fires and the predicate resolves. The counter is implemented by the `count()` *function* and stores count values for each system call probe function. See D Program Syntax Reference for more information on program structure and syntax.

Aggregations

Aggregations can be used to reduce large bodies of data to smaller, meaningful statistical metrics. Many common functions that are used to understand a set of data are aggregating functions. These functions include the following:

- Counting the number of elements in the set.
- Computing the minimum value of the set.
- Computing the maximum value of the set.
- Summing all the elements in the set.
- Creating a histogram of the values in the set, as quantized into certain bins.

Although you could code an application to calculate an aggregation for a set of data, when many probes are firing concurrently, they can overwrite each other's updates to the aggregating variable or the calculation can become a serial bottleneck.

DTrace aggregation functions apply to the data as it's traced, so that the dataset doesn't need to be stored and the aggregation is always available as events occur. In this way, aggregation functions are more efficient and exact, and avoid overwrites. See Aggregations for more information.

Speculation

While predicates can be used to filter out uninteresting events, they're only useful if you already know which events you need to filter. Because DTrace is often used to help debug particular system behaviors, DTrace includes a set of *speculation* functions that can be used to trace data speculatively.

Speculation is used to trace quantities temporarily until particular information is known, at which case the data can be discarded or committed. By performing speculative tracing you can trace data until you know whether it's useful. For example, to trace data about events that might trigger a particular return code or error, you could speculatively trace all events and discard the trace data if it doesn't match the return code that you're interested in. See Speculation for more information.

Buffers

As DTrace probes fire, the kernel writes data into various *buffers* that are read by the `dtrace` user-space utility, which prints requested data.

The generation of trace data by the kernel and the processing of that data by the `dtrace` utility operate asynchronously. The processing of the trace data can be tuned by setting buffer options and refresh rates. Buffer sizes can be tuned with options such as `aggsize`, `bufsize`, and `nspec`.

The various options that control buffer sizing and policies are described in DTrace Runtime and Compile-time Options Reference.

Stability

DTrace is a tracing tool that takes advantage of the probes that are included in code that can change over time. DTrace and the D compiler include features to dynamically compute and describe the *stability* of the D programs that you create. You can use these DTrace stability features to inform you of the stability attributes of D programs or to produce compile-time errors when a program has inappropriate interface dependencies.

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 helps you to assess risk when developing scripts and tools that are based on DTrace by indicating how likely an interface or DTrace entity might change in a future release or patch. The DTrace dependency class indicates whether an interface is common to all Oracle Linux platforms and processors or whether it's associated with a particular architecture. The two types of attributes that are used to describe interfaces can vary independently.

Applications that depend only on stable interfaces are likely to continue to function reliably on future minor releases and are unlikely to be broken by interim patches. Less stable interfaces can be used for experimentation, prototyping, tuning, and debugging on the current system. Use less stable with the understanding that they might change and become incompatible or even be dropped or replaced with alternatives in future minor releases.

Interfaces can be common to all Oracle Linux platforms and processors or might be associated with a particular system architecture. Dependency classes help indicate architecture dependencies and are orthogonal to stability levels. For example, a DTrace interface can be stable, but only available on x86_64 microprocessors. Or, the interface can be unstable, but common to all Oracle Linux platforms.

See DTrace Stability Reference for more information about the different stability levels and dependency classes.

3
DTrace Command Reference

The `dtrace` command is a generic front-end utility for the DTrace facility. The command implements an interface to invoke the D language compiler. The `dtrace` command can also retrieve buffered trace data from the DTrace kernel facility and includes a set of basic routines to format and print traced data.

About the dtrace Command

The `dtrace` command provides a generic interface to all the essential services that are provided by the DTrace facility.

The `dtrace` command includes options to do the following:

- List the set of probes and providers published by DTrace.
- Enable probes directly by using any of the probe description specifiers (provider, module, function, name).
- Run the D compiler and compile one or more D program files or programs written directly on the command line.
- Generate program stability reports.
- Change DTrace tracing and buffering behavior and enable extra D compiler features.

You can also use the `dtrace` command to create D scripts by using the command in a `#!` declaration to create an interpreter file. Finally, you can use the `-e` option to `dtrace` to compile D programs and find their properties without enabling any tracing.

dtrace Command Options

The `dtrace` command accepts the following options:

```
dtrace [-32 | -64] [-CeFGHhlqSvVwZ] [-b bufsz] [-c cmd] [-D name[=value]]
     [-I path] [-L path] [-o output] [-p pid] [-s script] [-U name]
     [-x opt[=val]] [-X a | c | s | t]
     [-P provider [[predicate] action]]
     [-m [provider:] module [[predicate] action]]
     [-f [[provider:] module:] function [[predicate] action]]
     [-n [[[provider:] module:] function:] name [[predicate] action]]
     [-i probe-id [[predicate] action]]
```

where *`predicate`* is any D predicate inside slashes `//` and *`action`* is any D statement list inside braces `{}`, according to the D language syntax.

The arguments accepted by the `-P`, `-m`, `-f`, `-n`, and `-i` options can include an optional D language *predicate* inside slashes `//` and optional D language *action* statement list inside braces `{}`. D program code specified on the command line must be appropriately quoted to avoid interpretation of metacharacters by the shell.

The following options are available:

-32 | -64
Sets whether to generate 32-bit or 64-bit D programs and ELF files. This option isn't usually required as `dtrace` selects the native data model as the default.

-b *bufsz*
Set principal trace buffer size (*bufsz*). The trace buffer size can include any of the size suffixes `k`, `m`, `g`, or `t`. If the buffer space can't be allocated, `dtrace` tries to reduce the buffer size or exit depending on the setting of the `bufresize` property.

-c *cmd*
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 ends. 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.

-C
Run the C preprocessor (`cpp`) over D programs before compiling them. You can pass options to the C preprocessor using the `-D`, `-U`, `-I`, and `-H` options. You can select the degree of C standard conformance if you use the `-X` option. For a description of the set of tokens defined by the D compiler when invoking the C preprocessor, see `-X`.

-D *name*[=*value*]
Define *name* when invoking cpp (enabled using the `-C` option). If you specify the equals sign (`=`) and optional *value*, the name is assigned the corresponding value. This option passes the `-D` option to each `cpp` invocation.

-e
Exit after compiling any requests, but before enabling any probes. You can combine this option with D compiler options. This combination verifies that the programs compile without executing them and enabling the corresponding instrumentation.

-f [[*provider*:]*module*:]*function*[[*predicate*]*action*]]
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 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. You can specify more than one `-f` option 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 `<-`. System call entry probe reports are indented and their output is prefixed with `=>`. System call 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 a relocatable ELF object which can be linked into another program. If the `-o` option is present, the ELF file is saved using the path name specified as the argument for this operand. If the `-o` option isn't present and the DTrace program is contained

with a file whose name is *filename*.d, then the ELF file is saved using the name *filename*.o. Otherwise the ELF file is saved using the name d.out.

-H
Print the path names of included files when invoking cpp (enabled using the -C option). This option passes the -H option to each cpp invocation, causing it to display the list of path names, one for each line, to stderr.

-h
Generate a header file containing macros that correspond to probes in the specified provider definitions. This option can generate a header file that's included by other source files for later use with the -G option. If the -o option is present, the header file is saved using the path name specified as the argument for that option. If the -o option isn't present and the DTrace program is contained with a file whose name is *filename*.d, then the header file is saved using the name *filename*.h.

-i *probe-id* [[*predicate*] *action*]
Specify probe identifier (*probe-id*) to trace or list (-l option). You can specify probe IDs using decimal integers as shown by dtrace -l. The -i argument can be suffixed with an optional D probe clause. You can specify more than one -i option at a time.

-I *path*
Add the specified directory *path* to the search path for #include files when invoking cpp (enabled using the -C option). This option passes the -I option to each cpp invocation. The specified *path* 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 provided using the -P, -m, -f, -n, -i, and -s options. If none of these options are specified, this option lists all probes.

-L
Add the specified directory *path* to the search path for DTrace libraries. DTrace libraries are used to contain common definitions that can be used when writing D programs. The specified *path* is added after the default library search path. If it exists, a subdirectory of *path* named after the minor version of the running kernel (for example, 3.8) is searched immediately before *path*. Dependency analysis is performed only within each directory, not across directories.

-m [[*provider:*] *module:* [[*predicate*] *action*]]
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 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 can be specified on the command line at a time.

-n [[[*provider:*] *module:*] *function:*] *name* [[*predicate*] *action*]
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 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 can be specified on the command line at a time.

-o *output*
Specify the *output* file for the -G, -h, and -l options, or for the traced data itself. If the -G option is present and the -s option's argument is of the form *filename*.d and -o isn't present, the default output file is *filename*.o. Otherwise the default output file is d.out.

-p *pid*
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 ends. 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.

-P *provider*[[*predicate*]*action*]
Specify provider name to trace or list (-l option). The remaining probe description fields module, function, and name are blank and match any probes regardless of the values in those fields. The -P argument can be suffixed with an optional D probe clause. You can specify more than one -P option on the command line at a time.

-q
Set quiet mode. dtrace suppresses messages such as the number of probes matched by the specified options and D programs and doesn't 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() is displayed to stdout.

-s
Compile the specified D program source file. If the -e option is present, the program is compiled but instrumentation isn't enabled. If the -l option is present, the program is compiled and the set of probes matched by it, is listed but instrumentation isn't enabled. If none of -e, -l, or -G are present, the instrumentation specified by the D program is enabled and tracing begins.

-S
Show D compiler intermediate code. The D compiler produces a report of the intermediate code generated for each D program to stderr.

-U *name*
Undefine the specified *name* when invoking cpp (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.

-V
Report the highest D programming interface for dtrace. The version information is printed to stdout and the dtrace command exits. When used with -v, also reports information on the version of the dtrace and associated library.

-w
Permit destructive actions in D programs specified using the -s, -P, -m, -f, -n, or -i options. If the -w option isn't specified, dtrace doesn't permit the compilation or enabling of a D program that contains destructive actions.

-x *opt[=val]*
Enable or change a DTrace runtime option or D compiler option. Boolean options are enabled by specifying their name. Options with values are set by separating the option name and value with an equals sign (=). See DTrace Runtime and Compile-time Options Reference.

-X a | c | s | t
Sets the ISO C conformance settings for the preprocessor. The options are:

- `a | c | t`: Any of these options sets the conformance to `-std=c99`. This is the default.
- `s`: This option sets the conformance to `-traditional-cpp`.

-Z
Permit probe descriptions that match zero probes. If the `-Z` option isn't specified, `dtrace` reports an error and exits 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 don't match any known probes.

dtrace Command Operands

You can specify zero or more extra arguments on the `dtrace` command line to define a set of macro variables, such as `$1`, `$2`, and so on, to be used in any D programs that are specified with the `-s` option or on the command line.

dtrace Command Exit Status

The following exit values are returned by the `dtrace` command:

0
Indicates that 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 an anonymous state was successfully retrieved. The `dtrace` command returns `0` even if the specified tracing requests meet errors or drops.

1
Indicates that a fatal error occurred. For D program requests, the `1` exit status indicates that program compilation failed or that the specified request couldn't be satisfied.

2
Indicates that invalid command line options or arguments were specified.

4

D Program Syntax Reference

This reference describes how to write D programs that can be used with DTrace to enable probes and perform operations.

Program Structure

A D program consists of a set of clauses that describe the probes to enable, an optional predicate that controls when to run, and one or more statements that often describe some functionality to implement when the probe fires.

D programs can also contain declarations of variables and definitions of new types. A probe clause declaration uses the following structure:

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

Probe Descriptions
Probe descriptions ideally express the full description for a probe and take the form:

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

The field descriptors are defined as follows:

provider
The name of the DTrace provider that the probe belongs to.

module
If the probe corresponds to a specific program location, the name of the kernel module, library, or user-space program in which the probe is found. Some probes might be associated with a module name that isn't tied to a particular source location in cases where they relate to more abstract tracepoints.

function
If the probe corresponds to a specific program location, the name of the program function in which the probe is found.

name
The name that provides some idea of the probe's semantic meaning, such as `BEGIN` or `END`.

DTrace recognizes a form of shorthand when referencing probes. By convention, if you don't specify all the fields of a probe description, DTrace can match a request to all the probes with matching values in the parts of the name that you do specify. For example, you can reference the probe name `BEGIN` in a script to match *any* probe with the name field `BEGIN`, regardless of

the value of the provider, module, and function fields. For example, you might see a probe referenced as:

```
BEGIN
```

If a probe is referenced in a D program and it doesn't use a full probe description, the fields are interpreted based on an order of precedence:

- A single component matches the probe name, expressed as:

  name

- Two components match the function and probe name, expressed as:

  function:*name*

- Three components match the module, function, and probe name

  module:*function*:*name*

Although probes can also be referenced by their ID, this value can change over time. The number of probes on the system doesn't directly correlate to the ID, because new provider modules can be loaded at any time and some providers also offer the ability to create new probes on-the-fly. Avoid using the numerical probe ID to reference a probe.
Probe descriptions also support a pattern-matching syntax similar to the shell *globbing* pattern matching syntax that's described in the `sh(1)` manual page. For example, you can use the asterisk symbol (`*`) to perform a wildcard match, as in the following description:

```
sdt:::tcp*
```

If any fields are blank in the probe description, a wildcard match is performed on that field. Unless matching several probes intentionally, specifying the full probe description to avoid unpredictable results is better practice.

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

To successfully match and enable a probe, the complete probe description must match on every field. A probe description field that isn't a pattern must exactly match the corresponding field of the probe. Note that a description field that's empty matches any probe.

Several probes can be included in a comma-separated list. By including several probes in the description, the same predicate, and function sequences are applied when each probe is activated.

Predicates
Predicates are expressions that appear between a pair of slashes (`//`) that are then evaluated at probe firing time to decide whether the associated functions must be processed. Predicates are the primary conditional construct that are 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 so that the functions are always processed when the probe is activated.
Predicate expressions can use any of the D operators and can include 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.

Statements
Statements are described by a list of expressions or functions that are separated by semicolons (`;`) and within braces (`{}`). An empty set of braces with no statements included causes the default action to be processed. The Default Action reports the probe activation.

A program can consist of several probe-clause declarations. Clauses run in program order.

A program can be stored on the file system and can be run by the DTrace utility. You can transform a program into an executable script by prepending the file with an interpreter directive that calls the `dtrace` command along with any required options, as a single argument, to run the program. See the `sh(1)` manual page for more information on adding the interpreter line to the beginning of a script. The interpreter directive might look as follows:

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

A script can also include D pragma directives to set runtime and compiler options. See DTrace Runtime and Compile-time Options Reference for more information on including this information in a script.

Types, Operators, and Expressions

D provides the ability to access and manipulate various data objects: variables and data structures can be created and changed, data objects that are defined in the OS 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 section describes the detailed set of rules for types, operators, and expressions.

Identifier Names and Keywords

D identifier names are composed of uppercase and lowercase 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. Avoid using these names in D programs. By convention, D programmers typically use mixed-case names for variables and all uppercase names for constants.

D language keywords are special identifiers that are reserved for use in the programming language syntax itself. These names are always specified in lowercase and must not be used for the names of D variables. The following table lists the keywords that are reserved for use by the D language.

Table 4-2 D Keywords

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

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 produces a syntax error if you try to use a keyword that's reserved for future use. The keywords that are 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. Note that 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 only be performed on integers in D programs. Floating-point constants can be used to initialize data structures, but floating-point arithmetic isn't permitted in D. D provides a 64-bit data model for use in writing programs.

The names of the integer types and their sizes in the 64-bit data model are shown in the following table. Integers are always represented in twos-complement form in the native byte-encoding order of a system.

Table 4-3 D Integer Data Types

Type Name	64-bit Size
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
long long	8 bytes

Integer types, including `char`, can be prefixed with the signed or unsigned qualifier. Integers are implicitly signed unless the unsigned qualifier isn't specified. The D compiler also provides the type aliases that are listed in the following table.

Table 4-4 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 listed in the previous table and are appropriately defined for each data model. For example, the `uint8_t` type name is an alias for the type unsigned `char`.

> **Note:**
>
> The predefined type aliases can't be used in files that are included by the preprocessor.

D provides floating-point types for compatibility with ANSI C declarations and types. Floating-point operators aren't available in D, but floating-point data objects can be traced and formatted with the `printf` function. You can use the floating-point types that are listed in the following table.

Table 4-5 D Floating-Point Data Types

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

D also provides the special type `string` to represent ASCII strings. Strings are discussed in more detail in DTrace String Processing.

Constants

Integer constants can be written in decimal (`12345`), octal (`012345`), or hexadecimal (`0x12345`) format. 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 suffixes listed in the following table to any integer constant to explicitly specify its D type.

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

Suffix	D type
`f` or `F`	`float`
`l` or `L`	`long double`

Character constants are written as a single character or escape sequence that's inside a pair of single quotes (`'a'`). Character constants are assigned the `int` type rather than `char` and are equivalent to an integer constant with a value that's determined by that character's value in the ASCII character set. See the `ascii(7)` manual page for a list of characters and their values. You can also use any of the special escape sequences that are listed in the following table. D uses the same escape sequences as those found in ANSI C.

Table 4-6 Character Escape Sequences

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

You can include more than one character specifier inside single quotes to create integers with individual bytes that are initialized according to the corresponding character specifiers. The bytes are read left-to-right from a character constant and assigned to the resulting integer in the order corresponding to the native endianness of the 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 can't contain a literal newline character. To create strings

containing newlines, use the `\n` escape sequence instead of a literal newline. String constants can contain any of the special character escape sequences that are shown for character constants before. Similar to ANSI C, strings are represented as arrays of characters that end with a null character (`\0`) that's implicitly added to each string constant 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.

Arithmetic Operators

Binary arithmetic operators are described in the following table. These operators all have the same meaning for integers that they do in ANSI C.

Table 4-7 Binary Arithmetic Operators

Operator	Description
`+`	Integer addition
`-`	Integer subtraction
`*`	Integer multiplication
`/`	Integer division
`%`	Integer modulus

Arithmetic in D can only be performed on integer operands or on pointers. Arithmetic can't be performed on floating-point operands in D programs. The DTrace execution environment doesn't take any action on integer overflow or underflow. You must check for these conditions in situations where overflow and underflow can occur.

However, 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 contains an invalid division operation that's detectable at compile time, a compile error is returned and the compilation fails. If the invalid division operation takes place at run time, processing of the current clause is quit, and the `ERROR` probe is activated. If the D program has no clause for the `ERROR` probe, the error is printed and tracing continues. Otherwise, the actions in the clause assigned to the `ERROR` probe are processed. Errors that are detected by DTrace have no effect on other DTrace users or on the OS kernel. You therefore don't need to be concerned about causing any damage if a D program inadvertently contains one of these errors.

In addition to these binary operators, the `+` and `-` operators can also be used as unary operators, and these operators have higher precedence than any of the binary arithmetic operators. The order of precedence and associativity properties for all D operators is presented in Operator Precedence. You can control precedence by grouping expressions in parentheses (`()`).

Relational Operators

Binary relational operators are described in the following table. These operators all have the same meaning that they do in ANSI C.

Table 4-8 D Relational Operators

Operator	Description
`<`	Left-hand operand is less than right-operand

Table 4-8 (Cont.) D Relational Operators

Operator	Description
<=	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 isn't equal to right-hand operand

Relational operators are most often 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's `false`.

Relational operators can 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. The following table shows some example D string comparisons and their results.

D string comparison	Result
`"coffee" < "espresso"`	Returns 1 (`true`)
`"coffee" == "coffee"`	Returns 1 (`true`)
`"coffee"" >= "mocha"`	Returns 0 (`false`)

Relational operators can also be used to compare a data object associated with an enumeration type with any of the enumerator tags defined by the enumeration.

Logical Operators

Binary logical operators are listed in the following table. The first two operators are equivalent to the corresponding ANSI C operators.

Table 4-9 D Logical Operators

Operator	Description
&&	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 often used in writing D predicates. The logical `AND` operator performs the following short-circuit evaluation: if the left-hand operand is false, the right-hand expression isn't evaluated. The logical `OR` operator also performs the following short-circuit

evaluation: if the left-hand operand is true, the right-hand expression isn't evaluated. The logical `XOR` operator doesn't short-circuit. Both expression operands are always evaluated.

In addition to the binary logical operators, the unary `!` operator can 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 the expressions are equivalent.

The logical operators can 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 bitwise operators that are listed in the following table for manipulating individual bits inside integer operands. These operators all have the same meaning as in ANSI C.

Table 4-10 D Bitwise Operators

Operator	Description
~	Unary operator that can 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
&	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 shift operators are used to move bits left or right in a particular 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 produces an error message if the compiler can detect this condition when you compile the D program.

Assignment Operators

Binary assignment operators are listed in the following table. You can only modify D variables and arrays. Kernel data objects and constants can not be modified using the D assignment operators. The assignment operators have the same meaning as they do in ANSI C.

Table 4-11 D Assignment Operators

Operator	Description
=	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 that were described earlier. For example, the expression `x = x + 1` is equivalent to the expression `x += 1`. These assignment operators adhere to 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 thus 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 they do in ANSI C. These operators can be applied to variables and to the individual elements of a struct, union, or array. The operators can be applied either before or after the variable name. If the operator appears before the variable name, the variable is first changed and then the resulting expression is

equal to the new value of the variable. For example, the following two code fragments produce identical results:

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

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

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

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

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

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

To use the increment and decrement operators on elements of an array or struct, place the operator after or before the full reference to the element:

```
int foo[5];
struct { int a; } bar;

bar.a++;
foo[1]++;
--foo[1];
```

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 that's referenced by the pointer.

Conditional Expressions

D doesn't provide the facility to use `if-then-else` constructs. Instead, conditional expressions, by using the ternary operator (`?:`), can be used to approximate some of this functionality. The ternary operator associates a triplet of expressions, 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 the previous example, the expression `i == 0` is first evaluated to determine whether it's true or false. If the expression is true, the second expression is evaluated and its value is returned. If the expression is false, the third expression is evaluated and its value is returned.

As with any D operator, you can use several `?:` 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-f`, or `A-F`, 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 <= 'f') ? c + 10 -
'a' : c + 10 - 'A';
```

To be evaluated for its truth value, the first expression that's used with `?:` must be a pointer or integer. The second and third expressions can be of any compatible types. You can't construct a conditional expression where, for example, one path returns a string and another path returns an integer. The second and third expressions must be true expressions that have a value. Therefore, data reporting functions can't be used in these expressions because those functions don't return a value. To conditionally trace data, use a predicate instead.

Type Conversions

When expressions are constructed by using operands of different but compatible types, type conversions are performed 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*.

Each integer type is ranked in the order `char`, `short`, `int`, `long`, `long long`, with the corresponding unsigned types assigned a rank higher than 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 with the lower rank is first *promoted* to the type of the higher rank. Promotion doesn't change the value of the operand: it only 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 that's determined by the conversion.

Integers and other types can also be explicitly *cast* from one type to another. Pointers and integers can be cast to any integer or pointer types, but not to other types.

An integer or pointer cast is formed using an expression such as the following:

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

In this example, the destination type is within 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 doesn't include floating-point arithmetic, no floating-point operand conversion or casting is permitted and no rules for implicit floating-point conversion are defined.

Operator Precedence

D includes complex rules for operator precedence and associativity. The rules provide precise compatibility with the ANSI C operator precedence rules. The entries in the following table are in order from highest precedence to lowest precedence.

Table 4-12 D Operator Precedence and Associativity

Operators	Associativity
`() [] -> .`	Left to right
`! ~ ++ -- + - * & (`*type*`) sizeof stringof offsetof xlate`	Right to left (Note that these are the unary operators)
`* / %`	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 comma (`,`) operator that's listed in the table is for compatibility with the ANSI C comma operator. It 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 for compatibility with C and usage isn't recommended.

The `()` entry listed in the table of operator precedence represents a function call. A comma is also used in D to list arguments to functions and to form lists of associative array keys. Note that this comma isn't the same as the comma operator and doesn't guarantee left-to-right evaluation. The D compiler provides no guarantee regarding the order of evaluation of arguments to a function or keys to an associative array. Be careful of using expressions with interacting side-effects, such as the pair of expressions `i` and `i++`, in these contexts.

The `[]` entry listed in the table of operator precedence represents an array or associative array reference. Note that aggregations are also treated as associative arrays. The `[]` operator can also be used to index into fixed-size C arrays.

The following table provides further explanation for the function of several miscellaneous operators that are provided by the D language.

Operators	Description
`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.

Operators	Description
unary `&`	Computes the address of an object.
unary `*`	Dereferences a pointer to an object.
`->` and `.`	Accesses a member of a structure or union type.

Type and Constant Definitions

This section describes how to declare type aliases and named constants in D. It also discusses D type and namespace management for program and OS types and identifiers.

typedefs

The `typedef` keyword is used to declare an identifier as an alias for an existing type. The `typedef` declaration is used outside of probe clauses in the following 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 D compiler uses the following declaration internally to create the `uint8_t` type alias:

```
typedef unsigned char uint8_t;
```

You can use type aliases 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`, as shown in the following example:

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

In the previous example, `struct foo` is defined using the same type as its alias, `foo_t`. Linux 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 by using a declaration such as the following:

```
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 shown 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`. When an enumeration is defined, the enumerators can be used anywhere in a D program that an integer constant is used. In addition, the enumeration `enum colors` is also defined as a type that's equivalent to an `int`. The D compiler permits a variable of `enum` type to be used anywhere an `int` can be used and permits 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 isn't needed.

Enumerators are visible in all the following clauses and declarations in a program. Therefore, you can't define the same enumerator identifier in more than one enumeration. However, you can define more than one enumerator with the same value in either the same or different enumerations. You can 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 that are defined in the OS kernel and its loadable modules. Note that these enumerators aren't globally visible in a D program. Kernel enumerators are only visible if you specify one as an argument in a comparison with an object of the corresponding enumeration type. This feature protects D programs against inadvertent identifier name conflicts, with the large collection of enumerations that are defined in the OS kernel.

Inlines

D named constants can also be defined by 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 by using the compiled syntax tree and not a set of lexical tokens. An `inline` directive is specified by using a declaration of the following form:

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

where *type* is a type declaration of an existing type, *name* is any valid D identifier that isn't previously defined as an inline or global variable, and *expression* is any valid D expression. After 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 can be used anywhere a global variable of the corresponding type is 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 that's defined by the `inline`, according to the same rules used for the D assignment operator (=). An inline expression can't reference the `inline` identifier itself: recursive definitions aren't permitted.

The DTrace software packages install several D source files in the system directory `/usr/lib64/dtrace/`*`installed-version`*, which contain inline directives that you can use in D programs.

For example, the `signal.d` library includes directives of the following form:

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

These inline definitions provide you with access to the current set of Oracle Linux signal names, as described in the `sigaction(2)` manual page. Similarly, the `errno.d` library contains inline directives for the C `errno` constants that are described in the `errno(3)` manual page.

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

Type Namespaces

In traditional languages such as ANSI C, type visibility is determined by whether a type is nested inside 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 that are defined in C header files are typically included in this outer scope. Unlike these languages, D provides access to types from several outer scopes.

