Abstract

This tutorial introduces the Dynamic Tracing (DTrace) feature of Oracle Linux and shows how you can use the D language to trace the behavior of the operating system and user-space programs. The tutorial includes practical examples that you can run and provides exercises with solutions that will enable you to learn more about using DTrace.

Document generated on: 2015-08-10 (revision: 3178)
Table of Contents

Preface ........................................................................................................................................................................ vii
1 Introducing DTrace .................................................................................................................................................. 1
   1.1 About Using this Tutorial ......................................................................................................................... 1
   1.2 About DTrace ............................................................................................................................................... 1
   1.3 About DTrace Providers .......................................................................................................................... 2
   1.4 Preparation: Installing and Configuring DTrace .................................................................................. 3
      1.4.1 Changing the Mode of the DTrace Helper Device ............................................................................... 4
      1.4.2 Loading DTrace Kernel Modules ...................................................................................................... 5
   1.5 Running a Simple DTrace Program ........................................................................................................... 6
2 Tracing Operating System Behavior .......................................................................................................................... 9
   2.1 Tracing Process Creation .......................................................................................................................... 9
   2.2 Tracing System Calls .................................................................................................................................. 10
   2.3 Performing an Action at Specified Intervals .......................................................................................... 11
   2.4 Using Predicates to Select Actions ........................................................................................................ 13
   2.5 Timing Events on a System ..................................................................................................................... 15
   2.6 Tracing Parent and Child Processes .......................................................................................................... 17
   2.7 Simple Data Aggregations ....................................................................................................................... 20
   2.8 More Complex Data Aggregations ........................................................................................................... 26
   2.9 Displaying System Call Errors ............................................................................................................... 31
3 Tracing User-Space Applications .................................................................................................................................. 35
   3.1 Preparation: Installing DTrace-Enabled Applications ........................................................................... 35
   3.2 Tracing a User-Space Application ........................................................................................................... 35
   3.3 Using Aggregations with User-Space Applications ................................................................................ 37
   3.4 Tracing the Flow of Execution ................................................................................................................ 38
   3.5 Detecting PHP Errors .............................................................................................................................. 41
   3.6 Using a Speculation for Error Analysis .................................................................................................. 42
4 Going Further with DTrace ...................................................................................................................................... 47
List of Examples

1.1 hello.d: A simple D program that uses the BEGIN probe ............................................. 6
1.2 goodbye.d: Simple D program that demonstrates the END probe .................................. 7
2.1 execcalls.d: Monitor the system as it executes programs ............................................. 10
2.2 syscall.d: Record open() system calls on a system .................................................... 21
2.3 syscall1.d: Modified version of syscall.d that displays more information ..................... 13
2.4 tick.d: Perform an action at regular intervals ......................................................... 13
2.5 tick1.d: Modified version of tick.d .............................................................................. 13
2.6 daterun.d: Display arguments to write() when date runs ........................................ 15
2.7 wrun.d: Modified version of daterun.d for the w command ...................................... 16
2.8 readtrace.d: Display time spent in read() calls ...................................................... 16
2.9 readtrace1.d: Modified version of readtrace.d that includes a predicate .................... 16
2.10 calltrace.d: Time all system calls for firefox ...................................................... 17
2.11 activity.d: Record fork() and exec() activity ...................................................... 17
2.12 activity1.d: Record fork() and exec() activity for a specified program .................. 19
2.13 countcalls.d: Count write, read, and open system calls over 100 seconds ............. 21
2.14 countsyscall.d: Count system calls invoked by a process ..................................... 21
2.15 countprogs.d: Count programs invoked by a specified user ...................................... 23
2.16 fdscount.d: Count the number of times that a program reads from different files ... 23
2.17 cswpercpu.d: Print number of context switches per CPU once per second ............... 25
2.18 diskact.d: Display the distribution of I/O throughput for block devices .................... 26
2.19 rwdiskact.d: Modified version of diskact.d that displays separate results for read and write I/O .. 28
2.20 fsact: Display cumulative read and write activity across a file system device .......... 29
2.21 errno.d: Display errno and the file name for failed open() calls ............................... 31
2.22 displayerrno.d: Modified version of errno.d that displays error names .................... 33
3.1 func.php: Infinitely looping, recursive test script ...................................................... 35
3.2 aggfunc.d: Aggregate counts for a PHP program ..................................................... 37
3.3 flow.d: Trace the flow of execution in func.php ...................................................... 38
3.4 flow1.d: Modified version of flow.d that also displays system calls .......................... 40
3.5 detpherr.d: Detect errors in PHP programs .............................................................. 41
3.6 detpherr1.d: Modified version of detpherr.d that includes the function name and a timestamp ...... 42
3.7 errortrace.d: Print complete trace after one or more errors occur .......................... 43
3.8 errortrace1.d: Print a function call trace as soon as an error occurs .......................... 45
Preface

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

**Audience**

This document is intended for administrators and developers who want to become familiar with the capabilities of DTrace and the D programming language. It is assumed that readers have a general understanding of the Linux operating system together with experience of using a programming language such as C or C++ and a scripting language such as PHP.

**Document Organization**

The document is organized as follows:

- **Chapter 1, *Introducing DTrace*** introduces the dynamic tracing (DTrace) facility that you can use to examine the behavior of the operating system and user-space programs that have been instrumented with DTrace probes.
- **Chapter 2, *Tracing Operating System Behavior*** provides examples of D programs that you can use to investigate what is happening in the operating system.
- **Chapter 3, *Tracing User-Space Applications*** provides examples of D programs that you can use to investigate what is happening in user-space programs.
- **Chapter 4, *Going Further with DTrace*** contains suggestions for where you can find more information about using DTrace.

**Related Documents**

The documentation for this product is available at:


**Conventions**

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
Chapter 1 Introducing DTrace

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. Version 0.4 of DTrace is described, which is supported for use with the Unbreakable Enterprise Kernel Release 3 (UEK R3).

1.1 About Using this Tutorial

This tutorial includes a variety of DTrace scripts and describes different ways in which you can use DTrace. Most of the 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 time or less time. You should already have a good understanding of Linux administration and system programming, and broad experience of using a programming language such as C or C++ and a scripting language such as PHP. If you are not familiar with terms such as system call, type, cast, signal, struct, or pointer, you might find difficulty in understanding some of the examples or completing some of the exercises. 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 programs.

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. 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.

Perform the examples and exercises in this tutorial on a system other than a production system.

The examples 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 dtrace 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.

1.2 About DTrace

DTrace is a comprehensive dynamic tracing facility that was first developed for the Oracle Solaris operating system, 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 places of interest called probes in the 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 from it and reports the data back to you.

Using DTrace’s D programming language, you can query the system probes to provide immediate, concise answers to arbitrary questions that you 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. You do not need to specify probes exactly, which allows DTrace to perform the same action for a number of different probes.

When you use the `dtrace` command to run a D program, you invoke the compiler for the D language. Once 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 run-time errors that might occur during your D program's execution, including dividing by zero, dereferencing invalid memory, and so on, and reports them to you.

