Oracle Linux
DTrace Tutorial

E50705-16
January 2025

Oracle Linux DTrace Tutorial,

E50705-16

Contents

3 Tracing User-Space Applications

4 Going Further With DTrace

Preface

> **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 or Oracle Linux 9 as soon as possible.
>
> For more information about DTrace, see *Oracle Linux: DTrace Release Notes* and Oracle Linux: Using DTrace for System Tracing.

Oracle Linux: DTrace Tutorial provides examples of how you can use the features of the Dynamic Tracing (DTrace) tool to examine the behavior of the operating system and user-space programs.

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

Introducing DTrace

> **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 or Oracle Linux 9 as soon as possible.
>
> For more information about DTrace, see *Oracle Linux: DTrace Release Notes* and Oracle Linux: Using DTrace for System Tracing.

This chapter introduces the dynamic tracing (DTrace) facility of Oracle Linux. You can use DTrace to examine the behavior of the operating system and of user-space programs that have been instrumented with DTrace probes. The Unbreakable Enterprise Kernel (UEK) build is enabled to run DTrace. UEK is the kernel that is compiled for use on Oracle Linux. You could run a Linux kernel other than UEK on Oracle Linux, but DTrace would most likely not be enabled and available, as UEK is the kernel that is included with Oracle Linux.

Note that more recent versions of UEK often have better DTrace functionality than earlier versions. The largest set of DTrace features and improvements are available on the latest DTrace-enabled kernels.

This tutorial assumes that you are using UEK on the x86_64 architecture. Note that UEK releases for any other architectures might not include support for all of the providers that are discussed in this tutorial.

About This Tutorial

This tutorial includes a variety of DTrace scripts and describes different ways in which you can use DTrace. Several examples have additional exercises that offer further practice in using DTrace. Each exercise provides an estimate of the time that you should allow to complete it. Depending on your level of programming knowledge, you might need more or less time. You should already have a good understanding of Linux administration and system programming, and broad experience using a programming language, such as C or C++, and a scripting language, such as Python. If you are not familiar with terms such as *system call*, *type*, *cast*, *signal*, *struct*, or *pointer*, you might have difficulty in understanding some of the examples or completing some of the exercises in this tutorial. However, each exercise provides a sample solution in case you do get stuck. You are encouraged to experiment with the examples to develop your skills at creating DTrace scripts.

> **Caution:**
>
> To run the examples and perform the exercises in this tutorial, you need to have `root` access to a system. Only the `root` user or a user with `sudo` access to run commands as `root` can use the `dtrace` utility. As `root`, you have total power over a system and so have total responsibility for that system. Note that although DTrace is designed so that you can use it safely without needing to worry about corrupting the operating system or other processes, there are ways to circumvent the built-in safety measures.
>
> To minimize risk, perform the examples and exercises in this tutorial on a system other than a production system.

The examples in this tutorial demonstrate the different ways that you can perform dynamic tracing of your system: by entering a simple D program as an argument to `dtrace` on the command line, by using the `dtrace` command to run a script that contains a D program, or by using an executable D script that contains a *hashbang* (`#!` or *shebang*) invocation of `dtrace`. When you create your own D programs, you can choose which method best suits your needs.

About DTrace

DTrace is a comprehensive dynamic tracing facility that was first developed for use on the Solaris operating system (now known as Oracle Solaris) and subsequently ported to Oracle Linux. You can use DTrace to explore the operation of your system to better understand how it works, to track down performance problems across many layers of software, or to locate the causes of aberrant behavior.

Using DTrace, you can record data at previously instrumented places of interest, which are referred to as *probes*, in kernel and user-space programs. A probe is a location to which DTrace can bind a request to perform a set of actions, such as recording a stack trace, a timestamp, or the argument to a function. Probes function like programmable sensors that record information. When a probe is triggered, DTrace gathers data that you have designated in a D script and reports this data back to you.

Using DTrace's D programming language, you can query the system probes to provide immediate and concise answers to any number of questions that you might formulate.

A D program describes the actions that occur if one or more specified probes is triggered. A probe is uniquely specified by the name of the DTrace provider that publishes the probe, the name of the module, library, or user-space program in which the probe is located, the name of the function in which the probe is located, and the name of the probe itself, which usually describes some operation or functionality that you can trace. Because you do not need to specify probes exactly, this allows DTrace to perform the same action for a number of different probes. Full and explicit representation of a single probe in the D language takes the form:

PROVIDER:*MODULE*:*FUNCTION*:*NAME*

When you use the `dtrace` command to run a D program, you invoke the compiler for the D language. When DTrace has compiled your D program into a safe, intermediate form, it sends it to the DTrace module in the operating system kernel for execution. The DTrace module activates the probes that your program specifies and executes the associated actions when your probes fire. DTrace handles any runtime errors that might occur during your D program's execution, including dividing by zero, dereferencing invalid memory, and so on, and reports them to you.

Note:

Modules Primer

There are two kinds of modules that are frequently referenced in most, if not all, detailed discussions of DTrace on Oracle Linux. To avoid confusion, you must identify which kind of module is being discussed with any mention of a *module*. The context usually gives plenty of clues, if you have knowledge of these kinds of modules.

The module that is being discussed in the sequence *PROVIDER*:*MODULE*:*FUNCTION*:*NAME* refers to a module in the sense that it is a distinct, orderly component that is used by DTrace to reference and represent areas of code. You can specify that a DTrace module reference point DTrace to some set of code or functionality. Output from a `dtrace` command uses *MODULE* to convey that some activity has occurred in such an area of code in the kernel or in a user-space program. This type of module can simply be referred to as a *DTrace module*.

A second and very different meaning for the term *module* is a Linux kernel module. The Linux kernel is divided into different functional components that are called modules: these modules might be loaded and unloaded separately from each other. The output of the `lsmod` command shows which Linux kernel modules are loaded on the system. These modules are referred to as *Linux kernel modules*, or within the context of discussing only Linux, simply *kernel modules*.

The following are two additional variations of other module references:

- Some Linux kernel modules that are specific to DTrace must be present to use DTrace on a Linux system. These particular kernel modules are specifically referenced as *dtrace kernel modules*. See the table in About DTrace Providers for a list of providers that are available from specific `dtrace` kernel modules.
- DTrace probes must be compiled into any kernel module in order for DTrace to monitor the activity in the kernel module. However, kernel modules with DTrace probes are not `dtrace` kernel modules, rather, they are referred to as *DTrace enabled kernel modules*. All kernel modules that can be traced by DTrace implicitly are DTrace enabled kernel modules and therefore are not typically referred to explicitly as DTrace enabled kernel modules, but with the shorthand, *kernel modules*.

Unless you explicitly permit DTrace to perform potentially destructive actions, you cannot construct an unsafe program that would cause DTrace to inadvertently damage either the operating system or any process that is running on your system. These safety features enable you to use DTrace in a production environment without worrying about crashing or corrupting your system. If you make a programming mistake, DTrace reports the error and deactivates your program's probes. You can then correct your program and try again.

For more information about using DTrace, see the Oracle Linux: DTrace Reference Guide.

About DTrace Providers

The following table lists the providers that are included with the Oracle Linux implementation of DTrace and the kernel modules that include the providers.

Provider	dtrace Kernel Module	Description
dtrace	dtrace	Provides probes that relate to DTrace itself, such as BEGIN, ERROR, and END. You can use these probes to initialize DTrace's state before tracing begins, process its state after tracing has completed, and handle unexpected execution errors in other probes.
fbt	fbt	Supports function boundary tracing (FBT) probes, which are at the entry and exits of kernel functions.
fasttrap	fasttrap	Supports user-space tracing of DTrace-enabled applications.
io	sdt	Provides probes that relate to data input and output. The io provider enables quick exploration of behavior observed through I/O monitoring.
IP	sdt	Provides probes for the IP protocol (both IPv4 and IPv6).
lockstat	sdt	Provides probes for locking events including: mutexes, read-write locks, and spinlock.
perf	sdt	Provides probes that correspond to each perf tracepoint, including typed arguments.
proc	sdt	Provides probes for monitoring process creation and termination, LWP creation and termination, execution of new programs, and signal handling.
profile	profile	Provides probes that are associated with an asynchronous interrupt event that fires at a fixed and specified time interval, rather than with any particular point of execution. You can use these probes to sample some aspect of a system's state.
sched	sdt	Provides probes related to CPU scheduling. Because CPUs are the one resource that all threads must consume, the sched provider is very useful for understanding systemic behavior.

Provider	`dtrace` Kernel Module	Description
`syscall`	`systrace`	Provides probes at the entry to and return from every system call. Because system calls are the primary interface between user-level applications and the operating system kernel, these probes can offer you an insight into the interaction between applications and the system.
`TCP`	`sdt`	Provides probes in the code that implements the TCP protocol, for both IPv4 and IPv6.
`UDP`	`sdt`	Provides probes in the code that implements the UDP protocol, for both IPv4 and IPv6.

SDT is a multi-provider, in that it implements multiple providers under the same provider.

The `fasttrap` provider is considered a meta-provider, which means it is a provider framework. The `fasttrap` meta-provider is used to facilitate the creation of providers that are instantiated for user-space processes.

See DTrace Providers in the Oracle Linux: DTrace Reference Guide for more information about providers and their probes.

Preparing to Install and Configure DTrace

> **Note:**
>
> The DTrace package (`dtrace-utils`) is available from ULN. To ensure best results, your system must be registered with ULN and should be installed with or updated to the latest Oracle Linux release.

To install and configure DTrace, perform the following steps:

1. If your system is not already running the latest UEK version:
   a. Update your system to the latest UEK release:

      ```
      # yum update
      ```

   b. Reboot the system and select the latest UEK version that is available in the boot menu. Typically, this is the default kernel.
2. Install the DTrace utilities package:

   ```
   # yum install dtrace-utils
   ```

Using Automatically Loaded DTrace Modules

> **Note:**
>
> The following is a quick-start method for using DTrace kernel modules that are automatically loaded. If you plan to manually load DTrace kernel modules, see Manually Loading DTrace Modules for instructions.

DTrace automatically loads some `dtrace` kernel modules when the `dtrace` command references the probes that are associated with a `dtrace` kernel module. You can use this convenient method to load `dtrace` modules, rather than manually loading them.

To find out which modules are automatically loaded in this manner, use the following command:

```
# cat /etc/dtrace-modules
sdt
systrace
profile
fasttrap
```

Additional modules can be added to this list after it is determined that they are fully tested.

To determine whether a particular module has been loaded in the Linux kernel, use the `lsmod` command. For example, you would run the following command to determine whether the `sdt` module is loaded:

```
# lsmod | grep sdt
```

If the module is not loaded, the command most likely will result in no output. If the module is loaded, the output is similar to the following:

```
sdt                    20480   0
dtrace                 151552  4 sdt,fasttrap,systrace,profile
```

Manually Loading DTrace Modules

> **Note:**
>
> The following information describes how to manually load DTrace kernel modules. If you plan to use DTrace kernel modules that are automatically loaded, you can skip this section of the tutorial. See Using Automatically Loaded DTrace Modules.

To use DTrace providers, their supported kernel modules must be loaded each time the kernel is booted.

If the `dtrace` kernel module is not already loaded, when the `dtrace` command is run, the `dtrace` module and all of the modules that are listed in `/etc/dtrace-modules` are all loaded automatically. However, if the `dtrace` kernel module is already loaded, the automatic kernel module loading mechanism is not triggered.