D is a language that provides dynamic observability across different layers of a software stack, including the OS kernel, an associated set of loadable kernel modules, and user processes that are running on the system. A single D program can instantiate probes to gather data from several kernel modules or other software entities that are compiled into independent binary objects. Therefore, more than one data type of the same name, sometimes with different definitions, might be present in the universe of types that are available to DTrace and the D compiler. To manage this situation, the D compiler associates each type with a namespace, which is identified by the containing program object. Types from a particular kernel level object,

such as the main kernel or a kernel module, can be accessed by specifying the object name and the back quote (`) scoping operator in any type name.

For a kernel module named `foo` that 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 following type names:

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

For example, the kernel includes a `task_struct` that's described in `include/linux/sched.h`. The definition of this struct depends on kernel configuration at build. You can find out information about the struct, such as its size, by referencing it as follows:

```
sizeof(struct vmlinux`task_struct)
```

The back quote 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. The C type namespace is initially populated with the standard ANSI C intrinsic types, such as `int`. In addition, type definitions that are acquired by using the C preprocessor (`cpp`), by running the `dtrace -C` command, are processed by, and added to the C scope. So, you can include C header files containing type declarations that 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`, and the built-in D type aliases, such as `uint64_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 a D program consisting of member types from other namespaces, the member types are copied into the D namespace by the declaration.

When the D compiler encounters a type declaration that doesn't specify an explicit namespace using the back quote operator, the compiler searches the set of active type namespaces to find a match by using the specified type name. The C namespace is always searched first, followed by the D namespace. If the type name isn't found in either the C or D namespace, the type namespaces of the active kernel modules are searched in load address order, which doesn't guarantee any ordering properties among the loadable modules. To avoid type name conflicts with other kernel modules, use the scoping operator when accessing types that are defined in loadable kernel modules.

The D compiler uses the compressed ANSI C debugging information that's provided with the core Linux kernel modules to access the types that are associated with the OS source code, without the need to access the corresponding C include files. Note that this symbolic debugging information might not be available for all kernel modules on the system. The D compiler reports an error if you try to access a type within the namespace of a module that lacks the compressed C debugging information that's intended for use with DTrace.

Variables

D provides several variable types: scalar variables, associative arrays, scalar arrays, and multidimensional scalar arrays. Variables can be created by declaring them explicitly, but are most often created implicitly on first use. Variables can be restricted to clause or thread scope to avoid name conflicts and to control the lifetime of a variable explicitly.

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 arrays of objects and composite structures. DTrace also represents strings as fixed-size scalars by permitting them to grow to a predefined maximum length.
To create a scalar variable, you can write an assignment expression of the following form:

```
name = expression ;
```

where *name* is any valid D identifier and *expression* is any value or expression that the variable contains.
DTrace includes several built-in scalar variables that can be referenced within D programs. The values of these variables are automatically populated by DTrace. See DTrace Built-in Variable Reference for a complete list of these variables.

Associative Arrays
Associative arrays are used to represent collections of data elements that can be retrieved by specifying a *key*. Associative arrays differ from normal, fixed-size arrays in that they have no predefined limit on the number of elements and can use any expression as a key. Furthermore, elements in an associative array aren't stored in consecutive storage locations.
To create an associative array, you can write an assignment expression of the following form:

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

Where *name* is any valid D identifier, *key* is a comma-separated list of one or more expressions, often as string values, and *expression* is the value that's contained by the array for the specified key.
The type of each object that's contained in the array is also fixed for all elements in the array. You can use any of the assignment operators that are defined in Types, Operators, and Expressions to change associative array elements, subject to the operand rules defined for each operator. The D compiler produces an appropriate error message if you try an incompatible assignment. You can use any type with an associative array key or value that can be used with a scalar variable.
You can reference values in an associative array by specifying the array name and the appropriate key.
You can delete an element in an associative array by assigning a literal `0` to it, regardless of the element datatype. When you delete elements in an array, the storage that's used for that element is deallocated and made available to the system for use.

Scalar Arrays
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 aren't used as often in D as associative arrays.

A D scalar array of 5 integers is declared by using the type `int` and suffixing the declaration with the number of elements in square brackets, for example:

```
int s[5];
```

The D expression `s[0]` refers to the first array element, `s[1]` refers to the second, and so on. DTrace performs bounds checking on the indexes of scalar arrays at compile time to help catch bad index references early.

> **Note:**
>
> Scalar arrays and associative arrays are syntactically similar. You can declare an associative array of integers referenced by an integer key as follows:
>
> ```
> int a[int];
> ```
>
> You can also reference this array using the expression `a[0]`, but from a storage and implementation perspective, the two arrays are different. The scalar array `s` consists of five consecutive memory locations numbered from zero, and the index refers to an offset in the storage that's allocated for the array. However, the associative array `a` has no predefined size and doesn't 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 are allocated by DTrace. Furthermore, these elements don't have to be consecutive. Associative array keys are abstract names for the corresponding values and 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 always creates a new associative array, even though in this expression `a` could also be interpreted as an assignment to a scalar array. Scalar arrays must be predeclared in this situation so that the D compiler can recognize the definition of the array size and infer that the array is a scalar array.

Multidimensional Scalar Arrays

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

```
int s[12][34];
```

A multidimensional scalar array is accessed by using similar notation. For example, to access the value stored at row `0` and column `1`, you would write the D expression as follows:

```
s[0][1]
```

Storage locations for multidimensional scalar array values are computed by multiplying the row number by the total number of columns declared and then adding the column number. Be careful not to confuse the multidimensional array syntax with the D syntax for associative array accesses, that's, `s[0][1]`, isn't the same as `s[0,1]`). If you use an incompatible key expression with an associative array or try an associative array access of a scalar array, the D compiler reports an appropriate error message and refuses to compile the program.

Variable Scope

Variable scoping is used to define where variable names are valid within a program and to avoid variable naming collisions. By using scoped variables you can control the availability of the variable instance to the whole program, a particular thread, or a specific clause.

The following table lists and describes the three primary variable scopes that are available. Note that external variables provide a fourth scope that falls outside of the control of the D program.

Scope	Syntax	Initial Value	Thread-safe?	Description
global	*myname*	0	No	Any probe that fires on any thread accesses the same instance of the variable.
Thread-local	`self->`*myname*	0	Yes	Any probe that fires on a thread accesses the thread-specific instance of the variable.
Clause-local	`this->`*myname*	Not defined	Yes	Any probe that fires accesses an instance of the variable specific to that particular firing of the probe.

Note:

Note the following information:

- Scalar variables and associative arrays have a global scope and aren't multi-processor safe (MP-safe). Because the value of such variables can be changed by more than one processor, a variable can become corrupted if more than one probe changes it.
- Aggregations are MP-safe even though they have a global scope because independent copies are updated locally before a final aggregation produces the global result.

Global Variables

Global variables are used to declare variable storage that's persistent across the entire D program. Global variables provide the broadest scope.

Global variables of any type can be defined in a D program, including associative arrays. The following are some example global variable definitions:

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

Global variables are created automatically on their first assignment and use the type appropriate for the right side of the first assignment statement. Except for scalar arrays, you don't need to explicitly declare global variables before using them. To create a declaration anyway, you must place it outside of program clauses, for example:

```
int x; /* declare int x as a global variable */
int x[unsigned long long, char];
syscall::read:entry
{
  x = 123;
  a[123, 'a'] = 456;
}
```

D variable declarations can't assign initial values. You can 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.

Thread-Local Variables

Thread-local variables are used to declare variable storage that's local to each OS 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.

Thread-local variables are referenced by applying the `->` operator to the special identifier `self`, for example:

```
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 that's used on the right-hand side of the first assignment statement, which is `int` in this example.

Each time the `self->read` variable is referenced in the D program, the data object that's referenced is the one associated with the OS thread that was executing when the corresponding DTrace probe fired. You can think of a thread-local variable as an associative array that's 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 OS data structure is used to create a thread, this thread doesn't reuse the same DTrace thread-local storage identity.

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

storage for the copy is defined to be filled with zeroes. As with associative array elements, underlying storage isn't 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.

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

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

You don't need to explicitly declare thread-local variables before using them. To create a declaration anyway, you must place it outside of program clauses by prepending the keyword `self`, for example:

```
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 that you can reuse names. Remember that `x` and `self->x` aren't the same variable if you overload names in a program.

Clause-Local Variables

Clause-local variable are used to restrict the storage of a variable to the particular firing of a probe. Clause-local is the narrowest scope. When a probe fires on a CPU, the D script is run in program order. Each clause-local variable is instantiated with an undefined value the first time it is used in the script. The same instance of the variable is used in all clauses until the D script has completed running for that particular firing of the probe.

Clause-local variables can be referenced and assigned by prefixing with `this->`:

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

To declare a clause-local variable explicitly before using it, you can do so by 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';
}
```

Note that if a program contains several clauses for a single probe, any clause-local variables remain intact as the clauses are run sequentially and clause-local variables are persistent across different clauses that are enabling the same probe. While clause-local variables are persistent across clauses that are enabling the same probe, their values are undefined in the first clause processed for a specified probe. To avoid unexpected results, assign each clause-local variable an appropriate value before using it.

Clause-local variables can be defined using any scalar variable type, but associative arrays can't 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. When a clause-local variable is defined, this name and type signature can be used in any later D program clause.

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 several times in the same D program clause, it's more efficient to copy it into a clause-local variable first and then reference the local variable repeatedly.

External Variables

The D language uses the back quote character (`) as a special scoping operator for accessing symbols or variables that are defined in the OS, outside of the D program itself.

DTrace instrumentation runs inside the Oracle Linux OS 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 OS kernel and device drivers.

For example, the Oracle Linux kernel contains a C declaration of a system variable named `max_pfn`. This variable is declared in C in the kernel source code as follows:

```
unsigned long max_pfn
```

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

```
trace(`max_pfn);
```

DTrace associates each kernel symbol with the type that's used for the symbol in the corresponding OS C code, which provides source-based access to the local OS data structures.

Kernel symbol names are kept in a separate namespace from D variable and function identifiers, so you don't need to be concerned about these names conflicting with other D variables. When you prefix a variable with a back quote, the D compiler searches the known kernel symbols and uses the list of loaded modules to find a matching variable definition. Because the Oracle Linux kernel can dynamically load modules with separate symbol namespaces, the same variable name might be used more than once in the active OS kernel. You can resolve these name conflicts by specifying the name of the kernel module that contains the variable to be accessed before the back quote in the symbol name. For example, you would refer to the address of the `_bar` function that's provided by a kernel module named `foo` as follows:

```
foo`_bar
```

You can apply any of the D operators to external variables, except for those that modify values, subject to the usual rules for operand types. When required, the D compiler loads the variable names that correspond to active kernel modules, so you don't need to declare these variables. You can't 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 that you're observing.

When you access external variables from a D program, you're accessing the internal implementation details of another program, such as the OS kernel or its device drivers. These implementation details don't form a stable interface upon which you can rely. Any D programs you write that depend on these details might not work when you next upgrade the corresponding piece of software. For this reason, external variables are typically used to debug performance or functionality problems by using DTrace.

Pointers

Pointers are memory addresses of data objects and reference memory used by the OS, by the user program, or by the D script. 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 explicitly 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. Doing so indicates you want to declare a pointer type, as shown in the following statement:

```
int *p;
```

The statement declares a D global variable named `p` that's a pointer to an integer. The declaration means that `p` is a 64-bit integer with a value that's the address of another integer located somewhere in memory. Because the compiled form of the D code is run at probe firing time inside the kernel itself, D pointers are typically pointers associated with the kernel's address space.

To create a pointer to a data object inside the kernel, you can compute its address by using the `&` operator. For example, the kernel source code declares an `unsigned long max_pfn` variable. You could trace the address of this variable by tracing the result of applying the `&` operator to the name of that object in D:

```
trace(&`max_pfn);
```

The `*` operator can be used to specify 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:

```
q = &`max_pfn; trace(*q);
```

```
trace(`max_pfn);
```

In this example, the first fragment creates a D global variable pointer `q`. Because the `max_pfn` object is of type `unsigned long`, the type of `&`max_pfn` is `unsigned long *`, a pointer to `unsigned long`. The type of `q` is implicit in the declaration. Tracing the value of `*q` follows the pointer back to the data object `max_pfn`. This fragment is therefore the same as the second fragment, which directly traces the value of the data object by using its name.

Pointer Safety

DTrace is a robust, safe environment for running D programs. You might write a buggy D program, but invalid D pointer accesses don't cause DTrace or the OS kernel to fail or crash in any way. Instead, the DTrace software detects any invalid pointer accesses, and returns a `BADADDR` fault; the current clause execution quits, an ERROR probe fires, and tracing continues unless the program called exit for the ERROR probe.

Pointers are required in D because they're an intrinsic part of the OS's implementation in C, but DTrace implements the same kind of safety mechanisms that are found in the Java programming language to prevent buggy programs from affecting themselves or each other. DTrace's error reporting is similar to the runtime environment for the Java programming language that detects a programming error and reports an exception.

To observe DTrace's error handling and reporting, you could write a deliberately bad D program using pointers. For example, 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 uses a cast expression to convert `NULL` to be a pointer to an integer. The program then dereferences the pointer by using the expression `*x`, assigns the result to another variable `y`, and then tries to trace `y`. When the D program is run, DTrace detects an invalid pointer access when the statement `y = *x` is processed and reports the following error:

```
dtrace: script '/tmp/badptr.d' matched 1 probe
dtrace: error on enabled probe ID 2 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #1 at BPF pc 156
```

Notice that the D program moves past the error and continues to run; the system and all observed processes remain unperturbed. You can also add an `ERROR` probe to any script to handle D errors. For details about the DTrace error mechanism, see ERROR Probe.

Pointer and Array Relationship

A scalar array is represented by a variable that's associated with the address of its first storage location. A pointer is also the address of a storage location with a defined type. Thus, 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(p[2]);

trace(a[2]);
```

In the first fragment, the pointer `p` is assigned to the address of the first element in scalar array `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 second fragment. One consequence of this equivalence is that D permits you to access any index of any pointer or array. If you access memory beyond the end of a scalar array's predefined size, you either get an unexpected result or DTrace reports 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; whereas, 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 manifested in the D syntax if you try to assign pointers and scalar arrays. If `x` and `y` are pointer variables, the expression `x = y` is legal; it copies the pointer address in `y` to the storage location that's named by `x`. If `x` and `y` are scalar array variables, the expression `x = y` isn't legal. Arrays can't be assigned as a whole in D. If `p` is a pointer and `a` is a scalar array, the statement `p = a` is permitted. This statement is equivalent to the statement `p = &a[0]`.

Pointer Arithmetic

As in C, pointer arithmetic in D isn't identical to integer arithmetic. Pointer arithmetic implicitly adjusts the underlying address by multiplying or dividing the operands by the size of the type referenced by the pointer.

The following D fragment illustrates this property:

```
int *x;

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

This fragment creates an integer pointer `x` and then traces its value, its value incremented by one, and its value incremented by two. If you create and run this program, DTrace reports the integer values `0`, `4`, and `8`.

Because `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 reference consecutive storage locations such as arrays. For example, if `x` was assigned to the address of an array `a`, the expression `x + 1` would be equivalent to the expression `&a[1]`. Similarly, the expression `*(x + 1)` would reference the value `a[1]`. Pointer arithmetic is implemented by the D compiler whenever a pointer value is incremented by using the `+`, `++`, or `=+` operators. Pointer arithmetic is also applied as follows; when an integer is subtracted from a pointer on the left-hand side, when a pointer is subtracted from another pointer, or when the `--` operator is applied to a pointer.

For example, the following D program would trace the result `2`:

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

BEGIN
{
  x = &a[0];
  y = &a[2];
```

```
  trace(y - x);
}
```

Generic Pointers

Sometimes it's 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 by using the type `void *`, where the keyword `void` represents the absence of specific type information, or by using the built-in type alias `uintptr_t`, which is aliased to an unsigned integer type of size that's appropriate for a pointer in the current data model. You can't apply pointer arithmetic to an object of type `void *`, and these pointers can't 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` can 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 can 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.

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're 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 this way, DTrace can more efficiently manage the memory required for programs. If you create composite structures, it's possible to construct expressions that retrieve the kernel address of DTrace object storage. Avoid creating such expressions in D programs. If you need to use such an expression, don't rely on the address being the same across probe firings.

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 runs D programs within the address space of the OS kernel itself. The Linux system manages many address spaces: one for the OS kernel itself, and one for each user process. Because each address space provides the illusion that it can access all 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, such as, `pipe()`, 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 kernel address space access, which would result in an invalid address error or in returning unexpected data to the D program, depending on 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 avoid confusion, take care when writing D programs to name and comment variables storing user addresses appropriately. You can also store user

addresses as `uintptr_t` so that you don't accidentally compile D code that dereferences them..

Structs and Unions

Collections of related variables can be grouped together into composite data objects called *structs* and *unions*. You define these objects in D by creating new type definitions for them. You can use any new types for any D variables, including associative array values. This section explores the syntax and semantics for creating and manipulating these composite types and the D operators that interact with them.

Structs

The D keyword `struct`, short for *structure*, is used to introduce a new type that's composed of a group of other types. The new `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 that contains only data members and no methods.

Suppose you want to create a more sophisticated system call tracing program in D that records several things about each `read()` and `write()` system call that's run for an application, for example, 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 four separate associative arrays, as shown in the following example:

```
int ts[string];       /* declare ts */
int calls[string];    /* declare calls */
int elapsed [string];  /* declare elapsed */
int maxbytes[string]; /* declare maxbytes */

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

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

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

```
  calls["write"], maxbytes["write"], elapsed["write"]);
}
```

You can make the program easier to read and maintain by using a struct. A struct provides a logical grouping pf data items that belong together. It also saves storage space because all data items can be stored with a single key.

First, declare a new `struct` type at the top of the D program source file:

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

The `struct` keyword is followed by an optional identifier that's used to refer back to the new type, which is now known as `struct callinfo`. The struct members are then within a set of braces `{}` and the entire declaration ends with a semicolon (`;`). Each struct member is defined by 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 defines the new type. It doesn't create any variables or allocate any storage in DTrace. When declared, you can use `struct callinfo` as a type throughout the remainder of the D program. Each variable of type `struct callinfo` stores a copy of the four variables that are described by our structure template. The members are 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 following form:

```
        variable-name.member-name
```

The following example is an improved program that uses the new structure type. In a text editor, type the following D program and save it in a file named `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 == $target/
{
  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 == $target/
{
  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);
}
```

Run the program to return the results for a command. For example run the `sudo dtrace -q -s rwinfo.d -c /bin/date` command.

```
sudo dtrace -q -s rwinfo.d -c date
```

The `date` program runs and is traced until it exits and fires the `END` probe which prints the results:

```
 ...
       calls max bytes elapsed nsecs
------ ----- --------- -------------
 read     2       4096         10689
 write    1         29          9817
```

Pointers to Structs

Referring to structs by using pointers is common in C and D. You can use the operator `->` to access struct members through a pointer. If `struct s` has a member `m`, and you have a pointer to this struct named `sp`, where `sp` is a variable of type `struct s *`, you can either use the `*` operator to first dereference the `sp` pointer to access the member:

```
struct s *sp;
(*sp).m
```

Or, you can use the `->` operator to achieve the same thing:

```
struct s *sp;
sp->m
```

DTrace provides several built-in variables that are pointers to structs. For example, the pointer `curpsinfo` refers to `struct psinfo` and its content provides a snapshot of information about the state of the process associated with the thread that fired the current probe. The following table lists a few example expressions that use `curpsinfo`, including their types and their meanings.

Example Expression	Type	Meaning
curpsinfo->pr_pid	pid_t	Current process ID
curpsinfo->pr_fname	char []	Executable file name
curpsinfo->pr_psargs	char []	Initial command line arguments

The next example uses the pr_fname member to identify a process of interest. In an editor, type the following script and save it in a file named procfs.d:

```
syscall::write:entry
/ curpsinfo->pr_fname == "date" /
{
  printf("%s run by UID %d\n", curpsinfo->pr_psargs, curpsinfo->pr_uid);
}
```

This clause uses the expression curpsinfo->pr_fname to access and match the command name so that the script selects the correct write() requests before tracing the arguments. Notice that by using operator == with a left-hand argument that's an array of char and a right-hand argument that's a string, the D compiler infers that the left-hand argument can be promoted to a string and a string comparison is performed. Type the command dtrace -q -s procs.d in one shell and then run several variations of the date command in another shell.

```
sudo dtrace -q -s procfs.d
```

The output that's displayed by DTrace might be similar to the following, indicating that curpsinfo->pr_psargs can show how the command is invoked and also any arguments that are included with the command:

```
date  run by UID 500
/bin/date  run by UID 500
date -R  run by UID 500
...
^C
```

Complex data structures are used often 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 Linux OS kernel and its system interfaces.

Unions

Unions are another kind of composite type available in ANSI C and D and are 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 particular 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. The memory alignment that's used for the union is the maximum alignment required by the union members.

Member Sizes and Offsets

You can determine the size in bytes of any D type or expression, including a `struct` or `union`, by 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 in 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 previous example program. The formal return type of the `sizeof` operator is the type alias `size_t`, which is defined as an unsigned integer that's 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 that's 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 you can use it 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 that's 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 that are larger than 64 bits can't 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 defining the masks yourself and using the `&` operator. The C and D compilers try to pack bits as efficiently as possible, but

they're free to do so in any order or fashion. Therefore, bit-fields aren't guaranteed to produce identical bit layouts across differing compilers or architectures. If you require stable bit layout, construct the bit masks yourself and extract the values by using the `&` operator.

A bit-field member is accessed by specifying its name with the "`.`" or `->` operators, similar to 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 can't be aligned on a byte boundary or be a round number of bytes in size, you can't 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 by using the `&` operator.

DTrace String Processing

DTrace provides facilities for tracing and manipulating strings. This section 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 to enable you to easily and unambiguously use them in tracing programs.

String Representation

In DTrace, strings are represented as an array of characters ending in a null byte, which is a byte with a value of 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 cannot exceed the length of the predefined string limit. However, the limit can be modified in your D program or on the `dtrace` command line by tuning the `strsize` option. 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 `char *`, in that it's the address of a sequence of characters, but the D compiler and D functions such as `trace` provide enhanced capabilities when applied to expressions of type string. For example, the string type removes the ambiguity of type `char *` when you need to trace the actual bytes of a string.

In the following D statement, if `s` is of type `char *`, DTrace traces the value of the pointer `s`, which means it traces an integer address value:

```
trace(s);
```

In the following D statement, by the definition of the `*` operator, the D compiler dereferences the pointer `s` and traces the single character at that location:

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

These behaviors enable you to manipulate character pointers that refer to either single characters, or to arrays of byte-sized integers that aren't strings and don't end with a null byte.

In the next D statement, 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`:

```
trace(s);
```

You can also perform lexical comparison of expressions of type string. See String Comparison.

String Constants

String constants are enclosed in pairs of 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 the system and by whatever limit you have set for the `strsize` DTrace runtime option. 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 can't contain a literal newline character. To create strings containing newlines, use the `\n` escape sequence instead of a literal newline. String constants can also contain any of the special character escape sequences that are defined for character constants.

String Assignment

Unlike the assignment of `char *` variables, strings are copied by value and not by reference. The string assignment operator `=` 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 use a declaration to create a string variable:

```
string s;
```

Or you can create a string variable by assigning it an expression of type `string`.

For example, the D statement:

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

creates a variable `s` of type `string` and copies the six bytes of the string `"hello"` into it (five printable characters, plus the null byte).

String assignment is analogous to the C library function `strcpy()`, with the exception that if the source string exceeds the limit of the storage of the destination string, the resulting string is automatically truncated by a null byte at this limit.

You can also assign to a string variable an expression of a type that's 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]`, a scalar array of `char` of any size, to be promoted to a string.

String Conversion

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

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

The expression is interpreted as an address to the string.

The `stringof` operator binds very tightly to the operand on its right-hand side. You can optionally surround the expression by using parentheses, for clarity.

Scalar type expressions, such as a pointer or integer, or a scalar array address can be converted to strings, in that the scalar is interpreted as an address to a char type. 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 prevents 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`. See String Assignment for a detailed description. See also Table 4-13, which lists the relational operators that can be used to compare strings.

Table 4-13 D Relational Operators for Strings

Operator	Description
<	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, similarly to the C library routine `strcmp()`. Each byte is compared by using its corresponding integer value in the ASCII character set until a null byte is read or the maximum string length is reached. See the `ascii(7)` manual page for more information. Some example D string comparisons and their results are shown in the following table.

D string comparison	Result
`"coffee" < "espresso"`	Returns 1 (true)
`"coffee" == "coffee"`	Returns 1 (true)
`"coffee"" >= "mocha"`	Returns 0 (false)

> **Note:**
>
> Identical Unicode strings might compare as being different if one or the other of the strings isn't normalized.

Aggregations

Aggregations enable you to accumulate data for statistical analysis. The aggregation is calculated at runtime, so that post-processing isn't required and processing is highly efficient and accurate. Aggregations function similarly to associative arrays, but are populated by aggregating functions. In D, the syntax for an aggregation is as follows:

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

The aggregation *name* is a D identifier that's prefixed with the special character `@`. All aggregations that are named in D programs are global variables. Aggregations can't have thread-local or clause-local scope. The aggregation names are kept in an identifier namespace that's separate from other D global variables. If you reuse names, remember that `a` and `@a` are *not* the same variable. The special aggregation name `@` can be used to name an anonymous aggregation in D programs. The D compiler treats this name as an alias for the aggregation name `@_`.

Aggregations can be regular or indexed. Indexed aggregations use keys, where *keys* are a comma-separated list of D expressions, similar to the tuples of expressions used for associative arrays. Regular aggregations are treated similarly to indexed aggregations, but don't use keys for indexing.

The *aggfunc* is one of the DTrace aggregating functions, and *args* is a comma-separated list of arguments appropriate to that function. Most aggregating functions take a single argument that represents the new datum.

Aggregation Functions

The following functions are aggregating functions that can be used in a program to collect data and present it in a meaningful way.

- avg: Stores the arithmetic average of the specified expressions in an aggregation.
- count: Stores an incremented count value in an aggregation.
- max: Stores the largest value among the specified expressions in an aggregation.
- min: Stores the smallest value among the specified expressions in an aggregation.
- sum: Stores the total value of the specified expression in an aggregation.
- stddev: Stores the standard deviation of the specified expressions in an aggregation.
- quantize: Stores a power-of-two frequency distribution of the values of the specified expressions in an aggregation. An optional increment can be specified.
- lquantize: Stores the linear frequency distribution of the values of the specified expressions, sized by the specified range, in an aggregation.
- llquantize: Stores the log-linear frequency distribution in an aggregation.

Printing Aggregations

By default, several aggregations are displayed in the order in which they're introduced in the D program. You can override this behavior by using the printa function to print the aggregations. The `printa` function also lets you precisely format the aggregation data by using a format string.

