DTrace User Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–5488–10
May 2006

060328@14558

Contents

Preface

The *DTrace User Guide* is a lightweight introduction to the powerful tracing and analysis tool DTrace. In this book, you will find a description of DTrace and its capabilities, as well as directions on how to use DTrace to perform relatively simple and common tasks.

Who Should Use This Book

DTrace is a comprehensive dynamic tracing facility that is built into Solaris. You can use the DTrace facility can be used to examine the behavior of user programs or the behavior of the operating system. DTrace can be used by system administrators or application developers on live production systems.

DTrace allows Solaris developers and administrators to:

- Implement custom scripts that use the DTrace facility
- Implement layered tools that use DTrace to retrieve trace data

This book is not a comprehensive guide to DTrace or the D scripting language. Please refer to the *Solaris Dynamic Tracing Guide* for in-depth reference information.

Before You Read This Book

Basic familiarity with a programming language such as C or a scripting language such as `awk(1)` or `perl(1)` will help you learn DTrace and the D programming language faster, but you need not be an expert in any of these areas. If you have never written a program or script before in any language, "Related Books" on page 5 provides references to other documents you might find useful.

Related Books

For an in depth reference to DTrace, see the *Solaris Dynamic Tracing Guide*. These books and papers are recommended and related to the tasks that you need to perform with DTrace:

- Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, 1988. ISBN 0–13–110370–9
- Mauro, Jim and McDougall, Richard. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001. ISBN 0-13-022496-0

- Vahalia, Uresh. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996. ISBN 0-13-101908-2

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

CHAPTER 1

Introduction

DTrace is a comprehensive dynamic tracing facility that is built into Solaris. DTrace can be used by administrators and developers, and can safely be used on live production systems. DTrace enables you to examine the behavior of user programs as well as the behavior of the operating system. Users of DTrace can create custom programs with the D scripting language. Custom programs provide the ability to dynamically instrument the system. Custom programs provide immediate, concise answers to specific questions about the behavior of particular applications.

DTrace Capabilities

The DTrace framework provides instrumentation points that are called *probes*. A DTrace user can use a probe to record and display relevant information about a kernel or user process. Each DTrace probe is activated by a specific behavior. This probe activation is referred to as *firing*. As an example, consider a probe that fires on entry into an arbitrary kernel function. This example probe can display the following information:

- Any argument that is passed to the function
- Any global variable in the kernel
- A timestamp that indicates when the function was called
- A stack trace that indicates the section of code that called the function
- The process that was running at the time the function was called
- The thread that made the function call

When a probe fires, you can specify a particular *action* for DTrace to take. A DTrace action usually records an interesting aspect of system behavior, such as a timestamp or a function argument.

Probes are implemented by *providers*. A probe provider is a kernel module that enables a given probe to fire. For example, the function boundary tracing provider `fbt` provides entry and return probes for almost every function in every kernel module.

DTrace has significant data management capabilities. These capabilities enable DTrace users to prune the data reported by probes, avoiding the overhead involved in generating and then filtering unwanted data. DTrace also provides mechanisms for tracing during the boot process and for retrieving data from a kernel crash dump. All of the instrumentation in DTrace is dynamic. Probes are enabled discretely at the time that the probes are used, and inactive probes present no instrumented code.

A DTrace *consumer* is any process that interacts with the DTrace framework. While `dtrace`(1M) is the primary DTrace consumer, other consumers exist. These additional consumers mostly consist of new versions of existing utilities such as `lockstat`(1M). The DTrace framework has no limit on the number of concurrent consumers.

The behavior of DTrace can be modified with the use of scripts that are written in the D language, which is structured similarly to C. The D language provides access to kernel C types and kernel static and kernel global variables. The D language supports ANSI C operators.

Architecture overview

The DTrace facility consists of the following components:

- User level consumer programs such as `dtrace`
- Providers, packaged as kernel modules, that provide probes to gather tracing data
- A library interface that consumer programs use to access the DTrace facility through the `dtrace`(7D) kernel driver

DTrace Providers

A provider represents a methodology for instrumenting the system. Providers make probes available to the DTrace framework. DTrace sends information to a provider regarding when to enable a probe. When an enabled probe fires, the provider transfers control to DTrace.

Providers are packaged as a set of kernel modules. Each module performs a particular kind of instrumentation to create probes. When you use DTrace, each provider has the ability to publish the probes it can provide to the DTrace framework. You can enable and bind tracing actions to any of the published probes.

Some providers have the capability to create new probes based on the user's tracing requests.

DTrace Probes

A probe has the following attributes:

- It is made available by a *provider*
- It identifies the *module* and the *function* that it instruments
- It has a *name*

These four attributes define a 4–tuple that serves as a unique identifier for each probe, in the format *provider:module:function:name*. Each probe also has a unique integer identifier.

DTrace Predicates

Predicates are expressions that are enclosed in slashes `/ /`. Predicates 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. If the predicate section is omitted, the actions are always executed when the probe fires.

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

DTrace Actions

Actions are user-programmable statements that the DTrace virtual machine executes within the kernel. Actions have the following properties:

- Actions are taken when a probe fires
- Actions are completely programmable in the D scripting language
- Most actions record a specified system state
- An action can change the state of the system in a precisely described way. Such actions are called *destructive actions*. Destructive actions are not allowed by default.
- Many actions use expressions in the D scripting language

D Scripting Language

You can invoke the DTrace framework directly from the command line with the `dtrace` command for simple functions. To use DTrace to perform more complex functions, write a script in the D scripting language. Use the `-s` option to load a specified script for DTrace to use. See Chapter 3 for information about using the D scripting language.

CHAPTER 2

DTrace Basics

This chapter provides a tour of the DTrace facility and provides examples of several basic tasks.

Listing Probes

You can list all DTrace probes by passing the `-l` option to the `dtrace` command:

```
# dtrace -l
ID   PROVIDER    MODULE        FUNCTION NAME
 1     dtrace                           BEGIN
 2     dtrace                           END
 3     dtrace                           ERROR
 4    syscall                     nosys entry
 5    syscall                     nosys return
 6    syscall                     rexit entry
 7    syscall                     rexit return
 8    syscall                   forkall entry
 9    syscall                   forkall return
10    syscall                      read entry
11    syscall                      read return
...
```

To count all the probes that are available on your system, you can type the following command:

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

The number of probes reported will vary depending on your operating platform and the software you have installed. Some probes do not list an entry under the `MODULE` or `FUNCTION` columns, such as the `BEGIN` and `END` probes in the previous example. Probes with blank entries in these fields do not correspond to a specifically instrumented program function or location. These probes refer to more abstract concepts, such as the end of a tracing request. A probe that has a module and function as part of its name is called an *anchored probe*. A probe that is not associated with a module and function is called an *unanchored probe*.

You can use additional options to list specific probes, as seen in the following examples.

EXAMPLE 2–1 Listing Probes by Specific Function

You can list probes that are associated with a specific function by passing that function name to DTrace with the `-f` option.

```
# dtrace -l -f cv_wait
ID      PROVIDER        MODULE        FUNCTION NAME
12921        fbt       genunix         cv_wait entry
12922        fbt       genunix         cv_wait return
```

EXAMPLE 2–2 Listing Probes by Specific Module

You can list probes that are associated with a specific module by passing that module name to DTrace with the `-m` option.

```
# dtrace -l -m sd
ID      PROVIDER        MODULE        FUNCTION NAME
17147        fbt            sd          sdopen entry
17148        fbt            sd          sdopen return
17149        fbt            sd         sdclose entry
17150        fbt            sd         sdclose return
17151        fbt            sd      sdstrategy entry
17152        fbt            sd      sdstrategy return
...
```

EXAMPLE 2–3 Listing Probes by Specific Name

You can list probes that have a specific name by passing that name to DTrace with the `-n` option.

```
# dtrace -l -n BEGIN
ID      PROVIDER        MODULE        FUNCTION NAME
1         dtrace                               BEGIN
```

EXAMPLE 2–4 Listing Probes by Provider of Origin

You can list probes that are originate from a specific provider by passing the provider name to DTrace with the `-P` option.

```
# dtrace -l -P lockstat
ID        PROVIDER        MODULE              FUNCTION NAME
469       lockstat       genunix           mutex_enter adaptive-acquire
470       lockstat       genunix           mutex_enter adaptive-block
471       lockstat       genunix           mutex_enter adaptive-spin
472       lockstat       genunix            mutex_exit adaptive-release
473       lockstat       genunix         mutex_destroy adaptive-release
474       lockstat       genunix        mutex_tryenter adaptive-acquire
...
```

EXAMPLE 2–5 Multiple Providers Supporting a Specific Function or Module

A specific function or specific module can be supported by multiple providers, as the following example shows.

```
# dtrace -l -f read
ID        PROVIDER        MODULE              FUNCTION NAME
  10       syscall                                read entry
  11       syscall                                read return
4036       sysinfo       genunix                  read readch
4040       sysinfo       genunix                  read sysread
7885           fbt       genunix                  read entry
7886           fbt       genunix                  read return
```

As the previous examples show, the output for a listing of probes displays the following information:

- The probe's uniquely assigned integer probe ID

Note – The probe ID is only unique within a given release or patch level of the Solaris operating system.