You can load modules manually by using the `modprobe` command. For example, to use the `fbt` kernel module if it is not in the default list, you would run the following command:

```
# modprobe fbt
```

The `modprobe` action also loads the `dtrace` kernel module as a dependency so that a subsequent `dtrace` command does not drive automatic loading of other `dtrace` modules until the `dtrace` kernel module is no longer loaded. The `drace` kernel module will no longer be loaded upon another boot of the system or after the manual removal of the `dtrace` kernel modules.

The suggested practice is to use the `dtrace -l` command to trigger automatic module loading and thereby also confirm basic `dtrace` functionality. Then, use the `modprobe` command to load additional modules that are not in the default list, such as `fbt`, as needed.

Example: Displaying Probes for a Provider

The following example shows how you would display the probes for a provider, such as `proc`, by using the `dtrace` command.

```
# dtrace -l -P proc
   ID   PROVIDER            MODULE                          FUNCTION NAME
  855       proc           vmlinux                          _do_fork lwp-create
  856       proc           vmlinux                          _do_fork create
  883       proc           vmlinux                           do_exit lwp-exit
  884       proc           vmlinux                           do_exit exit
  931       proc           vmlinux                 do_sigtimedwait signal-clear
  932       proc           vmlinux                   __send_signal signal-send
  933       proc           vmlinux                   __send_signal signal-discard
  941       proc           vmlinux                   send_sigqueue signal-send
  944       proc           vmlinux                      get_signal signal-handle
 1044       proc           vmlinux                   schedule_tail start
 1045       proc           vmlinux                   schedule_tail lwp-start
 1866       proc           vmlinux             do_execveat_common exec-failure
 1868       proc           vmlinux             do_execveat_common exec
 1870       proc           vmlinux             do_execveat_common exec-success
```

The output shows the numeric identifier of the probe, the name of the probe provider, the name of the probe module, the name of the function that contains the probe, and the name of the probe itself.

The full name of a probe is *`PROVIDER`*`:`*`MODULE`*`:`*`FUNCTION`*`:`*`NAME`*, for example, `proc:vmlinux:_do_fork:create`. If no ambiguity exists with other probes for the same provider, you can usually omit the *`MODULE`* and *`FUNCTION`* elements when specifying a probe. For example, you can refer to `proc:vmlinux:_do_fork:create` as `proc::_do_fork:create` or `proc:::create`. If several probes match your specified probe in a D program, the associated actions are performed for each probe.

These probes enable you to monitor how the system creates processes, executes programs, and handles signals.

If you checked previously and the `sdt` module was not loaded, check again to see if the `dtrace` command has loaded the module.

If the following message is displayed after running the `dtrace -l -P proc` command (instead of output similar to the output in the previous example), it is an indication that the module has not loaded:

```
No probe matches description
```

If the `sdt` module does not load automatically on a system with DTrace properly installed, it is because another DTrace module was manually loaded by using the `modprobe` command. Manually loading a DTrace module in this way effectively prevents any other modules from being automatically loaded by the `dtrace` command until the system is rebooted. In this instance, one workaround is to use the `modprod` command to manually load the `sdt` module. When the module has successfully loaded, you should see a probe listing similar to the output in Example: Displaying Probes for a Provider when you re-issue the `dtrace` command.

Exercise: Enabling and Listing DTrace Probes

Try listing the probes of the `syscall` provider. Notice that both `entry` and `return` probes are provided for each system call.

(Estimated completion time: 3 minutes)

Solution to Exercise: Enabling and Listing DTrace Probes

```
# dtrace -l -P syscall
  ID   PROVIDER            MODULE                  FUNCTION NAME
   4    syscall          vmlinux                      read entry
   5    syscall          vmlinux                      read return
   6    syscall          vmlinux                     write entry
   7    syscall          vmlinux                     write return
   8    syscall          vmlinux                      open entry
   9    syscall          vmlinux                      open return
  10    syscall          vmlinux                     close entry
  11    syscall          vmlinux                     close return
...
 646    syscall          vmlinux            pkey_mprotect entry
 647    syscall          vmlinux            pkey_mprotect return
 648    syscall          vmlinux               pkey_alloc entry
 649    syscall          vmlinux               pkey_alloc return
 650    syscall          vmlinux                pkey_free entry
 651    syscall          vmlinux                pkey_free return
 652    syscall          vmlinux                    statx entry
 653    syscall          vmlinux                    statx return
 654    syscall          vmlinux                  waitfd entry
 655    syscall          vmlinux                  waitfd return
```

> **Note:**
>
> The probe IDs numbers might differ from those on your system, depending on what other providers are loaded.

Running a Simple DTrace Program

The following example shows how you would use a text editor to create a new file called `hello.d` and then type a simple D program.

Example: Simple D Program That Uses the BEGIN Probe (hello.d)

```
/* hello.d -- A simple D program that uses the BEGIN probe */

BEGIN
```

```
{
  /* This is a C-style comment */
  trace("hello, world");
  exit(0);
}
```

A D program consists of a series of clauses, where each clause describes one or more probes to enable, and an optional set of actions to perform when the probe fires. The actions are listed as a series of statements enclosed in braces `{}` following the probe name. Each statement ends with a semicolon (`;`).

In this example, the function `trace` directs DTrace to record the specified argument, the string ”hello, world”, when the `BEGIN` probe fires, and then print it out. The function `exit()` tells DTrace to cease tracing and exit the `dtrace` command.

The full name of the `BEGIN` probe is `dtrace:::BEGIN`. `dtrace` provides three probes: `dtrace:::BEGIN`, `dtrace:::END`, and `dtrace:::ERROR`. Because these probe names are unique to the `dtrace` provider, their names can be shortened to `BEGIN`, `END`, and `ERROR`.

When you have saved your program, you can run it by using the `dtrace` command with the `-s` option, which specifies the name of the file that contains the D program:

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

DTrace interprets and runs the script. You will notice that in addition to the string `"hello,world"`, the default behavior of DTrace is to display information about the CPU on which the script was running when a probe fired, the ID of the probe, the name of the function that contains the probe, and the name of the probe itself. The function name is displayed as blank for `BEGIN`, as DTrace provides this probe.

You can suppress the probe information in a number of different ways, for example, by specifying the `-q` option:

```
# dtrace -q -s hello.d
hello, world
```

Exercise: Using the END Probe

Copy the `hello.d` program to the file `goodbye.d`. Edit this file so that it traces the string "goodbye, world" and uses the `END` probe instead of `BEGIN`. When you run this new script, you need to type `Ctrl-C` to cause the probe to fire and exit `dtrace`.

(Estimated completion time: 5 minutes)

Solution to Exercise and Example: Using the END Probe

The following is an example of a simple D program that demonstrates the use of the `END` probe:

```
/* goodbye.d -- Simple D program that demonstrates the END probe */

END
{
  trace("goodbye, world");
}
```

```
# dtrace -s goodbye.d
dtrace: script 'goodbye.d' matched 1 probe
^C
CPU     ID                    FUNCTION:NAME
  3      2                             :END   goodbye, world

# dtrace -q -s ./goodbye.d
                    ^C
goodbye, world
```

2
Tracing Operating System Behavior

> **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 or Oracle Linux 9 as soon as possible.
>
> For more information about DTrace, see *Oracle Linux: DTrace Release Notes* and Oracle Linux: Using DTrace for System Tracing.

This chapter provides examples of D programs that you can use to investigate what is happening in the operating system.

Tracing Process Creation

The `proc` probes enable you to trace process creation and termination, execution of new program images, and signal processing on a system. See proc Provider in the Oracle Linux: DTrace Reference Guide for a description of the `proc` probes and their arguments.

Example: Monitoring the System as Programs Are Executed (execcalls.d)

The following example shows the D program, `execcalls.d`, which uses `proc` probes to monitor the system as it executes process images:

```
/* execcalls.d -- Monitor the system as it executes programs */

proc::do_execveat_common:exec
{
  trace(stringof(args[0]));
}
```

The `args[0]` argument to the `exec` probe is set to the path name of the program that is being executed. You use the `stringof()` function to convert the type from `char *` to the D type `string`.

> **Note:**
>
> The `sdt` kernel module, which enables the `proc` provider probes, is most likely already loaded on the test system. Or, if not already loaded, the `sdt` kernel module will automatically load if you did not manually load a DTrace module since booting the system. See Manually Loading DTrace Modules for details. In the following example, the `sdt` kernel module needs to be manually loaded or it must be able to automatically load for proper functionality.

Type the `dtrace -s execcalls.d` command to run the D program in one window. Then start different programs from another window, while observing the output from `dtrace` in the first window. To stop tracing after a few seconds have elapsed, type `Ctrl-C` in the window that is running `dtrace`.

```
# dtrace -s execcalls.d
dtrace: script 'execcalls.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1   1185          do_execveat_common:exec /usr/sbin/sshd
  0   1185          do_execveat_common:exec /usr/sbin/unix_chkpwd
  0   1185          do_execveat_common:exec /bin/bash
  0   1185          do_execveat_common:exec /usr/bin/id
  0   1185          do_execveat_common:exec /usr/bin/hostname
  0   1185          do_execveat_common:exec /usr/bin/id
  0   1185          do_execveat_common:exec /usr/bin/id
  0   1185          do_execveat_common:exec /usr/bin/grep
  0   1185          do_execveat_common:exec /usr/bin/tty
  0   1185          do_execveat_common:exec /usr/bin/tput
  0   1185          do_execveat_common:exec /usr/bin/grep
  1   1185          do_execveat_common:exec /usr/sbin/unix_chkpwd
  1   1185          do_execveat_common:exec /usr/libexec/grepconf.sh
  1   1185          do_execveat_common:exec /usr/bin/dircolors
  0   1185          do_execveat_common:exec /usr/bin/ls
^C
```

The activity here shows a login to the same system (from another terminal) while the script is running.

The probe `proc::do_execveat_common:exec` fires whenever the system executes a new program and the associated action uses `trace()` to display the path name of the program.

Exercise: Suppressing Verbose Output From DTrace

Run the `execcalls.d` program again, but this time add the `-q` option to suppress all output except output from `trace()`. Notice how DTrace displays only what you traced with `trace()`.

(Estimated completion time: less than 5 minutes)

Solution to Exercise: Suppressing Verbose Output From DTrace

```
# dtrace -q -s execcalls.d
/usr/bin/id/usr/bin/tput/usr/bin/dircolors/usr/bin/id/
usr/lib64/qt-3.3/bin/gnome-terminal/usr/local/bin/gnome-terminal
/usr/bin/gnome-terminal/bin/bash/usr/bin/id/bin/grep/bin/basename
/usr/bin/tty/bin/ps
```

Tracing System Calls

System calls are the interface between user programs and the kernel, which perform operations on the programs' behalf.

The next example shows the next D program, `syscalls.d`, which uses `syscall` probes to record `open()` system call activity on a system.

Example: Recording open() System Calls on a System (syscalls.d)

```
/* syscalls.d -- Record open() system calls on a system */

syscall::open:entry
{
  printf("%-16s %-16s\n",execname,copyinstr(arg0));
}
```

In this example, the `printf()` function is used to display the name of the executable that is calling `open()` and the path name of the file that it is attempting to open.