If an aggregation isn't formatted with a `printa` statement in a D program, the `dtrace` command snapshots the aggregation data and prints the results after tracing has completed, using the default aggregation format. If an aggregation is formatted with a `printa` statement, the default behavior is disabled. You can achieve the same results by adding the `printa(@aggregation-name)` statement to an `END` probe clause in a program.

The default output format for the `avg`, `count`, `min`, `max`, `stddev`, and `sum` aggregating functions displays an integer decimal value corresponding to the aggregated value for each tuple. The default output format for the `quantize`, `lquantize`, and `llquantize` aggregating functions displays an ASCII histogram with the results. Aggregation tuples are printed as though `trace` had been applied to each tuple element.

Data Normalization

When aggregating data over some period, you might want to normalize the data based on some constant factor. This technique lets you compare disjointed 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 function lets you 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.

Speculation

DTrace includes a speculative tracing facility that can be used to tentatively trace data at one or more probe locations. You can then decide to commit the data to the principal buffer at another probe location. You can use speculation to trace data that only contains the output that's of interest; no extra processing is required and the DTrace overhead is minimized.

Speculation is achieved by:

- Setting up a temporary speculation buffer
- Instructing one or more clauses to trace to the speculation buffer
- Committing the data in the speculation buffer to the primary buffer; or discarding the speculation buffer.

You can decide to commit or discard speculation data when certain conditions are met, by using the appropriate functions within a clause. By using speculation, you can trace data for a set of probes until a condition is met and then either dispose of the data if it isn't useful, or keep it.

The following table describes DTrace speculation functions.

Table 4-14 DTrace Speculation Functions

Function	Args	Description
speculation	None	Returns an identifier for a new speculative buffer.
speculate	ID	Denotes that the remainder of the clause must be traced to the speculative buffer specified by ID.
commit	ID	Commits the speculative buffer that's associated with ID.
discard	ID	Discards the speculative buffer that's associated with ID.

Example 4-1 How to use speculation

The following example illustrates how to use speculation. All speculation functions must be used together for speculation to work correctly.

The speculation is created for the `syscall::open:entry` probe and the ID for the speculation is attached to a thread-local variable. The first argument of the `open()` system call is traced to the speculation buffer by using the `printf` function.

Three more clauses are included for the `syscall::open:return` probe. In the first of these clauses, the `errno` is traced to the speculative buffer. The predicate for the second of the clauses filters for a non-zero `errno` value and commits the speculation buffer. The predicate of the third of the clauses filters for a zero `errno` value and discards the speculation buffer.

The output of the program is returned for the primary data buffer, so the program effectively returns the file name and error number when an `open()` system call fails. If the call doesn't fail, the information that was traced into the speculation buffer is discarded.

```
syscall::open:entry
{
  /*
   * The call to speculation() creates a new speculation. If this fails,
   * dtrace 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 primary data buffer if
the
   * speculation is subsequently committed.
   */
  printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/
```

```
{
  /*
   * Trace the errno value into the speculation buffer.
   */
  speculate(self->spec);
  trace(errno);
}

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

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

5

DTrace Runtime and Compile-time Options Reference

DTrace uses reasonable default values and flexible default policies for runtime configuration. Tuning mechanisms in the form of DTrace compiler or runtime option can change the default behavior of the `dtrace` utility. You can find more information about the `dtrace` utility and various command line options in the `dtrace(8)` manual page.

Options that can be specified when running the `dtrace` utility can be categorized into three types:

- Compile-time Options: affect the compilation process but might also affect runtime behavior.
- Runtime Options: affect the runtime behavior of DTrace but which are often set at compile time.
- Dynamic Runtime Options: affect the runtime behavior of DTrace but which can be changed while tracing, by using the setopt function.

Setting DTrace Compile-time and Runtime Options

You can tune DTrace by setting or enabling a selection of runtime or compiler options. You can set options by either using the `-x` command line switch when running the `dtrace` command, or by specifying `pragma` lines in D programs. If an option takes a value, follow the option name with an equal sign (`=`) and the option value.

Value Suffixes

Use the following optional suffixes for values that denote size or time:

- `k` or `K`: kilobytes
- `m` or `M`: megabytes
- `g` or `G`: gigabytes
- `t` or `T`: terabytes
- `ns` or `nsec`: nanoseconds
- `us` or `usec`: microseconds
- `ms` or `msec`: milliseconds
- `s` or `sec`: seconds
- `m` or `min`: minutes
- `h` or `hour`: hours
- `d` or `day`: days
- `hz`: number per second

Example 5-1 Enabling Options Using the DTrace Utility

The `dtrace` command accepts option settings on the command line by using the `-x` switch, for example:

```
sudo dtrace -x nspec=4 -x bufsize=2g \
-x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

Example 5-2 DTrace Pragma Lines To Enable Options in a D Program

You can set options in a D program by using `#pragma D` followed by the string `option` and the option name and value. The following are examples of valid option settings:

```
#pragma D option nspec=4
#pragma D option bufsize=2g
#pragma D option switchrate=10hz
#pragma D option aggrate=100us
#pragma D option bufresize=manual
```

Compile-time Options

Compile-time options can control how DTrace programs are compiled into eBPF code that's loaded into kernel space.

aggpercpu
Compile-time option that reports aggregations as usual and on a per-cpu basis. Per-cpu aggregations can also be seen by adding `cpu` as an aggregation key

amin=<string>
Compile-time option that sets the stability attribute minimum.

argref
Compile-time option that disables the requirement to use all macro arguments.

core
Compile-time option that enables core dumping by `dtrace`.

cpp
Compile-time option that enables `cpp` to preprocess the input file.

cppargs
Compile-time option that specifies and extra arguments to pass to cpp (when using -C).

cpphdrs
Compile-time option that specifies the `-H` option to `cpp` to print the name of each header file used.

cpppath=<string>
Compile-time option that specifies the path name of `cpp`.

ctfpath
Compile-time option that can specify the path of vmlinux.ctfa.

ctypes=<string>
Compile-time option that specifies Compact Type Format (CTF) definitions of all C types used in a program at the end of a D compilation run.

debug
Compile-time option that enables DTrace debugging mode. This option is the same as setting the environment variable `DTRACE_DEBUG`.

debugassert
Compile-time option that can enable specific debug modes [UNTESTED].

defaultargs
Compile-time option that allows references to unspecified macro arguments. Use `0` as the value for an unspecified argument.

define=<string>
Compile-time option that specifies a macro name and optional value in the form *name*[=*value*]. This option is the same as running `dtrace -D`.

disasm
Compile-time option to specify requested disassembler listings (when using -S).

droptags
Compile-time option that specifies that drop tags are used.

dtypes=<string>
Compile-time option that specifies CTF definitions of all D types that are used in a program at the end of a D compilation run.

empty
Compile-time option that permits compilation of empty D source files.

errtags
Compile-time option that prefixes default error message with error tags.

evaltime=[exec|main|postinit|preinit]
Compile-time option that controls when DTrace starts tracing a new process. For dynamically linked binaries, tracing starts:

- **exec**: After `exec()`.
- **preinit**: After initialization of the dynamic linker to load the binary.
- **postinit**: After constructor execution. Default value.
- **main**: Before `main()` starts. Same as `postinit`.

For statically linked binaries, `preinit` is equivalent to `exec`.
For stripped, statically linked binaries, `postinit` and `main` are equivalent to `preinit`.

incdir=<string>
Compile-time option that adds an `#include` directory to the preprocessor search path. This option is the same as running `dtrace -I`.

iregs=<scalar>
Compile-time option that sets the size of the DTrace Intermediate Format (DIF) integer register set. The default value is `8`.

kdefs
Compile-time option that prevents unresolved kernel symbols.

knodefs
Compile-time option that permits unresolved kernel symbols.

late=[dynamic|static]
Compile-time option that specifies whether to permit references to dynamic translators:

- **dynamic**: Allow references to dynamic translators.
- **static**: Require translators to be statically defined.

lazyload=<true|false>
Compile-time option that specifies lazy loading for the DTrace Object Format (DOF) rather than active loading.

ldpath=<string>
Compile-time option that specifies the path of the dynamic linker loader (`ld`).

libdir=<string>
Compile-time option that adds a library directory to the library search path.

linkmode=[dynamic|kernel|static]
Compile-time option that specifies the symbol linking mode used by the assembler when processing external symbol references:

- **dynamic**: All symbols are treated as dynamic.
- **kernel**: Kernel symbols are treated as static and user symbols are treated as dynamic.
- **static**: All symbols are treated as static.

linknommap
Compile-time option to disable use of MMAP-based libelf support when linking USDT objects.

linktype=[dof|elf]
Compile-time option that specifies the output file type:

- **dof**: Produce a standalone DOF file.
- **elf**: Produce an ELF file that contains DOF.

modpath=<string>
Compile-time option that specifies the module path. The default path is `/lib/modules/`*version*.

nolibs
Compile-time option that prevents processing D system libraries.

pgmax=<scalar>
Compile-time option that sets a limit on the number of threads that DTrace can grab for tracing. The default value is `8`.

preallocate=<scalar>
Compile-time option that sets the amount of memory to preallocate.

procfspath=<string>
Compile-time option that sets the path to the `procfs` file system. The default path is `/proc`.

pspec
Compile-time option that enables interpretation of ambiguous specifiers as probe names.

stdc=[a|c|s|t]
Compile-time option that specifies ISO C conformance settings for the preprocessor when invoking `cpp` with the `-C` option.
The `a`, `c`, and `t` settings include the`-std=gnu99` option (conformance with 1999 C standard including GNU extensions).
The `s` setting includes the `-traditional-cpp` option (conformance with K&R C).

strip
Compile-time option that strips non-loadable sections from the program.

syslibdir=<string>
Compile-time option that sets the path name of system libraries.

tree=<scalar>
Compile-time option that sets the value of the DTrace tree dump bitmap.

tregs=<scalar>
Compile-time option that sets the size of the DIF tuple register set. The default value is `8`.

udefs
Compile-time option that prevents unresolved user symbols.

undef=<string>
Compile-time option that undefines a symbol when invoking the preprocessor. This option is the same as running `dtrace -U`.

unodefs
Compile-time option that permits unresolved user symbols.

useruid
Compile-time option to use first UID that isn't in the system range .

verbose
Compile-time option that enables DIF verbose mode, which shows each compiled DIF object (DIFO).

version=<string>
Compile-time option that requests a specific version of the DTrace library.

zdefs
Compile-time option that permits probe definitions that match zero probes.

Runtime Options

Runtime options can control how the DTrace utility behaves.

aggsize=<size>
Runtime option that sets the buffer size for aggregation.

bpflog=<size>
Runtime option that forces reporting of the BPF verifier log (even if verification was successful).

bpflogsize
Runtime option that sets the maximum size of the BPF verifier log.

bufsize=<size>
Runtime option that sets the principal buffer size. The default buffer size is set to 4 MB. This option is the same as running `dtrace -b`.

cleanrate=<time>
Runtime option that sets the cleaning rate.

cpu=<scalar>
Runtime option that restricts tracing to a particular CPU.

destructive
Runtime option that permits destructive functions to run. This option is the same as running `dtrace -w`.

dynvarsize=<size>
Runtime option that sets dynamic variable space size.

lockmem
Runtime option that sets the locked pages limit. This is set to `unlimited` by default.

maxframes=<scalar>
Runtime option that sets the maximum number of stack frames reported by the kernel.

noresolve
Runtime option that disables automatic resolving of userspace symbols.

nspec=<scalar>
Runtime option that sets the number of speculations.

pcapsize=<size>
Runtime option that sets the maximum packet data capture size.

scratchsize=<size>
Runtime option that sets the maximum DTrace scratch memory size. Some functions in DTrace require that *scratch memory*, is made available. For example, when you allocate memory in a program by using the `alloca()` function, scratch memory is used for this purpose. Scratch memory is only valid while a clause is being processed and is released when the clause has finished being processed. If there isn't enough scratch memory, a function in a DTrace script can return an error and any remaining processing of the clause might fail. The default value is 256 bytes.

specsize=<size>
Runtime option that sets the speculation buffer size.

stackframes=<scalar>
Runtime option that sets the number of stack frames. The default value is 20.

statusrate=<time>
Runtime option that sets the rate of status checking.

strsize=<size>
Runtime option that sets the string size. The default value is 256.

ustackframes=<scalar>
Runtime option that sets the number of user-land stack frames. The default value is 100.

Dynamic Runtime Options

Dynamic runtime options are specific to D programs themselves and are likely to change depending on program functionality and requirements.

aggrate=<time>
Dynamic runtime option that sets the amount of time between aggregation readings.

aggsortkey=<true|false>
Dynamic runtime option that sorts aggregations by key.

aggsortkeypos=<scalar>
Dynamic runtime option that sets the position, or number, of the aggregation key on which to sort.

aggsortpos=<scalar>
Dynamic runtime option that sets the position, or number, of the aggregation variable on which to sort

aggsortrev=<true|false>
Dynamic runtime option that sorts aggregations in reverse order.

flowindent
Dynamic runtime option that controls indentation.
Indent function entry and prefix with `->`.
Unindent function return and prefix with `<-`.
Indent system call entry and prefix with `=>`.
Unindent system call return and prefix with `<=`.
This option is the same as running `dtrace -F`.

quiet
Dynamic runtime option that restricts output to explicitly traced data. This option is the same as running `dtrace -q`.

quietresize
Dynamic runtime option that suppresses buffer-resize messages.

rawbytes
Dynamic runtime option that prints `trace` output in hexadecimal.

stackindent=<scalar>
Dynamic runtime option that sets the number of white space characters to use when indenting `stack` and `ustack` output. The default value is 14.

switchrate=<time>
Dynamic runtime option that sets the rate at which the buffer is read. You can increase the rate to help prevent data drops, or consider increasing the size of the principal buffer with the `bufsize` option.

6
DTrace Stability Reference

DTrace provides a mechanism to track the stability of interfaces and their architecture dependencies. This reference provides detail on how attributes are stored and described and their values.

DTrace Interface Stability Attributes

DTrace describes interfaces by using a triplet of attributes consisting of two stability levels and one dependency class. By convention, the interface attributes are written in the following order and are separated by slashes:

```
name_stability / data_stability / dependency_class
```

The *name stability* of an interface describes the stability level that's associated with its name, as it appears in a D program or on the `dtrace` command line. For example, the `execname` D variable is a Stable name.

The *data stability* of an interface is distinct from the stability that's associated with the interface name. This stability level describes the commitment to maintain the data formats that are 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 the following DTrace interface entities: providers, probe descriptions, D variables, D functions, types, and program statements.

Stability attributes are computed by selecting the minimum stability level and class from the corresponding values for each interface attributes triplet.

The DTrace utility can report on the calculated stability of a D program when run with the `-v` option. Use the -e option to prevent DTrace from running the program and to restrict output to only provide the report. For example, you can run:

```
sudo dtrace -ev -s myscript.d
```

Output similar to the following is displayed:

```
Stability attributes for description dtrace:::BEGIN:

    Minimum Probe Description Attributes
        Identifier Names: Stable
        Data Semantics:   Stable
        Dependency Class: Common

    Minimum Statement Attributes
        Identifier Names: Stable
        Data Semantics:   Private
        Dependency Class: Common
```

```
dtrace:::BEGIN

    Probe Description Attributes
        Identifier Names: Stable
        Data Semantics:   Stable
        Dependency Class: Common

    Argument Attributes
        Identifier Names: Stable
        Data Semantics:   Stable
        Dependency Class: Common

    Argument Types
        None
```

You can use the `-x amin=_attributes_` option with the `dtrace` command to force the D compiler to produce an error whenever any attributes computation results in a triplet of attributes less than the minimum values that you specify on the command line. Note that attributes are specified with three labels that are delimited `/`, according to the standard notation to describe stability. For example:

```
sudo dtrace -x amin=Evolving/Evolving/Common -s myscript.d
```

Stability attributes are computed for a probe description by taking the minimum stability attributes of all the specified probe description fields, according to the attributes that are published by the provider. DTrace providers export a stability attributes triplet for each of the four description fields for all the probes published by that provider. Therefore, a provider's name can have a greater stability than the individual probes that it exports. For simplicity, most providers use a single set of attributes for all the individual module function name values they publish. Providers also specify attributes for the `args[]` array because the stability of any probe arguments varies by provider.

If the provider field isn't specified in a probe description, then the description is assigned the Unstable/Unstable/Common stability attributes because the description might end up matching probes of providers that don't yet exist when used on a future Oracle Linux release. As such, Oracle doesn't provide guarantees about the future stability and behavior of the program. Always explicitly specify the provider when writing D program clauses. In addition, any probe description fields that contain pattern matching characters or macro variables, such as `$1`, are treated as unspecified because these description patterns might expand to match providers or probes to be released in future versions of DTrace and Oracle Linux.

Stability Levels

Stability levels describe the stability of software entities and DTrace interfaces. DTrace stability levels indicate how likely D programs and layered tools are to require corresponding changes when you upgrade or change the software stack.

Stability Value	Description
Internal	The interface is private to DTrace and represents an implementation detail of DTrace. Internal interfaces might change in minor or micro releases.

Stability Value	Description
Private	The interface is private to Oracle and represents an interface developed for use by other Oracle products that aren't yet publicly documented for use by customers and ISVs (independent software vendors). Private interfaces might change in minor or micro releases.
Obsolete	The interface is available 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 try 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 provides developers early access to new or changing technology or to an implementation artifact that's essential for observing or debugging system behavior for which a more stable solution is expected 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 occur in minor and major releases. These changes are avoided in micro releases whenever possible. If such a change is necessary, it's documented in the release notes for the affected release. Also, when feasible, migration aids are 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 describes the standard to which the interface conforms. Standards are typically controlled by a standards development organization. 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. Availability is provided for only the specified versions of a standard; availability in later versions isn't guaranteed.

Dependency Classes

Dependency classes are used to describe architectural dependencies for interfaces in DTrace.

Dependency Class	Description
Unknown	The interface has an unknown set of architectural dependencies. DTrace doesn't necessarily know the architectural dependencies of all entities, such as the data types defined in the OS implementation. The Unknown label is typically applied to interfaces of very low stability for which dependencies can't be computed. The interface might not be available when using DTrace on *any* architecture other than what you're currently using.
CPU	The interface is specific to the CPU model of the current system. Interfaces with CPU model dependencies might not be available on other CPU implementations, even if those CPUs export the same instruction set architecture (ISA).
Platform	The interface is specific to the hardware platform for the current system. A platform typically associates a set of system components and architectural characteristics. To display the current platform name, use the `uname -i` command. The interface might not be available on other hardware platforms.
Group	The interface is specific to the hardware platform group for the current system. A platform group typically associates a set of platforms with related characteristics together under a single name. To display the current platform group name, use the `uname -m` command. The interface is available on other platforms in the platform group, but it might not be available on hardware platforms that aren't members of the group.
ISA	The interface is specific to the ISA that's available for the microprocessors on the current system. The ISA describes a specification for software that can be run on the microprocessor, including details such as assembly language instructions and registers.
Common	The interface is common to all Oracle Linux platforms, regardless of the underlying hardware. DTrace programs and layered applications that depend only on Common interfaces can be run and deployed on other Oracle Linux platforms with the same Oracle Linux and DTrace revisions. Most DTrace interfaces are Common, so you can use them wherever you use Oracle Linux.

7

DTrace Built-in Variable Reference

DTrace includes a set of built-in scalar variables that can be used in D programs or scripts.

Macro Variables

Macro variables are variables that are populated at runtime and identify information about the running `dtrace` process or the process running the compiler.

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 an input file or script. The following table describes the macro variables that the D compiler provides.

Table 7-1 D Macro Variables

Name	Description	Reference
`$[0-9]+`	Macro arguments	See Macro Arguments
`$egid`	Effective group ID	See the `getegid(2)` manual page.
`$euid`	Effective user ID	See the `geteuid(2)` manual page.
`$gid`	Real group ID	See the `getgid(2)` manual page.
`$pid`	Process ID	See the `getpid(2)` manual page.
`$pgid`	Process group ID	See the `getpgid(2)` manual page.
`$ppid`	Parent process ID	See the `getppid(2)` manual page.
`$sid`	Session ID	See the `getsid(2)` manual page.
`$target`	Target process ID	See Target Process ID
`$uid`	Real user ID	See the `getuid(2)` manual page

The variables expand to the attribute value associated with the current `dtrace` process or whatever process is running the D compiler. All the macro variables expand to integers that correspond to system attributes, such as the process ID and the user ID, except the `$[0-9]+` macro arguments and the `$target` macro variable.

Using macro variables in interpreter files lets you create persistent D programs that you don't need to edit every time you want to use them. For example, to count all system calls, except

those that are run by the `dtrace` command, use the following D program clause containing `$pid`:

```
syscall:::entry
/pid != $pid/
{
  @calls = count();
}
```

This clause always behaves as expected, even though each invocation of the `dtrace` command has a different process ID. Macro variables can be used in a D program anywhere that an integer, identifier, or string can be used.

Macro variables are expanded only one time when the input file or script is parsed, not recursively.

Except in probe descriptions, each macro variable is expanded to form a separate input token and can't be concatenated with other text to yield a single token.

For example, if `$pid` expands to the value `456`, the D code in the following example would expand to the two adjacent tokens `123` and `456`, resulting in a syntax error, rather than the single integer token `123456`:

```
123$pid
```

However, in probe descriptions, macro variables are expanded and concatenated with adjacent text.

Macro variables are only expanded one time within each probe description field and they can't contain probe description delimiters (`:`).

Macro Arguments

The D compiler also provides a set of macro variables corresponding to any more argument operands that are specified as part of the `dtrace` command invocation. These *macro arguments* are accessed by using the built-in names `$0`, for the name of the D program file or `dtrace` command, `$1`, for the first extra operand, `$2` for the second operand, and so on. If you use the `-s` option, `$0` expands to the value of the name of the input file that's used with this option. For D programs that are specified on the command line, `$0` expands to the value of `argv[0]`, which is used to run the `dtrace` command 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 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 run as normal Linux commands and use information that's specified by a user or by another tool to change their behavior.

For example, the following D interpreter file traces `write()` system calls that are run by a particular process ID and saved in a file named `tracewrite`:

```
#!/usr/sbin/dtrace -s
syscall::write:entry
/pid == $1/
{
}
```

If you make this interpreter file executable, you can specify the value of `$1` by using an extra command line argument after the interpreter file, for example:

```
sudo chmod a+rx ./tracewrite
sudo ./tracewrite 12345
```

The resulting command invocation counts each `write()` system call that's made by the process ID 12345.

If a D program references a macro argument that isn't provided on the command line, an appropriate error message is printed and the program fails to compile, as shown in the following example output:

```
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

D programs can reference unspecified macro arguments if you set the `defaultargs` option. If `defaultargs` is set, unspecified arguments have the value `0`. See DTrace Runtime and Compile-time Options Reference for more information about D compiler options. The D compiler also produces an error message if other arguments that aren't referenced by the D program are specified on the command line.

The macro argument values must match the form of an integer, identifier, or string. If the argument doesn't match any of these forms, the D compiler reports 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 the shell:

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

If you want 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, `$$1`, which forces the D compiler to interpret the argument value as if it were a string surrounded by double quotes. All the usual D string escape sequences, per Table 4-6, are expanded inside any string macro arguments, regardless of whether they're referenced by using the `$arg` or `$$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 `$target` macro variable to create scripts to be applied to the user process of interest that you specify with the `-p` option or that you create by using the `dtrace` command with the `-c` option. The D programs that you specify on the command line or by using the `-s` option are compiled after processes are created or grabbed, and the `$target` variable expands to the integer process ID of the first such process.

For example, you could use the following D script to find the distribution of system calls that are made by a particular subject process. Save it in a file named `syscall.d`:

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

To find the number of system calls made by the `date` command, save the script in the file named `syscall.d`, then run the following command:

```
sudo dtrace -s syscall.d -c date
```

args[]

The typed and mapped arguments, if any, to the current probe. The `args[]` array is accessed using an integer index. Use `dtrace -l -v` and check `Argument Types` for the type of each argument of each probe. For example, consider the system call `prlimit()`. The prototype on its `man` page (`man -s 2 prlimit`) is consistent with its DTrace probe listing (`dtrace -lvn 'syscall:vmlinux:prlimit*:entry' | grep args`). Specifically, argument 2, if non NULL, points to a `struct rlimit` with the requested resource limit, which can be traced with:

```
syscall:vmlinux:prlimit*:entry
/args[2] != NULL/
{
    printf("request limit %d for resource %d\n", args[2]->rlim_cur, args[1]);
}
```

arg0, ..., arg9

```
int64_t arg0, ..., arg9
```

The built-in variables `arg0`, `arg1` and so on, are the first ten input arguments to a probe, untyped and unmapped, represented as 64-bit integers. Values are meaningful only for arguments defined for the current probe. For example, the command `dtrace -lvn 'rawfbt:vmlinux:ksys_write:entry` indicates that the probe has no typed arguments. Yet we know that kernel function `ksys_write()` has an `arg1` that points to a buffer that's to be written. It might be accessed using:

```
rawfbt:vmlinux:ksys_write:entry
/pid == $target/
```

```
{
    printf("%s\n", stringof(arg1));
}
```

caller

```
uintptr_t caller
```

The built-in variable `caller` references the program counter location of the current kernel thread at the time the probe fired.

curcpu

```
cpuinfo_t * curcpu
```

The built-in variable `curcpu` references the current physical CPU.

curthread

```
vmlinux`struct task_struct * curthread
```

The built-in variable `curthread` references a `vmlinux` data type, for which members can be found by searching for "task_struct" on the Internet.

epid

```
uint_t epid
```

The built-in variable `epid` references the enabled probe ID (EPID) for the current probe. This integer uniquely identifies a particular probe that's enabled with a specific predicate and set of functions.

errno

```
int errno
```

The built-in variable `errno` references the error value returned by the last system call run by this thread.

execname

```
string execname
```

The built-in variable `execname` references the name that was passed to `execve()` to run the current process.

fds

```
fileinfo_t fds[]
```

The built-in `variable fds[]` is an array which has the files the current process has opened in a `fileinfo_t` array, indexed by file descriptor number. See fileinfo_t.

gid

```
gid_t gid
```

The built-in variable `gid` references the real group ID of the current process.

id

```
uint_t id
```

The built-in variable `id` references 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`.

ipl

```
uint_t ipl
```

The built-in variable `ipl` references the interrupt priority level (IPL) on the current CPU at probe firing time.

> **Note:**
>
> This value is non-zero if interrupts are firing and zero otherwise. The non-zero value depends on whether preemption is active, and other factors, and can vary between kernel releases and kernel configurations.

pid

```
pid_t pid
```

The built-in variable `pid` references the process ID of the current process.

ppid

```
pid_t ppid
```

The built-in variable `ppid` references the parent process ID of the current process.

probefunc

```
string probefunc
```

The built-in variable `probefunc` references the function name part of the current probe's description.

probemod

```
string probemod
```

The built-in variable `probemod` references the module name part of the current probe's description.

probename

```
string probename
```

The built-in variable `probename` references the name part of the current probe's description.

probeprov

```
string probeprov
```

The built-in variable `probeprov` references the provider name part of the current probe's description.

stackdepth