- The provider name
- The module name, if applicable
- The function name, if applicable
- The probe name

Specifying Probes in DTrace

You can fully specify a probe by listing each component of the 4–tuple that uniquely identifies that probe. The format for the probe specification is *provider:module:function:name*. An empty component in a probe specification matches anything. For example, the specification `fbt::alloc:entry` specifies a probe with the following attributes:

- The probe must be from the `fbt` provider
- The probe may be in any module
- The probe must be in the `alloc` function
- The probe must be named `entry`

Elements on the left hand side of the 4–tuple are optional. The probe specification `::open:entry` is equivalent to the specification `open:entry`. Either specification will match probes from all providers and kernel modules that have a function name of `open` and are named `entry`.

```
# dtrace -l -n open:entry
  ID      PROVIDER        MODULE              FUNCTION NAME
  14       syscall                                open entry
7386           fbt       genunix                  open entry
```

You can also describe probes with a pattern matching syntax that is similar to the syntax that is described in the *File Name Generation* section of the `sh(1)` man page. The syntax supports the special characters `*`, `?`, `[`, and `]`. The probe description `syscall::open*:entry` matches both the `open` and `open64` system calls. The `?` character represents any single character in the name. The `[` and `]` characters are used to specify a set of specific characters in the name.

Enabling Probes

You enable probes with the `dtrace` command by specifying the probes without the `-l` option. Without further directions, DTrace performs the default action when the specified probe fires. The default probe action indicates only that the specified probe has fired and does not record any other data. The following code example enables every probe in the `sd` module.

EXAMPLE 2–6 Enabling Probes by Module

```
# dtrace -m sd
CPU     ID                    FUNCTION:NAME
  0  17329          sd_media_watch_cb:entry
  0  17330         sd_media_watch_cb:return
  0  17167                    sdinfo:entry
  0  17168                   sdinfo:return
  0  17151                sdstrategy:entry
  0  17152               sdstrategy:return
  0  17661         ddi_xbuf_qstrategy:entry
  0  17662        ddi_xbuf_qstrategy:return
  0  17649              xbuf_iostart:entry
  0  17341           sd_xbuf_strategy:entry
  0  17385              sd_xbuf_init:entry
  0  17386             sd_xbuf_init:return
  0  17342          sd_xbuf_strategy:return
  0  17177     sd_mapblockaddr_iostart:entry
  0  17178    sd_mapblockaddr_iostart:return
  0  17179             sd_pm_iostart:entry
  0  17365               sd_pm_entry:entry
  0  17366              sd_pm_entry:return
  0  17180            sd_pm_iostart:return
  0  17181           sd_core_iostart:entry
  0  17407         sd_add_buf_to_waitq:entry
...
```

The output in this example shows that the default action displays the CPU where the probe fired, the integer probe ID that is assigned by DTrace, the function where the probe fired, and the probe name.

EXAMPLE 2–7 Enabling Probes by Provider

```
# dtrace -P syscall
dtrace: description 'syscall' matched 452 probes
CPU     ID                    FUNCTION:NAME
```

EXAMPLE 2–7 Enabling Probes by Provider *(Continued)*

```
  0     99                    ioctl:return
  0     98                     ioctl:entry
  0     99                    ioctl:return
  0     98                     ioctl:entry
  0     99                    ioctl:return
  0    234                 sysconfig:entry
  0    235                sysconfig:return
  0    234                 sysconfig:entry
  0    235                sysconfig:return
  0    168                 sigaction:entry
  0    169                sigaction:return
  0    168                 sigaction:entry
  0    169                sigaction:return
  0     98                     ioctl:entry
  0     99                    ioctl:return
  0    234                 sysconfig:entry
  0    235                sysconfig:return
  0     38                       brk:entry
  0     39                      brk:return
...
```

EXAMPLE 2–8 Enabling Probes by Name

```
# dtrace -n zfod
dtrace: description 'zfod' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0   4080                  anon_zero:zfod
  0   4080                  anon_zero:zfod
^C
```

EXAMPLE 2–9 Enabling Probes by Fully Specified Name

```
# dtrace -n clock:entry
dtrace: description 'clock:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0   4198                     clock:entry
^C
```

DTrace Action Basics

Actions enable DTrace to interact with the system outside of the DTrace framework. The most common actions record data to a DTrace buffer. Other actions can stop the current process, raise a specific signal on the current process, or cease tracing. Actions that change the system state are considered *destructive actions*. Data recording actions record data to the *principal buffer* by default. The principal buffer is present in every DTrace invocation and is always allocated on a per-CPU basis. Tracing and buffer allocation can be restricted to a single CPU by using the `-cpu` option. See Chapter 11, "Buffers and Buffering," in *Solaris Dynamic Tracing Guide* for more information about DTrace buffering.

The examples in this section use D expressions that consist of built-in D variables. Some of the most commonly used D variables are listed below:

`pid` This variable contains the current process ID.

`execname` This variable contains the current executable name.

`timestamp` This variable contains the time since boot, expressed in nanoseconds.

`curthread` This variable contains a pointer to the `kthread_t` structure that represents the current thread.

`probemod` This variable contains the module name of the current probe.

`probefunc` This variable contains the function name of the current probe.

`probename` This variable contains the name of the current probe.

For a complete list of the built-in variables of the D scripting language, see Variables.

The D scripting language also provides built-in functions that perform specific actions. You can find a complete list of these built-in functions at Chapter 10, "Actions and Subroutines," in *Solaris Dynamic Tracing Guide*. The `trace()` function records the result of a D expression to the trace buffer, as in the following examples:

- `trace(pid)` traces the current process ID
- `trace(execname)` traces the name of the current executable
- `trace(curthread->t_pri)` traces the `t_pri` field of the current thread
- `trace(probefunc)` traces the function name of the probe

To indicate a particular action you want a probe to take, type the name of the action between `{}` characters, as in the following example.

EXAMPLE 2–10 Specifying a Probe's Action

```
# dtrace -n 'readch {trace(pid)}'
dtrace: description 'readch ' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0   4036                      read:readch          2040
  0   4036                      read:readch          2177
```

EXAMPLE 2–10 Specifying a Probe's Action *(Continued)*

```
  0   4036                       read:readch          2177
  0   4036                       read:readch          2040
  0   4036                       read:readch          2181
  0   4036                       read:readch          2181
  0   4036                       read:readch             7
...
```

Since the requested action is `trace(pid)`, the process identification number (PID) appears in the last column of the output.

EXAMPLE 2–11 Tracing an Executable Name

```
# dtrace -m 'ufs {trace(execname)}'
dtrace: description 'ufs ' matched 889 probes
CPU     ID                    FUNCTION:NAME
  0  14977                 ufs_lookup:entry            ls
  0  15748                ufs_iaccess:entry            ls
  0  15749               ufs_iaccess:return            ls
  0  14978                ufs_lookup:return            ls
...
  0  15007                   ufs_seek:entry         utmpd
  0  15008                  ufs_seek:return         utmpd
  0  14963                  ufs_close:entry         utmpd
^C
```

EXAMPLE 2–12 Tracing A System Call's Time of Entry

```
# dtrace -n 'syscall:::entry {trace(timestamp)}'
dtrace: description 'syscall:::entry ' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0    312                     portfs:entry    157088479572713
  0     98                      ioctl:entry    157088479637542
  0     98                      ioctl:entry    157088479674339
  0    234                  sysconfig:entry    157088479767243
...
  0     98                      ioctl:entry    157088481033225
  0     60                      fstat:entry    157088481050686
  0     60                      fstat:entry    157088481074680
^C
```

EXAMPLE 2–13 Specifying Multiple Actions

To specify multiple actions, list the actions separated by the `;` character.

EXAMPLE 2–13 Specifying Multiple Actions *(Continued)*

```
# dtrace -n 'zfod {trace(pid);trace(execname)}'
dtrace: description 'zfod ' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0   4080                   anon_zero:zfod    2195   dtrace
  0   4080                   anon_zero:zfod    2195   dtrace
  0   4080                   anon_zero:zfod    2195   dtrace
  0   4080                   anon_zero:zfod    2195   dtrace
  0   4080                   anon_zero:zfod    2195   dtrace
  0   4080                   anon_zero:zfod    2197   bash
  0   4080                   anon_zero:zfod    2207   vi
  0   4080                   anon_zero:zfod    2207   vi
...
```

Data Recording Actions

The actions in this section record data to the principal buffer by default, but each action may also be used to record data to speculative buffers. See "Speculative Tracing" on page 54 for more details on speculative buffers.