> **Note:**
>
> Use the `copyinstr()` function to convert the first argument (`arg0`) in the `open()` call to a string. Whenever a probe accesses a pointer to data in the address space of a user process, you must use one of the `copyin()`, `copyinstr()`, or `copyinto()` functions to copy the data from user space to a DTrace buffer in kernel space. In this example, it is appropriate to use `copyinstr()`, as the pointer refers to a character array. If the string is not null-terminated, you also need to specify the length of the string to `copyinstr()`, for example, `copyinstr(arg1, arg2)`, for a system call such as `write()`. See User Process Tracing in the Oracle Linux: DTrace Reference Guide.
>
> The `sdt` kernel module, which enables the `proc` provider probes, is most likely already loaded on the test system. Or, if not already loaded, the `sdt` kernel module will automatically load if you did not manually load a DTrace module since booting the system. See Manually Loading DTrace Modules for details.

In the following example, the `sdt` kernel module needs to be manually loaded or it must be able to automatically load for proper functionality:

```
# dtrace -q -s syscalls.d
udisks-daemon    /dev/sr0
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/present
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/energy_now
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/
voltage_max_design
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/
voltage_min_design
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/status
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/current_now
devkit-power-da  /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/
voltage_now
VBoxService      /var/run/utmp
firefox          /home/guest/.mozilla/firefox/qeaojiol.default/sessionstore.js
firefox          /home/guest/.mozilla/firefox/qeaojiol.default/sessionstore-1.js
```

```
firefox          /home/guest/.mozilla/firefox/qeaojiol.default/sessionstore-1.js
^C
```

Exercise: Using the printf() Function to Format Output

Amend the arguments to the `printf()` function so that `dtrace` also prints the process ID and user ID for the process. Use a conversion specifier such as `%-4d`.

See Output Formatting in the Oracle Linux: DTrace Reference Guide for a description of the `printf()` function.

The process ID and user ID are available as the variables `pid` and `uid`. Use the `BEGIN` probe to create a header for the output.

(Estimated completion time: 10 minutes)

Solution to Exercise: Using the printf() Function to Format Output

```
/* syscalls1.d -- Modified version of syscalls.d that displays more information */

BEGIN
{
  printf("%-6s %-4s %-16s %-16s\n","PID","UID","EXECNAME","FILENAME");
}

syscall::open:entry
{
  printf("%-6d %-4d %-16s %-16s\n",pid,uid, execname,copyinstr(arg0));
}
```

Note how this solution uses similar formatting strings to output the header and the data.

```
# dtrace -q -s syscalls1.d
PID    UID  EXECNAME         FILENAME
3220   0    udisks-daemon    /dev/sr0
2571   0    sendmail         /proc/loadavg
3220   0    udisks-daemon    /dev/sr0
2231   4    usb              /dev/usblp0
2231   4    usb              /dev/usb/lp0
2231   4    usb              /dev/usb/usblp0
...
^C
```

Performing an Action at Specified Intervals

The `profile` provider includes `tick` probes that you can use to sample some aspect of a system's state at regular intervals. Note that the `profile` kernel module must be loaded to use these probes.

Example: Using `tick.d`

The following is an example of the `tick.d` program.

```
/* tick.d -- Perform an action at regular intervals */

BEGIN
```

```
{
  i = 0;
}

profile:::tick-1sec
{
  printf("i = %d\n",++i);
}

END
{
  trace(i);
}
```

In this example, the program declares and initializes the variable `i` when the D program starts, increments the variable and prints its value once every second, and displays the final value of `i` when the program exits.

When you run this program, it produces output that is similar to the following, until you type `Ctrl-C`:

```
# dtrace -s tick.d
dtrace: script 'tick.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  1   5315                      :tick-1sec i = 1

  1   5315                      :tick-1sec i = 2

  1   5315                      :tick-1sec i = 3

  1   5315                      :tick-1sec i = 4

  1   5315                      :tick-1sec i = 5

  1   5315                      :tick-1sec i = 6

^C
  1   5315                      :tick-1sec i = 7

  0      2                           :END         7
```

To suppress all of the output except the output from `printf()` and `trace()`, specify the `-q` option:

```
# dtrace -q -s tick.d
i = 1
i = 2
i = 3
i = 4
^C
i = 5
5
```

Exercise: Using tick Probes

List the available `profile` provider probes. Experiment with using a different `tick` probe. Replace the `trace()` call in `END` with a `printf()` call.

See profile Provider in the Oracle Linux: DTrace Reference Guide for a description of the probes.

(Estimated completion time: 10 minutes)

Solution to Exercise and Example: Using tick Probes

```
# dtrace -l -P profile
  ID   PROVIDER           MODULE              FUNCTION NAME
   5    profile                                      profile-97
   6    profile                                      profile-199
   7    profile                                      profile-499
   8    profile                                      profile-997
   9    profile                                      profile-1999
  10    profile                                      profile-4001
  11    profile                                      profile-4999
  12    profile                                      tick-1
  13    profile                                      tick-10
  14    profile                                      tick-100
  15    profile                                      tick-500
  16    profile                                      tick-1000
  17    profile                                      tick-5000
5315    profile                                      tick-1sec
5316    profile                                      tick-10sec
```

Example: Modified Version of tick.d

```
/* tick1.d -- Modified version of tick.d */

BEGIN
{
  i = 0;
}

/* tick-500ms fires every 500 milliseconds */
profile:::tick-500ms
{
  printf("i = %d\n",++i);
}

END
{
  printf("\nFinal value of i = %d\n",i);
}
```

This example uses the `tick-500ms` probe, which fires twice per second.

```
# dtrace -s tick1.d
dtrace: script 'tick1.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  2    642                      :tick-500ms i = 1

  2    642                      :tick-500ms i = 2

  2    642                      :tick-500ms i = 3

  2    642                      :tick-500ms i = 4

^C
  2    642                      :tick-500ms i = 5

  3      2                             :END
Final value of i = 5
```

Using Predicates to Select Actions

Predicates are logic statements that choose whether DTrace invokes the actions that are associated with a probe. You can use predicates to focus tracing analysis on specific contexts under which a probe fires.

Example: Using daterun.d

The following example shows an executable DTrace script, `daterun.d`, which displays the file descriptor, output string, and string length specified to the `write()` system call whenever the `date` command is run on the system.

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

/* daterun.d -- Display arguments to write() when date runs */

syscall::write:entry
/execname == "date"/
{
  printf("%s(%d, %s, %d)\n", probefunc, arg0, copyinstr(arg1), arg2);
}
```

In the example, the predicate is `/execname == "date"/`, which specifies that if the probe `syscall::write:entry` is triggered, DTrace runs the associated action only if the name of the executable is `date`.

Make the script executable by changing its mode:

```
# chmod +x daterun.d
```

If you run the script from one window, while typing the `date` command in another window, output similar to the following is displayed in the first window:

```
# ./daterun.d
write(1, Thu Oct 31 11:14:43 GMT 2013
, 29)
```

Example: Listing Available syscall Provider Probes

The following example shows how you would list available `syscall` provider probes.

```
# dtrace -l -P syscall | less
   ID   PROVIDER            MODULE                  FUNCTION NAME
   18    syscall          vmlinux                      read entry
   19    syscall          vmlinux                      read return
   20    syscall          vmlinux                     write entry
   21    syscall          vmlinux                     write return
   22    syscall          vmlinux                      open entry
   23    syscall          vmlinux                      open return
   24    syscall          vmlinux                     close entry
   25    syscall          vmlinux                     close return
   26    syscall          vmlinux                   newstat entry
   27    syscall          vmlinux                   newstat return
...
  648    syscall          vmlinux                pkey_alloc entry
  649    syscall          vmlinux                pkey_alloc return
  650    syscall          vmlinux                 pkey_free entry
  651    syscall          vmlinux                 pkey_free return
```

```
  652    syscall           vmlinux                  statx entry
  653    syscall           vmlinux                  statx return
  654    syscall           vmlinux                 waitfd entry
  655    syscall           vmlinux                 waitfd return
```

Exercise: Using syscall Probes

Experiment by adapting the `daterun.d` script for another program. Make the new script produce output when the system is running `w`.

(Estimated completion time: 10 minutes)

Solution to Exercise: Using syscall Probes

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

/* wrun.d -- Modified version of daterun.d for the w command */

syscall::write:entry
/execname == "w"/
{
  printf("%s(%d, %s, %d)\n", probefunc, arg0, copyinstr(arg1, arg2), arg2);
}
```

The program uses the two-argument form of `copyinstr()`, as the string argument to `write()` might not be null-terminated:

```
# chmod +x wrun.d
# ./wrun.d
write(1,  12:14:55 up  3:21,  3 users,  load average: 0.14, 0.15, 0.18
, 62)
write(1, USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
, 69)
write(1, guest    tty1     :0               08:55    3:20m 11:23   0.17s pam: gdm-passwo
, 80)
write(1, guest    pts/0    :0.0             08:57    7.00s  0.17s  0.03s w
m: gdm-passwo
, 66)
write(1, guest    pts/1    :0.0             12:14    7.00s  0.69s  8.65s gnome-terminal

, 79)
...
^C
```

Timing Events on a System

Determining the time that a system takes to perform different activities is a fundamental technique for analyzing its operation and determining where bottlenecks might be occurring.

Example: Monitoring read() System Call Duration (readtrace.d)

The following is an example of the D program, `readtrace.d`.

```
/* readtrace.d -- Display time spent in read() calls */

syscall::read:entry
{
  self->t = timestamp; /* Initialize a thread-local variable */
```

```
}

syscall::read:return
/self->t != 0/
{
  printf("%s (pid=%d) spent %d microseconds in read()\n",
  execname, pid, ((timestamp - self->t)/1000)); /* Divide by 1000 for microseconds */

  self->t = 0; /* Reset the variable */
}
```

In the example, the `readtrace.d` program displays the command name, process ID, and call duration in microseconds whenever a process invokes the `read()` system call. The variable `self->t` is *thread-local*, meaning that it exists only within the scope of execution of a thread on the system. The program records the value of `timestamp` in `self->t` when the process calls `read()`, and subtracts this value from the value of `timestamp` when the call returns. The units of `timestamp` are nanoseconds, so you divide by 1000 to obtain a value in microseconds.

The following is output from running this program:

```
# dtrace -q -s readtrace.d
NetworkManager (pid=878) spent 10 microseconds in read()
NetworkManager (pid=878) spent 9 microseconds in read()
NetworkManager (pid=878) spent 2 microseconds in read()
in:imjournal (pid=815) spent 63 microseconds in read()
gdbus (pid=878) spent 7 microseconds in read()
gdbus (pid=878) spent 66 microseconds in read()
gdbus (pid=878) spent 63 microseconds in read()
irqbalance (pid=816) spent 56 microseconds in read()
irqbalance (pid=816) spent 113 microseconds in read()
irqbalance (pid=816) spent 104 microseconds in read()
irqbalance (pid=816) spent 91 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 63 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
irqbalance (pid=816) spent 61 microseconds in read()
sshd (pid=10230) spent 8 microseconds in read()
in:imjournal (pid=815) spent 6 microseconds in read()
sshd (pid=10230) spent 7 microseconds in read()
in:imjournal (pid=815) spent 5 microseconds in read()
sshd (pid=10230) spent 7 microseconds in read()
in:imjournal (pid=815) spent 6 microseconds in read()
sshd (pid=10230) spent 7 microseconds in read()
in:imjournal (pid=815) spent 5 microseconds in read()
^C
```

Exercise: Timing System Calls

Add a predicate to the `entry` probe in `readtrace.d` so that `dtrace` displays results for a disk space usage report that is selected by the name of its executable (`df`).