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 kernel or any process that is running on your system. These safety features allow you to use DTrace in a production environment without worrying about crashing or corrupting your system. If you make a programming mistake, DTrace 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 Dynamic Tracing Guide.

### 1.3 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.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Kernel Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtrace</td>
<td>dtrace</td>
<td>Provides probes that relate to DTrace itself, such as <code>BEGIN</code>, <code>ERROR</code>, and <code>END</code>. 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.</td>
</tr>
<tr>
<td>fasttrap</td>
<td>fasttrap</td>
<td>Supports user-space tracing of DTrace-enabled applications.</td>
</tr>
<tr>
<td>io</td>
<td>sdt</td>
<td>Provides probes that relate to data input and output. The <code>io</code> provider enables quick exploration of behavior observed through I/O monitoring tools such as <code>iostat</code>.</td>
</tr>
<tr>
<td>proc</td>
<td>sdt</td>
<td>Provides probes for monitoring process creation and termination, LWP creation and termination, execution of new programs, and signal handling.</td>
</tr>
<tr>
<td>profile</td>
<td>profile</td>
<td>Provides probes associated with an interrupt that fires at a fixed, specified time interval. These probes are associated with the asynchronous interrupt event rather than with any particular point of execution. You can use these probes to sample some aspect of a system's state.</td>
</tr>
<tr>
<td>sched</td>
<td>sdt</td>
<td>Provides probes related to CPU scheduling. Because CPUs are the one resource that all threads must consume, the <code>sched</code> provider is very useful for understanding systemic behavior.</td>
</tr>
</tbody>
</table>
| syscall  | systrace      | Provides probes at the entry to and return from every system call. Because system calls are the primary interface between user-level
Preparation: Installing and Configuring DTrace

<table>
<thead>
<tr>
<th>Provider</th>
<th>Kernel Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>applications and the operating system kernel, these probes can offer you an insight into the interaction between applications and the system.</td>
</tr>
</tbody>
</table>

See Providers in the Oracle Linux Dynamic Tracing Guide for more information about providers and their probes.

1.4 Preparation: Installing and Configuring DTrace

Note

The DTrace dtrace-utils package is available from ULN. Your system must be registered with ULN and be installed with or be updated to Oracle Linux Release 6 Update 4 or later or Oracle Linux 7.

To install and configure DTrace, perform the following steps:

1. On ULN, subscribe your system to the appropriate channels.

   For Oracle Linux 6 Update 4 or later, subscribe to the following channels:
   - Oracle Linux 6 Latest (x86_64) (ol6_x86_64_latest)
   - Unbreakable Enterprise Kernel Release 3 for Oracle Linux 6 (x86_64) - Latest (ol6_x86_64_UEKR3_latest)
   - Oracle Linux 6 Dtrace Userspace Tools (x86_64) - Latest (ol6_x86_64_Dtrace_userspace_latest)

   For Oracle Linux 7, subscribe to the following channels:
   - Oracle Linux 7 Latest (x86_64) (ol7_x86_64_latest)
   - Unbreakable Enterprise Kernel Release 3 for Oracle Linux 7 (x86_64) - Latest (ol7_x86_64_UEKR3)
   - Oracle Linux 7 Dtrace Userspace Tools (x86_64) - Latest (ol7_x86_64_Dtrace_userspace)

Note

Make sure that your system is not subscribed to the following channels:

   - Latest Unbreakable Enterprise Kernel for Oracle Linux 6 (x86_64) (ol6_x86_64_USEK_latest)
   - Dtrace for Oracle Linux 6 (x86_64) - Latest (ol6_x86_64_Dtrace_latest)
   - Dtrace for Oracle Linux 6 (x86_64) - Beta release (ol6_x86_64_Dtrace_BETA)
   - Unbreakable Enterprise Kernel Release 3 (3.8 based) for Oracle Linux 6 (x86_64) - Beta release (ol6_x86_64_UEKR_BETA)

These channels are applicable to UEK R2, DTrace for UEK R2, the beta release of DTrace for UEK R2, and the beta release of UEK R3.
2. If your system is not already running the latest version of the Unbreakable Enterprise Kernel Release 3 (UEK R3):
   a. Use `yum` to update your system to use UEK R3:
      
      ```
      # yum update
      ```
   b. Reboot the system, selecting the Oracle Linux Server (3.8.13) kernel in the GRUB menu if it is not the default kernel.

3. Use `yum` to install the DTrace utilities package:

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

   If you subsequently use `yum update` to install a new kernel, `yum` does not automatically install the matching `dtrace-modules` package that the kernel requires. If the appropriate `dtrace-modules` package for the running kernel is not present on the system, the `dtrace` command downloads and installs the package from ULN. To invoke this action without performing a trace, use a command such as the following:

   ```
   # dtrace -l
   ```

   Alternatively, run the following command to install the DTrace module that is appropriate to the running kernel:

   ```
   # yum install dtrace-modules-`uname -r`
   ```

   If you want to implement a `libdtrace` consumer or develop a DTrace provider, use `yum` to install the `dtrace-utils-devel` or `dtrace-modules-provider-headers` package respectively.

   To be able to trace user-space processes that are run by users other than `root`, change the mode of the DTrace helper device as described in Section 1.4.1, “Changing the Mode of the DTrace Helper Device”.

### 1.4.1 Changing the Mode of the DTrace Helper Device

The DTrace helper device (`/dev/dtrace/helper`) allows 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, for example:

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

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

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

---

**Note**

You must change the mode on the device before the user runs the program.

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

To change the mode of the device file, the udev rules file should contain the following line:

```
kernel=="dtrace/helper", MODE="0666"
```
To change the ACL settings for the device file, use a line such as the following in 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, run the `start_udev` command.

### 1.4.2 Loading DTrace Kernel Modules

Use the `modprobe` command to load the modules that support the DTrace probes that you want to use. For example, if you wanted to use the probes that the `proc` provider publishes, you would load the `sdt` module.

```
# modprobe sdt
```

**Note**

The `fasttrap`, `profile`, `sdt`, and `systrace` modules automatically load the `dtrace` module.

To display the probes that are provided by a provider such as `proc`, use the following form of the `dtrace` command:

```
# dtrace -l -P proc
```

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3466</td>
<td>proc</td>
<td>vmlinux</td>
<td>schedule_tail</td>
<td>start</td>
</tr>
<tr>
<td>3467</td>
<td>proc</td>
<td>vmlinux</td>
<td>schedule_tail</td>
<td>lwp-start</td>
</tr>
<tr>
<td>3469</td>
<td>proc</td>
<td>vmlinux</td>
<td>get_signal_to_deliver</td>
<td>signal-handle</td>
</tr>
<tr>
<td>3474</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_sigtimedwait</td>
<td>signal-clear</td>
</tr>
<tr>
<td>3475</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_fork</td>
<td>lwp-create</td>
</tr>
<tr>
<td>3476</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_fork</td>
<td>create</td>
</tr>
<tr>
<td>3477</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_exit</td>
<td>lwp-exit</td>
</tr>
<tr>
<td>3478</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_exit</td>
<td>exit</td>
</tr>
<tr>
<td>3479</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_execve_common</td>
<td>exec-failure</td>
</tr>
<tr>
<td>3480</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_execve_common</td>
<td>exec</td>
</tr>
<tr>
<td>3481</td>
<td>proc</td>
<td>vmlinux</td>
<td>do_execve_common</td>
<td>exec-success</td>
</tr>
<tr>
<td>3485</td>
<td>proc</td>
<td>vmlinux</td>
<td>__send_signal</td>
<td>signal-send</td>
</tr>
<tr>
<td>3486</td>
<td>proc</td>
<td>vmlinux</td>
<td>__send_signal</td>
<td>signal-discard</td>
</tr>
</tbody>
</table>

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 there is no ambiguity with other probes for the same provider, you can usually omit the `MODULE` and even the `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 allow you to monitor how the system creates processes, executes programs, and handles signals.