```
uint32_t stackdepth
```

The built-in variable `stackdepth` references the current thread's stack frame depth at probe firing time.

tid

```
id_t tid
```

The built-in variable `tid` references the thread ID of the current thread.

timestamp

```
uint64_t timestamp
```

The built-in variable `timestamp` references the current value of a nanosecond timestamp counter. This counter increments from an arbitrary point in the past. Therefore, only use the timestamp counter for relative computations.

ucaller

```
uint64_t ucaller
```

The built-in variable `ucaller` references the program counter location of the current user thread at the time the probe fired.

uid

```
uid_t uid
```

The built-in variable `uid` references the real user ID of the current process.

uregs

```
uint64_t uregs[]
```

The current thread's saved user-mode register values at probe firing time.

ustackdepth

```
uint32_t ustackdepth
```

The built-in variable `ustackdepth` references the user thread's stack frame depth at probe firing time.

vtimestamp

```
uint64_t vtimestamp
```

The built-in variable `vtimestamp` references the current value of a nanosecond timestamp counter that's virtualized to the amount of time that the current thread has been running on a CPU, minus the time spent in DTrace predicates and functions. This counter increments from an arbitrary point in the past. Therefore, only use the vtimestamp counter for relative time computations.

walltimestamp

```
int64_t walltimestamp
```

The built-in variable `walltimestamp` references the current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.

8

DTrace Function Reference

You use D function calls to invoke different kinds of services that DTrace provides.

Functions can be grouped according to their general use case and might appear in more than one grouping:

Data Recording Functions
Data recording functions record data to a DTrace buffer. These are the most common functions and the default DTrace function belongs to this category. By default, data recording functions record data to the *principal buffer*, but can also be directed to record data into a *speculative buffer*.
Data recording functions include:

- Default Action: The default action applies when DTrace encounters an empty clause for a probe. The default action is to trace the enabled probe identifier (EPID).
- printa: Displays and controls the formatting of an aggregation
- printf: Displays and controls the formatting of a string.
- trace: Traces the result of an expression to the directed buffer.
- tracemem: Copies the specified number of bytes of data from an address in memory to the current buffer.

Aggregation Functions
Aggregation functions provide calculated information about sets of DTrace data stored in aggregations.
The following functions are aggregation functions:

- avg: Stores the arithmetic average of the specified expressions in an aggregation.
- count: Stores an incremented count value in an aggregation.
- max: Stores the largest value among the specified expressions in an aggregation.
- min: Stores the smallest value among the specified expressions in an aggregation.
- sum: Stores the total value of the specified expression in an aggregation.
- stddev: Stores the standard deviation of the specified expressions in an aggregation.
- quantize: Stores a power-of-two frequency distribution of the values of the specified expressions in an aggregation. An optional increment can be specified.
- lquantize: Stores the linear frequency distribution of the values of the specified expressions, sized by the specified range, in an aggregation.
- llquantize: Stores the log-linear frequency distribution in an aggregation.

The following functions aren't aggregating functions but work on aggregations:

- clear: Clears the values from an aggregation while retaining aggregation keys.
- denormalize: Removes the normalization that's applied to a specified aggregation.

- normalize: Divides an aggregation value by a specified normalization factor.
- printa: Displays and controls the formatting of an aggregation

Speculation Functions
Speculation functions create or operate on speculative buffers. Speculation is used to trace quantities into speculation buffers that can either be committed to the primary buffer or discarded at a later point, when other important information is known.
The following functions are speculation functions:

- speculation: Creates a speculative trace buffer and returns its ID.
- speculate: A special function that causes DTrace to switch to using a speculation buffer identified by the specified ID for the remainder of a clause.
- commit: Commits the speculative buffer, specified by ID, to the principal buffer.
- discard: Discards a speculative buffer specified by the provided speculation ID.

String Manipulation Functions
String manipulation functions are typical in most programming languages and are used to perform common functional operations on strings. Many functions have analogs in the system library calls described in section 3 of the Oracle Linux manual pages. You can often find out more about these functions by examining the corresponding manual page. For example:

```
man 3 strchr
```

Several of these functions require temporary buffers, which persist only for duration of the clause. Preallocated scratch memory is used for such buffers.
The following string manipulation functions are available:

- index: Finds the first occurrence of a substring within a string.
- rindex: Finds the last occurrence of a specific substring within a string.
- lltostr: Converts an unsigned 64-bit integer to a string.
- strchr: Returns a substring that begins at the first matching occurrence of a specified character in a string.
- strjoin: Concatenates two specified strings and returns the resulting string.
- strlen: Returns the length of a string in bytes.
- strrchr: Returns a substring that begins at the last matching occurrence of a specified character in a string.
- strstr: Returns a substring starting at first occurrence of a specified substring within a string.
- strtok: Parse a string into a sequence of tokens using a specified delimiter.
- substr: Returns the substring from a string at a specified index position.

File Path Manipulation Functions
Similar to string manipulation functions, file path manipulation functions act on file paths or can provide the path name for a specified pointer. Some of these functions have analogs in the system library calls described in section 3 of the Oracle Linux manual pages.

- basename: Returns a string excluding any prefix ending in `/`.

- dirname: Returns the path up to the last level of a specified string.

Integer Conversion Functions

Similar to string manipulation functions, DTrace includes several integer conversion functions that can convert integers between host byte order and network byte order. These functions have analogs in the system library calls described in section 3 of the Oracle Linux manual pages.

The following integer conversion functions are available:

- htonl: Converts an unsigned 32-bit long integer from host byte order to network byte order.
- htonll: Converts an unsigned 64-bit long integer from host byte order to network byte order.
- htons: Converts a short 16-bit unsigned integer from host byte order to network byte order.
- ntohl: Converts a 32-bit long integer from network byte order to host byte order.
- ntohll: Converts a 64-bit long integer from network byte order to host byte order.
- ntohs: Converts a short 16-bit integer from network byte order to host byte order.

Copying Functions

Copying functions are functions that relate to copying information between memory addresses and DTrace buffers. Some of these functions are also considered process destructive functions because they change data in memory for a running process. Destructive functions must be explicitly enabled in DTrace.

- alloca: Allocates memory and returns a pointer.
- bcopy: Copies a specified size in bytes from a specified source address outside of scratch memory to a destination address inside scratch memory.
- copyin: Copies the specified size from the user address to a DTrace buffer and returns the address of the buffer.
- copyinstr: Copies a null-terminated C string from the specified user address to a DTrace buffer and returns the address of the buffer.
- copyinto: Copies the specified size in bytes from the specified user address into the DTrace scratch buffer and returns the buffer address.
- copyout: Copies the specified size from the specified DTrace buffer to the specified user space address.
- copyoutstr: Copies a specified string to a specified user space address.

Lock Analysis Functions

Lock analysis functions are used to check mutexes and file locks.

The following lock analysis functions are available:

- mutex_owned: Checks whether a thread holds the specified kernel mutex.
- mutex_owner: Returns the thread pointer to the current owner of the specified kernel mutex.
- mutex_type_adaptive: Returns a non zero value if a specified kernel mutex is adaptive.
- mutex_type_spin: Returns a non zero value if a specified kernel mutex is a spin mutex.
- rw_iswriter: Checks whether a writer is holding or waiting for the specified reader-writer lock.
- rw_read_held: Checks whether the specified reader-writer lock is held by a reader.

- rw_write_held: Checks whether the specified reader-writer lock is held by a writer.

Symbolic Names and Stack Analysis Functions

DTrace includes functions that either record stack traces to the buffer or which can print symbols and module names for pointers to addresses in user space or kernel space can be helpful for debugging processes.

The following functions return information about stack and addresses:

- stack: Records a stack trace to the buffer.
- func: Prints the symbol for a specified kernel space address. An alias for `sym`.
- mod: Prints the module name that corresponds to a specified kernel space address.
- sym: Prints the symbol for a specified kernel space address. An alias for `func`.
- ustack: Records a user stack trace to the directed buffer.
- uaddr: Prints the symbol for a specified address.
- ufunc: Prints the symbol for a specified user space address. An alias for `usym`.
- umod: Prints the module name that corresponds to a specified user space address.
- usym: Prints the symbol for a specified address. An alias for `ufunc`.

General System Functions

DTrace includes several functions to obtain information from the system or which are generalized for different use cases. Functions in this category include:

- getmajor: Returns the major device number for a specified device.
- getminor: Returns the minor device number for a specified device
- inet_ntoa: Returns a dotted, quad decimal string for a pointer to an IPv4 address.
- progenyof: Checks whether a calling process is in the progeny of a specified process ID.
- rand: Returns a pseudo random integer.

Destructive Functions

DTrace is designed to run code safely. By using destructive functions, you must explicitly enable them to relax the constraints that protect a system from actions that are run from DTrace.

Destructive functions can change a process or the entire system in some defined manner. These include functions such as stopping the current process, raising a specific signal on the current process or even spawning another system process. You can only use these functions if the facility to use destructive functions is explicitly enabled. When using the dtrace utility, you can enable destructive functions by using the `-w` command line option.

If you try to use destructive functions without explicitly enabling them, `dtrace` fails with a message similar to the following:

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

These functions must be used with caution, as such functions can affect every process on the system and any other system, implicitly or explicitly, depending upon the affected system's network services.

- copyout: Copies the specified size from the specified DTrace buffer to the specified user space address.

- copyoutstr: Copies a specified string to a specified user space address.
- freopen: Changes the file associated with `stdout` to a specified file.
- ftruncate: Truncates the output stream on `stdout`.
- raise: Sends a specified signal to the running process.
- system: Causes a specified program to be run on the system as if within a shell.

Special Functions
DTrace also includes functions that change DTrace behavior such as exiting tracing altogether or changing DTrace runtime options.

- exit: Stops all tracing and exits to return an exit value.
- setopt: Dynamically sets DTrace compiler or runtime options.

Default Action

The default action applies when DTrace encounters an empty clause for a probe. The default action is to trace the enabled probe identifier (EPID).

The default action copies trace data from the EPID to the principal buffer. The following information is returned: CPU, probe ID, probe function, and probe name.

The default action provides the most direct use of the `dtrace` command. For example, running the following command enables all the probes in the `vmlinux` module with the default action:

```
sudo dtrace -m vmlinux
```

Output similar to the following is displayed:

```
dtrace: description 'vmlinux' matched 35 probes
CPU     ID                    FUNCTION:NAME
  0     42                   __schedule:sleep
  0     34             dequeue_task:dequeue
  0     40                __schedule:off-cpu
  0     23        finish_task_switch:on-cpu
  0     24             enqueue_task:enqueue
  0     41                __schedule:preempt
...
```

Unimplemented Functions

DTrace implementations have varied in functionality, and some functions aren't relevant to Oracle Linux and might never be implemented. The following functions aren't currently implemented:

- `breakpoint`
- `chill`
- `dpath`
- `ddi_pathname`

- inet_ntoa6
- inet_ntop
- msgdsize
- msgsize
- panic
- pcap
- stop
- trunc

alloca

Allocates memory and returns a pointer.

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

The alloca function allocates *size* bytes out of scratch memory, and returns a pointer to the allocated memory. The returned pointer is guaranteed to have 8–byte alignment. Scratch memory is only valid during the processing of a clause. Memory that's allocated with alloca is deallocated when processing of the clause completes. If insufficient scratch memory is available, no memory is allocated and an error is generated.

Example 8-1 How to use alloca to assign a string to an allocated memory region and then to read it out again by using the pointer

```
BEGIN
{
        x = (string *)alloca(sizeof(string) + 1);
        *x = "abc";
        trace(*x);
        exit(0);
}
```

avg

Stores the arithmetic average of the specified expressions in an aggregation.

```
void avg(expr)
```

The avg function is an aggregation function to return the arithmetic average for a specified D expression.

Example 8-2 How to use avg to display the average time that processes spend in the system write call

The example stores the timestamp for the syscall::write:entry probe fires and then subtracts this value from the timestamp when the syscall::write:return fires. The average time is calculated based on the time difference between the two probes and stored in an aggregation so that it can be updated for each process that runs. When the program exits, the

aggregated average timestamp value is displayed for each process identified by the built-in variable `execname`.

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

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

Output similar to the following is displayed when the program exits:

```
 gnome-session                                          8260
  udisks-part-id                                         9279
  gnome-terminal                                         9378
  lsof                                                  14903
  ip                                                    15075
  date                                                  15371
  ...
  ps                                                    91792
  sestatus                                              98374
  pstree                                               102566
  udisks-daemon                                        250405
  gconfd-2                                           17880523
  cat                                                59752284
```

basename

Returns a string excluding any prefix ending in `/`.

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

The `basename` function creates a string that consists of a copy of the specified string, *str*, but excludes any prefix that ends in `/`, such as a directory path. The returned string is allocated out of scratch memory, and is therefore valid only during the processing of the clause. If insufficient scratch memory is available, `basename` doesn't run and an error is generated.

Example 8-3 How to use basename to return the last element of a path in a string

```
BEGIN
{
        printf("%s\n", basename("/foo/bar/baz"));
        printf("%s\n", basename("/foo/bar///baz/"));
        printf("%s\n", basename("/foo/bar/baz/"));
        printf("%s\n", basename("/foo/bar/baz//"));
}
```

Each of these statements renders the output: `baz`.

bcopy

Copies a specified size in bytes from a specified source address outside of scratch memory to a destination address inside scratch memory.

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

The `bcopy` function copies *size* bytes from the memory that's pointed to by *src* to the memory that's pointed to by *dest*. The source memory mustn't be in user space, and the destination memory must be within DTrace scratch memory.

Example 8-4 How to use bcopy to copy data from one memory location to another

In this example, the `bcopy` function is used to copy 14 characters from the `` `linux_banner `` pointer into a separate memory pointer, `s`, that's allocated 14 bytes of memory. The `printf` line prints a string of the value in stored in the pointer, `s`. The string that's printed is the same as the first 14 characters stored in `` `linux_banner ``.

```
BEGIN
{
        s = (char *)alloca(14);
        bcopy(`linux_banner, &s[0], 13);
        printf("%s\n", stringof(s));
        exit(0);
}
```

clear

Clears the values from an aggregation while retaining aggregation keys.

```
void clear(@ aggr)
```

The `clear` function takes an aggregation as its only parameter. The `clear` function clears only the aggregation's values, while the aggregation's keys are retained. If the key is referenced after the `clear` function is run, it has a zero value.

Example 8-5 How to use clear to show the system call rate only for the most recent ten-second period

The `clear` function is used inside the `tick-10sec` probe to clear the counter values inside the `@func` aggregation.

```
#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;
}
```

cleanpath

Creates a copy of a path without redundant elements.

```
string cleanpath(char *str)
```

The `cleanpath` function creates a string consisting 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's possible that `cleanpath` could take a valid path and return a shorter, invalid path.

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 exist in `/net/foo` and not in `/`. This limitation is because `cleanpath` is called in the context of a firing probe, where full symbolic link resolution of arbitrary names isn't possible. The returned string is allocated out of scratch memory and is therefore valid only during the clause. If insufficient scratch memory is available, `cleanpath` doesn't execute and an error is generated.

commit

Commits the speculative buffer, specified by ID, to the principal buffer.

```
void commit(int id)
```

The `commit` function is a special function that copies data from a speculative buffer, identified by the provided *id*, into the principal buffer. If more data exists in the specified speculative buffer than the 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 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 later. Thus, some time might elapse between a commit that begins on one CPU, while the data is copied from speculative buffers to principal buffers on all CPUs. This length of time is guaranteed to be no longer than the time dictated by the cleaning rate.

Further calls to the speculative buffer while a commit is active are handled as follows:

- `speculation`: the speculative buffer isn't available until each per-CPU speculative buffer has been copied into the corresponding per-CPU principal buffer.
- `speculate`, `commit`, or `discard`: calls are discarded or fail.

A clause containing a `commit` can't contain a data recording function. However, a clause can contain several `commit` calls to commit disjoint buffers.

Example 8-6 How to use speculation

The following example illustrates how to use speculation. All speculation functions must be used together for speculation to work correctly.

The speculation is created for the `syscall::open:entry` probe and the ID for the speculation is attached to a thread-local variable. The first argument of the `open()` system call is traced to the speculation buffer by using the `printf` function.

Three more clauses are included for the `syscall::open:return` probe. In the first of these clauses, the `errno` is traced to the speculative buffer. The predicate for the second of the clauses filters for a non-zero `errno` value and commits the speculation buffer. The predicate of the third of the clauses filters for a zero `errno` value and discards the speculation buffer.

The output of the program is returned for the primary data buffer, so the program effectively returns the file name and error number when an `open()` system call fails. If the call doesn't fail, the information that was traced into the speculation buffer is discarded.

```
syscall::open:entry
{
  /*
   * The call to speculation() creates a new speculation. If this fails,
   * dtrace 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 primary data buffer if
the
   * speculation is subsequently committed.
   */
  printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/
{
  /*
   * Trace the errno value into the speculation buffer.
   */
  speculate(self->spec);
  trace(errno);
}

syscall::open:return
/self->spec && errno != 0/
{
  /*
   * If errno is non-zero, commit the speculation.
   */
```

```
  commit(self->spec);
  self->spec = 0;
}

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

copyin

Copies the specified size from the user address to a DTrace buffer and returns the address of the buffer.

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

The `copyin` function copies the specified size in bytes from the specified user address, *addr*, 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 that's 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 doesn't correspond to a faulted-in page, or if insufficient scratch memory is available, NULL is returned, and an error is generated.

Example 8-7 How to use copyin to copy data from a system write call into the DTrace buffer

In this example, a probe is set for the entry point on write system calls. A predicate is set to filter for when the process execname matches the `bash` application. The `copyin` function is used to copy the first argument, `arg1`, and second argument, `arg2`, of the write call to a string which is printed by `printf`. This script prints the argument for the system write calls when somebody uses the `bash` application.

```
syscall::write:entry
/execname=="bash"/
{
    printf("%s", stringof(copyin(arg1,arg2)));
}
```

copyinstr

Copies a null-terminated C string from the specified user address to a DTrace buffer and returns the address of the buffer.

```
string copyinstr(uintptr_t addr [, size_t size])
```

The `copyinstr` function 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 that's associated with the current thread. An optional maximum length parameter sets a limit on the number of bytes that are examined beyond the address. The resulting string is always null-terminated and the string's length is limited to the value set by the compiler and runtime `strsize` option. As with the copyin function, the specified address must correspond to a faulted-in page in the current process. If the address doesn't correspond to a faulted-in page, or if insufficient scratch memory is available, NULL is returned, and an error is generated.

Example 8-8 How to use copyinstr to copy a string from an address space for a process to the DTrace buffer

In this example, a probe is set for the entry point on write system calls. A predicate is set to filter for when the process execname matches the `passwd` application. The `copyinstr` function is used to copy the first argument, `arg1`, of the write call to a string which is printed by `printf`. This script prints the arguments for the system write calls when somebody uses the `passwd` application to reset a password.

```
syscall::write:entry
/execname=="passwd"/
{
    printf("%s", copyinstr(arg1));
}
```

copyinto

Copies the specified size in bytes from the specified user address into the DTrace scratch buffer and returns the buffer address.

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

The `copyinto` function copies the specified size in bytes, *size*, from the specified user address, *addr*, into the specified DTrace scratch buffer, *dest*. The user address is interpreted as an address in the space of the process that's associated with the current thread. The address in question must correspond to a faulted-in page in the current process. If the address doesn't correspond to a faulted-in page, or if any of the destination memory lies outside of scratch memory, no copying takes place and an error is generated.

Example 8-9 How to use copyinto to copy data from a system write call into an allocated memory buffer

In this example, a probe is set for the entry point on write system calls. A predicate is set to filter for when the process execname matches the `podman` application. The `copyinto` function is used to copy 32 bytes of the first argument, `arg1`, of the write call into a pointer to an allocated memory buffer of 32 bytes, `ptr`. The script prints the a string representation of `ptr` when the `podman` application makes a system write call.

```
syscall::write:entry
/execname=="podman"/
{
        ptr = (char *)alloca(32);
        copyinto(arg1, 32, ptr);
```

```
        printf("'%s'", stringof(ptr));
}
```

copyout

Copies the specified size from the specified DTrace buffer to the specified user space address.

```
void copyout(void *src, uintptr_t addr, size_t size)
```

The `copyout` function is a destrructive function that copies the specified number of bytes from a specified DTrace buffer to a specified user space address. The user space address is in the address space of the process that associated with the current thread. If the user space address doesn't correspond to a valid, faulted-in page in the current address space, an error is generated.

Example 8-10 How to use copyout to copy data from a DTrace buffer to a specified user space address

The example shows how to use `copyout` to write a string value, `DTrace`, into the user space address for a `write` system call when a user runs the `ls` command. If you run this script, whenever anybody runs the `ls` command on the system, the string `DTrace` replaces the first 5 bytes returned by the command.

```
#pragma D option destructive
syscall::write:entry
/execname == "ls"/
{
    copyout("DTrace", arg1, 5);
}
```

copyoutstr

Copies a specified string to a specified user space address.

```
void copyoutstr(char * string, uintptr_t addr, size_t size)
```

The `copyoutstr` function is a destructive function that copies the specified string, *string*, to a specified address, *addr*, in the address space of the process associated with the current thread. A third argument, *size*, is used to control the length of the string. If the user space address doesn't correspond to a valid, faulted-in page in the current address space, an error is generated. Note that the string length is also limited to the value that's set by the compiler and runtime `strsize` option. If *size* exceeds the value `strsize` option, then the string length is limited to the value specified by the `strsize` option.

Example 8-11 How to use copyoutstr to copy a string to a specified user space address

In this example, the `syscall::newuname:entry` and `syscall::newuname:return` probes are used. The entry probe is used to populate a user space address with the first argument used in the entry probe. The return probe writes the string "DTraceHost" into the address of the first

argument. When any process makes the `newuname` system call, the hostname part of the call is rewritten.

```
#pragma D option destructive

syscall::newuname:entry
{
  self->a = arg0;
}

syscall::newuname:return
{
 copyoutstr("DtraceHost", self->a+65, 128);
}
```

When you run this script and then run the `uname -a` command, output similar to the following is displayed:

```
Linux DtraceHost 5.15.0-7.86.6.1.el8uek.x86_64 #2 SMP ... GNU/Linux
```

count

Stores an incremented count value in an aggregation.

```
void count()
```

The `count` function is an aggregation function that takes no arguments and returns the value for the number of times that it has been called.

Example 8-12 How to use count to display the number of write() system calls by process name

This example uses the `syscall::write:entry` probe and an aggregation to store the count value. The aggregation uses the built-in variable, `execname`, as a key.

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

When run, output similar to the following is displayed when the program exits:

```
dtrace: description 'syscall::write:entry' matched 1 probe
^C
  dirname                                                         1
  dtrace                                                          1
  gnome-panel                                                     1
  ps                                                              1
  basename                                                        2
  gconfd-2                                                        2
  java                                                            2
  bash                                                            9
  cat                                                             9
```

```
  gnome-session                                             9
  Xorg                                                     21
  firefox                                                 149
  gnome-terminal                                         9421
  ...
```

denormalize

Removes the normalization that's applied to a specified aggregation.

```
void denormalize(@ aggr)
```

The `denormalize` function removes any normalization that's applied to a specified aggregation. Normalization doesn't change the underlying data that makes up an aggregation, so the `denormalize` function removes the normalization to return the raw data directly.

Example 8-13 How denormalize is used in a script to present raw data

```
#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);
}
```

dirname

Returns the path up to the last level of a specified string.

```
string dirname(const char *string)
```

The `dirname` function creates a string that consists of all but the last level of the path name that's specified by a specified string, *string*. The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `dirname` doesn't run and an error is generated.

Example 8-14 How to use dirname to return the path up to the last element in a string

```
BEGIN
{
        printf("%s\n", dirname("/foo/bar/baz"));
        printf("%s\n", dirname("/foo/bar//baz/"));
        printf("%s\n", dirname("/foo/bar/baz/"));
        printf("%s\n", dirname("/foo/bar/baz//"));
}
```

Each of these statements renders the output: `/foo/bar`.

discard

Discards a speculative buffer specified by the provided speculation ID.

```
void discard(int id)
```

The `discard` function causes DTrace to discard a speculative buffer specified by the provided speculation ID, *id*.

When a speculative buffer is discarded, its contents are also discarded. If the speculation has only been active on the CPU calling `discard`, the buffer is immediately available for further calls to `speculation`. If the speculation has been active on more than one CPU, the discarded buffer is available for further `speculation` some time after the call to `discard`. The length of time between a `discard` on one CPU and the buffer being made available for later speculations is guaranteed to be no longer than the time that's dictated by the cleaning rate. If, at the time `speculation` is called, no buffer is available because all speculative buffers are being discarded or committed, `dtrace` generates a message similar to the following:

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

You can reduce the likelihood of all buffers being unavailable by tuning the number of speculation buffers or the cleaning rate.

Example 8-15 How to use speculation

The following example illustrates how to use speculation. All speculation functions must be used together for speculation to work correctly.

The speculation is created for the `syscall::open:entry` probe and the ID for the speculation is attached to a thread-local variable. The first argument of the `open()` system call is traced to the speculation buffer by using the `printf` function.

Three more clauses are included for the `syscall::open:return` probe. In the first of these clauses, the `errno` is traced to the speculative buffer. The predicate for the second of the clauses filters for a non-zero `errno` value and commits the speculation buffer. The predicate of the third of the clauses filters for a zero `errno` value and discards the speculation buffer.

The output of the program is returned for the primary data buffer, so the program effectively returns the file name and error number when an `open()` system call fails. If the call doesn't fail, the information that was traced into the speculation buffer is discarded.

```
syscall::open:entry
{
  /*
   * The call to speculation() creates a new speculation. If this fails,
   * dtrace 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 primary data buffer if
the
   * speculation is subsequently committed.
   */
  printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/
{
  /*
   * Trace the errno value into the speculation buffer.
   */
  speculate(self->spec);
  trace(errno);
}

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

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

exit

Stops all tracing and exits to return an exit value.

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

The `exit` function is used to immediately stop tracing and inform DTrace to do the following: stop tracing, perform any final processing, and call `exit()` with the specified *status* value. Because `exit` returns a status to user level, it's considered a data recording function. However, unlike other data recording functions, `exit` can't be speculatively traced. Note that because `exit` is a data recording function, it can be dropped.

When `exit` is called, only those DTrace functions that are already in progress on other CPUs are completed. No new functions occur on any CPU. The only exception to this rule is the processing of the `END` probe, which is called after the DTrace has processed the `exit` function, and indicates that tracing must stop.

Example 8-16 How to use exit to end all tracing and exit with an exit value

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

freopen

Changes the file associated with `stdout` to a specified file.

```
void freopen(const char pathname, ...)
```

The `freopen` function is typically a data recording function that changes the file that's associated with `stdout` to the file that's specified by the arguments in `printf` fashion.

If the `""` string is used, the output is again restored to `stdout`.

The `freopen` function isn't only data-recording but also destructive, because you can use it to overwrite arbitrary files.

Example 8-17 How to use freopen to write to a specified file and log a system call

The script opens with a pragma to enable destructive functions in DTrace. You can alternatively remove this line and run the script with `dtrace -w`. The `freopen` function is destructive because it writes to a file on the file system and can overwrite existing files. The example creates a temporary log file to track the process names that make a `mkdir` system call while the program is running.