(Estimated completion time: 10 minutes)

Solution to Exercise: Timing System Calls

The following example shows a modified version of the `readtrace.d` program that includes a predicate.

```
/* readtrace1.d -- Modified version of readtrace.d that includes a predicate */

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

syscall::read:return
/self->t != 0/
{
  printf("%s (pid=%d) spent %d microseconds in read()\n",
  execname, pid, ((timestamp - self->t)/1000));

  self->t = 0; /* Reset the variable */
}
```

The predicate `/execname == "df"/` tests whether the `df` program is running when the probe fires.

```
# dtrace -q -s readtrace1.d
df (pid=1666) spent 6 microseconds in read()
df (pid=1666) spent 8 microseconds in read()
df (pid=1666) spent 1 microseconds in read()
df (pid=1666) spent 50 microseconds in read()
df (pid=1666) spent 38 microseconds in read()
df (pid=1666) spent 10 microseconds in read()
df (pid=1666) spent 1 microseconds in read()
^C
```

Exercise: Timing All System Calls for cp (calltrace.d)

Using the `probefunc` variable and the `syscall:::entry` and `syscall:::return` probes, create a D program, `calltrace.d`, which times all system calls for the executable `cp`.

(Estimated completion time: 10 minutes)

Solution to Exercise: Timing All System Calls for cp (calltrace.d)

```
/* calltrace.d -- Time all system calls for cp */

syscall:::entry
/execname == "cp"/
{
  self->t = timestamp; /* Initialize a thread-local variable */
}

syscall:::return
/self->t != 0/
{
  printf("%s (pid=%d) spent %d microseconds in %s()\n",
  execname, pid, ((timestamp - self->t)/1000), probefunc);
```

```
  self->t = 0; /* Reset the variable */
}
```

Dropping the function name `read` from the probe specifications matches all instances of `entry` and `return` probes for `syscall`. The following is a check for system calls resulting from running the `cp` executable:

```
# dtrace -q -s calltrace.d
cp (pid=2801) spent 4 microseconds in brk()
cp (pid=2801) spent 5 microseconds in mmap()
cp (pid=2801) spent 15 microseconds in access()
cp (pid=2801) spent 7 microseconds in open()
cp (pid=2801) spent 2 microseconds in newfstat()
cp (pid=2801) spent 3 microseconds in mmap()
cp (pid=2801) spent 1 microseconds in close()
cp (pid=2801) spent 8 microseconds in open()
cp (pid=2801) spent 3 microseconds in read()
cp (pid=2801) spent 1 microseconds in newfstat()
cp (pid=2801) spent 4 microseconds in mmap()
cp (pid=2801) spent 12 microseconds in mprotect()
   ...
cp (pid=2801) spent 183 microseconds in open()
cp (pid=2801) spent 1 microseconds in newfstat()
cp (pid=2801) spent 1 microseconds in fadvise64()
cp (pid=2801) spent 17251 microseconds in read()
cp (pid=2801) spent 80 microseconds in write()
cp (pid=2801) spent 58 microseconds in read()
cp (pid=2801) spent 57 microseconds in close()
cp (pid=2801) spent 85 microseconds in close()
cp (pid=2801) spent 57 microseconds in lseek()
cp (pid=2801) spent 56 microseconds in close()
cp (pid=2801) spent 56 microseconds in close()
cp (pid=2801) spent 56 microseconds in close()
^C
```

Tracing Parent and Child Processes

When a process forks, it creates a child process that is effectively a copy of its parent process, but with a different process ID. For information about other differences, see the `fork(2)` manual page. The child process can either run independently from its parent process to perform some separate task. Or, a child process can execute a new program image that replaces the child's program image while retaining the same process ID.

Example: Using proc Probes to Report Activity on a System (activity.d)

The D program `activity.d` in the following example uses `proc` probes to report `fork()` and `exec()` activity on a system.

```
#pragma D option quiet

/* activity.d -- Record fork() and exec() activity */

proc::_do_fork:create
{
  /* Extract PID of child process from the psinfo_t pointed to by args[0] */
  childpid = args[0]->pr_pid;

  time[childpid] = timestamp;
  p_pid[childpid] = pid; /* Current process ID (parent PID of new child) */
```

```
  p_name[childpid] = execname; /* Parent command name */
  p_exec[childpid] = ""; /* Child has not yet been exec'ed */
}

proc::do_execveat_common:exec
/p_pid[pid] != 0/
{
  p_exec[pid] = args[0]; /* Child process path name */
}

proc::do_exit:exit
/p_pid[pid] != 0 &&  p_exec[pid] != ""/
{
  printf("%s (%d) executed %s (%d) for %d microseconds\n",
    p_name[pid], p_pid[pid], p_exec[pid], pid, (timestamp - time[pid])/1000);
}

proc::do_exit:exit
/p_pid[pid] != 0 &&  p_exec[pid] == ""/
{
  printf("%s (%d) forked itself (as %d) for %d microseconds\n",
    p_name[pid], p_pid[pid], pid, (timestamp - time[pid])/1000);
}
```

In the example, the statement `#pragma D option quiet` has the same effect as specifying the `-q` option on the command line.

The process ID of the child process (`childpid`), following a `fork()`, is determined by examining the `pr_pid` member of the `psinfo_t` data structure that is pointed to by the `args[0]` probe argument. For more information about the arguments to `proc` probes, see proc Provider in the Oracle Linux: DTrace Reference Guide.

The program uses the value of the child process ID to initialize globally unique associative array entries, such as `p_pid[childpid]`.

> **Note:**
>
> An *associative array* is similar to a normal array, in that it associates keys with values, but the keys can be of any type; they need not be integers.

When you run the program, you should see output similar to the following as you use the `ssh` command to access the same system from another terminal window. You might want to try running different programs from this new terminal window to generate additional output:

```
# dtrace -s activity.d
sshd (3966) forked itself (as 3967) for 3667020 microseconds
bash (3971) forked itself (as 3972) for 1718 microseconds
bash (3973) executed /usr/bin/hostname (3974) for 1169 microseconds
grepconf.sh (3975) forked itself (as 3976) for 1333 microseconds
bash (3977) forked itself (as 3978) for 967 microseconds
bash (3977) executed /usr/bin/tput (3979) for 1355 microseconds
bash (3980) executed /usr/bin/dircolors (3981) for 1212 microseconds
sshd (3966) executed /usr/sbin/unix_chkpwd (3968) for 31444 microseconds
sshd (3966) executed /usr/sbin/unix_chkpwd (3969) for 1653 microseconds
bash (3970) forked itself (as 3971) for 2411 microseconds
bash (3970) forked itself (as 3973) for 1830 microseconds
bash (3970) executed /usr/libexec/grepconf.sh (3975) for 3696 microseconds
bash (3970) forked itself (as 3977) for 3273 microseconds
```

```
bash (3970) forked itself (as 3980) for 1928 microseconds
bash (3970) executed /usr/bin/grep (3982) for 1570 microseconds
^C
```

Exercise: Using a Predicate to Control the Execution of an Action

Modify `activity.d` so that `dtrace` displays results for parent processes that are selected by their executable name, for example, `bash`, or by a program name that you specify as an argument to the `dtrace` command.

(Estimated completion time: 10 minutes)

Solution to Exercise: Using a Predicate to Control the Execution of an Action

The only change that is required to specify the name of an executable is to add a predicate to the `proc::_do_fork:create` probe, for example:

```
/execname == "bash"/
```

A more generic version of the program sets the predicate check value from a passed-in command-line argument instead, for example:

```
/execname == $1/
```

Example: Recording fork() and exec() Activity for a Specified Program (activity1.d)

The following example uses a predicate that is passed in from the command line.

```
#pragma D option quiet

/* activity1.d -- Record fork() and exec() activity for a specified program */

proc::_do_fork:create
/execname == $1/
{
  /* Extract PID of child process from the psinfo_t pointed to by args[0] */
  childpid = args[0]->pr_pid;

  time[childpid] = timestamp;
  p_pid[childpid] = pid; /* Current process ID (parent PID of new child) */
  p_name[childpid] = execname; /* Parent command name */
  p_exec[childpid] = ""; /* Child has not yet been exec'ed */
}

proc::do_execveat_common:exec
/p_pid[pid] != 0/
{
  p_exec[pid] = args[0]; /* Child process path name */
}

proc::do_exit:exit
/p_pid[pid] != 0 &&  p_exec[pid] != ""/
{
  printf("%s (%d) executed %s (%d) for %d microseconds\n",
```

```
    p_name[pid], p_pid[pid], p_exec[pid], pid, (timestamp - time[pid])/1000);
}

proc::do_exit:exit
/p_pid[pid] != 0 &&  p_exec[pid] == ""/
{
  printf("%s (%d) forked itself (as %d) for %d microseconds\n",
    p_name[pid], p_pid[pid], pid, (timestamp - time[pid])/1000);
}
```

As shown in the following example, you can now specify the name of the program to be traced as an argument to the `dtrace` command. Note that you need to escape the argument to protect the double quotes from the shell:

```
# dtrace -s activity.d '"bash"'
bash (10367) executed /bin/ps (10368) for 10926 microseconds
bash (10360) executed /usr/bin/tty (10361) for 3046 microseconds
bash (10359) forked itself (as 10363) for 32005 microseconds
bash (10366) executed /bin/basename (10369) for 1285 microseconds
bash (10359) forked itself (as 10370) for 12373 microseconds
bash (10360) executed /usr/bin/tput (10362) for 34409 microseconds
bash (10363) executed /usr/bin/dircolors (10364) for 29527 microseconds
bash (10359) executed /bin/grep (10365) for 21024 microseconds
bash (10366) forked itself (as 10367) for 11749 microseconds
bash (10359) forked itself (as 10360) for 41918 microseconds
bash (10359) forked itself (as 10366) for 14197 microseconds
bash (10370) executed /usr/bin/id (10371) for 11729 microseconds
^C
```

Simple Data Aggregations

DTrace provides several functions for aggregating the data that individual probes gather. These functions include `avg()`, `count()`, `max()`, `min()`, `stddev()`, and `sum()`, which return the mean, number, maximum value, minimum value, standard deviation, and summation of the data being gathered, respectively. See Aggregations in the Oracle Linux: DTrace Reference Guide for descriptions of aggregation functions.

DTrace indexes the results of an aggregation by using a tuple expression that similar to what is used for an associative array:

```
@name[list_of_keys] = aggregating_function(args);
```

The name of the aggregation is prefixed with an `@` character. The keys describe the data that the aggregating function is collecting. If you do not specify a name for the aggregation, DTrace uses `@` as an anonymous aggregation name, which is usually sufficient for simple D programs.

Example: Counting the Number of write() System Calls Invoked by Processes

In the following example, the command counts the number of `write()` system calls that are invoked by processes, until you type `Ctrl-C`.

```
# dtrace -n 'syscall::write:entry { @["write() calls"] = count(); }'
dtrace: description 'syscall:::' matched 1 probe
^C

  write() calls                                                 9
```

> **Note:**
>
> Rather than create a separate D script for this simple example, the probe and the action is specified on the `dtrace` command line.
>
> DTrace prints the result of the aggregation automatically. Alternatively, you can use the `printa()` function to format the result of the aggregation.

