Abstract

This manual provides information about the Dynamic Tracing (DTrace) facility (version 0.4) for Oracle Linux engineered by Oracle.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>ix</td>
</tr>
<tr>
<td>1 About DTrace</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Getting Started</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Providers and Probes</td>
<td>2</td>
</tr>
<tr>
<td>2 The D Programming Language</td>
<td>5</td>
</tr>
<tr>
<td>2.1 D Program Structure</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Probe Clauses and Declarations</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Probe Descriptions</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Predicates</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4 Actions</td>
<td>8</td>
</tr>
<tr>
<td>2.1.5 Order of Execution</td>
<td>8</td>
</tr>
<tr>
<td>2.1.6 Use of the C Preprocessor</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Compilation and Instrumentation</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Variables and Arithmetic Expressions</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Predicates</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Output Formatting</td>
<td>15</td>
</tr>
<tr>
<td>2.6 Arrays</td>
<td>17</td>
</tr>
<tr>
<td>2.7 External Symbols and Types</td>
<td>18</td>
</tr>
<tr>
<td>2.8 Types, Operators, and Expressions</td>
<td>19</td>
</tr>
<tr>
<td>2.8.1 Identifier Names and Keywords</td>
<td>19</td>
</tr>
<tr>
<td>2.8.2 Data Types and Sizes</td>
<td>20</td>
</tr>
<tr>
<td>2.8.3 Constants</td>
<td>21</td>
</tr>
<tr>
<td>2.8.4 Arithmetic Operators</td>
<td>22</td>
</tr>
<tr>
<td>2.8.5 Relational Operators</td>
<td>23</td>
</tr>
<tr>
<td>2.8.6 Logical Operators</td>
<td>23</td>
</tr>
<tr>
<td>2.8.7 Bitwise Operators</td>
<td>24</td>
</tr>
<tr>
<td>2.8.8 Assignment Operators</td>
<td>24</td>
</tr>
<tr>
<td>2.8.9 Increment and Decrement Operators</td>
<td>25</td>
</tr>
<tr>
<td>2.8.10 Conditional Expressions</td>
<td>26</td>
</tr>
<tr>
<td>2.8.11 Type Conversions</td>
<td>26</td>
</tr>
<tr>
<td>2.8.12 Precedence</td>
<td>27</td>
</tr>
<tr>
<td>2.9 Variables</td>
<td>28</td>
</tr>
<tr>
<td>2.9.1 Scalar Variables</td>
<td>28</td>
</tr>
<tr>
<td>2.9.2 Associative Arrays</td>
<td>29</td>
</tr>
<tr>
<td>2.9.3 Thread-Local Variables</td>
<td>30</td>
</tr>
<tr>
<td>2.9.4 Clause-Local Variables</td>
<td>32</td>
</tr>
<tr>
<td>2.9.5 Built-in Variables</td>
<td>34</td>
</tr>
<tr>
<td>2.9.6 External Variables</td>
<td>35</td>
</tr>
<tr>
<td>2.10 Pointers and Arrays</td>
<td>36</td>
</tr>
<tr>
<td>2.10.1 Pointers and Addresses</td>
<td>36</td>
</tr>
<tr>
<td>2.10.2 Pointer Safety</td>
<td>37</td>
</tr>
<tr>
<td>2.10.3 Array Declarations and Storage</td>
<td>38</td>
</tr>
<tr>
<td>2.10.4 Pointer and Array Relationship</td>
<td>39</td>
</tr>
<tr>
<td>2.10.5 Pointer Arithmetic</td>
<td>39</td>
</tr>
<tr>
<td>2.10.6 Generic Pointers</td>
<td>40</td>
</tr>
<tr>
<td>2.10.7 Multi-Dimensional Arrays</td>
<td>40</td>
</tr>
<tr>
<td>2.10.8 Pointers to DTrace Objects</td>
<td>41</td>
</tr>
<tr>
<td>2.10.9 Pointers and Address Spaces</td>
<td>41</td>
</tr>
<tr>
<td>2.11 Strings</td>
<td>42</td>
</tr>
<tr>
<td>2.11.1 String Representation</td>
<td>42</td>
</tr>
<tr>
<td>2.11.2 String Constants</td>
<td>42</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.11.3</td>
<td>String Assignment</td>
</tr>
<tr>
<td>2.11.4</td>
<td>String Conversion</td>
</tr>
<tr>
<td>2.11.5</td>
<td>String Comparison</td>
</tr>
<tr>
<td>2.12</td>
<td>Structs and Unions</td>
</tr>
<tr>
<td>2.12.1</td>
<td>Structs</td>
</tr>
<tr>
<td>2.12.2</td>
<td>Pointers to Structs</td>
</tr>
<tr>
<td>2.12.3</td>
<td>Unions</td>
</tr>
<tr>
<td>2.12.4</td>
<td>Member Sizes and Offsets</td>
</tr>
<tr>
<td>2.12.5</td>
<td>Bit-Fields</td>
</tr>
<tr>
<td>2.13</td>
<td>Type and Constant Definitions</td>
</tr>
<tr>
<td>2.13.1</td>
<td>typedefs</td>
</tr>
<tr>
<td>2.13.2</td>
<td>Enumerations</td>
</tr>
<tr>
<td>2.13.3</td>
<td>Inlines</td>
</tr>
<tr>
<td>2.13.4</td>
<td>Type Namespaces</td>
</tr>
<tr>
<td>3</td>
<td>Aggregations</td>
</tr>
<tr>
<td>3.1</td>
<td>Aggregating Functions</td>
</tr>
<tr>
<td>3.2</td>
<td>About Aggregations</td>
</tr>
<tr>
<td>3.3</td>
<td>Printing Aggregations</td>
</tr>
<tr>
<td>3.4</td>
<td>Data Normalization</td>
</tr>
<tr>
<td>3.5</td>
<td>Clearing Aggregations</td>
</tr>
<tr>
<td>3.6</td>
<td>Truncating Aggregations</td>
</tr>
<tr>
<td>3.7</td>
<td>Minimizing Drops</td>
</tr>
<tr>
<td>4</td>
<td>Actions and Subroutines</td>
</tr>
<tr>
<td>4.1</td>
<td>Actions</td>
</tr>
<tr>
<td>4.2</td>
<td>Default Action</td>
</tr>
<tr>
<td>4.3</td>
<td>Data Recording Actions</td>
</tr>
<tr>
<td>4.3.1</td>
<td>freopen</td>
</tr>
<tr>
<td>4.3.2</td>
<td>truncate</td>
</tr>
<tr>
<td>4.3.3</td>
<td>func</td>
</tr>
<tr>
<td>4.3.4</td>
<td>mod</td>
</tr>
<tr>
<td>4.3.5</td>
<td>printa</td>
</tr>
<tr>
<td>4.3.6</td>
<td>printf</td>
</tr>
<tr>
<td>4.3.7</td>
<td>stack</td>
</tr>
<tr>
<td>4.3.8</td>
<td>sym</td>
</tr>
<tr>
<td>4.3.9</td>
<td>trace</td>
</tr>
<tr>
<td>4.3.10</td>
<td>tracemem</td>
</tr>
<tr>
<td>4.3.11</td>
<td>ustack</td>
</tr>
<tr>
<td>4.3.12</td>
<td>uaddr</td>
</tr>
<tr>
<td>4.3.13</td>
<td>usym</td>
</tr>
<tr>
<td>4.4</td>
<td>Destructive Actions</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Process Destructive Actions</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Kernel Destructive Actions</td>
</tr>
<tr>
<td>4.5</td>
<td>Special Actions</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Speculative Actions</td>
</tr>
<tr>
<td>4.5.2</td>
<td>exit</td>
</tr>
<tr>
<td>4.5.3</td>
<td>setopt</td>
</tr>
<tr>
<td>4.6</td>
<td>Subroutines</td>
</tr>
<tr>
<td>4.6.1</td>
<td>alloca</td>
</tr>
<tr>
<td>4.6.2</td>
<td>basename</td>
</tr>
<tr>
<td>4.6.3</td>
<td>bcopy</td>
</tr>
<tr>
<td>4.6.4</td>
<td>cleanpath</td>
</tr>
<tr>
<td>4.6.5</td>
<td>copyin</td>
</tr>
<tr>
<td>4.6.6</td>
<td>copyinstr</td>
</tr>
<tr>
<td>4.6.7</td>
<td>copyinto</td>
</tr>
</tbody>
</table>
7.2 Creating a Speculation .......................................................... 95
7.3 Using a Speculation ............................................................ 96
7.4 Committing a Speculation .................................................... 96
7.5 Discarding a Speculation ..................................................... 97
7.6 Speculation Example .......................................................... 97
7.7 Speculation Options and Tuning .......................................... 98

8 The dtrace Utility ..................................................................... 99
8.1 Description ........................................................................... 99
8.2 Options ............................................................................... 99
8.3 Operands ............................................................................ 103
8.4 Exit Status .......................................................................... 103

9 Scripting ................................................................................. 105
9.1 Interpreter Files ................................................................... 105
9.2 Macro Variables .................................................................. 106
9.3 Macro Arguments ................................................................ 107
9.4 Target Process ID .............................................................. 108

10 Options and Tunables ........................................................... 111
10.1 Consumer Options ............................................................ 111
10.2 Modifying Options ............................................................ 115

11 Providers .............................................................................. 117
11.1 dtrace Provider ................................................................. 118
11.1.1 BEGIN Probe .................................................................. 118
11.1.2 END Probe .................................................................... 118
11.1.3 ERROR Probe ............................................................... 119
11.1.4 Stability ....................................................................... 120
11.2 profile Provider ................................................................. 120
11.2.1 profile-n Probes ............................................................ 120
11.2.2 tick-n Probes ............................................................... 121
11.2.3 Probe Creation ............................................................ 121
11.2.4 Stability ..................................................................... 121
11.3 syscall Provider ............................................................... 122
11.3.1 Probes ......................................................................... 122
11.3.2 Arguments ................................................................... 124
11.3.3 Stability ..................................................................... 124
11.4 sdt provider ....................................................................... 124
11.4.1 Creating SDT Probes .................................................... 124
11.4.2 Stability ..................................................................... 125
11.5 proc Provider ................................................................. 126
11.5.1 Probes ......................................................................... 126
11.5.2 Arguments ................................................................... 127
11.5.3 lwpinfo_t .................................................................... 128
11.5.4 psinfo_t ................................................................. 129
11.5.5 Examples .................................................................... 129
11.5.6 Stability ..................................................................... 132
11.6 sched Provider ................................................................. 132
11.6.1 Probes ......................................................................... 132
11.6.2 Arguments ................................................................... 134
11.6.3 cpuinfo_t ................................................................. 134
11.6.4 Examples .................................................................... 134
11.6.5 Stability ..................................................................... 146
11.7 io Provider ....................................................................... 146
11.7.1 Probes ......................................................................... 146
11.7.2 Arguments ................................................................... 147
11.7.3 bufinfo_t .................................................................... 148
Preface

The Oracle Linux DTrace Guide describes how to use DTrace. It also describes the DTrace providers in detail. Most of the information present in this document is generic to all releases of the Oracle Linux 6 and Oracle Linux 7 operating system.

Note

This release of DTrace supports systems that use the x86_64 processor architecture. It does not support systems that use 32-bit x86 processors.

Audience

This document is intended for anyone who needs to understand the behavior of their system. DTrace allows you to explore your system to understand how it works, track down performance problems, or locate the cause of aberrant behavior.

DTrace allows Oracle Linux administrators to:

- Dynamically enable and manage hundreds of probes
- Dynamically associate logical predicates and actions with probes
- Dynamically manage trace buffers and buffer policies
- Display and examine trace data from the live system
- Implement custom scripts that use the DTrace facility
- Implement layered tools that use DTrace to retrieve trace data

This guide will teach you everything you need to know about using DTrace. Basic familiarity with a programming language such as C or a scripting language such as awk or perl will help you learn DTrace and the D programming language faster, but you need not be an expert in any of these areas.

Document Organization

The document is organized as follows:

- Chapter 1, About DTrace provides an overview of DTrace.
- Chapter 2, The D Programming Language explains the D programming language.
- Chapter 3, Aggregations explains how to aggregate the data provided by the probes.
- Chapter 4, Actions and Subroutines describes the actions and subroutines supported by DTrace.
- Chapter 5, Buffers and Buffering describes the data buffering and management service provided by DTrace.
- Chapter 5, Buffers and Buffering explains how to format the output of D programs.
- Chapter 7, Speculative Tracing, describes the speculative tracing facility provided by DTrace.
- Chapter 8, The dtrace Utility describes the options supported by the dtrace command.
- Chapter 9, Scripting explains how to create interpreter files by using D programs. The interpreter files are similar to shell scripts that you can install as reusable interactive DTrace tools.
• Chapter 10, *Options and Tunables* explains the options and tuning parameters supported by the `dtrace` command.

• Chapter 11, *Providers* describes the providers supported by DTrace.

• Chapter 12, *User Process Tracing* explains how to use DTrace to understand the behavior of user processes.

• Chapter 13, *Statically Defined Tracing of User Applications* explains how to develop customized static probes for tracing user-space applications.

• Chapter 14, *Statically Defined Tracing of Kernel Modules* explains how to insert static probes in kernel modules.

• Chapter 15, *Performance Considerations* explains the performance considerations that you need to understand when using DTrace.

• Chapter 16, *Stability* describes the concepts related to stability in the context of D programs.

• Chapter 17, *Translators* describes the translators supported in D programs.

• Chapter 18, *Versioning*, explains the concepts related to versioning in the context of DTrace.

**Related Books**

The following books are recommended and related to the tasks that you need to perform with 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><strong>Convention</strong></th>
<th><strong>Meaning</strong></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><strong>italic</strong></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 About DTrace

Table of Contents

1.1 Getting Started .................................................................................................................. 1
1.2 Providers and Probes ......................................................................................................... 2

DTrace is a comprehensive dynamic tracing facility that is built into Oracle Linux that can be used by administrators and developers on live production systems to examine the behavior of user programs and the operating system. DTrace enables you to explore your system to understand how it works, track down performance problems, or locate the cause of aberrant behavior. DTrace lets you create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions you can formulate using the DTrace D programming language.

For information about how to obtain, install, and use the DTrace software, see the Oracle Linux DTrace Tutorial.

1.1 Getting Started

DTrace helps you understand a software system by enabling you to dynamically modify the operating system kernel to record additional data that you specify at locations of interest, called probes. A probe is a location or activity to which DTrace can bind a request to perform a set of actions, like recording a stack trace, a timestamp, or the argument to a function. Probes are like programmable sensors scattered all over your Oracle Linux system in interesting places. If you want to figure out what is going on, you use DTrace to program the appropriate sensors to record the information that is of interest to you. Then, as each probe fires, DTrace gathers the data from your probes and reports it back to you. If you do not specify any actions for a probe, DTrace just takes note each time the probe fires.

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

```
dtrace -n BEGIN
```

DTrace tells you that a probe was enabled and you will see a line of output indicating the BEGIN probe fired. Once you see this output, dtrace remains paused waiting for other probes to fire. Since no other probes are enabled and BEGIN only fires once, press Ctrl-C in your shell to exit dtrace and return to your shell prompt:

```
dtrace -n BEGIN
```

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

DTrace requests can be constructed using arbitrary numbers of probes and actions. Let us create a simple request using two probes by adding the END probe to the previous example command. The END probe fires once when tracing is completed. Type the following command, and then again press Ctrl-C in your shell after you see the line of output for the BEGIN probe:
Pressing Ctrl-C to exit dtrace triggers the END probe. dtrace reports this probe firing before exiting. In addition to constructing DTrace experiments on the command line, you can also write them in text files using the D programming language. In a text editor, create a new file called hello.d and type in your first D program:

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

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

```
# dtrace -s hello.d
```

Dtrace printed the same output as before followed by the text "hello, world". Unlike the previous example, you did not have to wait and press Ctrl-C. These changes were the result of the actions you specified for your BEGIN probe in hello.d. Let us explore the structure of your D program in more detail in order to understand what happened.

Each D program consists of a series of clauses, each clause describing one or more probes to enable, and an optional set of actions to perform when the probe fires. The actions are listed as a series of statements enclosed in braces {} following the probe name. Each statement ends with a semicolon (;). Your first statement uses the function trace() to indicate that DTrace should record the specified argument, the string "hello, world", when the BEGIN probe fires, and then print it out. The second statement uses the function exit() to indicate that DTrace should cease tracing and exit the dtrace command. DTrace provides a set of useful functions like trace() and exit() for you to call in your D programs. To call a function, you specify its name followed by a parenthesized list of arguments. The complete set of D functions is described in Chapter 4, Actions and Subroutines.

By now, if you are familiar with the C programming language, you have probably realized from the name and our examples that DTrace's D programming language is very similar to C. Indeed, D is derived from a large subset of C combined with a special set of functions and variables to help make tracing easy. You will learn more about these features in subsequent chapters. If you have written a C program before, you will be able to immediately transfer most of your knowledge to building tracing programs in D. If you have never written a C program before, learning D is still very easy. You will understand all of the syntax by the end of this chapter. But first, let us take a step back from language rules and learn more about how DTrace works, and then we will return to learning how to build more interesting D programs.

1.2 Providers and Probes

In the preceding examples, you learnt to use two simple probes named BEGIN and END. DTrace probes come from a set of kernel modules called providers, each of which performs a particular kind of instrumentation to create probes. When you use DTrace, each provider is given an opportunity to publish
the probes it can provide to the DTrace framework. You can then enable and bind your tracing actions to any of the probes that have been published. To list all of the available probes on your system, type the command:

```bash
# dtrace -l
```

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dtrace</td>
<td></td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dtrace</td>
<td></td>
<td>END</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>dtrace</td>
<td></td>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>dt_test</td>
<td>dt_test</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>syscall</td>
<td></td>
<td>read entry</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>syscall</td>
<td></td>
<td>read return</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>syscall</td>
<td></td>
<td>write entry</td>
<td></td>
</tr>
</tbody>
</table>

... many lines of output omitted ...

It might take some time to display all of the output. To count all the probes, type the command:

```bash
# dtrace -l | wc -l
4097
```

You might observe a different total on your machine, as the number of probes varies depending on your operating platform, the software you have installed, and the provider modules that you have loaded. Notice that each probe has the two names we mentioned earlier, an integer ID and a human-readable name. The human readable name is composed of four parts, shown as separate columns in the dtrace output. The four parts of a probe name are:

- **provider**: The name of the DTrace provider that is publishing this probe. For kernel probes, the provider name typically corresponds to the name of the DTrace kernel module that performs the instrumentation to enable the probe. For probes in user-space, it is the provider name that was defined for the program or library appended with the process ID of the running executable.

- **module**: If this probe corresponds to a specific program location, the name of the kernel module, library, or user-space program in which the probe is located.

- **function**: If this probe corresponds to a specific program location, the name of the program function in which the probe is located.

- **name**: The final component of the probe name is a name that gives you some idea of the probe’s semantic meaning, such as `BEGIN` or `END`.

When writing out the full human-readable name of a probe, write all four parts of the name separated by colons like this:

```
p:module:function:name
```

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

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

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

Now that you understand where probes originate from and how they are named, we are going to learn a little more about what happens when you enable probes and ask DTrace to do something, and then we will return to our whirlwind tour of D.
# Chapter 2 The D Programming Language

## Table of Contents

2.1 D Program Structure ................................................................. 6  
  2.1.1 Probe Clauses and Declarations .......................................... 6  
  2.1.2 Probe Descriptions ........................................................... 6  
  2.1.3 Predicates ........................................................................ 8  
  2.1.4 Actions ........................................................................... 8  
  2.1.5 Order of Execution ............................................................. 8  
  2.1.6 Use of the C Preprocessor .................................................. 8  
2.2 Compilation and Instrumentation ............................................... 9  
2.3 Variables and Arithmetic Expressions ......................................... 10  
2.4 Predicates ............................................................................... 12  
2.5 Output Formatting .................................................................. 15  
2.6 Arrays .................................................................................... 17  
2.7 External Symbols and Types ..................................................... 18  
2.8 Types, Operators, and Expressions ........................................... 19  
  2.8.1 Identifier Names and Keywords ........................................... 19  
  2.8.2 Data Types and Sizes .......................................................... 20  
  2.8.3 Constants .......................................................................... 21  
  2.8.4 Arithmetic Operators ........................................................... 22  
  2.8.5 Relational Operators ............................................................ 23  
  2.8.6 Logical Operators ............................................................... 23  
  2.8.7 Bitwise Operators ............................................................... 24  
  2.8.8 Assignment Operators ......................................................... 24  
  2.8.9 Increment and Decrement Operators .................................... 25  
  2.8.10 Conditional Expressions ...................................................... 26  
  2.8.11 Type Conversions ............................................................. 26  
  2.8.12 Precedence ...................................................................... 27  
2.9 Variables .................................................................................. 28  
  2.9.1 Scalar Variables ................................................................ 28  
  2.9.2 Associative Arrays ............................................................... 29  
  2.9.3 Thread-Local Variables ........................................................ 30  
  2.9.4 Clause-Local Variables ....................................................... 32  
  2.9.5 Built-in Variables ............................................................... 34  
  2.9.6 External Variables ............................................................. 35  
2.10 Pointers and Arrays ................................................................. 36  
  2.10.1 Pointers and Addresses ....................................................... 36  
  2.10.2 Pointer Safety ................................................................. 37  
  2.10.3 Array Declarations and Storage .......................................... 38  
  2.10.4 Pointer and Array Relationship .......................................... 39  
  2.10.5 Pointer Arithmetic ............................................................ 39  
  2.10.6 Generic Pointers ............................................................... 40  
  2.10.7 Multi-Dimensional Arrays .................................................. 40  
  2.10.8 Pointers to DTrace Objects ............................................... 41  
  2.10.9 Pointers and Address Spaces .............................................. 41  
2.11 Strings ................................................................................... 42  
  2.11.1 String Representation ......................................................... 42  
  2.11.2 String Constants ............................................................... 42  
  2.11.3 String Assignment ............................................................. 43  
  2.11.4 String Conversion ............................................................ 43
D is a systems programming language that allows you to interface with the operating system APIs and with the hardware.

2.1 D Program Structure

D programs consist of a set of clauses that describe probes to enable and predicates and actions to bind to these probes. D programs can also contain declarations of variables, as described in Section 2.9, “Variables”, and definitions of new types, described in Section 2.13, “Type and Constant Definitions”. This chapter formally describes the overall structure of a D program and features for constructing probe descriptions that match more than one probe. We will also discuss the use of the C preprocessor, `cpp`, with D programs.

2.1.1 Probe Clauses and Declarations

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

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

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

Declarations can be used to declare D variables and external C symbols as discussed in Section 2.9, “Variables”, or to define new types for use in D, as described in Section 2.13, “Type and Constant Definitions”. Special D compiler directives called pragmas may also appear anywhere in a D program, including outside of probe clauses. D pragmas are specified on lines beginning with a # character. D pragmas are used, for example, to set run-time DTrace options; see Chapter 10, Options and Tunables for details.

2.1.2 Probe Descriptions

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

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

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

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

The special characters in the following table are recognized in probe name patterns:

\begin{center}
\begin{tabular}{ | l | c |}
\hline
Symbol & Description \\
\hline
* & Matches any string, including the null string. \\
? & Matches any single character. \\
[... ] & Matches any one of the enclosed characters. A pair of characters separated by \texttt{-} matches any character between the pair, inclusive. If the first character after the \texttt{[} is \texttt{!}, any character not enclosed in the set is matched. \\
\texttt{\} } & Interpret the next character as itself, without any special meaning. \\
\hline
\end{tabular}
\end{center}

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

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

\begin{verbatim}
syscall::*lwp*:entry, syscall::*sock*:entry
{
    trace(timestamp);
}
\end{verbatim}

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

\begin{verbatim}
12345
{
    trace(timestamp);
}
\end{verbatim}
could be used to enable probe ID 12345, as reported by dtrace -l -i 12345.

Note
You should always write your D programs using human-readable probe descriptions. Integer probe IDs are not guaranteed to remain consistent as DTrace provider kernel modules are loaded and unloaded or following a reboot.

2.1.3 Predicates

Predicates are expressions enclosed in a pair of slashes (//) that are evaluated at probe firing time to determine whether the associated actions should be executed. Predicates are the primary conditional construct used for building more complex control flow in a D program. You can omit the predicate section of the probe clause entirely for any probe, in which case the actions are always executed when the probe fires.

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

2.1.4 Actions

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

2.1.5 Order of Execution

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

No other ordering constraints are imposed. It is not uncommon for the output from two distinct probes to appear interspersed or in the opposite order from which the probes fired.

2.1.6 Use of the C Preprocessor

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

The D compiler automatically loads the set of C type descriptions associated with the operating system implementation, but you can use the preprocessor to include other type definitions such as types used in your own C programs. You can also use the preprocessor to perform other tasks such as creating macros that expand to chunks of D code and other program elements. If you use the preprocessor with your D
program, you may only include files that contain valid D declarations. The D compiler can correctly interpret C header files that include only external declarations of types and symbols. The D compiler cannot parse C header files that include additional program elements such as C function source code, which produces an appropriate error message.

2.2 Compilation and Instrumentation

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

All of the instrumentation in DTrace is completely dynamic: probes are enabled discretely only when you are using them. No instrumented code is present for inactive probes, so your system does not experience any kind of performance degradation when you are not using DTrace. Once your experiment is complete and the `dtrace` command exits, all of the probes you used are automatically disabled and their instrumentation is removed, returning your system to its exact original state. No effective difference exists between a system where DTrace is not active and one where the DTrace software is not installed, other than a few megabytes of disk space that is required for type information and for DTrace itself.

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

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

Unlike programs written in C and C++ and similar to programs written in the Java programming language, DTrace D programs are compiled into a safe intermediate form that is used for execution when your probes fire. This intermediate form is validated for safety when your program is first examined by the DTrace kernel software. The DTrace execution environment also handles any run-time errors that might occur during your D program's execution, including dividing by zero, dereferencing invalid memory, and so on, and reports them to you. As a result, you can never construct an unsafe program that would cause DTrace to inadvertently damage the operating system kernel or one of the processes running on your system. These safety features allow you to use DTrace in a production environment without worrying about crashing or corrupting your system. If you make a programming mistake, DTrace will report your error to you, disable your instrumentation, and you can correct your mistake and try again. The DTrace error reporting and debugging features are described later in this book.

Figure 2.1, “Overview of the DTrace Architecture and Components” shows the different components of the DTrace architecture, including providers, probes, the DTrace kernel software, and the `dtrace` command.
Now that you understand how DTrace works, let us return to the tour of the D programming language and start writing some more interesting programs.

### 2.3 Variables and Arithmetic Expressions

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

```d
/*
 * Count off and report the number of seconds elapsed
 */
dtrace:::BEGIN
{
  i = 0;
}
profile:::tick-1sec
{
  i = i + 1;
  trace(i);
}
dtrace:::END
{
  trace(i);
}
```

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

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

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

\[ i = 0; \]

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Character or single byte integer</td>
</tr>
<tr>
<td>int</td>
<td>Default integer</td>
</tr>
<tr>
<td>short</td>
<td>Short integer</td>
</tr>
<tr>
<td>long</td>
<td>Long integer</td>
</tr>
<tr>
<td>long long</td>
<td>Extended long integer</td>
</tr>
</tbody>
</table>

The sizes of these types are dependent on the operating system kernel's data model, described in Section 2.8, “Types, Operators, and Expressions”. D also provides built-in friendly names for signed and unsigned integer types of various fixed sizes, as well as thousands of other types that are defined by the operating system.

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

```d
counter.d

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

This clause names the probe profile:::tick-1sec, which tells the profile provider to create a new probe which fires once per second on an available processor. The clause contains two statements, the first assigning i to the previous value plus one, and the second tracing the new value of i. All the usual C arithmetic operators are available in D; the complete list is found in Section 2.8, “Types, Operators, and Expressions”. Also as in C, the ++ operator can be used as shorthand for incrementing the corresponding variable by one. The trace function takes any D expression as its argument, so you could write counter.d more concisely as follows:
If you want to explicitly control the type of the variable `i`, you can surround the desired type in parentheses when you assign it in order to cast the integer zero to a specific type. For example, if you wanted to determine the maximum size of a `char` in D, you could change the `BEGIN` clause as follows:

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

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

### 2.4 Predicates

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

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

```d
dtrace:::BEGIN
{ i = 10; }
profile:::tick-1sec
/i > 0/
{ trace(i--); }
profile:::tick-1sec
/i == 0/
{ trace("blastoff!"); exit(0); }
```

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

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU ID FUNCTION:NAME
0 638 :tick-1sec 10
0 638 :tick-1sec 9
0 638 :tick-1sec 8
0 638 :tick-1sec 7
0 638 :tick-1sec 6
```
This example uses the `BEGIN` probe to initialize an integer `i` to 10 to begin the countdown. Next, as in the previous example, the program uses the `tick-1sec` probe to implement a timer that fires once per second. Notice that in `countdown.d`, the `tick-1sec` probe description is used in two different clauses, each with a different predicate and action list. The predicate is a logical expression surrounded by enclosing slashes `//` that appears after the probe name and before the braces `{}` that surround the clause statement list.

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

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

The relational operator `>` means *greater than* and returns the integer value zero for false and one for true. All of the C relational operators are supported in D; the complete list is found in Section 2.8, “Types, Operators, and Expressions”. If `i` is not yet zero, the script traces `i` and then decrements it by one using the `--` operator.

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

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

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

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

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

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

Now let us combine predicates with a new provider, the `syscall` provider, and create our first real D tracing program. The `syscall` provider permits you to enable probes on entry to or return from any Oracle Linux system call. The next example uses DTrace to observe every time your shell performs a `read()` or `write()` system call. First, open two windows, one to use for DTrace and the other containing the shell process that you are going to watch. In the second window, type the following command to obtain the process ID of this shell:
# echo $$
2860

Now go back to your first window and type the following D program and save it in a file named `rw.d`. As you type in the program, replace the integer constant `2860` with the process ID of the shell that was printed in response to your `echo` command.

```d
syscall::read:entry,
syscall::write:entry
/pid == 2860/
{
}
```

Notice that the body of `rw.d`'s probe clause is left empty because the program is only intended to trace notification of probe firings and not to trace any additional data. Once you have typed in `rw.d`, use `dtrace` to start your experiment and then go to your second shell window and type a few commands, pressing return after each command. As you type, you should see `dtrace` report probe firings in your first window, similar to the following example:

```d
# dtrace -s rw.d
dtrace: script 'rw.d' matched 2 probes
<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>read:entry</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>read:entry</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>read:entry</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>write:entry</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>read:entry</td>
</tr>
</tbody>
</table>

...^C
```

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

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

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

```d
/pid == 2860/
```

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

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>errno</td>
<td>int</td>
<td>Current <code>errno</code> value for system calls</td>
</tr>
<tr>
<td>execname</td>
<td>string</td>
<td>Name of the current process's executable file</td>
</tr>
<tr>
<td>pid</td>
<td>pid_t</td>
<td>Process ID of the current process</td>
</tr>
</tbody>
</table>
Now that you've written a real instrumentation program, try experimenting with it on different processes running on your system by changing the process ID and the system call probes that are instrumented. Then, you can make one more simple change and turn `rw.d` into a very simple version of a system call tracing tool like `strace`. An empty probe description field acts as a wildcard, matching any probe, so change your program to the following new source code to trace any system call executed by your shell:

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

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

### 2.5 Output Formatting

System call tracing is a powerful way to observe the behavior of most user processes. The following example improves upon the earlier `rw.d` program by formatting its output so you can more easily understand the output. Type the following program and save it in a file called `stracerw.d`:

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

In this example, the constant `2860` is replaced with the label `$1` in each predicate. This label allows you to specify the process of interest as an argument to the script; `$1` is replaced by the value of the first argument when the script is compiled. To execute `stracerw.d`, use the `dtrace` options `-q` and `-s`, followed by the process ID of your shell as the final argument. The `-q` option indicates that `dtrace` should be quiet and suppress the header line and the CPU and ID columns shown in the preceding examples. As a result, you only see the output for the data that you explicitly trace. Type the following command (replacing `2860` with the process ID of a shell process) and then press return a few times in the specified shell:

```
dtrace -q -s stracerw.d 2860
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>id_t</td>
<td>Thread ID of the current thread</td>
</tr>
<tr>
<td>probeprov</td>
<td>string</td>
<td>Current probe description's provider field</td>
</tr>
<tr>
<td>probemod</td>
<td>string</td>
<td>Current probe description's module field</td>
</tr>
<tr>
<td>probefunc</td>
<td>string</td>
<td>Current probe description's function field</td>
</tr>
<tr>
<td>probename</td>
<td>string</td>
<td>Current probe description's name field</td>
</tr>
</tbody>
</table>
Now let us examine your D program and its output in more detail. First, a clause similar to the earlier program instruments each of the shell’s calls to \texttt{read()} and \texttt{write()}. But for this example, we use a new function, \texttt{printf}, to trace the data and print it out in a specific format:

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

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

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

<table>
<thead>
<tr>
<th>Format Conversion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Print the corresponding value as a decimal integer</td>
</tr>
<tr>
<td>%s</td>
<td>Print the corresponding value as a string</td>
</tr>
<tr>
<td>%x</td>
<td>Print the corresponding value as a hexadecimal integer</td>
</tr>
</tbody>
</table>

DTrace \texttt{printf} works just like the C \texttt{printf()} library routine or the shell \texttt{printf} utility. If you have never seen \texttt{printf} before, the formats and options are explained in detail in Chapter 6, \textit{Output Formatting}. You should read this chapter carefully even if you are already familiar with \texttt{printf} from another language. In D, \texttt{printf} is provided as a built-in and some new format conversions are available to you designed specifically for DTrace.

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

\begin{verbatim}
# dtrace -q -s stracerw.d
dtrace: failed to compile script stracerw.d: line 5: printf( )
argument #2 is incompatible with conversion #1 prototype:
  conversion: %s
  prototype: char [] or string (or use stringof)
  argument: int
#
\end{verbatim}

To print the name of the read or write system call and its arguments, use the \texttt{printf} statement:

\begin{verbatim}
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
\end{verbatim}
to trace the name of the current probe function and the first three integer arguments to the system call, available in the DTrace variables \texttt{arg0, arg1, and arg2}. For more information about probe arguments, see Section 2.9, “Variables”. The first argument to \texttt{read()} and \texttt{write()} is a file descriptor, printed in decimal. The second argument is a buffer address, formatted as a hexadecimal value. The final argument is the buffer size, formatted as a decimal value. The format specifier \%4d is used for the third argument to indicate that the value should be printed using the \%d format conversion with a minimum field width of 4 characters. If the integer is less than 4 characters wide, \texttt{printf} inserts extra blanks to align the output.

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

\begin{verbatim}
syscall::read:return, syscall::write:return
\end{verbatim}

\begin{verbatim}/pid == $1/
{ printf("\tt = %d
", arg1);
}
\end{verbatim}

Notice that the \texttt{syscall} provider also publishes a probe named \texttt{return} for each system call in addition to \texttt{entry}. The DTrace variable \texttt{arg1} for the syscall \texttt{return} probes evaluates to the system call's return value. The return value is formatted as a decimal integer. The character sequences beginning with backwards slashes in the format string expand to tab (\texttt{\t}) and newline (\texttt{\n}) respectively. These \textit{escape sequences} help you print or record characters that are difficult to type. D supports the same set of escape sequences as C, C++, and the Java programming language. The complete list of escape sequences is found in Section 2.8, “Types, Operators, and Expressions”.

\section*{2.6 Arrays}

D permits you to define variables that are integers, as well as other types to represent strings and composite types called \textit{structs} and \textit{unions}. If you are familiar with C programming, you will be happy to know you can use any type in D that you can in C. If you are not a C expert, do not worry: the different kinds of data types are all described in Section 2.8, “Types, Operators, and Expressions”. D also supports a special kind of variable called an \textit{associative array}. An associative array is similar to a normal array in that it associates a set of keys with a set of values, but in an associative array the keys are not limited to integers of a fixed range.

D associative arrays can be indexed by a list of one or more values of any type. Together the individual key values form a \textit{tuple} that is used to index into the array and access or modify the value corresponding to that key. Every tuple used with a given associative array must conform to the same type signature; that is, each tuple key must be of the same length and have the same key types in the same order. The value associated with each element of a given associative array is also of a single fixed type for the entire array. For example, the following D statement defines a new associative array \texttt{a} of value type \texttt{int} with the tuple signature \texttt{string, int} and stores the integer value 456 in the array:

\begin{verbatim}
a["hello", 123] = 456;
\end{verbatim}

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

\begin{verbatim}
a["hello", 123]++;
\end{verbatim}

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

\begin{verbatim}
syscall::read:entry, syscall::write:entry
/pid == $1/
{ ts[probefunc] = timestamp;
}
\end{verbatim}
As with `stracerw.d`, specify the ID of the shell process when you execute `rwtme.d`. If you type a few shell commands, you will see the amount of time elapsed during each system call. Type in the following command and then press return a few times in your other shell:

```
dtrace -s rwtime.d `/usr/bin/pgrep -n bash`
```

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

D uses the backquote character (`) as a special scoping operator for accessing symbols that are defined in the operating system and not in your D program. For example, the Oracle Linux kernel contains a C declaration of a system variable named `max_pfn`. This variable is declared in C in the kernel source code as follows:

```c
unsigned long max_pfn
```

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

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

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

You have now completed a whirlwind tour of DTrace and you have learned many of the basic DTrace building blocks necessary to build larger and more complex D programs. The following chapters describe the complete set of rules for D and demonstrate how DTrace can make complex performance measurements and functional analysis of the system easy. Later, you will see how to use DTrace to connect user application behavior to system behavior, giving you the capability to analyze your entire software stack.

## 2.8 Types, Operators, and Expressions

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

### 2.8.1 Identifier Names and Keywords

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

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

<table>
<thead>
<tr>
<th>Table 2.2 D Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto*</td>
</tr>
<tr>
<td>break*</td>
</tr>
<tr>
<td>case*</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>const</td>
</tr>
</tbody>
</table>
D reserves for use as keywords a superset of the ANSI C keywords. The keywords reserved for future use by the D language are marked with "*". The D compiler will produce a syntax error if you attempt to use a keyword that is reserved for future use. The keywords defined by D but not defined by ANSI C are marked with "+". D provides the complete set of types and operators found in ANSI C. The major difference in D programming is the absence of control-flow constructs. Keywords associated with control-flow in ANSI C are reserved for future use in D.

### 2.8.2 Data Types and Sizes

D provides fundamental data types for integers and floating-point constants. Arithmetic may only be performed on integers in D programs. Floating-point constants may be used to initialize data structures, but floating-point arithmetic is not permitted in D. In Oracle Linux, D provides a 64-bit data model for use in writing programs; it does not support a 32-bit data model. The data model used when executing your program is the native data model associated with the active operating system kernel, which must also be 64-bit.

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

**Table 2.3 D Integer Data Types**

<table>
<thead>
<tr>
<th>Type Name</th>
<th>64-bit Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
</tr>
<tr>
<td>long long</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

Integer types may be prefixed with the signed or unsigned qualifier. If no sign qualifier is present, the type is assumed to be signed. The D compiler also provides the type aliases listed in the following table:

**Table 2.4 D Integer Type Aliases**

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>1-byte signed integer</td>
</tr>
<tr>
<td>int16_t</td>
<td>2-byte signed integer</td>
</tr>
<tr>
<td>int32_t</td>
<td>4-byte signed integer</td>
</tr>
<tr>
<td>int64_t</td>
<td>8-byte signed integer</td>
</tr>
<tr>
<td>IntPtr_t</td>
<td>Signed integer of size equal to a pointer</td>
</tr>
<tr>
<td>uint8_t</td>
<td>1-byte unsigned integer</td>
</tr>
<tr>
<td>uint16_t</td>
<td>2-byte unsigned integer</td>
</tr>
<tr>
<td>uint32_t</td>
<td>4-byte unsigned integer</td>
</tr>
<tr>
<td>uint64_t</td>
<td>8-byte unsigned integer</td>
</tr>
<tr>
<td>uintptr_t</td>
<td>Unsigned integer of size equal to a pointer</td>
</tr>
</tbody>
</table>
These type aliases are equivalent to using the name of the corresponding base type in the previous table and are appropriately defined for each data model. For example, the type name `uint8_t` is an alias for the type unsigned `char`. See Section 2.13, “Type and Constant Definitions” for information on how to define your own type aliases for use in your D programs.

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

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

<table>
<thead>
<tr>
<th>Type Name</th>
<th>64-bit Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
</tr>
<tr>
<td>long double</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

D also provides the special type `string` to represent ASCII strings. Strings are discussed in more detail in Section 2.11, “Strings”.

2.8.3 Constants

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>D type</th>
</tr>
</thead>
<tbody>
<tr>
<td>u or U</td>
<td>unsigned version of the type selected by the compiler</td>
</tr>
<tr>
<td>l or L</td>
<td>long</td>
</tr>
<tr>
<td>ul or UL</td>
<td>unsigned long</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long</td>
</tr>
<tr>
<td>ull or ULL</td>
<td>unsigned long long</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>D type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f or F</td>
<td>float</td>
</tr>
<tr>
<td>l or L</td>
<td>long double</td>
</tr>
</tbody>
</table>

Character constants are written as a single character or escape sequence enclosed in a pair of single quotes (‘a’). Character constants are assigned the type `int` and are equivalent to an integer constant whose value is determined by that character’s value in the ASCII character set. Refer to the `ascii(7)`
manual page for a list of characters and their values. You can also use any of the special escape sequences shown in the following table in your character constants. D supports the same escape sequences as are found in ANSI C.

Table 2.6 Character Escape Sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Represents</th>
<th>Escape Sequence</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alert</td>
<td>&quot;</td>
<td>backslash</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
<td>\000</td>
<td>octal value 000</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tab</td>
<td>\xhh</td>
<td>hexadecimal value 0xhh</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
<td>\0</td>
<td>null character</td>
</tr>
</tbody>
</table>

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

Strings constants of any length can be composed by enclosing them in a pair of double quotes ("hello"). A string constant may not contain a literal newline character. To create strings containing newlines, use the \n escape sequence instead of a literal newline. String constants may contain any of the special character escape sequences shown for character constants above. Similar to ANSI C, strings are represented as arrays of characters terminated by a null character (\0) that is implicitly added to each string constant that you declare. String constants are assigned the special D type string. The D compiler provides a set of special features for comparing and tracing character arrays that are declared as strings, as described in Section 2.11, “Strings”.

2.8.4 Arithmetic Operators

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

Table 2.7 Binary Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>integer addition</td>
</tr>
<tr>
<td>-</td>
<td>integer subtraction</td>
</tr>
<tr>
<td>*</td>
<td>integer multiplication</td>
</tr>
<tr>
<td>/</td>
<td>integer division</td>
</tr>
<tr>
<td>%</td>
<td>integer modulus</td>
</tr>
</tbody>
</table>

Arithmetic in D may only be performed on integer operands, or on pointers, as discussed in Section 2.10, “Pointers and Arrays”. Arithmetic may not be performed on floating-point operands in D programs. The DTrace execution environment does not take any action on integer overflow or underflow. You must check for these conditions yourself in situations where overflow and underflow can occur.

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

In addition to these binary operators, the + and – operators may also be used as unary operators as well; these operators have higher precedence than any of the binary arithmetic operators. The order of precedence and associativity properties for all the D operators is presented in Table 2.12, “D Operator Precedence and Associativity”. You can control precedence by grouping expressions in parentheses () .

### 2.8.5 Relational Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>left-hand operand is less than right-operand</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>left-hand operand is less than or equal to right-hand operand</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>left-hand operand is greater than right-operand</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>left-hand operand is greater than or equal to right-hand operand</td>
</tr>
<tr>
<td><code>==</code></td>
<td>left-hand operand is equal to right-hand operand</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>left-hand operand is not equal to right-hand operand</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>D string comparison</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;coffee&quot; &lt; &quot;espresso&quot;</td>
<td>Returns 1 (true)</td>
</tr>
<tr>
<td>&quot;coffee&quot; == &quot;coffee&quot;</td>
<td>Returns 1 (true)</td>
</tr>
<tr>
<td>&quot;coffee&quot; &gt;= &quot;mocha&quot;</td>
<td>Returns 0 (false)</td>
</tr>
</tbody>
</table>

Relational operators may also be used to compare a data object associated with an enumeration type with any of the enumerator tags defined by the enumeration. Enumerations are a facility for creating named integer constants and are described in more detail in Section 2.13, “Type and Constant Definitions”.

### 2.8.6 Logical Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Logical <strong>AND</strong>: true if both operands are true</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>^^</code></td>
<td>Logical <strong>XOR</strong>: true if exactly one operand is true</td>
</tr>
</tbody>
</table>


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

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

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

### 2.8.7 Bitwise Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift the left-hand operand left by the number of bits specified by the right-hand operand</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift the left-hand operand right by the number of bits specified by the right-hand operand</td>
</tr>
</tbody>
</table>

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

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

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

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

### 2.8.8 Assignment Operators

D provides the following binary assignment operators for modifying D variables. You can only modify D variables and arrays. Kernel data objects and constants may not be modified using the D assignment operators. The assignment operators have the same meaning as they do in ANSI C.
Table 2.11 D Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Set the left-hand operand equal to the right-hand expression value</td>
</tr>
<tr>
<td>+=</td>
<td>Increment the left-hand operand by the right-hand expression value</td>
</tr>
<tr>
<td>-=</td>
<td>Decrement the left-hand operand by the right-hand expression value</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply the left-hand operand by the right-hand expression value</td>
</tr>
<tr>
<td>/=</td>
<td>Divide the left-hand operand by the right-hand expression value</td>
</tr>
<tr>
<td>%=</td>
<td>Modulo the left-hand operand by the right-hand expression value</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise AND the left-hand operand with the right-hand expression value</td>
</tr>
<tr>
<td>^=</td>
<td>Bitwise XOR the left-hand operand with the right-hand expression value</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Shift the left-hand operand left by the number of bits specified by the right-hand expression value</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Shift the left-hand operand right by the number of bits specified by the right-hand expression value</td>
</tr>
</tbody>
</table>

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

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

### 2.8.9 Increment and Decrement Operators

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

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

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

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

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

The increment and decrement operators can be applied to integer or pointer variables. When applied to integer variables, the operators increment or decrement the corresponding value by one. When applied to pointer variables, the operators increment or decrement the pointer address by the size of the data type.
referenced by the pointer. Pointers and pointer arithmetic in D are discussed in Section 2.10, “Pointers and Arrays”.

### 2.8.10 Conditional Expressions

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

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

In this example, the expression `i == 0` is first evaluated to determine whether it is true or false. If the expression is true, the second expression is evaluated and its value is returned. If the expression is false, the third expression is evaluated and its value is returned.

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

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

The first expression used with `?:` must be a pointer or integer in order to be evaluated for its truth value. The second and third expressions may be of any compatible types. You may not construct a conditional expression where, for example, one path returns a string and another path returns an integer. The second and third expressions also may not invoke a tracing function such as `trace` or `printf`. If you want to conditionally trace data, use a predicate instead, as discussed in Section 2.4, “Predicates”.

### 2.8.11 Type Conversions

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

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

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

Integers and other types can also be explicitly cast from one type to another. In D, pointers and integers can be cast to any integer or pointer types, but not to other types. Rules for casting and promoting strings and character arrays are discussed in Section 2.11, “Strings”. An integer or pointer cast is formed using an expression such as:

```d
y = (int)x;
```
where the destination type is enclosed in parentheses and used to prefix the source expression. Integers are cast to types of higher rank by performing promotion. Integers are cast to types of lower rank by zeroing the excess high-order bits of the integer.

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

### 2.8.12 Precedence

Table 2.12, “D Operator Precedence and Associativity” lists the D rules for operator precedence and associativity. These rules are somewhat complex, but are necessary to provide precise compatibility with the ANSI C operator precedence rules. The table entries are in order from highest precedence to lowest precedence.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ + + - - * &amp; (type) sizeof stringof offsetof xlate</td>
<td>right to left</td>
</tr>
<tr>
<td>* / % + - &lt;&lt; &gt;&gt; &lt; &lt;= &gt; &gt;= == != &amp; ^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>right to left</td>
</tr>
<tr>
<td>+= -= *= /= %= &amp;= ^= ?= &lt;&lt;= &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof</td>
<td>Computes the size of an object (see Section 2.12, “Structs and Unions”)</td>
</tr>
<tr>
<td>offsetof</td>
<td>Computes the offset of a type member (see Section 2.12, “Structs and Unions”)</td>
</tr>
<tr>
<td>stringof</td>
<td>Converts the operand to a string (see Section 2.11, “Strings”)</td>
</tr>
<tr>
<td>xlate</td>
<td>Translates a data type (see Chapter 17, Translators)</td>
</tr>
<tr>
<td>unary &amp;</td>
<td>Computes the address of an object (see Section 2.10, “Pointers and Arrays”)</td>
</tr>
<tr>
<td>unary *</td>
<td>Dereferences a pointer to an object (see Section 2.10, “Pointers and Arrays”)</td>
</tr>
<tr>
<td>-&gt; and .</td>
<td>Accesses a member of a structure or union type (see Section 2.12, “Structs and Unions”)</td>
</tr>
</tbody>
</table>
The comma (,) operator listed in the table is for compatibility with the ANSI C comma operator, which can be used to evaluate a set of expressions in left-to-right order and return the value of the right most expression. This operator is provided strictly for compatibility with C and should generally not be used.

The () entry in the table of operator precedence represents a function call; examples of calls to functions such as printf and trace are presented in Chapter 6, Output Formatting. A comma is also used in D to list arguments to functions and to form lists of associative array keys. This comma is not the same as the comma operator and does not guarantee left-to-right evaluation. The D compiler provides no guarantee as to the order of evaluation of arguments to a function or keys to an associative array. You should be careful of using expressions with interacting side-effects, such as the pair of expressions i and i++, in these contexts.

The [] entry in the table of operator precedence represents an array or associative array reference. Examples of associative arrays are presented in Section 2.9.2, “Associative Arrays”. A special kind of associative array called an aggregation is described in Chapter 3, Aggregations. The [] operator can also be used to index into fixed-size C arrays as well, as described in Section 2.10, “Pointers and Arrays”.

2.9 Variables

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

Note

- Scalar variables and associative arrays have a global scope and are not multi-processor safe (MP-safe). As the value of such variables can be changed by more than one processor, there is a chance that a variable can become corrupted if more than one probe modifies it.
- Aggregations are MP-safe even though they have a global scope.

2.9.1 Scalar Variables

Scalar variables are used to represent individual fixed-size data objects, such as integers and pointers. Scalar variables can also be used for fixed-size objects that are composed of one or more primitive or composite types. D provides the ability to create arrays of objects as well as composite structures. DTrace also represents strings as fixed-size scalars by permitting them to grow up to a predefined maximum length. Control over string length in your D program is discussed further in Section 2.11, “Strings”.

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

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

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

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

```d
int x; /* declare an integer x for later use */
BEGIN
    x = 123;
    ...
END
```

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

The D language definition places no limit on the size and number of D variables, but limits are defined by the DTrace implementation and by the memory available on your system. The D compiler will enforce any of the limitations that can be applied at the time you compile your program. You can learn more about how to tune options related to program limits in Chapter 10, Options and Tunables.

### 2.9.2 Associative Arrays

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

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

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

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

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

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

The type of each object contained in the array is also fixed for all elements in a given array. Because a was first assigned using the integer 456, every subsequent value stored in the array will also be of type `int`. You can use any of the assignment operators defined in Section 2.8, “Types, Operators, and Expressions” to modify associative array elements, subject to the operand rules defined for each operator. The D compiler will produce an appropriate error message if you attempt an incompatible assignment. You can use any type with an associative array key or value that you can use with a scalar variable. You cannot nest an associative array within another associative array as a key or value.

You can reference an associative array using any tuple that is compatible with the array key signature. The rules for tuple compatibility are similar to those for function calls and variable assignments: the tuple
Thread-Local Variables

must be of the same length and each type in the list of actual parameters must be compatible with the corresponding type in the formal key signature. For example, if an associative array `x` is defined as follows:

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

then the key signature is of type unsigned `long long` and the values are of type `int`. This array can also be referenced using the expression `x['a']` because the tuple consisting of the character constant `'a'` of type `int` and length one is compatible with the key signature unsigned `long long` according to the arithmetic conversion rules described in Section 2.8.11, “Type Conversions”.

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

```d
int x[unsigned long long, char];
BEGIN
{
  x[123ull, 'a'] = 456;
}
```

Once an associative array is defined, references to any tuple of a compatible key signature are permitted, even if the tuple in question has not been previously assigned. Accessing an unassigned associative array element is defined to return a zero-filled object. A consequence of this definition is that underlying storage is not allocated for an associative array element until a non-zero value is assigned to that element. Conversely, assigning an associative array element to zero causes DTrace to deallocate the underlying storage. This behavior is important because the dynamic variable space out of which associative array elements are allocated is finite; if it is exhausted when an allocation is attempted, the allocation fails and an error message is generated indicating a dynamic variable drop. Always assign zero to associative array elements that are no longer in use. See Chapter 10, Options and Tunables for other techniques to eliminate dynamic variable drops.

### 2.9.3 Thread-Local Variables

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

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

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

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

Once you have defined a thread-local variable, you can reference it for any thread in the system even if the variable in question has not been previously assigned for that particular thread. If a thread's copy of the
Thread-local variables have not yet been assigned, the data storage for the copy is defined to be filled with zeroes. As with associative array elements, underlying storage is not allocated for a thread-local variable until a non-zero value is assigned to it. Also as with associative array elements, assigning zero to a thread-local variable causes DTrace to deallocate the underlying storage. Always assign zero to thread-local variables that are no longer in use. See Chapter 10, Options and Tunables for other techniques to fine-tune the dynamic variable space from which thread-local variables are allocated.

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

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

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

```d
self int x; /* declare int x as a thread-local variable */
syscall::read:entry
{
    self->x = 123;
}
```

Thread-local variables are kept in a separate namespace from global variables so you can reuse names. Remember that `x` and `self->x` are not the same variable if you overload names in your program.

The following example shows how to use thread-local variables. In a text editor, type in the following program and save it in a file named `rtime.d`:

```d
syscall::read:entry
{
    self->t = timestamp;
}
syscall::read:return
/self->t != 0/
{
    printf("%d/%d spent %d nsecs in read()\n", pid, tid, timestamp - self->t);
    /*
    * We are done with this thread-local variable; assign zero to it
    * to allow the DTrace runtime to reclaim the underlying storage.
    */
    self->t = 0;
}
```

Now go to your shell and start the program running. Wait a few seconds and you should start to see some output. If no output appears, try running a few commands.

```
# dtrace -q -s rtime.d
3987/3987 spent 12786263 nsecs in read()
2183/2183 spent 13410 nsecs in read()
2183/2183 spent 12850 nsecs in read()
2183/2183 spent 10057 nsecs in read()
3583/3583 spent 14527 nsecs in read()
3583/3583 spent 12571 nsecs in read()
3583/3583 spent 9778 nsecs in read()
3583/3583 spent 9498 nsecs in read()
3583/3583 spent 9778 nsecs in read()
2183/2183 spent 13968 nsecs in read()
2183/2183 spent 72076 nsecs in read()
```
Clause-Local Variables

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

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

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

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

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

### 2.9.4 Clause-Local Variables

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

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

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

```d
this int x; /* an integer clause-local variable */
this char c; /* a character clause-local variable */
BEGIN
{  
    this->x = 123;
    this->c = 'D';
}
```

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

```d
int me; /* an integer global variable */
```
**Clause-Local Variables**

```d
this int foo; /* an integer clause-local variable */
tick-1sec
{ /*
  * Set foo to be 10 if and only if this is the first clause executed.
  */
  this->foo = (me % 3 == 0) ? 10 : this->foo;
  printf("Clause 1 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
tick-1sec
{ /*
  * Set foo to be 20 if and only if this is the first clause executed.
  */
  this->foo = (me % 3 == 0) ? 20 : this->foo;
  printf("Clause 2 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
tick-1sec
{ /*
  * Set foo to be 30 if and only if this is the first clause executed.
  */
  this->foo = (me % 3 == 0) ? 30 : this->foo;
  printf("Clause 3 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
```

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

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

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

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

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

The following table provides a complete list of D built-in variables. All of these variables are scalar global variables; no thread-local or clause-local variables or built-in associative arrays are currently defined by D.

Table 2.13 DTrace Built-in Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>args[]</code></td>
<td>The typed arguments to the current probe, if any. The <code>args[]</code> array is accessed using an integer index, but each element is defined to be the type corresponding to the given probe argument. For example, if <code>args[]</code> is referenced by a <code>read()</code> system call probe, <code>args[0]</code> is of type <code>int</code>, <code>args[1]</code> is of type <code>void *</code>, and <code>args[2]</code> is of type <code>size_t</code>.</td>
</tr>
<tr>
<td><code>int64_t arg0, ... , arg9</code></td>
<td>The first ten input arguments to a probe represented as raw 64-bit integers. If fewer than ten arguments are passed to the current probe, the remaining variables return zero.</td>
</tr>
<tr>
<td><code>uintptr_t caller</code></td>
<td>The program counter location of the current kernel thread at the time the probe fired.</td>
</tr>
<tr>
<td><code>chipid_t chip</code></td>
<td>The CPU chip identifier for the current physical chip. See Section 11.6, “sched Provider” for more information.</td>
</tr>
<tr>
<td><code>processorid_t cpu</code></td>
<td>The CPU identifier for the current CPU. See Section 11.6, “sched Provider” for more information.</td>
</tr>
<tr>
<td><code>cpuinfo_t *curcpu</code></td>
<td>The CPU information for the current CPU. See Section 11.6, “sched Provider” for more information.</td>
</tr>
<tr>
<td><code>lwpsinfo_t *curlwpsinfo</code></td>
<td>The process state of the current thread. See Section 11.5, “proc Provider” for more information.</td>
</tr>
<tr>
<td><code>psinfo_t *curpsinfo</code></td>
<td>The process state of the process associated with the current thread. See Section 11.5, “proc Provider” for more information.</td>
</tr>
<tr>
<td><code>kthread_t *curthread</code></td>
<td>The address of the kernel's process descriptor (with type <code>struct task_struct</code>) for the current thread.</td>
</tr>
<tr>
<td><code>string cwd</code></td>
<td>The name of the current working directory of the process associated with the current thread.</td>
</tr>
<tr>
<td><code>uint_t epid</code></td>
<td>The enabled probe ID (EPID) for the current probe. This integer uniquely identifies a particular probe that is enabled with a specific predicate and set of actions.</td>
</tr>
<tr>
<td><code>int errno</code></td>
<td>The error value returned by the last system call executed by this thread.</td>
</tr>
<tr>
<td><code>string execname</code></td>
<td>The name that was passed to <code>execve()</code> to execute the current process.</td>
</tr>
<tr>
<td><code>fileinfo_t fds[]</code></td>
<td>The files that the current process has opened in an <code>fileinfo_t</code> array indexed by file descriptor number. See Section 11.7.5, “fileinfo_t”.</td>
</tr>
</tbody>
</table>

Note

You must load the `sdt` kernel module for `fds[]` to be available.

| `gid_t gid` | The real group ID of the current process. |
| `uint_t id` | The probe ID for the current probe. This ID is the system-wide unique identifier for the probe as published by DTrace and listed in the output of `dtrace -l`. |
| `uint_t ipl` | The interrupt priority level (IPL) on the current CPU at probe firing time. |
### External Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>lgrp_id_t lgrp</td>
<td>The latency group ID for the latency group of which the current CPU is a member. See Section 11.6, “sched Provider” for more information.</td>
<td>This value is non-zero if interrupts are firing and zero otherwise. The non-zero value depends on whether preemption is active (as well as other things), and can vary between kernel releases and kernel configurations.</td>
</tr>
<tr>
<td>pid_t pid</td>
<td>The process ID of the current process.</td>
<td></td>
</tr>
<tr>
<td>pid_t ppid</td>
<td>The parent process ID of the current process.</td>
<td></td>
</tr>
<tr>
<td>string probefunc</td>
<td>The function name portion of the current probe's description.</td>
<td></td>
</tr>
<tr>
<td>string probemod</td>
<td>The module name portion of the current probe's description.</td>
<td></td>
</tr>
<tr>
<td>string probename</td>
<td>The name portion of the current probe's description.</td>
<td></td>
</tr>
<tr>
<td>string probeprov</td>
<td>The provider name portion of the current probe's description.</td>
<td></td>
</tr>
<tr>
<td>psetid_t pset</td>
<td>The processor set ID for the processor set containing the current CPU. See Section 11.6, “sched Provider” for more information.</td>
<td>This value is always zero.</td>
</tr>
<tr>
<td>string root</td>
<td>The name of the root directory of the process associated with the current thread.</td>
<td></td>
</tr>
<tr>
<td>uint_t stackdepth</td>
<td>The current thread's stack frame depth at probe firing time.</td>
<td></td>
</tr>
<tr>
<td>id_t tid</td>
<td>The task ID of the current thread.</td>
<td></td>
</tr>
<tr>
<td>uint64_t timestamp</td>
<td>The current value of a nanosecond timestamp counter. This counter increments from an arbitrary point in the past and should only be used for relative computations.</td>
<td></td>
</tr>
<tr>
<td>uintptr_t ucaller</td>
<td>The program counter location of the current user thread at the time the probe fired.</td>
<td></td>
</tr>
<tr>
<td>uid_t uid</td>
<td>The real user ID of the current process.</td>
<td></td>
</tr>
<tr>
<td>uint64_t vtimestamp</td>
<td>The current value of a nanosecond timestamp counter that is virtualized to the amount of time that the current thread has been running on a CPU, minus the time spent in DTrace predicates and actions. This counter increments from an arbitrary point in the past and should only be used for relative time computations.</td>
<td></td>
</tr>
<tr>
<td>int64_t walltime</td>
<td>The current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.</td>
<td></td>
</tr>
</tbody>
</table>

Functions built into the D language such as `trace` are discussed in [Chapter 4, Actions and Subroutines](#).

#### 2.9.6 External Variables

D uses the backquote character (`) as a special scoping operator for accessing variables that are defined in the operating system and not in your D program. For example, the Oracle Linux kernel contains a C declaration of a system variable named `max_pfn`. This variable is declared in C in the kernel source code as follows:

```c
unsigned long max_pfn
```
To access the value of this variable in a D program, use the D notation:

\`max_pfn

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

When you access external variables from a D program, you are accessing the internal implementation details of another program such as the operating system kernel or its device drivers. These implementation details do not form a stable interface upon which you can rely. Any D programs you write that depend on these details might cease to work when you next upgrade the corresponding piece of software. For this reason, external variables are typically used to debug performance or functionality problems using DTrace. To learn more about the stability of your D programs, refer to Chapter 16, Stability.

Kernel symbol names are kept in a separate namespace from D variable and function identifiers, so you need not worry about these names conflicting with your D variables. When you prefix a variable with a backquote, the D compiler searches the known kernel symbols and uses the list of loaded modules to find a matching variable definition. Because the Oracle Linux kernel supports dynamically loaded modules with separate symbol namespaces, the same variable name might be used more than once in the active operating system kernel. You can resolve these name conflicts by specifying the name of the kernel module whose variable should be accessed prior to the backquote in the symbol name. For example, to refer to the address of the _bar function provided by a kernel module named foo, you would write:

\texttt{foo\_bar}

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

### 2.10 Pointers and Arrays

Pointers are memory addresses of data objects in the operating system kernel or in the address space of a user process. D provides the ability to create and manipulate pointers and store them in variables and associative arrays. This chapter describes the D syntax for pointers, operators that can be applied to create or access pointers, and the relationship between pointers and fixed-size scalar arrays. Also discussed are issues relating to the use of pointers in different address spaces.

\begin{note}
If you are an experienced C or C++ programmer, you can skim most of this chapter as the D pointer syntax is the same as the corresponding ANSI C syntax. You should read Section 2.10.8, “Pointers to DTrace Objects” and Section 2.10.1, “Pointers and Addresses” as they describe features and issues that are specific to DTrace.
\end{note}

### 2.10.1 Pointers and Addresses

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

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

```d
int *p;
```

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

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

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

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

```d
p = &`max_pfn; trace(*p);
trace(`max_pfn);
```

The first fragment creates a D global variable `p`. Because the `max_pfn` object is of type `unsigned long`, the type of the result of `&` `max_pfn` is `unsigned long *` (that is, pointer to `unsigned long`). Tracing the value of `*p` follows the pointer back to the data object `max_pfn`. This fragment is therefore the same as the second fragment, which directly traces the value of the data object by using its name.

### 2.10.2 Pointer Safety

If you are a C or C++ programmer, you may be a bit frightened after reading the previous section because you know that misuse of pointers in your programs can cause your programs to crash. DTrace is a robust, safe environment for executing your D programs where these mistakes cannot cause program crashes. You may indeed write a buggy D program, but invalid D pointer accesses do not cause DTrace or the operating system kernel to fail or crash in any way. Instead, the DTrace software detects any invalid pointer accesses, disables your instrumentation, and reports the problem back to you for debugging.

If you have programmed in the Java programming language, you probably know that the Java language does not support pointers for precisely the same reasons of safety. Pointers are needed in D because they are an intrinsic part of the operating system's implementation in C, but DTrace implements the same kind of safety mechanisms found in the Java programming language that prevent buggy programs from damaging themselves or each other. DTrace's error reporting is similar to the run-time environment for the Java programming language that detects a programming error and reports an exception back to you.

To see DTrace's error handling and reporting, write a deliberately bad D program using pointers. In an editor, type the following D program and save it in a file named `badptr.d`:

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

```
# dtrace -s badptr.d
```

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

The other problem that can arise from programs that use invalid pointers is an **alignment error**. Alignment errors in D usually indicate that your pointer has an invalid or corrupt value due to a bug in your D program.

For details about the DTrace error mechanism, see Section 11.1.3, “ERROR Probe”.

### 2.10.3 Array Declarations and Storage

D supports **scalar arrays** in addition to the dynamic associative arrays described in Section 2.9, “Variables”. Scalar arrays are a fixed-length group of consecutive memory locations that each store a value of the same type. Scalar arrays are accessed by referring to each location with an integer starting from zero. Scalar arrays correspond directly in concept and syntax with arrays in C and C++. Scalar arrays are not used as frequently in D as associative arrays and their more advanced counterparts *aggregations*, but you might need to use them to access existing operating system array data structures declared in C. Aggregations are described in Chapter 3, *Aggregations*.

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

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

**Figure 2.2, “Scalar Array Representation”** shows a visual representation of the array storage:

**Figure 2.2 Scalar Array Representation**

```
```

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

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

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

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

### 2.10.4 Pointer and Array Relationship

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

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

In the first fragment, the pointer \(p\) is assigned to the address of the first array element in \(a\) by applying the \& operator to the expression \(a[0]\). The expression \(p[2]\) traces the value of the third array element (index 2). Because \(p\) now contains the same address associated with \(a\), this expression yields the same value as \(a[2]\), shown in the second fragment. One consequence of this equivalence is that C and D permit you to access any index of any pointer or array. Array bounds checking is not performed for you by the compiler or DTrace runtime environment. If you access memory beyond the end of an array's predefined value, you either get an unexpected result or DTrace reports an invalid address error, as shown in the previous example. As always, you cannot damage DTrace itself or your operating system, but you do need to debug your D program.

The difference between pointers and arrays is that a pointer variable refers to a separate piece of storage that contains the integer address of some other storage. An array variable names the array storage itself, not the location of an integer that in turn contains the location of the array. Figure 2.3, “Pointer and Array Storage” illustrates this difference.

#### Figure 2.3 Pointer and Array Storage

![Diagram showing pointer and array storage](image)

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

### 2.10.5 Pointer Arithmetic

Since pointers are just integers used as addresses of other objects in memory, D provides a set of features for performing arithmetic on pointers. However, pointer arithmetic is not identical to integer arithmetic.
Generic Pointers

Pointer arithmetic implicitly adjusts the underlying address by multiplying or dividing the operands by the size of the type referenced by the pointer. The following D fragment illustrates this property:

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

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

Since `x` is a pointer to an `int` (size 4 bytes), incrementing `x` adds 4 to the underlying pointer value. This property is useful when using pointers to refer to consecutive storage locations such as arrays. For example, if `x` were assigned to the address of an array `a` like the one shown in Figure 2.3, “Pointer and Array Storage”, the expression `x + 1` would be equivalent to the expression `&a[1]`. Similarly, the expression `*(x + 1)` would refer to the value `a[1]`. Pointer arithmetic is implemented by the D compiler whenever a pointer value is incremented using the `+`, `++`, or `+=` operators. Pointer arithmetic is also applied when an integer is subtracted from a pointer on the left-hand side, when a pointer is subtracted from another pointer, or when the `--` operator is applied to a pointer. For example, the following D program would trace the result 2:

```d
int *x, *y;
int a[5];
BEGIN
{
  x = &a[0];
  y = &a[2];
  trace(y - x);
}
```

### 2.10.6 Generic Pointers

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

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

### 2.10.7 Multi-Dimensional Arrays

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

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

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

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

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

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

### 2.10.8 Pointers to DTrace Objects

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

In ANSI C, pointers can also be used to perform indirect function calls or to perform assignments, such as placing an expression using the unary * dereference operator on the left-hand side of an assignment operator. In D, these types of expressions using pointers are not permitted. You may only assign values directly to D variables using their name or by applying the array index operator [ ] to a D scalar or associative array. You may only call functions defined by the DTrace environment by name as specified in Chapter 4, *Actions and Subroutines*. Indirect function calls using pointers are not permitted in D.

### 2.10.9 Pointers and Address Spaces

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

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

To access user process memory from a DTrace probe, you must apply one of the `copyin`, `copyinstr`, or `copyinto` functions described in Chapter 4, *Actions and Subroutines* to the user address space pointer. Take care when writing your D programs to name and comment variables storing user addresses
appropriately to avoid confusion. You can also store user addresses as `uintptr_t` so you do not accidentally compile D code that dereferences them. Techniques for using DTrace on user processes are described in Chapter 12, User Process Tracing.

### 2.11 Strings

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

#### 2.11.1 String Representation

Strings are represented in DTrace as an array of characters terminated by a null byte (that is, a byte whose value is zero, usually written as `\0`). The visible part of the string is of variable length, depending on the location of the null byte, but DTrace stores each string in a fixed-size array so that each probe traces a consistent amount of data. Strings may not exceed the length of this predefined string limit, but the limit can be modified in your D program or on the `dtrace` command line by tuning the `strsize` option. Refer to Chapter 10, Options and Tunables for more information on tunable DTrace options. The default string limit is 256 bytes.

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

```d
trace(s);
```

if `s` is of type `char *`, DTrace traces the value of the pointer `s` (that is, it traces an integer address value). In the D statement:

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

by the definition of the `*` operator, the D compiler dereferences the pointer `s` and traces the single character at that location. These behaviors allow you to manipulate character pointers that refer to either single characters, or to arrays of byte-sized integers that are not strings and do not end with a null byte. In the D statement:

```d
trace(s);
```

if `s` is of type `string`, the string type indicates to the D compiler that you want DTrace to trace a null terminated string of characters whose address is stored in the variable `s`. You can also perform lexical comparison of expressions of type string, as described in Section 2.11.5, “String Comparison”.

#### 2.11.2 String Constants

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

A string constant may not contain a literal newline character. To create strings containing newlines, use the `\n` escape sequence instead of a literal newline. String constants may also contain any of the
special character escape sequences defined for character constants (see Table 2.6, "Character Escape Sequences").

### 2.11.3 String Assignment

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

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

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

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

### 2.11.4 String Conversion

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

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

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

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

### 2.11.5 String Comparison

D overloads the binary relational operators and permits them to be used for string comparisons as well as integer comparisons. The relational operators perform string comparison whenever both operands are of type `string`, or when one operand is of type `string` and the other operand can be promoted to type `string`, as described in Section 2.11.3, "String Assignment". Table 2.14, "D Relational Operators for Strings" lists the relational operators that can be used to compare strings.

#### Table 2.14 D Relational Operators for Strings

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Left-hand operand is less than right-operand</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Left-hand operand is less than or equal to right-hand operand</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Left-hand operand is greater than right-operand</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Left-hand operand is greater than or equal to right-hand operand</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>==</td>
<td>Left-hand operand is equal to right-hand operand</td>
</tr>
<tr>
<td>!=</td>
<td>Left-hand operand is not equal to right-hand operand</td>
</tr>
</tbody>
</table>

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

The relational operators compare the two input strings byte-by-byte, similar to the C library routine `strcmp()`. Each byte is compared using its corresponding integer value in the ASCII character set, as shown in the `ascii(7)` manual page, until a null byte is read or the maximum string length is reached. Some example D string comparisons and their results are:

<table>
<thead>
<tr>
<th>D string comparison</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;coffee&quot; &lt; &quot;espresso&quot;</td>
<td>Returns 1 (true)</td>
</tr>
<tr>
<td>&quot;coffee&quot; == &quot;coffee&quot;</td>
<td>Returns 1 (true)</td>
</tr>
<tr>
<td>&quot;coffee&quot; &gt;= &quot;mocha&quot;</td>
<td>Returns 0 (false)</td>
</tr>
</tbody>
</table>

Note
Seemingly identical Unicode strings might compare as being different if one or the other of the strings is not normalized.

### 2.12 Structs and Unions

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

#### 2.12.1 Structs

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

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

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

However, this clause is inefficient because DTrace must create three separate associative arrays and store separate copies of the identical tuple values corresponding to `probefunc` for each one. Instead, you can
conserves space and makes your program easier to read and maintain by using a struct. First, declare a new `struct` type at the top of the D program source file:

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

The `struct` keyword is followed by an optional identifier used to refer back to our new type, which is now known as `struct callinfo`. The struct members are then enclosed in a set of braces `{}` and the entire declaration is terminated by a semicolon (`;`). Each struct member is defined using the same syntax as a D variable declaration, with the type of the member listed first followed by an identifier naming the member and another semicolon (`;`).

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

You can use the member identifier names to access the individual member values using the “. .” operator by writing an expression of the form:

`variable-name.member-name`

The following example is an improved program using the new structure type. Go to your editor, type in the following D program and save it in a file named `rwinfo.d`:

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

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

syscall::read:entry, syscall::write:entry
/pid == $1/
{
    i[probefunc].ts = timestamp;
    i[probefunc].calls++;
    i[probefunc].maxbytes = arg2 > i[probefunc].maxbytes ?
                           arg2 : i[probefunc].maxbytes;
}

syscall::read:return, syscall::write:return
/i[probefunc].ts != 0 && pid == $1/
{
    i[probefunc].elapsed += timestamp - i[probefunc].ts;
}

END
{
    printf("      calls max bytes elapsed nsecs\n");
    printf("-------- ------ --------- ---------------\n");
    printf("  read %5d %9d %d\n", i["read"].calls, i["read"].maxbytes, i["read"].elapsed);
    printf("  write %5d %9d %d\n", i["write"].calls, i["write"].maxbytes, i["write"].elapsed);
    printf("\n");
}`
```
After you have typed in the program, run `dtrace -q -s rwinfo.d`, specifying one of your shell processes. Then type in a few commands in your shell and, when you have finished entering your shell commands, type `Ctrl-C` to fire the END probe and print the results, for example:

```
# dtrace -q -s rwinfo.d `pgrep -n bash`
^C
```
calls max bytes elapsed nsecs
------ ----- --------- -------------
read  25 1024 8775036488
write 33   22 1859173
#
```

### 2.12.2 Pointers to Structs

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

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

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

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

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

<table>
<thead>
<tr>
<th>Example Expression</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>curpsinfo-&gt;pr_pid</code></td>
<td><code>pid_t</code></td>
<td>Current process ID</td>
</tr>
<tr>
<td><code>curpsinfo-&gt;pr_fname</code></td>
<td><code>char []</code></td>
<td>Executable file name</td>
</tr>
<tr>
<td><code>curpsinfo-&gt;pr_psargs</code></td>
<td><code>char []</code></td>
<td>Initial command-line arguments</td>
</tr>
</tbody>
</table>

For more information, see Section 11.5.4, “psinfo_t”.

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

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

This clause uses the expression `curpsinfo->pr_fname` to access and match the command name so that the script selects the correct `write()` requests before tracing the arguments. Notice that by using operator `==` with a left-hand argument that is an array of char and a right-hand argument that is a string, the D compiler infers that the left-hand argument should be promoted to a string and a string comparison should be performed. Enter the command `dtrace -q -s procs.d` in one shell, and the `date` command several times in another shell. You see output from DTrace similar to the following example:

```
# dtrace -q -s procfs.d
```
Complex data structures are used frequently in C programs, so the ability to describe and reference structs from D also provides a powerful capability for observing the inner workings of the Oracle Linux operating system kernel and its system interfaces.

2.12.3 Unions

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

2.12.4 Member Sizes and Offsets

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

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

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

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

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

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

2.12.5 Bit-Fields

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

```
struct s
{
    int a : 1;
    int b : 3;
}
Type and Constant Definitions

```c
int c : 12;
```

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

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

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

2.13 Type and Constant Definitions

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

2.13.1 typedefs

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

```c
typedef existing-type new-type;
```

where `existing-type` is any type declaration and `new-type` is an identifier to be used as the alias for this type. For example, the D compiler uses the following declaration internally to create the `uint8_t` type alias:

```c
typedef unsigned char uint8_t;
```

You can use type aliases anywhere that a normal type can be used, such as the type of a variable or associative array value or tuple member. You can also combine `typedef` with more elaborate declarations such as the definition of a new `struct`, for example:

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

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

2.13.2 Enumerations

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

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

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

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

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

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

The D enumeration syntax is the same as the corresponding syntax in ANSI C. D also provides access to enumerations defined in the operating system kernel and its loadable modules, but these enumerators are not globally visible in your D program. Kernel enumerators are only visible if you specify one as an argument in a comparison with an object of the corresponding enumeration type. This feature protects your D programs against inadvertent identifier name conflicts with the large collection of enumerations defined in the operating system kernel.

The following example D program, which displays information about I/O requests, uses the enumerators `B_READ` and `B_WRITE` to differentiate between read and write operations.

```d
io:::done,
io:::start,
io:::wait-done,
io:::wait-start
{
    printf("%8s %10s: %d %16s (%s size %d @ sect %d)\n",
        args[1]->dev_statname, probename,
        timestamp & 1000000000, execname,
        args[0]->b_flags & B_READ ? "R" :
        args[0]->b_flags & B_WRITE ? "W" : "?",
        args[0]->b_bcount, args[0]->b_blkno);
}
```

### 2.13.3 Inlines

D named constants can also be defined using `inline` directives, which provide a more general means of creating identifiers that are replaced by predefined values or expressions during compilation. Inline directives are a more powerful form of lexical replacement than the `#define` directive provided by the C preprocessor because the replacement is assigned an actual type and is performed using the compiled
Type Namespaces

syntax tree and not simply a set of lexical tokens. An inline directive is specified using a declaration of the form:

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

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

```
inline string hello = "hello";
inline int number = 100 + 23;
BEGIN
    trace(hello);
    trace(number);
END
```

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

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

The DTrace software packages install a number of D source files in the system directory /usr/lib64/dtrace containing inline directives you can use in your D programs. For example, the signal.d library includes directives of the form:

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

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

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

2.13.4 Type Namespaces

This section discusses D namespaces and namespace issues related to types. In traditional languages such as ANSI C, type visibility is determined by whether a type is nested inside of a function or other declaration. Types declared at the outer scope of a C program are associated with a single global namespace and are visible throughout the entire program. Types defined in C header files are typically included in this outer scope. Unlike these languages, D provides access to types from multiple outer scopes.

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

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

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

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

```d
struct foo.bar
foo.bar_t
```

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

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

The D type namespace is initially populated with the D type intrinsics such as `int` and `string` as well as the built-in D type aliases such as `uint64_t`. Any new type declarations that appear in the D program source are automatically added to the D type namespace. If you create a complex type such as a `struct` in your D program consisting of member types from other namespaces, the member types are copied into the D namespace by the declaration.