The `trace()` function

`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 `tracemem()` function

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

The `tracemem()` action copies data from an address in memory to a buffer . The number of bytes that this action copies is specified in *nbytes*. The address that the data is copied from is specified in *addr* as a D expression. The buffer that the data is copied to is specified in *buf*.

The `printf()` function

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

Like the `trace()` action, the `printf()` action traces D expressions. However, the `printf()` action lets you control formatting in ways similar to the `printf`(3C) function. Like the `printf` function, 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 the `dtrace` command according to the specified format string.

For more information on the `printf()` action, see Chapter 12, "Output Formatting," in *Solaris Dynamic Tracing Guide*.

The `printa()` function

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

The `printa()` action enables you to display and format aggregations. See Chapter 9, "Aggregations," in *Solaris Dynamic Tracing Guide* for more detail on aggregations. If a *format* value is not provided, the `printa()` action only traces a directive to the DTrace consumer. The consumer that receives that directive processes and displays the aggregation with the default format. See Chapter 12, "Output Formatting," in *Solaris Dynamic Tracing Guide* for a more detailed description of the `printa()` format string.

The `stack()` function

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

The `stack()` action records a kernel stack trace to the directed buffer. The depth of the kernel stack is given by the value given in *nframes*. If no value is given for *nframes*, the stack action records a number of stack frames specified by the `stackframes` option.

The `ustack()` function

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

The `ustack()` action records a user stack trace to the directed buffer. The depth of the user stack is equal to the value specified in *nframes*. If there is no value for *nframes*, the `ustack` action records a number of stack frames that is specified by the `ustackframes` option. The `ustack()` action determines the address of the calling frames when the probe fires. The `ustack()` action does not translate the stack frames into symbols until the DTrace consumer processes the `ustack()` action at the user level. If a value for *strsize* is specified and not zero, the `ustack()` action allocates the specified amount of string space and uses it to perform address-to-symbol translation directly from the kernel.

The `jstack()` function

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

The `jstack()` action is an alias for `ustack()` that uses the value specified by the `jstackframes` option for the number of stack frames. The `jstack` action uses the value specified by the `jstackstrsize` option to determine the string space size. The `jstacksize` action defaults to a non-zero value.

Destructive Actions

You must explicitly enable destructive actions in order to use them. You can enable destructive actions by using the `-w` option. If you attempt to use destructive actions in `dtrace` without explicitly enabling them, `dtrace` fails with a message similar to the following example:

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

For more information on DTrace actions, including destructive actions, see Chapter 10, "Actions and Subroutines," in *Solaris Dynamic Tracing Guide*.

Process Destructive Actions

Some actions are destructive only to a particular process. These actions are available to users with the `dtrace_proc` or `dtrace_user` privileges. See Chapter 35, "Security," in *Solaris Dynamic Tracing Guide* for details on DTrace security privileges.

The `stop()` function

When a probe fires with the `stop()` action enabled, the process that fired that probe stops upon leaving the kernel. This process stops in the same way as a process that is stopped by a `proc`(4) action.

The `raise()` function

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

The `raise()` action sends the specified signal to the currently running process.

The `copyout()` function

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

The `copyout()` action copies data from a buffer to an address in memory. The number of bytes that this action copies is specified in *nbytes*. The buffer that the data is copied from is specified in *buf*. The address that the data is copied to is specified in *addr*. That address is in the address space of the process that is associated with the current thread.

The `copyoutstr()` function

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

The `copyoutstr()` action copies a string to an address in memory. The string to copy is specified in *str*. The address that the string is copied to is specified in *addr*. That address is in the address space of the process that is associated with the current thread.

The `system()` function

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

The `system()` action causes the program specified by *program* to be executed by the system as if it were given to the shell as input.

Kernel Destructive Actions

Some destructive actions are destructive to the entire system. Use these actions with caution. These actions affect every process on the system and may affect other systems, depending upon the affected system's network services.

The `breakpoint()` function

```
void breakpoint(void)
```

The `breakpoint()` action induces a kernel breakpoint, causing the system to stop and transfer control to the kernel debugger. The kernel debugger will emit a string that denotes the DTrace probe that triggered the action.

The `panic()` function

```
void panic(void)
```

When a probe with the `panic()` action triggers, the kernel panics. This action can force a system crash dump at a time of interest. You can use this action in conjunction with ring buffering and postmortem analysis to diagnose a system problem. For more information, see Chapter 11, "Buffers and Buffering," in *Solaris Dynamic Tracing Guide* and Chapter 37, "Postmortem Tracing," in *Solaris Dynamic Tracing Guide* respectively.

The `chill()` function

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

When a probe with the `chill()` action triggers, DTrace spins for the specified number of nanoseconds. The `chill()` action is useful for exploring problems related to timing. Because interrupts are disabled while in DTrace probe context, any use of `chill()` will induce interrupt latency, scheduling latency, dispatch latency.

DTrace Aggregations

For performance-related questions, aggregated data is often more useful than individual data points. DTrace provides several built-in aggregating functions. When an aggregating function is applied to subsets of a collection of data, then applied again to the results of the analysis of those subsets, the results are identical to the results returned by the aggregating function when it is applied to the collection as a whole.

The DTrace facility stores a running count of data items for aggregations. The aggregating functions store only the current intermediate result and the new element that the function is being applied to. The intermediate results are allocated on a per-CPU basis. Because this allocation scheme does not require locks, the implementation is inherently scalable.

DTrace Aggregation Syntax

A DTrace aggregation takes the following general form:

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

In this general form, the variables are defined as follows:

`name` The name of the aggregation, preceded by the `@` character.

`keys` A comma-separated list of D expressions.

`aggfunc` One of the DTrace aggregating functions.

`args` A comma-separated list of arguments appropriate to the aggregating function.

TABLE 2–1 DTrace Aggregating Functions

Function Name	Arguments	Result
`count`	none	The number of times that the `count` function is called.
`sum`	scalar expression	The total value of the specified expressions.
`avg`	scalar expression	The arithmetic average of the specified expressions.
`min`	scalar expression	The smallest value among the specified expressions.
`max`	scalar expression	The largest value among the specified expressions.
`lquantize`	scalar expression, lower bound, upper bound, step value	A linear frequency distribution of the values of the specified expressions that is sized by the specified range. This aggregating function increments the value in the *highest* bucket that is *less* than the specified expression.

TABLE 2–1 DTrace Aggregating Functions *(Continued)*

Function Name	Arguments	Result
`quantize`	scalar expression	A power-of-two frequency distribution of the values of the specified expressions. This aggregating function increments the value in the *highest* power-of-two bucket that is *less* than the specified expression.

EXAMPLE 2–14 Using an Aggregating Function

This example uses the `count` aggregating function to count the number of `write`(2) system calls per process. The aggregation does not output any data until the `dtrace` command is terminated. The output data represents a summary of the data collected during the time that the `dtrace` command was active.

```
# cat writes.d
#!/usr/sbin/dtrace -s
syscall::write:entry]
{   @numWrites[execname] = count();
}

# ./writes.d
dtrace: script 'writes.d' matched 1 probe
^C
  dtrace                          1
  date                            1
  bash                            3
  grep                           20
  file                          197
  ls                            201
```

CHAPTER 3

Scripting With the D Language

This chapter discusses the basic information that you need to start writing your own D language scripts.

Writing D Scripts

Complex sets of DTrace probes can become difficult to manage on the command line. The `dtrace` command supports scripts. You can specify a script by passing the `-s` option, along with the script's file name, to the `dtrace` command. You can also create executable DTrace interpreter files. A DTrace interpreter file always begins with the line **`#!/usr/sbin/dtrace -s`**.

Executable D Scripts

This example script, named `syscall.d`, traces the executable name every time the executable enters each system call:

```
syscall:::entry
{
   trace(execname);
}
```

Note that the filename ends with a .d suffix. This is the conventional ending for D scripts. You can run this script off the DTrace command line with the following command:

```
# dtrace -s syscall.d
dtrace: description 'syscall ' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0    312                    pollsys:entry    java
  0     98                      ioctl:entry    dtrace
  0     98                      ioctl:entry    dtrace
  0    234                  sysconfig:entry    dtrace
  0    234                  sysconfig:entry    dtrace
```

```
  0    168                   sigaction:entry    dtrace
  0    168                   sigaction:entry    dtrace
  0     98                       ioctl:entry    dtrace
^C
```

You can run the script by entering the filename at the command line by following two steps. First, verify that the first line of the file invokes the interpreter. The interpreter invocation line is `#!/usr/sbin/dtrace -s`. Then set the execute permission for the file.