Example: Counting the Number of read() and write() System Calls

The following example counts the number of both `read()` and `write()` system calls.

```
# dtrace -n 'syscall::write:entry,syscall::read:entry { @[strjoin(probefunc,"()
calls")] = count(); }'
dtrace: description 'syscall::write:entry,syscall::read:entry' matched 2 probes
^C

  write() calls                                                   150
  read() calls                                                   1555
```

Exercise: Counting System Calls Over a Fixed Period

Write a D program named `countcalls.d` that uses a `tick` probe and `exit()` to stop collecting data after 100 seconds and display the number of `open()`, `read()` and `write()` calls.

(Estimated completion time: 15 minutes)

Solution to Exercise and Example: Counting Write, Read, and Open System Calls Over 100 Seconds (countcalls.d)

```
/* countcalls.d -- Count write, read, and open system calls over 100 seconds */

profile:::tick-100sec
{
  exit(0);
}

syscall::write:entry, syscall::read:entry, syscall::open:entry
{
  @[strjoin(probefunc,"() calls")] = count();
}
```

The action that is associated with the `tick-100s` probe means that `dtrace` exits after 100 seconds and prints the results of the aggregation.

```
# dtrace -s countcalls.d
dtrace: script 'countcalls.d' matched 4 probes
CPU     ID                    FUNCTION:NAME
  3    643                     :tick-100sec

  write() calls                                                   1062
  open() calls                                                    1672
  read() calls                                                   29672
```

Example: Counting System Calls Invoked by a Process (countsyscalls.d)

The D program `countsyscalls.d` shown in the following example counts the number of times a process that is specified by its process ID invokes different system calls.

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

/* countsyscalls.d -- Count system calls invoked by a process */

syscall:::entry
/pid == $1/
{
  @num[probefunc] = count();
}
```

After making the `syscalls.d` file executable, you can run it from the command line, specifying a process ID as its argument.

The following example shows how you would monitor the use of the `emacs` program that was previously invoked. After the script is invoked, within `emacs` a couple files are opened, modified, and then saved before exiting the D script.

Make the script executable:

```
# chmod +x countsyscalls.d
```

From another command line, type:

```
# emacs foobar.txt
```

Now, start the script and use the opened `emacs` window:

```
# ./countsyscalls.d $(pgrep -u root emacs)
                   ^C

  chmod                                                             1
  exit_group                                                        1
  futex                                                             1
  getpgrp                                                           1
  lseek                                                             1
  lsetxattr                                                         1
  rename                                                            1
  fsync                                                             2
  lgetxattr                                                         2
  alarm                                                             3
  rt_sigaction                                                      3
  unlink                                                            3
  mmap                                                              4
  munmap                                                            4
  symlink                                                           4
  fcntl                                                             6
  newfstat                                                          6
  getgid                                                            7
  getuid                                                            7
  geteuid                                                           8
  openat                                                            8
  access                                                            9
  getegid                                                          14
  open                                                             14
  getdents                                                         15
```

```
  close                                                    17
  readlink                                                 19
  newlstat                                                 33
  newstat                                                 155
  read                                                    216
  timer_settime                                           231
  write                                                   314
  pselect6                                                376
  rt_sigreturn                                            393
  ioctl                                                   995
  rt_sigprocmask                                         1261
  clock_gettime                                          3495
```

In the preceding example, the `pgrep` command is used to determine the process ID of the `emacs` program that the `root` user is running.

Exercise: Tracing Processes That Are Run by a User

Create a program `countprogs.d` that counts and displays the number of times a user (specified by their user name) runs different programs. You can use the `id -u` *`user`* command to obtain the ID that corresponds to a user name.

(Estimated completion time: 10 minutes)

Solution to Exercise and Example: Counting Programs Invoked by a Specified User (countprogs.d)

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

/* countprogs.d -- Count programs invoked by a specified user */

proc::do_execveat_common:exec
/uid == $1/
{
  @num[execname] = count();
}
```

The predicate `/uid == $1/` compares the effective UID for each program that is run against the argument specified on the command line. You can use the `id -u` *`user`* command to find out the ID of the guest user account, for example:

```
# chmod +x countprogs.d
# ./countprogs.d $(id -u guest)
^C

less 1
lesspipe.sh 1
sh 1
bash 9
```

You can use the same command for the `root` user, which is typically user `0`. For testing purposes, you might want to have the user account under a test login by using another window and then run some nominal programs.

Example: Counting the Number of Times a Program Reads From Different Files in 10 Seconds (fdscount.d)

The following D program counts the number of times a program reads from different files, within ten seconds, and displays just the top five results.

```
# emacs fdscount.d
# dtrace -C -D ENAME='"emacs"' -qs fdscount.d

  /usr/share/terminfo/x/xterm                                       2
  /dev/urandom                                                      3
  /usr/share/emacs/24.3/lisp/calendar/time-date.elc                 5
  /dev/tty                                                          8
  /usr/share/emacs/24.3/lisp/term/xterm.elc                         8
```

Use the `fds[]` built-in array to determine which file corresponds to the file descriptor argument `arg0` to `read()`. The `fi_pathname` member of the `fileinfo_t` structure that is indexed in `fds[]` by `arg0` contains the full pathname of the file.

See fileinfo_t in the Oracle Linux: DTrace Reference Guide for more information about the members of the `fileinfo_t` structure.

The `trunc()` function in the `END` action instructs DTrace to display just the top five results from the aggregation.

DTrace has access to the `profile:::tick-10s` probe, the `fds[]` built-in array, and the `syscall::read:entry` probe. You specify a C preprocessor directive to `dtrace` that sets the value of the *ENAME* variable, such as to `emacs`. Although, you could choose any executable. Note that you must use additional single quotes to escape the string quotes, for example:

```
# dtrace -C -D ENAME='"emacs"' -qs fdscount.d

/usr/share/terminfo/x/xterm 2
/dev/tty 3
/dev/urandom 3
/usr/share/emacs/24.3/lisp/calendar/time-date.elc 5
/usr/share/emacs/24.3/lisp/term/xterm.elc 8
```

If the executable under test shows a `/proc/` *pid*`/maps` entry in the output, it refers to a file in the `procfs` file system that contains information about the process's mapped memory regions and permissions. Seeing `pipe:` *inode* and `socket:` *inode* entries would refer to inodes in the `pipefs` and `socketfs` file systems.

Exercise: Counting Context Switches on a System

Create an executable D program named `cswpercpu.d` that displays a timestamp and prints the number of context switches per CPU and the total for all CPUs once per second, together with the CPU number or `"total"`.

- Using the `BEGIN` probe, print a header for the display with columns labelled `Timestamp`, `CPU`, and `Ncsw`.
- Using the `sched:::on-cpu` probe to detect the end of a context switch, use `lltostr()` to convert the CPU number for the context in which the probe fired to a string, and use `count()` to increment the aggregation variable `@n` once with the key value set to the CPU number string and once with the key value set to `"total"`.

See sched Provider in the Oracle Linux: DTrace Reference Guide for a description of the `sched:::on-cpu` probe.

- Using the `profile:::tick-1sec` probe, use `printf()` to print the data and time, use `printa()` to print the key (the CPU number string or `"total"`) and the aggregation value. The date and time are available as the value of `walltimestamp` variable, which you can print using the `%Y` conversion format
- Use `clear()` to reset the aggregation variable `@n`.

(Estimated completion time: 40 minutes)

Solution to Exercise and Example: Counting Context Switches on a System

The following example shows the executable D program `cswpercpu.d`. The program displays a timestamp and prints the number of context switches, per-CPU, and the total for all CPUs, once per second, together with the CPU number or `"total"`:

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

/* cswpercpu.d -- Print number of context switches per CPU once per second */

#pragma D option quiet

dtrace:::BEGIN
{
  /* Print the header */
  printf("%-25s %5s %15s", "Timestamp", "CPU", "Ncsw");
}

sched:::on-cpu
{
  /* Convert the cpu number to a string */
  cpustr = lltostr(cpu);
  /* Increment the counters */
  @n[cpustr] = count();
  @n["total"] = count();
}

profile:::tick-1sec
{
  /* Print the date and time before the first result */
  printf("\n%-25Y ", walltimestamp);

  /* Print the aggregated counts for each CPU and the total for all CPUs */
  printa("%5s %@15d\n                          ", @n);

  /* Reset the aggregation */
  clear(@n);
}

# chmod +x cswpercpu.d
# ./cswpercpu.d
Timestamp                      CPU            Ncsw
2013 Nov  6 20:47:26             1             148
                                 0             155
                                 3             200
                                 2             272
                             total             775

2013 Nov  6 20:47:27             1             348
```

```
                                0             364
                                3             364
                                2             417
                            total            1493

2013 Nov  6 20:47:28            3              47
                                1             100
                                0             121
                                2             178
                            total             446
                            ^C
```

You might want to experiment with aggregating the total time that is spent context switching and the average time per context switch. For example, you can experiment by initializing a thread-local variable to the value of `timestamp` in the action to a `sched:::off-cpu` probe, and subtracting this value from the value of `timestamp` in the action to `sched:::on-cpu`. Use the `sum()` and `avg()` aggregation functions, respectively.

Working With More Complex Data Aggregations

Use the `lquantize()` and `quantize()` functions to display linear and power-of-two frequency distributions of data. See Aggregations in the Oracle Linux: DTrace Reference Guide for a description of aggregation functions.

Example: Displaying the Distribution of Read Sizes Resulting From a Command

As shown in the following example, you can display the distribution of the sizes specified to `arg2` of `read()` calls that were invoked by all instances of `find` that are running. After running the script, start a search with `find` in another window, such as `find .` or `find /..`

```
# dtrace -n 'syscall::read:entry /execname=="find"/
{@dist["find"]=quantize(arg2);}'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C

   find
           value  ------------- Distribution ------------- count
             256 | 0
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 6
            1024 | 0
            2048 | 0
            4096 |@@@@@@@@@@ 2
            8192 | 0
```

If the program is as simple as the program in the previous example, it is often convenient to run it from the command line.

Example: Displaying the Distribution of I/O Throughput for Block Devices (diskact.d)

In the following example, the `diskact.d` script uses `io` provider probes that are enabled by the `sdt` kernel module to display the distribution of I/O throughput for the block devices on the system.