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

The D compiler uses compressed ANSI C debugging information provided with the core Oracle Linux kernel modules in order to automatically access the types associated with the operating system source code without the need for accessing the corresponding C include files. This symbolic debugging information might not be available for all kernel modules on your system. The D compiler will report an error if you attempt to access a type within the namespace of a module that lacks compressed C debugging information intended for use with DTrace.
Chapter 3 Aggregations

Table of Contents

3.1 Aggregating Functions ................................................................. 53
3.2 About Aggregations ................................................................. 54
3.3 Printing Aggregations ............................................................... 61
3.4 Data Normalization ................................................................. 61
3.5 Clearing Aggregations ............................................................. 64
3.6 Truncating Aggregations .......................................................... 64
3.7 Minimizing Drops ................................................................. 65

When instrumenting the system to answer performance-related questions, it is useful to consider how
data can be aggregated to answer a specific question rather than thinking in terms of data gathered by
individual probes. For example, if you wanted to know the number of system calls by user ID, you would
not necessarily care about the datum collected at each system call. You simply want to see a table of user
IDs and system calls. Historically, you would answer this question by gathering data at each system call,
and postprocessing the data using a tool like awk or perl. However, in DTrace the aggregating of data is
a first-class operation. This chapter describes the DTrace facilities for manipulating aggregations.

3.1 Aggregating Functions

An aggregating function is one that has the following property:

\[ \text{func(} \text{func}(x_0) \ U \ \text{func}(x_1) \ U \ ... \ U \ \text{func}(x_n) \text{)} = \text{func}(x_0 \ U \ x_1 \ U \ ... \ U \ x_n) \]

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

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

Many common functions for understanding a set of data are aggregating functions. These functions include
counting the number of elements in the set, computing the minimum value of the set, computing the
maximum value of the set, and summing all elements in the set. Determining the arithmetic mean of the set
can be constructed from the function to count the number of elements in the set and the function to sum
the elements in the set.
However, several useful functions are not aggregating functions. These functions include computing the mode (the most common element) of a set, the median value of the set, or the standard deviation of the set.

Applying aggregating functions to data as it is traced has a number of advantages:

- The entire data set need not be stored. Whenever a new element is to be added to the set, the aggregating function is calculated given the set consisting of the current intermediate result and the new element. After the new result is calculated, the new element may be discarded. This process reduces the amount of storage required by a factor of the number of data points, which is often quite large.

- Data collection does not induce pathological scalability problems. Aggregating functions enable intermediate results to be kept per-CPU instead of in a shared data structure. DTrace then applies the aggregating function to the set consisting of the per-CPU intermediate results to produce the final system-wide result.

### 3.2 About Aggregations

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

```
@name{ keys } = aggfunc( args );
```

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

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

### Table 3.1 DTrace Aggregating Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>None</td>
<td>The number of times called.</td>
</tr>
<tr>
<td>sum</td>
<td>Scalar expression</td>
<td>The total value of the specified expressions.</td>
</tr>
<tr>
<td>avg</td>
<td>Scalar expression</td>
<td>The arithmetic average of the specified expressions.</td>
</tr>
<tr>
<td>min</td>
<td>Scalar expression</td>
<td>The smallest value among the specified expressions.</td>
</tr>
<tr>
<td>max</td>
<td>Scalar expression</td>
<td>The largest value among the specified expressions.</td>
</tr>
<tr>
<td>stddev</td>
<td>Scalar expression</td>
<td>The standard deviation of the specified expressions.</td>
</tr>
<tr>
<td>lquantize</td>
<td>Scalar expression, lower bound, upper bound, step value</td>
<td>A linear frequency distribution, sized by the specified range, of the values of the specified expressions. Increments the value in the <code>highest</code> bucket that is <code>less</code> than the specified expression.</td>
</tr>
<tr>
<td>quantize</td>
<td>Scalar expression</td>
<td>A power-of-two frequency distribution of the values of the specified expressions. Increments the value in the <code>highest</code> power-of-two bucket that is <code>less</code> than the specified expression.</td>
</tr>
</tbody>
</table>
For example, to count the number of `write()` system calls in the system, you could use an informative string as a key and the `count` aggregating function:

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

The `dtrace` command prints aggregation results by default when the process terminates, either as the result of an explicit `END` action or when you press `Ctrl-C`. The following example output shows the result of running this command, waiting for a few seconds, and pressing `Ctrl-C`:

```plaintext
# dtrace -s writes.d
dtrace: script './writes.d' matched 1 probe
^C
write system calls                               179
```

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

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

The following example output shows the result of running this command, waiting for a few seconds, and pressing `Ctrl-C`:

```plaintext
# dtrace -s writesbycmd.d
dtrace: script 'writesbycmd.d' matched 1 probe
^C
dirname                                                           1
dtrace                                                            1
gnome-panel                                                       1
mozilla-xremote                                                   1
ps                                                                1
avahi-daemon                                                      2
basename                                                          2
gconfd-2                                                          2
java                                                              2
master                                                            2
pickup                                                            2
qmgr                                                              2
sed                                                               2
dbus-daemon                                                       3
rtkit-daemon                                                      3
uname                                                             3
w                                                                 5
bash                                                              9
cat                                                               9
gnome-session                                                     9
Xorg                                                              21
firefox                                                           149
gnome-terminal                                                   9421
```

Alternatively, you might want to further examine writes organized by both executable name and file descriptor. The file descriptor is the first argument to `write()`, so the following example uses a key consisting of both `execname` and `arg0`:

```plaintext
syscall::write:entry
{   @counts[execname, arg0] = count();
}
Running this command results in a table with both executable name and file descriptor, as shown in the following example:

```bash
# dtrace -s writesbycmdfd.d
dtrace: script 'writesbycmdfd.d' matched 1 probe
^C
basename                                                  1        1
dbus-daemon                                              70        1
dircolors                                                 1        1
dtrace                                                    1        1
gnome-panel                                               35        1
gnome-terminal                                            16        1
gnome-terminal                                            18        1
init                                                      4        1
master                                                   89        1
ps                                                        1        1
pulseaudio                                               20        1
tput                                                      1        1
Xorg                                                      2        2
#
```

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

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

The following example output shows the result of running this command, waiting for a few seconds, and pressing `Ctrl-C`:

```bash
# dtrace -s writetime.d
dtrace: script 'writetime.d' matched 2 probes
^C
gnome-session                                                  8260
udisks-part-id                                                 9279
gnome-terminal                                                9378
mozilla-xremote                                               10061
abrt-handle-eve                                               13414
vgdisplay                                                     13459
avahi-daemon                                                  14043
vgscan                                                        14190
uptime                                                        14533
lsof                                                          14903
ip                                                             15075
date                                                          15371
...
ps                                                             91792
sestatus                                                      98374
pstree                                                       102566
sysctl                                                       175427
iptables                                                     192835
udisks-daemon                                                250405
python                                                       282544
dbus-daemon                                                 491069
```

---

About Aggregations
The average can be useful, but often does not provide sufficient detail to understand the distribution of data points. To understand the distribution in further detail, use the `quantize` aggregating function as shown in the following example:

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

Because each line of output becomes a frequency distribution diagram, the output of this script is substantially longer than previous ones. The following example shows a selection of sample output:

```bash
# dtrace -s wrquantize.d
```

```
...  
...  
bash  
value ------------- Distribution ------------- count
     8192    |                                      | 0
     16384  |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 4
     32768  |                                          | 0
     65536  |                                      | 0
     131072 |@@@@@@@@@@@@@@|                          | 1
     262144 |                                      | 0

gnome-terminal
value ------------- Distribution ------------- count
     4096    |                                      | 0
     8192    |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 5
     16384   |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 5
     32768   |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 4
     65536   |@@|                                  | 1
     131072  |@@@@@@@@@@@@@@|                          | 0

Xorg
value ------------- Distribution ------------- count
     2048    |                                      | 0
     4096    |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 4
     8192    |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 8
     16384   |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@| 7
     32768   |@@|                                  | 2
     65536   |@@|                                  | 1
     131072  |@@@@@@@@@@@@@@|                          | 0
     262144  |                                      | 0
     524288  |                                      | 0
     1048576 |                                      | 0
     2097152 |@@|                                  | 2
     4194304 |                                      | 0

firefox
value ------------- Distribution ------------- count
     2048    |                                      | 0
     4096    |@@|                                  | 22
```
The rows for the frequency distribution are always power-of-two values. Each row indicates the count of the number of elements greater than or equal to the corresponding value, but less than the next larger row value. For example, the above output shows that firefox had 107 writes taking between 16,384 nanoseconds and 32,767 nanoseconds, inclusive.

While quantize is useful for getting quick insight into the data, you might want to examine a distribution across linear values instead. To display a linear value distribution, use the lquantize aggregating function. The lquantize function takes three arguments in addition to a D expression: a lower bound, an upper bound, and a step. For example, if you wanted to look at the distribution of writes by file descriptor, a power-of-two quantization would not be effective. Instead, use a linear quantization with a small range, as shown in the following example:

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

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

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

# gnome-session
time  ------------ Distribution ------------ count
25 |                                       0
26 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 9
27 |                                       0

# gnome-terminal
time  ------------ Distribution ------------ count
15 |                                       0
16 | @@                                      1
17 |                                       0
18 |                                       0
19 |                                       0
20 |                                       0
21 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 6
22 | @@                                      1
23 | @@                                      1
24 |                                       0
25 |                                       0
26 |                                       0
27 |                                       0
28 |                                       0
29 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 6
30 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 6
31 |                                       0
```

You can also use the lquantize aggregating function to aggregate on time since some point in the past. This technique allows you to observe a change in behavior over time. The following example displays the change in system call behavior over the lifetime of a process executing the date command:
syscall::execve: return
/execname == "date"/
{
    self->start = timestamp;
}

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

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

This script provides greater insight into system call behavior when many date processes are executed. To see this result, run

```
s -c 'while true; do date >/dev/null; done'
```

in one window, while executing the D script in another. The script produces a profile of the system call behavior of the date command:

```
# dtrace -s dateprof.d
dtrace: script 'dateprof.d' matched 298 probes
^C

system calls over time

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>@@</td>
<td>23428</td>
</tr>
<tr>
<td>100</td>
<td>@@@@@@@@@@</td>
<td>56263</td>
</tr>
<tr>
<td>200</td>
<td>@@@@@@@@@</td>
<td>61271</td>
</tr>
<tr>
<td>300</td>
<td>@@@@@@@@@</td>
<td>58132</td>
</tr>
<tr>
<td>400</td>
<td>@@@@@@@@@</td>
<td>54617</td>
</tr>
<tr>
<td>500</td>
<td>@@@@@@@@@</td>
<td>45454</td>
</tr>
<tr>
<td>600</td>
<td>@@@@@@@@@</td>
<td>26049</td>
</tr>
<tr>
<td>700</td>
<td>@@@@@@@@@</td>
<td>38859</td>
</tr>
<tr>
<td>800</td>
<td>@@@@@@@@@</td>
<td>51569</td>
</tr>
<tr>
<td>900</td>
<td>@@@@@@@@@</td>
<td>42553</td>
</tr>
<tr>
<td>1000</td>
<td>@</td>
<td>11339</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>4020</td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td>2236</td>
</tr>
<tr>
<td>1300</td>
<td></td>
<td>1264</td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td>812</td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>706</td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td>764</td>
</tr>
<tr>
<td>1700</td>
<td></td>
<td>586</td>
</tr>
<tr>
<td>1800</td>
<td></td>
<td>266</td>
</tr>
<tr>
<td>1900</td>
<td></td>
<td>155</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>2100</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>2200</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>2300</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>2400</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>2600</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>2700</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>
```
This output provides a rough idea of the different phases of the `date` command with respect to the services required of the kernel. To better understand these phases, you might want to understand which system calls are being called when. If so, you could change the D script to aggregate on the variable `probefunc` instead of a constant string.

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

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

syscall::execve:return
/ self->ts /
{
    t = timestamp - self->ts;
    @execavg[probefunc] = avg(t);
    @execsd[probefunc] = stddev(t);
    self->ts = 0;
}

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

With sample output as follows:

```
# dtrace -q -s stddev.d
^C
AVERAGE:
    execve             253839

STDDEV:
    execve             260226
```

**Note**

The standard deviation is approximated as \( \sqrt{((\Sigma x^2)/N) - (\Sigma x/N)^2} \). This is an imprecise approximation, but it should suffice for most purposes to which DTrace is put.
3.3 Printing Aggregations

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

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

3.4 Data Normalization

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

The following example shows how to aggregate data by system call:

```d
#pragma D option quiet
BEGIN
{
   /*
    * Get the start time, in nanoseconds.
    */
    start = timestamp;
}
syscall:::entry
{
    @func[execname] = count();
}
END
{
    /*
     * Normalize the aggregation based on the number of seconds we have
     * been running. (There are 1,000,000,000 nanoseconds in one second.)
     */
    normalize(@func, (timestamp - start) / 1000000000);
}
```

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

```d
memballoon                                                        1
udisks-daemon                                                     1
vmstats                                                           1
```
normalize sets the normalization factor for the specified aggregation, but this action does not modify the underlying data. denormalize takes only an aggregation. Adding the denormalize action to the preceding example returns both raw system call counts and per-second rates:

```bash
#pragma D option quiet
BEGIN
    start = timestamp;
}
syscall:::entry
{
    @func[execname] = count();
}
END
{
    this->seconds = (timestamp - start) / 1000000000;
    printf("Ran for %d seconds.\n", this->seconds);
    printf("Per-second rate: \n\n");
    normalize(@func, this->seconds);
    printa(@func);
    printf("\nRaw counts:\n");
    denormalize(@func);
    printa(@func);
}
```

Running the above script for a brief period of time produces output similar to the following example:

```
# dtrace -s denorm.d
^C
Ran for 7 seconds.
Per-second rate:
  audispd                        0
  auditd                        0
  memballoon                     0
  rtkit-daemon                   0
  timesync                       1
  gnome-power-man                1
  vmstats                        1
  automount                     2
  udisks-daemon                  2
  gnome-panel                    2
  metacity                       2
  gnome-settings-qpidd           3
```
Aggregations can also be renormalized. If `normalize` is called more than once for the same aggregation, the normalization factor is the factor specified in the most recent call. The following example prints per-second rates over time:

```c
#pragma D option quiet
BEGIN
{  
  start = timestamp;
}
syscall:::entry
{  
  @func[execname] = count();
}
tick-10sec
{  
  normalize(@func, (timestamp - start) / 1000000000);
  printa(@func);
}```
3.5 Clearing Aggregations

When using DTrace to build simple monitoring scripts, you can periodically clear the values in an aggregation using the `clear` function. This function takes an aggregation as its only parameter. The `clear` function clears only the aggregation's values; the aggregation's keys are retained. Therefore, the presence of a key in an aggregation that has an associated value of zero indicates that the key had a non-zero value that was subsequently set to zero as part of a `clear`. To discard both an aggregation's values and its keys, use `trunc`. See Section 3.6, “Truncating Aggregations” for details.

The following example uses `clear` to show the system call rate only for the most recent ten-second period:

```bash
#pragma D option quiet
BEGIN
    {
        last = timestamp;
    }
syscall:::entry
    {
        @func[execname] = count();
    }
tick-10sec
    {
        normalize(@func, (timestamp - last) / 1000000000);
        printa(@func);
        clear(@func);
        last = timestamp;
    }
```

3.6 Truncating Aggregations

When looking at aggregation results, you often care only about the top several results. The keys and values associated with anything other than the highest values are not interesting. You might also wish to discard an entire aggregation result, removing both keys and values. The DTrace `trunc` function is used for both of these situations.

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

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

The following example displays only the per-second system call rates of the top ten system-calling applications in a ten-second period:

```bash
#pragma D option quiet
BEGIN
    {
        last = timestamp;
    }
syscall:::entry
```
{  
  @func[execname] = count();
}

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

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

```
# dtrace -s truncagg.d
^C
memballoon                                                        4
rtkit-daemon                                                      6
vmstats                                                           8
automount                                                        12
udisks-daemon                                                    12
gnome-panel                                                      20
gnome-settings-                                                  28
gvfs-afc-volume                                                  30
metacity                                                         30
qpidd                                                            34
hald                                                             38
gnome-terminal                                                   54
hald-addon-inpu                                                  183
VBoxClient                                                      240
Xorg                                                            354
X11-NOTIFY                                                      476
java                                                            666
dtrace                                                         1655
sh                                                           167108
date                                                         312258
```

### 3.7 Minimizing Drops

Because DTrace buffers some aggregation data in the kernel, space might not be available when a new key is added to an aggregation. In this case, the data will be dropped, a counter will be incremented, and `dtrace` generates a message indicating an aggregation drop. This situation rarely occurs because DTrace keeps state information, consisting of the aggregation's key and intermediate results, at user level where space can grow dynamically. In the unlikely event that an aggregation drop occurs, you can increase the aggregation buffer size with the `aggsize` option to reduce the likelihood of drops. You can also use this option to minimize the memory footprint of DTrace. As with any size option, `aggsize` can be specified with any size suffix. The resizing policy of this buffer is dictated by the `bufresize` option. For more information about buffering, see Chapter 5, *Buffers and Buffering*. For more information about options, see Chapter 10, *Options and Tunables*.

An alternative method to eliminate aggregation drops is to increase the rate at which aggregation data is consumed at user level. This rate defaults to once per second, and may be explicitly tuned with the `aggrate` option. As with any rate option, `aggrate` can be specified with any time suffix, but defaults to rate-per-second. For more information about the `aggsize` option, see Chapter 10, *Options and Tunables*. 
Chapter 4 Actions and Subroutines

Table of Contents

4.1 Actions .................................................................................................................. 68
4.2 Default Action ......................................................................................................... 68
4.3 Data Recording Actions .......................................................................................... 69
  4.3.1 fopen ............................................................................................................... 69
  4.3.2 truncate ........................................................................................................... 69
  4.3.3 func ................................................................................................................ 69
  4.3.4 mod ................................................................................................................ 69
  4.3.5 printa .............................................................................................................. 70
  4.3.6 printf .............................................................................................................. 70
  4.3.7 stack ............................................................................................................... 70
  4.3.8 sym ................................................................................................................. 71
  4.3.9 trace ............................................................................................................... 71
  4.3.10 tracemem ...................................................................................................... 71
  4.3.11 usock ............................................................................................................. 72
  4.3.12 uaddr ............................................................................................................. 73
  4.3.13 usym ............................................................................................................. 73
4.4 Destructive Actions ................................................................................................. 73
  4.4.1 Process Destructive Actions ........................................................................... 73
  4.4.2 Kernel Destructive Actions ........................................................................... 75
4.5 Special Actions ....................................................................................................... 76
  4.5.1 Speculative Actions ....................................................................................... 76
  4.5.2 exit .................................................................................................................. 76
  4.5.3 setopt .............................................................................................................. 76
4.6 Subroutines ............................................................................................................. 77
  4.6.1 alloca .............................................................................................................. 77
  4.6.2 basename ........................................................................................................ 77
  4.6.3 bcopy ............................................................................................................. 77
  4.6.4 cleanpath ....................................................................................................... 77
  4.6.5 copyin ............................................................................................................. 77
  4.6.6 copyinstr ....................................................................................................... 78
  4.6.7 copyinto ........................................................................................................ 78
  4.6.8 d_path ............................................................................................................. 78
  4.6.9 dirname .......................................................................................................... 78
  4.6.10 getmajor ....................................................................................................... 78
  4.6.11 getminor ....................................................................................................... 78
  4.6.12 htonl ............................................................................................................ 79
  4.6.13 htonll ......................................................................................................... 79
  4.6.14 htons ........................................................................................................... 79
  4.6.15 index ............................................................................................................. 79
  4.6.16 inet_ntoa .................................................................................................... 79
  4.6.17 inet_ntoa6 .................................................................................................. 79
  4.6.18 inet_ntop .................................................................................................... 79
  4.6.19 Itostr ............................................................................................................ 79
  4.6.20 mutex_owned ............................................................................................... 80
  4.6.21 mutex_owner ............................................................................................... 80
  4.6.22 mutex_type_adaptive ................................................................................. 80
  4.6.23 mutex_type_spin ....................................................................................... 80
  4.6.24 ntohl ........................................................................................................... 80

67
4.1 Actions

Actions enable your DTrace programs to interact with the system outside of DTrace. The most common actions record data to a DTrace buffer. Other actions are available, such as stopping the current process, raising a specific signal on the current process, or ceasing tracing altogether. Some of these actions are destructive in that they change the system, albeit in a well-defined way. These actions may only be used if destructive actions have been explicitly enabled. By default, data recording actions record data to the principal buffer. For more information about the principal buffer and buffer policies, see Chapter 5, Buffers and Buffering.

4.2 Default Action

A clause can contain any number of actions and variable manipulations. If a clause is left empty, the default action is taken. The default action is to trace the enabled probe identifier (EPID) to the principal buffer. The EPID identifies a particular enabling of a particular probe with a particular predicate and actions. From the EPID, DTrace consumers can determine the probe that induced the action. Indeed, whenever any data is traced, it must be accompanied by the EPID to enable the consumer to make sense of the data. Therefore, the default action is to trace the EPID and nothing else.

Using the default action allows for simple use of dtrace. For example, the following command enables all probes in the vmlinux module with the default action:

```
# dtrace -m vmlinux
```

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

```
# dtrace -m vmlinux
dtrace: description 'vmlinux' matched 35 probes
CPU ID FUNCTION:NAME
  0 42 __schedule:sleep
  0 34 dequeue_task:dequeue
  0 40 __schedule:off-cpu
```
4.3 Data Recording Actions

The data recording actions comprise the core DTrace actions. Each of these actions records data to the principal buffer by default, but each action may also be used to record data to speculative buffers. See Chapter 5, Buffers and Buffering for more details on the principal buffer. See Chapter 7, Speculative Tracing for more details on speculative buffers. The descriptions in this section refer only to the directed buffer, indicating that data is recorded either to the principal buffer or to a speculative buffer if the action follows a speculate.

4.3.1 freopen

```c
void freopen(const char *pathname)
```

*freopen* changes the file that is associated with *stdout* to the file specified by *pathname*.

⚠️ Caution

*freopen* is potentially destructive, because you can use it to overwrite arbitrary files.

4.3.2 ftruncate

```c
void ftruncate(void)
```

*ftruncate* truncates the output stream on *stdout*.

4.3.3 func

```c
_symaddr func(uintptr_t address)
```

*func* prints the symbol that corresponds to a specified kernel-space address, for example: *vmlinux'*max_pfn.*func* is an alias for *sym*.

4.3.4 mod

```c
_symaddr mod(uintptr_t address)
```

*mod* prints the name of the module that corresponds to a specified kernel-space address, for example: *vmlinux*. 
### 4.3.5 printa

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

The `printa` action enables you to display and format aggregations. See Chapter 3, **Aggregations** for more detail on aggregations. If a `format` is not specified, `printa` traces only a directive to the DTrace consumer that the specified aggregation should be processed and displayed using the default format. If a `format` is specified, the aggregation is formatted. See Section 6.2, “printa” for a more detailed description of the `printa` format string.

`printa` traces only a **directive** that the aggregation should be processed by the DTrace consumer. It does not process the aggregation in the kernel. Therefore, the time between the tracing of the `printa` directive and the actual processing of the directive depends on the factors that affect buffer processing. These factors include the aggregation rate, the buffering policy and, if the buffering policy is **switching**, the rate at which buffers are switched. See Chapter 3, **Aggregations** and Chapter 5, **Buffers and Buffering** for detailed descriptions of the factors.

### 4.3.6 printf

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

Like `trace`, the `printf` action traces D expressions. However, `printf` allows for elaborate `printf`-style formatting. Like `printf`, the parameters consists of a `format` string followed by a variable number of arguments. By default, the arguments are traced to the directed buffer. The arguments are later formatted for output by `dtrace` according to the specified format string. For example, the first two examples of `trace` from Chapter 1, **About DTrace** could be combined in a single `printf`:

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

For more information on `printf`, see Section 6.1, “printf”.

### 4.3.7 stack

```c
void stack(int nframes)
void stack(void)
```

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

```bash
# dtrace -n gettimeofday:entry '{stack()}'
dtrace: description 'gettimeofday:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
0    196               gettimeofday:entry
     vmlinux`pollwake
     vmlinux`dtrace_stacktrace+0x30
     vmlinux`__brk_limit+0xle1832d7
     vmlinux`__brk_limit+0xle1913a1
     vmlinux`pollwake
     vmlinux`do_gettimeofday+0x1a
     vmlinux`ktime_get_ts+0xad
     vmlinux`sastrace_syscall+0xde
     vmlinux`audit_syscall_entry+0x1d7
     vmlinux`system_call_fastpath+0x16
0    196               gettimeofday:entry
     vmlinux`dtrace_stacktrace+0x30
     vmlinux`__brk_limit+0xle1832d7
     vmlinux`__brk_limit+0xle1913a1
     vmlinux`security_file_permission+0x8b
```
The **stack** action is different from other actions in that it can also be used as the key to an aggregation:

```bash
dtrace -n execve:entry'[@[stack()] = count()]'
dtrace: description 'execve:entry' matched 1 probe
```

4.3.8 **sym**

```c
_symaddr sym(uintptr_t address)
```

**sym** prints the symbol that corresponds to a specified kernel-space address, for example: `vmlinux'`max_pfn.sym` is an alias for **func**.

4.3.9 **trace**

```c
void trace(expression)
```

The most basic action is the **trace** action, which takes a D expression as its argument and traces the result to the directed buffer. The following statements are examples of **trace** actions:

```
trace(execname);
trace(curlwpsinfo->pr_pri);
trace(timestamp / 1000);
trace("lbolt");
```

4.3.10 **tracemem**

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

The **tracemem** action takes a D expression as its first argument, `address`, and a constant as its second argument, `nbytes`. **tracemem** copies the memory from the address specified by `address` into the directed buffer for the length specified by `nbytes`. 
The output format depends on the data printed. If `dtrace` decides that the data looks like an ASCII string, it prints it as text, and terminates the output with a null character (0). When `dtrace` decides that the data is binary, it prints it in hexadecimal form.

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

You can force `tracemem` to use always binary format by using the `rawbytes` option.

### 4.3.11 `ustack`

**Note**

If you want to perform symbol lookup in a stripped executable, you must specify the `--export-dynamic` option when linking the program. This option causes the linker to add all symbols to the dynamic symbol table (that is, the set of symbols which are visible from dynamic objects at run time). If you use `gcc` to link the objects, specify the option as `--export-dynamic` to pass the correct option to the linker.

If you want to look up symbols in shared libraries or unstripped executables, the `--export-dynamic` option is not required.

DTrace supports the use of `ustack` with both 32-bit and 64-bit binaries.

```c
void ustack(int nframes, int ssize)
void ustack(int nframes)
void ustack(void)
```

The `ustack` action records a user stack trace to the directed buffer. The user stack is `nframes` in depth. If `nframes` is not specified, the number of stack frames recorded is the number specified by the `ustackframes` option. While `ustack` is able to determine the address of the calling frames when the probe fires, the stack frames are not translated into symbols until the `ustack` action is processed at user level by the DTrace consumer. If `ssize` is specified and non-zero, `ustack` allocates the specified amount of string space, and use it to perform address-to-symbol translation directly from the kernel. Such direct user symbol translation is only used with stacktrace helpers that support this usage with DTrace. If such frames cannot be translated, the frames appear only as hexadecimal addresses.

The following example traces a stack with no address-to-symbol translation:

```bash
# dtrace -n syscall::write:entry'/pid == $target/{ustack();
 exit(0)}' -c)./mytestprog -v
```

```
dtrace: description 'syscall::write:entry' matched 1 probe
mytestprog (Version 1.0)
CPU    ID                    FUNCTION:NAME
  2      6                      write:entry
  mytestprog`printver+0x2f
  mytestprog`0x401338
  mytestprog`main+0xc7
  mytestprog`0x401338
  libc.so.6`__libc_start_main+0x9f
  mytestprog`main
  mytestprog`0x400ad0
  mytestprog`__libc_csu_init
  mytestprog`0x400ad0
  mytestprog`0x400af9
```
The `ustack` symbol translation occurs after the stack data is recorded. Therefore, the corresponding user process might exit before symbol translation can be performed, making stack frame translation impossible. If the user process exits before symbol translation is performed, `dtrace` outputs a warning message, followed by the hexadecimal stack frames.

### 4.3.12 uaddr

DTrace supports the use of `uaddr` with both 32-bit and 64-bit binaries.

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

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

### 4.3.13 usym

DTrace supports the use of `usym` with both 32-bit and 64-bit binaries.

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

`usym` prints the symbol for a specified address. This is analogous to how `uaddr` works, but without the hexadecimal offsets.

### 4.4 Destructive Actions

Some DTrace actions are destructive in that they change the state of the system in some well-defined way. Destructive actions may not be used unless they have been explicitly enabled. When using `dtrace`, you can enable destructive actions using the `-w` option. If you attempt to perform destructive actions without explicitly enabling them, `dtrace` fails with a message similar to the following example:

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

#### 4.4.1 Process Destructive Actions

Some destructive actions are destructive only to a particular process.

##### 4.4.1.1 copyout

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

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

##### 4.4.1.2 copyoutstr

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

The `copyoutstr` action copies the string specified by `str` to the address specified by `addr` in the address space of the process associated with the current thread. If the user-space address does not correspond to a valid, faulted-in page in the current address space, an error will be generated. The string length is limited to the value set by the `strsize` option. See Chapter 10, *Options and Tunables* for details.

##### 4.4.1.3 raise

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

The `raise` action allows you to trigger a signal in the process associated with the current thread. The signal specified by `signal` is sent to the process.
The `raise` action sends the specified signal to the currently running process. This action is similar to using the `kill` command to send a signal to a process. The `raise` action can be used to send a signal at a precise point in the execution of a process.

### 4.4.1.4 stop

```c
void stop(void)
```

The `stop` action forces the process that fires the enabled probe to stop when it next leaves the kernel, as if stopped by a proc action. The `stop` action can be used to stop a process at any DTrace probe point. This action can be used to capture a program in a particular state that would be difficult to achieve with a simple breakpoint, and then attach a traditional debugger such as `gdb` to the process. You can also use the `gcore` utility to save the state of a stopped process in a core file for later analysis.

### 4.4.1.5 system

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

The `system` action causes the specified `program` to be executed as if it were given to the shell as input. The `program` string may contain any of the `printf` or `printa` format conversions. Arguments must be specified that match the format conversions. Refer to Chapter 6, *Output Formatting* for details on valid format conversions.

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

```bash
# dtrace -wqn tick-1sec '{system("date")}'
Tue Oct 16 10:21:34 BST 2012
Tue Oct 16 10:21:35 BST 2012
Tue Oct 16 10:21:36 BST 2012
^C
```

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

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

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

```bash
# dtrace -s whosend.d
SIGINT sent to top by root
SIGINT sent to bash by root
SIGINT sent to bash by A Nother
^C
SIGINT sent to dtrace by root
```

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

tick-1sec
/n++ < 5/
{
    printf("walltime : %Y\n", walltimestamp);
    printf("date : ");
    system("date");
    printf("\n");
}
tick-1sec
/n == 5/
{
    exit(0);
}

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

```
# dtrace -s time.d
walltime : 2012 Oct 16 10:26:07
date : Tue Oct 16 10:26:11 BST 2012

walltime : 2012 Oct 16 10:26:08
date : Tue Oct 16 10:26:11 BST 2012

walltime : 2012 Oct 16 10:26:09
date : Tue Oct 16 10:26:11 BST 2012

walltime : 2012 Oct 16 10:26:10
date : Tue Oct 16 10:26:11 BST 2012

walltime : 2012 Oct 16 10:26:11
date : Tue Oct 16 10:26:11 BST 2012
```

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

## 4.4.2 Kernel Destructive Actions

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

### 4.4.2.1 chill

```c
void chill(int nanoseconds)
```

The `chill` action causes DTrace to spin for the specified number of nanoseconds. `chill` is primarily useful for exploring problems that might be timing related. For example, you can use this action to open race condition windows, or to bring periodic events into or out of phase with one another. Because interrupts are disabled while in DTrace probe context, any use of `chill` results in interrupt, scheduling, or dispatch latency. Therefore, `chill` can cause unexpected systemic effects and it should not used indiscriminately. Because system activity relies on periodic interrupt handling, DTrace refuses to execute the `chill` action for more than 500 milliseconds out of each one-second interval on any given CPU. If the maximum `chill` interval is exceeded, DTrace reports an illegal operation error, as shown in the following example:

```
# dtrace -w -n syscall::openat:entry'(chill(500000001))'
```
Special Actions

4.4.2.2 panic

void panic(void)

The panic action causes a kernel panic when triggered. This action should be used to force a system crash dump at a time of interest. You can use this action together with ring buffering to understand a problem. For more information, see Chapter 5, Buffers and Buffering. When the panic action is used, a panic message appears that denotes the probe causing the panic. rsyslogd also emits a message upon reboot. The message buffer of the crash dump contains the probe and event control block (ECB) responsible for the panic action.

4.5 Special Actions

This section describes actions that are not data recording actions or destructive actions.

4.5.1 Speculative Actions

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

4.5.2 exit

void exit(int status)

The exit action is used to immediately stop tracing, and to inform the DTrace consumer that it should cease tracing, perform any final processing, and call exit() with the specified status value. Because exit returns a status to user level, it is a data recording action. However, unlike other data storing actions, exit cannot be speculatively traced. exit causes the DTrace consumer to exit regardless of buffer policy. Because exit is a data recording action, it can be dropped.

When exit is called, only DTrace actions already in progress on other CPUs are completed. No new actions occur on any CPU. The only exception to this rule is the processing of the END probe, which is called after the DTrace consumer has processed the exit action, and indicates that tracing should stop.

4.5.3 setopt

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

The setopt action allows you to dynamically specify a DTrace option, for example:

setopt("quiet");
setopt("bufsize", "50m");
4.6 Subroutines

Subroutines differ from actions because they generally only affect internal DTrace state. Therefore, there are no destructive subroutines, and subroutines never trace data into buffers. Many subroutines have analogs in the application programming interfaces that are described in the section 3 manual pages.

4.6.1 alloca

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

alloca allocates size bytes out of scratch space, and returns a pointer to the allocated memory. The returned pointer is guaranteed to have 8-byte alignment. Scratch space is only valid for the duration of a clause. Memory allocated with alloca will be deallocated when the clause completes. If insufficient scratch space is available, alloca will be deallocated and an error is generated.

4.6.2 basename

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

basename creates a string that consists of a copy of the specified string, but excluding any prefix that ends in /, such as a directory path. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, basename does not execute and an error is generated.

4.6.3 bcopy

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

bcopy copies size bytes from the memory pointed to by src to the memory pointed to by dest. All of the source memory must lie outside of scratch memory and all of the destination memory must lie within it. If these conditions are not met, no copying takes place and an error is generated.

4.6.4 cleanpath

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

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

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

4.6.5 copyin

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

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

### 4.6.6 copyinstr

```c
string copyinstr(uintptr_t addr)
string copyinstr(uintptr_t addr, size_t maxlen)
```

`copyinstr` copies a null-terminated C string from the specified user address `addr` into a DTrace scratch buffer, and returns the address of this buffer. The user address is interpreted as an address in the space of the process associated with the current thread. The `maxlen` parameter, if specified, sets a limit on the number of bytes past `addr` that are examined (the resulting string is always null-terminated). The resulting string's length is limited to the value set by the `strsize` option; see Chapter 10, Options and Tunables for details. As with `copyin`, the specified address must correspond to a faulted-in page in the current process. If the address does not correspond to a faulted-in page, or if insufficient scratch space is available, NULL is returned, and an error is generated.

### 4.6.7 copyinto

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

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

### 4.6.8 d_path

```c
string d_path(struct path *ptr)
```

`d_path` creates a string that contains the absolute pathname of the `struct path` pointed to by `ptr`. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `d_path` does not execute and an error is generated.

### 4.6.9 dirname

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

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

### 4.6.10 getmajor

```c
dev_t getmajor(dev_t dev)
```

`getmajor` returns the major device number for the device specified by `dev`.

### 4.6.11 getminor

```c
dev_t getminor(dev_t dev)
```

`getminor` returns the minor device number for the device specified by `dev`. 
4.6.12 htonl

```c
uint32_t htonl(uint32_t hostlong)
```

`htonl` converts `hostlong` from host-byte order to network-byte order.

4.6.13 htonll

```c
uint64_t htonll(uint64_t hostlonglong)
```

`htonll` converts `hostlonglong` from host-byte order to network-byte order.

4.6.14 htons

```c
uint16_t htons(uint16_t hostshort)
```

`htons` converts `hostshort` from host-byte order to network-byte order.

4.6.15 index

```c
int index(const char *s, const char *subs)
int index(const char *s, const char *subs, int start)
```

`index` locates the position of the first occurrence of the substring `subs` in the string `s`, starting at the optional position `start`. If the specified value of `start` is less than 0, it is implicitly set to 0. If `s` is an empty string, `index` returns 0. If no match is found for `subs` in `s`, `index` returns -1.

4.6.16 inet_ntoa

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

`inet_ntoa` takes a pointer `addr` to an IPv4 address and returns it as a dotted quad decimal string. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `inet_ntoa` does not execute and an error is generated.

4.6.17 inet_ntoa6

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

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

4.6.18 inet_ntop

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

`inet_ntop` takes a pointer `addr` to an IP address and returns a string version depending on the provided address family. Supported address families are `AF_INET` and `AF_INET6`, both of which are defined for use in D programs. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, `inet_ntop` does not execute and an error is generated.

4.6.19 lltostr

```c
string lltostr(int64_t longlong)
```
lltostr converts longlong to a string. The returned string is allocated out of scratch memory, and is therefore valid only for the duration of the clause. If insufficient scratch space is available, lltostr does not execute and an error is generated.

4.6.20 mutex_owned

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

mutex_owned returns non-zero if the calling thread currently holds the specified kernel mutex, or zero if the specified adaptive mutex is currently unowned.

4.6.21 mutex_owner

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

mutex_owner returns the thread pointer of the current owner of the specified adaptive kernel mutex. mutex_owner returns NULL if the specified adaptive mutex is currently unowned, or if the specified mutex is a spin mutex.

4.6.22 mutex_typeadaptive

```c
int mutex_typeadaptive(kmutex_t *mutex)
```

All mutexes in the Oracle Linux kernel are adaptive, so this function always returns 1.

4.6.23 mutex_type_spin

```c
int mutex_type_spin(kmutex_t *mutex)
```

All mutexes in the Oracle Linux kernel are adaptive, so this function always returns 0.

4.6.24 ntohl

```c
uint32_t ntohl(uint32_t netlong)
```

ntohl converts netlong from host-byte order to network-byte order.

4.6.25 ntohll

```c
uint64_t ntohll(uint64_t netlonglong)
```

ntohll converts netlonglong from host-byte order to network-byte order.

4.6.26 ntohs

```c
uint16_t ntohs(uint16_t netshort)
```

ntohs converts netshort from host-byte order to network-byte order.

4.6.27 progenyof

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

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

    int rand(void)

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

4.6.29 rindex

    int rindex(const char *s, const char *subs)
    int rindex(const char *s, const char *subs, int start)

rindex locates the position of the last occurrence of the substring subs in the string s, starting at the optional position start. If the specified value of start is less than 0, it is implicitly set to 0. If s is an empty string, rindex returns 0. If no match is found for subs in s, rindex returns -1.

4.6.30 rw_iswriter

    int rw_iswriter(krwlock_t *rwlock)

rw_iswriter returns non-zero if the specified reader-writer lock rwlock is held by a writer. If the lock is desired by a writer, is held only by readers and no writer is blocked, or if the lock is not held at all, rw_iswriter returns zero.

4.6.31 rw_read_held

    int rw_read_held(krwlock_t *rwlock)

rw_read_held returns non-zero if the specified reader-writer lock rwlock is currently held by a reader. If the lock is held only by writers or is not held at all, rw_read_held returns zero.

4.6.32 rw_write_held

    int rw_write_held(krwlock_t *rwlock)

rw_write_held returns non-zero if the specified reader-writer lock rwlock is currently held by a writer. If the lock is held only by readers or is not held at all, rw_write_held returns zero.

4.6.33 speculation

    int speculation(void)

speculation reserves a speculative trace buffer for use with speculate and returns an identifier for this buffer. See Chapter 7, Speculative Tracing for details.

4.6.34 strchr

    string strchr(const char *s, char c)

strchr returns a pointer to the first occurrence of the character c in the string s. If no match is found, strstr returns 0. This function does not work with wide or multi-byte characters.

4.6.35 strjoin

    string strjoin(char *str1, char *str2)
4.6.36 strlen

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

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

4.6.37 strrchr

```c
string strrchr(const char *s, char c)
```

(strrchr) returns a pointer to the last occurrence of the character `c` in the string `s`. If no match is found, `strrstr` returns 0. This function does not work with wide or multi-byte characters.

4.6.38 strstr