### Exercise 1.1: Enabling and listing DTrace probes

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

(Estimated completion time: less than 5 minutes)
Running a Simple DTrace Program

### Solution to Exercise 1.1

```bash
# modprobe systrace
# dtrace -l -P syscall
```

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>syscall</td>
<td>vmlinux</td>
<td>read entry</td>
</tr>
<tr>
<td>5</td>
<td>syscall</td>
<td>vmlinux</td>
<td>read return</td>
</tr>
<tr>
<td>6</td>
<td>syscall</td>
<td>vmlinux</td>
<td>write entry</td>
</tr>
<tr>
<td>7</td>
<td>syscall</td>
<td>vmlinux</td>
<td>write return</td>
</tr>
<tr>
<td>8</td>
<td>syscall</td>
<td>vmlinux</td>
<td>open entry</td>
</tr>
<tr>
<td>9</td>
<td>syscall</td>
<td>vmlinux</td>
<td>open return</td>
</tr>
<tr>
<td>10</td>
<td>syscall</td>
<td>vmlinux</td>
<td>close entry</td>
</tr>
<tr>
<td>11</td>
<td>syscall</td>
<td>vmlinux</td>
<td>close return</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>syscall</td>
<td>vmlinux</td>
<td>finit_module entry</td>
</tr>
<tr>
<td>601</td>
<td>syscall</td>
<td>vmlinux</td>
<td>finit_module return</td>
</tr>
<tr>
<td>602</td>
<td>syscall</td>
<td>vmlinux</td>
<td>waitfd entry</td>
</tr>
<tr>
<td>603</td>
<td>syscall</td>
<td>vmlinux</td>
<td>waitfd return</td>
</tr>
</tbody>
</table>

**Note**

The probe IDs numbers might differ on your system, depending on what other providers you have loaded.

---

### 1.5 Running a Simple DTrace Program

Using a text editor, create a new file called `hello.d` and type in this D program:

**Example 1.1 hello.d: A simple D program that uses the BEGIN probe**

```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 (`;`). The function `trace` tells 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`. As these probe names are unique to the `dtrace` provider, we can shorten their names to `BEGIN`, `END`, and `ERROR`.

After you have saved your program, you can run it by using the `dtrace` command with the `-s` option to specify the name of the file that contains the D program. Type the following command:

```bash
# 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 1.2: 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 make the probe fire and exit dtrace.

(Estimated completion time: 5 minutes)

**Solution to Exercise 1.2**

**Example 1.2 goodbye.d: Simple D program that demonstrates the END probe**

```d
/* 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
```

The next chapter shows how you can use DTrace to explore different aspects of what is happening in the operating system.
2.1 Tracing Process Creation

The proc probes allow 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 Dynamic Tracing Guide for a description of the proc probes and their arguments.

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

Example 2.1 execcalls.d: Monitor the system as it executes programs

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

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

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

Before using dtrace to run the script, load the sdt kernel module to enable the proc provider probes. (This is only necessary if the module has not already been loaded.)

```
mmodprobe sdt
```

Enter the command dtrace -s execcalls.d to run the D program in one window. Then start different programs from another window, and observe 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
 0    600      do_execve_common:exec /bin/uname
 0    600      do_execve_common:exec /bin/mkdir
 0    600      do_execve_common:exec /bin/sed
 0    600      do_execve_common:exec /usr/bin/dirname
 1    600      do_execve_common:exec /usr/local/bin/firefox
 1    600      do_execve_common:exec /usr/bin/firefox
 1    600      do_execve_common:exec /bin/basename
 1    600      do_execve_common:exec /bin/uname
 1    600      do_execve_common:exec /usr/bin/mozilla-plugin-config
 1    600      do_execve_common:exec /usr/lib64/nspluginwrapper/plugin-config
 1    600      do_execve_common:exec /bin/sed
 1    600      do_execve_common:exec /usr/lib64/firefox-3.6/run-mozilla.sh
 1    600      do_execve_common:exec /bin/basename
 1    600      do_execve_common:exec /bin/uname
 1    600      do_execve_common:exec /usr/lib64/firefox-3.6/firefox
```

The probe proc::do_execve_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 2.1: Suppressing verbose output from DTrace

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

(Estimated completion time: less than 5 minutes)

Solution to Exercise 2.1

```
# 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
```

2.2 Tracing System Calls

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

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

Example 2.2 `syscalls.d`: Record open() system calls on a system

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

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

In this example, we use the `printf()` function 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**

We 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()`. For more information, see *User Process Tracing* in the Oracle Linux Dynamic Tracing Guide.

Before using `dtrace` to run the script, we load the `systrace` kernel module to enable the `syscall` provider probes. (This is only necessary if the module has not already been loaded.)

```
# modprobe systrace
# dtrace -q -s syscalls.d
udisks-daemon /dev/sr0
devkit-power-da /sys/devices/LNXSYSTM:00/.../PNP0C0A:00/power_supply/BAT0/present
```
Performing an Action at Specified Intervals

Exercise 2.2: 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 Dynamic Tracing 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 2.2

Example 2.3 syscalls1.d: Modified version of syscalls.d that displays more information

```plaintext
/* 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 the example uses similar formatting strings to output the header and the data.

```plaintext
# 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
...
```

2.3 Performing an Action at Specified Intervals

The `profile` provider provides the `tick` probes that you can use to sample some aspect of a system's state at regular intervals. You must load the `profile` kernel module to use these probes.
Performing an Action at Specified Intervals

The following example program, `tick.d`, declares and initializes the variable `i` when the D program starts, displays its initial value, increments the variable and prints its value once every second, and displays the final value of `i` when the program exits.

Example 2.4 tick.d: Perform an action at regular intervals

```d
/* tick.d -- Perform an action at regular intervals */
BEGIN
{
    i = 0;
}
profile:::tick-1sec
{
    printf("i = %d\n",++i);
}
END
{
    trace(i);
}
```

When you run this program, it produces output such as the following until you type `Ctrl-C`:

```
# modprobe profile
# dtrace -s tick.d
dtrace: script 'tick.d' matched 3 probes
CPU   ID FUNCTION:NAME
1 618   :tick-1sec i = 1
1 618   :tick-1sec i = 2
1 618   :tick-1sec i = 3
1 618   :tick-1sec i = 4
1 618   :tick-1sec i = 5
^C
0 2   :END  5
```

You can suppress all output except that from `printf()` and `trace()` by specifying the `-q` option:

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

Exercise 2.3: 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 Dynamic Tracing Guide` for a description of the probes.