```
#pragma D option quiet

/* diskact.d -- Display the distribution of I/O throughput for block devices */

io:::start
{
  start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
  /*
     You want to get an idea of our throughput to this device in KB/sec
     but you have values that are measured in bytes and nanoseconds.
     You want to calculate the following:

     bytes / 1024
     ------------------------
     nanoseconds / 1000000000

     As DTrace uses integer arithmetic and the denominator is usually
     between 0 and 1 for most I/O, the calculation as shown will lose
     precision. So, restate the fraction as:

     bytes         1000000000      bytes * 976562
     ----------- * ------------- = --------------
     nanoseconds   1024            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", @);
}
```

The `#pragma D option quiet` statement is used to suppress unwanted output and the `printa()` function is used to display the results of the aggregation.

See io Provider in the Oracle Linux: DTrace Reference Guide for a description of the arguments to the `io:::start` and `io:::done` probes.

See Output Formatting in the Oracle Linux: DTrace Reference Guide for a description of the `printa()` function.

After running the program for approximately a minute, type `Ctrl-C` to display the results:

```
# dtrace -s diskact.d
                    ^C

xvda2 (<unknown>)

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
               1 | 0
```

```
xvdc (<unknown>)

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
               1 | 0

xvdc1 (<unknown>)

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
               1 | 0

  dm-0 (<unknown>)

           value  ------------- Distribution ------------- count
             256 | 0
             512 |@@ 1
            1024 |@@ 1
            2048 |@@@@@@ 3
            4096 |@@@@@@@@@@ 5
            8192 |@@@@@@@@@@@@@@@@@@ 9
           16384 |@@@@ 2
           32768 | 0
```

Exercise: Displaying Read and Write I/O Throughput Separately

Create a version of `diskact.d` that aggregates the results separately for reading from, and writing to, block devices. Use a `tick` probe to collect data for 10 seconds.

- In the actions for `io:::start` and `io:::done`, assign the value of `args[0]->b_flags & B_READ ? "READ" : "WRITE"` to the variable `iodir`.
- In the actions for `io:::start` and `io:::done`, add `iodir` as a key to the `start[]` associative array.
- In the action for `io:::done`, add `iodir` as a key to the anonymous aggregation variable `@[]`.
- Modify the format string for `printa()` to display the value of the `iodir` key.

(Estimated completion time: 20 minutes)

Solution to Exercise: Displaying Read and Write I/O Throughput Separately

The following example shows a modified version of the `diskact.d` script, which displays separate results for read and write I/O:

```
#pragma D option quiet

/* rwdiskact.d -- Modified version of diskact.d that displays
                  separate results for read and write I/O     */

profile:::tick-10sec
{
  exit(0);
}

io:::start
```

```
{
  iodir = args[0]->b_flags & B_READ ? "READ" : "WRITE";
  start[args[0]->b_edev, args[0]->b_blkno, iodir] = timestamp;
}

io:::done
{
  iodir = args[0]->b_flags & B_READ ? "READ" : "WRITE";
  this->elapsed = timestamp - start[args[0]->b_edev,args[0]->b_blkno,iodir];
  @[args[1]->dev_statname, args[1]->dev_pathname, iodir] =
    quantize((args[0]->b_bcount * 976562) / this->elapsed);
  start[args[0]->b_edev, args[0]->b_blkno,iodir] = 0;}

END
{
  printa(" %s (%s) %s \n%@d\n", @);
}
```

In the example, adding the `iodir` variable to the tuple in the aggregation variable enables DTrace to display separate aggregations for read and write I/O operations.

```
# dtrace -s rwdiskact.d

^C
  xvda2 (<unknown>) WRITE

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
               1 | 0

  xvdc (<unknown>) WRITE

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
               1 | 0

  xvdc1 (<unknown>) WRITE

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
               1 | 0

  nfs (<nfs>) READ

           value  ------------- Distribution ------------- count
              -1 | 0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
               1 | 0

  dm-0 (<unknown>) WRITE

           value  ------------- Distribution ------------- count
            4096 | 0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
           16384 | 0
```

Example: Displaying Cumulative Read and Write Activity Across a File System Device (fsact)

The following example is a `bash` shell script that uses an embedded D program to display cumulative read and write block counts for a local file system according to their location on the file system's underlying block device. The `lquantize()` aggregation function is used to display the results linearly as tenths of the total number of blocks on the device.

```
#!/bin/bash

# fsact -- Display cumulative read and write activity across a file system device
#
#          Usage: fsact [<filesystem>]

# Could load the required DTrace modules, if they were not autoloaded.
# grep profile /proc/modules > /dev/null 2>&1 || modprobe profile
# grep sdt /proc/modules > /dev/null 2>&1 || modprobe sdt

# If no file system is specified, assume /
[ $# -eq 1 ] && FSNAME=$1 || FSNAME="/"
[ ! -e $FSNAME ] && echo "$FSNAME not found" && exit 1

# Determine the mountpoint, major and minor numbers, and file system size
MNTPNT=$(df $FSNAME | gawk '{ getline; print $1; exit }')
MAJOR=$(printf "%d\n" 0x$(stat -Lc "%t" $MNTPNT))
MINOR=$(printf "%d\n" 0x$(stat -Lc "%T" $MNTPNT))
FSSIZE=$(stat -fc "%b" $FSNAME)

# Run the embedded D program
dtrace -qs /dev/stdin << EOF
io:::done
/args[1]->dev_major == $MAJOR && args[1]->dev_minor == $MINOR/
{
  iodir = args[0]->b_flags & B_READ ? "READ" : "WRITE";
  /* Normalize the block number as an integer in the range 0 to 10 */
  blkno = (args[0]->b_blkno)*10/$FSSIZE;
  /* Aggregate blkno linearly over the range 0 to 10 in steps of 1 */
  @a[iodir] = lquantize(blkno,0,10,1)
}

tick-10s
{
  printf("%Y\n",walltimestamp);
  /* Display the results of the aggregation */
  printa("%s\n%@d\n",@a);
  /* To reset the aggregation every tick, uncomment the following line */
  /* clear(@a); */
}
EOF
```

You embed the D program in a shell script so that you can set up the parameters that are needed, which are the major and minor numbers of the underlying device and the total size of the file system in file system blocks. You then access these parameters directly in the D code.

> **Note:**
>
> An alternate way of passing values into the D program is to use C preprocessor directives, for example:
>
> ```
> dtrace -C -D MAJ=$MAJOR -D MIN=$MINOR -D FSZ=$FSSIZE -qs /dev/stdin << EOF
> ```
>
> You can then refer to the variables in the D program by their macro names instead of their shell names:
>
> ```
> /args[1]->dev_major == MAJ && args[1]->dev_minor == MIN/
>
> blkno = (args[0]->b_blkno)*10/FSZ;
> ```

The following example shows output from running the `fsact` command after making the script executable, then running `cp -R` on a directory and `rm -rf` on the copied directory:

```
# chmod +x fsact
# ./fsact
2018 Feb 16 16:59:46
READ

           value  ------------- Distribution ------------- count
             < 0 | 0
               0 |@@@@@@@ 8
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 32
               2 | 0
               3 | 0
               4 | 0
               5 | 0
               6 | 0
               7 | 0
               8 | 0
               9 | 0
           >= 10 |@@@@@@@ 8

WRITE

           value  ------------- Distribution ------------- count
               9 | 0
           >= 10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
42                                 0

^C
```

Displaying System Call Errors

The following information pertains to using the D program `errno.d` to display system call errors.

Example: Displaying System Call Errors (`errno.d`)

The following is an example of the D program, `errno.d`. In this example, the program displays the value of `errno` and the file name if an error occurs when using the `open()` system call to open a file.

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

/* errno.d -- Display errno and the file name for failed open() calls */

syscall::open:entry
{
  self->filename = copyinstr(arg0);
}

syscall::open:return
/arg0 < 0/
{
  printf("errno = %-2d   file = %s\n", errno, self->filename);
}
```

If an error occurs in the `open()` system call, the `return` probe sets the `arg0` argument to `-1` and the value of the built-in `errno` variable indicates the nature of the error. A predicate is used to test the value of `arg0`. Alternatively, you could test whether the value of `errno` is greater than zero.

When you have saved this script to a file and made the file executable, you can then run it to display information about any failures of the `open()` system call that occur on the system. After you have started the script, in a separate terminal window, you can run commands that result in an error, such as running the `ls` command to list a file that does not exist. Or, as in the following example, from another terminal the `cat` command has been issued on a directory, which results in an error:

```
# ./errno.d

errno = 2    file = /usr/share/locale/en_US.UTF-8/LC_MESSAGES/libc.mo
errno = 2    file = /usr/share/locale/en_US.utf8/LC_MESSAGES/libc.mo
errno = 2    file = /usr/share/locale/en_US/LC_MESSAGES/libc.mo
errno = 2    file = /usr/share/locale/en.UTF-8/LC_MESSAGES/libc.mo
errno = 2    file = /usr/share/locale/en.utf8/LC_MESSAGES/libc.mo
errno = 2    file = /usr/share/locale/en/LC_MESSAGES/libc.mo
^C
```

Exercise: Displaying More Information About System Call Errors

Adapt `errno.d` to display the name of the error instead of its number for any failed system call.

- The numeric values of errors such as `EACCES` and `EEXIST` are defined in `/usr/include/asm-generic/errno-base.h` and `/usr/include/asm-generic/errno.h`. DTrace defines inline names (which are effectively constants) for the numeric error values in `/usr/lib64/dtrace/` *kernel-version* `/errno.d`. Use an associative array named `error[]` to store the mapping between the inline names and the error names that are defined in `/usr/include/asm-generic/errno-base.h`.
- Use `printf()` to display the user ID, the process ID, the program name, the error name, and the name of the system call.
- Use the `BEGIN` probe to print column headings.
- Use the value of `errno` rather than `arg0` to test whether an error from the range of mapped names has occurred in a system call.

(Estimated completion time: 30 minutes)

Solution to Exercise: Displaying More Information About System Call Errors

The following is an example that shows a modified version of `errno.d`, which displays error names.

Example: Modified Version of errno.d Displaying Error Names (displayerrno.d)

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

/* displayerrno.d -- Modified version of errno.d that displays error names */

BEGIN
{
  printf("%-4s %-6s %-10s %-10s %s\n", "UID", "PID", "Prog", "Error", "Func");

  /* Assign error names to the associative array error[] */
  error[EPERM]   = "EPERM";    /* Operation not permitted */
  error[ENOENT]  = "ENOENT";   /* No such file or directory */
  error[ESRCH]   = "ESRCH";    /* No such process */
  error[EINTR]   = "EINTR";    /* Interrupted system call */
  error[EIO]     = "EIO";      /* I/O error */
  error[ENXIO]   = "ENXIO";    /* No such device or address */
  error[E2BIG]   = "E2BIG";    /* Argument list too long */
  error[ENOEXEC] = "ENOEXEC";  /* Exec format error */
  error[EBADF]   = "EBADF";    /* Bad file number */
  error[ECHILD]  = "ECHILD";   /* No child processes */
  error[EAGAIN]  = "EAGAIN";   /* Try again or operation would block */
  error[ENOMEM]  = "ENOMEM";   /* Out of memory */
  error[EACCES]  = "EACCES";   /* Permission denied */
  error[EFAULT]  = "EFAULT";   /* Bad address */
  error[ENOTBLK] = "ENOTBLK";  /* Block device required */
  error[EBUSY]   = "EBUSY";    /* Device or resource busy */
  error[EEXIST]  = "EEXIST";   /* File exists */
  error[EXDEV]   = "EXDEV";    /* Cross-device link */
  error[ENODEV]  = "ENODEV";   /* No such device */
  error[ENOTDIR] = "ENOTDIR";  /* Not a directory */
  error[EISDIR]  = "EISDIR";   /* Is a directory */
  error[EINVAL]  = "EINVAL";   /* Invalid argument */
  error[ENFILE]  = "ENFILE";   /* File table overflow */
  error[EMFILE]  = "EMFILE";   /* Too many open files */
  error[ENOTTY]  = "ENOTTY";   /* Not a typewriter */
  error[ETXTBSY] = "ETXTBSY";  /* Text file busy */
  error[EFBIG]   = "EFBIG";    /* File too large */
  error[ENOSPC]  = "ENOSPC";   /* No space left on device */
  error[ESPIPE]  = "ESPIPE";   /* Illegal seek */
  error[EROFS]   = "EROFS";    /* Read-only file system */
  error[EMLINK]  = "EMLINK";   /* Too many links */
  error[EPIPE]   = "EPIPE";    /* Broken pipe */
  error[EDOM]    = "EDOM";     /* Math argument out of domain of func */
  error[ERANGE]  = "ERANGE";   /* Math result not representable */
}