```
#pragma D option destructive
dtrace:::BEGIN
{
        freopen("/tmp/dlog");
```

```
}
syscall:vmlinux:mkdir:entry
{
        printf("%Y-> %s \n",walltimestamp,execname);
}
```

ftruncate

Truncates the output stream on `stdout`.

```
void ftruncate()
```

The `ftruncate` function is a data recording function that truncates the output stream on `stdout`.

Example 8-18 How to use ftruncate to truncate the stdout output stream, by using a counter

```
tick-10ms
{
    printf("%d\n", i++);
}

tick-10ms
/i == 10/
{
    ftruncate();
}

tick-10ms
/i == 20/
{
    exit(0);
}
```

When the example script is run using `sudo dtrace -o /tmp/result -s` *`/path/to/script`*. Standard output is saved to `/tmp/result`. The program implements a counter that's triggered every 10 ms and is designed to count up to 20 before exiting. The counter prints to standard output for every count, but when the counter reaches 10, `ftruncate` is called to truncate standard output. When the program exits and you can view the contents of `/tmp/result` you can see that the standard output preceding the 11th counter is removed.

func

Prints the symbol for a specified kernel space address. An alias for `sym`.

```
_symaddr func(uintptr_t addr)
```

The `func` function is a data recording function that prints the symbol that corresponds to a specified kernel space address, *addr*. The `func` function is an alias for sym.

Example 8-19 How the func function can return the symbol for a kernel space address

This example uses a bash script to pick a test symbol from `/proc/kallmodsyms` that can be used as a reference in the DTrace program that returns the symbols for the module and function.

```
#!/bin/bash
read ADD <<< $(awk '/ksys_write/ {print $1}' /proc/kallsyms)
dtrace -qn 'BEGIN {func(0x'$ADD'); exit(0)}'
```

getmajor

Returns the major device number for a specified device.

```
vmlinux`dev_t getmajor(vmlinux`dev_t))
```

The `getmajor` function returns the major device number for a specified device.

getminor

Returns the minor device number for a specified device

```
vmlinux`dev_t getminor(vmlinux`dev_t)
```

The `getminor` function returns the minor device number for a specified device.

htonl

Converts an unsigned 32-bit long integer from host byte order to network byte order.

```
uint32_t htonl(uint32_t)
```

The `htonl` function converts an unsigned 32-bit long integer from host byte order to network byte order.

htonll

Converts an unsigned 64-bit long integer from host byte order to network byte order.

```
uint64_t htonll(uint64_t)
```

The `htonll` function converts an unsigned 64-bit long integer from host-byte order to network-byte order.

htons

Converts a short 16-bit unsigned integer from host byte order to network byte order.

```
uint16_t htons(uint16_t)
```

The `htons` function converts a short 16-bit unsigned integer from host byte order to network byte order.

index

Finds the first occurrence of a substring within a string.

```
int index(const char * str, const char * substr [, int start])
```

The `index` function finds the position of the first occurrence of a substring, *substr*, in a string, *str*, starting at an optional position, *start*. If the specified value of the start position is less than 0, it's implicitly set to `0`. If the string is empty, `index` returns `0`. If no match is found for the substring within the string, `index` returns `-1`.

Example 8-20 How to use index to identify the first occurrence of a substring within a string

```
BEGIN {
         x = "#canyoufindapenguininthisstring?";
         y = "penguin";
         printf("The penguin appears at character %3d\n", index(x, y));
         exit(0)
}
```

inet_ntoa

Returns a dotted, quad decimal string for a pointer to an IPv4 address.

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

The `inet_ntoa` function takes a pointer to an IPv4 address and returns it as a dotted, quad decimal string. The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `inet_ntoa` doesn't run and an error is generated. See the `inet(3)` manual page for more information.

Example 8-21 How to use inet_ntoa to return dotted IPv4 address notation for a pointer to an IPv4 address

In the example, an IP address pointer is created in scratch memory and populated so that the `inet_ntoa` function can process it and return a string value.

```
typedef vmlinux`__be32 ipaddr_t;
ipaddr_t *ip4a;
BEGIN
```

```
{
        ip4a = alloca(sizeof(ipaddr_t));
        *ip4a = 0x0100007f;
        printf("%s\n", inet_ntoa(ip4a));
        exit(0);
}
```

inet_ntoa6

Returns an RFC 1884 convention 2 string for a pointer to an IPv6 address.

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

The `inet_ntoa6` function takes a pointer *addr* to an IPv6 address and returns it as an RFC 1884 convention 2 string, with lowercase hexadecimal digits. The returned string is allocated out of scratch memory and is therefore valid only during the clause. If insufficient scratch memory is available, `inet_ntoa6` doesn't run and an error is generated.

inet_ntop

Returns an RFC 1884 convention 2 string for a pointer to an IPv6 address.

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

The `inet_ntop` function takes a pointer *addr* to an IP address and returns a string version that depends on the provided address family, either AF_INET, or AF_INET6. The returned string is allocated out of scratch memory and is therefore valid only during the clause. If insufficient scratch memory is available, `inet_ntop` doesn't run and an error is generated.

link_ntop

Returns a string translation of a hardware address.

```
string link_ntop(int hardware_type, void *addr)
```

The `link_ntop` function translates a hardware address into a string. The *hardware_type* can be `ARPHRD_ETHER` or `ARPHRD_INFINIBAND`. The returned string is allocated out of scratch memory, and is therefore valid only during the clause. If insufficient scratch space is available, `link_ntop` doesn't run and an error is generated. The function is the link-level equivalent of `inet_ntop`.

llquantize

Stores the log-linear frequency distribution in an aggregation.

```
void llquantize(expr, int32_t factor, int32_t from, int32_t to [, int32_t
steps [, int32_t incr]])
```

The `llquantize` function is an aggregation function used to display a log-linear frequency distribution for an expression. The logarithmic base, *factor*, is specified along with lower, *from*, and upper, *to*, exponents and the number of steps, *steps*, per order of magnitude. If the number of steps isn't provided, a default value of 1 is used. An optional integer, *incr*, can be provided to specify the amount to increment each step by.

The log-linear `llquantize` aggregating function combines the capabilities of both the log and linear functions. While the `quantize` function uses base 2 logarithms, with `llquantize`, you specify the base, and the minimum and, maximum exponents. Further, each logarithmic range is subdivided linearly by the number of steps specified and the increment value, if specified.

Example 8-22 How to use llquantize to visualize system call latencies

The script monitors all system call entry and return calls. The time spent in each call is calculated using the timestamp for each. An aggregation is used to create a log-linear quantization with factor of 10 ranging from magnitude 3 to magnitude 5 (inclusive) with 10 steps per magnitude. The output from this script visualizes the latency of system calls in the microsecond range.

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

syscall:::return
/ self->ts /
{
  @ = llquantize(timestamp - self->ts, 10, 3, 5, 5);
  self->ts = 0;
}
```

```
           value  ------------- Distribution ------------- count
           -1000 |                                         0
   abs() < 1000 |@@@@@@@@@@@@@@@@                         2888133
            1000 |@@@@@                                   1017345
            2000 |@@@@                                    714432
            4000 |@                                       266057
            6000 |@                                       118797
            8000 |                                        84332
           10000 |@                                       152108
           20000 |@                                       125154
           40000 |                                        49334
           60000 |                                        38374
           80000 |                                        31739
          100000 |                                        91033
          200000 |                                        51153
          400000 |                                        20343
          600000 |                                        10685
          800000 |                                        6970
    >= 1000000 |@@@@@@@@@@@@                            2081856
```

lltostr

Converts an unsigned 64-bit integer to a string.

```
string lltostr(int64_t)
```

The `lltostr` function converts an unsigned 64-bit integer to a string. The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `lltostr` doesn't run and an error is generated.

Example 8-23 How to use lltostr to convert a 64-bit integer to a string

The example shows that the `printf` function treats the value as a string. The pragma option in the script sets the maximum string size to 7 bytes, so the string that's returned by the `lltostr` function is truncated to 1234567.

```
#pragma D option strsize=7

BEGIN
{
    printf("%s\n", lltostr(1234567890));
}
```

lquantize

Stores the linear frequency distribution of the values of the specified expressions, sized by the specified range, in an aggregation.

```
void lquantize(expr, int32_t from, int32_t to [, int32_t step])
```

The `lquantize` function is an aggregation function used to display a linear value distribution. The `lquantize` function takes four arguments: a D expression, *expr*, a lower bound, *from*, an upper bound, *to*, and an optional *step*. Note that the default step value is `1`.

Example 8-24 How to use lquantize to display the distribution of write() calls by file descriptor

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

Output similar to the following might be displayed after the program exits:

```
 ...
  gnome-session
           value  ------------- Distribution ------------- count
              25 |                                         0
              26 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 9
              27 |                                         0
```

```
  gnome-terminal
           value  ------------- Distribution ------------- count
              15 |                                         0
              16 |@@                                       1
              17 |                                         0
              18 |                                         0
              19 |                                         0
              20 |                                         0
              21 |@@@@@@@@@                                4
              22 |@@                                       1
              23 |@@                                       1
              24 |                                         0
              25 |                                         0
              26 |                                         0
              27 |                                         0
              28 |                                         0
              29 |@@@@@@@@@@@@@@@                          6
              30 |@@@@@@@@@@@@@@@                          6
              31 |                                         0
 ...
```

max

Stores the largest value among the specified expressions in an aggregation.

```
void max(expr)
```

The `max` function is an aggregation function to store the largest value for an expression in an aggregation.

Example 8-25 How to use max to display the maximum time that processes spend in the system write call

The example stores the timestamp for the `syscall::write:entry` probe fires and then subtracts this value from the timestamp when the `syscall::write:return` fires. The maximum time is calculated based on the time difference between the two probes and stored in an aggregation so that it can be updated for each process that runs. When the program exits, the aggregated maximum timestamp value is displayed for each process identified by the built-in variable `execname`.

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

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

Output similar to the following is displayed when the program exits:

```
  ProxyResolution                                          4891
  firewalld                                                7892
  RDD Process                                             11028
  Utility Process                                         11344
  gdbus                                                   11474
  GLXVsyncThread                                          14181
  python3                                                 15286
  Socket Process                                          15294
  rtkit-daemon                                            16547
  pmdakvm                                                 17089
  NetworkManager                                          18246
  pmdaxfs                                                 19661
  sudo                                                    19917
...
```

min

Stores the smallest value among the specified expressions in an aggregation.

```
void min(expr)
```

The `min` function is an aggregation function to store the smallest value for an expression in an aggregation.

Example 8-26 How to use max to display the minimum time that processes spend in the system write call

The example stores the timestamp for the `syscall::write:entry` probe fires and then subtracts this value from the timestamp when the `syscall::write:return` fires. The minimum time is calculated based on the time difference between the two probes and stored in an aggregation so that it can be updated for each process that runs. When the program exits, the aggregated minimum timestamp value is displayed for each process identified by the built-in variable `execname`.

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

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

Output similar to the following is displayed when the program exits:

```
  IPC I/O Parent                                           1087
  gmain                                                    1091
  libvirt-dbus                                             1501
  pmcd                                                     1601
```

```
  libvirtd                                                  1615
  threaded-ml                                               1673
  Timer                                                     2130
  NetworkManager                                            2140
  Socket Thread                                             2275
  InputThread                                               2420
...
```

mod

Prints the module name that corresponds to a specified kernel space address.

```
_symaddr mod(uintptr_t addr)
```

The `mod` function is a data recording function that prints the name of the module that corresponds to a specified kernel space address.

Example 8-27 How to use mod to print the module name for a pointer to a specified kernel space address

This example uses a bash script to pick a test symbol from `/proc/kallmodsyms` that can be used as a reference in the DTrace program that returns the symbol for the module. Note that where a module is effectively empty in `/proc/kallmodsyms` it's the same as a value of `vmlinux`.

```
#!/bin/bash
read ADD <<< `awk '/ksys_write/ {print $1}' /proc/kallmodsyms`
dtrace -qn 'BEGIN {mod(0x'$ADD'); exit(0) }'
```

mutex_owned

Checks whether a thread holds the specified kernel mutex.

```
int mutex_owned(vmlinux`struct mutex *)
```

The `mutex_owned` function returns non-zero if the calling thread holds the specified kernel mutex, or zero otherwise.

Example 8-28 How to use mutex_owned to check whether the calling thread holds a mutex

```
fbt::mutex_lock:entry
{
        this->mutex = arg0;
}

fbt::mutex_lock:return
{
        this->owned = mutex_owned((struct mutex *)this->mutex);
}

fbt::mutex_lock:return
/!this->owned/
{
```

```
        printf("mutex_owned() returned 0, expected non-zero\n");
        exit(1);
}
```

mutex_owner

Returns the thread pointer to the current owner of the specified kernel mutex.

```
vmlinux`struct task_struct mutex_owner(vmlinux`struct mutex *)
```

The `mutex_owner` function returns the thread pointer of the current owner of the specified adaptive kernel mutex. `mutex_owner` returns `NULL` if the specified adaptive mutex is unowned or if the specified mutex is a spin mutex.

Example 8-29 How to use mutex_owner to check whether the calling thread doesn't have ownership of a mutex

```
fbt::mutex_lock:entry
{
        this->mutex = arg0;
}

fbt::mutex_lock:return
{
        this->owner = mutex_owner((struct mutex *)this->mutex);
}

fbt::mutex_lock:return
/this->owner != curthread/
{
        printf("current thread is not current owner of owned lock\n");
        exit(1);
}
```

mutex_type_adaptive

Returns a non zero value if a specified kernel mutex is adaptive.

```
int mutex_type_adaptive(vmlinux`struct mutex *)
```

The `mutex_type_adaptive` function returns a non zero value if a specified kernel mutex is adaptive. All mutexes in the Oracle Linux kernel are adaptive, so the `mutex_type_adaptive` function always returns `1`.

Example 8-30 How to use mutex_type_adaptive to check whether a mutex isn't adaptive

Because all mutexes on Oracle Linux are adaptive, the final clause in this program is never processed.

```
fbt::mutex_lock:entry
{
        this->mutex = arg0;
```

```
}

fbt::mutex_lock:return
{
        this->adaptive = mutex_type_adaptive((struct mutex *)this->mutex);
}

fbt::mutex_lock:return
/!this->adaptive/
{
        printf("mutex_type_adaptive returned 0, expected non-zero\n");
        exit(1);
}
```

mutex_type_spin

Returns a non zero value if a specified kernel mutex is a spin mutex.

```
mutex_type_spin(int(vmlinux`struct mutex *))
```

The `mutex_type_spin` function returns a non zero value if a specified kernel mutex is a spin mutex. All mutexes in the Oracle Linux kernel are adaptive, so the `mutex_type_spin` function always returns `0`.

Example 8-31 How to use mutex_type_spin to check whether a mutex is a spin mutex

Because all mutexes on Oracle Linux are adaptive, the final clause in this program is never processed.

```
fbt::mutex_lock:entry
{
        this->mutex = arg0;
}

fbt::mutex_lock:return
{
        this->spin = mutex_type_spin((struct mutex *)this->mutex);
}

fbt::mutex_lock:return
/this->spin/
{
        printf("mutex_type_spin returned non-zero, expected 0\n");
        exit(1);
}
```

normalize

Divides an aggregation value by a specified normalization factor.

```
void normalize(@ aggr, uint64_t)
```

The `normalize` function divides an aggregation value by a normalization factor to provide a better view of data within an aggregation. The function takes the aggregation and the normalization factor as arguments. A program used to aggregate data over a period but that presents the data as a per-second occurrence rather than an absolute value is a typical example of a use case for this function.

Example 8-32 How to use normalize to show the number of system calls per second for processes

The normalize function is called against the aggregation. The time is divided to by 1,000,000,000 to convert nanoseconds to seconds.

```
#pragma D option quiet

BEGIN
{
  start = timestamp;
}

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

END
{
  normalize(@func, (timestamp - start) / 1000000000);
}
```

ntohl

Converts a 32-bit long integer from network byte order to host byte order.

```
uint32_t ntohl(uint32_t)
```

The `ntohl` function converts a 32-bit long integer from network byte order to host byte order. See the `byteorder(3)` manual page for more information.

ntohll

Converts a 64-bit long integer from network byte order to host byte order.

```
uint64_t ntohll(uint64_t)
```

The `ntohll` function converts a 64-bit long integer from network byte order to host byte order. See the `byteorder(3)` manual page for more information.

ntohs

Converts a short 16-bit integer from network byte order to host byte order.

```
uint16_t ntohs(uint16_t)
```

The `ntohs` function converts a short 16-bit integer from network byte order to host byte order. See the `byteorder(3)` manual page for more information.

print

Prints information about a variable.

```
void print(void *addr)
```

The `print` function is a data recording function that prints information about the variable at the specified address. The function prints the address, type, and data values. This function is aware of kernel and user defined data types, and it recursively descends hierarchies of structures to report on their members. Zero-valued fields are skipped. The dynamic `printsize` option can be used to restrict the amount of data displayed. In the case of a `printsize` overrun, ellipsis (`...`) are shown in the output.

printa

Displays and controls the formatting of an aggregation

```
void printa([string format,] @aggr )
```

The `printa` function is a data recording function that enables you to display and format aggregations. The function takes an aggregation and optionally a string to specify the output formatting using printf formatting directives. If no formatting string is specified, `printa` the specified aggregation is displayed using the default format. If *format* is specified, the aggregation is formatted.

See the `printf(1)` manual page for more information on formatting directives. Note that although DTrace's implementation of `printf` is aligned with the correlating system function, some differences apply. Notably, you can use the `%d` formatting directive to represent any length of an integer. Furthermore, `printa` also handles the appropriate formatting for each aggregation.

Example 8-33 How to use printa to print basic formatting for different aggregations

```
BEGIN
{
        @a = avg(1);
        @b = count();
        @c = lquantize(1, 1, 10);
        printa("@a = %@u\n", @a);
        printa("@b = %@u\n", @b);
        printa("@c = %@d\n", @c);
```

```
        exit(0);
}
```

printf

Displays and controls the formatting of a string.

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

The `printf` function is a data recording function that traces expressions and enables elaborate `printf`-style formatting. The parameters consist of a *format* string, followed by a variable number of arguments. The arguments are traced to the directed buffer and are later formatted for output by the `dtrace` command, according to the specified format string.

See the `printf(1)` manual page for more information on formatting directives. Note that although DTrace's implementation of `printf` is aligned with the correlating system function, some differences apply. Notably, you can use the `%d` formatting directive to represent any length of an integer.

Example 8-34 How to use printf to print a formatted string

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

progenyof

Checks whether a calling process is in the progeny of a specified process ID.

```
int progenyof(pid_t)
```

The `progenyof` function returns non zero if the calling process is among the progeny of the specified process ID. The calling process is the process associated with the thread that triggers the matched probe.

Example 8-35 How to use progenyof to limit a clause to list the write system calls for all child processes of a specified process ID

```
syscall::write:entry
/progenyof($1)/
{
   @[pid,execname,probefunc]=count()
}
```

This script could be run as follows, to monitor all the system calls that are triggered by a running instance of an application, such as the gnome-terminal-server:

```
sudo dtrace -n 'syscall::write:entry /progenyof($1)/
{@[pid,execname,probefunc]=count()}' $(pidof gnome-terminal-server)
```

quantize

Stores a power-of-two frequency distribution of the values of the specified expressions in an aggregation. An optional increment can be specified.

```
void quantize(expr [, uint32_t incr])
```

The `quantize` function is an aggregation function to distribution of information in a histogram for an expression, *expr*. An optional integer value, *incr*, can be specified to find the amount that the values are incremented by to weight the output. This function makes it easier to see a graphical representation of the values returned by an expression.

The rows for the frequency distribution are always power-of-two values. Each row indicates a count of the number of elements that are greater than or equal to the corresponding value, but less than the next larger row's value.

Example 8-36 How to use quantize to display the distribution of write() call times by process

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

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

Output similar to the following is displayed after the program exits:

```
  bash
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
           32768 |                                         0
           65536 |                                         0
          131072 |@@@@@@@@@                                1
          262144 |                                         0

  gnome-terminal
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@                          5
           16384 |@@@@@@@@@@@@@@@                          5
           32768 |@@@@@@@@@@@@                             4
           65536 |@@@                                      1
          131072 |                                         0

  Xorg
           value  ------------- Distribution ------------- count
```

```
           2048 |                                         0
           4096 |@@@@@@@                                  4
           8192 |@@@@@@@@@@@@@@                           8
          16384 |@@@@@@@@@@@@@                            7
          32768 |@@@                                      2
          65536 |@@                                       1
         131072 |                                         0
         262144 |                                         0
         524288 |                                         0
        1048576 |                                         0
        2097152 |@@@                                      2
        4194304 |                                         0

  firefox
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@                                      22
            8192 |@@@@@@@@@@@@                             90
           16384 |@@@@@@@@@@@@@@                           107
           32768 |@@@@@@@@@@                               72
           65536 |@@@                                      28
          131072 |                                         3
          262144 |                                         0
          524288 |                                         1
         1048576 |                                         1
         2097152 |                                         0
...
```

raise

Sends a specified signal to the running process.

```
void raise(int)
```

The `raise` function is a destructive function that sends the specified signal to the running process. This function is similar to using the `kill` command to send a signal to the process. The `raise` function can be used to send a signal at a precise point in the runtime of the process.

See the `sigaction(2)` and `kill(1)` manual pages for more information on how process signals work.

Example 8-37 How to use raise to stop a running process

The script opens with a pragma to enable destructive functions in DTrace. You can alternatively remove this line and run the script with `dtrace -w`. The predicate for this script evaluates the process id against a provided argument. The clause includes the raise function with a SIGINT signal that stops the process immediately.

```
#pragma D option destructive
syscall:::
/pid==$1/
{
   raise(SIGINT);
```

```
   exit(0)
}
```

You must provide the process ID that you intend to stop for this script to function correctly. An example test run might be as follows:

```
xclock & sudo dtrace -wn 'syscall::: /pid==$1/{ raise(SIGINT); exit(0) }' $
(pidof xclock)
```

rand

Returns a pseudo random integer.

```
int rand(void)
```

The `rand` function returns a pseudo random integer. The value returned is a weak pseudo random number and we don't recommend using it for any cryptographic application.

Example 8-38 How to use rand to generate a pseudo random integer

The example uses the trace function to print the generated integer in the trace output.

```
BEGIN{
   trace(rand());
}
```

rindex

Finds the last occurrence of a specific substring within a string.

```
int rindex(const char * str, const char * substr[, int start])
```

The `rindex` function finds the position of the last occurrence of a substring, *substr*, in a string, *str*, starting at an optional position, *start*. If the specified value of start position is less than 0, it's implicitly set to `0`. If the string is an empty string, `rindex` returns `0`. If no match is found for the substring within the string, `rindex` returns `-1`.

Example 8-39 How to use rindex to identify the last occurrence of a substring within a string

```
BEGIN {
         x = "#findthelastpenguininthepenguinstring!";
         y = "penguin";
         printf("The last penguin appears at character %3d\n", rindex(x, y));
         exit(0)
 }
```

rw_iswriter

Checks whether a writer is holding or waiting for the specified reader-writer lock.

```
int rw_iswriter(vmlinux`rwlock_t *rwlock)
```

The `rw_iswriter` function returns non zero if a writer is holding or waiting for the specified reader-writer lock (*rwlock*). If the lock is held only by readers and no writer is blocked, or if the lock isn't held at all, `rw_iswriter` returns zero.

Example 8-40 How to use rw_iswriter to check whether a writer is holding or waiting for a specified reader-writer lock

The example contains two clauses. The first clause triggers for when the `_raw_write_lock` is entered, and uses `rw_iswriter` function to print whether a lock is held. At this stage, no lock is held, so the output returns `0`. When the `_raw_write_lock` returns, a lock is held and the `rw_iswriter` function returns `1` and exits.

```
fbt:vmlinux:_raw_write_lock:entry
{
        self->wlock = (rwlock_t *)arg0;
        printf("write entry  %x\n", 0 != rw_iswriter(self->wlock));
}

fbt:vmlinux:_raw_write_lock:return
/self->wlock/
{
        printf("write return %x\n", 0 != rw_iswriter(self->wlock));
        exit(0)
}
```

rw_read_held

Checks whether the specified reader-writer lock is held by a reader.

```
int rw_read_held(vmlinux`rwlock_t *rwlock)
```

The `rw_read_held` function returns non zero if the specified reader-writer lock (*rwlock*) is held by a reader. If the lock is held only by writers or isn't held at all, `rw_read_held` returns zero.

Example 8-41 How to use rw_iswriter to check whether a writer is holding or waiting for a specified reader-writer lock

The example includes two clauses. The first clause triggers for when the `_raw_read_lock` is entered, and uses `rw_read_held` function to print whether a lock is held. At this stage, no lock is held, so the output returns 0. When the `_raw_read_lock` returns, a lock is held and the `rw_read_held` function returns 1.

```
fbt:vmlinux:_raw_read_lock:entry
{
        self->rlock = (rwlock_t *)arg0;
        printf("read  entry  %x\n", 0 != rw_read_held(self->rlock));
```

```
}

fbt:vmlinux:_raw_read_lock:return
/self->rlock/
{
        printf("read  return %x\n", 0 != rw_read_held(self->rlock));
        exit(0);
}
```

rw_write_held

Checks whether the specified reader-writer lock is held by a writer.

```
int rw_write_held(vmlinux`rwlock_t *rwlock)
```

The `rw_write_held` function returns non zero if the specified reader-writer lock (*rwlock*) is held by a writer. If the lock is held only by readers or isn't held at all, `rw_write_held` returns zero.

Example 8-42 How to use rw_write_held to check whether a writer is holding a specified reader-writer lock

The example uses two clauses. The first clause triggers for when the `_raw_write_lock` is entered, and uses `rw_write_held` function to print whether a write lock is held. At this stage, no lock is held, so the output returns `0`. When the `_raw_write_lock` returns, a lock is held and the `rw_write_held` function returns `1` and the script exits.

```
fbt:vmlinux:_raw_write_lock:entry
{
        self->wlock = (rwlock_t *)arg0;
        printf("write entry  %x\n", 0 != rw_write_held(self->wlock));
}

fbt:vmlinux:_raw_write_lock:return
/self->wlock/
{
        printf("write return %x\n", 0 != rw_write_held(self->wlock));
        exit(0)
}
```

setopt

Dynamically sets DTrace compiler or runtime options.

```
void setopt(const char *[, const char *])
```

The `setopt` function is a special function that can be used to specify a DTrace runtime or compiler option dynamically. See DTrace Runtime and Compile-time Options Reference for more information.

Example 8-43 How to use setopt to set compiler or runtime options inside a program

```
setopt("quiet");
setopt("bufsize", "50m");
setopt("aggrate", "2hz");
```

speculate

A special function that causes DTrace to switch to using a speculation buffer identified by the specified ID for the remainder of a clause.

```
void speculate(int)
```

The `speculate` function is a special function that causes DTrace to use a speculative buffer specified by the provided *id* for the remainder of a clause.