(Estimated completion time: 10 minutes)
Solution to Exercise 2.3

```bash
# dtrace -l -P profile
ID PROVIDER MODULE FUNCTION NAME
635 profile                     tick-1
636 profile                     tick-10
637 profile                    tick-100
638 profile                   tick-500
639 profile                  tick-1000
640 profile                 tick-5000
```

Example 2.5 tick1.d: Modified version of tick.d

```c
/* 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 that fires twice per second.

```bash
# 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
```

2.4 Using Predicates to Select Actions

Predicates are logic statements that select 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.

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

Example 2.6 daterun.d: Display arguments to write() when date runs

```bash
#!/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 \texttt{/execname == "date"/}, which specifies that if the probe \texttt{syscall::write:entry} is triggered, DTrace runs the associated action only if the name of the executable is \texttt{date}.

Make the script executable by changing its mode:

\begin{verbatim}
# chmod +x daterun.d
\end{verbatim}

\begin{itemize}
\item \textbf{Note}
\end{itemize}

Before running the script, remember to use \texttt{modprobe} to load the \texttt{systrace} kernel module if this module has not already been loaded.

If you run the script from one window, while typing the \texttt{date} command in another, you see output such as the following in the first window:

\begin{verbatim}
#/daterun.d
write(1, Thu Oct 31 11:14:43 GMT 2013 , 29)
\end{verbatim}

\begin{itemize}
\item \textbf{Exercise 2.4: Using syscall probes}
\end{itemize}

List the available \texttt{syscall} provider probes. Experiment by adapting the \texttt{daterun.d} script for another program that produces output, such as \texttt{w}.

(Estimated completion time: 10 minutes)

\begin{itemize}
\item \textbf{Solution to Exercise 2.4}
\end{itemize}

\begin{verbatim}
# dtrace -l -P syscall
\end{verbatim}

\begin{center}
\begin{tabular}{lcccc}
\hline
ID & PROVIDER & MODULE & FUNCTION NAME & العربة
\hline
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 & \\
\ldots
598 & syscall & vmlinux & kcmp entry & \\
599 & syscall & vmlinux & kcmp return & \\
600 & syscall & vmlinux & finit_module entry & \\
601 & syscall & vmlinux & finit_module return & \\
602 & syscall & vmlinux & waitfd entry & \\
603 & syscall & vmlinux & waitfd return & \\
\hline
\end{tabular}
\end{center}
Example 2.7 wrun.d: Modified version of daterun.d for the w command

```bash
#!/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
```

2.5 Timing Events on a System

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

The following D program (readtrace.d) displays the command name, process ID, and call duration in microseconds whenever a process invokes the `read()` system call.

Example 2.8 readtrace.d: Display time spent in read() calls

```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 */
}
```

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 from the value of \texttt{timestamp} when the call returns. The units of \texttt{timestamp} are nanoseconds so we divide by 1000 to obtain a value in microseconds.

The following is example output from running this program.

```
# dtrace -q -s readtrace.d
nome-terminal (pid=2774) spent 27 microseconds in read()
gnome-terminal (pid=2774) spent 16 microseconds in read()
hald-addon-inpu (pid=1662) spent 26 microseconds in read()
hald-addon-inpu (pid=1662) spent 17 microseconds in read()
Xorg (pid=2046) spent 18 microseconds in read()
...^C
```

Exercise 2.5: Timing system calls

Add a predicate to the \texttt{entry} probe in \texttt{readtrace.d} so that \texttt{dtrace} displays results for Mozilla Firefox selected by the name of its executable (\texttt{firefox}).

Using the \texttt{probefunc} variable and the \texttt{syscall:::entry} and \texttt{syscall:::return} probes, create a D program, \texttt{calltrace.d}, that times all system calls for a named executable such as \texttt{firefox}.

(Estimated completion time: 10 minutes)

Solution to Exercise 2.5

Example 2.9 \texttt{readtrace1.d}: Modified version of \texttt{readtrace.d} that includes a predicate

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

syscall::read:entry
/execname == "firefox"/
{
  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 \texttt{/execname == "firefox"} tests whether the \texttt{firefox} program is running when the probe fires.

```
# chmod +x readtrace1.d
# dtrace -q -s readtrace1.d
firefox (pid=4047) spent 33 microseconds in read()
firefox (pid=4047) spent 32 microseconds in read()
firefox (pid=4047) spent 40 microseconds in read()
firefox (pid=4047) spent 23 microseconds in read()
firefox (pid=4047) spent 27 microseconds in read()
^C
```
Example 2.10 calltrace.d: Time all system calls for firefox

```d
/* calltrace.d -- Time all system calls for firefox */
syscall:::entry
/execname == "firefox"/
{
    self->t = timestamp; /* Initialize a thread-local variable */
}
syscall:::return
/self->t != 0/
{
    printf("%s (pid=%d) spent %d microseconds in %s()
", 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`.

```bash
# chmod +x calltrace.d
# dtrace -q -s calltrace.d
firefox (pid=4088) spent 42 microseconds in clock_gettime()
firefox (pid=4088) spent 27 microseconds in futex()
firefox (pid=4088) spent 88 microseconds in write()
firefox (pid=4088) spent 37 microseconds in clock_gettime()
firefox (pid=4088) spent 35 microseconds in gettimeofday()
```

2.6 Tracing Parent and Child Processes

When a process forks, it creates a child process that is effectively a copy of its parent process, but which has a different process ID. (Other differences are described on the `fork(2)` manual page.) The child process can either run independently of its parent process to perform some separate task, or it can execute a new program image that replaces the child’s program image while retaining the same process ID.

The next D program uses `proc` probes to trace `activity.d`, reports `fork()` and `exec()` activity on a system.

Example 2.11 activity.d: Record fork() and exec() activity

```d
@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 */
}
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 pointed to by the `args[0]` probe argument. For more information about the arguments to `proc` probes, see proc Provider in the Oracle Linux Dynamic Tracing 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 see output similar to the following, as you run different programs.

```
# dtrace -s activity.d
bash (3572) executed /bin/ls (4323) for 6422 microseconds
bash (3572) executed /usr/bin/w (4324) for 128960 microseconds
firefox (4325) executed /bin/basename (4326) for 8548 microseconds
firefox (4325) executed /bin/uname (4327) for 1999 microseconds
mozilla-plugin- (4328) executed /bin/uname (4329) for 2151 microseconds
mozilla-plugin- (4328) executed /usr/lib64/nspluginwrapper/plugin-config (4330)
    for 8182 microseconds
firefox (4325) executed /usr/lib64/xulrunner-1.9.2/mozilla-xremote-client (4331)
    for 40067 microseconds
firefox (4333) forked itself (as 4334) for 200 microseconds
firefox (4333) executed /bin/sed (4335) for 1070 microseconds
firefox (4336) forked itself (as 4337) for 229 microseconds
firefox (4336) executed /bin/sed (4338) for 1161 microseconds
...```

**Exercise 2.6: Using a predicate to control the execution of an action**

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

(Estimated completion time: 10 minutes)
Solution to Exercise 2.6

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 uses the following predicate instead:

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

The following example uses this form of the predicate:

**Example 2.12 activity1.d: Record fork() and exec() activity for a specified program**