```c
string strstr(const char *s, const char *subs)
```

 strstr returns a pointer to the first occurrence of the substring `subs` in the string `s`. If `s` is an empty string, `strstr` returns a pointer to an empty string. If no match is found, `strstr` returns 0.

4.6.39 strtok

```c
string strtok(const char *s, const char *delim)
```

(strtok) parses a string into a sequence of tokens using `delim` as the delimiting string. When you first call `strtok`, specify the string to be parsed in `s`. In each subsequent call to obtain the next token, specify `str` as NULL. You can specify a different delimiter for each call. The internal pointer that `strtok` uses to traverse `s` is only valid within multiple enablings of the same probe. That is, it behaves like an implicit clause-local variable. `strtok` returns NULL if there are no more tokens.

4.6.40 substr

```c
string substr(const char *s, int index)
string substr(const char *s, int index, int length)
```

(substr) returns the substring of the string `s` starting at the position `index`. If `length` is specified, `substr` limits the substring to that length.
Data buffering and management is an essential service provided by the DTrace framework for its clients, such as `dtrace`. This chapter explores data buffering in detail and describes options you can use to change DTrace's buffer management policies.

### 5.1 Principal Buffers

The principal buffer is present in every DTrace invocation and is the buffer to which tracing actions record their data by default. These actions include `exit`, `printa`, `printf`, `stack`, `trace`, and `tracemem`.

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

### 5.2 Principal Buffer Policies

DTrace permits tracing in highly constrained contexts in the kernel. In particular, DTrace permits tracing in contexts in which kernel software may not reliably allocate memory. The consequence of this flexibility of context is that there always exists a possibility that DTrace might attempt to trace data when there is no space available. DTrace must have a policy to deal with such situations when they arise, but you might wish to tune the policy based on the needs of a given experiment. Sometimes the appropriate policy might be to discard the new data. Other times it might be desirable to reuse the space containing the oldest recorded data to trace new data. Most often, the desired policy is to minimize the likelihood of running out of available space in the first place. To accommodate these varying demands, DTrace supports several different buffer policies. This support is implemented with the `bufpolicy` option, and can be set on a per-consumer basis. See Chapter 10, *Options and Tunables* for more details on setting options.

#### 5.2.1 switch Policy

By default, the principal buffer has a `switch` buffer policy. Under this policy, per-CPU buffers are allocated in pairs: one buffer is active and the other buffer is inactive. When a DTrace consumer attempts to read a buffer, the kernel first switches the inactive and active buffers. Buffer switching is done in such a manner that there is no window in which tracing data may be lost. Once the buffers are switched, the newly inactive buffer is copied out to the DTrace consumer. This policy assures that the consumer always sees a self-consistent buffer: a buffer is never simultaneously traced to and copied out. This technique also avoids introducing a window in which tracing is paused or otherwise prevented. The rate at which the buffer is switched and read out is controlled by the consumer with the `switchrate` option. As with any rate option, `switchrate` may be specified with any time suffix, but defaults to rate-per-second. For more details on `switchrate` and other options, see Chapter 10, *Options and Tunables*. 

---

**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Principal Buffers</td>
<td>83</td>
</tr>
<tr>
<td>5.2 Principal Buffer Policies</td>
<td>83</td>
</tr>
<tr>
<td>5.2.1 switch Policy</td>
<td>83</td>
</tr>
<tr>
<td>5.2.2 fill Policy</td>
<td>84</td>
</tr>
<tr>
<td>5.2.3 fill Policy and END Probes</td>
<td>84</td>
</tr>
<tr>
<td>5.2.4 ring Policy</td>
<td>84</td>
</tr>
<tr>
<td>5.3 Other Buffers</td>
<td>85</td>
</tr>
<tr>
<td>5.4 Buffer Sizes</td>
<td>85</td>
</tr>
<tr>
<td>5.5 Buffer Resizing Policy</td>
<td>85</td>
</tr>
</tbody>
</table>
Under the switch policy, if a given enabled probe would trace more data than there is space available in the active principal buffer, the data is dropped and a per-CPU drop count is incremented. In the event of one or more drops, dtrace displays a message similar to the following example:

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

If a given record is larger than the total buffer size, the record is dropped regardless of buffer policy. You can reduce or eliminate drops by either increasing the size of the principal buffer with the bufsize option or by increasing the switching rate with the switchrate option.

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

### 5.2.2 fill Policy

For some problems, you might wish to use a single in-kernel buffer. While this approach can be implemented with the switch policy and appropriate D constructs by incrementing a variable in D and predicating an exit action appropriately, such an implementation does not eliminate the possibility of drops. To request a single, large in-kernel buffer, and continue tracing until one or more of the per-CPU buffers has filled, use the fill buffer policy. Under this policy, tracing continues until an enabled probe attempts to trace more data than can fit in the remaining principal buffer space. When insufficient space remains, the buffer is marked as filled and the consumer is notified that at least one of its per-CPU buffers has filled. Once dtrace detects a single filled buffer, tracing is stopped, all buffers are processed and dtrace exits. No further data is traced to a filled buffer even if the data would fit in the buffer.

To use the fill policy, set the bufpolicy option to fill. For example, the following command traces every system call entry into a per-CPU 2 KB buffer with the buffer policy set to fill:

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

### 5.2.3 fill Policy and END Probes

END probes usually do not fire until tracing has been explicitly stopped by the DTrace consumer. END probes are guaranteed to fire only on one CPU, but the CPU on which the probe fires is undefined. With fill buffers, tracing is explicitly stopped when at least one of the per-CPU principal buffers has been marked as filled. If the fill policy is selected, the END probe might fire on a CPU that has a filled buffer. To accommodate END tracing in fill buffers, DTrace calculates the amount of space potentially consumed by END probes and subtracts this space from the size of the principal buffer. If the net size is negative, DTrace refuses to start, and dtrace outputs an error message:

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

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

### 5.2.4 ring Policy

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

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

When used to create a ring buffer, dtrace does not display any output until the process is terminated. At that time, the ring buffer is consumed and processed. dtrace processes each ring buffer in CPU order. Within a CPU's buffer, trace records will be displayed in order from oldest to youngest. Just as with the
**Other Buffers**

The following example demonstrates the use of a `#pragma option` directive to enable ring buffering:

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

syscall::entry
/execname == $1/
{
    trace(timestamp);
}
syscall::rexit:entry
{
    exit(0);
}
```

### 5.3 Other Buffers

Principal buffers exist in every DTrace enabling. Beyond principal buffers, some DTrace consumers may have additional in-kernel data buffers: an aggregation buffer, discussed in Chapter 3, *Aggregations* and one or more speculative buffers, discussed in Chapter 7, *Speculative Tracing*.

### 5.4 Buffer Sizes

The size of each buffer can be tuned on a per-consumer basis. Separate options are provided to tune each buffer size, as shown in the following table:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Size Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td>aggsizes</td>
</tr>
<tr>
<td>Principal</td>
<td>bufsize</td>
</tr>
<tr>
<td>Speculative</td>
<td>specsizes</td>
</tr>
</tbody>
</table>

Each of these options is set with a value that denotes the size. As with any size option, the value may have an optional size suffix. See Chapter 10, *Options and Tunables* for more details. For example, to set the buffer size to 10 megabytes on the `dtrace` command line:

```plaintext
# dtrace -P syscall -x bufsize=10m
```

Alternatively, you can use the `-b` option:

```plaintext
# dtrace -P syscall -b 10m
```

Finally, you can set `bufsize` using a pragma:

```plaintext
#pragma D option bufsize=10m
```

The buffer size you select denotes the size of the buffer on each CPU. Moreover, for the `switch` buffer policy, `bufsize` denotes the size of each buffer on each CPU. The default buffer size is four megabytes.

### 5.5 Buffer Resizing Policy

Occasionally, the system might not have adequate free kernel memory to allocate a buffer of desired size either because not enough memory is available or because the DTrace consumer has exceeded one of the tunable limits described in Chapter 10, *Options and Tunables*. You can configure the policy for buffer
Buffer Resizing Policy

allocation failure using `bufresize` option, which defaults to `auto`. Under the `auto` buffer resize policy, the size of a buffer is halved until a successful allocation occurs. `dtrace` generates a message if a buffer as allocated is smaller than the requested size:

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

or:

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

Alternatively, you can require manual intervention after buffer allocation failure by setting `bufresize` to `manual`. Under this policy, an allocation failure causes DTrace to fail to start:

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

The buffer resizing policy of all buffers (principal, speculative and aggregation) is dictated by the `bufresize` option.
Chapter 6 Output Formatting

Table of Contents

6.1 printf ................................................................. 87
  6.1.1 Conversion Specifications .................................. 87
  6.1.2 Flag Specifiers .................................................. 88
  6.1.3 Width and Precision Specifiers ................................ 89
  6.1.4 Size prefixes ..................................................... 89
  6.1.5 Conversion Formats .............................................. 89
6.2 printa .................................................................. 91
6.3 trace Default Format ............................................... 93

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

6.1 printf

The printf function combines the ability to trace data, as if by the trace function, with the ability to output the data and other text in a specific format that you describe. The printf function tells DTrace to trace the data associated with each argument after the first argument, and then to format the results using the rules described by the first printf argument, known as a format string. The format string is a regular string that contains any number of format conversions, each beginning with a % character, that describe how to format the corresponding argument. The first conversion in the format string corresponds to the second printf argument, the second conversion to the third argument, and so on. All of the text between conversions is printed verbatim. The character following the % conversion character describes the format to use for the corresponding argument.

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

• The D compiler compares the arguments to the conversions in the format string. If an argument's type is incompatible with the format conversion, the D compiler provides an error message explaining the problem.

• The D compiler does not require the use of size prefixes with printf format conversions. The C printf routine requires that you indicate the size of arguments by adding prefixes such as %ld for long or %lld for long long. The D compiler knows the size and type of your arguments, so these prefixes are not required in your D printf statements.

• DTrace provides additional format characters that are useful for debugging and observability. For example, the %a format conversion can be used to print a pointer as a symbol name and offset.

In order to implement these features, you must specify the format string in the DTrace printf function as a string constant in your D program. Format strings cannot be dynamic variables of type string.

6.1.1 Conversion Specifications

Each conversion specification in the format string is introduced by the % character, after which the following information appears in sequence:
• Zero or more flags (in any order), that modify the meaning of the conversion specification as described in the next section.

• An optional minimum field width. If the converted value has fewer bytes than the field width, the value will be padded with spaces on the left by default, or on the right if the left-adjustment flag (−) is specified. The field width can also be specified as an asterisk (*), in which case the field width is set dynamically based on the value of an additional argument of type int.

• An optional precision that indicates the minimum number of digits to appear for the d, i, o, u, x, and X conversions (the field is padded with leading zeroes); the number of digits to appear after the radix character for the e, E, and f conversions, the maximum number of significant digits for the g and G conversions; or the maximum number of bytes to be printed from a string by the s conversion. The precision takes the form of a period (.) followed by either an asterisk (*) as described in Section 6.1.3, “Width and Precision Specifiers”, or a decimal digit string.

• An optional sequence of size prefixes that indicate the size of the corresponding argument. The size prefixes are not required in D, and are provided for compatibility with the C printf() function.

• A conversion specifier that indicates the type of conversion to be applied to the argument.

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

6.1.2 Flag Specifiers

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

<table>
<thead>
<tr>
<th>Flag Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>’</td>
<td>The integer portion of the result of a decimal conversion (%d, %f, %g, %G, %i, or %u) is formatted with thousands grouping characters using the non-monetary grouping character. Some locales, including the POSIX C locale, do not provide non-monetary grouping characters for use with this flag. (The relevant locale is the locale in which dtrace is running.)</td>
</tr>
<tr>
<td>−</td>
<td>The result of the conversion is left-justified within the field. The conversion is right-justified if this flag is not specified.</td>
</tr>
<tr>
<td>+</td>
<td>The result of signed conversion always begins with a sign (+ or −). If this flag is not specified, the conversion begins with a sign only when a negative value is converted.</td>
</tr>
<tr>
<td>space</td>
<td>If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space is placed before the result. If the space and + flags both appear, the space flag is ignored.</td>
</tr>
<tr>
<td>#</td>
<td>The value is converted to an alternate form if an alternate form is defined for the selected conversion. The alternate formats for conversions are described along with the corresponding conversion.</td>
</tr>
<tr>
<td>0</td>
<td>For d, e, E, f, g, G, i, o, u, x, and X conversions, leading zeroes (following any indication of sign or base) are used to pad to the field width. No space padding is performed. If the 0 and − flags both appear, the 0 flag is ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag is ignored. If the 0 and ’ flags both appear, the grouping characters are inserted before the zero padding.</td>
</tr>
</tbody>
</table>
6.1.3 Width and Precision Specifiers

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

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

The field width can also be specified using a `?` character to indicate that the field width should be set based on the number of characters required to format an address in hexadecimal in the data model of the operating system kernel. The width is set to 8 if the kernel is using the 32-bit data model, or to 16 if the kernel is using the 64-bit data model. The precision for the conversion can be specified as a decimal digit string following a period (.) or by an asterisk (*) following a period. If an asterisk is used to specify the precision, an additional argument of type `int` prior to the conversion argument is accessed to determine the precision. If both width and precision are specified as asterisks, the order of arguments to `printf` for the conversion should appear in the following order: width, precision, value.

6.1.4 Size prefixes

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

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

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

6.1.5 Conversion Formats

Each conversion character sequence results in fetching zero or more arguments. If insufficient arguments are provided for the format string, the format string is exhausted and arguments remain, or an undefined conversion format is specified, the D compiler issues an appropriate error message. The following table lists the conversion character sequences.

<table>
<thead>
<tr>
<th>Conversion Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>The pointer or <code>uintptr_t</code> argument is printed as a kernel symbol name in the form <code>module'symbol-name</code> plus an optional hexadecimal byte offset. If</td>
</tr>
</tbody>
</table>
## Conversion Formats

<table>
<thead>
<tr>
<th>Conversion Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>the value does not fall within the range defined by a known kernel symbol, the value is printed as a hexadecimal integer.</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Exactly like <code>%a</code>, but used for user symbols.</td>
</tr>
<tr>
<td>c</td>
<td>The <code>char</code>, <code>short</code>, or <code>int</code> argument is printed as an ASCII character.</td>
</tr>
<tr>
<td>C</td>
<td>The <code>char</code>, <code>short</code>, or <code>int</code> argument is printed as an ASCII character if the character is a printable ASCII character. If the character is not a printable character, it is printed using the corresponding escape sequence as shown in Table 2.6, “Character Escape Sequences”.</td>
</tr>
<tr>
<td>d</td>
<td>The <code>char</code>, <code>int</code>, <code>long</code>, <code>long long</code>, or <code>short</code> argument is printed as a decimal (base 10) integer. If the argument is <code>signed</code>, it is printed as a signed value. If the argument is <code>unsigned</code>, it is printed as an unsigned value. This conversion has the same meaning as <code>i</code>.</td>
</tr>
<tr>
<td>e, E</td>
<td>The <code>double</code>, <code>float</code>, or <code>long double</code> argument is converted to the style <code>[-]ddde[+-]dd</code>, where there is one digit before the radix character and the number of digits after it is equal to the precision. The radix character is non-zero if the argument is non-zero. If the precision is not specified, the default precision value is 6. If the precision is 0 and the <code>#</code> flag is not specified, no radix character appears. The <code>E</code> conversion format produces a number with <code>E</code> instead of <code>e</code> introducing the exponent. The exponent always contains at least two digits. The value is rounded up to the appropriate number of digits.</td>
</tr>
<tr>
<td>f</td>
<td>The <code>double</code>, <code>float</code>, or <code>long double</code> argument is converted to the style <code>[-]ddd.ddd</code>, where the number of digits after the radix character is equal to the precision specification. If the precision is not specified, the default precision value is 6. If the precision is 0 and the <code>#</code> flag is not specified, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded up to the appropriate number of digits.</td>
</tr>
<tr>
<td>g, G</td>
<td>The <code>double</code>, <code>float</code>, or <code>long double</code> argument is printed in the style <code>f</code> or <code>e</code> (or in style <code>E</code> in the case of a <code>G</code> conversion character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted: style <code>e</code> (or <code>E</code>) is used only if the exponent resulting from the conversion is less than (-4), or greater than or equal to the precision. Trailing zeroes are removed from the fractional part of the result. A radix character appears only if it is followed by a digit. If the <code>#</code> flag is specified, trailing zeroes are not removed from the result.</td>
</tr>
<tr>
<td>i</td>
<td>The <code>char</code>, <code>int</code>, <code>long</code>, <code>long long</code>, or <code>short</code> argument is printed as a decimal (base 10) integer. If the argument is <code>signed</code>, it is printed as a signed value. If the argument is <code>unsigned</code>, it is printed as an unsigned value. This conversion has the same meaning as <code>d</code>.</td>
</tr>
<tr>
<td>k</td>
<td>The <code>stack</code> argument is printed as if by a call to <code>trace()</code>. Handles kernel-level stacks. This argument is valid only with <code>printa</code> because <code>stack</code> may not be called from a D expression (a D program context is required).</td>
</tr>
<tr>
<td>o</td>
<td>The <code>char</code>, <code>int</code>, <code>long</code>, <code>long long</code>, or <code>short</code> argument is printed as an unsigned octal (base 8) integer. Arguments that are <code>signed</code> or <code>unsigned</code> may be used with this conversion. If the <code>#</code> flag is specified, the precision of the result is increased if necessary to force the first digit of the result to be a zero.</td>
</tr>
<tr>
<td>p</td>
<td>The pointer or <code>uintptr_t</code> argument is printed as a hexadecimal (base 16) integer. D accepts pointer arguments of any type. If the <code>#</code> flag is specified, a non-zero result has <code>0x</code> prepended to it.</td>
</tr>
</tbody>
</table>
### Conversion Characters

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s</strong></td>
<td>The argument must be an array of char or a string. Bytes from the array or string are read up to a terminating null character or the end of the data and interpreted and printed as ASCII characters. If the precision is not specified, it is taken to be infinite, so all characters up to the first null character are printed. If the precision is specified, only that portion of the character array that is displayed in the corresponding number of screen columns is printed. If an argument of type char * is to be formatted, it should be cast to string or prefixed with the D,string operator to indicate that DTrace should trace the bytes of the string and format them.</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>The argument must be an array of char or a string. The argument is processed as if by the %s conversion, but any ASCII characters that are not printable are replaced by the corresponding escape sequence described in Table 2.6, “Character Escape Sequences”.</td>
</tr>
<tr>
<td><strong>u</strong></td>
<td>The char, int, long, long long, or short argument is printed as an unsigned decimal (base 10) integer. Arguments that are signed or unsigned can be used with this conversion, and the result is always formatted as unsigned.</td>
</tr>
<tr>
<td><strong>wc</strong></td>
<td>The int argument is converted to a wide character (wchar_t) and the resulting wide character is printed.</td>
</tr>
<tr>
<td><strong>wc</strong></td>
<td>The int argument is converted to a wide character (wchar_t) and the resulting wide character is printed.</td>
</tr>
<tr>
<td><strong>ws</strong></td>
<td>The argument must be an array of wchar_t. Bytes from the array are read up to a terminating null character or the end of the data and interpreted and printed as wide characters. If the precision is not specified, it is taken to be infinite, so all wide characters up to the first null character are printed. If the precision is specified, only that portion of the wide character array that is displayed in the corresponding number of screen columns is printed.</td>
</tr>
<tr>
<td><strong>x, X</strong></td>
<td>The char, int, long, long long, or short argument is printed as an unsigned hexadecimal (base 16) integer. Arguments that are signed or unsigned may be used with this conversion. If the x form of the conversion is used, the letter digits abcdef are used. If the X form of the conversion is used, the letter digits ABCDEF are used. If the # flag is specified, a non-zero result has 0x (for %x) or 0X (for %X) prepended to it.</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>The uint64_t argument is interpreted to be the number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970, and is printed in the following format: &quot;%Y %a %b %e %T %z&quot;. The current number of nanoseconds since 00:00 UTC, January 1, 1970 is available as the walltimestamp variable.</td>
</tr>
</tbody>
</table>
| **%**     | Print a literal % character. No argument is converted. The entire conversion specification must be %%.

### 6.2 printa

The printa function allows you to format the results of aggregations in a D program. The function is invoked using one of two forms:

```d
printa(@aggregation-name);
printa(format-string, @aggregation-name);
```
If the first form of the function is used, the `dtrace` command takes a consistent snapshot of the aggregation data and produces output equivalent to the default output format used for aggregations, described in Chapter 3, Aggregations. If the second form of the function is used, `dtrace` takes a consistent snapshot of the aggregation data and produces output according to the conversions specified in the format string, according to the following rules:

- The format conversions must match the tuple signature used to create the aggregation. Each tuple element may only appear once. For example, if you aggregate a count using the following D statements:

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

  and then add the D statement `printa(format-string, @a)` to a probe clause, `dtrace` takes a snapshot of the aggregation data and produces output as if you had entered these statements:

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

  and so on for each tuple defined in the aggregation.

- Unlike `printf`, the format string you use for `printa` need not include all elements of the tuple. That is, you can have a tuple of length 3 and only one format conversion. Therefore, you can omit any tuple keys from your `printa` output by changing your aggregation declaration to move the keys you want to omit to the end of the tuple and then omit corresponding conversion specifiers for them in the `printa` format string.

- The aggregation result can be included in the output by using the additional `@` format flag character, which is only valid when used with `printa`. The `@` flag can be combined with any appropriate format conversion specifier, and may appear more than once in a format string so that your tuple result can appear anywhere in the output and can appear more than once. The set of conversion specifiers that can be used with each aggregating function are implied by the aggregating function's result type. The aggregation result types are:

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg</td>
<td>uint64_t</td>
</tr>
<tr>
<td>count</td>
<td>uint64_t</td>
</tr>
<tr>
<td>lquantize</td>
<td>int64_t</td>
</tr>
<tr>
<td>max</td>
<td>uint64_t</td>
</tr>
<tr>
<td>min</td>
<td>uint64_t</td>
</tr>
<tr>
<td>quantize</td>
<td>int64_t</td>
</tr>
<tr>
<td>sum</td>
<td>uint64_t</td>
</tr>
</tbody>
</table>

  For example, to format the results of `avg`, you can apply the `%d`, `%i`, `%o`, `%u`, or `%x` format conversions. The `quantize` and `lquantize` functions format their results as an ASCII table rather than as a single value.

  The following D program shows a complete example of `printa`, using the `profile` provider to sample the value of `caller`, and then formatting the results as a simple table:

  ```d
  profile:::tick-1000
  {
    @a[caller] = count();
  }
  END
  {
    printa("%@8u %a\n", @a);
  }
  ```
If you use `dtrace` to execute this program, wait a few seconds, and then press `Ctrl-C`, you see output similar to the following example:

```bash
# dtrace -s printa.d
```

```
dtrace: script 'printa.d' matched 2 probes
^C
CPU     ID                    FUNCTION:NAME
0      2                             :END        8 0xffffffff8105f128
   1313 0xffffffff8110cb90
   6358 0xffffffff81052fb9
   36347 0xffffffffa03c92d7
...```

### 6.3 trace Default Format

If you use `trace` rather than `printf` to capture data, the `dtrace` command formats the results using a default output format. If the data is 1, 2, 4, or 8 bytes in size, the result is formatted as a decimal integer value. If the data is any other size and is a sequence of printable characters if interpreted as a sequence of bytes, it will be printed as an ASCII string. If the data is any other size and is not a sequence of printable characters, it will be printed as a series of byte values formatted as hexadecimal integers.
This chapter discusses the DTrace facility for *speculative tracing*, the ability to tentatively trace data and then later decide whether to commit the data to a tracing buffer or discard it. In DTrace, the primary mechanism for filtering out uninteresting events is the predicate mechanism, discussed in Section 2.1, “D Program Structure”. Predicates are useful when you know at the time that a probe fires whether or not the probe event is of interest. For example, if you are only interested in activity associated with a certain process or a certain file descriptor, you know when the probe fires if it is associated with the process or file descriptor of interest. However, in other situations, you might not know whether a given probe event is of interest until some time after the probe fires.

For example, if a system call is occasionally failing with a common error code (for example, `EIO` or `EINVAL`), you might want to examine the code path leading to the error condition. To capture the code path, you could enable every probe — but only if the failing call can be isolated in such a way that a meaningful predicate can be constructed. If the failures are sporadic or non-deterministic, you would be forced to trace all events that might be interesting, and later post-process the data to filter out the ones that were not associated with the failing code path. In this case, even though the number of interesting events may be reasonably small, the number of events that must be traced is very large, making post-processing difficult.

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

### 7.1 Speculation Interfaces

The following table describes the DTrace speculation functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Args</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>speculation</td>
<td>None</td>
<td>Returns an identifier for a new speculative buffer</td>
</tr>
<tr>
<td>speculate</td>
<td>ID</td>
<td>Denotes that the remainder of the clause should be traced to the speculative buffer specified by ID</td>
</tr>
<tr>
<td>commit</td>
<td>ID</td>
<td>Commits the speculative buffer associated with ID</td>
</tr>
<tr>
<td>discard</td>
<td>ID</td>
<td>Discards the speculative buffer associated with ID</td>
</tr>
</tbody>
</table>

### 7.2 Creating a Speculation

The `speculation` function allocates a speculative buffer, and returns a speculation identifier. The speculation identifier should be used in subsequent calls to the `speculate` function. Speculative buffers
are a finite resource: if no speculative buffer is available when `speculation` is called, an ID of zero is returned and a corresponding DT race error counter is incremented. An ID of zero is always invalid, but may be passed to `speculate`, `commit` or `discard`. If a call to `speculation` fails, `dtrace` generates a message similar to the following example:

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

The number of speculative buffers defaults to one, but can be optionally tuned higher. See Section 7.7, “Speculation Options and Tuning” for more information.

### 7.3 Using a Speculation

To use a speculation, an identifier returned from `speculation` must be passed to the `speculate` function in a clause before any data-recording actions. All subsequent data-recording actions in a clause containing a `speculate` will be speculatively traced. The D compiler will generate a compile-time error if a call to `speculate` follows data recording actions in a D probe clause. Therefore, clauses may contain speculatively tracing or non-speculative tracing requests, but not both.

Aggregating actions, destructive actions, and the `exit` action may never be speculative. Any attempt to take one of these actions in a clause containing a `speculate` results in a compile-time error. A `speculate` may not follow a `speculate`: only one speculation is permitted per clause. A clause that contains only a `speculate` speculatively traces the default action, which is defined to trace only the enabled probe ID. See Chapter 4, Actions and Subroutines for a description of the default action.

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

```d
syscall::openat:entry
{
    self->spec = speculation();
}

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

### 7.4 Committing a Speculation

You commit speculations using the `commit` function. When a speculative buffer is committed, its data is copied into the principal buffer. If there is more data in the specified speculative buffer than there is available space in the principal buffer, no data is copied and the drop count for the buffer is incremented. If the buffer has been speculatively traced to on more than one CPU, the speculative data on the committing CPU is copied immediately, while speculative data on other CPUs is copied some time after the `commit`. Thus, some time might elapse between a `commit` beginning on one CPU and the data being copied from speculative buffers to principal buffers on all CPUs. This time is guaranteed to be no longer than the time dictated by the cleaning rate. See Section 7.7, “Speculation Options and Tuning” for more details.

A committing speculative buffer will not be made available to subsequent `speculation` calls until each per-CPU speculative buffer has been completely copied into its corresponding per-CPU principal buffer. Similarly, subsequent calls to `speculate` to the committing buffer will be silently discarded, and subsequent calls to `commit` or `discard` will silently fail. Finally, a clause containing a `commit` cannot contain a data recording action, but a clause may contain multiple `commit` calls to commit disjoint buffers.
7.5 Discarding a Speculation

You discard speculations using the `discard` function. When a speculative buffer is discarded, its contents are thrown away. If the speculation has only been active on the CPU calling `discard`, the buffer is immediately available for subsequent calls to `speculation`. If the speculation has been active on more than one CPU, the discarded buffer will be available for subsequent `speculation` some time after the call to `discard`. The time between a `discard` on one CPU and the buffer being made available for subsequent speculations is guaranteed to be no longer than the time dictated by the cleaning rate. If, at the time `speculation` is called, no buffer is available because all speculative buffers are currently being discarded or committed, `dtrace` generates a message similar to the following example:

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

The likelihood of all buffers being unavailable can be reduced by tuning the number of speculation buffers or the cleaning rate. See Section 7.7, “Speculation Options and Tuning”, for details.

7.6 Speculation Example

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

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

syscall::open:entry
{
    /*
    * The call to speculation() creates a new speculation. If this fails, 
    * dtrace will generate an error message indicating the reason for  
    * the failed speculation(), but subsequent speculative tracing will be 
    * silently discarded. 
    */
    self->spec = speculation();
    speculate(self->spec);

    /*
    * Because this printf() follows the speculate(), it is being
    * speculatively traced; it will only appear in the data buffer if the
    * speculation is subsequently committed.
    */
    printf("%s", copyinstr(arg0));
}

syscall::open:return
/self->spec/ 
{
    /*
    * To balance the output with the -F option, we want to be sure that
    * every entry has a matching return. Because we speculated the
    * open entry above, we want to also speculate the open return.
    * This is also a convenient time to trace the errno value.
    */
    speculate(self->spec);
    trace(errno);
}

syscall::open:return
/self->spec && errno != 0/ 
{
    /*
    * If errno is non-zero, we want to commit the speculation.
    */
    commit(self->spec);
```
```c
self->spec = 0;
}
syscall::open:return
/self->spec && errno == 0/
{
  /*
   * If errno is not set, we discard the speculation.
   */
  discard(self->spec);
  self->spec = 0;
}
```

Running this script produces output similar to the following example:

```plaintext
# ./specopen.d
dtrace: script './specopen.d' matched 4 probes
CPU FUNCTION
1  => open                                  /var/ld/ld.config
1  <= open                                          2
1  => open                                  /images/UnorderedList16.gif
1  <= open                                          4
...```

## 7.7 Speculation Options and Tuning

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

```
dtrace: 38 speculative drops
```

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

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

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

You can reduce the likelihood of failed speculations of this nature by increasing the number of speculative buffers with the `nspec` option. The value of `nspec` defaults to one.

Alternatively, `speculation` can fail if all speculative buffers are busy. In this case, the error message is similar to the following example:

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

This message indicates that `speculation` was called after `commit` was called for a speculative buffer, but before that buffer was actually committed on all CPUs. You can reduce the likelihood of failed speculations of this nature by increasing the rate at which CPUs are cleaned with the `cleanrate` option. The value of `cleanrate` defaults to 101.
Chapter 8 The dtrace Utility

Table of Contents

8.1 Description ................................................................. 99
8.2 Options ............................................................................ 99
8.3 Operands ......................................................................... 103
8.4 Exit Status ....................................................................... 103

The dtrace command is a generic front-end to the DTrace facility. The command implements a simple interface to invoke the D language compiler, the ability to retrieve buffered trace data from the DTrace kernel facility, and a set of basic routines to format and print traced data. This chapter provides a complete reference for the dtrace command.

8.1 Description

The dtrace command provides a generic interface to all of the essential services provided by the DTrace facility, including options to:

• List the set of probes and providers currently published by DTrace
• Enable probes directly using any of the probe description specifiers (provider, module, function, name)
• Run the D compiler and compile one or more D program files or programs written directly on the command-line
• Generate program stability reports (see Chapter 16, Stability)
• Modify DTrace tracing and buffering behavior and enable additional D compiler features (see Chapter 10, Options and Tunables)

dtrace can also be used to create D scripts by using it in a #! declaration to create an interpreter file (see Chapter 9, Scripting). Finally, you can use the -e option to dtrace to compile D programs and determine their properties without actually enabling any tracing.

8.2 Options

The dtrace command accepts the following options:

```
dtrace [-CeFGhIlqSvVwZ] [-b bufsize] [-c command] [-D name[-value]] [-I pathname] [-L pathname]
[-o pathname] [-p PID] [-s source_pathname]
[-U name] [-x option[-value]] [-X[a|c|s|t]]
[-P provider[[predicate]action]]
[-m [[provider:module][predicate]action]]
[-f [[provider:module:function][predicate]action]]
[-n [[provider:module:function:name][predicate]action]]
[-i probe-id[[predicate]action]]
```

where predicate is any D predicate enclosed in slashes // and action is any D statement list enclosed in braces {} according to the D language syntax. If D program code is provided as an argument to the -P, -m, -f, -n, or -i options, this text must be appropriately quoted to avoid interpretation by the shell.

The options are as follows:

```
-b bufsize Set the principal trace buffer size, which can include any of the size suffixes k (kilobyte), m (megabyte), g (gigabyte), or t
```
(terabyte). If the buffer space cannot be allocated, dtrace attempts to reduce the buffer size or exit depending on the setting of the bufresize property.

**-c command**

Run the specified command and exit upon its completion. If you specify more than one -c option, dtrace exits when all the commands have exited, and reports the exit status for each child process as it terminates. dtrace makes the process ID of the first command available to D programs as the $target macro variable.

**-C**

Run the C preprocessor (cpp) on D programs before compiling them. You can pass options to the C preprocessor by using the -D, -H, -I, and -U options. You can use the -X option to select the degree of conformance with the C standard.

**-D name[=value]**

Define the specified macro name and optional value when invoking cpp using the -c option. You can specify the -D option multiple times to the command.

**-e**

Exit after compiling any requests and before enabling any probes. You can combine this option with the -D option to verify that your D programs compile without executing them or enabling the corresponding instrumentation.

**-f [[provider:]module:] function [[predicate]action]**

Specify a function (optionally specifying the provider and module) that you want to trace or list. You can append an optional D-probe clause. You can specify the -f option multiple times to the command.

**-F**

Reduce trace output by combining the output for function and system call entry and return points. dtrace indents entry probe reports and leaves return probe reports unindented. dtrace prefixes the output from function entry probe reports with -> and the output from function return probe reports with <-. dtrace prefixes the output from system call entry probe reports with => and the output from system call return probe reports with <=.

**-G**

Generate an ELF file that contains an embedded D program. dtrace saves the DTrace probes that are specified in the program using a relocatable ELF object that can be linked with another program. If you specify the -o option, dtrace saves the ELF file to the specified path name. If you do not specify the -o option, the ELF file is given the same name as the source file for the D program, except with a .o extension instead of .s. Otherwise, the ELF file is saved with the name d.out.

**-h**

Create a header file based on probe definitions in the file that is specified as the argument to the -s option. If you specify the -o option, dtrace saves the header file to the specified path name. If you do not specify the -o option, the header file is given the same name as the source file for the D program, except with a .h extension instead of .d. You should amend the source file of the program to be traced so that it includes this header file.

**-H**

Print the path names of included files on stderr when you invoke cpp using the -c option.
Options

- `i probe_ID` ([predicate]action)
  Specify a probe identifier that you want to trace or list. You must specify the probe ID as a decimal integer (as displayed by `dtrace -l`). You can append an optional D-probe clause. You can specify the `-i` option multiple times to the command.

- `I pathname`
  Add the specified directory path to the search path for `#include` files when you invoke `cpp` using the `-C` option. The specified directory is inserted at the head of the default directory list.

- `-l`
  List probes instead of enabling them. `dtrace` filters the list of probes based on the arguments to the `-f`, `-i`, `-m`, `-n`, `-P`, and `-s` options. If no options are specified, `dtrace` lists all probes.

- `-L pathname`
  Add the specified directory path to the end of the library search path. Use this option to specify the path to DTrace libraries, which contain common definitions for D programs.

- `-m` ([provider:]module ([predicate]action)]
  Specify a module (optionally specifying the provider) that you want to trace or list. You can append an optional D-probe clause. You can specify the `-m` option multiple times to the command.

- `-n` ([provider:]module: function:name ([predicate]action]
  Specify a probe name (optionally specifying the provider, module, and function) that you want to trace or list. You can append an optional D-probe clause. You can specify the `-n` option multiple times to the command.

- `-o` pathname
  Specify the output file for the `-G` and `-l` options, or for traced data.

- `-p` PID
  Grab a process specified by its process ID, cache its symbol tables, and exit upon its completion. If you specify more than one `-p` option, `dtrace` exits when all the processes have exited, and reports the exit status for each process as it terminates. `dtrace` makes the first process ID specified available to D programs as the macro variable `$target`.

- `-P` provider['D-probe_clause']
  Specify a provider that you want to trace or list. You can append an optional D-probe clause. You can specify the `-P` option multiple times to the command.

- `-q`
  Set quiet mode. `dtrace` suppresses informational messages, column headers, the CPU ID, the probe ID, and additional newlines. Only data that is traced and formatted by the `printa()`, `printf()`, and `trace()` D program statements is displayed on `stdout`. This option is equivalent to specifying `#pragma D option quiet` in a D program.

- `-s` source_pathname
  Specifies the name of a D program source file to be compiled by `dtrace`.
  
  If you specify the `-h` option, `dtrace` creates a header file using the probe definitions in the file.

  If you specify the `-G` option, `dtrace` generates a relocatable ELF object that can be linked with another program.

  If you specify the `-e` option, `dtrace` compiles the program, but it does not enable any instrumentation.
If you specify the -l option, dtrace compiles the program and lists the set of matching probes, but it does not enable any instrumentation.

If you do not specify an option, dtrace enables the instrumentation specified by the D program and begins tracing.

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

-U name
Undefine the specified name when invoking cpp using the -C option. You can specify the -U option multiple times to the command.

-v
Set verbose mode. dtrace produces a program stability report showing the minimum interface stability and dependency level for any specified D programs.

-V
Write the highest D programming interface version supported by dtrace to stdout.

-w
Permit destructive actions by D programs. If you do not specify this option, dtrace does not compile or enable a D program that contains destructive actions. This option is equivalent to specifying #pragma D option destructive in a D program.

-x option[=value]
Enable or modify a DTrace runtime option or D compiler option.

-X[a|c|t]
Include the option -std=gnu99 (conformance with 1999 C standard including GNU extensions) when invoking cpp using the -C option.

-Xs
Include the option -traditional-cpp (conformance with K&R C) when invoking cpp using the -C option.

Regardless of the -x mode, the following additional C preprocessor definitions are always specified and valid in all modes:

- __linux
- __unix
- __SVR4
- __i386 (on x86 systems only when 32–bit programs are compiled)
- __amd64 (on x86 systems only when 64–bit programs are compiled)
- __`uname -s` `uname -r` (for example, __Linux_2.6.39-201.0.1.el6uek.x86_64__)
- __SUNW_D=1
- __SUNW_D_VERSION=0x.MMmmmuuu__ (where MM is the Major release value in hexadecimal, mmm is the Minor release value in hexadecimal, and uuu is the Minor release value in hexadecimal; see Chapter 18, Versioning for more information about DTrace versioning)
Operands

-z  Permit probe descriptions that do not match any probes. If you do not specify this option, `dtrace` reports an error and exits if a probe description does not match a known probe.

8.3 Operands

Zero or more additional arguments may be specified on the `dtrace` command line to define a set of macro variables (`$1`, `$2`, and so on) to be used in any D programs specified using the `-s` option or on the command-line. The use of macro variables is described further in Chapter 9, *Scripting*.

8.4 Exit Status

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

- **0**: The specified requests were completed successfully. For D program requests, the 0 exit status indicates that programs were successfully compiled, probes were successfully enabled, or anonymous state was successfully retrieved. `dtrace` returns 0 even if the specified tracing requests encountered errors or drops.

- **1**: A fatal error occurred. For D program requests, the 1 exit status indicates that program compilation failed or that the specified request could not be satisfied.

- **2**: Invalid command-line options or arguments were specified.
Chapter 9 Scripting

Table of Contents

9.1 Interpreter Files .............................................................................................................. 105
9.2 Macro Variables ................................................................................................................ 106
9.3 Macro Arguments ............................................................................................................. 107
9.4 Target Process ID ............................................................................................................ 108

You can use the dtrace utility to create interpreter files out of D programs similar to shell scripts that you can install as reusable interactive DTrace tools. The D compiler and dtrace command provide a set of macro variables that are expanded by the D compiler that make it easy to create DTrace scripts. This chapter provides a reference for the macro variable facility and tips for creating persistent scripts.