To use a speculation, an identifier that's returned from `speculation` must be passed to the `speculate` function in a clause before any data-recording functions. All subsequent data-recording functions in a clause containing a `speculate` are speculatively traced. The D compiler generates a compile-time error if a call to `speculate` follows data-recording functions in a D probe clause. Therefore, clauses might contain speculative tracing or non-speculative tracing requests, but not both.

Aggregating functions, destructive functions, and the `exit` function can never be speculative. Any attempt to take one of these functions in a clause containing a `speculate` results in a compile-time error. Also, a `speculate` can't follow a `speculate`. Only one speculation is permitted per clause. A clause that contains only a `speculate` speculatively traces the default function, which is defined to trace only the enabled probe ID.

Example 8-44 How to use speculation

The following example illustrates how to use speculation. All speculation functions must be used together for speculation to work correctly.

The speculation is created for the `syscall::open:entry` probe and the ID for the speculation is attached to a thread-local variable. The first argument of the `open()` system call is traced to the speculation buffer by using the `printf` function.

Three more clauses are included for the `syscall::open:return` probe. In the first of these clauses, the `errno` is traced to the speculative buffer. The predicate for the second of the clauses filters for a non-zero `errno` value and commits the speculation buffer. The predicate of the third of the clauses filters for a zero `errno` value and discards the speculation buffer.

The output of the program is returned for the primary data buffer, so the program effectively returns the file name and error number when an `open()` system call fails. If the call doesn't fail, the information that was traced into the speculation buffer is discarded.

```
syscall::open:entry
{
  /*
   * The call to speculation() creates a new speculation. If this fails,
   * dtrace 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 primary data buffer if
the
   * speculation is subsequently committed.
   */
  printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/
{
  /*
   * Trace the errno value into the speculation buffer.
   */
  speculate(self->spec);
  trace(errno);
}

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

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

speculation

Creates a speculative trace buffer and returns its ID.

```
int speculation(void)
```

The `speculation` function reserves a speculative trace buffer for use with speculate and returns an identifier for this buffer.

Example 8-45 How to use speculation

The following example illustrates how to use speculation. All speculation functions must be used together for speculation to work correctly.

The speculation is created for the `syscall::open:entry` probe and the ID for the speculation is attached to a thread-local variable. The first argument of the `open()` system call is traced to the speculation buffer by using the `printf` function.

Three more clauses are included for the `syscall::open:return` probe. In the first of these clauses, the `errno` is traced to the speculative buffer. The predicate for the second of the clauses filters for a non-zero `errno` value and commits the speculation buffer. The predicate of the third of the clauses filters for a zero `errno` value and discards the speculation buffer.

The output of the program is returned for the primary data buffer, so the program effectively returns the file name and error number when an `open()` system call fails. If the call doesn't fail, the information that was traced into the speculation buffer is discarded.

```
syscall::open:entry
{
  /*
   * The call to speculation() creates a new speculation. If this fails,
   * dtrace 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 primary data buffer if
the
   * speculation is subsequently committed.
   */
  printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/
{
  /*
   * Trace the errno value into the speculation buffer.
   */
  speculate(self->spec);
  trace(errno);
}

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

syscall::open:return
/self->spec && errno == 0/
{
```

```
  /*
   * If errno is not set, discard the speculation.
   */
  discard(self->spec);
  self->spec = 0;
}
```

stack

Records a stack trace to the buffer.

```
stack stack([uint32_t frames])
```

The `stack` function records a kernel stack trace to the directed buffer. The function includes an option to specify the number of frames deep to record from the kernel stack. If no value is specified, the number of stack frames recorded is the number that's specified by the `stackframes` runtime option. The `dtrace` command reports frames, either up to the root frame or until the specified limit has been reached, whichever comes first.

The `stack` function, having a non-`void` return value, can also be used as the key to an aggregation.

Example 8-46 How to use stack to obtain a kernel stack trace for a particular probe

```
fbt::ksys_write:entry
{
        stack();
        exit(0);
}
```

stddev

Stores the standard deviation of the specified expressions in an aggregation.

```
void stddev(expr)
```

The `stddev` function is an aggregation function that returns the standard deviation for an expression.

The standard deviation is imprecisely approximated as $\sqrt{((\Sigma(x^2)/N) - (\Sigma x/N)^2)}$. This value is sufficient for most DTrace purposes.

Example 8-47 How to use stddev to display the standard deviation of time taken to run processes

The example stores the timestamp for the `syscall::execve:entry` probe fires and then subtracts this value from the timestamp when the `syscall::execve:return` fires. The standard deviation is calculated based on the time difference between the two probes and stored in an aggregation so that it can be updated for each process that runs. When the program exits, the aggregated standard deviation value is displayed.

```
syscall::execve:entry
{
```

```
 self->ts = timestamp;
}

syscall::execve:return
/ self->ts /
{
  t = timestamp - self->ts;
  @execsd[execname] = stddev(t);
  self->ts = 0;
}

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

Output similar to the following is displayed when the program exits:

```
STDDEV:
  head                                                    0
  lsb_release                                             0
  mkdir                                                   0
  pidof                                                   0
  pkla-check-auth                                         0
  tr                                                      0
  uname                                                   0
  getopt                                               5646
  basename                                             7061
  sed                                                  7236
```

strchr

Returns a substring that begins at the first matching occurrence of a specified character in a string.

```
string strchr(const char *string, char char)
```

The `strchr` function returns a substring that matches the first occurrence of a specified character, *char*, in the specified string, *string*. If no match is found, `strstr` returns `0`. Note that this function doesn't work with wide characters or multibyte characters.

The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `strchr` doesn't run and an error is generated.

Example 8-48 How to use strchr to return a string starting at the first occurrence of a character

```
 BEGIN
 {
         str = "fooeyfooeyfoo";
         c = 'y';
         # the following line prints "yfooeyfoo"
```

```
        printf("\"%s\"\n", strchr(str, c));
        exit(0)
 }
```

strjoin

Concatenates two specified strings and returns the resulting string.

```
string strjoin(const char *string1, const char *string2)
```

The `strjoin` function returns the concatenation of two specified strings. The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `strjoin` doesn't run and an error is generated.

Example 8-49 How to use strjoin to concatenate two strings together

```
BEGIN {
     string1="foo";
     string2="bar";
     printf("%s",strjoin(string1,string2));
     exit(0);
 }
```

strlen

Returns the length of a string in bytes.

```
size_t strlen(const char *string)
```

The `strlen` function returns the length of a specified string in bytes, excluding the terminating null byte.

Example 8-50 How to use strlen to return the length of a string

```
BEGIN {
     string1="foo bar?";
     printf("%d",strlen(string1));
     exit(0);
 }
```

strrchr

Returns a substring that begins at the last matching occurrence of a specified character in a string.

```
string strrchr(const char *, char)
```

The `strrchr` function returns a substring that begins at the last occurrence of a matching character in a specified string. If no match is found, `strrchr` returns `0`. This function doesn't work with wide characters or multibyte characters.

The returned string is allocated out of scratch memory and is therefore valid only during processing of the clause. If insufficient scratch memory is available, `strrchr` doesn't run and an error is generated.

Example 8-51 How to use strrchr to return the pointer to the last occurrence of a character

```
BEGIN
{
        str = "fooeyfooeyfoo";
        c = 'y';
        # the following line prints "yfoo"
        printf("\"%s\"\n", strrchr(str, c));
        exit(0)
}
```

strstr

Returns a substring starting at first occurrence of a specified substring within a string.

```
string strstr(const char *string, const char *substring)
```

The `strstr` function returns a substring starting at the first occurrence of a specified substring in the specified string. If the specified string is empty, `strstr` returns an empty string. If no match is found, `strstr` returns `0`.

Example 8-52 How to use strstr to return a substring starting at the first occurrence of a substring in a string

```
 BEGIN {
     string1="foo bar?";
     substring=" ba";
     # the following line prints " bar?"
     printf("%s",strstr(string1,substring));
     exit(0);
 }
```

strtok

Parse a string into a sequence of tokens using a specified delimiter.

```
string strtok(const char *string, const char *delimiter)
```

The `strtok` function parses a string into a sequence of tokens by using a specified delimiter as the delimiting string . When you initially call `strtok`, specify the string to be parsed. In each following call to obtain the next token, specify the string as NULL. You can specify a different delimiter for each call. The internal pointer that `strtok` uses to traverse the string is only valid within more than one enabling of the same probe. The `strtok` function returns NULL if no more tokens are found.

Example 8-53 How to use strtok to break a comma delimited string into tokens.

In this example, `strtok` is used to break a comma delimited string into tokens. Because DTrace doesn't include flow-control structures similar to while loops, you must use predicates to emulate this functionality to step through each token. The example, shows how to walk through the first two tokens generated by the string. Each predicate gets the next token and checks that it's not a NULL value, which would represent the end of the string.

```
BEGIN
 {
     this->str = "Carrots,Barley,Oatmeal,Corn,Beans";
 }

 BEGIN
 /(this->field = strtok(this->str, ",")) == NULL/
 {
         exit(1);
 }

 BEGIN
 {
         printf("First token: %s\n", this->field);
 }

 BEGIN
 /(this->field = strtok(NULL, ",")) == NULL/
 {
         exit(2);
 }

 BEGIN
 {
         printf("Second token: %s\n", this->field);
         exit(0)
 }
```

substr

Returns the substring from a string at a specified index position.

```
string substr(const char * string, int index[, int length])
```

The `substr` function returns the substring of a string, *string*, starting at the specified index position, *index*. An optional length parameter, *length*, can be specified to limit the substring to a specified length.

Example 8-54 How to use substr to return a substring from a specified index

In the example, the length of the substring returned is limited to 4 characters.

```
BEGIN {
     string1="daddyorchips";
     trace(substr(string1,7,4))
```

```
        exit(0)
    }
```

sum

Stores the total value of the specified expression in an aggregation.

```
void sum(expr)
```

The `sum` function is an aggregation function to used to obtain the total value of a specified expression, *expr*.

Example 8-55 How to use sum to aggregate a value over a period

This example increments a variable, *i*, by 100 every 10 ms until *i* has a value of 1000. An aggregation is used to calculate the sum of values of *i*. This is equal to the expression: 0+100+200+300+400+500+600+700+800+900=4500.

```
BEGIN
{
        i = 0;
}

tick-10ms
/i < 1000/
{
        @a = sum(i);
        i += 100;
}

tick-10ms
/i == 1000/
{
        exit(0);
}
```

sym

Prints the symbol for a specified kernel space address. An alias for `func`.

```
_symaddr sym(uintptr_t addr)
```

The `sym` function is a data recording function that prints the symbol that corresponds to a specified kernel space address, *addr*. The `sym` function is an alias for func.

Example 8-56 How the sym function can return the symbol for a kernel space address

This example uses a bash script to pick a test symbol from `/proc/kallmodsyms` that can be used as a reference in the DTrace program that returns the symbol for the function.

```
#!/bin/bash
read ADD <<< `awk '/ksys_write/ {print $1}' /proc/kallmodsyms`
dtrace -qn 'BEGIN {sym(0x'$ADD'); exit(0) }'
```

system

Causes a specified program to be run on the system as if within a shell.

```
void system(const char command)
```

The `system` function is a destructive function that causes the specified program to be run as though provided to the shell as input. The program string can contain any of the `printf` or `printa` format conversions. Arguments that match the format conversions must be specified.

Note that a command specified for the `system` function doesn't run in the context of the firing probe. Rather, it occurs when the buffer containing the details of the `system` function are processed at user level.

Example 8-57 How to use system to run the system date command after every second

Note that the pragma lines include the destructive option to permit DTrace to run destructive functions for this example.

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

tick-1sec
{
system("date")
}
```

trace

Traces the result of an expression to the directed buffer.

```
void trace(expr)
```

The `trace` function is the most fundamental DTrace function. This function takes a D expression as its argument and then traces the result to the directed buffer.

If the `trace` function is used on a buffer, the output format depends on the data type. 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's printed as an ASCII string and ends with a null character (`0`). If the data is any other size, and isn't a sequence of printable characters, it's printed as a series of byte values that's formatted as hexadecimal integers.

You can force the `trace` function to always use the binary format by specifying the `rawbytes` dynamic runtime option.

Example 8-58 How to use trace to display a variety of different outputs

The example shows the trace function being used to return output for a built-in variable, an expression, and a string value.

```
BEGIN
{
trace(execname);
trace(timestamp / 1000);
trace("somehow managed to get here");
}
```

tracemem

Copies the specified number of bytes of data from an address in memory to the current buffer.

```
void tracemem(addr, size_t bytes[, size_t limit])
```

The `tracemem` function copies a specified number of bytes of data, *bytes*, from an address in memory, *addr*, to the current buffer. The address that the data is copied from is specified as a D expression. An optional third argument, *limit*, can be used to limit the size of the data that's copied to the buffer. The limit can be a variable amount, but it must be less than or equal to the size of the memory data that you specified to copy from memory, or it's ignored.

Limiting the data that's copied to the buffer is useful when the data that you're copying has a known upper bound, but the actual number of bytes can vary. DTrace statically reserves *bytes* in the output buffer at compile time. You can reserve a larger amount of memory in the output buffer at run time by setting the number of *bytes*, but dynamically control the amount of memory used by specifying a dynamic *limit*.

Example 8-59 How to use tracemem to trace 256 bytes from an address in memory for the current thread

The example creates a pointer to the current thread by using the built-in variable `curthread`.

```
BEGIN {
     p = curthread;
     tracemem(p, 256);
     exit(0);
 }
```

uaddr

Prints the symbol for a specified address.

```
_usymaddr uaddr(uintptr_t)
```

The `uaddr` function prints the symbol for a specified address, including hexadecimal offset, which enables the same symbol resolution that `ustack` provides.

Example 8-60 How to use uaddr to obtain the symbol for an address

```
uaddropenatdateucaller

sudo dtrace -n syscall::openat:entry'/pid == $target/{usym(ucaller);}' -c
'date'
```

Generates output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  5 147861                     openat:entry
libc.so.6`_nl_find_locale
  5 147861                     openat:entry
0x0
Mon 20 Feb 18:11:30 GMT 2023
```

ufunc

Prints the symbol for a specified user space address. An alias for `usym`.

```
_usymaddr ufunc(uintptr_t)
```

The `ufunc` function is a data recording function that prints the symbol that corresponds to a specified user space address. The `func` function is an alias for usym.

Example 8-61 How to use usym to obtain the symbol for an address

```
usymopenatdateucaller

sudo dtrace -n syscall::openat:entry'/pid == $target/{usym(ucaller);}' -c
'date'
```

Generates output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  2 147861                     openat:entry
libc.so.6`_nl_find_locale
Mon 20 Feb 18:12:58 GMT 2023
  2 147861                     openat:entry   0x0
```

umod

Prints the module name that corresponds to a specified user space address.

```
_usymaddr umod(uintptr_t)
```

The `umod` function is a data recording function that prints the name of the module that corresponds to a specified user space address.

Example 8-62 How to use umod to print the module name for an address

The example shows how to use `umod` to print the module names for `openat` system calls by the `date` command.

```
sudo dtrace -qn syscall::openat:entry'/pid == $target/{umod(ucaller);}' -c
'date'
```

Generates output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  7 147861                     openat:entry
libc.so.6
  7 147861                     openat:entry
0x0
Mon 20 Feb 18:07:43 GMT 2023
```

ustack

Records a user stack trace to the directed buffer.

```
stack ustack([uint32_t nframes, uint32_t strsize])
```

The `ustack` function records a user stack trace to the directed buffer. The user stack is, at most, *nframes* in depth. If *nframes* isn't specified, the number of stack frames recorded is the number specified by the `ustackframes` option. While `ustack` can determine the address of the calling frames when the probe fires, the stack frames aren't translated into symbols until the `ustack` function is processed at user level by the DTrace utility. If *strsize* is specified and is non zero, `ustack` allocates the specified amount of string space and then uses it to perform address-to-symbol translation directly from the kernel. Such direct user symbol translation is used only with stacktrace helpers that support this usage with DTrace. If such frames can't be translated, the frames appear only as hexadecimal addresses.

The `ustack` symbol translation 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` outputs a warning message, followed by the hexadecimal stack frames.

Example 8-63 How to use ustack to trace a stack with no address-to-symbol translation

The example shows how to use `ustack` to trace the stack for an `openat` system call by the `date` command.

```
sudo dtrace -qn syscall::openat:entry'/pid == $target/{ustack();}' -c 'date'
```

Generates output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  2 147861                     openat:entry
              libc.so.6`__open64_nocancel+0x45
Mon 20 Feb 17:38:15 GMT 2023
              libc.so.6`_nl_find_locale+0xfc
```

```
              libc.so.6`setlocale+0x1cf
              date`0x556ebae140ad
              0x7a696c616d726f6e

  2 147861                      openat:entry
              0x7f6d63fc2e65
```

usym

Prints the symbol for a specified address. An alias for `ufunc`.

```
_usymaddr usym(uintptr_t)
```

The `usym` function prints the symbol for a specified address, which is analogous to how uaddr works, but without the hexadecimal offsets. The `usym` function is an alias for ufunc.

Example 8-64 How to use usym to obtain the symbol for an address

```
usymopenatdateucaller

sudo dtrace -n syscall::openat:entry'/pid == $target/{usym(ucaller);}' -c
'date'
```

Generates output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  2 147861                     openat:entry
libc.so.6`_nl_find_locale
Mon 20 Feb 18:12:58 GMT 2023
  2 147861                     openat:entry   0x0
```

9
DTrace Provider Reference

DTrace exposes different providers that publish probes that are grouped together for particular instrumentation or functionality.

CPC Provider

The CPU performance counter (`cpc`) provider makes available probes that are associated with CPU performance counter events.

A probe fires when a specified number of events of a type in a chosen processor mode has occurred. When a probe fires, you can sample aspects of system state and make inferences about system behavior. A reasonable value for the event counter value depends on the event and also on the workload. To keep probe firings from being excessive, start with a high value. Lower the value to improve statistical accuracy.

CPU performance counters are a finite resource and the number of probes that can be enabled depends upon hardware capabilities. An error is returned when the number of `cpc` probes enabled exceed the hardware capability. If hardware resources are unavailable, probes fail until resources become available.

Start with higher event counter values for CPC probes and reduce them through trial-and-error as you work toward a more accurate representation of system activity.

cpc Probes

Probes made available by the `cpc` provider have the following probe description format:

```
cpc:::<event name>-<mode>-<count>
```

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

Table 9-1 Probe Name Components

Component	Meaning
event name	The platform specific or generic event name.
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.
count	The number of events that must occur on a CPU for a probe to be fired on that CPU. Note that the count is a configurable value. If the count value is too high, then the probe fires less often and the statistics are less reliable. If the count value is too low, the probe fires too often and the system is inundate with tracing activity. When selecting a count value, start with a higher value and then decrease it to get more accurate statistics.

Note that when you list CPC probes, example count values are provided in the probe listings. The count values are artificially set high as a guideline.

cpc Probe Arguments

The following table lists the argument types for the `cpc` probes.

Table 9-2 Probe Arguments

`arg0`	The program counter (PC) in the kernel at the time that the probe fired, or 0 if the current process wasn't running 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 running 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.

cpc Examples

The following example illustrates the use of a probe published by the `cpc` provider.

cycles-all-50000000

The example performs a count for each process name that triggers the performance counter probe on a count value of 50000000.

```
cpc:::cycles-all-50000000
 {
         @[execname] = count();
 }
```

cpc Stability

The `cpc` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

DTrace Provider

The `dtrace` provider includes several probes that are specific to DTrace itself.

Use these probes to initialize state before tracing begins, process state after tracing has completed, and to 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 have completed. This probe can be used to initialize any state that's needed 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:

```
dtrace:::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, which means the output of `stack` or `ustack`, and the value of context-specific variables such as `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 doesn't 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. The `printa` function is therefore often used in the `END` probe. The `BEGIN` and `END` probes can be used together to measure the total time that's spent tracing, for example:

```
dtrace:::BEGIN
{
  start = timestamp;
}

/*
```

```
 * ... other tracing functions...
 */

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

As with the `BEGIN` probe, no arguments are defined for the `END` probe. The context in which the `END` probe fires is arbitrary and can't be depended upon.

> **Note:**
>
> The exit function causes tracing to stop and the `END` probe to fire. However, a delay exists between the invocation of the `exit` function and when the `END` probe fires. During this delay, no further probes can fire. After a probe invokes the `exit` function, the `END` probe isn't fired until DTrace determines that `exit` has been called and stops tracing. The rate at which the exit status is checked can be set by using `statusrate` option.

ERROR Probe

The `ERROR` probe fires when a runtime error occurs during the processing of a clause for a DTrace probe.

When a runtime error occurs, DTrace doesn't process the rest of the clause that resulted in the error. If an ERROR probe is included in the script, it's triggered immediately. After the ERROR probe is processed, tracing continues. If you want a D runtime error to stop all further tracing, you must include an `exit()` action in the clause for the ERROR probe.

In the following example, a clause attempts to dereference a `NULL` pointer and causes the `ERROR` probe to fire. Save it in a file named `error.d`:

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

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

When you run this program, output similar to the following is displayed:

```
dtrace: script 'error.d' matched 2 probes
dtrace: error on enabled probe ID 3 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #1 at BPF pc 142
CPU     ID                    FUNCTION:NAME
  0      3                           :ERROR Hit an error!
```

The output indicates that the ERROR probe fired and that dtrace reported the error. dtrace has its own enabling of the ERROR probe so that it can report errors. Using the ERROR probe, you can create custom error handling.

The arguments to the ERROR probe are described in the following table.

Argument	Description
arg1	The enabled probe identifier (EPID) of the probe that caused the error.
arg2	The index of the action that caused the fault.
arg3	The DIF offset into the action or -1 if not applicable.
arg4	The fault type.
arg5	Value that's particular to the fault type.

The following table describes the various fault types that can be specified in arg4 and the values that arg5 can take for each fault type.

arg4 Value	Description	arg5 Meaning
DTRACEFLT_UNKNOWN	Unknown fault type	None
DTRACEFLT_BADADDR	Access to unmapped or invalid address	Address accessed
DTRACEFLT_BADALIGN	Unaligned memory access	Address accessed
DTRACEFLT_ILLOP	Illegal or invalid operation	None
DTRACEFLT_DIVZERO	Integer divide by zero	None
DTRACEFLT_NOSCRATCH	Insufficient scratch memory to satisfy scratch allocation	None
DTRACEFLT_KPRIV	Attempt to access a kernel address or property without sufficient privileges	Address accessed or 0 if not applicable
DTRACEFLT_UPRIV	Attempt to access a user address or property without sufficient privileges	Address accessed or 0 if not applicable
DTRACEFLT_TUPOFLOW	DTrace internal parameter tuple stack overflow	None
DTRACEFLT_BADSTACK	Invalid user process stack	Address of invalid stack pointer
DTRACEFLT_BADSIZE	Invalid size fault that appears when an invalid size is passed to a function such as alloca(), bcopy() or copyin().	The invalid size.
DTRACEFLT_BADINDEX	Index out of bounds in a scalar array.	The index that was specified.
DTRACEFLT_LIBRARY	Library level fault	None.

If the actions that are taken in the ERROR probe cause an error, that error is silently dropped. The ERROR probe isn't recursively invoked.

dtrace Stability

The `dtrace` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

Element	Name Stability	Data Stability	Dependency Class
Provider	Stable	Stable	Common
Module	Private	Private	Unknown
Function	Private	Private	Unknown
Name	Stable	Stable	Common
Arguments	Stable	Stable	Common

FBT Provider

The `fbt` (Function Boundary Tracing) provider includes probes that are associated with the entry to and return from most functions in the Oracle Linux kernel. Therefore, there could be tens of thousands of `fbt` probes.

While the FBT implementation is highly specific to the instruction set architecture, FBT has been implemented on both x86 and 64-bit Arm platforms. Some functions in each instruction set are highly optimized by the compiler and can't be instrumented by FBT. Probes for these functions aren't present in DTrace, but you can check what's available by running:

```
sudo dtrace -lP fbt
```

An effective use of FBT probes requires knowledge of the kernel implementation. Therefore, we recommend that you use FBT only when developing kernel software or when other providers aren't sufficient.

Because of the large number of FPB probes that are available, be specific about the modules and functions that you enable probes for. Performance can be impacted when the full range of FBT probes are enabled at the same time.

fbt Probes

FBT provides an `entry` probe and a `return` probe for most functions in the kernel.

fbt Probe Arguments

The arguments to `entry` probes are the same as the arguments to the corresponding operating system kernel function. These arguments can be accessed as `int64_t` values by using the `arg0`, `arg1`, `arg2`, ... variables.

If the function has a return value, the return value is stored in `arg1` of the `return` probe. If a function doesn't have a return value, `arg1` isn't defined.

fbt Examples

You can use the `fbt` provider to explore the kernel's implementation. The following example script creates an aggregation on the number of times different functions allocate kernel virtual

memory. The results of the aggregation are printed when the script exits. This would help somebody to monitor what functions are memory intensive. Type the following D source code and save it in a file named `getkmemalloc.d`:

```
#pragma D option quiet
fbt::kmem*alloc*:entry
{
  @[caller] = count();
}
dtrace:::END
{
  printa("%40a %@10d\n", @);
}
```

Running this script results in output similar to the following:

```
              vmlinux`vm_area_alloc+0x1a          1
           vmlinux`__sigqueue_alloc+0x65          1
         vmlinux`__create_xol_area+0x4d           1
         vmlinux`__create_xol_area+0x6f           1
              vmlinux`vmstat_start+0x39           1
          vmlinux`proc_alloc_inode+0x1d           1
        vmlinux`proc_self_get_link+0x5b           1
      vmlinux`security_inode_alloc+0x24           1
            vmlinux`avc_alloc_node+0x1c           1
      vmlinux`ep_ptable_queue_proc+0x3d           2
           vmlinux`kernfs_fop_open+0xbf           2
         vmlinux`kernfs_fop_open+0x2e8            2
          vmlinux`disk_seqf_start+0x25            2
             vmlinux`__alloc_skb+0x16c            6
                vmlinux`skb_clone+0x4b            6
                vmlinux`ep_insert+0xbb            8
              vmlinux`ep_insert+0x34c             8
                vmlinux`__d_alloc+0x29            9
       vmlinux`kernfs_iop_get_link+0x33           9
              vmlinux`single_open+0x2a           15
            vmlinux`proc_reg_open+0x6e           17
                 vmlinux`seq_open+0x2a           21
             vmlinux`__alloc_file+0x23           29
       vmlinux`security_file_alloc+0x24          29
      vmlinux`getname_flags.part.0+0x2c          40
```

The output shows the internal kernel functions that are making calls to the `kmem*alloc` system calls and can be used to find which kernel functions most often allocate kernel virtual memory on a system.

fbt Stability

The `fbt` provider uses DTrace's stability mechanism to describe its stabilities. These stability values are listed in the following table.