```d
/* 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_execve_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);
}
```
You can now specify the name of the program to be traced as an argument to `dtrace`, for example:

```
# 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 14197 microseconds
bash (10359) forked itself (as 10366) for 41918 microseconds
bash (10370) executed /usr/bin/id (10371) for 11729 microseconds
```

Note that you need to escape the argument to protect the double quotes from the shell.

2.7 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. See Aggregations in the Oracle Linux Dynamic Tracing Guide for a description of aggregation functions.

DTrace indexes the results of an aggregation using a tuple expression similar to that 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.

For example, the following command counts the number of `write()` system calls 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, we specify the probe and the action on the `dtrace` command line.

DTrace prints out the result of the aggregation automatically. Alternatively, you can use the `printa()` function to format the result of the aggregation.

The next 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

20
Simple Data Aggregations

read() calls 1555

Exercise 2.7: 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 2.7

Example 2.13 `countcalls.d`: Count write, read, and open system calls over 100 seconds

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

```text
# 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
```

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

Example 2.14 `countsyscalls.d`: Count system calls invoked by a process

```bash
#!/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:

```bash
# chmod +x countsyscalls.d
# ./countsyscalls.d $(pgrep -u guest firefox)
```

```text
newuname 1
```
Simple Data Aggregations

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>getdents</td>
<td>2</td>
</tr>
<tr>
<td>getsockname</td>
<td>2</td>
</tr>
<tr>
<td>clone</td>
<td>3</td>
</tr>
<tr>
<td>close</td>
<td>3</td>
</tr>
<tr>
<td>sched_setscheduler</td>
<td>3</td>
</tr>
<tr>
<td>mmap</td>
<td>6</td>
</tr>
<tr>
<td>sched_get_priority_max</td>
<td>6</td>
</tr>
<tr>
<td>sched_get_priority_min</td>
<td>6</td>
</tr>
<tr>
<td>open</td>
<td>7</td>
</tr>
<tr>
<td>munmap</td>
<td>9</td>
</tr>
<tr>
<td>lseek</td>
<td>16</td>
</tr>
<tr>
<td>newfstat</td>
<td>31</td>
</tr>
<tr>
<td>access</td>
<td>45</td>
</tr>
<tr>
<td>write</td>
<td>49</td>
</tr>
<tr>
<td>fcntl</td>
<td>58</td>
</tr>
<tr>
<td>newstat</td>
<td>104</td>
</tr>
<tr>
<td>futex</td>
<td>1045</td>
</tr>
<tr>
<td>writev</td>
<td>3102</td>
</tr>
<tr>
<td>clock_gettime</td>
<td>4079</td>
</tr>
<tr>
<td>poll</td>
<td>7938</td>
</tr>
<tr>
<td>read</td>
<td>9746</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>10165</td>
</tr>
</tbody>
</table>

In this example, we use `pgrep` to determine the process ID of the `firefox` program that the user `guest` is running.

**Exercise 2.8: Tracing the processes that a user runs**

Create a program, `countprogs.d`, that counts and display the number of times that 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 2.8**

**Example 2.15 countprogs.d: Count programs invoked by a specified user**

```d
#!/usr/sbin/dtrace -qs
/* countprogs.d -- Count programs invoked by a specified user */

proc::do_execve_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. We use the `id -u user` command to find out the ID of the `guest` user account.

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>1</td>
</tr>
<tr>
<td>firefox</td>
<td>2</td>
</tr>
<tr>
<td>npviewer</td>
<td>4</td>
</tr>
<tr>
<td>bash</td>
<td>12</td>
</tr>
</tbody>
</table>
The following D program counts the number of times that a program reads from different files in ten seconds, and displays only the top five results.

Example 2.16 fdscount.d: Count the number of times that a program reads from different files

```d
/* fdscount.d -- Count the number of times that a program reads from different files */
tick-10s {
    exit(0);
}
syscall::read:entry /execname==ENAME/ {
    @[fds[arg0].fi_pathname] = count();
}
END {
    trunc(@,5);
}
```

We 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 indexed in `fds[]` by `arg0` contains the full pathname of the file.

See `fileinfo_t` in the *Oracle Linux Dynamic Tracing Guide* for more information about the members of the `fileinfo_t` structure.

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

Before running the program, we load the `profile`, `sdt` and `systrace` kernel modules so that DTrace has access to the `profile:::tick-10s` probe, the `fds[]` built-in array, and the `syscall::read:entry` probe. We specify a C preprocessor directive to `dtrace` that sets the value of the `ENAME` variable to "thunderbird", which is the name of the Mozilla Thunderbird executable. We also need to use additional single quotes to escape the string quotes.

```
# modprobe profile
# modprobe sdt
# modprobe systrace
# dtrace -C -D ENAME="""thunderbird"""" -qs fdscount.d

/home/guest/.thunderbird/default/panacea.dat 105
/home/guest/4846/maps 120
/home/guest/.thunderbird/default/ImapMail/mydom.com/Sent-1.msf 281
pipe:[57103] 531
socket:[57480] 22084
```

The `/proc/pid/maps` entry in the output is a file in the `procfs` file system that contains information about the process's mapped memory regions and their permissions. The `pipe:[inode]` and `socket:[inode]` entries refer to inodes in the `pipefs` and `socketfs` file systems.
Exercise 2.9: 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 Dynamic Tracing 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 2.9

Example 2.17 cswpercpu.d: Print number of context switches per CPU once per second

```d
#!/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%-25s ", walltimestamp);
/* Print the aggregated counts for each CPU and the total for all CPUs */
printa("%-5s %@15d
                     ", @n);
/* Reset the aggregation */
clear(@n);
}
# chmod +x cswpercpu.d
#.cswpercpu.d
```

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>CPU</th>
<th>Ncsw</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013 Nov 6 20:47:26</td>
<td>1</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>272</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>775</td>
</tr>
<tr>
<td>2013 Nov 6 20:47:27</td>
<td>1</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>417</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>1493</td>
</tr>
<tr>
<td>2013 Nov 6 20:47:28</td>
<td>3</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>178</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>446</td>
</tr>
</tbody>
</table>

You might like to experiment with aggregating the total time spent context switching and the average time per context switch. You can do this 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`. You can use the `sum()` and `avg()` aggregation functions respectively.
2.8 More Complex Data Aggregations

You can use the `lquantize()` and `quantize()` functions to display linear and power-of-two frequency distributions of data. See Aggregations in the Oracle Linux Dynamic Tracing Guide for a description of aggregation functions.

In the following example, we display the distribution of the sizes specified to `arg2` of `read()` calls that were invoked by all instances of `firefox` that are running.