EXAMPLE 3–1 Running a D Script from the Command Line

```
# cat syscall.d
#!/usr/sbin/dtrace -s

syscall:::entry
{
   trace(execname);
}

# chmod +x syscall.d
# ls -l syscall.d
-rwxr-xr-x   1 root     other       62 May 12 11:30 syscall.d
# ./syscall.d
dtrace: script './syscall.d' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0     98                       ioctl:entry    dtrace
  0     98                       ioctl:entry    dtrace
  0    312                     pollsys:entry    java
  0    312                     pollsys:entry    java
  0    312                     pollsys:entry    java
  0     98                       ioctl:entry    dtrace
  0     98                       ioctl:entry    dtrace
  0    234                   sysconfig:entry    dtrace
  0    234                   sysconfig:entry    dtrace
^C
```

D Literal Strings

The D language supports literal strings. DTrace represents strings as an array of characters terminated by a null byte. The visible part of the string varies in length depending on the location of the null byte. DTrace stores each string in a fixed-size array to ensure that each probe traces a consistent amount of data. Strings cannot exceed the length of the predefined string limit. The limit can be modified in your D program or on the `dtrace` command line by tuning the `strsize` option. Refer to Chapter 16, "Options and Tunables," in *Solaris Dynamic Tracing Guide* 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. See Chapter 6, "Strings," in *Solaris Dynamic Tracing Guide* for more information about D literal strings.

EXAMPLE 3–2 Using D Literal Strings With The `trace()` Function

```
# cat string.d

#!/usr/sbin/dtrace -s

fbt::bdev_strategy:entry
{
   trace(execname);
   trace(" is initiating a disk I/O\n");
}
```

The `\n` symbol at the end of the literal string produces a new line. To run this script, enter the following command:

```
# dtrace -s string.d
dtrace: script 'string.d' matched 1 probes
CPU     ID                    FUNCTION:NAME
  0   9215              bdev_strategy:entry   bash is initiating a disk I/O

  0   9215              bdev_strategy:entry   vi is initiating a disk I/O

  0   9215              bdev_strategy:entry   vi is initiating a disk I/O

  0   9215              bdev_strategy:entry   sched is initiating a disk I/O

^C
```

The `-q` option of the `dtrace` command only records the actions that are explicitly stated in the script or command line invocation. This option suppresses the default output that the `dtrace` command normally produces.

```
# dtrace -q -s string.d
ls is initiating a disk I/O
cat is initiating a disk I/O
fsflush is initiating a disk I/O
vi is initiating a disk I/O
^C
```

Creating D Scripts That Use Arguments

You can use the `dtrace` command to create executable interpreter files. The file must have execute permission. The initial line of the file must be `#!/usr/sbin/dtrace -s`. You can specify other options to the `dtrace` command on this line. You must specify the options with only one dash (`-`). List the `s` option last, as in the following example.

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

You can specify options for the `dtrace` command by using `#pragma` lines in the D script, as in the following D fragment:

```
# cat -n mem2.d
     1  #!/usr/sbin/dtrace -s
     2
     3  #pragma D option quiet
     4  #pragma D option verbose
     5
     6  vminfo:::
     ...
```

The following table lists the option names that you can use in `#pragma` lines.

TABLE 3–1 DTrace Consumer Options

Option Name	Value	`dtrace` Alias	Description
`aggrate`	*time*		Rate of aggregation reading
`aggsize`	*size*		Aggregation buffer size
`bufresize`	`auto` or `manual`		Buffer resizing policy
`bufsize`	*size*	`-b`	Principal buffer size
`cleanrate`	*time*		Cleaning rate
`cpu`	*scalar*	`-c`	CPU on which to enable tracing
`defaultargs`	—		Allow references to unspecified macro arguments
`destructive`	—	`-w`	Allow destructive actions
`dynvarsize`	*size*		Dynamic variable space size
`flowindent`	—	`-F`	Indent function entry and prefix with `->`; unindent function return and prefix with `<-`
`grabanon`	—	`-a`	Claim anonymous state
`jstackframes`	*scalar*		Number of default stack frames `jstack()`

TABLE 3–1 DTrace Consumer Options (Continued)

Option Name	Value	dtrace Alias	Description
jstackstrsize	*scalar*		Default string space size for jstack()
nspec	*scalar*		Number of speculations
quiet	—	-q	Output only explicitly traced data
specsize	*size*		Speculation buffer size
strsize	*size*		String size
stackframes	*scalar*		Number of stack frames
stackindent	*scalar*		Number of whitespace characters to use when indenting stack() and ustack() output
statusrate	*time*		Rate of status checking
switchrate	*time*		Rate of buffer switching
ustackframes	*scalar*		Number of user stack frames

A D script can refer to a set of built in macro variables. These macro variables are defined by the D compiler.

`$[0-9]+`	Macro arguments
`$egid`	Effective group-ID
`$euid`	Effective user-ID
`$gid`	Real group-ID
`$pid`	Process ID
`$pgid`	Process group ID
`$ppid`	Parent process ID
`$projid`	Project ID
`$sid`	Session ID
`$target`	Target process ID
`$taskid`	Task ID
`$uid`	Real user-ID

EXAMPLE 3–3 PID Argument Example

This example passes the PID of a running `vi` process to the `syscalls2.d` D script. The D script terminates when the `vi` command exits.

```
# cat -n syscalls2.d
     1  #!/usr/sbin/dtrace -qs
     2
     3  syscall:::entry
     4  /pid == $1/
     5  {
     6    @[probefunc] = count();
     7  }
     8  syscall::rexit:entry
     9  {
    10    exit(0);
    11  }

# pgrep vi
2208
# ./syscalls2.d 2208

  rexit                                                  1
  setpgrp                                                1
  creat                                                  1
  getpid                                                 1
  open                                                   1
  lstat64                                                1
  stat64                                                 1
  fdsync                                                 1
  unlink                                                 1
  close                                                  1
  alarm                                                  1
  lseek                                                  1
  sigaction                                              1
  ioctl                                                  1
  read                                                   1
  write                                                  1
```

DTrace Built-in Variables

The following list includes all of the built-in variables for the DTrace framework.

`int64_t arg0, ..., arg9`	The first ten input arguments to a probe represented as raw 64-bit integers. If fewer than ten arguments are passed to the current probe, the remaining variables return zero.
`args[]`	The typed arguments to the current probe, if any. The `args[]` array is accessed using an integer index, but each element is defined to be the type corresponding to the given probe argument. For example, if the `args[]` array is referenced by a `read`(2) system call probe, `args[0]` is of type `int`, `args[1]` is of type `void *`, and `args[2]` is of type `size_t`.
`uintptr_t caller`	The program counter location of the current thread just before entering the current probe.
`chipid_t chip`	The CPU chip identifier for the current physical chip. See Chapter 26, "sched Provider," in *Solaris Dynamic Tracing Guide* for more information.
`processorid_t cpu`	The CPU identifier for the current CPU. See Chapter 26, "sched Provider," in *Solaris Dynamic Tracing Guide* for more information.
`cpuinfo_t *curcpu`	The CPU information for the current CPU. See Chapter 26, "sched Provider," in *Solaris Dynamic Tracing Guide* for more information.
`lwpsinfo_t *curlwpsinfo`	The lightweight process (LWP) state of the LWP associated with the current thread. This structure is described in further detail in the `proc`(4) man page.
`psinfo_t *curpsinfo`	The process state of the process associated with the current thread. This structure is described in further detail in the This structure is described in further detail in the `proc`(4) man page.
`kthread_t *curthread`	The address of the operating system kernel's internal data structure for the current thread, the `kthread_t`. The `kthread_t` is defined in <sys/thread.h>. Refer to *Solaris Internals* for more information on this variable and other operating system data structures.
`string cwd`	The name of the current working directory of the process associated with the current thread.
`uint_t epid`	The enabled probe ID (EPID) for the current probe. This integer uniquely identifiers a particular probe that is enabled with a specific predicate and set of actions.
`int errno`	The error value returned by the last system call executed by this thread.
`string execname`	The name that was passed to `exec`(2) to execute the current process.
`gid_t gid`	The real group ID of the current process.