Element	Name Stability	Data Stability	Dependency Class
Provider	Evolving	Evolving	Common

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

IO Provider

The `io` provider makes available probes that relate to data input and output.

For example, you can use the `io` provider to understand I/O by device, I/O type, I/O size, process, or application name.

io Probes

The following table describes the probes for the `io` provider. For all `io` probes, the module is `vmlinux` and the function is an empty string.

Table 9-3 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.
`done`	Fires after an I/O request has been fulfilled. The done probe fires after the I/O completes, but before completion processing has been performed on the buffer. `B_DONE` isn't set in `b_flags` at the time the done probe fires.
`wait-start`	Fires immediately before a thread begins to wait pending completion of an I/O request. Some time after the `wait-start` probe fires, the `wait-done` probe fires in the same thread.
`wait-done`	Fires when a thread finishes waiting for the completion of an I/O request. The `wait-done` probe fires only after the `wait-start` probe has fired in the same thread.

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, don't trigger `io` probes because of a `readdir()` request.

io Probe Arguments

The following table describes the arguments for the `io` probes. The `argN` are implementation specific. Use `args[]` to access the probe arguments.

Table 9-4 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 *

> **Note:**
>
> DTrace doesn't provide the option to use `fileinfo_t` with `io` probes. In Oracle Linux, no information is accessible at the level where the `io` probes fire about the file where an I/O request originated.

bufinfo_t

The `bufinfo_t` structure is the abstraction that describes an I/O request. The buffer that corresponds to an I/O request is pointed to by `args[0]` in the `start`, `done`, `wait-start`, and `wait-done` probes. Detailed information about this data structure can be found in `/usr/lib64/dtrace/version/io.d`. The definition of `bufinfo_t` 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_lblkno;    /* logical block # on device */
  uint64_t b_blkno;     /* expanded block # on device */
  size_t b_resid;       /* not supported */
  size_t b_bufsize;     /* size of allocated buffer */
  caddr_t b_iodone;     /* I/O completion routine */
  int b_error;          /* not supported */
  dev_t b_edev;         /* extended device */
} bufinfo_t;
```

> **Note:**
>
> DTrace translates the members of `bufinfo_t` from the `buffer_head` or `bio` for the Oracle Linux I/O request structure, depending on the kernel version.

`b_flags` indicates the state of the I/O buffer, and consists of a bitwise-or of different state values. The following table describes the values for the states.

Table 9-5 b_flags Values

b_flags	Value	Description
B_ASYNC	0x000400	Indicates that the I/O request is asynchronous and isn't waited upon. The wait-start and wait-done probes don't fire for asynchronous I/O requests. Some I/Os directed to be asynchronous might not set B_ASYNC. The asynchronous I/O subsystem could implement the asynchronous request by having a separate worker thread perform a synchronous I/O operation.
B_BUSY	0x000001	
B_DONE	0x000002	
B_ERROR	0x000004	
B_PAGEIO	0x000010	Indicates that the buffer is being used in a paged I/O request.
B_PHYS	0x000020	Indicates that the buffer is being used for physical (direct) I/O to a user data area.
B_READ	0x000040	Indicates that data is to be read from the peripheral device into main memory.
B_WRITE	0x000100	Indicates that the data is to be transferred from main memory to the peripheral device.

b_bcount: Is the number of bytes to be transferred as part of the I/O request.

b_addr: Is the virtual address of the I/O request, when known.

b_lblkno: 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.

b_blkno: Identifies which block on the device is to be accessed.

b_bufsize: Contains the size of the allocated buffer.

b_iodone: Identifies a specific routine in the kernel that's called when the I/O is complete.

b_edev: Contains the major and minor device numbers of the device accessed. You can 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 that corresponds to the destination device of an I/O is pointed to by `args[1]` in the `start`, `done`, `wait-start`, and `wait-done` probes. Detailed information about this data structure can be found in `/usr/lib64/dtrace/version/io.d`. The definition of `devinfo_t` is as follows:

```
typedef struct devinfo {
  int dev_major;           /* major number */
  int dev_minor;           /* minor number */
  int dev_instance;        /* not supported */
  string dev_name;         /* name of device */
  string dev_statname;     /* name of device + instance/minor */
  string dev_pathname;     /* pathname of device */
} devinfo_t;
```

> **Note:**
>
> DTrace translates the members of `devinfo_t` from the `buffer_head` for the Oracle Linux I/O request structure.

`dev_major`: Is the major number of the device.

`dev_minor`: Is the minor number of the device.

`dev_name`: Is the name of the device driver that manages the device.

`dev_statname`: Is the name of the device as reported by `iostat`. This field is provided so that aberrant `iostat` output can be quickly correlated to actual I/O activity.

`dev_pathname`: Is the full path of the device. The path that's specified by `dev_pathname` includes components expressing the device node, the instance number, and the minor node. However, note that all three of these elements aren't 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. So, two devices that have the same `dev_statname` migh differ in their `dev_pathname`.

fileinfo_t

> **Note:**
>
> On Oracle Linux, the `fileinfo_t` argument `args[2]` of the `io` probes isn't supported. However, you can use the `fileinfo_t` structure to obtain information about a process's open files by using the built-in variable `fds[]` array.

The `fileinfo_t` structure provides information about a file. The presence of file information is contingent upon the file system providing this information when dispatching I/O requests. Some file systems, especially third-party file systems, might not provide this information. Also, I/O requests might emanate from a file system for which no file information exists. For

example, any I/O from or to file system metadata isn't associated with any one file. Finally, some highly optimized file systems might aggregate I/O from disjoint files into a single I/O request. In this case, the file system might provide the file information either for the file that represents most of the I/O or for the file that represents some I/O. Or, the file system might provide no file information at all in this case.

Detailed information about this data structure can be found in `/usr/lib64/dtrace/version/io.d`. The definition of `fileinfo_t` is as follows:

```
typedef struct fileinfo {
  string fi_name;            /* basename */
  string fi_dirname;         /* not supported */
  string fi_pathname;        /* not supported */
  loff_t fi_offset;          /* offset within file */
  string fi_fs;              /* file system */
  string fi_mount;           /* not supported */
  int fi_oflags;             /* open() flags for file descriptor */
} fileinfo_t;
```

The `fi_name` field contains the name of the file but doesn't include any directory components. If no file information is associated with an I/O, the `fi_name` field is set to the string `<none>`. In some rare cases, the pathname that's associated with a file might be unknown. In this case, the `fi_name` field is set to the string `<unknown>`.

The `fi_dirname` field contains only the directory component of the file name. As with `fi_name`, this string can be set to `<none>`, if no file information is present, or `<unknown>` if the pathname that's associated with the file isn't known.

The `fi_pathname` field contains the full pathname to the file. As with `fi_name`, this string can be set to `<none>`, if no file information is present, or `<unknown>` if the pathname that's associated with the file isn't known.

The `fi_offset` field contains the offset within the file , or `-1`, if either file information isn't present or if the offset is otherwise unspecified by the file system.

The `fi_fs` field contains the name of the file system type, or `<none>`, if no information is present.

The `fi_oflags` field contains the flags that were specified when opening the file.

io Examples

The following example script displays information for every I/O as it's issued. Type the following source code and save it in a file named `iosnoop.d`.

```
#pragma D option quiet

BEGIN
{
  printf("%10s %2s\n", "DEVICE", "RW");
}

io:::start
{
  printf("%10s %2s\n", args[1]->dev_statname,
```

```
  args[0]->b_flags & B_READ ? "R" : "W");
}
```

The output from this script is similar to the following:

```
    DEVICE RW
     dm-00  R
     dm-00  R
     dm-00  R
     dm-00  R
     dm-00  R
     dm-00  R
...
```

You can make the example script slightly more sophisticated by using an associative array to track the time (in milliseconds) spent on each I/O, as shown in the following example:

```
#pragma D option quiet

BEGIN
{
  printf("%10s %2s %7s\n", "DEVICE", "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 %2s %3d.%03d\n", args[1]->dev_statname,
  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 changed script adds a MS (milliseconds) column to the output.

You can aggregate on device, application, process ID, and bytes transferred, then save it in a file named `whoio.d`, 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 that's similar to the following:

```
    DEVICE                  APP        PID           BYTES
     dm-00               evince      14759           16384
     dm-00          flush-252:0       1367           45056
     dm-00                 bash      14758          131072
     dm-00       gvfsd-metadata       2787          135168
     dm-00               evince      14758          139264
     dm-00               evince      14338          151552
     dm-00          jbd2/dm-0-8        390          356352
```

If you're copying data from one device to another, you might want to know if one of the devices acts as a limiter on the copy. To answer this question, you need to know the effective throughput of each device, rather than the number of bytes per second that each device is transferring. For exampe, you can find throughput by using the following script and saving it in a file named `copy.d`:

```
#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]/
{
  /*
   * We want to get an idea of our throughput to this device in KB/sec.
   * What we have, however, is nanoseconds and bytes. That is we want
   * to calculate:
   *
   * bytes / 1024
   * ------------------------
   * nanoseconds / 1000000000
   *
   * But we cannot calculate this using integer arithmetic without losing
   * precision (the denominator, for one, is between 0 and 1 for nearly
   * all I/Os). So we restate the fraction, and cancel:
   *
   * bytes       1000000000      bytes       976562
   * --------- * ------------- = --------- * -------------
   * 1024        nanoseconds     1           nanoseconds
   *
   * This is easy to calculate using integer arithmetic.
   */
  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 previous script for several seconds while copying data from a hard disk to a USB drive yields the following output:

```
sdc1 (/dev/sdc1)

           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         3
             128 |                                         1
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  2257
             512 |                                         1
            1024 |                                         0

 dm-00 (/dev/dm-00)

           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 previous output shows that the USB drive (`sdc1`) is clearly the limiting device. The throughput of `sdc1` is between 256K/sec and 512K/sec, while `dm-00` delivered I/O at anywhere from 8 MB/second to over 64 MB/second.

io Stability

The `io` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

Lockstat Provider

The `lockstat` provider provides probes that can be used to study lock usage and contention.

lockstat Probes

For all `lockstat` probes, the module name is `vmlinux` and the function name is an empty string.

Locks are classified as:

- *Spin*: This is spin wait if there's contention.
- *Adaptive*: This is either spin wait, or else block, if there's contention.
- *Readers-writer*.

The following probes fire when a lock is acquired or released:

- `spin-acquire`.
- `spin-release`.
- `adaptive-acquire`.
- `adaptive-release`.
- `rw-acquire`.
- `rw-release`.

The following probe fires before a lock is acquired if there was contention for the lock and if the probe is enabled:

- `spin-spin`.
- `adaptive-block`, `adaptive-spin`. One probe fires or else the other, depending on which wait is used.
- `rw-spin`.

Finally, an `adaptive-acquire-error` probe indicates an error acquiring an adaptive lock.

lockstat Probe Arguments

The following table lists the argument types for the `lockstat` probes. The `argN` are implementation specific. Use `args[]` to access the probe arguments.

Table 9-6 lockstat Probe Arguments

Probe	args[0]	args[1]	args[2]
adaptive-acquire	struct mutex *	—	—
adaptive-acquire-error	struct mutex *	int	—
adaptive-block	struct mutex *	uint64_t	—
adaptive-release	struct mutex *	—	—

Table 9-6 (Cont.) lockstat Probe Arguments

Probe	args[0]	args[1]	args[2]
adaptive-spin	struct mutex *	uint64_t	—
rw-acquire	struct rwlock *	int	—
rw-release	struct rwlock *	int	—
rw-spin	struct rwlock *	uint64_t	int
spin-acquire	spinlock_t *	—	—
spin-release	spinlock_t *	—	—
spin-spin	spinlock_t *	uint64_t	—

> **Note:**
>
> `args[0]` has a pointer to the lock in question. The probes that fire in case of contention report a `uint64_t args[1]`, which is the wait time in nanoseconds. The `rw` probes also report an `int` that's either `RW_READER` or `RW_WRITER`. Finally, `adaptive-acquire-error` reports an `int` with a non zero error.

lockstat Examples

The following examples illustrate the use of the probes that are published by the `lockstat` provider.

adaptive-acquire and spin-acquire

Type the following D source code and save it in a file named `whatlock.d`:

```
lockstat:::spin-acquire,
lockstat:::adaptive-acquire
/pid == $target/
{
  @locks[probename] = count();
}
```

Run the program on the `date` command using `sudo dtrace -qs whatlock.d -c date`. The D output looks similar to:

```
 adaptive-acquire                                                6
 spin-acquire                                                  134
```

It might be surprising that so many locks are acquired with the `date` command. The large number of locks is a natural artifact of the fine-grained locking required of a scalable system such as the Oracle Liux kernel.

lockstat Stability

The `lockstat` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

Pid Provider

The `pid` provider traces a user process, both function `entry` and `return`, and an arbitrary instruction.

pid Probes

A probe is fully specified by naming its provider, module, function, and name.

A `pid` provider name is `pid`*`$pid`*, where *`$pid`* is the process ID (pid) of the process you're interested in. The process ID must be specified (no wild cards), but symbolic values can be used. For example:

```
sudo dtrace -lP pid
```

The previous command doesn't find any probes because no process ID was specified. Instead, use commands such as:

```
sudo dtrace -ln 'pid1234567:libc::entry'
sudo dtrace -p 1234567 -ln 'pid$target:libc::entry'
sudo dtrace -c ./a.out -n 'pid$target:libc::entry { trace("hi"); }'
sudo dtrace -n 'pid$1:libc::entry { trace("hi"); }'
```

In this example, `1234567` is a fictious pid value that should be replaced with a pid value appropriate to the system, `$target` is a macro that expands to the pid of the target command specified by `-p` or `-c`, and `$1` is the command line argument to `dtrace`. In this example, the following is displayed:

```
1234567
```

The module name is the module within the executable. Special cases include the load object of the executable, which might be referred to as `a.out`, and a shared library that can be referred to by:

- A full pathname, for example, `/usr/lib/libc.so.1`.
- A basename, for example, `libc.so.1`.

- An initial basename match up to a `.` suffix, for example, `libc.so` or `libc`.

The function name is typically the name of the function where the probe is located. If a function is inlined by the compiler, it's not available for pid tracing.

There is a special function `-`. In this case, the module name must be blank or refer to the `a.out` module. Further, the probe name must be an absolute hexadecimal offset to some instruction within the `a.out` module.

The probe name is one of:

- `entry`: refers to the entry to the associated function.
- `return`: refers to a return from the associated function. In traditional implementations of DTrace, the probe fired at some return instruction. In the current implementation, a `return` is implemented with a `uretprobe`, firing in the caller function.
- An instruction offset. The hexadecimal offset, without a leading `0x`, is relative to the named function, but it's an absolute offset when the function name is `-`.

pid Probe Arguments

For `entry` probes, the probe arguments are the same arguments as those of the probed function.

For `return` probes, `arg1` is the return value of the probed function.

For offset probes, there are no probe arguments.

pid Examples

Consider the following program, named `main.c`, that calls a function `foo()`:

```
int foo(int i, int j) {
    return (i + j) - 6666;
}

int main(int c, char **v) {
    return foo(1234, 8765) != 3333;
}
```

The arguments to `foo()` are `1234` and `8765`, while the return value is `3333`.

Compile the program:

```
gcc main.c
```

We create a D script named `D1.d`:

```
pid$target:a.out:foo:entry,
pid$target:a.out:foo:return
{
    printf("%x %s:%s\n", uregs[R_PC], probefunc, probename);
}

pid$target:a.out:foo:entry
```

```
{
    printf("entry args: %d %d\n", arg0, arg1);
}

pid$target:a.out:foo:return
{
    printf("return arg: %d\n", arg1);
}
```

Run the D script:

```
sudo dtrace -c ./a.out -qs D1.d
```

The output looks similar to:

```
401106 foo:entry
entry args: 1234 8765
40113d foo:return
return arg: 3333
```

On `foo() entry` and `return`, we print the PC from the target thread's saved user-mode register values at probe firing time, along with the probe function and name. In addition, we print the entry probe's two arguments and the return probe's `arg1`. We see that the `foo()` entry arguments are `1234` and `8765`, and the return value is `3333`, as expected.

To understand the PCs, run `objdump`:

```
objdump -d a.out
```

The output looks similar to:

```
0000000000401106 <foo>:
  401106:    55                       push   %rbp
  401107:    48 89 e5                 mov    %rsp,%rbp
  40110a:    89 7d fc                 mov    %edi,-0x4(%rbp)
  [...]

000000000040111f <main>:
  40111f:    55                       push   %rbp
  [...]
  401138:    e8 c9 ff ff ff           callq  401106 <foo>
  40113d:    83 f0 01                 xor    $0x1,%eax
  [...]
```

Much of the output has been suppressed, but we see the `foo() entry` PC is `0x401106`, as reported by our D script. The return PC is `0x40113d`, which is the PC immediately after the `foo()` call.

> **Note:**
>
> In other versions of DTrace, the return PC is for the return instruction in the called function. On Linux, we use a return `uprobe` (a `uretprobe`) which returns an instruction in the caller, as we saw.

Finally, we illustrate how to probe on a specific instruction. We select the third instruction in `foo()`, PC `0x40110a`. This is at a relative offset of 4 bytes from the start of `foo()`. This D script is named D2.d:

```
pid$target:a.out:foo:4,
pid$target:a.out:-:40110a
{
    printf("%x %s:%s\n", uregs[R_PC], probefunc, probename);
}
```

Run the D script:

```
sudo dtrace -c ./a.out -qs D2.d
```

The output looks similar to:

```
40110a foo:4
40110a -:40110a
```

We probe on the chosen instruction, using both a relative offset `foo:4`, and an absolute offset `-:40110a`. Both probes fire, both reporting the same PC `0x40110a`.

pid Stability

The `pid` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

Proc Provider

The `proc` provider makes available the probes that pertain to the following activities: process creation and termination, LWP creation and termination, execution of new program images, and signal sending and handling.

proc Probes

The probes for the `proc` provider are listed in the following table.

Table 9-7 proc Probes

Probe	Description
`create`	Fires when a process (or process thread) is created using `fork()` or `vfork()`, which both invoke `clone()`. The `psinfo_t` corresponding to the new child process is pointed to by `args[0]`.
`exec`	Fires whenever a process loads a new process image using a variant of the `execve()` system call. 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` or `exec-success` probe subsequently fires 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 that 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 <asm-generic/signal.h>` codes, is contained in `args[0]`.
`lwp-create`	Fires when a process thread is created, the latter typically as a result of `pthread_create()`. The `lwpsinfo_t` corresponding to the new thread is pointed to by `args[0]`. The `psinfo_t` of the process that created the thread is pointed to by `args[1]`.
`lwp-exit`	Fires when a process or process thread is exiting, due either to a signal or to an explicit call to `exit` or `pthread_exit()`.

Table 9-7 (Cont.) proc Probes

Probe	Description
`lwp-start`	Fires within the context of a newly created process or process thread. The `lwp-start` probe fires before any user-level instructions are executed. If the thread is the first created for the process, the `start` probe fires, 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 process or to a thread created by a process. The `signal-send` probe fires in the context of the sending process or 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 fires before any user-level instructions are executed in the process.

Note:

No fundamental difference between a process and a thread that a process creates, exists in Linux. The threads of a process are set up so that they can share resources, but each thread has its own entry in the process table with its own process ID.

proc Probe Arguments

The following table lists the argument types for the `proc` probes. See proc Probes for a description of the arguments. The `argN` are implementation specific. Use `args[]` to access the probe arguments.

Table 9-8 proc Probe Arguments

Probe	args[0]	args[1]	args[2]
create	psinfo_t *	—	—
exec	char *	—	—
exec-failure	int	—	—
exec-success	—	—	—
exit	int	—	—
lwp-create	lwpsinfo_t *	psinfo_t *	—
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`. Detailed information about this data structure can be found in `/usr/lib64/dtrace/version/procfs.d`. Some structure members, while still recognized for historical reasons, aren't implemented. The definition of the `lwpsinfo_t` structure is as follows:

```
typedef struct lwpsinfo {
        int pr_flag;                    /* lwp flags (DEPRECATED) */
        int pr_lwpid;                   /* lwp id */
        uintptr_t pr_addr;              /* internal address of lwp */
        uintptr_t pr_wchan;             /* NOT IMPLEMENTED */
        char pr_stype;                  /* NOT IMPLEMENTED */
        char pr_state;                  /* numeric lwp state */
        char pr_sname;                  /* printable char for pr_state */
        char pr_nice;                   /* NOT IMPLEMENTED */
        short pr_syscall;               /* NOT IMPLEMENTED */
        char pr_oldpri;                 /* NOT IMPLEMENTED */
        char pr_cpu;                    /* NOT IMPLEMENTED */
        int pr_pri;                     /* priority */
        ushort_t pr_pctcpu;             /* NOT IMPLEMENTED */
        ushort_t pr_pad;                /* struct padding */
        timestruc_t pr_start;           /* NOT IMPLEMENTED */
        timestruc_t pr_time;            /* NOT IMPLEMENTED */
```

```
        char pr_clname[8];              /* NOT IMPLEMENTED */
        char pr_name[16];               /* name */
        processorid_t pr_onpro;         /* processor last ran on */
        processorid_t pr_bindpro;       /* NOT IMPLEMENTED */
        psetid_t pr_bindpset;           /* NOT IMPLEMENTED */
        int pr_lgrp;                    /* NOT IMPLEMENTED */
        int pr_filler[4];               /* struct padding */

} lwpsinfo_t;
```

> **Note:**
>
> Lightweight processes don't exist in Linux. Rather, in Oracle Linux, processes and threads are represented by process descriptors of type `struct task_struct` in the task list. DTrace translates the members of `lwpsinfo_t` from the `task_struct` for the Oracle Linux process.

The `pr_flag` is set to `1` if the thread is stopped. Otherwise, it's set to `0`.

In Oracle Linux, the `pr_stype` field is unsupported, and hence is always `0`.

The following table describes the values that `pr_state` can take, including the corresponding character values for `pr_sname`.

Table 9-9 pr_state Values

pr_state Value	pr_sname Value	Description
`SRUN` (2)	`R`	The thread is runnable or is running on a CPU. The `sched:::enqueue` probe fires immediately before a thread's state is transitioned to `SRUN`. The `sched:::on-cpu` probe will fire a short time after the thread starts to run. The equivalent Oracle Linux task state is `TASK_RUNNING`.
`SSLEEP` (1)	`S`	The thread is sleeping. The `sched:::sleep` probe will fire immediately before a thread's state is transitioned to `SSLEEP`. The equivalent Oracle Linux task state is `TASK_INTERRUPTABLE` or `TASK_UNINTERRUPTABLE`.

Table 9-9 (Cont.) pr_state Values

pr_state Value	pr_sname Value	Description
SSTOP (4)	T	The thread is stopped, either because of an explicit proc directive or some other stopping mechanism. The equivalent Oracle Linux task state is __TASK_STOPPED or __TASK_TRACED.
SWAIT (7)	W	The thread is waiting on wait queue. The sched:::cpucaps-sleep probe will fire immediately before the thread's state transitions to SWAIT. The equivalent Oracle Linux task state is TASK_WAKEKILL or TASK_WAKING.
SZOMB (3)	Z	The thread is a zombie. The equivalent Oracle Linux task state is EXIT_ZOMBIE, EXIT_DEAD, or TASK_DEAD.

psinfo_t

Several proc probes have an argument of type psinfo_t. Detailed information about this data structure can be found in /usr/lib64/dtrace/*version*/procfs.d. The definition of the psinfo_t structure, is as follows:

```
typedef struct psinfo {
        int pr_flag;                    /* process flags (DEPRECATED) */
        int pr_nlwp;                    /* number of active lwps (Linux: 1) */
        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 */
        uid_t pr_gid;                   /* real group id */
        uid_t pr_egid;                  /* effective group id */
        uintptr_t pr_addr;              /* address of process */
        size_t pr_size;                 /* size of process image (in KB) */
        size_t pr_rssize;               /* resident set sie (in KB) */
        size_t pr_pad1;
        struct tty_struct *pr_ttydev;   /* controlling tty (or -1) */
        ushort_t pr_pctcpu;             /* % of recent cpu time used */
        ushort_t pr_pctmem;             /* % of recent memory used */
        timestruc_t pr_start;           /* process start time */
        timestruc_t pr_time;            /* usr+sys cpu time for process */
        timestruc_t pr_ctime;           /* usr+sys cpu time for children */
        char pr_fname[16];              /* name of exec'd file */
```

```
        char pr_psargs[80];             /* initial chars of arg list */
        int pr_wstat;                   /* if zombie, wait() status */
        int pr_argc;                    /* initial argument count */
        uintptr_t pr_argv;              /* address of initial arg vector */
        uintptr_t pr_envp;              /* address of initial env vector */
        char pr_dmodel;                 /* data model */
        char pr_pad2[3];
        taskid_t pr_taskid;             /* task id */
        dprojid_t pr_projid;            /* project id */
        int pr_nzomb;                   /* number of zombie lwps (Linux: 0) */
        poolid_t pr_poolid;             /* pool id */
        zoneid_t pr_zoneid;             /* zone id */
        id_t pr_contract;               /* process contract */
        int pr_filler[1];
        lwpsinfo_t pr_lwp;

} psinfo_t;
```

> **Note:**
>
> Lightweight processes don't exist in Linux. In Oracle Linux, processes and threads are represented by process descriptors of type `struct task_struct` in the task list. DTrace translates the members of `psinfo_t` from the `task_struct` for the Oracle Linux process.

`pr_dmodel` is set to either `PR_MODEL_ILP32`, denoting a 32–bit process, or `PR_MODEL_LP64`, denoting a 64–bit process.

proc Examples

The following examples illustrate the use of the probes that are published by the `proc` provider.

create

The following example shows how you can use the `create` probe to show the pids that are creating other pids. Both the creating and resulting pids are shown in the output. Add the following D source code and save it in a file named `create.d`:

```
proc:::create
{
  printf("%d created %d\n",
      args[0]->pr_ppid,
      args[0]->pr_pid);
}
```

Run the D script. The D script shows output similar to the following:

```
CPU     ID                    FUNCTION:NAME
  0 111864                        :create 2670 created 691647

  2 111864                        :create 1 created 691648
```

```
  2 111864                        :create 691074 created 691649

  3 111864                        :create 691649 created 691650

  3 111864                        :create 691649 created 691651
...
```

exec, exec-success and exec-failure

The following example shows how you can use the `exec`, `exec-success` and `exec-failure` probes to easily determine which programs are being run, and by which parent process. Type the following D source code and save it in a file named `whoexec.d`:

```
#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 results in output similar to the following:

```
WHO                  WHAT                 COUNT
bash                 date                 1
bash                 grep                 1
bash                 ssh                  1
bash                 wc                   1
bash                 ls                   2
bash                 sed                  2
...
```

start and exit

To determine how long programs are running, from creation to termination, you can enable the `start` and `exit` probes, as shown in the following example. Save it in a file named `progtime.d`:

```
proc:::start
{
  self->start = timestamp;
}

proc:::exit
/self->start/
{
  @[execname] = quantize(timestamp - self->start);
  self->start = 0;
}
```

Running the example script on a build server for several seconds results in output similar to the following:

```
...
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
...
```

signal-send

The following example shows how you can use the `signal-send` probe to determine the sending and receiving of process associated with any signal. Type the following D source code and save it in a file named `sig.d`:

```
#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:

```
              SENDER            RECIPIENT          SIG COUNT
       kworker/u16:7               dtrace            2 1
       kworker/u16:7                 sudo            2 1
           swapper/2             sssd_kcm           34 1
           swapper/6              pmlogger           14 1
```

proc Stability

The `proc` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

Profile Provider

The `profile` provider includes probes that are associated with an interrupt that fires at some regular, specified time interval.