```bash
# dtrace -n 'syscall::read:entry /execname=="firefox"/@dist["firefox"]=quantize(arg2);}'
```

```
firefox
value  -------- Distribution -------- count
0 |             0
1 |@             566
2 |             0
4 |             0
8 |             7
16 |            4
32 |             0
64 |             0
128|             8
256|@            436
512|             8
1024|@@          959
2048|@          230
4096|@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @13785
8192|             3
16384|            4
32768|             0
65536|             0
131072|           73
262144|           0
```

If the program is simple as this one, it is often convenient to run it from the command line.

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

**Example 2.18 diskact.d: Display the distribution of I/O throughput for block devices**

```bash
#pragma D option quiet

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

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

io:::done
/start[args[0]->b_e dev, args[0]->b_blkno]/
{
    /*
    We want to get an idea of our throughput to this device in KB/sec
    but we have values measured in bytes and nanoseconds.
    We want to calculate:
    
    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 we restate the fraction as:

\[
\frac{\text{bytes}}{\text{nanoseconds}} = \frac{1000000000 \times 976562}{1024}\]

This is easy to calculate using integer arithmetic.

```c
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;
```

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

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

See `Output Formatting` in the Oracle Linux Dynamic Tracing Guide for a description of the `printa()` function.

After running the program for about a minute, we type `Ctrl-C` to display the results:
Exercise 2.10: 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:::start`, assign the value of `args[0]->b_flags & B_READ ? "READ" : "WRITE"` to the variable `iodir`.
- In the actions for `io:::start` and `io:::start`, add `iodir` as a key to the `start[]` associative array.
- In the action for `io:::start`, 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 2.10

Example 2.19 `rwdiskact.d`: Modified version of `diskact.d` that displays separate results for read and write I/O

```d
#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_eudev, args[0]->b_blkno, iodir] = timestamp;
}

io:::done
{
  iodir = args[0]->b_flags & B_READ ? "READ" : "WRITE";
  this->elapsed = timestamp - start[args[0]->b_eudev, 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_eudev, args[0]->b_blkno, iodir] = 0;
}

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

Adding the `iodir` variable to the tuple in the aggregation variable allows DTrace to display separate aggregations for read and write I/O operations.
More Complex Data Aggregations

The next 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. We use the `lquantize()` aggregation function to display the results linearly in tenths of the total distance across the device.

Example 2.20 fsact: Display cumulative read and write activity across a file system device

```bash
#!/bin/bash
#
# fsact -- Display cumulative read and write activity across a file system device
#
# Usage: fsact [<filesystem>]
#
# Load the required DTrace modules
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);
}
```

```bash
# dtrace -s rwdiskact.d

```

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

```

```bash
dm-3 (/dev/dm-3) WRITE

```

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>512</td>
<td>@@@</td>
<td>6</td>
</tr>
<tr>
<td>1024</td>
<td>@@@@@@@@@@@@@@@@</td>
<td>33</td>
</tr>
<tr>
<td>2048</td>
<td>@@@@@@@@@@@@@@@@@@@</td>
<td>41</td>
</tr>
<tr>
<td>4096</td>
<td>@@@@@</td>
<td>8</td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

```bash
```
We embed the D program in a shell script so that we can set up the parameters that we need: the major and minor numbers of the underlying device and the total size of the file system in file system blocks. We then write the values of these parameters directly into the here-script.

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/
```

The following is sample output from running `fsact` after making the script executable.

```
# chmod +x fsact
# ./fsact
2013 Nov 10 17:14:35
WRITE

value   Distribution   count
< 0     0
0       3
1       3
2       1
3       0
4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 442
5 @@@@@@@                          16
6 @                                 1
7 @                                  0
8 @                                  1
9 @                                  0

2013 Nov 10 17:14:45
WRITE

value   Distribution   count
< 0     0
0       118
1       273
2       151
3       48
4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 874
5 @@@@@@@                          231
6 @                                 0
7 @                                  0
8 @                                  0
9 @                                  44
>> 10                               0
```
2.9 Displaying System Call Errors

The following D program, `errno.d`, displays the value of `errno` and the file name if an error occurs when using `open()` to open a file:

Example 2.21 `errno.d`: Display `errno` and the file name for failed `open()` calls

```d
#!/usr/sbin/dtrace -qs
/* errno.d -- Display errno and the file name for failed open() calls */
system::open:entry
{
    self->filename = copyinstr(arg0);
}
system::open:return
/arg0 < 0/
{
    printf("errno = %-2d   file = %s
", 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. We use a predicate to test the value of `arg0`. Alternatively, we could test whether the value of `errno` is greater than zero.

After saving this script to a file, and making the file executable, you can run it to display information about any failures of the `open()` system call that occur on the system:

```
# ./errno.d
errno = 2    file = /var/ld/ld.config
errno = 4    file = /images/UnorderedList16.gif
^C
```
Exercise 2.11: 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/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 2.11

Example 2.22 displayerrno.d: Modified version of errno.d that displays error names

#!/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[ENOIO] = "ENOIO"; /* 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[EINVAL] = "EINVAL"; /* Invalid argument */
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[ENOEXEC] = "ENOEXEC"; /* No such device */
error[ENOTDIR] = "ENOTDIR"; /* Not a directory */
error[EINVAL] = "EINVAL"; /* Invalid argument */
error[ENOMEM] = "ENOMEM"; /* Out of memory */
error[EPIPE] = "EPIPE"; /* Broken pipe */
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.
Chapter 3 Tracing User-Space Applications

This chapter provides examples of D programs that you can use to investigate what is happening in user-space programs.

3.1 Preparation: Installing DTrace-Enabled Applications

DTrace-enabled versions of user-space applications are made available via the playground repository of Oracle Public Yum (http://public-yum.oracle.com/repo/OracleLinux/OL6/playground/latest/x86_64/). These applications have been instrumented to contain statically defined DTrace probes. You can find details about the probes for PHP at http://php.net/manual/features.dtrace.php.

Caution

The packages that are provided in the playground repository are intended for experimentation only and you should not use them with production systems. Oracle does not offer support for these packages and does not accept any liability for their use.

To enable access to the playground channel on Oracle Public Yum, create an entry such as the following in /etc/yum.conf or in a repository file in the /etc/yum.repos.d directory:

```bash
[ol6_playground_latest]
name=Latest mainline stable kernel for Oracle Linux 6 ($basearch) - Unsupported
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
gpgcheck=1
enabled=1
```

To enable the channel, set the value of the enabled parameter for the channel to 1. You can then use `yum` to install the packages, for example:

```
# yum install php55
```

3.2 Tracing a User-Space Application

The following PHP program, named `func.php`, runs forever cycling through the `heads()` and `tails()` functions to various depths of recursion, depending on random number generation. This example is a PHP program that you run from the command line, but DTrace can just as easily trace PHP web applications. You can think of the infinite loop that calls the function `initialize()` as simulating multiple requests to a web-based application.