9.1 Interpreter Files

Similar to your shell and utilities such as awk and perl, dtrace can be used to create executable interpreter files. An interpreter file begins with a line of the form:

```
#!/pathname [arg]
```

where pathname is the path of the interpreter and arg is a single optional argument. When an interpreter file is executed, the system invokes the specified interpreter. If arg was specified in the interpreter file, it is passed as an argument to the interpreter. The path to the interpreter file itself and any additional arguments specified when it was executed are then appended to the interpreter argument list. Therefore, you always need to create DTrace interpreter files with at least these arguments:

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

When your interpreter file is executed, the argument to the -s option is be the pathname of the interpreter file itself. dtrace then reads, compiles, and executes this file as if you had typed the following command in your shell:

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

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

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

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

```
chmod a+rx interp.d
./interp.d
```

Remember that the #! directive must comprise the first two characters of your file with no intervening or preceding white space. The D compiler knows to automatically ignore this line when it processes the interpreter file.
dtrace uses getopt() to process command-line options, so you can combine multiple options in your single interpreter argument. For example, to add the -q option to the preceding example you could change the interpreter directive to:

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

**Note**

If you specify multiple option letters, the -s option must always end the list of options so that the next argument (the interpreter file name) is processed as the argument to the -s option.

If you need to specify more than one option that requires an argument in your interpreter file, use the #pragma D option directive to set your options. Several dtrace command-line options have #pragma equivalents that you can use, as described in Chapter 10, Options and Tunables.

### 9.2 Macro Variables

The D compiler defines a set of built-in macro variables that you can use when writing D programs or interpreter files. Macro variables are identifiers that are prefixed with a dollar sign ($) and are expanded once by the D compiler when processing your input file. Table 9.1, “D Macro Variables” lists the macro variables that the D compiler provides.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$egid</td>
<td>Effective group ID</td>
<td>See the getegid(2) manual page</td>
</tr>
<tr>
<td>$euid</td>
<td>Effective user ID</td>
<td>See the geteuid(2) manual page</td>
</tr>
<tr>
<td>$gid</td>
<td>Real group ID</td>
<td>See the getgid(2) manual page</td>
</tr>
<tr>
<td>$pid</td>
<td>Process ID</td>
<td>See the getpid(2) manual page</td>
</tr>
<tr>
<td>$pgid</td>
<td>Process group ID</td>
<td>See the getpgid(2) manual page</td>
</tr>
<tr>
<td>$ppid</td>
<td>Parent process ID</td>
<td>See the getppid(2) manual page</td>
</tr>
<tr>
<td>$sid</td>
<td>Session ID</td>
<td>See the getsid(2) manual page</td>
</tr>
<tr>
<td>$target</td>
<td>Target process ID</td>
<td>See Section 9.4, “Target Process ID”</td>
</tr>
<tr>
<td>$uid</td>
<td>Real user ID</td>
<td>See the getuid(2) manual page</td>
</tr>
</tbody>
</table>

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

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

```d
syscall:::entry
/pid !- $pid/
{
  @calls = count();
}
```
This clause always produces the desired result, even though each invocation of the `dtrace` command will have a different process ID. Macro variables can be used anywhere an integer, identifier, or string can be used in a D program.

Macro variables are expanded only once (that is, not recursively) when the input file is parsed. Each macro variable is expanded to form a separate input token, and cannot be concatenated with other text to yield a single token. For example, if `$pid` expands to the value 456, the D code:

```
123$pid
```

would expand to the two adjacent tokens 123 and 456, resulting in a syntax error, rather than the single integer token 123456.

Macro variables are expanded and concatenated with adjacent text inside of D probe descriptions at the start of your program clauses.

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

### 9.3 Macro Arguments

The D compiler also provides a set of macro variables corresponding to any additional argument operands specified as part of the `dtrace` command invocation. These macro arguments are accessed using the built-in names `$0` for name of the D program file or `dtrace` command, `$1` for the first additional operand, `$2` for the second operand, and so on. If you use the `-s` option, `$0` expands to the value of the name of the input file used with this option. For D programs specified on the command-line, `$0` expands to the value of `argv[0]` used to exec `dtrace` itself.

Macro arguments can expand to integers, identifiers, or strings, depending on the form of the corresponding text. As with all macro variables, macro arguments can be used anywhere integer, identifier, and string tokens can be used in a D program. All of the following examples could form valid D expressions assuming appropriate macro argument values:

```
execname == $1  /* with a string macro argument */
x += $1         /* with an integer macro argument */
trace(x->$1)    /* with an identifier macro argument */
```

Macro arguments can be used to create DTrace interpreter files that act like real Oracle Linux commands and use information specified by a user or by another tool to modify their behavior. For example, the following D interpreter file traces `write()` system calls executed by a particular process ID:

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

If you make this interpreter file executable, you can specify the value of `$1` using an additional command-line argument to your interpreter file:

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

The resulting command invocation counts each `write()` system call executed by process ID 12345.

If your D program references a macro argument that is not provided on the command line, an appropriate error message is printed and your program fails to compile:
D programs can reference unspecified macro arguments if you set the `defaultargs` option. If `defaultargs` is set, unspecified arguments have the value 0. See Chapter 10, *Options and Tunables* for more information about D compiler options. The D compiler also produces an error message if additional arguments are specified on the command line that are not referenced by your D program.

The macro argument values must match the form of an integer, identifier, or string. If the argument does not match any of these forms, the D compiler reports an appropriate error message. When specifying string macro arguments to a DTrace interpreter file, surround the argument in an extra pair of single quotes to avoid interpretation of the double quotes and string contents by your shell:

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

If you want your D macro arguments to be interpreted as string tokens even if they match the form of an integer or identifier, prefix the macro variable or argument name with two leading dollar signs (for example, `$$1`) to force the D compiler to interpret the argument value as if it were a string surrounded by double quotes. All the usual D string escape sequences (see Table 2.6, "Character Escape Sequences") are expanded inside of any string macro arguments, regardless of whether they are referenced using the `$arg` or `$$arg` form of the macro. If the `defaultargs` option is set, unspecified arguments that are referenced with the `$$arg` form have the value of the empty string (`""`).

### 9.4 Target Process ID

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

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

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

```bash
# dtrace -s syscall.d -c date
dtrace: script 'syscall.d' matched 296 probes
Tue Oct 16 15:12:07 BST 2012
```

<table>
<thead>
<tr>
<th>System Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>1</td>
</tr>
<tr>
<td>arch_prctl</td>
<td>1</td>
</tr>
<tr>
<td>clock_gettime</td>
<td>1</td>
</tr>
<tr>
<td>exit_group</td>
<td>1</td>
</tr>
<tr>
<td>getrlimit</td>
<td>1</td>
</tr>
<tr>
<td>lseek</td>
<td>1</td>
</tr>
<tr>
<td>rt_sigprocmask</td>
<td>1</td>
</tr>
<tr>
<td>set_robust_list</td>
<td>1</td>
</tr>
<tr>
<td>set_tid_address</td>
<td>1</td>
</tr>
<tr>
<td>write</td>
<td>1</td>
</tr>
<tr>
<td>futex</td>
<td>2</td>
</tr>
<tr>
<td>rt_sigaction</td>
<td>2</td>
</tr>
<tr>
<td>brk</td>
<td>3</td>
</tr>
<tr>
<td>munmap</td>
<td>3</td>
</tr>
<tr>
<td>Function</td>
<td>Count</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>read</td>
<td>5</td>
</tr>
<tr>
<td>open</td>
<td>6</td>
</tr>
<tr>
<td>mprotect</td>
<td>7</td>
</tr>
<tr>
<td>close</td>
<td>8</td>
</tr>
<tr>
<td>newfstat</td>
<td>8</td>
</tr>
<tr>
<td>mmap</td>
<td>16</td>
</tr>
</tbody>
</table>
Chapter 10 Options and Tunables

To allow for customization, DTrace affords its consumers several important degrees of freedom. To minimize the likelihood of requiring specific tuning, DTrace is implemented using reasonable default values and flexible default policies. However, situations might arise that require tuning the behavior of DTrace on a consumer-by-consumer basis. This chapter describes the DTrace options and tunables, and the interfaces that you can use to modify them.

10.1 Consumer Options

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

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggpercpu</td>
<td>Compile-time</td>
<td>—</td>
<td>Aggregate per CPU.</td>
</tr>
<tr>
<td>aggrate</td>
<td>Dynamic run-time</td>
<td>time</td>
<td>Rate of aggregation reading.</td>
</tr>
<tr>
<td>aggsiz</td>
<td>Run-time</td>
<td>size</td>
<td>Aggregation buffer size.</td>
</tr>
<tr>
<td>aggsortkey</td>
<td>Dynamic run-time</td>
<td>false or true</td>
<td>Sort aggregations by key.</td>
</tr>
<tr>
<td>aggsortkeypos</td>
<td>Dynamic run-time</td>
<td>scalar</td>
<td>Number of the aggregation key on which to sort.</td>
</tr>
<tr>
<td>aggsortpos</td>
<td>Dynamic run-time</td>
<td>scalar</td>
<td>Number of the aggregation variable on which to sort.</td>
</tr>
<tr>
<td>aggsortrev</td>
<td>Dynamic run-time</td>
<td>false or true</td>
<td>Sort aggregations in reverse order.</td>
</tr>
<tr>
<td>amin</td>
<td>Compile-time</td>
<td>string</td>
<td>Stability attribute minimum.</td>
</tr>
<tr>
<td>argref</td>
<td>Compile-time</td>
<td>—</td>
<td>Do not require all macro arguments to be used.</td>
</tr>
<tr>
<td>bufpolicy</td>
<td>Run-time</td>
<td>fill, ring, or switch</td>
<td>Buffer policy.</td>
</tr>
<tr>
<td>bufresize</td>
<td>Run-time</td>
<td>auto or manual</td>
<td>Buffer resizing policy.</td>
</tr>
<tr>
<td>Option Name</td>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bufsize</td>
<td>Run-time</td>
<td>size</td>
<td>Principal buffer size. Equivalent to dtrace option -b.</td>
</tr>
<tr>
<td>cleanrate</td>
<td>Run-time</td>
<td>time</td>
<td>Cleaning rate.</td>
</tr>
<tr>
<td>core</td>
<td>Compile-time</td>
<td>false or true</td>
<td>Enable core dumping by dtrace.</td>
</tr>
<tr>
<td>cpp</td>
<td>Compile-time</td>
<td>—</td>
<td>Use cpp to preprocess the input file.</td>
</tr>
<tr>
<td>cpphdrs</td>
<td>Compile-time</td>
<td>—</td>
<td>Specify the -H option to cpp to print the name of each header file used.</td>
</tr>
<tr>
<td>cpppath</td>
<td>Compile-time</td>
<td>string</td>
<td>Specify the path name of cpp.</td>
</tr>
<tr>
<td>cpu</td>
<td>Run-time</td>
<td>scalar</td>
<td>CPU on which to enable tracing.</td>
</tr>
<tr>
<td>ctypes</td>
<td>Compile-time</td>
<td>string</td>
<td>Write out Compact Type Format (CTF) definitions of all C types used in a program at the end of a D compilation run.</td>
</tr>
<tr>
<td>debug</td>
<td>Compile-time</td>
<td>—</td>
<td>Enable DTrace debugging mode. Equivalent to setting the environment variable DTRACE_DEBUG.</td>
</tr>
<tr>
<td>defaultargs</td>
<td>Compile-time</td>
<td>—</td>
<td>Allow references to unspecified macro arguments. Use 0 as the value for an unspecified argument.</td>
</tr>
<tr>
<td>define</td>
<td>Compile-time</td>
<td>string</td>
<td>Define a macro name and optional value in the form name[=value]. Equivalent to dtrace option -D.</td>
</tr>
<tr>
<td>destructive</td>
<td>Run-time</td>
<td>—</td>
<td>Allow destructive actions. Equivalent to dtrace option -w.</td>
</tr>
<tr>
<td>droptags</td>
<td>Compile-time</td>
<td>—</td>
<td>Specifies that drop tags are used.</td>
</tr>
<tr>
<td>dtypes</td>
<td>Compile-time</td>
<td>string</td>
<td>Write out CTF definitions of all D types used in a program at the end of a D compilation run.</td>
</tr>
<tr>
<td>dynvarsize</td>
<td>Run-time</td>
<td>size</td>
<td>Dynamic variable space size.</td>
</tr>
<tr>
<td>empty</td>
<td>Compile-time</td>
<td>—</td>
<td>Permit compilation of empty D source files.</td>
</tr>
<tr>
<td>errtags</td>
<td>Compile-time</td>
<td>—</td>
<td>Prefix default error message with error tags.</td>
</tr>
<tr>
<td>evaltime</td>
<td>Compile-time</td>
<td>exec,main, postinit, or postinit</td>
<td>Control when DTrace halts a new process after grabbing it:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exec</td>
<td>Halt the process immediately after exec().</td>
</tr>
<tr>
<td></td>
<td></td>
<td>main</td>
<td>Halt after constructor execution, immediately before main(). For</td>
</tr>
</tbody>
</table>

See Chapter 5, Buffers and Buffering.
See Chapter 7, Speculative Tracing.
See Chapter 9, Scripting.
See Chapter 4, Actions and Subroutines.
See Section 2.9, “Variables”.
### Consumer Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stripped binaries, <code>main</code> and <code>postinit</code> are silently converted to <code>preinit</code> because a symbol table is required to locate <code>main()</code></td>
<td></td>
<td></td>
<td><strong>postinit</strong> Equivalent to <code>main</code>. (On Oracle Solaris, DTrace grabs the process immediately after the <code>#pragma init</code> constructors are executed.)</td>
</tr>
</tbody>
</table>
| Halt after initialization of the dynamic linker loader (`ld.so`) and before constructor invocation. This is the default behavior. |          | | **preinit** For statically linked binaries, `preinit` is equivalent to `exec`, and it might not skip `ld.so` initialization, which can happen after `main()`.
| For stripped, statically linked binaries, both `postinit` and `main` are equivalent to `preinit`, because the `main` symbol cannot be looked up if there is no symbol table. |          | | | For statically linked binaries, `preinit` is equivalent to `exec`, and it might not skip `ld.so` initialization, which can happen after `main()`.
<p>| For stripped, statically linked binaries, both <code>postinit</code> and <code>main</code> are equivalent to <code>preinit</code>, because the <code>main</code> symbol cannot be looked up if there is no symbol table. |          | | | | For stripped, statically linked binaries, both <code>postinit</code> and <code>main</code> are equivalent to <code>preinit</code>, because the <code>main</code> symbol cannot be looked up if there is no symbol table. |
| Indent function entry and prefix with <code>-&gt;</code>. | Dynamic runtime | —       | <strong>flowindent</strong> Indent function entry and prefix with <code>-&gt;</code>. |
| Unindent function return and prefix with <code>&lt;-</code>. | Dynamic runtime | —       | <strong>flowindent</strong> Unindent function return and prefix with <code>&lt;-</code>. |
| Indent system call entry and prefix with <code>=&gt;</code>. | Dynamic runtime | —       | <strong>flowindent</strong> Indent system call entry and prefix with <code>=&gt;</code>. |
| Unindent system call return and prefix with <code>&lt;=</code>. | Dynamic runtime | —       | <strong>flowindent</strong> Unindent system call return and prefix with <code>&lt;=</code>. |
| Equivalent to <code>dtrace</code> option <code>-F</code>. |          |         | <strong>flowindent</strong> Equivalent to <code>dtrace</code> option <code>-F</code>. |
| Add a <code>#include</code> directory to the preprocessor search path. Equivalent to <code>dtrace</code> option <code>-I</code>. | Compile-time | string  | <strong>includir</strong> Add a <code>#include</code> directory to the preprocessor search path. Equivalent to <code>dtrace</code> option <code>-I</code>. |
| Do not permit unresolved kernel symbols. | Compile-time | —       | <strong>kdefs</strong> Do not permit unresolved kernel symbols. |
| Permit unresolved kernel symbols. | Compile-time | —       | <strong>knodefs</strong> Permit unresolved kernel symbols. |
| Specify whether references to dynamic translators are permitted: | Compile-time | dynamic or static | <strong>late</strong> Specify whether references to dynamic translators are permitted: |
| Allow references to dynamic translators. | Compile-time | dynamic  | <strong>late</strong> Allow references to dynamic translators. |
| Require translators to be statically defined. | Compile-time | static   | <strong>late</strong> Require translators to be statically defined. |
| Specify that the DTrace Object Format (DOF) should be lazily loaded rather than actively loaded. | Compile-time | false or true | <strong>lazyload</strong> Specify that the DTrace Object Format (DOF) should be lazily loaded rather than actively loaded. |
| Specify the path of the dynamic linker loader (<code>ld</code>). | Compile-time | string  | <strong>ldpath</strong> Specify the path of the dynamic linker loader (<code>ld</code>). |</p>
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libdir</td>
<td>Compile-time</td>
<td>string</td>
<td>Add a library directory to the library search path.</td>
</tr>
<tr>
<td>linkmode</td>
<td>Compile-time</td>
<td>dynamic, kernel, or static</td>
<td>Specify the symbol linking mode used by the assembler when processing external symbol references:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>dynamic</strong> All symbols are treated as dynamic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>kernel</strong> Kernel symbols are treated as static and user symbols are treated as dynamic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>static</strong> All symbols are treated as static.</td>
</tr>
<tr>
<td>linktype</td>
<td>Compile-time</td>
<td>dof or elf</td>
<td>Specify the output file type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>dof</strong> Produce a standalone DOF file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>elf</strong> Produce an ELF file that contains DOF.</td>
</tr>
<tr>
<td>modpath</td>
<td>Compile-time</td>
<td>string</td>
<td>Module path. The default path is /lib/modules/version.</td>
</tr>
<tr>
<td>nolibs</td>
<td>Compile-time</td>
<td>—</td>
<td>Do not process D system libraries.</td>
</tr>
<tr>
<td>nspec</td>
<td>Run-time</td>
<td>scalar</td>
<td>Number of speculations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 7, Speculative Tracing.</td>
</tr>
<tr>
<td>pgmax</td>
<td>Compile-time</td>
<td>scalar</td>
<td>Limit on the number of threads that DTrace can grab for tracing. The default value is 8.</td>
</tr>
<tr>
<td>preallocate</td>
<td>Compile-time</td>
<td>scalar</td>
<td>Amount of memory to preallocate.</td>
</tr>
<tr>
<td>procfspath</td>
<td>Compile-time</td>
<td>string</td>
<td>Path to the procfs file system. The default path is /proc.</td>
</tr>
<tr>
<td>pspec</td>
<td>Compile-time</td>
<td>—</td>
<td>Interpret ambiguous specifiers as probe names.</td>
</tr>
<tr>
<td>quiet</td>
<td>Dynamic run-time</td>
<td>—</td>
<td>Output only explicitly traced data. Equivalent to dtrace option -q.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 8, The dtrace Utility.</td>
</tr>
<tr>
<td>quietresize</td>
<td>Dynamic run-time</td>
<td>—</td>
<td>Suppress buffer-resize messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 5, Buffers and Buffering.</td>
</tr>
<tr>
<td>rawbytes</td>
<td>Dynamic run-time</td>
<td>—</td>
<td>Always print tracemem output in hexadecimal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 4, Actions and Subroutines.</td>
</tr>
<tr>
<td>specsize</td>
<td>Run-time</td>
<td>size</td>
<td>Speculation buffer size.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 7, Speculative Tracing.</td>
</tr>
<tr>
<td>stackframes</td>
<td>Run-time</td>
<td>scalar</td>
<td>Number of stack frames.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 4, Actions and Subroutines.</td>
</tr>
<tr>
<td>stackindent</td>
<td>Dynamic run-time</td>
<td>scalar</td>
<td>Number of white space characters to use when indenting stack and ustack output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Chapter 4, Actions and Subroutines.</td>
</tr>
</tbody>
</table>
### Modifying Options

Options may be set in a D script by using `#pragma D` followed by the string `option` and the option name. If the option takes a value, the option name should be followed by an equals sign (`=`) and the option value. The following examples are all valid option settings:

```d
#pragma D option nspec=4
```

### 10.2 Modifying Options

Values that denote sizes may be given an optional suffix of `k`, `m`, `g`, or `t` to denote kilobytes, megabytes, gigabytes, and terabytes respectively. Values that denote times may be given an optional suffix of `ns`, `us`, `ms`, `s` or `hz` to denote nanoseconds, microseconds, milliseconds, seconds, and number per second, respectively.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>statusrate</strong></td>
<td>Run-time</td>
<td>time</td>
<td>Rate of status checking.</td>
</tr>
</tbody>
</table>
| **stdc** | Compile-time   | a, c, s, or t | Specify ISO C conformance settings for the preprocessor when invoking `cpp` using the `-C` option.  
  *The a, c, and t settings include the option `-std=gnu99` (conformance with 1999 C standard including GNU extensions).*  
  *The s setting includes the option `-traditional-cpp` (conformance with K&R C).* |
| **strip** | Compile-time | —         | Strip non-loadable sections from the program. |
| **strsize** | Run-time      | size      | String size.  
  *See Section 2.11, “Strings”.* |
| **switchrate** | Dynamic run-time | time      | Rate of buffer switching.  
  *See Chapter 5, Buffers and Buffering.* |
| **syslibdir** | Compile-time | string    | Path name of system libraries. |
| **tree** | Compile-time   | scalar    | Value of the DTrace tree dump bitmap. |
| **tregs** | Compile-time   | scalar    | Size of the DIF tuple register set. The default value is 8. |
| **udefs** | Compile-time | —         | Do not permit unresolved user symbols. |
| **undef** | Compile-time   | string    | Undefine a symbol when invoking the preprocessor. Equivalent to `dtrace` option `-U`. |
| **undefs** | Compile-time | —         | Permit unresolved user symbols. |
| **ustackframes** | Run-time      | scalar    | Number of user-land stack frames.  
  *See Chapter 4, Actions and Subroutines.* |
| **verbose** | Compile-time | —         | DIF verbose mode, which shows each compiled DIF object (DIFO). |
| **version** | Compile-time   | string    | Request a specific version of the native DTrace library. |
| **zdefs** | Compile-time | —         | Permit probe definitions that match zero probes. |

Modifying Options
The `dtrace` command also accepts option settings on the command line as an argument to the `-x` option. For example:

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

If an invalid option is specified, `dtrace` indicates that the option name is invalid and exits:

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
```

Similarly, if an option value is not valid for the given option, `dtrace` indicates that the value is invalid:

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
```

If an option is set more than once, subsequent settings overwrite earlier settings. Some options can only be set. The presence of such an option sets it, and you cannot subsequently unset it.
# Chapter 11 Providers

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 dtrace Provider</td>
<td>118</td>
</tr>
<tr>
<td>11.1.1 BEGIN Probe</td>
<td>118</td>
</tr>
<tr>
<td>11.1.2 END Probe</td>
<td>118</td>
</tr>
<tr>
<td>11.1.3 ERROR Probe</td>
<td>119</td>
</tr>
<tr>
<td>11.1.4 Stability</td>
<td>120</td>
</tr>
<tr>
<td>11.2 profile Provider</td>
<td>120</td>
</tr>
<tr>
<td>11.2.1 profile-n Probes</td>
<td>120</td>
</tr>
<tr>
<td>11.2.2 tick-n Probes</td>
<td>121</td>
</tr>
<tr>
<td>11.2.3 Probe Creation</td>
<td>121</td>
</tr>
<tr>
<td>11.2.4 Stability</td>
<td>121</td>
</tr>
<tr>
<td>11.3 syscall Provider</td>
<td>122</td>
</tr>
<tr>
<td>11.3.1 Probes</td>
<td>122</td>
</tr>
<tr>
<td>11.3.2 Arguments</td>
<td>124</td>
</tr>
<tr>
<td>11.3.3 Stability</td>
<td>124</td>
</tr>
<tr>
<td>11.4 sdt provider</td>
<td>124</td>
</tr>
<tr>
<td>11.4.1 Creating SDT Probes</td>
<td>124</td>
</tr>
<tr>
<td>11.4.2 Stability</td>
<td>125</td>
</tr>
<tr>
<td>11.5 proc Provider</td>
<td>126</td>
</tr>
<tr>
<td>11.5.1 Probes</td>
<td>126</td>
</tr>
<tr>
<td>11.5.2 Arguments</td>
<td>127</td>
</tr>
<tr>
<td>11.5.3 lwpsinfo_t</td>
<td>128</td>
</tr>
<tr>
<td>11.5.4 psinfo_t</td>
<td>129</td>
</tr>
<tr>
<td>11.5.5 Examples</td>
<td>129</td>
</tr>
<tr>
<td>11.5.6 Stability</td>
<td>132</td>
</tr>
<tr>
<td>11.6 sched Provider</td>
<td>132</td>
</tr>
<tr>
<td>11.6.1 Probes</td>
<td>132</td>
</tr>
<tr>
<td>11.6.2 Arguments</td>
<td>134</td>
</tr>
<tr>
<td>11.6.3 cpuinfo_t</td>
<td>134</td>
</tr>
<tr>
<td>11.6.4 Examples</td>
<td>134</td>
</tr>
<tr>
<td>11.6.5 Stability</td>
<td>146</td>
</tr>
<tr>
<td>11.7 io Provider</td>
<td>146</td>
</tr>
<tr>
<td>11.7.1 Probes</td>
<td>146</td>
</tr>
<tr>
<td>11.7.2 Arguments</td>
<td>147</td>
</tr>
<tr>
<td>11.7.3 bufinfo_t</td>
<td>148</td>
</tr>
<tr>
<td>11.7.4 devinfo_t</td>
<td>148</td>
</tr>
<tr>
<td>11.7.5 fileinfo_t</td>
<td>149</td>
</tr>
<tr>
<td>11.7.6 Examples</td>
<td>150</td>
</tr>
<tr>
<td>11.7.7 Stability</td>
<td>152</td>
</tr>
<tr>
<td>11.8 fasttrap Provider</td>
<td>153</td>
</tr>
<tr>
<td>11.8.1 Probes</td>
<td>153</td>
</tr>
<tr>
<td>11.8.2 Stability</td>
<td>153</td>
</tr>
</tbody>
</table>

This chapter lists and explains the existing DTrace providers.
The dtrace provider provides several probes related to DTrace itself. You can use these probes to initialize state before tracing begins, process state after tracing has completed, and handle unexpected execution errors in other probes.

### 11.1.1 BEGIN Probe

The BEGIN probe fires before any other probe. No other probe will fire until all BEGIN clauses have completed. This probe can be used to initialize any state that is needed in other probes. The following example shows how to use the BEGIN probe to initialize an associative array to map between \texttt{mmap()} protection bits and a textual representation:

```plaintext
BEGIN
{
  prot[0] = "---";
  prot[1] = "r--";
  prot[2] = "-w-";
  prot[3] = "rw-";
  prot[4] = "--x";
  prot[5] = "r-x";
  prot[6] = "-wx";
  prot[7] = "rwx";
}
syscall::mmap:entry
{
  printf("mmap with prot = %s", prot[arg2 & 0x7]);
}
```

The BEGIN probe fires in an unspecified context. This means that the output of \texttt{stack} or \texttt{ustack}, and the value of context-specific variables (for example, \texttt{execname}), are all arbitrary. These values should not be relied upon or interpreted to infer any meaningful information. No arguments are defined for the BEGIN probe.

### 11.1.2 END Probe

The END probe fires after all other probes. This probe will not fire until all other probe clauses have completed. This probe can be used to process state that has been gathered or to format the output. The \texttt{printa} action is therefore often used in the END probe. The BEGIN and END probes can be used together to measure the total time spent tracing:

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

See Section 3.4, “Data Normalization” and Section 6.2, “printa” for other common uses of the END probe.

As with the BEGIN probe, no arguments are defined for the END probe. The context in which the END probe fires is arbitrary and should not be depended upon.
When tracing with the `bufpolicy` option set to `fill`, adequate space is reserved to accommodate any records traced in the `END` probe. See Section 5.2.3, “fill Policy and END Probes” for details.

**Note**

The `exit` action causes tracing to stop and the `END` probe to fire. However, there is some delay between the invocation of the `exit` action and the `END` probe firing. During this delay, no probes will fire. After a probe invokes the `exit` action, the `END` probe is not fired until the DTrace consumer determines that `exit` has been called and stops tracing. The rate at which the exit status is checked can be set using `statusrate` option. For more information, see Chapter 10, *Options and Tunables*.

### 11.1.3 ERROR Probe

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

```
BEGIN
{
  *(char *)NULL;
}
ERROR
{
  printf("Hit an error!");
}
```

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

```
# dtrace -s error.d
  dtrace: script 'error.d' matched 2 probes
  CPU     ID                    FUNCTION:NAME
           3                           :ERROR Hit an error!
dtrace: error on enabled probe ID 1 (ID 1: dtrace::BEGIN):
  invalid address (0x0) in action #1 at DIF offset 16
^[C
```

The output shows that the `ERROR` probe fired and that `dtrace` reported the error. `dtrace` has its own enabling of the `ERROR` probe to allow it to report errors. Using the `ERROR` probe, you can create your own custom error handling.

The arguments to the `ERROR` probe are shown in the following table:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1</td>
<td>The enabled probe identifier (EPID) of the probe that caused the error</td>
</tr>
<tr>
<td>arg2</td>
<td>The index of the action that caused the fault</td>
</tr>
<tr>
<td>arg3</td>
<td>The DIF offset into the action or -1 if not applicable</td>
</tr>
<tr>
<td>arg4</td>
<td>The fault type</td>
</tr>
<tr>
<td>arg5</td>
<td>Value particular to the fault type</td>
</tr>
</tbody>
</table>

The following table describes the various fault types that can be specified in `arg4` and the values that `arg5` can take for each fault type:

<table>
<thead>
<tr>
<th>arg4 Value</th>
<th>Description</th>
<th>arg5 Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTRACEFLT_UNKNOWN</td>
<td>Unknown fault type</td>
<td>None</td>
</tr>
</tbody>
</table>
arg4 Value | Description | arg5 Meaning
---|---|---
DTRACEFLT_BADADDR | Access to unmapped or invalid address | Address accessed
DTRACEFLT_BADALIGN | Unaligned memory access | Address accessed
DTRACEFLT_ILLOP | Illegal or invalid operation | None
DTRACEFLT_DIVZERO | Integer divide by zero | None
DTRACEFLT_NOSCRATCH | Insufficient scratch space to satisfy scratch allocation | None
DTRACEFLT_KPRIV | Attempt to access a kernel address or property without sufficient privileges | Address accessed or 0 if not applicable
DTRACEFLT_UPRIV | Attempt to access a user address or property without sufficient privileges | Address accessed or 0 if not applicable
DTRACEFLT_TUPOFLOW | DTrace internal parameter stack overflow | None
DTRACEFLT_BADSTACK | Invalid user process stack | Address of invalid stack pointer

If the actions taken in the ERROR probe itself cause an error, that error is silently dropped. The ERROR probe is not recursively invoked.

### 11.1.4 Stability

The dtrace provider uses DTrace’s stability mechanism to describe its stabilities as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Stable</td>
<td>Stable</td>
<td>Common</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Stable</td>
<td>Stable</td>
<td>Common</td>
</tr>
<tr>
<td>Arguments</td>
<td>Stable</td>
<td>Stable</td>
<td>Common</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, see Chapter 16, Stability.

### 11.2 profile Provider

The profile provider provides probes associated with a time-based interrupt firing every fixed, specified time interval. Such unanchored probes are not associated with any particular point of execution, but rather with the asynchronous interrupt event. You can use these probes to sample some aspect of system state and then use the samples to infer system behavior. If the sampling rate is high, or the sampling time is long, an accurate inference is possible. Using DTrace actions, you can use the profile provider to sample practically anything in the system. For example, you could sample the state of the current thread, the state of the CPU, or the current machine instruction.

### 11.2.1 profile-n Probes

profile-n probes fire at a fixed interval at high-interrupt level on all active CPUs. The units of \( n \) default to a frequency expressed as a rate of firing per second, but the value can also have an optional suffix as shown in Table 11.1, “Valid Time Suffixes”, which specifies either a time interval or a frequency.
### Table 11.1 Valid Time Suffixes

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Time Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsec or ns</td>
<td>nanoseconds</td>
</tr>
<tr>
<td>usec or us</td>
<td>microseconds</td>
</tr>
<tr>
<td>msec or ms</td>
<td>milliseconds</td>
</tr>
<tr>
<td>sec or s</td>
<td>seconds</td>
</tr>
<tr>
<td>min or m</td>
<td>minutes</td>
</tr>
<tr>
<td>hour or h</td>
<td>hours</td>
</tr>
<tr>
<td>day or d</td>
<td>days</td>
</tr>
<tr>
<td>hz</td>
<td>hertz (frequency expressed as rate per second)</td>
</tr>
</tbody>
</table>

#### 11.2.2 tick-n Probes

*tick-n* probes fire at fixed intervals at a high interrupt level on only one CPU per interval. Unlike *profile-n* probes, which fire on every CPU, *tick-n* probes fire on only one CPU per interval and the CPU on which they fire can change over time. The units of *n* default to a frequency expressed as a rate of firing per second, but the value can also have an optional time suffix as shown in Table 11.1, "Valid Time Suffixes", which specifies either a time interval or a frequency.

*tick-n* probes have several uses, such as providing some periodic output or taking a periodic action.

![Note]

By default, the highest supported tick frequency is 5000 Hz (*tick-5000*).

#### 11.2.3 Probe Creation

Unlike other providers, the *profile* provider creates probes dynamically on an as-needed basis. Thus, the desired probe might not appear in a listing of all probes (for example, by using `dtrace -l -P profile`) but the probe is created when it is explicitly enabled.

A time interval that is too short causes the machine to continuously field time-based interrupts and denies service on the machine. The *profile* provider silently refuses to create a probe that would result in an interval of less than two hundred microseconds.

#### 11.2.4 Stability

The *profile* provider uses DTrace’s stability mechanism to describe its stabilities as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>Common</td>
</tr>
<tr>
<td>Module</td>
<td>Unstable</td>
<td>Unstable</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Evolving</td>
<td>Evolving</td>
<td>Common</td>
</tr>
<tr>
<td>Arguments</td>
<td>Evolving</td>
<td>Evolving</td>
<td>Common</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, see Chapter 16, *Stability*.
11.3 syscall Provider

The syscall provider makes available a probe at the entry to and return from every system call in the system. Because system calls are the primary interface between user-level applications and the operating system kernel, the syscall provider can offer tremendous insight into application behavior with respect to the system.

11.3.1 Probes

syscall provides a pair of probes for each system call: an entry probe that fires before the system call is entered, and a return probe that fires after the system call has completed but before control has transferred back to user-level. For all syscall probes, the function name is set to be the name of the instrumented system call and the module name is undefined.

Often, the system call names provided by syscall correspond to names in the section 2 manual pages. However, some probes provided by the syscall provider do not directly correspond to any documented system call. The common reasons for this discrepancy are described in the following sections.

11.3.1.1 System Call Anachronisms

In some cases, the name of the system call as provided by the syscall provider may be a reflection of an ancient implementation detail.

11.3.1.2 Subcoded System Calls

Some system calls may be implemented as suboperations of another system call. For example, socketcall() is the common kernel entry point for the socket system calls.

11.3.1.3 New System Calls

Oracle Linux implements at-suffixed system interfaces as individual system calls, for example:

- faccessat()
- fchmodat()
- fchownat()
- fstatat64()
- futimensat()
- linkat()
- mkdirat()
- mknodat()
- name_to_handle_at()
- newfstatat()
- open_by_handle_at()
- openat()
- readlinkat()
- renameat()
Probes

- symlinkat()
- unlinkat()
- utimensat()

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

11.3.1.4 Replaced System Calls

In Oracle Linux, the following old system calls have been replaced and are not called by the newer \texttt{glibc} interfaces. The old interfaces remain, but they are reimplemented not as system calls in their own right, but as calls to the new system calls as indicated:

<table>
<thead>
<tr>
<th>Old Call</th>
<th>New Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{access((p, m))}</td>
<td>\texttt{faccessat(AT_FDCWD, (p, m, 0))}</td>
</tr>
<tr>
<td>\texttt{chmod((p, m))}</td>
<td>\texttt{fchmodat(AT_FDCWD, (p, m, 0))}</td>
</tr>
<tr>
<td>\texttt{chown((p, u, g))}</td>
<td>\texttt{fchownat(AT_FDCWD, (p, u, g, 0))}</td>
</tr>
<tr>
<td>\texttt{creat((p, m))}</td>
<td>\texttt{openat(AT_FDCWD, (p, O_WRONLY\texttt{\mid}O_CREAT\texttt{\mid}O_TRUNC, m))}</td>
</tr>
<tr>
<td>\texttt{fchmod((fd, m))}</td>
<td>\texttt{fchmodat(fd, NULL, (m, 0))}</td>
</tr>
<tr>
<td>\texttt{fchown((fd, u, g))}</td>
<td>\texttt{fchownat(fd, NULL, (u, g, 0))}</td>
</tr>
<tr>
<td>\texttt{fstat((fd, s))}</td>
<td>\texttt{fstatat(fd, NULL, (s, 0))}</td>
</tr>
<tr>
<td>\texttt{lchown((p, u, g))}</td>
<td>\texttt{fchownat(AT_FDCWD, (p, u, g, AT_SYMLINK\texttt{_NOFOLLOW}))}</td>
</tr>
<tr>
<td>\texttt{link((p1, p2))}</td>
<td>\texttt{linkat(AT_FDCWD, (p1, AT_FDCWD, p2, 0))}</td>
</tr>
<tr>
<td>\texttt{lstat((p, s))}</td>
<td>\texttt{fstatat(AT_FDCWD, (p, s, AT_SYMLINK\texttt{_NOFOLLOW}))}</td>
</tr>
<tr>
<td>\texttt{mkdir((p, m))}</td>
<td>\texttt{mkdirat(AT_FDCWD, (p, m))}</td>
</tr>
<tr>
<td>\texttt{mknod((p, m, d))}</td>
<td>\texttt{mknodat(AT_FDCWD, (p, m, d))}</td>
</tr>
<tr>
<td>\texttt{open((p, o, m))}</td>
<td>\texttt{openat(AT_FDCWD, (p, o, m))}</td>
</tr>
<tr>
<td>\texttt{readlink((p, b, s))}</td>
<td>\texttt{readlinkat(AT_FDCWD, (p, b, s))}</td>
</tr>
<tr>
<td>\texttt{rename((p1, p2))}</td>
<td>\texttt{renameat(AT_FDCWD, (p1, AT_FDCWD, p2))}</td>
</tr>
<tr>
<td>\texttt{rmdir((p))}</td>
<td>\texttt{unlinkat(AT_FDCWD, (p, AT_REMOVEDIR))}</td>
</tr>
<tr>
<td>\texttt{stat((p, s))}</td>
<td>\texttt{fstatat(AT_FDCWD, (p, s, 0))}</td>
</tr>
<tr>
<td>\texttt{symlink((p1, p2))}</td>
<td>\texttt{symlinkat(p1, AT_FDCWD, p2)}</td>
</tr>
<tr>
<td>\texttt{unlink((p))}</td>
<td>\texttt{unlinkat(AT_FDCWD, (p, 0))}</td>
</tr>
</tbody>
</table>

11.3.1.5 Large File System Calls

A 32-bit program that supports \textit{large files} that exceed two gigabytes in size must be able to process 64-bit file offsets. Because large files require use of large offsets, large files are manipulated through a parallel set of system interfaces. \textbf{Table 11.2, “syscall Large File Probes”} lists some of the \texttt{syscall} probes for the large file system call interfaces.
Table 11.2 syscall Large File Probes

<table>
<thead>
<tr>
<th>Large File syscall Probe</th>
<th>System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>getdents64</td>
<td>getdents()</td>
</tr>
<tr>
<td>pread64 *</td>
<td>pread()</td>
</tr>
<tr>
<td>pwrite64 *</td>
<td>pwrite()</td>
</tr>
</tbody>
</table>

11.3.1.6 Private System Calls

Some system calls are private implementation details of Oracle Linux subsystems that span the user-kernel boundary.

11.3.2 Arguments

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

11.3.3 Stability

The syscall provider uses DTrace's stability mechanism to describe its stabilities as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>Common</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Unstable</td>
<td>Unstable</td>
<td>ISA</td>
</tr>
<tr>
<td>Name</td>
<td>Evolving</td>
<td>Evolving</td>
<td>Common</td>
</tr>
<tr>
<td>Arguments</td>
<td>Unstable</td>
<td>Unstable</td>
<td>ISA</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, refer to Chapter 16, Stability.