/* Specify any syscall return probe and test that the value of errno is in range */

syscall:::return
/errno > 0 && errno <= ERANGE/
{
  printf("%-4d %-6d %-10s %-10s %s()\n", uid, pid, execname, error[errno], probefunc);
}
```

```
# chmod +x displayerrno.d
# ./displayerrno.d
UID  PID    Prog       Error      Func
500  3575   test       EACCES     open()
500  3575   test       EINTR      clock_gettime()
^C
```

You could modify this program so that it displays verbose information about the nature of the error, in addition to the name of the error.

3
Tracing User-Space Applications

> **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 or Oracle Linux 9 as soon as possible.
>
> For more information about DTrace, see *Oracle Linux: DTrace Release Notes* and Oracle Linux: Using DTrace for System Tracing.

This chapter provides information about how to trace a user-space application and includes examples of D programs that you can use to investigate what is happening in an example user-space program.

Preparing for Tracing User-Space Applications

The DTrace helper device (`/dev/dtrace/helper`) enables a user-space application that contains DTrace probes to send probe provider information to DTrace.

To trace user-space processes that are run by users other than `root`, you must change the mode of the DTrace helper device to allow the user to record tracing information, as shown in the following examples.

Example: Changing the Mode of the DTrace Helper Device

The following example shows how you would enable the tracing of user-space applications by users other than the `root` user.

```
# chmod 666 /dev/dtrace/helper
```

Alternatively, if the `acl` package is installed on your system, you would use an ACL rule to limit access to a specific user, for example:

```
# setfacl -m u:guest:rw /dev/dtrace/helper
```

> **Note:**
>
> For DTrace to reference the probe points, you must change the mode on the device before the user begins running the program.

You can also create a `udev` rules file such as `/etc/udev/rules.d/10-dtrace.rules` to change the permissions on the device file each time the system boots.

The following example shows how you would change the mode of the device file by adding the following line to the `udev` rules file:

```
kernel=="dtrace/helper", MODE="0666"
```

The following example shows how you would change the ACL settings for the device file by adding a line similar to the following to the `udev` rules file:

```
kernel=="dtrace/helper", RUN="/usr/bin/setfacl -m u:guest:rw /dev/dtrace/helper"
```

To apply the `udev` rule without needing to restart the system, you would run the **start_udev** command.

Sample Application

This section provides a sample application to be used in subsequent exercises and examples in this chapter. The example, which illustrates a simple program, favors brevity and probing opportunity rather than completeness or efficiency.

> **Note:**
>
> The following simple program is provided for example purposes *only* and is not intended to efficiently solve a practical problem nor exhibit preferred coding methods.

The sample program finds the lowest factor of a number, which you input. The program is comprised of the following four files: `makefile`, `primelib.h`, `primelib.c`, and `primain.c` , which are stored in the same working directory.

Description and Format of the makefile File

The following example shows the contents of the `makefile` file.

> **Note:**
>
> A `makefile` must use tabs for indentation so that the `make` command can function properly. Also, be sure that tabs are retained if the file is copied and then used.

```
default: prime

# compile the library primelib first
primelib.o: primelib.c
    gcc -c primelib.c

# compile the main program
primain.o: primain.c
    gcc -c primain.c

# link and create executable file "prime"
prime: primelib.o primain.o
    gcc primain.o primelib.o -o prime -lm

clean:
```

```
    -rm -f *.o
    -rm -f prime
```

Description of the primelib.h Source File

The following example shows the contents of the `primelib.h` file.

```
int findMaxCheck( int inValue );
int seekFactorA( int input, int maxtry );
int seekFactorB( int input );
```

Description of the primelib.c Source File

The following example shows the contents of the `primelib.c` file.

```
#include <stdio.h>
#include <math.h>

/*
 * utility functions which are called from the main source code
 */

// Find and return our highest value to check -- which is the square root
int findMaxCheck( int inValue )  {
  float sqRoot;
  sqRoot = sqrt( inValue );
  printf("Square root of %d is %lf\n", inValue, sqRoot);
  return floor( sqRoot );
  return inValue/2;
}

int debugFlag = 0;

// Search for a factor to the input value, proving prime on return of zero
int seekFactorA( int input, int maxtry )  {
  int divisor, factor = 0;
  for( divisor=2; divisor<=maxtry; ++divisor ) {
    if( 0 == input%divisor ) {
      factor = divisor;
      break;
    }
    else if ( debugFlag != 0 )
      printf( "modulo %d yields: %d\n", divisor, input%divisor );
  }
  return factor;
}

// Search for a factor to the input value, proving prime on return of zero
// This is a different method than "A", using one argument
int seekFactorB( int input )  {
  int divisor, factor = 0;
  if( 0 == input%2 ) return 2;
  for( divisor=3; divisor<=input/2; divisor+=2 ) {
    if( 0 == input%divisor ) {
      factor = divisor;
      break;
    }
  }
  return factor;
}
```

Description of the primain.c Source File

The following example shows the contents of the `primain.c` file.

```
#include <stdio.h>
#include "primelib.h"

/*
 * Nominal C program churning to provide a code base we might want to
 * instrument with D
*/

// Search for a divisor -- thereby proving composite value of the input.
int main()  {
  int targVal, divisor, factorA=0, factorB=0;

  printf( "Enter a positive target integer to test for prime status: " );
  scanf( "%d", &targVal );

  // Check that the user input is valid
  if( targVal < 2 ) {
    printf( "Invalid input value -- exiting now\n" );
    return -2;
  }

  // Search for a divisor using method and function A
  int lastCheck;
  lastCheck = findMaxCheck( targVal );
  printf( "%d highest value to check as divisor\n", lastCheck );
  factorA = seekFactorA( targVal, lastCheck );

  // Search for a divisor using method and function B
  factorB = seekFactorB( targVal );

  // Warn if the methods give different results
  if (factorA != factorB)
    printf( "%d does not equal %d! How can this be?\n", factorA, factorB );

  // Print results
  if( !factorA )
    printf( "%d is a prime number\n", targVal );
  else
    printf( "%d is not prime because there is a factor %d\n",
        targVal, factorA );
  return 0;
}
```

Compiling the Program and Running the prime Executable

With the four files previously described located in the same working directory, compile the program by using the `make` command as follows:

```
# make
gcc -c primelib.c
gcc -c primain.c
gcc primain.o primelib.o -o prime -lm
```

Running the `make` command creates an executable named `prime`, which can be run to find the lowest prime value of the input, as shown in the following two examples:

```
# ./prime
Enter a positive target integer to test for prime status: 5099
Square root of 5099 is 71.407280
71 highest value to check as divisor
5099 is a prime number

# ./prime
Enter a positive target integer to test for prime status: 95099
Square root of 95099 is 308.381256
308 highest value to check as divisor
95099 is not prime because there is a factor 61
```

After compiling the program and running the `prime` executable, you can practice adding USDT probes to an application, as described in Adding USDT Probes to an Application.

Adding USDT Probes to an Application

In this section, we practice adding USDT probes to an application. For background information and other details, see Adding Probes to an Application in the Oracle Linux: DTrace Reference Guide.

To get started, you will need to create a `.d` file, as described in Defining Providers and Probes in the Oracle Linux: DTrace Reference Guide.

> **Note:**
>
> This `.d` file is not a script that is run in the same way that is shown in previous examples in this tutorial, but is rather the `.d` source file that you use when compiling and linking your application. To avoid any confusion, use a different naming convention for this file than you use for scripts.

After creating the `.d` file, you then need to create the required probe points to use in the following examples. This information is added to the `primain.c` source file. The probe points that are used in this practice are those listed in the following table. These probes represent a sequence of operations and are used after the user entry is completed and checked.

Table 3-1 Probe Points to Use After User Entry Completed and Checked

Description	Probe
User entry complete and checked	`userentry( int )`
Return and input to `seekFactorA()`	`factorreturnA( int, int )`
Return and input to `seekFactorB()`	`factorreturnB( int, int )`
Immediately prior to the program exiting	`final()`

Exercise: Creating a dprime.d File

To reflect the previously described probe points and data, create a file named `dprime.d` and store the file in the same working directory as the other source files.

> **Note:**
>
> Typically, you would provide additional information in the `.d` file, such as stability attributes, per the details that were previously referenced in the Oracle Linux: DTrace Reference Guide. For the sake of brevity, expedience, and simplicity, those details are not included in this introductory example.

(Estimated completion time: less than 5 minutes)

Solution to Exercise: Creating a dprime.d File

```
provider primeget
{
  probe query__userentry( int );
  probe query__maxcheckval( int, int );
  probe query__factorreturnA( int, int );
  probe query__factorreturnB( int, int );
  probe query__final();
};
```

Example: Creating a .h File From a dprime.d File

The next step is to create a `.h` file from the `dprime.d` file, as shown here:

```
# dtrace -h -s dprime.d
```

The `dprime.h` file that is created contains a reference to each of the probe points that are defined in the `dprime.d` file.

Next, in the application source file, `primain.c`, we add a reference to the `#include "dprime.h"` file and add the appropriate probe macros at the proper locations.

In the resulting `primain.c` file, the probe macros (shown in bold font for example purposes only) are easy to recognize, as they appear in uppercase letters:

```
#include <stdio.h>
#include "primelib.h"
#include "dprime.h"

/*
 * Nominal C program churning to provide a code base we might want to
 * instrument with D
*/

// Search for a divisor -- thereby proving composite value of the input.
int main()  {
  int targVal, divisor, factorA=0, factorB=0;

  printf( "Enter a positive target integer to test for prime status: " );
  scanf( "%d", &targVal );

  // Check that the user input is valid
  if( targVal < 2 ) {
    printf( "Invalid input value -- exiting now\n" );
    return -2;
  }
  if (PRIMEGET_QUERY_USERENTRY_ENABLED())
```

```
    PRIMEGET_QUERY_USERENTRY(targVal);

  // Search for a divisor using method and function A
  int lastCheck;
  lastCheck = findMaxCheck( targVal );
  printf( "%d highest value to check as divisor\n", lastCheck );
  if (PRIMEGET_QUERY_MAXCHECKVAL_ENABLED())
    PRIMEGET_QUERY_MAXCHECKVAL(lastCheck, targVal);

  factorA = seekFactorA( targVal, lastCheck );
  if (PRIMEGET_QUERY_FACTORRETURNA_ENABLED())
    PRIMEGET_QUERY_FACTORRETURNA(factorA, targVal);

  // Search for a divisor using method and function B
  factorB = seekFactorB( targVal );
 if (PRIMEGET_QUERY_FACTORRETURNB_ENABLED())
    PRIMEGET_QUERY_FACTORRETURNB(factorB, targVal);

  // Warn if the methods give different results
  if (factorA != factorB)
    printf( "%d does not equal %d! How can this be?\n", factorA, factorB );