Example 3.1 `func.php`: Infinitely looping, recursive test script

```php
<?php
/* func.php -- Infinitely looping, recursive test script */

function heads(&$d) {
    $d++;
    $i = @(1/(abs($d)-3));
    switch (mt_rand(0,2)) {
    case 0:
        heads($d); break;
    case 1:
```
To allow user programs to be traced, enable the `fasttrap` provider in DTrace:

```
# modprobe fasttrap
```

Set `func.php` running in a window:

```
# php ./func.php
```

We can now start to trace the operation of this program from another window.

```
# dtrace -q -n 'function-entry {printf("Called %s() in %s at line %d\n", \n    copyinstr(arg0), copyinstr(arg1), arg2)}'
Called initialize() in /root/errfunc.php at line 42
Called heads() in /root/errfunc.php at line 3
Called tails() in /root/errfunc.php at line 22
Called initialize() in /root/errfunc.php at line 42
Called heads() in /root/errfunc.php at line 3
Called heads() in /root/errfunc.php at line 3
Called heads() in /root/errfunc.php at line 3
Called heads() in /root/errfunc.php at line 3
Called initialize() in /root/errfunc.php at line 42
Called heads() in /root/errfunc.php at line 3
Called heads() in /root/errfunc.php at line 3
Called tails() in /root/errfunc.php at line 22
```
Note

When tracing user-space programs, we must always use the argN arguments (whose type is int64_t) and, if necessary, cast the type as appropriate to the real type of the probe argument. The args[N] variables, to which DTrace automatically assigns the type that is defined for the probe, are not available. You must use the appropriate type converter such as stringof() or a cast such as (string).

We use copyinstr() in the example because arg0 and arg1 both refer to the addresses of character arrays in the address space of a user process. copyinstr() copies the data from user space to a DTrace buffer in kernel space and converts it to type string.

3.3 Using Aggregations with User-Space Applications

You can use aggregations to collect statistical data for user-space applications. For example, the following D program, aggfunc.d, aggregates the number of function calls across all PHP instances as well as by individual instances:

Example 3.2 aggfunc.d: Aggregate counts for a PHP program

```
#!/usr/sbin/dtrace -Zqs
/* aggfunc.d -- Aggregate counts for a PHP program */
php*:::function-entry
{
    @bypid[pid] = count();
    @byfunc[copyinstr(arg0)] = count();
    @bypidandfunc[pid,copyinstr(arg0)] = count();
}
END
{
    printf("Counts by pid\n");
    printf(" %-40s @d\n", @bypid);
    printf(" %-40s @d\n", @byfunc);
    printf(" %-d @d and function\n", @bypidandfunc);
}
```

The -Z option allows you to start dtrace before any instances of php are running and no matching probes are available.

Run aggfunc.d in one window and start several instances of php func.php from other windows. Allow aggfunc.d to run for several seconds before typing Ctrl-C. The program returns output such as the following:

```
# ./aggfunc.d
^C
Counts by pid
5646 15
5287 28
```
3.4 Tracing the Flow of Execution

The next D program, flow.d, traces the flow of execution through func.php.

Example 3.3 flow.d: Trace the flow of execution in func.php

```d
/* flow.d -- Trace the flow of execution in func.php */

self int indent;

php$target:::function-entry
/copyinstr(arg0) == "initialize"/
{
    self->follow = 1;
    self->indent += 2;
}

php$target:::function-entry
/self->follow/
{
    printf("%*s ", self->indent, "->");
    printf("%s() entry\n", copyinstr(arg0));
    self->indent += 2;
}

php$target:::function-return
/self->follow/
{
    self->indent -= 2;
    printf("%*s ", self->indent, "<-" );
    printf("%s() return\n", copyinstr(arg0));
}

php$target:::function-return
/copyinstr(arg0) == "initialize"/
{
    self->follow = 0;
}
```

The program uses the PHP `function-entry` and `function-return` probes to trace the execution.

The probe provider is specified as `php$target`. As `$target` evaluates to the process ID of the program that is being traced, we start the PHP program by specifying it as the argument to the `−c` option to `dtrace`:

```
# dtrace -c 'php ./func.php' -q -s ./flow.d
```
Exercise 3.1: Tracing system call invocation from user programs

Add probe clauses to flow.d that use the syscall:::entry and syscall:::return probes to show where PHP makes system calls and the return from these calls. Use the symbols => and <= to denote the call and its return. Prepend the output lines with the name of the probe module.

(Estimated completion time: 20 minutes)
Solution to Exercise 3.1

Example 3.4 flow1.d: Modified version of flow.d that also displays system calls

/* flow1.d -- Modified version of flow.d that also displays system calls */

self int indent;

php$target:::function-entry
copyinstr(arg0) == "initialize"/
{
  self->follow = 1;
  self->indent += 2;
}

php$target:::function-entry
/self->follow/
{
  printf("%-12s", probemod);
  printf("%*s ", self->indent, "->");
  printf("%s() %*s
", copyinstr(arg0), 48-self->indent, "");
  self->indent += 2;
}

php$target:::function-return
/self->follow/
{
  printf("%-12s", probemod);
  self->indent -= 2;
  printf("%*s ", self->indent, "<-");  
  printf("%s() %*s
", copyinstr(arg0), 48-self->indent, "");
}

syscall:::entry
/self->follow/
{
  printf("%-12s", probemod);
  printf("%*s ", self->indent, "->");
  printf("%s() %*s
", probefunc, 48-self->indent, "");
  self->indent += 2;
}

syscall:::return
/self->follow/
{
  printf("%-12s", probemod);
  printf("%*s ", self->indent, "<-");  
  printf("%s() %*s
", probefunc, 48-self->indent, "");
}

php$target:::function-return
/copyinstr(arg0) == "initialize"/
{
  self->follow = 0;
}
3.5 Detecting PHP Errors

The program `func.php` uses the `@` operator to suppress some PHP errors. Instead of using the `scream` extension to track down application failures, which would require you to rebuild PHP and restart the web servers, you can use the `error` probe to report all PHP errors, including any suppressed errors. The following D program, `detphperr.d`, shows an example of how to use the `error` probe.

**Example 3.5 detphperr.d: Detect errors in PHP programs**

```d
#!/usr/sbin/dtrace -qs
/* detphperr.d -- Detect errors in PHP programs */

tphperr.err
{
  printf("PHP error\n");
  printf("  error message             %s\n", copyinstr(arg0));
  printf("  request file              %s\n", copyinstr(arg1));
  printf("  line number               %d\n", (int)arg2);
}
```

detphperr.d outputs a summary of any error that occurs in the PHP programs that are running on the system, for example:

```d
# .detphperr.d
PHP error
  error message             fopen(/tmp/foo.bar): failed to open stream: No such file or directory
  request file              /var/www/html/ex2.php
  line number               76

PHP error
  error message             Call to undefined function foo()
  request file              /var/www/html/ex3.php
  line number               69

PHP error
  error message             Division by zero
  request file              /var/www/html/ex1.php
  line number               66
...
```
You can use a program such as `detphperr.d` to report errors that might indicate incorrectly queries or attempted SQL injection attacks, for example:

```
# ./detphperr.d
...
```

**Exercise 3.2: Displaying more information about PHP errors**

Modify `detphperr.d` to display the date and time at which the error occurred by using the `walltimestamp` variable with the `%Y` conversion format, and record the function in which the probe fired by using the `probefunc` variable.

Use `detphperr.d` to detect errors as they occur in the running PHP program `func.php`.