`uint_t id`	The probe ID for the current probe. This ID is the system-wide unique identifier for the probe as published by DTrace and listed in the output of `dtrace -l`.
`uint_t ipl`	The interrupt priority level (IPL) on the current CPU at the time that the probe fires. Refer to *Solaris Internals* for more information on interrupt levels and interrupt handling in the Solaris operating system kernel.
`lgrp_id_t lgrp`	The locality group ID for the latency group of which the current CPU is a member. See Chapter 26, "sched Provider," in *Solaris Dynamic Tracing Guide* for more information on CPU management in DTrace. See Chapter 4, "Locality Group APIs," in *Programming Interfaces Guide* for more information about locality groups.
`pid_t pid`	The process ID of the current process.
`pid_t ppid`	The parent process ID of the current process.
`string probefunc`	The function name portion of the current probe's description.
`string probemod`	The module name portion of the current probe's description.
`string probename`	The name portion of the current probe's description.
`string probeprov`	The provider name portion of the current probe's description.
`psetid_t pset`	The processor set ID for the processor set that contains the current CPU. See Chapter 26, "sched Provider," in *Solaris Dynamic Tracing Guide* for more information.
`string root`	The name of the root directory of the process associated with the current thread.
`uint_t stackdepth`	The current thread's stack frame depth at probe firing time.
`id_t tid`	The thread ID of the current thread. For threads that are associated with user processes, this value is equal to the result of a call to `pthread_self`(3C).
`uint64_t timestamp`	The current value of a nanosecond timestamp counter. This counter increments from an arbitrary point in the past and should only be used for relative computations.
`uid_t uid`	The real user ID of the current process.
`uint64_t uregs[]`	The current thread's saved user-mode register values at probe firing time. Use of the `uregs[]` array is discussed in Chapter 33, "User Process Tracing," in *Solaris Dynamic Tracing Guide*.
`uint64_t vtimestamp`	The current value of a nanosecond timestamp counter. The counter is virtualized to the amount of time that the current thread has been running on a CPU. The counter does not include the time that is

	spent in DTrace predicates and actions. This counter increments from an arbitrary point in the past and should only be used for relative time computations.
`uint64_t walltimestamp`	The current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.

CHAPTER 4

Using DTrace

This chapter examines the use of DTrace for common basic tasks, and has information on several different types of tracing.

Performance Monitoring

Several DTrace providers implement probes that correspond to existing performance monitoring tools:

- The `vminfo` provider implements probes that correspond to the `vmstat`(1M) tool
- The `sysinfo` provider implements probes that correspond to the `mpstat`(1M) tool
- The `io` provider implements probes that correspond to the `iostat`(1M) tool
- The `syscall` provider implements probes that correspond to the `truss`(1) tool

You can use the DTrace facility to extract the same information that the bundled tools provide, but with greater flexibility. The DTrace facility provides arbitrary kernel information that is available at the time that the probes fire. The DTrace facility enables you to receive information such as process identification, thread identification, and stack traces.

Examining Performance Problems With The `sysinfo` Provider

The `sysinfo` provider makes available probes that correspond to the `sys` kernel statistics. These statistics provide the input for system monitoring utilities such as `mpstat`. The `sysinfo` provider probes fire immediately before the `sys` named `kstat` increments. The probes that are provided by the `sysinfo` provider are in the following list.

`bawrite` Probe that fires whenever a buffer is about to be asynchronously written out to a device.

`bread`	Probe that fires whenever a buffer is physically read from a device. `bread` fires *after* the buffer has been requested from the device, but *before* blocking pending its completion.
`bwrite`	Probe that fires whenever a buffer is about to be written out to a device, whether synchronously *or* asynchronously.
`cpu_ticks_idle`	Probe that fires when the periodic system clock has made the determination that a CPU is *idle*. Note that this probe fires in the context of the system clock and therefore fires on the CPU running the system clock. The `cpu_t` argument (`arg2`) indicates the CPU that has been deemed idle.
`cpu_ticks_kernel`	Probe that fires when the periodic system clock has made the determination that a CPU is executing in the *kernel*. This probe fires in the context of the system clock and therefore fires on the CPU running the system clock. The `cpu_t` argument (`arg2`) indicates the CPU that has been deemed to be executing in the kernel.
`cpu_ticks_user`	Probe that fires when the periodic system clock has made the determination that a CPU is executing in *user mode*. This probe fires in the context of the system clock and therefore fires on the CPU running the system clock. The `cpu_t` argument (`arg2`) indicates the CPU that has been deemed to be running in user-mode.
`cpu_ticks_wait`	Probe that fires when the periodic system clock has made the determination that a CPU is otherwise idle, but some threads are waiting for I/O on the CPU. This probe fires in the context of the system clock and therefore fires on the CPU running the system clock. The `cpu_t` argument (`arg2`) indicates the CPU that has been deemed waiting on I/O.
`idlethread`	Probe that fires whenever a CPU enters the idle loop.
`intrblk`	Probe that fires whenever an interrupt thread blocks.
`inv_swtch`	Probe that fires whenever a running thread is forced to involuntarily give up the CPU.
`lread`	Probe that fires whenever a buffer is logically read from a device.
`lwrite`	Probe that fires whenever a buffer is logically written to a device
`modload`	Probe that fires whenever a kernel module is loaded.
`modunload`	Probe that fires whenever a kernel module is unloaded.
`msg`	Probe that fires whenever a `msgsnd`(2) or `msgrcv`(2) system call is made, but before the message queue operations have been performed.
`mutex_adenters`	Probe that fires whenever an attempt is made to acquire an owned adaptive lock. If this probe fires, one of the `lockstat` provider's `adaptive-block` or `adaptive-spin` probes also fires.
`namei`	Probe that fires whenever a name lookup is attempted in the filesystem.

`nthreads`	Probe that fires whenever a thread is created.
`phread`	Probe that fires whenever a raw I/O read is about to be performed.
`phwrite`	Probe that fires whenever a raw I/O write is about to be performed.
`procovf`	Probe that fires whenever a new process cannot be created because the system is out of process table entries.
`pswitch`	Probe that fires whenever a CPU switches from executing one thread to executing another.
`readch`	Probe that fires after each successful read, but before control is returned to the thread that is performing the read. A read can occur through the `read`(2), `readv`(2) or `pread`(2) system calls. `arg0` contains the number of bytes that were successfully read.
`rw_rdfails`	Probe that fires whenever an attempt is made to read-lock a reader or writer when the lock is held by a writer or desired by a writer. If this probe fires, the `lockstat` provider's `rw-block` probe also fires.
`rw_wrfails`	Probe that fires whenever an attempt is made to write-lock a reader or writer lock when the lock is held by readers or by another writer. If this probe fires, the `lockstat` provider's `rw-block` probe also fires.
`sema`	Probe that fires whenever a `semop`(2) system call is made, but before any semaphore operations have been performed.
`sysexec`	Probe that fires whenever an `exec`(2) system call is made.
`sysfork`	Probe that fires whenever a `fork`(2) system call is made.
`sysread`	Probe that fires whenever a `read`, `readv`, or `pread` system call is made.
`sysvfork`	Probe that fires whenever a `vfork`(2) system call is made.
`syswrite`	Probe that fires whenever a `write`(2), `writev`(2), or `pwrite`(2) system call is made.
`trap`	Probe that fires whenever a processor trap occurs. Note that some processors, in particular UltraSPARC variants, handle some lightweight traps through a mechanism that does not cause this probe to fire.
`ufsdirblk`	Probe that fires whenever a directory block is read from the UFS file system. See `ufs`(7FS) for details on UFS.
`ufsiget`	Probe that fires whenever an inode is retrieved. See `ufs`(7FS) for details on UFS.
`ufsinopage`	Probe that fires after an in-core inode *without* any associated data pages has been made available for reuse. See `ufs`(7FS) for details on UFS.

`ufsipage`	Probe that fires after an in-core inode *with* associated data pages has been made available for reuse. This probe fires after the associated data pages have been flushed to disk. See `ufs`(7FS) for details on UFS.
`wait_ticks_io`	Probe that fires when the periodic system clock has made the determination that a CPU is otherwise idle but some threads are waiting for I/O on the CPU. This probe fires in the context of the system clock and therefore fires on the CPU running the system clock. The `cpu_t` argument (`arg2`) indicates the CPU that is described as waiting for I/O. No semantic difference between `wait_ticks_io` and `cpu_ticks_wait`; `wait_ticks_io` exists solely for historical reasons.
`writech`	Probe that fires after each successful write, but before control is returned to the thread performing the write. A write can occur through the `write`, `writev`, or `pwrite` system calls. `arg0` contains the number of bytes that were successfully written.
`xcalls`	Probe that fires whenever a cross-call is about to be made. A cross-call is the operating system's mechanism for one CPU to request immediate work of another CPU.