11.4 sdt provider

The Statically Defined Tracing (SDT) provider creates probes at sites that a software programmer has formally designated. The SDT mechanism allows programmers to consciously choose locations of interest to users of DTrace and to convey some semantic knowledge about each location through the probe name.

As the sdt probes that are defined for the Oracle Linux kernel are likely to change over time, they are not listed here. Both the name stability and data stability of the probes are Private, which reflects the kernel's implementation and should not be interpreted as a commitment to preserve these interfaces. For more information about the DTrace stability mechanism, see Chapter 16, Stability.

11.4.1 Creating SDT Probes

If you are a device driver developer, you might be interested in creating your own SDT probes for your Oracle Linux driver. The disabled probe effect of SDT is essentially the cost of several no-operation machine instructions. You are therefore encouraged to add SDT probes to your device drivers as needed. Unless these probes negatively affect performance, you can leave them in your shipping code.

See Chapter 14, Statically Defined Tracing of Kernel Modules.
DTrace also provides a mechanism for application developers to define user-space static probes, as described in Chapter 13, *Statically Defined Tracing of User Applications*.

### 11.4.1.1 Declaring Probes

SDT probes are declared using the `DTRACE_PROBE`, `DTRACE_PROBE1`, `DTRACE_PROBE2`, `DTRACE_PROBE3`, and `DTRACE_PROBE4` macros from `<linux/sdt.h>`. The module name and function name of an SDT-based probe corresponds to the kernel module and function of the probe. The name of the probe depends on the name given in the `DTRACE_PROBE` macro. If the name does not contain two consecutive underscores (`__`), the name of the probe is as written in the macro. If the name contains two consecutive underscores, the probe name converts the consecutive underscores to a single dash (`-`). For example, if a `DTRACE_PROBE` macro specifies `transaction__start`, the SDT probe is named `transaction-start`. This substitution allows C code to provide macro names that are not valid C identifiers without specifying a string.

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

SDT can also act as a metaprovider by registering probes so they appear to come from other providers such as `io`, `proc`, and `sched`, which do not have dedicated modules of their own. For example, `kernel(exit.c)` contains calls to the `DTRACE_PROC` macro, which is defined in `<linux/sdt.h>` as:

```c
#define DTRACE_PROC(name) 
  DTRACE_PROBE(__proc_##name);
```

Probes that use such macros appear to come from a provider other than `sdt`. The leading double underscore, provider name, and trailing underscore in the `name` argument are used to match the provider and are not included in the probe name. At present, you cannot use this functionality to create probes for providers other than those that are hard-coded into DTrace.

### 11.4.1.2 Probe Arguments

The arguments for each SDT probe are the arguments specified in the corresponding `DTRACE_PROBE` macro reference. The number of arguments depends on which macro was used to create the probe: `DTRACE_PROBE1` specifies one argument, `DTRACE_PROBE2` specifies two arguments, and so on. When declaring your SDT probes, you can minimize their disabled probe effect by not dereferencing pointers and not loading from global variables in the probe arguments. Both pointer dereferencing and global variable loading may be done safely in D actions that enable probes, so DTrace users can request these actions only when they are needed.

### 11.4.2 Stability

The `sdt` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Private</td>
<td>Private</td>
<td>ISA</td>
</tr>
<tr>
<td>Arguments</td>
<td>Private</td>
<td>Private</td>
<td>ISA</td>
</tr>
</tbody>
</table>
For more information about the stability mechanism, refer to Chapter 16, Stability.

### 11.5 proc Provider

The proc provider makes available probes pertaining to the following activities: process creation and termination, LWP creation and termination, executing new program images, and sending and handling signals.

#### 11.5.1 Probes

Table 11.3, “proc Probes” lists the proc probes.

**Table 11.3 proc Probes**

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Fires when a process (or process thread) is created using fork() or vfork() (which both invoke clone()). The psinfo_t corresponding to the new child process is pointed to by args[0].</td>
</tr>
<tr>
<td>exec</td>
<td>Fires whenever a process loads a new process image using a variant of the execve() system call. The exec probe fires before the process image is loaded. Process variables like execname and curpsinfo therefore contain the process state before the image is loaded. Some time after the exec probe fires, either the exec-failure or exec-success probe subsequently fires in the same thread. The path of the new process image is pointed to by args[0].</td>
</tr>
<tr>
<td>exec-failure</td>
<td>Fires when an exec() variant has failed. The exec-failure probe fires only after the exec probe has fired in the same thread. The errno value is provided in args[0].</td>
</tr>
<tr>
<td>exec-success</td>
<td>Fires when an exec() variant has succeeded. Like the exec-failure probe, the exec-success probe fires only after the exec probe has fired in the same thread. By the time that the exec-success probe fires, process variables like execname and curpsinfo contain the process state after the new process image has been loaded.</td>
</tr>
<tr>
<td>exit</td>
<td>Fires when the current process is exiting. The reason for exit, which is expressed as one of the SIGCHLD &lt;asm-generic/signal.h&gt; codes, is contained in args[0].</td>
</tr>
<tr>
<td>lwp-create</td>
<td>Fires when a process thread is created, the latter typically as a result of pthread_create(). The lwpinfo_t corresponding to the new thread is pointed to by args[0]. The psinfo_t of the process that created the thread is pointed to by args[1].</td>
</tr>
<tr>
<td>lwp-exit</td>
<td>Fires when a process or process thread is exiting, due either to a signal or to an explicit call to exit or pthread_exit().</td>
</tr>
<tr>
<td>lwp-start</td>
<td>Fires within the context of a newly created process or process thread. The lwp-start probe fires before any user-level instructions are executed. If the thread is the first created for the process, the start probe fires, followed by lwp-start.</td>
</tr>
<tr>
<td>signal-clear</td>
<td>Probes that fires when a pending signal is cleared because the target thread was waiting for the signal in sigwait(), sigwaitinfo(), or sigtimedwait(). Under these conditions, the pending signal is cleared and the signal number is returned to the caller. The signal number is in args[0]. signal-clear fires in the context of the formerly waiting thread.</td>
</tr>
</tbody>
</table>
### Arguments

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal-discard</td>
<td>Fires when a signal is sent to a single-threaded process, and the signal is both unblocked and ignored by the process. Under these conditions, the signal is discarded on generation. The <code>lwpsinfo_t</code> and <code>psinfo_t</code> of the target process and thread are in <code>args[0]</code> and <code>args[1]</code>, respectively. The signal number is in <code>args[2]</code>.</td>
</tr>
<tr>
<td>signal-handle</td>
<td>Fires immediately before a thread handles a signal. The <code>signal-handle</code> probe fires in the context of the thread that will handle the signal. The signal number is in <code>args[0]</code>. A pointer to the <code>siginfo_t</code> structure that corresponds to the signal is in <code>args[1]</code>. The address of the signal handler in the process is in <code>args[2]</code>.</td>
</tr>
<tr>
<td>signal-send</td>
<td>Fires when a signal is sent to a process or to a thread created by a process. The <code>signal-send</code> probe fires in the context of the sending process or thread. The <code>lwpsinfo_t</code> and <code>psinfo_t</code> of the receiving process and thread are in <code>args[0]</code> and <code>args[1]</code>, respectively. The signal number is in <code>args[2]</code>. <code>signal-send</code> is always followed by <code>signal-handle</code> or <code>signal-clear</code> in the receiving process and thread.</td>
</tr>
<tr>
<td>start</td>
<td>Fires in the context of a newly created process. The <code>start</code> probe fires before any user-level instructions are executed in the process.</td>
</tr>
</tbody>
</table>

**Note**

In Oracle Linux, there is no fundamental difference between a process and a thread that a process creates. The threads of a process are set up so that they can share resources, but each thread has its own entry in the process table with its own process ID.

### 11.5.2 Arguments

Table 11.4, "proc Probe Arguments" lists the argument types for the `proc` probes. See Table 11.3, "proc Probes" for a description of the arguments.

**Table 11.4 proc Probe Arguments**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td><code>psinfo_t *</code></td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>exec</td>
<td><code>char *</code></td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>exec-failure</td>
<td><code>int</code></td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>exec-success</td>
<td>__</td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>exit</td>
<td><code>int</code></td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>lwp-create</td>
<td><code>lwpsinfo_t *</code></td>
<td><code>psinfo_t *</code></td>
<td>__</td>
</tr>
<tr>
<td>lwp-exit</td>
<td>__</td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>lwp-start</td>
<td>__</td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>signal-clear</td>
<td><code>int</code></td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>signal-discard</td>
<td><code>lwpsinfo_t *</code></td>
<td><code>psinfo_t *</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td>signal-handle</td>
<td><code>int</code></td>
<td><code>siginfo_t *</code></td>
<td><code>void (*)(void)</code></td>
</tr>
<tr>
<td>signal-send</td>
<td><code>lwpsinfo_t *</code></td>
<td><code>psinfo_t *</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td>start</td>
<td>__</td>
<td>__</td>
<td>__</td>
</tr>
</tbody>
</table>
11.5.3 lwpsinfo_t

Several proc probes have arguments of type lwpsinfo_t. The definition of the lwpsinfo_t structure as available to DTrace consumers is as follows:

```c
typedef struct lwpsinfo {
    int pr_flag; /* flags */
    id_t pr_lwpid; /* thread id */
    uintptr_t pr_addr; /* internal address of thread */
    uintptr_t pr_wchan; /* not supported */
    char pr_type; /* not supported */
    char pr_state; /* numeric thread state */
    char pr_sname; /* printable character for pr_state */
    int pr_pri; /* priority, high value = high priority */
    char pr_name[PRCLSZ]; /* scheduling class name */
    processorid_t pr_onpro; /* processor which last ran this thread */
} lwpsinfo_t;
```

**Note**

Lightweight processes do not exist on Oracle Linux. In Oracle Linux, processes and threads are both represented by process descriptors of type struct task_struct in the task list. DTrace translates the members of lwpsinfo_t from the task_struct for the Oracle Linux process.

pr_flag is set to 1 if the thread is stopped. Otherwise, it is set to 0.

Table 11.5, “pr_state Values” lists the values that pr_state can take, together with the corresponding character values for pr_sname.

**Table 11.5 pr_state Values**

<table>
<thead>
<tr>
<th>pr_state Value</th>
<th>pr_sname Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRUN (2)</td>
<td>R</td>
<td>The thread is runnable or is currently running on a CPU. The sched:::enqueue probe fires immediately before a thread's state is transitioned to SRUN. The sched:::on-cpu probe will fire a short time after the thread starts to run. The equivalent Oracle Linux task state is TASK_RUNNING.</td>
</tr>
<tr>
<td>SSLEEP (1)</td>
<td>S</td>
<td>The thread is sleeping. The sched:::sleep probe will fire immediately before a thread's state is transitioned to SSLEEP. The equivalent Oracle Linux task state is TASK_INTERRUPTABLE or TASK_UNINTERRUPTABLE.</td>
</tr>
<tr>
<td>SSTOP (4)</td>
<td>T</td>
<td>The thread is stopped, either due to an explicit proc directive or some other stopping mechanism. The equivalent Oracle Linux task state is ___TASK_STOPPED or ___TASK_TRACED.</td>
</tr>
<tr>
<td>SWAIT (7)</td>
<td>W</td>
<td>The thread is waiting on wait queue. The sched:::cpucaps-sleep probe will fire immediately before the thread's state transitions to SWAIT. The equivalent Oracle Linux task state is TASK_WAKEKILL or TASK_WAKING.</td>
</tr>
</tbody>
</table>
### 11.5.4 psinfo_t

Several `proc` probes have an argument of type `psinfo_t`. The definition of the `psinfo_t` structure as available to DTrace consumers is as follows:

```c
typedef struct psinfo {
    int pr_flag;                    /* process flags (deprecated) */
    int pr_nlwp;                    /* not supported */
    pid_t pr_pid;                   /* unique process id */
    pid_t pr_ppid;                  /* process id of parent */
    pid_t pr_pgid;                  /* pid of process group leader */
    pid_t pr_sid;                   /* session id */
    uid_t pr_uid;                   /* real user id */
    uid_t pr_euid;                  /* effective user id */
    uid_t pr_gid;                   /* real group id */
    uid_t pr_egid;                  /* effective group id */
    uintptr_t pr_addr;              /* address of process */
    size_t pr_size;                 /* not supported */
    size_t pr_rssize;               /* not supported */
    struct tty_struct *pr_ttydev;   /* controlling tty (or -1) */
    ushort_t pr_pctcpu;             /* not supported */
    ushort_t pr_pctmem;             /* not supported */
    timestruc_t pr_start;           /* process start time */
    timestruc_t pr_time;            /* not supported */
    timestruc_t pr_ctime;           /* not supported */
    char pr_fname[16];              /* name of exec'ed file */
    char pr_psargs[80];             /* initial chars of arg list */
    int pr_wstat;                   /* not supported */
    int pr_argc;                    /* initial argument count */
    uintptr_t pr_argv;              /* address of initial arg vector */
    uintptr_t pr_envp;              /* address of initial env vector */
    char pr_dmodel;                 /* data model */
    taskid_t pr_taskid;             /* not supported */
    projid_t pr_projid;             /* not supported */
    int pr_nzomb;                   /* not supported */
    poolid_t pr_poolid;             /* not supported */
    zoneid_t pr_zoneid;             /* not supported */
    id_t pr_contract;               /* not supported */
    lwpsinfo_t pr_lwp;              /* not supported */
} psinfo_t;
```

**Note**

Lightweight processes do not exist on Oracle Linux. In Oracle Linux, processes and threads are both represented by process descriptors of type `struct task_struct` in the task list. DTrace translates the members of `psinfo_t` from the `task_struct` for the Oracle Linux process.

`pr_dmodel` is set to either `PR_MODEL_ILP32`, denoting a 32–bit process, or `PR_MODEL_LP64`, denoting a 64–bit process.

### 11.5.5 Examples

The following sections provide examples of using probes published by the `proc` provider.
11.5.5.1 exec

You can use the exec probe to easily determine which programs are being executed, and by whom, as shown in the following example:

```csharp
#pragma D option quiet

proc:::exec
{
    self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
    @[self->parent, execname] = count();
    self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
    self->parent = NULL;
}

END
{
    printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
    printa("%-20s %-20s %@d\n", @);
}
```

Running the example script for a short period of time results in output similar to the following example:

```bash
# dtrace -s ./whoexec.d
^C

WHO                  WHAT                 COUNT
abrtd                abrt-handle-eve      1
firefox              basename             1
firefox              mkdir                1
firefox              mozilla-plugin-      1
firefox              mozilla-xremote      1
firefox              run-mozilla.sh       1
firefox              uname                1
gnome-panel          firefox              1
kworker/u:1          modprobe             1
modprobe             modprobe.ksplice      1
mozilla-plugin-      plugin-config        1
mozilla-plugin-      plugin-conf         1
mozilla-plugin-      plugin-conf         1
nice                 sosreport            1
run-mozilla.sh       basename             1
run-mozilla.sh       dirname              1
run-mozilla.sh       firefox              1
run-mozilla.sh       firefox              1
run-mozilla.sh       name                 1
sh                   abrt-action-save     1
sh                   blkid                1
sh                   brctl                1
sh                   cut                  1
...
```

11.5.5.2 start and exit

If you want to know how long programs are running from creation to termination, you can enable the start and exit probes, as shown in the following example:

```csharp
proc:::start
{
```
Running the example script for several seconds on a build server results in output similar to the following example:

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

```
 cc
 value  ------------- Distribution ------------- count
 33554432 |                                         0
 67108864 |@@@                                      3
 134217728 |@                                        1
 268435456 |                                         0
 536870912 |@@@@                                     4
 1073741824 |@@@@@@@@@@@@@@                           13
 2147483648 |@@@@@@@@@@@@                             11
 4294967296 |@@@                                      3
 8589934592 |                                         0

 sh
 value  ------------- Distribution ------------- count
 262144 |                                         0
 524288 |@                                        5
 1048576 |@@@@@@@                                  29
 2097152 |                                         0
 4194304 |                                         0
 8388608 |@@@                                      12
 16777216 |@@                                       9
 33554432 |@@                                       9
 67108864 |@@                                       9
 134217728 |@@                                      17
 268435456 |@@@@                                     20
 536870912 |@@@@@@                                   26
 1073741824 |@@@@@@                                   26
 2147483648 |@@@                                      14
 4294967296 |@@                                       11
 8589934592 |                                         3
 17179869184 |                                         1
```

### 11.5.5.3 signal-send

You can use the `signal-send` probe to determine the sending and receiving process associated with any signal, as shown in the following example:

```
#pragma D option quiet
proc:::signal-send
{
 @execname, stringof(args[1]->pr_fname), args[2] = count();
}
END
{
 printf("%20s %20s %12s %s\n",
  "SENDER", "RECIPIENT", "SIG", "COUNT");
```

131
11.5.6 Stability

The proc provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Arguments</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, see Chapter 16, Stability.

11.6 sched Provider

The sched provider makes available probes related to CPU scheduling. Because CPUs are the one resource that all threads must consume, the sched provider is very useful for understanding systemic behavior. For example, using the sched provider, you can understand when and why threads sleep, run, change priority, or wake other threads.

11.6.1 Probes

Table 11.6, “sched Probes” lists the sched probes.

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>change-pri</td>
<td>Fires whenever a thread's priority is about to be changed. The lwpsinfo_t of the thread is pointed to by args[0]. The thread's current priority is in the pr_pri field of this structure. The psinfo_t of the process containing the thread is pointed to by args[1]. The thread's new priority is contained in args[2].</td>
</tr>
<tr>
<td>dequeue</td>
<td>Fires immediately before a runnable thread is dequeued from a run queue. The lwpsinfo_t of the thread being dequeued is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1]. The cpuinfo_t of the CPU from which the thread is being dequeued is pointed to by args[2]. If the thread is being dequeued from a run queue that is not associated with a particular CPU, the cpu_id member of this structure will be -1.</td>
</tr>
</tbody>
</table>
| enqueue | Fires immediately before a runnable thread is enqueued to a run queue. The lwpsinfo_t of the thread being enqueued is pointed to by args[0]. The psinfo_t of the process containing the thread is pointed to by args[1]. The
<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probes</td>
<td></td>
</tr>
<tr>
<td>cpuinfo_t</td>
<td>of the CPU to which the thread is being enqueued is pointed to by args[2]. If the thread is being enqueued from a run queue that is not associated with a particular CPU, the cpu_id member of this structure will be -1. The value in args[3] is a boolean indicating whether the thread will be enqueued to the front of the run queue. The value is non-zero if the thread will be enqueued at the front of the run queue, and zero if the thread will be enqueued at the back of the run queue.</td>
</tr>
<tr>
<td>off-cpu</td>
<td>Fires when the current CPU is about to end execution of a thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is ending execution. The lwpsinfo_t structure of the thread that the current CPU will next execute is pointed to by args[0]. The psinfo_t of the process containing the next thread is pointed to by args[1].</td>
</tr>
<tr>
<td>on-cpu</td>
<td>Fires when a CPU has just begun execution of a thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is beginning execution. The curpsinfo variable describes the process containing the current thread.</td>
</tr>
<tr>
<td>preempt</td>
<td>Fires immediately before the current thread is preempted. After this probe fires, the current thread will select a thread to run and the off-cpu probe will fire for the current thread. In some cases, a thread on one CPU will be preempted, but the preempting thread will run on another CPU in the meantime. In this situation, the preempt probe will fire, but the dispatcher will be unable to find a higher priority thread to run and the remain-cpu probe will fire instead of the off-cpu probe.</td>
</tr>
<tr>
<td>remain-cpu</td>
<td>Fires when a scheduling decision has been made, but the dispatcher has elected to continue to run the current thread. The curcpu variable indicates the current CPU. The curlwpsinfo variable indicates the thread that is beginning execution. The curpsinfo variable describes the process containing the current thread.</td>
</tr>
<tr>
<td>sleep</td>
<td>Fires immediately before the current thread sleeps on a synchronization object. The type of the synchronization object is contained in the pr_stype member of the lwpsinfo_t pointed to by curlwpsinfo. The address of the synchronization object is contained in the pr_wchan member of the lwpsinfo_t pointed to by curlwpsinfo. The meaning of this address is a private implementation detail, but the address value may be treated as a token unique to the synchronization object.</td>
</tr>
<tr>
<td>surrender</td>
<td>Fires when a CPU has been instructed by another CPU to make a scheduling decision — often because a higher-priority thread has become runnable.</td>
</tr>
<tr>
<td>tick</td>
<td>Fires as a part of clock tick-based accounting. In clock tick-based accounting, CPU accounting is performed by examining which threads and processes are running when a fixed-interval interrupt fires. The lwpsinfo_t that corresponds to the thread that is being assigned CPU time is pointed to by args[0]. The psinfo_t that corresponds to the process that contains the thread is pointed to by args[1].</td>
</tr>
<tr>
<td>wakeup</td>
<td>Fires immediately before the current thread wakes a thread sleeping on a synchronization object. The lwpsinfo_t of the sleeping thread is pointed to by args[0]. The psinfo_t of the process containing the sleeping thread is pointed to by args[1]. The type of the synchronization object is contained in the pr_stype member of the lwpsinfo_t of the sleeping thread. The address of the synchronization object is contained in the pr_wchan member of the lwpsinfo_t of the sleeping thread. The meaning of this address is a private implementation detail, but the address value may be treated as a token unique to the synchronization object.</td>
</tr>
</tbody>
</table>
11.6.2 Arguments

Table 11.7, “sched Probe Arguments” lists the argument types for the sched probes; Table 11.6, “sched Probes” describes the arguments.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>change-pri</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>pri_t</td>
<td>—</td>
</tr>
<tr>
<td>dequeue</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>cpuinfo_t*</td>
<td>—</td>
</tr>
<tr>
<td>enqueue</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>cpuinfo_t*</td>
<td>int</td>
</tr>
<tr>
<td>off-cpu</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>on-cpu</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>preempt</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>remain-cpu</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>sleep</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>surrender</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>tick</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>wakeup</td>
<td>lwpsinfo_t*</td>
<td>psinfo_t*</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

11.6.3 cpuinfo_t

The cpuinfo_t structure defines a CPU. As Table 11.7, “sched Probe Arguments” indicates, arguments to both the enqueue and dequeue probes include a pointer to a cpuinfo_t. Additionally, the cpuinfo_t corresponding to the current CPU is pointed to by the curcpu variable. The definition of the cpuinfo_t structure is as follows:

```c
typedef struct cpuinfo {
    processorid_t cpu_id;  /* CPU identifier */
    psetid_t cpu_pset;     /* not supported */
    chipid_t cpu_chip;     /* chip identifier */
    lgrp_id_t cpu_lgrp;    /* not supported */
    processor_info_t cpu_info; /* CPU information */
} cpuinfo_t;
```

- `cpu_id` is the processor identifier.
- `cpu_chip` is the identifier of the physical chip. Physical chips may contain several CPU cores.
- `cpu_info` is the processor_info_t structure associated with the CPU.

11.6.4 Examples

The following sections contain examples of using sched probes.

11.6.4.1 on-cpu and off-cpu

One common question you might want answered is which CPUs are running threads and for how long. You can use the on-cpu and off-cpu probes to easily answer this question on a system-wide basis as shown in the following example:
Examples

```c
sched::on-cpu
{
    self->ts = timestamp;
}

sched::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

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

```bash
# dtrace -s ./where.d
dtrace: script './where.d' matched 5 probes
^C
0
  value  ------------ Distribution ------------ count
  2048 |                                     0
  4096 |@@@                                      37
  8192 |@@@@@@@@@@@@@@@@@@@                    212
  16384 |@@                                      30
  32768 |@@                                      10
  65536 |@@                                     17
  131072 |@@                                     12
  262144 |@@                                     9
  524288 |@@                                      6
 1048576 |@@                                      5
 2097152 |@@                                      1
 4194304 |@@                                      3
  8388608 |@@@@                                    75
 16777216 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              201
 33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@              21
 67108864 |@@@                                      6

1
  value  ------------ Distribution ------------ count
  2048 |                                     0
  4096 |@@                                      6
  8192 |@@@@                                     23
 16384 |@@@@                                     18
 32768 |@@@@                                     22
 65536 |@@@@                                     22
 131072 |@@                                      7
 262144 |@@                                      5
 524288 |@@                                      2
 1048576 |@@                                      3
 2097152 |@@                                      9
 4194304 |@@                                      4
 8388608 |@@@@                                     18
16777216 |@@@@                                      19
 33554432 |@@@@                                      16
 67108864 |@@@@                                      21
 134217728 |@@                                      14
 268435456 |@@                                      0
```

This output shows that on CPU 1 threads tend to run for less than 100 microseconds at a stretch, or for approximately 10 milliseconds. A notable gap between the two clusters of data is shown in the histogram. You also might be interested in knowing which CPUs are running a particular process. You can use the `on-cpu` and `off-cpu` probes for answering this question as well. The following script displays which CPUs run a specified application over a period of ten seconds:

```bash`
#pragma D option quiet
dtrace::BEGIN
{    
```
start = timestamp;
}

sched:::on-cpu
/execname == $$1/
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = sum(timestamp - self->ts);
    self->ts = 0;
}

profile:::tick-1sec
/++x == 10/
{
    exit(0);
}

dtrace:::END
{
    printf("CPU distribution over %d seconds:\n\n",
        (timestamp - start) / 1000000000);
    printf("CPU microseconds\n--- ------------
"); normalize(0, 1000);
    printa("%3d %@d
", @);
}

Running the above script on a large mail server and specifying the IMAP daemon results in output similar to the following example:

```
# dtrace -s ./whererun.d imapd
CPU distribution of imapd over 10 seconds:

CPU microseconds
--- ------------
  15 10102
  12 16377
  21 25317
  19 25504
  17 35653
  13 41539
  14 46669
  20 57753
  22 70088
  16 115860
  23 127775
  18 160517
```

Oracle Linux takes into account the amount of time that a thread has been sleeping when selecting a CPU on which to run the thread: a thread that has been sleeping for less time tends not to migrate. You can use the off-cpu and on-cpu probes to observe this behavior:

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
    self->cpu = cpu;
    self->ts = timestamp;
}
sched:::on-cpu
/self->ts/
{
```


```c

void{
    if (self->cpu == cpu ?
        "sleep time, no CPU migration": "sleep time, CPU migration") ==
    lquantize((timestamp - self->ts) / 1000000, 0, 500, 25);
    self->ts = 0;
    self->cpu = 0;
}
```

Running the above script for approximately 30 seconds results in output similar to the following example:

```
$ dtrace -s ./howlong.d
```

```
dtrace: script './howlong.d' matched 5 probes

^C

sleep time, CPU migration

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>@@@@</td>
<td>6838</td>
</tr>
<tr>
<td>25</td>
<td>@@@@</td>
<td>4714</td>
</tr>
<tr>
<td>50</td>
<td>@@</td>
<td>3108</td>
</tr>
<tr>
<td>75</td>
<td>@</td>
<td>1304</td>
</tr>
<tr>
<td>100</td>
<td>@</td>
<td>1557</td>
</tr>
<tr>
<td>125</td>
<td>@</td>
<td>1425</td>
</tr>
<tr>
<td>150</td>
<td>@</td>
<td>894</td>
</tr>
<tr>
<td>175</td>
<td>@</td>
<td>1526</td>
</tr>
<tr>
<td>200</td>
<td>@</td>
<td>2010</td>
</tr>
<tr>
<td>225</td>
<td>@</td>
<td>1933</td>
</tr>
<tr>
<td>250</td>
<td>@</td>
<td>1982</td>
</tr>
<tr>
<td>275</td>
<td>@</td>
<td>2051</td>
</tr>
<tr>
<td>300</td>
<td>@</td>
<td>2021</td>
</tr>
<tr>
<td>325</td>
<td>@</td>
<td>1708</td>
</tr>
<tr>
<td>350</td>
<td>@</td>
<td>1113</td>
</tr>
<tr>
<td>375</td>
<td>@</td>
<td>502</td>
</tr>
<tr>
<td>400</td>
<td>@</td>
<td>220</td>
</tr>
<tr>
<td>425</td>
<td>@</td>
<td>106</td>
</tr>
<tr>
<td>450</td>
<td>@</td>
<td>54</td>
</tr>
<tr>
<td>475</td>
<td>@</td>
<td>40</td>
</tr>
<tr>
<td>&gt;= 500</td>
<td>@</td>
<td>1716</td>
</tr>
</tbody>
</table>
```

```
sleep time, no CPU migration

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>@@@@@@@@@@@@@@</td>
<td>58413</td>
</tr>
<tr>
<td>25</td>
<td>@@@@@@</td>
<td>14793</td>
</tr>
<tr>
<td>50</td>
<td>@@@</td>
<td>10050</td>
</tr>
<tr>
<td>75</td>
<td>@@</td>
<td>3858</td>
</tr>
<tr>
<td>100</td>
<td>@</td>
<td>6242</td>
</tr>
<tr>
<td>125</td>
<td>@</td>
<td>6555</td>
</tr>
<tr>
<td>150</td>
<td>@</td>
<td>3980</td>
</tr>
<tr>
<td>175</td>
<td>@</td>
<td>5987</td>
</tr>
<tr>
<td>200</td>
<td>@</td>
<td>9024</td>
</tr>
<tr>
<td>225</td>
<td>@</td>
<td>9070</td>
</tr>
<tr>
<td>250</td>
<td>@</td>
<td>10745</td>
</tr>
<tr>
<td>275</td>
<td>@</td>
<td>11898</td>
</tr>
<tr>
<td>300</td>
<td>@</td>
<td>11704</td>
</tr>
<tr>
<td>325</td>
<td>@</td>
<td>10846</td>
</tr>
<tr>
<td>350</td>
<td>@</td>
<td>6962</td>
</tr>
<tr>
<td>375</td>
<td>@</td>
<td>3292</td>
</tr>
<tr>
<td>400</td>
<td>@</td>
<td>1713</td>
</tr>
<tr>
<td>425</td>
<td>@</td>
<td>585</td>
</tr>
<tr>
<td>450</td>
<td>@</td>
<td>201</td>
</tr>
<tr>
<td>475</td>
<td>@</td>
<td>96</td>
</tr>
<tr>
<td>&gt;= 500</td>
<td>@</td>
<td>3946</td>
</tr>
</tbody>
</table>
```

The example output shows that there are many more occurrences of non-migration than migration. Also, when sleep times are longer, migrations are more likely. The distributions are noticeably different in the sub 100 millisecond range, but look very similar as the sleep times get longer. This result would seem to indicate that sleep time is not factored into the scheduling decision once a certain threshold is exceeded.
11.6.4.2 enqueue and dequeue

When a CPU becomes idle, the dispatcher looks for work enqueued on other (non-idle) CPUs. The following sample code uses the dequeue probe to understand how often applications are transferred and by which CPU:

```c
#pragma D option quiet
sched:::dequeue
/args[2]->cpu_id != -1 && cpu != args[2]->cpu_id &&
(curlwpsinfo->pr_flag & PR_IDLE)/
{
    @[stringof(args[1]->pr_fname), args[2]->cpu_id] =
    lquantize(cpu, 0, 100);
}
END
[
    printa("%s stolen from CPU %d by: 
%@d
", );
]
```

The following output is an extract from running the above script on a 4-CPU system:

```
# dtrace -s ./whosteal.d
^C...
nscd stolen from CPU 1 by:
   value   Distribution   count
    1       --------------  --
    2       @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Instead of knowing which CPUs took which work, you might want to know the CPUs on which processes and threads are waiting to run. You can use the enqueue and dequeue probes together to answer this question:

```c
sched:::enqueue
[
    a[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id] =
    timestamp;
]
sched:::dequeue
/a[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id]/
{
    @[stringof(args[1]->pr_lwpid), args[2]->cpu_id] =
    quantize(timestamp -
        a[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id));
    a[0]->pr_lwpid, args[1]->pr_pid, args[2]->cpu_id] = 0;
}
```

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

```
# dtrace -s qtime.d
dtrace: script 'qtime.d' matched 2 probes
```
Instead of looking at wait times, you might want to examine the length of the run queue over time. Using the `enqueue` and `dequeue` probes, you can set up an associative array to track the queue length:

```
sched:::enqueue
{
    this->len = qlen[args[2]->cpu_id]++;
    @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
}
```

```
sched:::dequeue
{/qlen[args[2]->cpu_id]/
{
    qlen[args[2]->cpu_id]--;
}
```

Running this script for approximately 30 seconds on a largely idle dual-core processor system results in output similar to the following example:

```
dtrace -s qlen.d
```

```
dtrace: script 'qlen.d' matched 2 probes
```

```
^C
```

```
 1

    value  -------------- Distribution -------------- count
    < 0     |                                         0
    0 @     | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
The output is roughly what you would expect for an idle system: the majority of the time that a runnable thread is enqueued, the run queues were very short (three or fewer threads in length). However, given that the system was largely idle, the exceptional data points at the bottom of each table might be unexpected. For example, why were the run queues as long as 8 runnable threads? To explore this question, you could write a D script that displays the contents of the run queue when the length of the run queue is long. This problem is complicated because D enablings cannot iterate over data structures, and therefore cannot simply iterate over the entire run queue. Even if D enablings could do so, you should avoid dependencies on the kernel's internal data structures.

For this type of script, you would enable the `enqueue` and `dequeue` probes and use both speculations and associative arrays. Whenever a thread is enqueued, the script increments the length of the queue and records the timestamp in an associative array keyed by the thread. You cannot use a thread-local variable in this case because a thread might be enqueued by another thread. The script then checks to see if the queue length exceeds the maximum. If it does, the script starts a new speculation, and records the timestamp and the new maximum. Then, when a thread is dequeued, the script compares the enqueue timestamp to the timestamp of the longest length: if the thread was enqueued before the timestamp of the longest length, the thread was in the queue when the longest length was recorded. In this case, the script speculatively traces the thread's information. Once the kernel dequeues the last thread that was enqueued at the timestamp of the longest length, the script commits the speculation data. This script is shown below:

```d
#pragma D option quiet
#pragma D option nspec=4
#pragma D option specsize=100k

int maxlen;
int spec[int];
sched::enqueue
{   this->len = ++qlen[this->cpu = args[2]->cpu_id];
    in[args[0]->pr_addr] = timestamp;
}
sched::enqueue
/this->len > maxlen && spec[this->cpu]/
{   /*
      * There is already a speculation for this CPU. We just set a new
      * record, so we’ll discard the old one.
      */
      discard(spec[this->cpu]);
}
sched::enqueue
/this->len > maxlen/
{   /*
      * We have a winner. Set the new maximum length and set the timestamp
      * of the longest length.
      */
```
Running the above script on the same system results in output similar to the following example:

```
# dtrace -s whoqueue.d
Run queue of length 1:
  2850/2850 (java)
Run queue of length 2:
  4034/4034 (kworker/0:1)
  16/16 (sync_supers)
Run queue of length 3:
  10/10 (ksoftirqd/1)
  1710/1710 (hald-addon-inpu)
  25350/25350 (dtrace)
Run queue of length 4:
  2852/2852 (java)
  2850/2850 (java)
  1710/1710 (hald-addon-inpu)
  2099/2099 (Xorg)
Run queue of length 5:
  3149/3149 (notification-da)
  2417/2417 (gnome-settings-)
  2437/2437 (gnome-panel)
  2461/2461 ( wnck-applet)
  2432/2432 (metacity)
Run queue of length 9:
  3685/3685 (firefox)
  3149/3149 (notification-da)
  2417/2417 (gnome-settings-)
```
11.6.4.3 sleep and wakeup

You can use the `wakeup` probe to determine what is waking a particular process, and when over a given period, as shown in the following example:

```c
#pragma D option quiet
dtrace:::BEGIN
{
  start = timestamp;
}

sched:::wakeup
/stringof(args[1]->pr_fname) == "gnome-terminal"/
{
  @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);
}

profile:::tick-1sec
/++x == 10/
{
  exit(0);
}
```

The output from running this script is shown below:

```
# dtrace -s gterm.d
Xorg

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>@@@@@@@@@@@@@@@@</td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td>@@@@@@@@@</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>@@@@@@@@@</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>@@@@@@@@@</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>@@@@@@@@@</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>@@@</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>@@@</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>@@</td>
<td>15</td>
</tr>
<tr>
<td>&gt;= 10</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

The output shows that the X server is waking the `gnome-terminal` process, as you interact with the system.

You can additionally use the `sleep` probe along with the `wakeup` probe to understand which applications are blocking on which applications, and for how long, as shown in the following example:

```c
#pragma D option quiet
sched:::sleep
{
  bedtime[curlwpinfo->pr_addr] = timestamp;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
The tail of the output from running the example script for several seconds on a desktop system resembles the following example:

```
# dtrace -s whofor.d
^C
Xorg sleeping on metacity:
    value  ------------- Distribution ------------- count
    0       0
    2       2

Xorg sleeping on metacity:
    value  ------------- Distribution ------------- count
    0       0
    2       2

11.6.4.4 preempt and remain-cpu

Because Oracle Linux is a preemptive system, higher priority threads preempt lower priority ones. Preemption can induce a significant latency bubble in the lower priority thread, so you might want to know which threads are being preempted by which other threads. The following example shows how to use the preempt and remain-cpu probes to display this information:

```
#pragma D option quiet
sched:::preempt
{
    self->preempt = 1;
}

sched:::remain-cpu
/self->preempt/
{
    self->preempt = 0;
}

sched:::off-cpu
/self->preempt/
{
    /*
     * If we were told to preempt ourselves, see who we ended up giving
     * the CPU to.
     */
    @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname, curlwpinfo->pr_pri] = count();
    self->preempt = 0;
}
END
{
    printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",}
Running the above script for several seconds on a desktop system results in output similar to the following example:

```
# dtrace -s whopreempt.d
^C
PREEMPTOR PRI  PREEMPTED PRI  #
   firefox   120  kworker/0:0 120  1
  gnome-panel 120   swapper 120  1
  gnome-panel 120   wnck-applet 120  1
    jbd2/dm-0-8 120   swapper 120  1
   khugepaged  139  kworker/0:0 120  1
  ksoftirqd/1  120   kworker/0:0 120  1
   kworker/0:0  120   gnome-terminal 120  1
   kworker/0:2  120   Xorg 120  1
   kworker/0:2  120   java 120  1
   kworker/1:0  120   Xorg 120  1
   nautilus   120     Xorg 120  1
  rtkit-daemon  0     rtkit-daemon 120  1
  rtkit-daemon 120     swapper 120  1
   watchdog/0  0     swapper 120  1
   watchdog/1  0   kworker/0:0 120  1
   wnck-applet 120     Xorg 120  1
   wnck-applet 120     swapper 120  1
   automount  120   kworker/0:0 120  2
    gnome-power-man 120   kworker/0:0 120  2
   kworker/0:0  120    swapper 120  2
   kworker/1:0  120    dtrace 120  2
   metacity   120   kworker/0:0 120  2
    notification-da  120    swapper 120  2
   udisks-daemon  120   kworker/0:0 120  2
    automount  120    swapper 120  3
    gnome-panel 120       Xorg 120  3
    gnome-settings-  120       Xorg 120  3
    gnome-settings-  120    swapper 120  3
    gnome-terminal  120    swapper 120  3
     java   120   kworker/0:0 120  3
   ksoftirqd/0  120     swapper 120  3
   kworker/0:2  120     swapper 120  3
   metacity   120       Xorg 120  3
    nautilus   120   kworker/0:0 120  3
     qpid  120     swapper 120  3
     metacity  120     swapper 120  4
    gvfs-afc-volume  120    swapper 120  5
     java   120       Xorg 120  5
    notification-da  120       Xorg 120  5
    notification-da  120   kworker/0:0 120  5
     Xorg   120   kworker/0:0 120  6
  wnck-applet  120   kworker/0:0 120  10
 VBoxService  120     swapper 120  13
   dtrace   120     swapper 120  14
   kworker/1:0  120   kworker/0:0 120  16
   dtrace   120   kworker/0:0 120  20
    Xorg   120     swapper 120  90
    hald-addon-input  120    swapper 120 100
     java   120     swapper 120 108
    gnome-terminal  120   kworker/0:0 120 110
```