(Estimated completion time: 20 minutes)

**Solution to Exercise 3.2**

**Example 3.6 detphperr1.d: Modified version of detphperr.d that includes the function name and a timestamp**

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

/* detphperr1.d -- Modified version of detphperr.d that includes
the function name and a timestamp */

php*:::error {
    printf("PHP error\n");
    printf(" timestamp                 %Y\n", walltimestamp);
    printf(" error message             %s\n", copyinstr(arg0));
    printf(" request file              %s\n", copyinstr(arg1));
    printf(" function                  %s()\n", probefunc);
    printf(" line number               %d\n", (int)arg2);
}
```

```
# chmod +x detphperr1.d
# ./detphperr1.d
```

```
timestamp                  2013 Oct 31 15:31:50
errormsg                  Division by zero
request_file              /var/www/html/func.php
function                  tails()
lineno                    25
^C
```

### 3.6 Using a Speculation for Error Analysis

You can use the speculative tracing facility in DTrace to trace data, and then later decide whether to commit the data to a tracing buffer or to discard the data. When a probe fires, predicates allow you to filter out uninteresting events if you know whether an event is of interest at that time. However, in some situations, you might not know whether a probe event is of interest until some time after the probe fires.
For example, if a program is occasionally failing with an error, you might want to examine the code path leading to the error condition. You can write trace data at one or more probe locations to speculative buffers, and then choose which data to commit to the principal buffer at another probe location. As a result, your trace data contains only the output of interest, no post-processing is required, and the DTrace overhead is minimized.

The `speculation()` function creates a speculative buffer and returns a speculation identifier, which you use in subsequent calls to the `speculate()` function. Typically, you assign the speculation identifier to a thread-local variable, and then use that variable as a predicate to other probes and as an argument to `speculate()`.

You call the `speculate()` function before performing any data-recording actions. DTrace directs all the data that you subsequently record in the clause to a speculative buffer. You can create only one speculation per clause.

If a speculative buffer contains data that you want to retain, use the `commit()` function to copy its contents to the principal buffer. If the speculative buffer contains data that you want to delete, use the `discard()` function.

For more information, see Speculative Tracing in the Oracle Linux Dynamic Tracing Guide.

In the following D program, `errortrace.d`, we use a speculation to display the function calls below `initialize()` in `func.php` if an error occurs but not otherwise.

**Example 3.7 errortrace.d: Print complete trace after one or more errors occur**

```d
/* errortrace.d -- Print complete trace after one or more errors occur */

self int indent;

BEGIN {
    /* Read the name of the function from the command line */
    topfunc = $1;
}

php$target:::function-entry
/copyinstr(arg0) == topfunc/ {
    /* Initialize and start speculation */
    self->errflag = 0;
    self->spec = speculation();
    self->indent += 2;
    self->func = topfunc;
    speculate(self->spec);
}

php$target:::function-entry
/self->spec/ {
    /* Add function call to speculation */
    speculate(self->spec);
    self->func = copyinstr(arg0);
    printf("%s %s()\n", self->indent, "->", copyinstr(arg0));
    self->indent += 2;
}

php$target:::function-return
/self->spec/ {
    /* Add function return to speculation */
    self->indent -= 2;
    speculate(self->spec);
    printf("%s %s()\n", self->indent, "<-", copyinstr(arg0));
}

php$target:::function-return
```
/copyinstr(arg0) == topfunc && self->errflag == 1/  
{ /* Commit speculation if error flag is set */
  commit(self->spec);
  self->spec = 0;
  self->indent = 0;
 }

php$target:::function-return
/copyinstr(arg0) == topfunc && self->errflag == 0/  
{ /* Discard speculation if error flag is not set */
  discard(self->spec);
  self->spec = 0;
  self->indent = 0;
 }

php$target:::error
/\self->spec/  
{  
  speculate(self->spec);
  self->errflag = 1; /* Set error flag if an error occurs */
  printf("%*s PHP error: \"%s\" in line %d at line %d of \"%s\", self->func, (int)arg2,
         basename(copyinstr(arg1)));
  }

Running the program for several seconds produces output such as the following:

```
# dtrace -c 'php func.php' -s 'errortrace.d' -x nspec=10 "initialize"
-> initialize()
  -> heads()
    -> heads()
      -> heads()
        PHP error: "Division by zero" in heads() at line 6 of func.php
      -> tails()
        PHP error: "Division by zero" in heads() at line 6 of func.php
    -> heads()
      <- heads()
      <- heads()
    <- tails()
    <- heads()
    <- heads()
  <- heads()
  <- initialize()
^C
```

We pass the name of the top-level function `initialize` as an argument to `dtrace`, which requires us to protect the string quotes from being removed by the shell. For more information, see Macro Arguments in the Oracle Linux Dynamic Tracing Guide.

The output shows that two errors occurred before the program returned to the `initialize()` function.

The `-x nspec=10` option creates more speculative buffers so that DTrace does not report speculation errors because of a shortage of available buffers.

The error occurs in `heads()` or `tails()` when PHP evaluates $i = @(1/(abs($d)-3)); and the value of $d is 3.

**Exercise 3.3: Committing a speculation**

Instead of committing the speculation when the PHP program returns to `initialize()`, create a version of `errortrace.d` that commits the speculation as soon as the error occurs.

(Estimated completion time: 30 minutes)
Solution to Exercise 3.3

Example 3.8 errortrace1.d: Print a function call trace as soon as an error occurs

```d
/* errortrace1.d -- Print a function call trace as soon as an error occurs */

self int indent;
BEGIN
{
  topfunc = $1;
}
php$target:::function-entry
/copyinstr(arg0) == topfunc/
{
  self->spec = speculation();
  self->indent += 2;
  self->func = topfunc;
  speculate(self->spec);
}
php$target:::function-entry
/self->spec/
{
  speculate(self->spec);
  self->func = copyinstr(arg0);
  printf("%s %s()\n", self->indent, "->", copyinstr(arg0));
  self->indent += 2;
}
php$target:::function-return
/self->spec/
{
  speculate(self->spec);
  self->indent -= 2;
  printf("%s %s()\n", self->indent, "<-", copyinstr(arg0));
}
php$target:::function-return
/copyinstr(arg0) == topfunc/
{
  discard(self->spec);
  self->spec = 0;
  self->indent = 0;
}
php$target:::error
/self->spec/
{
  /* Commit the speculation when the error occurs */
  speculate(self->spec);
  printf("%s PHP error: \"%s\" at line %d of %s\n", self->func, (int)arg2,
    basename(copyinstr(arg1)));
  commit(self->spec);
}
```
Using a Speculation for Error Analysis

# dtrace -c 'php ./func.php' -q -s errortrace1.d -x nspec=10 "initialize"
-> initialize()
  -> heads()
    -> heads()
      PHP error: "Division by zero" in heads() at line 6 of func.php
  -> initialize()
    -> tails()
      -> tails()
        PHP error: "Division by zero" in tails() at line 25 of func.php
^C
Chapter 4 Going Further with DTrace

For more information about using DTrace on Oracle Linux, see the *Oracle Linux Dynamic Tracing Guide*.


Many examples of D programs are also available from Brendan Gregg's *DTrace Tools* website at [http://www.brendangregg.com/dtrace.html](http://www.brendangregg.com/dtrace.html).

You can discuss DTrace at the *DTrace Forum* on OTN.