EXAMPLE 4–1 Using the `quantize` Aggregation Function With the `sysinfo` Probes

The quantize aggregation function displays a power-of-two frequency distribution bar graph of its argument. The following example uses the `quantize` function to determine the size of the `read` calls that are performed by all processes on the system over a period of ten seconds. The `arg0` argument for the `sysinfo` probes states the amount by which to increment the statistic. This value is `1` for most `sysinfo` probes. Two exceptions are the `readch` and `writech` probes. For these probes, the `arg0` argument is set to the actual number of bytes that are read or are written, respectively.

```
# cat -n read.d
   1  #!/usr/sbin/dtrace -s
   2  sysinfo:::readch
   3  {
   4     @[execname] = quantize(arg0);
   5  }
   6
   7  tick-10sec
   8  {
   9     exit(0);
  10  }

# dtrace -s read.d
dtrace: script 'read.d' matched 5 probes
CPU        ID                    FUNCTION:NAME
  0     36754                      :tick-10sec

  bash
         value  ---------- Distribution ---------- count
```

EXAMPLE 4–1 Using the `quantize` Aggregation Function With the `sysinfo` Probes *(Continued)*

```
           0 |                                         0
           1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 13
           2 |                                         0

  file
       value  ---------- Distribution ---------- count
          -1 |                                         0
           0 |                                         2
           1 |                                         0
           2 |                                         0
           4 |                                         6
           8 |                                         0
          16 |                                         0
          32 |                                         6
          64 |                                         6
         128 |@@                                       16
         256 |@@@@                                     30
         512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            199
        1024 |                                         0
        2048 |                                         0
        4096 |                                         1
        8192 |                                         1
       16384 |                                         0

  grep
       value  ---------- Distribution ---------- count
          -1 |                                         0
           0 |@@@@@@@@@@@@@@@@@@@                      99
           1 |                                         0
           2 |                                         0
           4 |                                         0
           8 |                                         0
          16 |                                         0
          32 |                                         0
          64 |                                         0
         128 |                                         1
         256 |@@@@                                     25
         512 |@@@@                                     23
        1024 |@@@@                                     24
        2048 |@@@@                                     22
        4096 |                                         4
        8192 |                                         3
       16384 |                                         0
```

EXAMPLE 4–2 Finding the Source of Cross-Calls

In this example, consider the following output form the `mpstat`(1M) command:

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 2189   0 1302    14    1  215   12   54   28    0 12995   13  14   0  73
  1 3385   0 1137   218  104  195   13   58   33    0 14486   19  15   0  66
  2 1918   0 1039    12    1  226   15   49   22    0 13251   13  12   0  75
  3 2430   0 1284   220  113  201   10   50   26    0 13926   10  15   0  75
```

The values in the `xcal` and `syscl` columns are atypically high, reflecting a possible drain on system performance. The system is relatively idle and is not spending an unusual amount of time waiting for I/O. The numbers in the `xcal` column are scaled per second and are read from the `xcalls` field of the `sys` kstat. To see which executables are responsible for the cross-calls, enter the following `dtrace` command:

```
# dtrace -n 'xcalls {@[execname] = count()}'
dtrace: description 'xcalls ' matched 3 probes
^C
  find                                                     2
  cut                                                      2
  snmpd                                                    2
  mpstat                                                  22
  sendmail                                               101
  grep                                                   123
  bash                                                   175
  dtrace                                                 435
  sched                                                  784
  xargs                                                22308
  file                                                 89889
#
```

This output indicates that the bulk of the cross calls are originating from `file`(1) and `xargs`(1) processes. You can find these processes with the `pgrep`(1) and `ptree`(1) commands.

```
# pgrep xargs
15973
# ptree 15973
204   /usr/sbin/inetd -s
  5650  in.telnetd
    5653  -sh
      5657  bash
        15970 /bin/sh ./findtxt configuration
          15971 cut -f1 -d:
            15973 xargs file
              16686 file /usr/bin/tbl /usr/bin/troff /usr/bin/ul /usr/bin/vgrind /usr/bin/catman
```

This output indicates that the `xargs` and `file` commands form part of a custom user shell script. To locate this script, you can perform the following commands:

EXAMPLE 4–2 Finding the Source of Cross-Calls *(Continued)*

```
# find / -name findtxt
/usrs1/james/findtxt
# cat /usrs1/james/findtxt
#!/bin/sh
find / -type f | xargs file | grep text | cut -f1 -d: > /tmp/findtxt$$
cat /tmp/findtxt$$ | xargs grep $1
rm /tmp/findtxt$$
#
```

This script runs many process concurrently. A large amount of interprocess communication is happening through pipes. The number of pipes makes the script resource intensive. The script attempts to find every text file on the system and then searches each file for a specific text.

Tracing User Processes

This section focuses on the DTrace facilities that are useful for tracing user process activity and provides examples to illustrate their use.

Using the `copyin()` and `copyinstr()` Subroutines

DTrace probes execute in the Solaris kernel. Probes use the `copyin()` or `copyinstr()` subroutines to copy user process data into the kernel's address space.

Consider the following `write()` system call:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

The following D program illustrates an incorrect attempt to print the contents of a string that is passed to the `write` system call:

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

When you 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 is an address that refers to memory in the process that is executing the system call. Use the `copyinstr()` subroutine to read the string at that address. Record the result with the `printf()` action:

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

The output of this script shows all of the strings that are passed to the `write` system call.

Avoiding Errors

The `copyin()` and `copyinstr()` subroutines cannot read from user addresses which have not yet been touched. A valid address might cause an error if the page that contains that address has not been faulted in by an access attempt. Consider the following example:

```
# dtrace -n syscall::open:entry'{ trace(copyinstr(arg0)); }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 2 (ID 50: syscall::open:entry): invalid address
(0x9af1b) in action #1 at DIF offset 52
```

In the output from the previous example, the application was functioning properly and the address in `arg0` was valid. However, the address in `arg0` referred to a page that the corresponding process had not accessed. To resolve this issue, wait for the kernel or application to use the data before tracing the data. For example, you might wait until the system call returns to apply `copyinstr()`, as shown in the following example:

```
# 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
CPU     ID                    FUNCTION:NAME
  2     51                      open:return   /dev/null
```

Eliminating `dtrace` Interference

If you trace every call to the `write` system call, you will cause a cascade of output. Each call to the `write()` function causes the `dtrace` command to call the `write()` function as it displays the output. This feedback loop is a good example of how the `dtrace` command can interfere with the desired data. You can use a simple predicate to avoid this behavior, as shown in the following example:

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

The `$pid` macro variable expands to the process identifier of the process that enabled the probes. The `pid` variable contains the process identifier of the process whose thread was running on the CPU where the probe was fired. The predicate `/pid != $pid/` ensures that the script does not trace any events related to the running of this script.

`syscall` Provider

The `syscall` provider enables you to trace every system call entry and return. You can use the `prstat`(1M) command to see examine process behavior.

```
$ prstat -m -p 31337
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
 13499 user1     53  44 0.0 0.0 0.0 0.0 2.5 0.0  4K  24  9K   0 mystery/6
```

This example shows that the process is consuming a large amount of system time. One possible explanation for this behavior is that the process is executing a large number of system calls. You can use a simple D program specified on the command line to see which system calls are happening most often:

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }'
dtrace: description 'syscall:::entry' matched 215 probes
^C

  open                                                              1
  lwp_park                                                          2
  times                                                             4
  fcntl                                                             5
  close                                                             6
  sigaction                                                         6
  read                                                             10
  ioctl                                                            14
  sigprocmask                                                     106
  write                                                          1092
```

This report shows a large number of system calls to the `write()` function. 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[arg2] = quantize(); }'
dtrace: description 'syscall::write:entry' matched 1 probe
^C

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   1037
               2 |@                                        3
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@                                        3
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |@                                        5
```

```
            2048 |                                               0
```

The output shows that the process is executing many `write()` system calls with a relatively small amount of data.

The `ustack()` Action

The `ustack()` action traces the user thread's stack. If a process that opens many files occasionally fails in the `open()` system call, you can use the `ustack()` action to discover the code path that executes the failed `open()`:

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}

syscall::open:return
/self->path != NULL && arg1 == -1/
{
    printf("open for '%s' failed", self->path);
    ustack();
}
```

This script also illustrates the use of the `$1` macro variable. This macro variable takes the value of the first operand that is specified on the `dtrace` command line:

```
# dtrace -s ./badopen.d 31337
dtrace: script './badopen.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0     40                      open:return open for '/usr/lib/foo' failed
              libc.so.1`__open+0x4
              libc.so.1`open+0x6c
              420b0
              tcsh`dosource+0xe0
              tcsh`execute+0x978
              tcsh`execute+0xba0
              tcsh`process+0x50c
              tcsh`main+0x1d54
              tcsh`_start+0xdc
```

The `ustack()` action records program counter (PC) values for the stack. The `dtrace` command resolves those PC values to symbol names by looking though the process's symbol tables. The `dtrace` command prints out PC values that cannot be resolved as hexadecimal integers.