### 11.6.4.5 tick

If **NOHZ** is set to **off**, Oracle Linux uses *tick-based CPU accounting*, in which a system clock interrupt fires at a fixed interval and attributes CPU utilization to the processes running at the time of the tick. The following example shows how to use the **tick** probe to observe this attribution:
One deficiency of tick-based accounting is that the system clock that performs accounting is often also responsible for dispatching any time-related scheduling activity. As a result, if a thread is to perform some amount of work every clock tick (that is, every 10 milliseconds), the system either over-accounts or under-accounts for the thread, depending on whether the accounting is done before or after time-related dispatching scheduling activity. If accounting is performed before time-related dispatching, the system under-accounts for threads running at a regular interval. If such threads run for less than the clock tick interval, they can effectively hide behind the clock tick. The following example examine whether a system has any such threads:

```
sched:::tick,
sched:::enqueue
{
    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

The output of the example script is two distributions of the millisecond offset within a ten millisecond interval, one for the `tick` probe and another for `enqueue`:
The output histogram named `tick` shows that the clock tick is firing at a 1 millisecond offset because the system clock frequency is 1000Hz). In this example, the output for `enqueue` is evenly spread across the ten millisecond interval and no spike is visible at 1 millisecond, so it appears that the threads are being not being scheduled on a time basis.

11.6.5 Stability

The `sched` provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Arguments</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, see Chapter 16, *Stability*.

11.7 io Provider

The `io` provider makes available probes that relate to data input and output. The `io` provider enables quick exploration of behavior observed through I/O monitoring tools such as `iostat`. For example, you can use the `io` provider to understand I/O by device, I/O type, I/O size, process, or application name.

11.7.1 Probes

Table 11.8, “io Probes” lists the `io` probes.

**Table 11.8 `io` Probes**

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code></td>
<td>Fires when an I/O request is about to be made either to a peripheral device or to an NFS server. The <code>bufinfo_t</code> corresponding to the I/O request is pointed to by <code>args[0]</code>. The <code>devinfo_t</code> of the device to which the I/O is being issued</td>
</tr>
</tbody>
</table>
Arguments

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>The bufinfo_t corresponding to the I/O request is pointed to by args[0].</td>
</tr>
<tr>
<td></td>
<td>The devinfo_t of the device to which the I/O was issued is pointed to by</td>
</tr>
<tr>
<td></td>
<td>args[1]. The fileinfo_t of the file that corresponds to the I/O request is</td>
</tr>
<tr>
<td></td>
<td>pointed to by args[2]. Note that file information availability depends on</td>
</tr>
<tr>
<td></td>
<td>the file system making the I/O request. See Section 11.7.5, &quot;fileinfo_t&quot;</td>
</tr>
<tr>
<td></td>
<td>for more information.</td>
</tr>
<tr>
<td>done</td>
<td>Fires after an I/O request has been fulfilled. The bufinfo_t corresponding</td>
</tr>
<tr>
<td></td>
<td>to the I/O request is pointed to by args[0]. The done probe fires after the</td>
</tr>
<tr>
<td></td>
<td>I/O completes, but before completion processing has been performed on the</td>
</tr>
<tr>
<td></td>
<td>buffer. As a result B_DONE is not set in b_flags at the time the done probe</td>
</tr>
<tr>
<td></td>
<td>fires. The devinfo_t of the device to which the I/O was issued is pointed to</td>
</tr>
<tr>
<td></td>
<td>by args[1]. The fileinfo_t of the file that corresponds to the I/O request</td>
</tr>
<tr>
<td></td>
<td>is pointed to by args[2].</td>
</tr>
<tr>
<td>wait-start</td>
<td>Fires immediately before a thread begins to wait pending completion of a</td>
</tr>
<tr>
<td></td>
<td>given I/O request. The buf structure corresponding to the I/O request for</td>
</tr>
<tr>
<td></td>
<td>which the thread waits is pointed to by args[0]. The devinfo_t of the device</td>
</tr>
<tr>
<td></td>
<td>to which the I/O was issued is pointed to by args[1]. The fileinfo_t of the</td>
</tr>
<tr>
<td></td>
<td>file that corresponds to the I/O request is pointed to by args[2]. Some</td>
</tr>
<tr>
<td></td>
<td>time after the wait-start probe fires, the wait-done probe fires in the same</td>
</tr>
<tr>
<td></td>
<td>thread.</td>
</tr>
<tr>
<td>wait-done</td>
<td>Fires when a thread finishes waiting for the completion of a given I/O</td>
</tr>
<tr>
<td></td>
<td>request. The bufinfo_t corresponding to the I/O request for which the thread</td>
</tr>
<tr>
<td></td>
<td>will wait is pointed to by args[0]. The devinfo_t of the device to which the</td>
</tr>
<tr>
<td></td>
<td>I/O was issued is pointed to by args[1]. The fileinfo_t of the file that</td>
</tr>
<tr>
<td></td>
<td>corresponds to the I/O request is pointed to by args[2]. The wait-done probe</td>
</tr>
<tr>
<td></td>
<td>fires only after the wait-start probe has fired in the same thread.</td>
</tr>
</tbody>
</table>

The io probes fire for all I/O requests to peripheral devices, and for all file read and file write requests to an NFS server. Requests for metadata from an NFS server, for example, do not trigger io probes due to a readdir() request.

11.7.2 Arguments

Table 11.9, “io Probe Arguments” lists the argument types for io probes. The arguments are described in Table 11.8, “io Probes”.

Table 11.9 io Probe Arguments

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>struct buf *</td>
<td>devinfo_t *</td>
<td>fileinfo_t *</td>
</tr>
<tr>
<td>done</td>
<td>struct buf *</td>
<td>devinfo_t *</td>
<td>fileinfo_t *</td>
</tr>
<tr>
<td>wait-start</td>
<td>struct buf *</td>
<td>devinfo_t *</td>
<td>fileinfo_t *</td>
</tr>
<tr>
<td>wait-done</td>
<td>struct buf *</td>
<td>devinfo_t *</td>
<td>fileinfo_t *</td>
</tr>
</tbody>
</table>

Each io probe has arguments consisting of a pointer to a buf structure, a pointer to a devinfo_t structure, and a pointer to a fileinfo_t structure. These structures are described in the following sections.

Note

DTrace does not currently support the use of fileinfo_t with io probes. In Oracle Linux, no information is readily accessible at the level where the io probes fire about the file where an I/O request originated.
11.7.3 bufinfo_t

The bufinfo_t structure is the abstraction that describes an I/O request. The buffer corresponding to an I/O request is pointed to by args[0] in the start, done, wait-start, and wait-done probes. The definition of bufinfo_t is as follows:

```c
typedef struct bufinfo {
    int b_flags;         /* flags */
    size_t b_bcount;     /* number of bytes */
    caddr_t b_addr;      /* buffer address */
    uint64_t b_blkno;    /* expanded block # on device */
    uint64_t b_lblkno;   /* block # on device */
    size_t b_resid;      /* not supported */
    size_t b_bufsize;    /* size of allocated buffer */
    caddr_t b_iodone;    /* I/O completion routine */
    int b_error;         /* not supported */
    dev_t b_edev;        /* extended device */
} bufinfo_t;
```

**Note**

DTrace translates the members of bufinfo_t from the buffer_head for the Oracle Linux I/O request structure.

*b_flags* indicates the state of the I/O buffer, and consists of a bitwise-or of different state values. Table 11.10, “b_flags Values” lists the values of the supported states.

**Table 11.10 b_flags Values**

<table>
<thead>
<tr>
<th>b_flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_READ</td>
<td>0x000040</td>
<td>Indicates that data is to be read from the peripheral device into main memory.</td>
</tr>
<tr>
<td>B_WRITE</td>
<td>0x000100</td>
<td>Indicates that the data is to be transferred from main memory to the peripheral device.</td>
</tr>
</tbody>
</table>

*b_bcount* is the number of bytes to be transferred as part of the I/O request.

*b_addr* is the virtual address of the I/O request, unless B_PAGEIO is set. The address is a kernel virtual address unless B_PHYS is set, in which case it is a user virtual address. If B_PAGEIO is set, the *b_addr* field contains kernel private data. Only one of B_PHYS and B_PAGEIO can be set, or neither flag is set.

*b_lblkno* identifies which logical block on the device is to be accessed. The mapping from a logical block to a physical block (such as the cylinder, track, and so on) is defined by the device.

*b(bufsize)* contains the size of the allocated buffer.

*b_iodev* identifies a specific routine in the kernel that is called when the I/O is complete.

*b_edev* contains the major and minor device numbers of the device accessed. You can use the D subroutines getmajor and getminor to extract the major and minor device numbers from the *b_edev* field.

11.7.4 devinfo_t

The devinfo_t structure provides information about a device. The devinfo_t structure corresponding to the destination device of an I/O is pointed to by args[1] in the start, done, wait-start, and wait-done probes. The definition of devinfo_t is as follows:
typedef struct devinfo {
  int dev_major;          /* major number */
  int dev_minor;          /* minor number */
  int dev_instance;       /* not supported */
  string dev_name;        /* name of device */
  string dev_statname;    /* name of device + instance/minor */
  string dev_pathname;    /* pathname of device */
} devinfo_t;

Note

DTrace translates the members of devinfo_t from the buffer_head for the Oracle Linux I/O request structure.

dev_major is the major number of the device.

dev_minor is the minor number of the device.

dev_name is the name of the device driver that manages the device.

dev_statname is the name of the device as reported by iostat. This field is provided so that aberrant iostat output can be quickly correlated to actual I/O activity.

dev_pathname is the full path of the device. The path specified by dev_pathname includes components expressing the device node, the instance number, and the minor node. However, all three of these elements are not necessarily expressed in the statistics name. For some devices, the statistics name consists of the device name and the instance number. For other devices, the name consists of the device name and the number of the minor node. As a result, two devices that have the same dev_statname may differ in their dev_pathname.

11.7.5 fileinfo_t

Note

DTrace does not currently support the use of fileinfo_t with the args[2] argument of io probes. You can use the fileinfo_t structure to obtain information about a process's open files via the fds[] array. See Section 2.9.5, “Built-in Variables”.

The fileinfo_t structure provides information about a file. args[2] in the start, done, wait-start, and wait-done probes points to the file to which an I/O request corresponds. The presence of file information is contingent upon the file system providing this information when dispatching I/O requests. Some file systems, especially third-party file systems, might not provide this information. Also, I/O requests might emanate from a file system for which no file information exists. For example, any I/O from or to file system metadata is not associated with any one file. Finally, some highly optimized file systems might aggregate I/O from disjoint files into a single I/O request. In this case, the file system might provide the file information either for the file that represents the majority of the I/O or for the file that represents some of the I/O. Alternatively, the file system might provide no file information at all in this case.

The definition of fileinfo_t is as follows:

typedef struct fileinfo {
  string fi_name;        /* name (basename of fi_pathname) */
  string fi_dirname;     /* directory (dirname of fi_pathname) */
  string fi_pathname;    /* full pathname */
  offset_t fi_offset;    /* offset within file */
  string fi_fs;          /* file system */
  string fi_mount;       /* not supported */
  int fi_oflags;         /* open() flags for file descriptor */
} fileinfo_t;
Examples

The `fi_name` field contains the name of the file but does not include any directory components. If no file information is associated with an I/O, the `fi_name` field is set to the string `<none>`. In some rare cases, the pathname associated with a file might be unknown. In this case, the `fi_name` field is set to the string `<unknown>`.

The `fi_dirname` field contains only the directory component of the file name. As with `fi_name`, this string may be set to `<none>` if no file information is present, or `<unknown>` if the pathname associated with the file is not known.

The `fi_pathname` field contains the full pathname to the file. As with `fi_name`, this string may be set to `<none>` if no file information is present, or `<unknown>` if the pathname associated with the file is not known.

The `fi_offset` field contains the offset within the file, or -1 if either file information is not present or if the offset is otherwise unspecified by the file system.

The `fi_fs` field contains the name of the file system type, or `<none>` if no information is present.

The `fi_oflags` field contains the flags that were specified when opening the file.

11.7.6 Examples

The following example script displays information for every I/O as it is issued:

```c
#pragma D option quiet
BEGIN {
    printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}
io:::start {
    printf("%10s %58s %2s\n", args[1]->dev_statname,
           args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W");
}
```

The output from this script resembles the following example:

```bash
# dtrace -s ./iosnoop.d
device FILE RW
  DM-00 /usr/bin/evince R
  DM-00 /usr/bin/evince R
  DM-00 <unknown> R
  DM-00 <unknown> R
  DM-00 <unknown> R
  DM-00 <none> R
...
```

The `<none>` entries in the output indicate that the I/O request does not correspond to the data in any particular file. Such I/O requests are due to metadata of one form or another. The `<unknown>` entries in the output indicate that the pathname for the file is not known.

You can make the example script slightly more sophisticated by using an associative array to track the time in milliseconds spent on each I/O, as shown in the following example:

```c
#pragma D option quiet
BEGIN {
```
The modified script adds a MS (milliseconds) column to the output.

You can aggregate on device, application, process ID and bytes transferred, as shown in the following example:

```
#pragma D option quiet
io:::start
{
  @[args[1]->dev_statname, execname, pid] = sum(args[0]->b_bcount);
}
END
{
  printf("%10s %20s %10d %15d
", args[0]->dev_statname, execname, pid, @);
}
```

Running this script for a few seconds results in output similar to the following example:

```
$ dtrace -s whoio.d
^C
DEVICE                  APP        PID           BYTES
---                             ---        ---        -------
dm-00               evince      14759           16384
dm-00          flush-252:0       1367           45056
dm-00                 bash      14758          131072
dm-00       gvfsd-metadata       2787          135168
dm-00               evince      14758          139264
dm-00               evince      14338          151552
dm-00          jbd2/dm-0-8        390          356352
```

If you are copying data from one device to another, you might want to know if one of the devices acts as a limiter on the copy. To answer this question, you need to know the effective throughput of each device rather than the number of bytes per second that each device is transferring. You can determine throughput with the following example script:

```
#pragma D option quiet
io:::start
{
  start[args[0]->b_eudev, args[0]->b_blkno] = timestamp;
}
io:::done
/start[args[0]->b_eudev, args[0]->b_blkno]/
{
  this->elapsed = timestamp - start[args[0]->b_eudev, args[0]->b_blkno];
  printf("%10s %s %s %s %s %s
", "DEVICE", "FILE", "RW", "MS");
  printf("%10s %58s %2s %7s
", args[1]->dev_statname, args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W",
          this->elapsed / 10000000, (this->elapsed / 1000) % 1000);
  start[args[0]->b_eudev, args[0]->b_blkno] = 0;
}
```

Running this script for a few seconds results in output similar to the following example:

```
```

If you are copying data from one device to another, you might want to know if one of the devices acts as a limiter on the copy. To answer this question, you need to know the effective throughput of each device rather than the number of bytes per second that each device is transferring. You can determine throughput with the following example script:
We want to get an idea of our throughput to this device in KB/sec. What we have, however, is nanoseconds and bytes. That is we want to calculate:

\[
\frac{\text{bytes}}{1024} \div \frac{\text{nanoseconds}}{1000000000}
\]

But we cannot calculate this using integer arithmetic without losing precision (the denominator, for one, is between 0 and 1 for nearly all I/Os). So we restate the fraction, and cancel:

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

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;
```

Running the example script for several seconds while copying data from a hard disk to a USB drive yields the following output:

```
sdc1 (/dev/sdc1)

value  Distribution  count
32 |  0
64 |  3
128 |  1
256 | 2257
512 |  1
1024 |  0

dm-00 (/dev/dm-00)

value  Distribution  count
128 |  0
256 |  1
512 |  0
1024 |  2
2048 |  0
4096 |  2
8192 | 172
16384 |  52
32768 | 108
65536 |  34
131072 |  0
```

The output shows that the USB drive (sdc1) is clearly the limiting device. The throughput of sdc1 is between 256K/sec and 512K/sec, while dm-00 delivered I/O at anywhere from 8 MB/second to over 64 MB/second.

### 11.7.7 Stability

The io provider uses DTrace's stability mechanism to describe its stabilities, as shown in the following table.
11.8 fasttrap Provider

The `fasttrap` provider performs dynamic instrumentation of arbitrary instructions in user-space threads.

For more information about enabling statically defined probes in user-space programs, see Chapter 13, *Statically Defined Tracing of User Applications*.

11.8.1 Probes

The `fasttrap` provider makes available a single probe that fires each time that a DTrace-enabled user process executes an instruction.

11.8.2 Stability

The `fasttrap` provider uses DTrace’s stability mechanism to describe its stabilities, as shown in the following table.

<table>
<thead>
<tr>
<th>Element</th>
<th>Name Stability</th>
<th>Data Stability</th>
<th>Dependency Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Module</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Function</td>
<td>Private</td>
<td>Private</td>
<td>Unknown</td>
</tr>
<tr>
<td>Name</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
<tr>
<td>Arguments</td>
<td>Evolving</td>
<td>Evolving</td>
<td>ISA</td>
</tr>
</tbody>
</table>

For more information about the stability mechanism, see Chapter 16, *Stability*
Chapter 12 User Process Tracing

Table of Contents

12.1 copyin and copyinstr ................................................................. 155
  12.1.1 Avoiding Errors ................................................................. 156
12.2 Eliminating dtrace Interference ............................................... 156
12.3 syscall Provider ..................................................................... 157
12.4 ustack Action ........................................................................ 158
12.5 uregs[] Array ........................................................................ 159

DTrace is a powerful tool for understanding the behavior of user processes. DTrace can be invaluable when debugging, analyzing performance problems, or simply understanding the behavior of a complex application. This chapter focuses on the DTrace facilities relevant for tracing user process activity and provides examples to illustrate their use.

12.1 copyin and copyinstr

DTrace's interaction with processes is a little different than most traditional debuggers or observability tools. Many such tools appear to execute within the scope of the process, letting users dereference pointers to program variables directly. Rather than appearing to execute within or as part of the process itself, DTrace probes execute in the Oracle Linux kernel. To access process data, a probe needs to use the copyin or copyinstr subroutines to copy user process data into the address space of the kernel.

For example, consider the following write() system call:

```d
ssize_t write(int fd, const void *buf, size_t nbytes);
```

The following D program illustrates an incorrect attempt to print the contents of a string passed to the write() system call:

```d
syscall::write:entry
{
  printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

If you try to run this script, DTrace produces error messages similar to the following example:

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \invalid address (0x10038a000) in action #1
```

The arg1 variable, containing the value of the buf parameter, is an address that refers to memory in the process executing the system call. To read the string at that address, use the copyinstr subroutine and record its result with the printf action:

```d
syscall::write:entry
{
  printf("%s", copyinstr(arg1)); /* correct use of arg1 */
}
```

The output of this script shows all of the strings being passed to the write() system call. Occasionally, however, you might see irregular output similar to the following example:

```
0         37                     write:entry mada&^%**&
```

The copyinstr subroutine acts on an input argument that is the user address of a null-terminated ASCII string. However, buffers passed to the write() system call might refer to binary data rather than ASCII
Avoiding Errors

Strings or to ASCII strings which do not include a terminating null byte. To print only as much of the string as the caller intended, use the two parameter version of the `copyinstr` subroutine which includes the size of the targeted string buffer:

```c
syscall::write:entry { printf("%s", copyinstr(arg1, arg2)); }
```

Alternatively, use the `copyin` subroutine which takes an address and size:

```c
syscall::write:entry {
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Notice that the `stringof` operator is necessary so that DTrace properly converts the user data retrieved using `copyin` to a string. The use of `stringof` is not necessary with `copyinstr` because it always returns type `string`.

### 12.1.1 Avoiding Errors

The `copyin` and `copyinstr` subroutines cannot read from user addresses which have not yet been touched so even a valid address may cause an error if the page containing that address has not yet been faulted in by being accessed. Consider the following example:

```bash
# dtrace -n syscall::open:entry '{ trace(copyinstr(arg0)); }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU ID FUNCTION:NAME
1 8 open:entry /dev/sr0
1 8 open:entry /var/run/utmp
1 8 open:entry /dev/sr0

invalid address (0x9af1b) in action #1 at DIF offset 52
```

In the example output, the application was functioning properly, and the address in `arg0` was valid, but it referred to a page that had not yet been accessed by the corresponding process. To resolve this issue, wait for kernel or application to use the data before tracing it. For example, you might wait until the system call returns to apply `copyinstr`, as shown in the following example:

```bash
# dtrace -n syscall::open:entry '{ self->file = arg0; }' 
   -n syscall::open:return '{ trace(copyinstr(self->file)); self->file = 0; }'
dtrace: description 'syscall::open:entry' matched 1 probe
dtrace: description 'syscall::open:return' matched 1 probe
CPU ID FUNCTION:NAME
0 9 open:return /dev/sr0
1 9 open:return /usr/lib64/gconv/gconv-modules.cache
0 9 open:return /dev/sr0
0 9 open:return public/pickup
1 9 open:return maildrop
1 9 open:return /dev/sr0
1 9 open:return /dev/sr0
1 9 open:return /var/run/utmp
...
```

### 12.2 Eliminating dtrace Interference

If you trace every call to the `write()` system call, you cause a cascade of output. Each call causes the `dtrace` command to call `write()` as it displays the output, and so on. This feedback loop is a good example of how `dtrace` can interfere with the desired data. You can use a simple predicate to prevent such unwanted data from being traced:

```c
syscall::write:entry
/pid !- $pid/
```
The $pid macro variable expands to the process identifier of the process that enabled the probes. The pid variable contains the process identifier of the process whose thread was running on the CPU where the probe was fired. Therefore the predicate /pid != $pid/ ensures that the script does not trace any events related to itself.

12.3 syscall Provider

The syscall provider enables you to trace every system call entry and return. System calls can be a good starting point for understanding the behavior of a process, especially if the process seems to be spending a large amount of time executing or blocked in the kernel as shown by commands such as ps and top.

For example, consider a process with a process ID of 31337 that is consuming a large amount of system time. One possible explanation for this behavior is that the process is executing a large number of system calls. You can specify a simple D program on the command-line to see which system calls are happening most often:

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }
```

```
dtrace: description 'syscall:::entry' matched 215 probes
^C
```

<table>
<thead>
<tr>
<th>syscall</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill</td>
<td>1</td>
</tr>
<tr>
<td>clone</td>
<td>4</td>
</tr>
<tr>
<td>pipe</td>
<td>4</td>
</tr>
<tr>
<td>setpgid</td>
<td>4</td>
</tr>
<tr>
<td>rt_sigreturn</td>
<td>6</td>
</tr>
<tr>
<td>sendmsg</td>
<td>7</td>
</tr>
<tr>
<td>socket</td>
<td>7</td>
</tr>
<tr>
<td>access</td>
<td>8</td>
</tr>
<tr>
<td>getegid</td>
<td>8</td>
</tr>
<tr>
<td>geteuid</td>
<td>8</td>
</tr>
<tr>
<td>getgid</td>
<td>8</td>
</tr>
<tr>
<td>getuid</td>
<td>8</td>
</tr>
<tr>
<td>wait4</td>
<td>12</td>
</tr>
<tr>
<td>close</td>
<td>15</td>
</tr>
<tr>
<td>read</td>
<td>23</td>
</tr>
<tr>
<td>newstat</td>
<td>25</td>
</tr>
<tr>
<td>write</td>
<td>42</td>
</tr>
<tr>
<td>ioctl</td>
<td>65</td>
</tr>
<tr>
<td>rt_sigaction</td>
<td>168</td>
</tr>
<tr>
<td>rt_sigprocmask</td>
<td>198</td>
</tr>
<tr>
<td>write</td>
<td>1092</td>
</tr>
</tbody>
</table>

This report shows which system calls are being called most often, in this case, the write() system call. You can use the syscall provider to further examine the source of all the write() system calls:

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes = quantize(arg2); }
```

```
dtrace: description 'syscall::write:entry' matched 1 probe
^C
```

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1037</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
12.4 ustack Action

Note

If you want to perform symbol lookup in a stripped executable, you must specify the \texttt{--export-dynamic} option when linking the program. This option causes the linker to add all symbols to the dynamic symbol table (that is, the set of symbols which are visible from dynamic objects at run time). If you use \texttt{gcc} to link the objects, specify the option as \texttt{-Wl,--export-dynamic} to pass the correct option to the linker.

If you want to look up symbols in shared libraries or unstripped executables, the \texttt{--export-dynamic} option is not required.

Tracing a process thread's stack at the time a particular probe is activated is often useful for examining a problem in more detail. The \texttt{ustack} action traces the user thread's stack. If, for example, a process that opens many files occasionally fails in the \texttt{open()} system call, you can use the \texttt{ustack} action to discover the code path that executes the failed \texttt{open}:

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}
syscall::open:return
/self->path != NULL && errno != 0/
{
    printf("open for 'ts' failed", self->path);
    ustack();
}
```

This script also illustrates the use of the \texttt{$1} macro variable which takes the value of the first operand specified on the \texttt{dtrace} command-line:

```
# dtrace -s ./badopen.d 3430
dtrace: script './badopen.d' matched 2 probes
CPU   ID      FUNCTION:NAME
1   489      openat:return open for '/usr/lib/foo' failed
           libc.so.6`sleep+0xe0
           ld-linux-x86-64.so.2`do_lookup_x+0x847
           libc.so.6`0x3cb8003630
           libc.so.6`0x3cb8003c48
           libc.so.6`0x3cb800e2c8
           libc.so.6`0x3cb8003c48
           looper`0x400612
           libc.so.6`getenv+0x2a
           looper`0x4003c8
           looper`0x4009e0
           looper`0x3cb800e2c8
           looper`0x4009b0
           looper`0x3cb800e2c8
           looper`0x4009b0
           looper`doOpenLoop+0x33
           looper`0x400e9c
```
The ustack action records program counter (PC) values for the stack and dtrace resolves the PC values to symbol names by looking through the process's symbol tables. If dtrace cannot resolve the PC value to a symbol, it prints out the value as a hexadecimal integer.

If a process exits or is killed before the ustack data is formatted for output, dtrace might be unable to convert the PC values in the stack trace to symbol names, and it displays them as hexadecimal integers.

### 12.5 uregs[] Array

The uregs[] array enables you to access individual user registers. Table 12.1, “x86 uregs[] Constants” lists the index constants into the uregs[] array for each supported architecture.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Register</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_PC</td>
<td>program counter register</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_SP</td>
<td>stack pointer register</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_R0</td>
<td>first return code</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_R1</td>
<td>second return code</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_CS</td>
<td>%cs</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_GS</td>
<td>%gs</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_ES</td>
<td>%es</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_DS</td>
<td>%ds</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EDI</td>
<td>%edi</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_ESI</td>
<td>%esi</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EBP</td>
<td>%ebp</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EAX</td>
<td>%eax</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_ESP</td>
<td>%esp</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EBX</td>
<td>%ebx</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_ECX</td>
<td>%ecx</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EDX</td>
<td>%edx</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_TRAPNO</td>
<td>%trapno</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_ERR</td>
<td>%err</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EIP</td>
<td>%eip</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_CS</td>
<td>%cs</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_EFL</td>
<td>%efl</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_UESP</td>
<td>%uesp</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>R_SS</td>
<td>%ss</td>
<td>x86, AMD64</td>
</tr>
<tr>
<td>Constant</td>
<td>Register</td>
<td>Architecture</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>R_RSP</td>
<td>%rsp</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RFL</td>
<td>%rf1</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RIP</td>
<td>%rip</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RAX</td>
<td>%rax</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RCX</td>
<td>%rcx</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RDX</td>
<td>%rdx</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RBP</td>
<td>%rbp</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RSI</td>
<td>%rsi</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_RDI</td>
<td>%rdi</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R8</td>
<td>%r8</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R9</td>
<td>%r9</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R10</td>
<td>%r10</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R11</td>
<td>%r11</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R12</td>
<td>%r12</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R13</td>
<td>%r13</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R14</td>
<td>%r14</td>
<td>AMD64</td>
</tr>
<tr>
<td>R_R15</td>
<td>%r15</td>
<td>AMD64</td>
</tr>
</tbody>
</table>
Chapter 13 Statically Defined Tracing of User Applications

Table of Contents

13.1 Choosing the Probe Points ................................................................. 161
13.2 Adding Probes to an Application ....................................................... 161
  13.2.1 Defining Providers and Probes .................................................... 162
  13.2.2 Adding Probes to Application Code .............................................. 162
  13.2.3 Testing if a Probe Is Enabled ...................................................... 163
  13.2.4 Building Applications with Probes .............................................. 163
  13.2.5 Using Statically Defined Probes .................................................. 164

DTrace provides a facility for user application developers to define customized probes in application code. These static probes impose little to no overhead when disabled and are dynamically enabled like all other DTrace probes. You can use static probes to describe application semantics to users of DTrace without exposing or requiring implementation knowledge of your applications. This chapter describes how to define static probes in user applications and how to use DTrace to enable such probes in user processes.

Note

DTrace supports statically defined tracing of user applications for both 32-bit and 64-bit binaries.

For information about using static probes with kernel modules, see Chapter 14, Statically Defined Tracing of Kernel Modules.

13.1 Choosing the Probe Points

DTrace allows developers to embed static probe points in application code, including both complete applications and shared libraries. These probes can be enabled wherever the application or library is running, either in development or in production. You should define probes that have a semantic meaning that is readily understood by your DTrace user community. For example, you could define query-receive and query-respond probes for a web server that correspond to a client submitting a request and the web server responding to that request. These example probes are easily understood by most DTrace users and correspond to the highest level abstractions for the application, rather than lower level implementation details. DTrace users can use these probes to understand the time distribution of requests. If your query-receive probe presented the URL request strings as an argument, a DTrace user could determine which requests were generating the most disk I/O by combining this probe with the io provider.

You should also consider the stability of the abstractions you describe when choosing probe names and locations. Will this probe persist in future releases of the application, even if the implementation changes? Does the probe make sense on all system architectures or is it specific to a particular instruction set? This chapter will discuss the details of how these decisions guide your static tracing definitions.

13.2 Adding Probes to an Application

DTrace probes for libraries and executables are defined in an ELF section in the corresponding application binary. This section describes how to define your probes, add them to your application source code, and augment your application's build process to include the DTrace probe definitions.
13.2.1 Defining Providers and Probes

You define DTrace probes in a .d source file, which is then used when compiling and linking your application. First, select an appropriate name for your user application provider. In a .d source file, add a provider definition similar to the following example:

```d
provider myserv {
    ...
};
```

Next, add a definition for each probe and the corresponding arguments. The following example defines the two probes discussed in Section 13.1, “Choosing the Probe Points”. The first probe has two arguments, both of type char *, and the second probe has no arguments. The D compiler converts two consecutive underscores (_) to a dash (-) in the probe name.

```d
provider myserv {
    probe query__receive(char *, char *);
    probe query__respond();
};
```

You can add stability attributes to your provider definition so that consumers of your probes understand the likelihood of change in future versions of your application. See Chapter 16, Stability for more information on the DTrace stability attributes. Stability attributes are defined as shown in the following example:

```d
#pragma D attributes Evolving/Evolving/Common provider myserv provider
#pragma D attributes Private/Private/Unknown provider myserv module
#pragma D attributes Private/Private/Unknown provider myserv function
#pragma D attributes Evolving/Evolving/Common provider myserv name
#pragma D attributes Evolving/Evolving/Common provider myserv args

provider myserv {
    probe query__receive(char *, char *);
    probe query__respond();
};
```

13.2.2 Adding Probes to Application Code

When you have defined your probes in a .d file, you need to augment your source code to indicate the locations that should trigger your probes. Consider the following example C application source code:

```c
void main_look(void)
{
    ...
    query = wait_for_new_query();
    process_query(query);
    ...
}
```

To add probes to an application, use the -h option to dtrace to generate a header file based on the probe definitions. For example, the following command generates the header file `myserv.h`, which contains macro definitions that correspond to the probe definitions in `myserv.d`:

```bash
# dtrace -h -s myserv.d
```

This method is recommended as the coding is easier to implement and understand. The method is also compatible with both C and C++. In addition, as the generated macros depend on the types that you define in the provider definition, the compiler can perform type checking on them.
For example, you can add a probe site by using the `MYSERV_QUERY_RECEIVE` macro that `dtrace -h` defines in `myserv.h`:

```c
#include "myserv.h"
...
void main_look(void)
{
    ...
    query = wait_for_new_query();
    MYSERV_QUERY_RECEIVE(query->clientname, query->msg);
    process_query(query);
    ...
}
```

In this example, the name of the macro encodes both the provider name and the probe name.

### 13.2.3 Testing if a Probe Is Enabled

The computational overhead of a DTrace probe is usually equivalent to a few no-op instructions. However, setting up probe arguments can be expensive, particularly in the case of dynamic languages where the code has to determine the name of a class or method at runtime.

In addition to the probe macro, the `dtrace -h` command creates an *is-enabled probe* macro for each probe that you specify in the provider definition. To ensure that your program computes the arguments to a DTrace probe only when required, you can use the is-enabled probe test to verify whether the probe is currently enabled, for example:

```c
if (MYSERV_QUERY_RECEIVE_ENABLED())
    MYSERV_QUERY_RECEIVE(query->clientname, query->msg);
```

If the probe arguments are computationally expensive to calculate, the slight overhead incurred by performing the is-enabled probe test is more than offset when the probe is not enabled.

### 13.2.4 Building Applications with Probes

You must augment the build process for your application to include the DTrace provider and probe definitions. A typical build process takes each source file and compiles it to create a corresponding object file. The compiled object files are then linked together to create the finished application binary, as shown in the following example:

```bash
src1.o: src1.c
    gcc -c src1.c
src2.o: src2.c
    gcc -c src2.c
myserv: src1.o src2.o
    gcc -o myserv src1.o src2.o
```

If you have included DTrace probe definitions in your application, you need to add appropriate `Makefile` rules to your build process to execute the `dtrace` command.

The `dtrace` command post-processes the object files created by the preceding compiler commands and generates the object file `myserv.o` from `myserv.d` and the other object files. The `-G` option is used to link provider and probe definitions with a user application.

The `-Wl,--export-dynamic` link options to `gcc` are required to support symbol lookup in a stripped executable at runtime, for example, by running `ustack()`.

If you inserted probes in the source code by using the macros defined in a header file that was created by `dtrace -h`, you need to include that command in the Makefile, for example:
Using Statically Defined Probes

The rules in the `Makefile` take account of the dependency of the header file on the probe definition.

13.2.5 Using Statically Defined Probes

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

To allow access to the probes in a DTrace-enabled, user-space program, enable the `fasttrap` provider by loading the `fasttrap` module:

```
# modprobe fasttrap
```

If the program to be traced is run by a user other than `root`, 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
# ls -l /dev/dtrace
```

The full name of a probe in a user application takes the usual `providerPID:module:function:name` form, where:

- **provider** The name of the provider as defined in the provider definition file.
- **PID** The process ID of the running executable.

Note: You must change the mode on the device before the user runs the program.
Using Statically Defined Probes

module The name of the executable.

function The name of the function where the probe is located.

name The name of the probe as defined in the provider definition file with any two consecutive underscores (__) replaced by a dash (-).

For example, for a myserv process with a PID of 1173, the full name of the query-receive probe would be myserv1173:myserv:main_look:query-receive.

The following simple example shows how to invoke a traced process from dtrace itself.

```bash
# dtrace -c ./myserv -qs /dev/stdin <<EOF
$target:::query-receive
    { printf("%s:%s:%s %s %s\n", probeprov, probemod, probefunc, probename, 
        stringof(args[0]), stringof(args[1])); }
$target:::query-respond
    { printf("%s:%s:%s\n", probeprov, probemod, probefunc, probename); }
EOF
```

myserv1173:myserv:main_look:query-receive foo1 msg1
myserv1173:myserv:process_query:query-respond
myserv1173:myserv:main_look:query-receive bar2 msg1
myserv1173:myserv:process_query:query-respond
...

Note

For the query-receive probe, we use stringof() to cast args[0] and args[1] to type string. Otherwise, we would see a DTrace compilation error similar to the following.

dtrace: failed to compile script /dev/stdin: line 7:
printf( ) argument #5 is incompatible with conversion #4 prototype:
conversion: %s
prototype: char [] or string (or use stringof)
argument: char *

If we had defined the probe arguments as type string instead of char * in the probe definition file, we would see a compilation warning, for example:

In file included from src1.c:5:
myserv.h:39: warning: parameter names (without types) in function declaration

In this case, casting the probe arguments to type string would no longer be required.

The following script illustrates the complete process of instrumenting, compiling and tracing a simple user-space program:

```bash
#!/bin/bash
# Define the probes
cat > prov.d <<EOF
provider myprog
{ probe dbquery__entry(char *);
 probe dbquery__result(int);
}
EOF
```
Using Statically Defined Probes

```c
#include <stdio.h>
#include "prov.h"

int main(void)
{
    char *query = "select value from table where name = 'foo'");
    /* If the dbquery-entry probe is enabled, trigger it */
    if (MYPROG_DBQUERY_ENTRY_ENABLED())
        MYPROG_DBQUERY_ENTRY(query);
    /* Pretend to run query and obtain result */
    sleep(1);
    int result = 42;
    /* If the dbquery-result probe is enabled, trigger it */
    if (MYPROG_DBQUERY_RESULT_ENABLED())
        MYPROG_DBQUERY_RESULT(result);
    return (0);
}
```

```makefile
# Create the Makefile
cat > Makefile <<EOF
prov.h: prov.d
dtrace -h -s prov.d

test.o: test.c prov.h
gcc -c test.c

prov.o: prov.d test.o
dtrace -G -s prov.d test.o

test: prov.o
gcc -o test prov.o test.o
EOF

# Make the executable
make test

# Trace the program
dtrace -c ./test -qs /dev/stdin <<EOF
$target:::dbquery-entry
{
    self->ts = timestamp;
    printf("Query = %s
", stringof(args[0]));
}

$target:::dbquery-result
{
    printf("Query time = %d microseconds; Result = %d
", (timestamp - self->ts) / 1000, args[0]);
}
EOF
```

The output from running this script shows the compilation steps as well as the results of tracing the program:

```
# chmod +x testscript
# ./testscript
dtrace -h -s prov.d
gcc -c test.c
dtrace -G -s prov.d test.o
gcc -o test prov.o test.o
Query = select value from table where name = 'foo'
```
Query time = 1000481 microseconds; Result = 42
DTrace provides a facility for developers to define customized probes in kernel modules. These static probes appear as additional probes of the \texttt{sdt} provider and impose little to no overhead if the \texttt{sdt} module is not loaded. (For x86\_64, the overhead is a single-byte NOP followed by a 4-byte NOP.) This chapter provides a full example of how to define and use static probes in a kernel module.

The general principles for naming probes and choosing insertion points are the same for kernel modules as for user-space applications. You should define probes that have a semantic meaning that is readily understood by your DTrace user community. Typically, you might name probes for the routine in which you place them and their position in that routine. For example, if your probes provide information about data values on entry to or return from a routine named \texttt{foo}, you might name them \texttt{foo-entry} and \texttt{foo-return}. The data values returned by such probes could present the routine as a \textit{black box}, rather than return intermediate values from the internal implementation of the module. To gather data from deeper inside a module, you might insert additional probes with names such as \texttt{foo-stage1} or \texttt{foo-post-hardware-init}.

In one respect, using static probes with kernel modules can actually be simpler than for user-space applications. You do not need to modify the build files unless you want to conditionally compile a module to include the probes. Inserting the probes in the source code is slightly more complex as you cannot use \texttt{dtrace -h} to generate the probe macros. However, using a \texttt{DTRACE_PROBE} macro to insert a probe is a relatively simple change to make to the source code.

You can insert \texttt{sdt} static probes in any Oracle Linux kernel module for which you have the source files and the necessary build infrastructure. However, DTrace supports statically defined tracing of 64-bit kernel modules only.

For more information about the \texttt{sdt} provider, see Section 11.4, “\texttt{sdt provider}”.

For an introduction to the concepts of statically defined tracing as applied to user-space applications, see Chapter 13, \textit{Statically Defined Tracing of User Applications}.

### 14.1 Inserting Static Probe Points

You can embed static probes within the source code where you want to capture the current state of a module and its data.

The following example pseudo character device driver consists of three source files:

- \texttt{revdev.h} Header file for the module.
- \texttt{rev_mod.c} Defines the module's properties and its \texttt{init} and \texttt{exit} routines.
- \texttt{rev_dev.c} Defines the driver's \texttt{open}, \texttt{read}, \texttt{release}, \texttt{unlocked_ioctl}, and \texttt{write} routines. The static probes are inserted in the \texttt{read}, \texttt{unlocked_ioctl}, and \texttt{write} routines, although probes could also be inserted in the other routines if required.
revdev.h

No existing lines of code are modified in this example. Only line insertions are required for the `<linux/sdt.h>` header file and probe macro definitions.

The changes made in this example are highlighted like this.