Such probes aren't associated with any particular point of execution, but rather with the asynchronous interrupt event. You can use these probes to sample some aspect of the system state and then use the samples to infer system behavior. If the sampling rate is high or the sampling time is long, an accurate inference is possible. Using DTrace functions, you can use

the `profile` provider to sample many aspects of the system. For example, you could sample the state of the current thread, the state of the CPU, or the current machine instruction.

profile-*n* Probes

The `profile-n` probes fire at a fixed interval, at a high-interrupt level on all active CPUs.

The units of *n* default to a frequency that's expressed as a rate of firing per second, but the value can also have an optional suffix , as shown in Table 9-10, which specifies either a time interval or a frequency. The following table describes valid time suffixes for a `tick-` *n* probe.

Table 9-10 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 expressed as rate per second)

tick-*n* Probes

The `tick-n` probes fire at fixed intervals, at a high interrupt level on only one CPU per interval.

Unlike `profile-n` probes, which fire on every CPU, `tick-n` probes fire on only one CPU per interval and the CPU on which they fire can change over time. The units of *n* default to a frequency expressed as a rate of firing per second, but the value can also have an optional time suffix as shown in Table 9-10, which specifies either a time interval or a frequency.

The `tick-n` probes have several uses, such as providing some periodic output or taking a periodic action.

> **Note:**
>
> The highest available tick frequency is 5000 Hz (`tick-5000`).

profile Probe Arguments

The following table describes the arguments for the `profile` probes.

Table 9-11 `profile` Probe Arguments

Probe	arg0	arg1
`profile-n`	`pc`	`upc`
`tick-n`	`pc`	`upc`

The arguments are as follows:

- `pc`: kernel program counter
- `upc`: user-space program counter

profile Probe Creation

Unlike other providers, the `profile` provider creates probes dynamically on an as-needed basis. Thus, the preferred probe might not appear in a listing of all probes, for example, when using the `dtrace -l -P profile` command, but the probe is created when it's explicitly enabled.

A time interval that's too short causes the machine to continuously field time-based interrupts and denies service on the machine. The `profile` provider refuses to create a probe that would result in an interval of less than two hundred microseconds and returns an error.

prof Stability

The `profile` provider uses DTrace's stability mechanism to describe its stabilities. These stability values are listed in the following table.

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

Rawtp Provider

The `rawtp` provider gives DTrace users access to the raw tracepoints exposed by the kernel tracing system, including access to the untranslated arguments of the associated tracepoint events.

To see what raw tracepoints are available on a system, use:

```
sudo dtrace -lP rawtp
```

To see the types of the untranslated arguments, use:

```
sudo dtrace -lvP rawtp
```

rawtp Stability

The `rawtp` provider uses DTrace's stability mechanism to describe its stabilities. These stability values are listed in the following table.

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

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.

sched Probes

The probes for the `sched` provider are listed in the following table. For all `sched` probes, the module is `vmlinux` and the function is an empty string.

Table 9-12 sched Probes

Probe	Description
`dequeue`	Fires immediately before a runnable thread is dequeued from a run queue. The run queue's associated CPU is described by `args[2]`. If there is no associated CPU, the `cpu_id` member of the `args[2]` structure is `-1`."
`enqueue`	Fires immediately before a runnable thread is enqueued to a run queue. If the run queue isn't associated with a particular CPU, the `cpu_id` member of the `args[2]` structure is `-1`. The value in `args[3]` is a Boolean, non zero if the thread is enqueued at the front of the run queue, and zero if at the back.
`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's ending execution, while `args[0]` and `args[1]` refer to the next thread to run on the CPU.
`on-cpu`	Fires when a CPU has begun execution of a thread. The current CPU, thread, and process, are described by `curcpu`, `curlwpsinfo`, and `curpsinfo`, respectively.

Table 9-12 (Cont.) sched Probes

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.
wakeup	Fires immediately before the current thread wakes a thread sleeping on a synchronization object. Here, `args[0]` and `args[1]` refer to the sleeping thread, as an `lwpsinfo_t *` and `psinfo_t *`, respectively. The type and address of the synchronization object are contained in the `pr_stype` and `pr_wchan` members of the `lwpsinfo_t` of the sleeping thread. The meaning of this address is a private implementation detail, but the address value might be treated as a token unique to the synchronization object.

sched Probe Arguments

Many of these probes refer to a particular thread. For these probes, the thread's `lwpsinfo_t` is pointed to by `args[0]` and the `psinfo_t` of the process containing the thread by `args[1]`. A few probes refer to a particular CPU. Its `cpuinfo_t` is pointed to by `args[2]`. Only `enqueue` has an `args[3]`, and that argument is a Boolean, as described. The `argN` values are implementation specific. Instead, use `args[]` to access the probe arguments.

The following table contains a summary of the `sched` provider probe arguments.

Table 9-13 sched Probe Arguments

Probe	args[0]	args[1]	args[2]	args[3]
dequeue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	—
enqueue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	int
off-cpu	lwpsinfo_t *	psinfo_t *	—	—
on-cpu	—	—	—	—
surrender	lwpsinfo_t *	psinfo_t *	—	—
tick	lwpsinfo_t *	psinfo_t *	—	—
wakeup	lwpsinfo_t *	psinfo_t *	—	—

lwpsinfo_t and psinfo_t

The `lwpsinfo_t` and `psinfo_t` structures are described in Proc Provider.

cpuinfo_t

The `cpuinfo_t` structure defines a CPU. The `args[2]` arguments for the `enqueue` and `dequeue` probes point to the `cpuinfo_t` for the CPU associated with the run queue, which is sometimes different from the current CPU, whose `cpuinfo_t` is pointed to by the `curcpu` variable.

The definition of the `cpuinfo_t` structure is:

```
typedef struct cpuinfo {
  processorid_t cpu_id;
  psetid_t cpu_pset;         /* not supported */
  chipid_t cpu_chip;
  lgrp_id_t cpu_lgrp;        /* not supported */
} cpuinfo_t;
```

sched Examples

The following examples illustrate the use of the probes that are published by the `sched` provider.

on-cpu and off-cpu

One common question that you might want answered is which CPUs are running threads and for how long? The following example shows how you can use the `on-cpu` and `off-cpu` probes to easily answer this question on a system-wide basis. Type the following D source code and save it in a file named `where.d`:

```
sched:::on-cpu
{
  self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
  @[cpu] = quantize(timestamp - self->ts);
  self->ts = 0;
}
```

Run the script. After a few seconds, cancel the script using `Ctrl-C`. The output looks similar to:

```
        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 previous output shows that on CPU 1 threads tend to run for less than 131072 nanoseconds (on order of 100 microseconds) at a stretch, or for 8388608 to 134217728 nanoseconds (about 10 to 100 milliseconds). A noticeable 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 also use the `on-cpu` and `off-cpu` probes for answering this question. The following script displays which CPUs run a specified application over a period of ten seconds. Save it in a file named `whererun.d`.:

```
#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-10sec
{
  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 previous script on a large mail server and specifying the IMAP daemon (using `sudo dtrace -qs whererun.d imapd`) results in output that's similar to the following:

```
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 Linux considers the amount of time that a thread has been sleeping when selecting a CPU on which to run the thread, as a thread that has been sleeping for less time tends not to migrate. Use the `off-cpu` and `on-cpu` probes to observe this behavior. Type the following source code and save it in a file named `howlong.d`:

```
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;
}
```

Run the script. After around 30 seconds, cancel the script using `Ctrl+C`. The output looks similar to:

```
 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 previous output reveals more occurrences of non-migration than migration. Also, when sleep times are longer, migrations are more likely. The distributions are different in the under 100 millisecond range, but look similar as the sleep times get longer. This result would seem to indicate that sleep time isn't factored into the scheduling decision when a certain threshold is exceeded.

enqueue and dequeue

You might want to know on which CPUs processes and threads are waiting to run. You can use the `enqueue` probe along with the `dequeue` probe to answer this question. Type the following source code and save it in a file named `qtime.d`:

```
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 previous script for several seconds results in output that's similar to the following:

```
1
   value  ------------- Distribution ------------- count
    8192 |                                         0
   16384 |                                         1
   32768 |@                                        47
   65536 |@@@@@@@                                  365
  131072 |@@@@@@@@@@@@                             572
  262144 |@@@@@@@@@@@@                             570
  524288 |@@@@@@@                                  354
 1048576 |@                                        57
 2097152 |                                         7
 4194304 |                                         1
 8388608 |                                         1
16777216 |                                         0

0
   value  ------------- Distribution ------------- count
    8192 |                                         0
   16384 |                                         6
   32768 |@                                        49
   65536 |@@@@@                                    261
  131072 |@@@@@@@@@@@@@@@                          753
  262144 |@@@@@@@@@@@@@                            704
  524288 |@@@@@@@@@                                455
 1048576 |@                                        74
 2097152 |                                         9
```

```
         4194304 |                                         2
         8388608 |                                         0
```

Rather than 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. Type the following source code and save it in a file named `qlen.d`:

```
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 previous script on a largely idle dual-core processor system for about 30 seconds results in output that's similar to the following:

```
        1
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  8124
               1 |@@@@@@                                   1558
               2 |@                                        160
               3 |                                         51
               4 |                                         24
               5 |                                         13
               6 |                                         11
               7 |                                         9
               8 |                                         6
               9 |                                         0

        0
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  8569
               1 |@@@@@@@@@@                               2429
               2 |@                                        292
               3 |                                         25
               4 |                                         8
               5 |                                         5
               6 |                                         4
               7 |                                         4
               8 |                                         1
               9 |                                         0
```

The output is what you might expect for an idle system: most the time that a runnable thread is enqueued, the run queues were short (three or fewer threads in length). However, as that the system was largely idle, the exceptional data points at the bottom of each table might be unexpected. For example, why were the run queues as long as 8 runnable threads? To explore

this question further, 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 probes can't iterate over data structures, and therefore can't iterate over the entire run queue. Even if D probes could do so, avoid dependencies on the kernel's internal data structures.

For this type of script, you would enable the `enqueue` and `dequeue` probes and then use both speculations and associative arrays. For example, type the following source code and save it in a file named `whoqueue.d`:

```
#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. We just set a new
   * record, so we'll discard the old one.
   */
  discard(spec[this->cpu]);
}

sched:::enqueue
/this->len > maxlen/
{
  /*
   * We 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]/
{
  /*
   * We 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;
}
```

In this script, whenever a thread is enqueued, it increments the length of the queue and records the timestamp in an associative array keyed by the thread. You can't 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, and if so, 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. When the kernel dequeues the last thread that was enqueued at the timestamp of the longest length, the script commits the speculation data.

Running the previous script on the same system results in output that's similar to the following:

```
Run queue of length 1:
 2850/2850 (java)
Run queue of length 2:
 4034/4034 (kworker/0:1)
 16/16 (sync_supers)
Run queue of length 3:
 10/10 (ksoftirqd/1)
 1710/1710 (hald-addon-inpu)
 25350/25350 (dtrace)
Run queue of length 4:
 2852/2852 (java)
 2850/2850 (java)
 1710/1710 (hald-addon-inpu)
 2099/2099 (Xorg)
Run queue of length 5:
 3149/3149 (notification-da)
 2417/2417 (gnome-settings-)
 2437/2437 (gnome-panel)
 2461/2461 (wnck-applet)
 2432/2432 (metacity)
Run queue of length 9:
 3685/3685 (firefox)
```

```
 3149/3149 (notification-da)
 2417/2417 (gnome-settings-)
 2437/2437 (gnome-panel)
 2852/2852 (java)
 2452/2452 (nautilus)
 2461/2461 (wnck-applet)
 2432/2432 (metacity)
 2749/2749 (gnome-terminal)
^C
```

wakeup

The following example shows how you might use the `wakeup` probe to find what's waking a particular process, and when, over a time period. Type the following source code and save it in a file named `gterm.d`:

```
#pragma D option quiet

dtrace:::BEGIN
{
  start = timestamp;
}

sched:::wakeup
/stringof(args[1]->pr_fname) == "gnome-terminal"/
{
  @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);
}

profile:::tick-10sec
{
  exit(0);
}
```

Running this script results in output similar to:

```
  Xorg
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@                         69
               1 |@@@@@@@@                                 35
               2 |@@@@@@@@@@                               42
               3 |                                         2
               4 |                                         0
               5 |                                         0
               6 |                                         0
               7 |@@@@                                     16
               8 |                                         0
               9 |@@@                                      15
           >= 10 |                                         0
```

This output shows that the X server is waking the `gnome-terminal` process as you interact with the system.

tick

Oracle Linux might use *tick-based CPU accounting*, where a system clock interrupt fires at a fixed interval and attributes CPU utilization to the processes that are running at the time of the tick. The following example shows how you would use the `tick` probe to observe this attribution.

```
sudo dtrace -n sched:::tick'{ @[stringof(args[1]->pr_fname)] = count() }'
```

Enter `Ctrl+C`, and output similar to the following is shown:

```
  VBoxService                                                   1
  gpk-update-icon                                               1
  hald-addon-inpu                                               1
  jbd2/dm-0-8                                                   1
  automount                                                     2
  gnome-session                                                 2
  hald                                                          2
  gnome-power-man                                               3
  ksoftirqd/0                                                   3
  kworker/0:2                                                   3
  notification-da                                               4
  devkit-power-da                                               6
  nautilus                                                      9
  dbus-daemon                                                  11
  gnome-panel                                                  11
  gnome-settings-                                              11
  dtrace                                                       19
  khugepaged                                                   22
  metacity                                                     27
  kworker/0:0                                                  41
  swapper                                                      56
  firefox                                                      58
  wnck-applet                                                  61
  gnome-terminal                                               67
  java                                                         84
  Xorg                                                        227
```

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. If a thread is to perform some amount of work every clock tick (say, every 10 milliseconds), the system either over accounts or under accounts for the thread, depending on whether the accounting is done before or after time-related dispatching scheduling activity. If accounting is performed before time-related dispatching, the system under accounts for threads running at a regular interval. If such threads run for less than the clock tick interval, they can effectively hide behind the clock tick.

The following example examines whether a system has any such threads. Type the following source code and save it in a file named `tick.d`:

```
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`:

```
  tick
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@                                    29
               1 |@@@@@@@@@@@@@@@@@@@@                     106
               2 |@@@@@                                    27
               3 |@                                        7
               4 |@@                                       10
               5 |@@                                       12
               6 |@                                        4
               7 |@                                        8
               8 |@@                                       9
               9 |@@@                                      17
           >= 10 |                                         0

  enqueue
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@                                     82
               1 |@@@@                                     86
               2 |@@@@                                     76
               3 |@@@                                      65
               4 |@@@@@                                    101
               5 |@@@@                                     79
               6 |@@@@                                     75
               7 |@@@@                                     76
               8 |@@@@                                     89
               9 |@@@@                                     75
           >= 10 |                                         0
```

The output histogram named `tick` shows that the clock tick is firing at a 1 millisecond offset. In this example, the output for `enqueue` is evenly spread across the ten millisecond interval and no spike is visible at 1 millisecond, so it seems the threads aren't being scheduled on a time basis.

sched Stability

The `sched` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

SDT Provider

The Statically Defined Tracing (SDT) provider (`sdt`) creates probes at sites that a software programmer has formally designated. Thus, the SDT provider is chiefly of interest only to developers of new providers. Most users access SDT only indirectly by using other providers.

The SDT mechanism enables programmers to consciously choose locations of interest to users of DTrace and to convey some semantic knowledge about each location through the probe name.

Importantly, SDT can act as a metaprovider by registering probes so that they appear to come from other providers, such as `io`, `lockstat`, `proc`, and `sched`.

Both the name stability and the data stability of the probes are Private, which reflects the kernel's implementation and should not be interpreted as a commitment to preserve these interfaces.

Creating sdt Probes

If you are a device driver developer, you might be interested in creating `sdt` probes for an Oracle Linux driver that you are working on. The disabled probe effect of SDT is only the cost of several no-operation machine instructions. You are therefore encouraged to add `sdt` probes to device driver code as needed. Unless these probes negatively affect performance, you can leave them in shipped code.

DTrace also provides a mechanism for application developers to define user-space static probes.

Declaring Probes

The `sdt` probes are declared by using the `DTRACE_PROBE` macro from `<linux/sdt.h>`.

The module name and function name of an SDT-based probe correspond to the kernel module name and function name where the probe is declared. DTrace includes the kernel module name and function name as part of the tuple used to identify the probe in the probe description, so you don't need to explicitly include this information when devising the probe name. You can still specify the module and function name when referring to the probe in a DTrace program to prevent namespace collisions. Use the `dtrace -l -m` *`mymodule`* command to list the probes that *mymodule* has installed and the full names that are seen by DTrace users.

The name of the probe depends on the name that's provided in the `DTRACE_PROBE` macro. If the name doesn't contain two consecutive underscores (__), the name of the probe is as written in the macro. If the name contains two consecutive underscores, the probe name converts the consecutive underscores to a single dash (-). For example, if a `DTRACE_PROBE` macro specifies `transaction__start`, the SDT probe is named `transaction-start`. This substitution enables C code to provide macro names that aren't valid C identifiers without specifying a string.

SDT can also act as a metaprovider by registering probes so that they appear to come from other providers, such as `io`, `proc`, and `sched`, which don't have dedicated modules of their own. For example, `kernel/exit.c` contains calls to the `DTRACE_PROC` macro, which are defined as follows in `<linux/sdt.h>`:

```
# define DTRACE_PROC(name) \
         DTRACE_PROBE(__proc_##name);
```

Probes that use such macros appear to come from a provider other than `sdt`. The leading double underscore, provider name, and trailing underscore in the `name` argument are used to match the provider and aren't included in the probe name.

sdt Probe Arguments

The arguments for each `sdt` probe are the arguments that are specified in the kernel source code in the corresponding `DTRACE_PROBE` macro reference. When declaring `sdt` probes, you can minimize their disabled probe effect by not dereferencing pointers and by not loading from global variables in the probe arguments. Both pointer dereferencing and global variable loading can be done safely in D functions that enable probes, so DTrace users can request these functions only when they're needed.

sdt Stability

The `sdt` provider uses DTrace's stability mechanism to describe its stabilities. These values are listed in the following table.

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

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 OS kernel, the `syscall` provider can offer tremendous insight into application behavior about the system.

syscall 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 been transferred back to user-level. For all `syscall` probes, the function name is set as the name of the instrumented system call.

Often, the system call names that are provided by `syscall` correspond to names in the Section 2 manual pages. However, some `syscall` provider probes don't directly correspond to any documented system call, such as the case where a system call might be a sub operation of another system call or where a system call might be private in that they span the user-kernel boundary.

syscall Probe Arguments

For `entry` probes, the arguments, `arg0` ... `argn` , are 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 a system call failure.

syscall Stability

The `syscall` provider uses DTrace's stability mechanism to describe its stabilities. These stability values are listed in the following table.

Element	Name Stability	Data Stability	Dependency Class
Provider	Evolving	Evolving	Common
Module	Private	Private	Unknown
Function	Private	Private	Instruction set architecture (ISA)
Name	Evolving	Evolving	Common
Arguments	Private	Private	ISA

USDT Provider

Use the USDT provider, for user space statically defined tracing, to instrument user space code with probes that are meaningful for an application.

For example, if an application has `put` and `get` operations, you can insert `put` and `get` instrumentation points in the source code, even if each operation is implemented on several code paths. A DTrace user could then enable such probes to trace activity, even without knowing how those operations are implemented in the source code. As usual, there are negligible performance impacts for DTrace probes when the probes aren't enabled. USDT probes can also appear in shared libraries.

USDT is for user space processes. For kernel modules, statically defined tracing is handled by the related SDT mechanism.

Defining USDT Providers and Probes

Define USDT providers and probes in a `.d` file that you add to the source code. For example, a file `myproviders.d` contains:

```
provider myprov
{
  probe my__put(int, int);
  probe my__get();
};
```

In this example, the provider name is `myprov`, but, as with the `pid` provider, DTrace users must append the process ID (pid) of the process or processes that interest them. In contrast to the `pid` provider, the USDT provider descriptions can use wildcards for the pids. For example, specifying `myprov1234` traces this provider's probes only for process ID `1234`. In contrast, `myprov*` traces this provider's probes for all processes that have been appropriately

instrumented, even processes that haven't yet started. Or, as with the `pid` provider, you can use a symbolic pid, such as `myprov$target` for the process started with the `-c` option.

The provider definition lists its probes, along with any probe arguments. The D compiler converts two consecutive underscores (__) to a dash (-) in the probe name.

This following example runs command `./a.out`, then traces the USDT probe `my-put` on that one process, displaying the two arguments to that probe.

```
sudo dtrace -c ./a.out -n 'myprov$target:::my-put { printf("put %d %d\n",
arg0, arg1); }'
```

The following example traces all processes with `myprov` probes, even if they haven't yet started. This example uses the `-Z` option of `dtrace` in case zero processes match at the time `dtrace` is started.

```
sudo dtrace -Z -n 'myprov*:::my-put { printf("put %d %d\n", arg0, arg1); }'
```

Adding USDT Probes to Application Code

Consider this C code `func.c`:

```
#include "myproviders.h"

void foo(void)
{
    ...
    if (MYPROV_MY_PUT_ENABLED()) {
        int arg0, arg1;

        arg0 = bar(1111);
        arg1 = bar(2222);
        MYPROV_MY_PUT(arg0, arg1);
    }
    ...
    MYPROV_MY_GET();
    ...
}
```

This example includes a header file that's automatically generated. The name of this header file is derived from the file name that defines the macros to access the probes. It defines macros that provide access to the USDT probes.

You can place probes in the code, referring to the probes by using the macros, which concatenate provider, and probe names, and converting to uppercase. In this example, the macros are `MYPROV_MY_PUT()` and `MYPROV_MY_GET()`.

An optional optimization is to test if a probe is enabled. While the computational overhead of a disabled DTrace probe is often similar to a few no-op instructions, setting up probe arguments can be expensive. In this example, `bar(1111)` and `bar(2222)` might be costly function calls. Therefore, for each probe, DTrace also supplies an `is-enabled` macro, named by appending `_ENABLED`. In the example, `MYPROV_MY_PUT_ENABLED()` for the `my-put` probe, to help minimize the cost of any work associated with disabled probes.

Building Applications With USDT Probes

The `dtrace` command becomes part of the build procedure, which can be thought of in four parts:

1. Generate the header file that defines the macros to access the probes. For example:

   ```
   dtrace -h -s myproviders.d
   ```

   The previous command produces the `myproviders.h` header file. While `dtrace` requires root privileges for runtime tracing, generating the header file doesn't have this requirement.

2. Compile the source code, which includes the `dtrace` generated header file based on the provider and probe definitions. For example, for several source files:

   ```
   gcc -c func1.c
   gcc -c func2.c
   gcc -c func3.c
   ```

3. Post process each object file using `dtrace`. For example:

   ```
   dtrace -G -s myproviders.d func1.o func2.o func3.o
   ```

   Again, `dtrace` doesn't require root privileges for this step. This step also generates the object file `myproviders.o` from `myproviders.d` and the other object files, linking provider and probe definitions with a user application.

4. Link the final executable. The `-Wl,--export-dynamic` link options to `gcc` are required for symbol lookup in a stripped executable at runtime, for example, when you use the D function `ustack()`. For example:

   ```
   gcc -Wl,--export-dynamic,--strip-all myproviders.o func1.o func2.o func3.o
   ```

USDT Examples

1. Create a file `myproviders.d` that contains:

   ```
   provider myprov
   {
       probe my__put(int, int);
       probe my__get();
   };
   ```

2. Create a C program `func.c` that contains:

   ```
   #include <stdio.h>
   #include <unistd.h>
   #include "myproviders.h"

   int bar(int in)
   {
       printf("bar evaluates %d\n", in);
   ```

```
    return 3 * in;
}

void foo(void)
{
    if (MYPROV_MY_PUT_ENABLED()) {
        int arg0, arg1;
        arg0 = bar(1111);
        arg1 = bar(2222);
        MYPROV_MY_PUT(arg0, arg1);
    }
    MYPROV_MY_GET();
}

int main(int c, char **v)
{
    while (1) {
        usleep(1000 * 1000);
        foo();
    }

    return 0;
}
```

3. Build the application using:

```
dtrace -h -s myproviders.d
gcc -c func.c
dtrace -G -s myproviders.d func.o
gcc -Wl,--export-dynamic,--strip-all myproviders.o func.o
```

4. You could run this program in several ways. For example:
   - You could run the tracing program using:

     ```
     sudo dtrace -c ./a.out -q -n '
         myprov$target:::my-put { printf("put %d %d\n", arg0, arg1); }
         myprov$target:::my-get { printf("get\n"); }
         tick-5sec {exit(0)}'
     ```

     This first example, runs the `a.out` command with the `-c` option. The `-q` quiet option suppresses extraneous output. The D script is on the command line, and it prints both args for the `put` probe and reports the `get` probe. The example refers symbolically to `$target`, the pid of the target command is specified with `-c`.

     After five seconds, the `dtrace` job finishes, ending the target command.
   - Or, you could run this example using:

     ```
     ./a.out &
     pid=$!
     sudo dtrace -q -n '
              myprov'$pid':::my-put { printf("put %d %d\n", arg0, arg1); }
              myprov'$pid':::my-get { printf("get\n"); }
              tick-5sec {exit(0)}'
     kill $pid
     ```

In this example, the command is already running with process ID `$pid`. We refer to the specific numerical pid that interests us. The `dtrace` command doesn't, in this case, end the process of interest. We handle that separately.

- One more possibility is to run the program using:

```
sudo dtrace -Z -q -n '
    myprov*:::my-put { printf("put %d %d\n", arg0, arg1); }
    myprov*:::my-get { printf("get\n"); }
    tick-10sec {exit(0)}' &
./a.out &
```

In this example, the `-Z` option allows for zero probe matches at first. The probes match later, when a USDT process has started. After a short delay, a USDT process is started. At some point, the USDT process is ended.

In each of these cases, the output is printed after one second, and looks similar to:

```
...
bar evaluates 1111
bar evaluates 2222
put 3333 6666
get
...
```

The D script can omit the `put` probe using:

```
sudo dtrace -c ./a.out -q -n '
    myprov$target:::my-get { printf("get\n"); }
    tick-5sec {exit(0)}'
```

In this case, not only does the `put` probe not fire, but also the `is-enabled` conditional `MYPROV_MY_PUT_ENABLED()` is false. Therefore, the `bar()` function isn't called. The only output displayed each second is:

```
get
```