  // Print results
  if( !factorA )
    printf( "%d is a prime number\n", targVal );
  else
    printf( "%d is not prime because there is a factor %d\n",
        targVal, factorA );
  if (PRIMEGET_QUERY_FINAL_ENABLED())
    PRIMEGET_QUERY_FINAL();

  return 0;
}
```

> **Note:**
>
> Any * `_ENABLED()` probe will translate into a truth value if the associated probe is enabled (some consumer is using it), and a false value if the associated probe is not enabled.

Before continuing, ensure that the probes are enabled and appear as the macros listed in the `dprime.h` file. See Testing if a Probe Is Enabled in the Oracle Linux: DTrace Reference Guide.

> **Note:**
>
> Make sure to include any desired values in the macros, if they exist, so that the probe can also identify those values.

Next, you will need to modify the `makefile` file. For step-by-step instructions, See Building Applications With Probes in the Oracle Linux: DTrace Reference Guide.

Exercise: Directing makefile to Re-Create the dprime.h File

Add a target that instructs `dtrace` to re-create the `dprime.h` file in the event that changes are subsequently made to the `dprime.d` file. This step ensures that you do not have to manually run the `dtrace -h -s dprime.d` command if any changes are made.

This exercise also has you direct `dtrace` to create a `prime.o` file.

(Estimated completion time: 10 minutes)

Solution to Exercise: Directing makefile to Re-Create the dprime.h File

```
default: prime

# re-create new dprime.h if dprime.d file has been changed
dprime.h: dprime.d
    dtrace -h -s dprime.d

# compile the library primelib first
primelib.o: primelib.c
    gcc -c primelib.c

# compile the main program
primain.o: primain.c dprime.h
    gcc -c primain.c

# have dtrace post-process the object files
prime.o: dprime.d primelib.o primain.o
    dtrace -G -s dprime.d primelib.o primain.o -o prime.o

# link and create executable file "prime"
prime: prime.o
    gcc -Wl,--export-dynamic,--strip-all -o prime prime.o primelib.o primain.o dprime.h -
lm


clean:
    -rm -f *.o
    -rm -f prime
    -rm -f dprime.h
```

Example: Testing the Program

After creating a fresh build, test that the executable is still working as expected:

```
# make clean
rm -f *.o
rm -f prime
rm -f dprime.h


# make
gcc -c primelib.c
dtrace -h -s dprime.d
gcc -c primain.c
dtrace -G -s dprime.d primelib.o primain.o -o prime.o
gcc -Wl,--export-dynamic,--strip-all -o prime prime.o primelib.o primain.o dprime.h -lm
```

```
# ./prime
Enter a positive target integer to test for prime status: 6799
Square root of 6799 is 82.456047
82 highest value to check as divisor
6799 is not prime because there is a factor 13
```

Using USDT Probes

This section provides some practice in the nominal use of the USDT probes that were created in Adding USDT Probes to an Application.

Initially, the probes are not visible because the application is not running with the probes, as shown in the following output:

```
# dtrace -l -P 'prime*'
  ID   PROVIDER            MODULE                          FUNCTION NAME
dtrace: failed to match prime*:::: No probe matches description
```

Start the application, but do not enter any value until you have listed the probes:

```
# ./prime
Enter a positive target integer to test for prime status:
```

From another command line, issue a probe listing:

```
# dtrace -l -P 'prime*'
   ID      PROVIDER            MODULE                          FUNCTION NAME
 2475 primeget26556             prime                              main query-
factorreturnA
 2476 primeget26556             prime                              main query-
factorreturnB
 2477 primeget26556             prime                              main query-final
 2478 primeget26556             prime                              main query-maxcheckval
 2479 primeget26556             prime                              main query-userentry
```

> **Note:**
>
> The provider name is a combination of the defined `provider primeget`, from the `dprime.d` file, and the PID of the running application `prime`. The output of the following command displays the PID of prime:
>
> ```
> # ps aux | grep prime
> root 26556 0.0 0.0 7404 1692 pts/0 S+ 21:50 0:00 ./prime
> ```

If you want to be able to run USDT scripts for users other than `root`, the helper device must have the proper permissions. Alternatively, you can run the program with the probes in it as the `root` user. See Example: Changing the Mode of the DTrace Helper Device for more information about changing the mode of the DTrace helper device.

One method for getting these permissions is to run the following command to change the configuration so that users other than the `root` user can send probe provider information to DTrace:

```
# setfacl -m u:guest:rw /dev/dtrace/helper
```

Start the application again, but do not enter any values until the probes are listed:

```
# ./prime
Enter a positive target integer to test for prime status:
```

From another command line, issue a probe listing:

```
# dtrace -l -P 'prime*'
   ID     PROVIDER            MODULE                  FUNCTION NAME
 2456 primeget2069             prime                      main query-factorreturnA
 2457 primeget2069             prime                      main query-factorreturnB
 2458 primeget2069             prime                      main query-final
 2459 primeget2069             prime                      main query-maxcheckval
 2460 primeget2069             prime                      main query-userentry
```

Example: Using simpleTimeProbe.d to Show the Elapsed Time Between Two Probes

The following example shows how you would create a simple script that measures the time elapsed between the first probe and the second probe (`query-userentry` to `query-maxcheckval`).

```
/* simpleTimeProbe.d */

/* Show how much time elapses between two probes */

primeget*:::query-userentry
{
  self->t = timestamp; /* Initialize a thread-local variable with the time */
}

primeget*:::query-maxcheckval
/self->t != 0/
{
  timeNow = timestamp;
  /* Divide by 1000 for microseconds */
  printf("%s (pid=%d) spent %d microseconds between userentry & maxcheckval\n",
         execname, pid, ((timeNow - self->t)/1000));

  self->t = 0; /* Reset the variable */
}
```

Start the execution of the target application:

```
# ./prime
Enter a positive target integer to test for prime status:
```

Then, run the DTrace script from another window:

```
# dtrace -q -s simpleTimeProbe.d
```

As the application is running, the output of the script is also running in parallel:

```
# ./prime
Enter a positive target integer to test for prime status: 7921
Square root of 7921 is 89.000000
89 highest value to check as divisor
7921 is not prime because there is a factor 89
# ./prime
Enter a positive target integer to test for prime status: 995099
Square root of 995099 is 997.546509
997 highest value to check as divisor
```

```
995099 is not prime because there is a factor 7
# ./prime
Enter a positive target integer to test for prime status: 7921
Square root of 7921 is 89.000000
89 highest value to check as divisor
7921 is not prime because there is a factor 89
```

On the command line where the script is being run, you should see output similar to the following:

```
# dtrace -q -s simpleTimeProbe.d
prime (pid=2328) spent 45 microseconds between userentry & maxcheckval
prime (pid=2330) spent 41 microseconds between userentry & maxcheckval
prime (pid=2331) spent 89 microseconds between userentry & maxcheckval
^C
```

Example: Using timeTweenprobes.d to Show the Elapsed Time Between Each Probe

You can broaden the script to monitor all of the following probes in the application:

- query-userentry
- query-maxcheckval
- query-factorreturnA
- query-factorreturnB
- query-final

```
/* timeTweenProbes.d */

/* show how much time elapses between each probe */

BEGIN
{
  iterationCount = 0;
}


primeget*:::query-userentry
{
  printf("%s (pid=%d) running\n", execname, pid);
  self->t = timestamp; /* Initialize a thread-local variable with time */
}

primeget*:::query-maxcheckval
/self->t != 0/
{
  timeNow = timestamp;
  printf(" maxcheckval spent %d microseconds since userentry\n",
         ((timeNow - self->t)/1000));  /* Divide by 1000 for microseconds */
  self->t = timeNow; /* set the time to recent sample */
}

primeget*:::query-factorreturnA
/self->t != 0/
{
  timeNow = timestamp;
  printf(" factorreturnA spent %d microseconds since maxcheckval\n",
         ((timeNow - self->t)/1000));  /* Divide by 1000 for microseconds */
```

```
  self->t = timeNow; /* set the time to recent sample */
}

primeget*:::query-factorreturnB
/self->t != 0/
{
  timeNow = timestamp;
  printf(" factorreturnB spent %d microseconds since factorreturnA\n",
         ((timeNow - self->t)/1000));  /* Divide by 1000 for microseconds */
  self->t = timeNow; /* set the time to recent sample */
}

primeget*:::query-final
/self->t != 0/
{
  printf(" prime spent %d microseconds from factorreturnB until ending\n",
         ((timestamp - self->t)/1000));
  self->t = 0; /* Reset the variable */
  iterationCount++;
}

END
{
  trace(iterationCount);
}
```

Again, start the execution of the target application first, then run the script from another window:

```
# ./prime
Enter a positive target integer to test for prime status: 995099
Square root of 995099 is 997.546509
997 highest value to check as divisor
995099 is not prime because there is a factor 7
# ./prime
Enter a positive target integer to test for prime status: 7921
Square root of 7921 is 89.000000
89 highest value to check as divisor
7921 is not prime because there is a factor 89
# ./prime
Enter a positive target integer to test for prime status: 95099
Square root of 95099 is 308.381256
308 highest value to check as divisor
95099 is not prime because there is a factor 61
# ./prime
Enter a positive target integer to test for prime status: 95099
Square root of 95099 is 308.381256
308 highest value to check as divisor
95099 is not prime because there is a factor 61
# ./prime
Enter a positive target integer to test for prime status: 5099
Square root of 5099 is 71.407280
71 highest value to check as divisor
5099 is a prime number
```

The corresponding output from the script is similar to the following:

```
# dtrace -q -s ./timeTweenProbes.d
prime (pid=2437) running
 maxcheckval spent 96 microseconds since userentry
 factorreturnA spent 9 microseconds since maxcheckval
 factorreturnB spent 6 microseconds since factorreturnA
```

```
 prime spent 9 microseconds from factorreturnB until ending
prime (pid=2439) running
 maxcheckval spent 45 microseconds since userentry
 factorreturnA spent 10 microseconds since maxcheckval
 factorreturnB spent 7 microseconds since factorreturnA
 prime spent 9 microseconds from factorreturnB until ending
prime (pid=2440) running
 maxcheckval spent 43 microseconds since userentry
 factorreturnA spent 11 microseconds since maxcheckval
 factorreturnB spent 8 microseconds since factorreturnA
 prime spent 10 microseconds from factorreturnB until ending
prime (pid=2441) running
 maxcheckval spent 53 microseconds since userentry
 factorreturnA spent 10 microseconds since maxcheckval
 factorreturnB spent 7 microseconds since factorreturnA
 prime spent 10 microseconds from factorreturnB until ending
prime (pid=2442) running
 maxcheckval spent 40 microseconds since userentry
 factorreturnA spent 9 microseconds since maxcheckval
 factorreturnB spent 48 microseconds since factorreturnA
 prime spent 10 microseconds from factorreturnB until ending

^C
5
```

As is observed in the previous example, there is now a set of DTrace features that can be used with the probes that were created.

4
Going Further With DTrace

> **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 or Oracle Linux 9 as soon as possible.
>
> For more information about DTrace, see *Oracle Linux: DTrace Release Notes* and Oracle Linux: Using DTrace for System Tracing.

For more information about using DTrace on Oracle Linux, see Oracle Linux: DTrace Reference Guide .

The latest DTrace development work and source code for Linux is available at https://github.com/oracle/dtrace-utils/.

You may also reference the information at https://www.oracle.com/linux/downloads/linux-dtrace.html.