```
#include <asm/uaccess.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/mutex.h>
#include <linux/types.h>
#include <linux/sdt.h>

#define DEVICE "revdev"
```

The DTRACE_PROBE macros that are defined in `/lib/modules/`uname -r`/build/include/linux/sdt.h` support from zero to eight arguments.

You can define your own macros for the inserted probes as shown in the example. Unlike user-space static probes, you cannot use `dtrace -h` to create a header file that includes suitable probe definitions. You do not need to create a provider definition file for the probes.

The probes are named according to the first argument of the DTRACE_PROBE macro. The suffix \textit{N} in the macro name \texttt{DTRACE\_PROBE\_N} refers the number of arguments that are passed to the probe. The first argument to the probe macro is the probe name. As described in Section 11.4.1.1, “Declaring Probes”, two consecutive underscores are converted to a single dash. The remaining macro arguments are pairs of arguments that define the DTrace \texttt{argN} variables that are assigned when the probe fires. Each pair of arguments defines the variable type and a variable name, for example:

```
#define REVDEV_WRITE_ENTRY_PROBE(name, fp, buf, n) \ 
  DTRACE_PROBE3(write__##name, file *, fp, char *, buf) \ 
  DTRACE_PROBE2(write__##name, char *, buf, size_t, n) \ 
```

The values of \texttt{fp}, \texttt{buf}, and \texttt{n} are made available by the \texttt{arg0}, \texttt{arg1}, and \texttt{arg2} variables in DTrace when the probe fires.

The provider, module, and function elements of the complete probe are named for \texttt{sdt}, the driver module name (without the \texttt{.ko}), and the driver routine.

The probes inherit the stability attributes of the \texttt{sdt} provider.

rev_mod.c

No changes are made in this example, which does not insert any probes in the module's \texttt{init} and \texttt{exit} routines. However, there is no restriction on inserting probes in these routines.
rev_dev.c

No existing lines of code are modified in this example. Only line insertions are required for the entry and return probes in each of the read, unlocked_ioctl, and write routines.

The changes made in this example are highlighted like this.
static void revstr(char *s) { /* After Kernighan and Ritchie */
    int i, j, t;
    for (i = 0, j = strlen(s)-1; i < j; i++, j--)
        t = s[i], s[i] = s[j], s[j] = t;
}

static ssize_t revdev_read(struct file *fp, char* buf, size_t n, loff_t *o){
    int retval;
    REVDEV_READ_ENTRY_PROBE(entry, fp, devbuf.data);
    revstr(devbuf.data);
    n = strlen(devbuf.data);
    retval = copy_to_user(buf, devbuf.data, n);
    REVDEV_READ_RETURN_PROBE(return, buf, n);
    if (retval != 0) return -EINVAL;
    return 0;
}

static ssize_t revdev_write(struct file *fp, const char* buf, size_t n, loff_t *o){
    int retval;
    REVDEV_WRITE_ENTRY_PROBE(entry, fp, buf, n);
    retval = copy_from_user(devbuf.data, buf, n);
    devbuf.data[n-retval] = '\0';
    REVDEV_WRITE_RETURN_PROBE(return, devbuf.data, n);
    if (retval != 0) return -EINVAL;
    return 0;
}

static int revdev_close(struct inode *inode, struct file *fp){
    mutex_unlock(&revdev_mutex);
    return 0;
}

const struct file_operations revdev_fops = {
    .owner = THIS_MODULE,
    .read = revdev_read,
    .write = revdev_write,
    .unlocked_ioctl = revdev_ioctl,
    .open = revdev_open,
    .release = revdev_close,
};

14.2 Building Modules with Static Probes

Note

This example requires that you link the module against a version of Unbreakable
Enterprise Kernel Release 3 (UEK R3) that supports the DTrace modules.

A bug in the current implementation means that a module containing SDT probes
must be built from two or more source files.

The following Kbuild and Makefile are used to build the example pseudo driver module revdev.ko
and a test program named testrevdev.

Kbuild

bj-m            += revdev.o
revdev-y        := rev_dev.o rev_mod.o
Makefile

KERNEL_DIR = /lib/modules/`uname -r`/build

modules:: testrevdev
install:: modules_install
testrevdev: testrevdev.c
gcc -o testrevdev testrevdev.c
%::
  $(MAKE) -C $(KERNEL_DIR) M=`pwd` $@

The source file for **testrevdev** is **testrevdev.c**.

testrevdev.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DEVICE_FILE "/dev/revdev"

int main() {
    char buf[81];
    int i, fd, n;

    if ((fd = open(DEVICE_FILE, O_RDWR)) != 0) {
        perror("open");
        exit(1);
    }

    i=0;
    while (1) {
        (i++)%20;
        printf("Write: ");
        scanf(" %80[^
]", buf);
        n = strlen(buf);
        if (!strncmp(buf, "exit", 4))
            break;
        else if (!strncmp(buf, "ioctl", 5))
            ioctl(fd,128,i);
        else {
            write(fd, buf, n);
            read(fd, buf, n);
            buf[n]='\0';
            printf(" Read: %s\n", buf);
        }
    }

    close(fd);
    exit(0);
}
```

When run, **testrevdev** reads a string that you enter, writes the string to the **revdev** device, and reads the reversed string from the device.

If the input string begins with **ioctl**, the program calls **ioctl** on the open file descriptor, which invokes the device's **unlocked_ioctl** routine. An input string that begins with **exit** terminates the program.

To build the module and test program, use the **make** command:

```bash
# make
make -C /lib/modules/`uname -r`/build M=`pwd` modules
```
14.3 Using DTrace to Test Modules with Static Probes

You can use DTrace to display information when one of the embedded static probes in a module fires.

To test the example module *revdev.ko*:

1. Set up a udev rule to create the `/dev/revdev` device file:

   ```
   # echo "KERNEL=="revdev", MODE="0660"" > /etc/udev/rules.d/10-revdev.rules
   ```

2. Load the *dtrace*, *sdt*, *systrace*, and *revdev.ko* modules:

   ```
   # modprobe dtrace  # modprobe sdt  # modprobe systrace  # insmod revdev.ko
   ```

   You can use *dtrace* to test that the probes are now available:

   ```
   # dtrace -l -m revdev
<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_ioctl</td>
<td>ioctl-return</td>
</tr>
<tr>
<td>5</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_ioctl</td>
<td>ioctl-entry</td>
</tr>
<tr>
<td>6</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_write</td>
<td>write-return</td>
</tr>
<tr>
<td>7</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_write</td>
<td>write-entry</td>
</tr>
<tr>
<td>8</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_read</td>
<td>read-return</td>
</tr>
<tr>
<td>9</td>
<td>sdt</td>
<td>revdev</td>
<td>revdev_read</td>
<td>read-entry</td>
</tr>
</tbody>
</table>
   ```

3. Enter the following DTrace script *(traceflow)*:

   ```
   #!/usr/sbin/dtrace -qs
   #pragma D option nspec=10
   
   self int indent;
   
   syscall:::entry
   /execname == "testrevdev"/ {
   self->specflag = 0;
   self->spec = speculation();
   self->indent += 2;
   speculate(self->spec);
   }
   
   syscall:::entry
   /self->spec/ {
   speculate(self->spec);
   printf("%s", self->indent, ">");
   printf("%s() entry
", probefunc);
   self->indent += 2;
   ```
Using DTrace to Test Modules with Static Probes

```c
} syscall:::return /self->spec/
{ speculate(self->spec); self->indent -= 2;
printf("%*s ", self->indent, "<- ");
printf("%s() return\n", probefunc);
}

syscall:::return /self->spec && self->specflag == 0/
{discard(self->spec);
self->indent -= 2;
self->spec = 0;
}

syscall:::return /self->spec && self->specflag == 1/
{commit(self->spec);
self->indent -= 2;
self->spec = 0;
}
sdt:revdev::ioctl-entry /self->spec/
{ speculate(self->spec);
self->specflag = 1;
printf("%*s ", self->indent, "=> ");
printf("%s() entry file: %s cmd: %d arg: %d\n", probefunc, d_path(&(((struct file *)arg0)->f_path)), arg1, arg2);
}
sdt:revdev::ioctl-return /self->spec/
{ speculate(self->spec);
printf("%*s ", self->indent, "<= ");
printf("%s() return cpstr: %s\n", probefunc, stringof((char*)arg0));
}
sdt:revdev::read-entry /self->spec/
{ speculate(self->spec);
self->specflag = 1;
printf("%*s ", self->indent, "=> ");
printf("%s() entry file: %s devbuf: %s\n", probefunc, d_path(&(((struct file *)arg0)->f_path)),
stringof((char *)arg1));
}
sdt:revdev::read-return /self->spec/
{ speculate(self->spec);
printf("%*s ", self->indent, "<= ");
printf("%s() return string: %s len: %d\n", probefunc, stringof((char *)arg0), arg1);
}
sdt:revdev::write-entry /self->spec/
```
When one of the inserted probes fires, `traceflow` displays information about data values in the module by using the probe argument variables (arg0, arg1, arg2,...).

**Note**

Argument variables that return pointer types such as `file *` and `char *` must be explicitly cast.

The script uses `d_path` and `stringof` to create printable file paths and strings. For example, `(struct file *)arg0` casts the value of `arg0` to a file pointer `(struct file *)`. The `f_path` member of the `struct file` contains the path structure `(struct path)` for a file. As `d_path` takes a path pointer `(struct path *)` as its argument, the & operator is used to return a pointer to the `struct path`.

See Section 4.6.8, "d_path" and Section 2.11.4, "String Conversion".

4. Use `chmod` to make `traceflow` executable:

```bash
# chmod 750 tracedev
```

5. In one window, run `traceflow`:

```bash
# ./traceflow
```

6. In another window, run `testrevdev` and enter input, for example:

```bash
# ./testrevdev
Write: hello
Read: olleh
Write: world
Read: dlrow
Write: ioctl
Write: ioctl
Write: exit
```

In the window where `traceflow` is running, you should see output similar to the following as DTrace responds to the probes in `revdev.ko` firing:

```bash
# ./traceflow
-> write() entry
  => revdev_write() entry file: /dev/revdev string: hello len: 5
<= revdev_write() return string: hello len: 5
<= write() return
  -> read() entry
  => revdev_read() entry file: /dev/revdev devbuf: hello
```
Using DTrace to Test Modules with Static Probes

< revdev_read() return string: olleh len: 5
< read() return
 -> write() entry
  -> revdev_write() entry file: /dev/revdev string: world len: 5
< revdev_write() return string: world len: 5
< write() return
 -> read() entry
  -> revdev_read() entry file: /dev/revdev devbuf: world
< revdev_read() return string: dlrow len: 5
< read() return
 -> ioctl() entry
  -> revdev_ioctl() entry file: /dev/revdev cmd: 128 arg: 3
< revdev_ioctl() return cpstr: Odd
< ioctl() return
 -> ioctl() entry
  -> revdev_ioctl() entry file: /dev/revdev cmd: 128 arg: 4
< revdev_ioctl() return cpstr: Even
< ioctl() return
Chapter 15 Performance Considerations

Table of Contents

15.1 Limit Enabled Probes .................................................................................................................. 179
15.2 Use Aggregations ....................................................................................................................... 179
15.3 Use Cacheable Predicates ........................................................................................................... 180

Because DTrace causes additional work in the system, enabling DTrace always affects system performance in some way. Often, this effect is negligible, but it can become substantial if many probes are enabled with costly enablings. This chapter describes techniques for minimizing the performance effect of DTrace.

15.1 Limit Enabled Probes

Dynamic instrumentation techniques enable DTrace to provide unparalleled tracing coverage of the kernel and of arbitrary user processes. While this coverage allows revolutionary new insight into system behavior, it also can cause enormous probe effect. If tens of thousands or hundreds of thousands of probes are enabled, the effect on the system can easily be substantial. Therefore, you should only enable as many probes as you need to solve a problem. For example, you should not enable all \texttt{syscall} probes if a more concise enabling will answer your question. Your question might allow you to concentrate on a specific module of interest or a specific function.

DTrace can also be used in situations where large numbers of probes must be enabled for a question to be answered. Enabling a large number of probes might slow down the system quite a bit, but it never induces fatal failure on the machine. You should therefore not hesitate to enable many probes if required.

15.2 Use Aggregations

As discussed in Chapter 3, \textit{Aggregations}, DTrace aggregations allow for a scalable way of aggregating data. Associative arrays might appear to offer similar functionality to aggregations. However, by nature of being global, general-purpose variables, they cannot offer the linear scalability of aggregations. You should therefore prefer to use aggregations over associative arrays when possible. For example, the following D program uses an associative array to aggregate data:

```
syscall:::entry
{  
  totals[execname]++;  
}
syscall::rexit:entry
{  
  printf("%40s \d\n", execname, totals[execname]);  
  totals[execname] = 0;  
}
```

The following D program is preferable as it uses an aggregation to achieve the same result:

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

END
15.3 Use Cacheable Predicates

DTrace predicates are used to filter unwanted data from the experiment by tracing data only if a specified condition is found to be true. When enabling many probes, you generally use predicates of a form that identifies a specific thread or threads of interest, such as /self->traceme/ or /pid == 12345/. Although many of these predicates evaluate to a false value for most threads in most probes, the evaluation itself can become costly when done for many thousands of probes. To reduce this cost, DTrace caches the evaluation of a predicate if it includes only thread-local variables (for example, /self->traceme/) or immutable variables (for example, /pid == 12345/). The cost of evaluating a cached predicate is much smaller than the cost of evaluating a non-cached predicate, especially if the predicate involves thread-local variables, string comparisons, or other relatively costly operations. While predicate caching is transparent to the user, it does imply some guidelines for constructing optimal predicates, as shown in the following table:

<table>
<thead>
<tr>
<th>Cacheable</th>
<th>Uncacheable</th>
</tr>
</thead>
<tbody>
<tr>
<td>self-&gt;mumble</td>
<td>mumblecurthread</td>
</tr>
<tr>
<td></td>
<td>mumblepid</td>
</tr>
<tr>
<td></td>
<td>tid</td>
</tr>
<tr>
<td>execname</td>
<td>curpsinfo-&gt;pr_fname</td>
</tr>
<tr>
<td></td>
<td>((struct task_struct *)curthread)-&gt;comm</td>
</tr>
<tr>
<td>pid</td>
<td>curpsinfo-&gt;pr_pid</td>
</tr>
<tr>
<td></td>
<td>((struct task_struct *)curthread)-&gt;pid</td>
</tr>
<tr>
<td>tid</td>
<td>curlwpsinfo-&gt;pr_lwpid</td>
</tr>
<tr>
<td></td>
<td>((struct task_struct *)curthread)-&gt;pid</td>
</tr>
<tr>
<td>curthread</td>
<td>curthread-&gt;any_member</td>
</tr>
<tr>
<td></td>
<td>curlwpsinfo-&gt;any_member</td>
</tr>
<tr>
<td></td>
<td>curpsinfo-&gt;any_member</td>
</tr>
</tbody>
</table>

The following example uses an associative array in the predicate and is not cacheable:

```c
syscall::read:entry
{
    follow[pid, tid] = 1;
}
syscall::read:return
follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

Using a cacheable thread-local variable is preferable:

```c
syscall::read:entry
{
    self->follow = 1;
}
```
To be cacheable, a predicate must consist exclusively of cacheable expressions. The following predicates are all cacheable:

- `/execname == "myprogram"/`
- `/execname == $1/`
- `/pid == 12345/`
- `/pid == $1/`
- `/self->traceme == 1/`

The following examples, which use global variables, are not cacheable:

- `/execname == one_to_watch/`
- `/traceme[execname]/`
- `/pid == pid_i_care_about/`
- `/self->traceme == my_global/`
Developers are provided with early access to new technologies as well as observability tools that allow users to peer into the internal implementation details of user and kernel software. Unfortunately, new technologies and internal implementation details are both prone to changes as interfaces and implementations evolve and mature when software is upgraded or patched. Application and interface stability levels are documented using a set of labels to help set user expectations for what kinds of changes might occur in different kinds of future releases. No one stability attribute appropriately describes the arbitrary set of entities and services that can be accessed from a D program. DTrace and the D compiler therefore include features to dynamically compute and describe the stability levels of D programs you create. This chapter discusses the DTrace features for determining program stability to help you design stable D programs. You can use the DTrace stability features to inform you of the stability attributes of your D programs, or to produce compile-time errors when your program has undesirable interface dependencies.

### 16.1 Stability Levels

DTrace provides two types of stability attributes for entities such as built-in variables, functions, and probes: a stability level and an architectural dependency class. The DTrace stability level assists you in making risk assessments when developing scripts and tools based on DTrace by indicating how likely an interface or DTrace entity is to change in a future release or patch. The DTrace dependency class tells you whether an interface is common to all Oracle Linux platforms and processors, or whether the interface is associated with a particular architecture. The two types of attributes used to describe interfaces can vary independently.

The stability values used by DTrace appear in the following list in order from lowest to highest stability. Applications that depend only on Stable interfaces should reliably continue to function correctly on future minor releases and will not be broken by interim patches. The less stable interfaces allow experimentation, prototyping, tuning, and debugging on your current system, but should be used with the understanding that they might change incompatibly or even be dropped or replaced with alternatives in future minor releases.

The DTrace stability values also help you understand the stability of the software entities you are observing, in addition to the stability of the DTrace interfaces themselves. Therefore, DTrace stability values also tell you how likely your D programs and layered tools are to require corresponding changes when you upgrade or change the software stack you are observing.

<table>
<thead>
<tr>
<th>Stability Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal</strong></td>
<td>The interface is private to DTrace and represents an implementation detail of DTrace. Internal interfaces might change in minor or micro releases.</td>
</tr>
<tr>
<td><strong>Private</strong></td>
<td>The interface is private to Oracle and represents an interface developed for use by other Oracle products that is not yet publicly documented for use by customers and ISVs. Private interfaces might change in minor or micro releases.</td>
</tr>
<tr>
<td>Stability Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Obsolete</td>
<td>The interface is supported in the current release but is scheduled to be removed, most likely in a future minor release. The D compiler might produce warning messages if you attempt to use an Obsolete interface.</td>
</tr>
<tr>
<td>External</td>
<td>The interface is controlled by an entity other than Oracle. Oracle makes no claims regarding either source or binary compatibility for External interfaces between any two releases. Applications based on these interfaces might not work in future releases, including patches that contain External interfaces.</td>
</tr>
<tr>
<td>Unstable</td>
<td>The interface is provided to give developers early access to new or rapidly changing technology or to an implementation artifact that is essential for observing or debugging system behavior for which a more stable solution is anticipated in the future. Oracle makes no claims about either source or binary compatibility for Unstable interfaces from one minor release to another.</td>
</tr>
<tr>
<td>Evolving</td>
<td>The interface might eventually become Standard or Stable but is still in transition. When non-upward compatible changes become necessary, they will occur in minor and major releases. These changes will be avoided in micro releases whenever possible. If such a change is necessary, it will be documented in the release notes for the affected release, and when feasible, migration aids will be provided for binary compatibility and continued D program development.</td>
</tr>
<tr>
<td>Stable</td>
<td>The interface is a mature interface.</td>
</tr>
<tr>
<td>Standard</td>
<td>The interface complies with an industry standard. The corresponding documentation for the interface will describe the standard to which the interface conforms. Standards are typically controlled by a standards development organization, and changes can be made to the interface in accordance with approved changes to the standard. This stability level can also apply to interfaces that have been adopted (without a formal standard) by an industry convention. Support is provided for only the specified versions of a standard; support for later versions is not guaranteed.</td>
</tr>
</tbody>
</table>

### 16.2 Dependency Classes

Since Oracle Linux and DTrace support a variety of operating platforms and processors, DTrace also labels interfaces with a **dependency class** that tells you whether an interface is common to all Oracle Linux platforms and processors, or whether the interface is associated with a particular system architecture. The dependency class is orthogonal to the stability levels described earlier. For example, a DTrace interface can be Stable but only supported on x86_64 microprocessors, or it can be Unstable but common to all Oracle Linux systems. The DTrace dependency classes are described in the following table in order from least common (that is, most specific to a particular architecture) to most common (that is, common to all architectures).

<table>
<thead>
<tr>
<th>Dependency Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>The interface has an unknown set of architectural dependencies. DTrace does not necessarily know the architectural dependencies of all entities, such as data types defined in the operating system implementation. The Unknown label is typically applied to interfaces of very low stability for which dependencies cannot be computed. The interface might not be available when using DTrace on any architecture other than the one you are currently using.</td>
</tr>
<tr>
<td>CPU</td>
<td>The interface is specific to the CPU model of the current system. Interfaces with CPU model dependencies might not be available on other CPU implementations, even if those CPUs export the same instruction set architecture (ISA).</td>
</tr>
</tbody>
</table>
16.3 Interface Attributes

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

```
name_stability / data_stability / dependency_class
```

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

The data stability of an interface is distinct from the stability associated with the interface name. This stability level describes the commitment to maintain the data formats used by the interface and any associated data semantics.

The dependency class of an interface is distinct from its name and data stability, and describes whether the interface is specific to the current operating platform or microprocessor.

DTrace and the D compiler track the stability attributes for all of the DTrace interface entities, including providers, probe descriptions, D variables, D functions, types, and program statements themselves, as we will see shortly. Notice that all three values can vary independently. For example, the `curthread` D variable has Stable/Private/Common attributes: the variable name is Stable and is Common to all Oracle Linux operating platforms, but this variable provides access to a Private data format that is an artifact of the Oracle Linux kernel implementation. Most D variables are provided with Stable/Stable/Common attributes, as are the variables you define.

16.4 Stability Computations and Reports

The D compiler performs stability computations for each of the probe descriptions and action statements in your D programs. You can use the `-v` option to display a report of your program's stability. The following example uses a program written on the command line:
You may also wish to combine the `-v` option with the `-e` option, which tells `dtrace` to compile but not execute your D program, so that you can determine program stability without having to enable any probes and execute your program. Here is another example stability report:

```
# dtrace -ev -n dtrace:::BEGIN'(trace(curthread->t_procp);)'
```

Stability data for description dtrace:::BEGIN:

- Minimum probe description attributes:
  - Identifier Names: Evolving
  - Data Semantics: Evolving
  - Dependency Class: Common

- Minimum probe statement attributes:
  - Identifier Names: Stable
  - Data Semantics: Private
  - Dependency Class: Common

Notice that in our new program, we have referenced the D variable `curthread`, which has a Stable name, but Private data semantics (that is, if you look at it, you are accessing Private implementation details of the kernel), and this status is now reflected in the program's stability report. Stability attributes in the program report are computed by selecting the minimum stability level and class out of the corresponding values for each interface attributes triplet.

Stability attributes are computed for a probe description by taking the minimum stability attributes of all specified probe description fields according to the attributes published by the provider. The attributes of the available DTrace providers are shown in the chapter corresponding to each provider. DTrace providers export a stability attributes triplet for each of the four description fields for all probes published by that provider. Therefore, a provider's name may have a greater stability than the individual probes it exports. For example, the probe description:

```
fbt:::
```

indicating that DTrace should trace entry and return from all kernel functions, has greater stability than the probe description:

```
fbt:foo:bar:entry
```

which names a specific internal function `bar` in the kernel module `foo`. For simplicity, most providers use a single set of attributes for all of the individual module function name values that they publish. Providers also specify attributes for the `args[]` array, as the stability of any probe arguments varies by provider.

If the provider field is not specified in a probe description, then the description is assigned the stability attributes Unstable/Unstable/Common because the description might end up matching probes of providers that do not yet exist when used on a future Oracle Linux version. As such, Oracle is not able to provide
guarantees about the future stability and behavior of this program. You should always explicitly specify
the provider when writing your D program clauses. In addition, any probe description fields that contain
pattern matching characters (see Section 2.1, “D Program Structure”) or macro variables such as $1
(see Chapter 9, Scripting) are treated as if they are unspecified because these description patterns might
expand to match providers or probes released in future versions of DTrace and Oracle Linux.

Stability attributes are computed for most D language statements by taking the minimum stability and class
of the entities in the statement. For example, the following D language entities have the following attributes:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D built-in variable</td>
<td>Stable/Private/ Common</td>
</tr>
<tr>
<td>curthread</td>
<td></td>
</tr>
<tr>
<td>D user-defined variable</td>
<td>Stable/Stable/Common</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

If you write the following D program statement:

```d
x += curthread->t_pri;
```

then the resulting attributes of the statement are Stable/Private/Common, the minimum attributes
associated with the operands `curthread` and `x`. The stability of an expression is computed by taking the
minimum stability attributes of each of the operands.

Any D variables you define in your program are automatically assigned the attributes Stable/Stable/
Common. In addition, the D language grammar and D operators are implicitly assigned the attributes
Stable/Stable/Common. References to kernel symbols using the backquote (`) operator are always
assigned the attributes Private/Private/Unknown because they reflect implementation artifacts. Types that
you define in your D program source code, specifically those that are associated with the C and D type
namespace, are assigned the attributes Stable/Stable/Common. Types that are defined in the operating
system implementation and provided by other type namespaces are assigned the attributes Private/
Private/Unknown. The D type cast operator yields an expression whose stability attributes are the minimum
of the input expression's attributes and the attributes of the cast output type.

If you use the C preprocessor to include C system header files, these types will be associated with the
C type namespace and will be assigned the attributes Stable/Stable/Common as the D compiler has no
choice but to assume that you are taking responsibility for these declarations. It is therefore possible to
mislead yourself about your program's stability if you use the C preprocessor to include a header file
containing implementation artifacts. You should always consult the documentation corresponding to the
header files you are including in order to determine the correct stability levels.

### 16.5 Stability Enforcement

When developing a DTrace script or layered tool, you may wish to identify the specific source of
stability issues or ensure that your program has a desired set of stability attributes. You can use the `-x amin=_attributes_` option to force the D compiler to produce an error when any attributes computation
results in a triplet of attributes less than the minimum values you specify on the command line. The
following example demonstrates the use of `-x amin` using a snippet of D program source. Notice that
attributes are specified using three labels delimited by `/` in the usual order.

```bash
$ dtrace -x amin=Evolve/Evolving/Common
-ev -n dtrace::BEGIN'{trace(curthread->t_procp);}'
dtrace: invalid probe specifier dtrace::BEGIN{trace(curthread->t_procp)};:
   in action list: attributes for scalar curthread (Stable/Private/Common)
   are less than predefined minimum
```

187
Chapter 17 Translators

Table of Contents

17.1 Translator Declarations ................................................................. 189
17.2 xlate Operator .............................................................................. 190
17.3 Process Model Translators .............................................................. 191
17.4 Stable Translations ......................................................................... 191

In Chapter 16, Stability, we learned about how DTrace computes and reports program stability attributes. Ideally, we would like to construct our DTrace programs by consuming only Stable or Evolving interfaces. Unfortunately, when debugging a low-level problem or measuring system performance, you may need to enable probes that are associated with internal operating system routines such as functions in the kernel, rather than probes associated with more stable interfaces such as system calls. The data available at probe locations deep within the software stack is often a collection of implementation artifacts rather than more stable data structures such as those associated with the Oracle Linux system call interfaces. In order to aid you in writing stable D programs, DTrace provides a facility to translate implementation artifacts into stable data structures accessible from your D program statements.

17.1 Translator Declarations

A translator is a collection of D assignment statements provided by the supplier of an interface that can be used to translate an input expression into an object of struct type. To understand the need for and use of translators, we will consider as an example the ANSI C standard library routines defined in stdio.h. These routines operate on a data structure named FILE whose implementation artifacts are abstracted away from C programmers. A standard technique for creating a data structure abstraction is to provide only a forward declaration of a data structure in public header files, while keeping the corresponding struct definition in a separate private header file.

If you are writing a C program and wish to know the file descriptor corresponding to a FILE struct, you can use the fileno() function to obtain the descriptor rather than dereferencing a member of the FILE struct directly. The Oracle Linux header files enforce this rule by defining FILE as an opaque forward declaration tag so it cannot be dereferenced directly by C programs that include <stdio.h>. Inside the /lib/libc.so.6 library, you could imagine that fileno is implemented in C something like this (of course, the real implementation is nothing like this fictitious example):

```c
int fileno(FILE *fp)
{
    struct file_impl *ip = (struct file_impl *)fp;
    return (ip->fd);
}
```

Our hypothetical fileno takes a FILE pointer as an argument and casts it to a pointer to a corresponding internal libc structure, struct file_impl, and then returns the value of the fd member of the implementation structure.

Unfortunately, observability software such as DTrace needs to be able to peer inside the implementation if it is to provide useful results. DTrace cannot call arbitrary C functions defined in Oracle Linux libraries or in the kernel. You could declare a copy of struct file_impl in your D program in order to instrument the routines declared in stdio.h, but then your D program would rely on Private implementation artifacts of the library that might break in a future micro or minor release, or even in a patch. Ideally, we want to
provide a construct for use in D programs that is bound to the implementation of the library and is updated accordingly, but still provides an additional layer of abstraction associated with greater stability.

A new translator is created using a declaration of the form:

```d
translator output-type < input-type input-identifier > { 
    member-name = expression ;
    member-name = expression ;
    ...
};
```

The `output-type` names a `struct` that will be the result type for the translation. The `input-type` specifies the type of the input expression, and is surrounded in angle brackets `<>` and followed by an `input-identifier` that can be used in the translator expressions as an alias for the input expression. The body of the translator is surrounded in braces `{}` and terminated with a semicolon (`;`), and consists of a list of `member-names` and identifiers corresponding to translation expressions. Each member declaration must name a unique member of the `output-type` and must be assigned an expression of a type compatible with the member type, according to the rules for the D assignment (`=`) operator.

For example, we could define a `struct` of stable information about `stdio` files based on some of the available `libc` interfaces:

```d
struct file_info {
    int file_fd; /* file descriptor from fileno() */
    int file_eof; /* eof flag from feof() */
};
```

We could then define a hypothetical D translator from `FILE` to `file_info` as follows:

```d
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

In our hypothetical translator, the input expression is of type `FILE *` and is assigned the `input-identifier` `F`. The identifier `F` can then be used in the translator member expressions as a variable of type `FILE *` that is only visible within the body of the translator declaration. To determine the value of the output `file_fd` member, the translator performs a cast and dereference similar to the hypothetical implementation of `fileno()` shown above. A similar translation is performed to obtain the value of the `EOF` indicator.

### 17.2 xlate Operator

The D operator `xlate` is used to perform a translation from an input expression to one of the defined translation output structures. The `xlate` operator is used in an expression of the form:

```d
xlate <output-type> ( input-expression )
```

For example, to invoke the hypothetical translator for `FILE` structs defined above and access the `file_fd` member, you would write the expression:

```d
xlate <struct file_info *> (f)->file_fd;
```

where `f` is a D variable of type `FILE *`. The `xlate` expression itself is assigned the type defined by the `output-type`. Once a translator is defined, it can be used to translate input expressions to either the translator output `struct` type, or to a pointer to that `struct`.

If you translate an input expression to a `struct`, you can either dereference a particular member of the output immediately using the `"."` operator, or you can assign the entire translated `struct` to another
D variable to make a copy of the values of all the members. If you dereference a single member, the D
compiler will only generate code corresponding to the expression for that member. You may not apply the
& operator to a translated struct to obtain its address, as the data object itself does not exist until it is
copied or one of its members is referenced.

If you translate an input expression to a pointer to a struct, you can either dereference a particular
member of the output immediately using the -> operator, or you can dereference the pointer using the
unary * operator, in which case the result behaves as if you translated the expression to a struct. If you
dereference a single member, the D compiler will only generate code corresponding to the expression for
that member. You may not assign a translated pointer to another D variable as the data object itself does
not exist until it is copied or one of its members is referenced, and therefore cannot be addressed.

A translator declaration may omit expressions for one or more members of the output type. If an xlate
expression is used to access a member for which no translation expression is defined, the D compiler
will produce an appropriate error message and abort the program compilation. If the entire output type is
copied by means of a structure assignment, any members for which no translation expressions are defined
will be filled with zeroes.

In order to find a matching translator for an xlate operation, the D compiler examines the set of available
translators in the following order:

• The compiler looks for a translation from the exact input expression type to the exact output type.

• The compiler resolves the input and output types by following any typedef aliases to the underlying
type names, and then looks for a translation from the resolved input type to the resolved output type.

• The compiler looks for a translation from a compatible input type to the resolved output type. The
compiler uses the same rules as it does for determining compatibility of function call arguments with
function prototypes in order to determine if an input expression type is compatible with a translator’s
input type.

If no matching translator can be found according to these rules, the D compiler produces an appropriate
error message and program compilation fails.

### 17.3 Process Model Translators

The DTrace library file /usr/lib64/dtrace/procfs.d provides a set of translators for use in your D
programs to translate from the operating system kernel implementation structure for a process descriptor
(struct task_struct) to the stable structures psinfo and lwpsinfo. These structures define useful
Stable information about processes and threads such as the process ID, process priority, command name,
initial arguments, and other data displayed by the ps command.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Input Type Attributes</th>
<th>Output Type</th>
<th>Output Type Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct task_struct *</td>
<td>Private/Private/Common</td>
<td>psinfo_t *</td>
<td>Stable/Stable/Common</td>
</tr>
<tr>
<td>struct task_struct *</td>
<td>Private/Private/Common</td>
<td>lwpsinfo_t *</td>
<td>Stable/Stable/Common</td>
</tr>
</tbody>
</table>

### 17.4 Stable Translations

While a translator provides the ability to convert information into a stable data structure, it does not
necessarily resolve all stability issues that can arise in translating data. For example, if the input expression
for an xlate operation itself references Unstable data, the resulting D program is also Unstable because
program stability is always computed as the minimum stability of the accumulated D program statements
and expressions. Therefore, it is sometimes necessary to define a specific stable input expression for a
translator in order to permit stable programs to be constructed. The D inline mechanism can be used to
facilitate such stable translations.

The DTrace procfs.d library provides the curlwpsinfo and curpsinfo variables described earlier
as stable translations. For example, the curpsinfo and curlwpsinfo variables are actually inline
declared as follows:

```d
inline psinfo_t *curpsinfo = xlate <psinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curpsinfo
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

Both curpsinfo and curlwpsinfo are defined as inline translations from the curthread variable, a
pointer to the kernel's Private data structure representing a process descriptor, to the Stable lwpsinfo_t
type. The D compiler processes this library file and caches the inline declarations, making curpsinfo
and curlwpsinfo appear as any other D variable. The #pragma statement following the declaration is
used to explicitly reset the attributes of the curpsinfo and curlwpsinfo identifiers to Stable/Stable/
Common, masking the reference to curthread in the inline expressions.
Chapter 18 Versioning

Table of Contents

18.1 Versions and Releases ................................................................. 193
18.2 Versioning Options ................................................................. 194
18.3 Provider Versioning ................................................................. 195

In Chapter 16, Stability, we learned about the DTrace features for determining the stability attributes of D programs that you create. Once you have created a D program with the appropriate stability attributes, you may also wish to bind this program to a particular version of the D programming interface. The D interface version is a label applied to a particular set of types, variables, functions, constants, and translators made available to you by the D compiler. If you specify a binding to a specific version of the D programming interface, you ensure that you can recompile your program on future versions of DTrace without encountering conflicts between program identifiers that you define and identifiers defined in future versions of the D programming interface. You should establish version bindings for any D programs that you wish to install as persistent scripts (see Chapter 9, Scripting) or use in layered tools.

Note

DTrace versioning on Oracle Linux is not currently interoperable with DTrace versioning on other operating system platforms.

18.1 Versions and Releases

The D compiler labels sets of types, variables, functions, constants, and translators corresponding to a particular software release using a version string. A version string is a period-delimited sequence of decimal integers that takes one of the following forms:

\[
x\]

Major release

\[x.y\]

Minor release

\[x.y.z\]

Micro release

Versions are compared by comparing the integers from left to right. If the leftmost integers are not equal, the string with the greater integer is the greater (and therefore more recent) version. If the leftmost integers are equal, the comparison proceeds to the next integer in order from left to right to determine the result. All unspecified integers in a version string are interpreted as having the value zero during a version comparison.

The DTrace version strings correspond to the standard nomenclature for interface versions. A change in the D programming interface is accompanied by a new version string. Table 18.1, “DTrace Release Versions” summarizes the version strings used by DTrace and the likely significance of the corresponding DTrace software release.

Table 18.1 DTrace Release Versions

<table>
<thead>
<tr>
<th>Release</th>
<th>Version</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>(x.0)</td>
<td>A Major release is likely to contain major feature additions; adhere to different, possibly incompatible Standard revisions; and though unlikely, could change, drop, or replace Standard</td>
</tr>
</tbody>
</table>
**Versioning Options**

<table>
<thead>
<tr>
<th>Release</th>
<th>Version</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor</td>
<td>x.y</td>
<td>Compared to an x.0 or earlier version (where y is not equal to zero), a new Minor release is likely to contain minor feature additions, compatible Standard and Stable interfaces, possibly incompatible Evolving interfaces, or likely incompatible Unstable interfaces. These changes may include new built-in D types, variables, functions, constants, and translators. In addition, a Minor release may remove support for interfaces previously labeled as Obsolete (see Chapter 16, Stability).</td>
</tr>
<tr>
<td>Micro</td>
<td>x.y.z</td>
<td>Micro releases are intended to be interface compatible with the previous release (where z is not equal to zero), but are likely to include bug fixes, performance enhancements, and support for additional hardware.</td>
</tr>
</tbody>
</table>

In general, each new version of the D programming interface will provide a superset of the capabilities offered by the previous version, with the exception of any Obsolete interfaces that have been removed.

### 18.2 Versioning Options

By default, any D programs you compile using `dtrace -s` or specify using the `dtrace -P, -m, -f, -n,` or `-i` command-line options are bound to the most recent D programming interface version offered by the D compiler. You can determine the current D programming interface version by using the `-V` option:

```
$ dtrace -V
```

```
dtrace: Sun D 1.6.3
```

If you wish to establish a binding to a specific version of the D programming interface, you can set the `version` option to an appropriate version string. Similar to other DTrace options (see Chapter 10, Options and Tunables), you can set the version option on the command line using `dtrace -x`:

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

Alternatively, you can use the `#pragma D option` syntax to set the option in your D program source file:

```
#pragma D option version=0.3
BEGIN
{
  trace("hello");
}
```

If you use the `#pragma D option` syntax to request a version binding, you must place this directive at the top of your D program file prior to any other declarations and probe clauses. If the version binding argument is not a valid version string or refers to a version not offered by the D compiler, an appropriate error message will be produced and compilation will fail. You can therefore also use the version binding facility to cause execution of a D script on an older version of DTrace to fail with an obvious error message.

Before compiling your program declarations and clauses, the D compiler loads the set of D types, functions, constants, and translators for the appropriate interface version into the compiler namespaces. Therefore, any version binding options you specify simply control the set of identifiers, types, and translators that are visible to your program in addition to the variables, types, and translators that your program defines. Version binding prevents the D compiler from loading newer interfaces that may define identifiers or translators that conflict with declarations in your program source code and would therefore
cause a compilation error. See Section 2.8.1, “Identifier Names and Keywords” for tips on how to pick identifier names that are unlikely to conflict with interfaces offered by future versions of DTrace.

## 18.3 Provider Versioning

Unlike interfaces offered by the D compiler, interfaces offered by DTrace providers (that is, probes and probe arguments) are not affected by or associated with the D programming interface or the previously described version binding options. The available provider interfaces are established as part of loading your compiled instrumentation into the DTrace software in the operating system kernel and vary depending on your instruction set architecture, operating platform, processor, the software installed on your Oracle Linux system, and your current security privileges. The D compiler and DTrace runtime examine the probes described in your D program clauses and report appropriate error messages when probes requested by your D program are not available. These features are orthogonal to the D programming interface version because DTrace providers do not export interfaces that can conflict with definitions in your D programs; that is, you can only enable probes in D, you cannot define them, and probe names are kept in a separate namespace from other D program identifiers.

DTrace providers are delivered with a particular release of Oracle Linux and are described in the corresponding version of the Oracle Linux DTrace Guide. The chapter of this guide corresponding to each provider will also describe any relevant changes to or new features offered by a given provider. You can use `dtrace -l` to explore the set of providers and probes available on your Oracle Linux system. Providers label their interfaces using the DTrace stability attributes, and you can use the DTrace stability reporting features (see Chapter 16, Stability) to determine whether the provider interfaces used by your D program are likely to change or be offered in future Oracle Linux releases.