When a process exits or is killed before the `ustack()` data is formatted for output, the `dtrace` command might be unable to convert the PC values in the stack trace to symbol names. In that event the dtrace command displays these values as hexadecimal integers. To work around this limitation, specify a process of interest with the -c or -p option to dtrace. If the process ID or command is not known in advance, the following example D program that can be used to work around the limitation. The example uses the open system call probe, but this technique can be used with any script that uses the ustack action.

```
 syscall::open:entry
{
    ustack();
    stop_pids[pid] = 1;
}

syscall::rexit:entry
/stop_pids[pid] != 0/
{
    printf("stopping pid %d", pid);
    stop();
    stop_pids[pid] = 0;
}
```

The previous script stops a process just before the process exits, if the ustack() action has been applied to a thread in that process. This technique ensures that the dtrace command can resolve the PC values to symbolic names. The value of stop_pids[pid] is set to 0 after clearing the dynamic variable.

The pid Provider

The pid provider enables you to trace any instruction in a process. Unlike most other providers, pid probes are created on demand, based on the probe descriptions found in your D programs.

User Function Boundary Tracing

The simplest mode of operation for the pid provider is as the user space analogue to the fbt provider. The following example program traces all function entries and returns that are made from a single function. The $1 macro variable expands to the first operand on the command line. This macro variable is the process ID for the process to trace. The $2 macro variable expands to the second operand on the command line. This macro variable is the name of the function that all function calls are traced from.

EXAMPLE 4–3 userfunc.d: Trace User Function Entry and Return

```
pid$1::$2:entry
{
    self->trace = 1;
}
```

EXAMPLE 4–3 `userfunc.d`: Trace User Function Entry and Return *(Continued)*

```
pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

This script produces output that is similar to the following example:

```
# ./userfunc.d 15032 execute
dtrace: script './userfunc.d' matched 11594 probes
  0  -> execute
  0    -> execute
  0      -> Dfix
  0      <- Dfix
  0      -> s_strsave
  0        -> malloc
  0        <- malloc
  0      <- s_strsave
  0      -> set
  0        -> malloc
  0        <- malloc
  0      <- set
  0      -> set1
  0        -> tglob
  0        <- tglob
  0      <- set1
  0      -> setq
  0        -> s_strcmp
  0        <- s_strcmp
...
```

The `pid` provider can only be used on processes that are already running. You can use the `$target` macro variable and the `dtrace` options `-c` and `-p` to create and instrument processes of interest using the `dtrace` facility. The following D script determines the distribution of function calls that are made to `libc` by a particular subject process:

```
pid$target:libc.so::entry
{
```

```
    @[probefunc] = count();
}
```

To determine the distribution of such calls made by the `date`(1) command, execute the following command:

```
# dtrace -s libc.d -c date
dtrace: script 'libc.d' matched 2476 probes
Fri Jul 30 14:08:54 PDT 2004
dtrace: pid 109196 has exited

  pthread_rwlock_unlock                                              1
  _fflush_u                                                          1
  rwlock_lock                                                        1
  rw_write_held                                                      1
  strftime                                                           1
  _close                                                             1
  _read                                                              1
  __open                                                             1
  _open                                                              1
  strstr                                                             1
  load_zoneinfo                                                      1

...
  _ti_bind_guard                                                    47
  _ti_bind_clear                                                    94
```

Tracing Arbitrary Instructions

You can use the `pid` provider to trace any instruction in any user function. Upon demand, the `pid` provider creates a probe for every instruction in a function. The name of each probe is the offset of its corresponding instruction in the function expressed as a hexadecimal integer. To enable a probe that is associated with the instruction at offset `0x1c` in function `foo` of module `bar.so` in the process with PID 123, use the following command.

```
# dtrace -n pid123:bar.so:foo:1c
```

To enable all of the probes in the function `foo`, including the probe for each instruction, you can use the command:

```
# dtrace -n pid123:bar.so:foo:
```

The following example demonstrates how to combine the `pid` provider with speculative tracing to trace every instruction in a function.

EXAMPLE 4–4 `errorpath.d`: Trace User Function Call Error Path

```
pid$1::$2:entry
{
```

EXAMPLE 4–4 `errorpath.d`: Trace User Function Call Error Path *(Continued)*

```
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
    self->spec = 0;
}

pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

When `errorpath.d` executes, the output of the script is similar to the following example.

```
# ./errorpath.d 100461 _chdir
dtrace: script './errorpath.d' matched 19 probes
CPU     ID                    FUNCTION:NAME
  0  25253                    _chdir:entry 81e08 6d140 ffbfcb20 656c73 0
  0  25253                    _chdir:entry
  0  25269                        _chdir:0
  0  25270                        _chdir:4
  0  25271                        _chdir:8
  0  25272                        _chdir:c
  0  25273                       _chdir:10
  0  25274                       _chdir:14
  0  25275                       _chdir:18
  0  25276                       _chdir:1c
  0  25277                       _chdir:20
  0  25278                       _chdir:24
  0  25279                       _chdir:28
  0  25280                       _chdir:2c
  0  25268                   _chdir:return
```

Anonymous Tracing

This section describes tracing that is not associated with any DTrace consumer. Anonymous tracing is used in situations when no DTrace consumer processes can run. Only the super user may create an anonymous enabling. Only one anonymous enabling can exist at any time.

Anonymous Enablings

To create an anonymous enabling, use the `-A` option with a `dtrace` command invocation that specifies the desired probes, predicates, actions and options. The `dtrace` command adds a series of driver properties that represent your request to the configuration file for the `dtrace`(7D) driver. The configuration file is typically `/kernel/drv/dtrace.conf`. The `dtrace` driver reads these properties when the driver is loaded. The driver enables the specified probes with the specified actions and creates an *anonymous state* to associate with the new enabling. The `dtrace` driver is normally loaded on demand, along with any drivers that act as `dtrace` providers. To allow tracing during boot, the `dtrace` driver must be loaded as early as possible. The `dtrace` command adds the necessary `forceload` statements to `/etc/system` (see `system`(4) for each required `dtrace` provider and for the `dtrace` driver.

When the system boots, the `dtrace` driver sends a message indicating that the configuration file has been successfully processed. An anonymous enabling can set any of the options that are available during normal use of the `dtrace` command.

To remove an anonymous enabling, specify the `-A` option to the `dtrace` command without any probe descriptions.

Claiming Anonymous State

When the machine has completely booted, you can claim an existing anonymous state by specifying the `-a` option with the `dtrace` command. By default, the `-a` option claims the anonymous state and processes the existing data, then continues to run. To consume the anonymous state and exit, add the `-e` option.

When the anonymous state has been consumed from the kernel, the anonymous state cannot be replaced. If you attempt to claim an anonymous tracing state that does not exist, the `dtrace` command generates a message that is similar to the following example:

```
dtrace: could not enable tracing: No anonymous tracing state
```

If drops or errors occur, the `dtrace` command generates the appropriate messages when the anonymous state is claimed. The messages for drops and errors are the same for both anonymous and non-anonymous state.

Anonymous Tracing Examples

The following example shows an anonymous DTrace enabling for every probe in the `iprb`(7D) module:

```
# dtrace -A -m iprb
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

After rebooting, the `dtrace` driver prints a message on the console to indicate that the driver is enabling the specified probes:

```
  ...
  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
  Use is subject to license terms.
  NOTICE: enabling probe 0 (:iprb::)
  NOTICE: enabling probe 1 (dtrace:::ERROR)
  configuring IPv4 interfaces: iprb0.
  ...
```

After rebooting the machine, specifying the `-a` option with the `dtrace` command consumes the anonymous state:

```
# dtrace -a
  CPU     ID                    FUNCTION:NAME
    0  22954                      _init:entry
    0  22955                     _init:return
    0  22800                  iprbprobe:entry
    0  22934          iprb_get_dev_type:entry
    0  22935         iprb_get_dev_type:return
    0  22801                 iprbprobe:return
    0  22802                 iprbattach:entry
    0  22874               iprb_getprop:entry
    0  22875              iprb_getprop:return
    0  22934          iprb_get_dev_type:entry
    0  22935         iprb_get_dev_type:return
    0  22870             iprb_self_test:entry
    0  22871            iprb_self_test:return
    0  22958            iprb_hard_reset:entry
    0  22959           iprb_hard_reset:return
    0  22862        iprb_get_eeprom_size:entry
    0  22826              iprb_shiftout:entry
    0  22828            iprb_raiseclock:entry
    0  22829           iprb_raiseclock:return
  ...
```

The following example focuses only on functions that are called from `iprbattach()`.

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

Run the following commands to clear the previous settings from the driver configuration file, install the new anonymous tracing request, and reboot:

```
# dtrace -AFs iprb.d
dtrace: cleaned up old anonymous enabling in /kernel/drv/dtrace.conf
dtrace: cleaned up forceload directives in /etc/system
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

After rebooting, the `dtrace` driver prints a different message on the console to indicate the slightly different enabling:

```
  ...
  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
  Use is subject to license terms.
  NOTICE: enabling probe 0 (fbt::iprbattach:entry)
  NOTICE: enabling probe 1 (fbt:::)
  NOTICE: enabling probe 2 (fbt::iprbattach:return)
  NOTICE: enabling probe 3 (dtrace:::ERROR)
  configuring IPv4 interfaces: iprb0.
  ...
```

After the machine has finished booting, run the `dtrace` command with the `-a` and the `-e` options to consume the anonymous data and then exit.

```
# dtrace -ae
  CPU FUNCTION
    0  -> iprbattach
    0    -> gld_mac_alloc
    0      -> kmem_zalloc
    0        -> kmem_cache_alloc
    0          -> kmem_cache_alloc_debug
    0            -> verify_and_copy_pattern
```

```
  0           <- verify_and_copy_pattern
  0           -> tsc_gethrtime
  0           <- tsc_gethrtime
  0           -> getpcstack
  0           <- getpcstack
  0           -> kmem_log_enter
  0           <- kmem_log_enter
  0         <- kmem_cache_alloc_debug
  0       <- kmem_cache_alloc
  0     <- kmem_zalloc
  0   <- gld_mac_alloc
  0   -> kmem_zalloc
  0     -> kmem_alloc
  0       -> vmem_alloc
  0         -> highbit
  0         <- highbit
  0         -> lowbit
  0         <- lowbit
  0         -> vmem_xalloc
  0           -> highbit
  0           <- highbit
  0           -> lowbit
  0           <- lowbit
  0           -> segkmem_alloc
  0             -> segkmem_xalloc
  0               -> vmem_alloc
  0                 -> highbit
  0                 <- highbit
  0                 -> lowbit
  0                 <- lowbit
  0                 -> vmem_seg_alloc
  0                   -> highbit
  0                   <- highbit
  0                   -> highbit
  0                   <- highbit
  0                   -> vmem_seg_create
...
```

Speculative Tracing

This section discusses the DTrace facility for *speculative tracing*. Speculative tracing is the ability to tentatively trace data and decide whether to *commit* the data to a tracing buffer or *discard* it. The primary mechanism to filter out uninteresting events is the *predicate* mechanism. Predicates are useful when you know at the time that a probe fires whether or not the probe event is of interest. Predicates are not well suited to dealing with situations where you do not know if a given probe event is of interest or not until after the probe fires.

If a system call is occasionally failing with a common error code, you might want to examine the code path that leads to the error condition. You can use the speculative tracing facility to tentatively trace data at one or more probe locations, then decide to commit the data to the principal buffer at another probe location. The resulting trace data contains only the output of interest and requires no postprocessing.

Speculation Interfaces

The following table describes the DTrace speculation functions.

TABLE 4–1 DTrace Speculation Functions

Function Name	Arguments	Description
`speculation`	None	Returns an identifier for a new speculative buffer
`speculate`	ID	Denotes that the remainder of the clause should be traced to the speculative buffer specified by ID
`commit`	ID	Commits the speculative buffer that is associated with ID
`discard`	ID	Discards the speculative buffer associated with ID

Creating a Speculation

The `speculation()` function allocates a speculative buffer and returns a speculation identifier. Use the speculation identifier in subsequent calls to the `speculate()` function. A speculation identifier of zero is always invalid, but can be passed to `speculate()`, `commit()` or `discard()`. If a call to `speculation()` fails, the `dtrace` command generates a message that is similar to the following example.

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

Using a Speculation

To use a speculation, use a clause to pass an identifier that has been returned from `speculation()` to the `speculate()` function before any data-recording actions. All data-recording actions in a clause that contains a `speculate()` are speculatively traced. The D compiler generates a compile-time error if a call to `speculate()` follows data recording actions in a D probe clause. Clauses can contain either speculative tracing requests 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 that contains a `speculate()` results in a compile-time error. A `speculate()` function may not follow a previous `speculate()` function. Only one speculation is permitted per clause. A clause that contains only a `speculate()` function will speculatively trace the default action, which is defined to trace only the enabled probe ID.

The typical use of the `speculation()` function is to assign the result of the `speculation()` function to a thread-local variable. That thread-local variable acts as a subsequent predicate to other probes, as well as an argument to `speculate()`.

EXAMPLE 4–5 Typical Use of The `speculation()` Function

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

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

Committing a Speculation

Commit speculations by using the `commit()` function. When you commit a speculative buffer the buffer's data is copied into the principal buffer. If the data in the speculative buffer exceeds the available space in the principal buffer, no data is copied and the drop count for the buffer increments. If the buffer has been speculatively traced on more than one CPU, the speculative data on the committing CPU is copied immediately, while speculative data on other CPUs is copied after the `commit()`.

A speculative buffer that is being committed is not available to subsequent `speculation()` calls until each per-CPU speculative buffer is completely copied into its corresponding per-CPU principal buffer. Subsequent attempts to write the results of a `speculate()` function call to the committing buffer discard the data without generating an error. Subsequent calls to `commit()` or `discard()` also fail without generating an error. A clause that contains a `commit()` function cannot contain a data recording action, but a clause can contain multiple `commit()` calls to commit disjoint buffers.

Discarding a Speculation

Discard speculations by using the `discard()` function. If the speculation has only been active on the CPU that is calling the `discard()` function, the buffer is immediately available for subsequent calls to the `speculation()` function. If the speculation has been active on more than one CPU, the discarded buffer will be available for subsequent calls to the `speculation()` function after the call to `discard()`. If no speculative buffers are available at the time that the `speculation()` function is called a `dtrace` message that is similar to the following example is generated:

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

Speculation Example

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

EXAMPLE 4–6 `specopen.d`: Code Flow for Failed `open()`

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

syscall::open:entry,
syscall::open64:entry
{
    /*
     * The call to speculation() creates a new speculation.  If this fails,
     * dtrace(1M) will generate an error message indicating the reason for
     * the failed speculation(), but subsequent speculative tracing will be
     * silently discarded.
     */
    self->spec = speculation();
    speculate(self->spec);

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

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

syscall::open:return,
syscall::open64:return
/self->spec/
```

EXAMPLE 4–6 `specopen.d`: Code Flow for Failed `open()` *(Continued)*

```
{
    /*
     * 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,
syscall::open64:return
/self->spec && errno != 0/
{
    /*
     * If errno is non-zero, we want to commit the speculation.
     */
    commit(self->spec);
    self->spec = 0;
}

syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
     */
    discard(self->spec);
    self->spec = 0;
}
```

When you run the previous script, the script generates output that is similar to the following example.

```
# ./specopen.d
dtrace: script './specopen.d' matched 24282 probes
CPU FUNCTION
  1  => open                              /var/ld/ld.config
  1    -> open
  1      -> copen
  1        -> falloc
  1          -> ufalloc
  1            -> fd_find
  1              -> mutex_owned
```

```
  1              <- mutex_owned
  1            <- fd_find
  1            -> fd_reserve
  1              -> mutex_owned
  1              <- mutex_owned
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_reserve
  1          <- ufalloc
  1          -> kmem_cache_alloc
  1            -> kmem_cache_alloc_debug
  1              -> verify_and_copy_pattern
  1              <- verify_and_copy_pattern
  1              -> file_cache_constructor
  1                -> mutex_init
  1                <- mutex_init
  1              <- file_cache_constructor
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1            <- kmem_cache_alloc_debug
  1          <- kmem_cache_alloc
  1          -> crhold
  1          <- crhold
  1        <- falloc
  1        -> vn_openat
  1          -> lookupnameat
  1            -> copyinstr
  1            <- copyinstr
  1            -> lookuppnat
  1              -> lookuppnvp
  1                -> pn_fixslash
  1                <- pn_fixslash
  1                -> pn_getcomponent
  1                <- pn_getcomponent
  1                -> ufs_lookup
  1                  -> dnlc_lookup
  1                    -> bcmp
  1                    <- bcmp
  1                  <- dnlc_lookup
  1                  -> ufs_iaccess
  1                    -> crgetuid
  1                    <- crgetuid
  1                    -> groupmember
  1                      -> supgroupmember
```

```
1                      <- supgroupmember
1                    <- groupmember
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                  -> vn_rele
1                  <- vn_rele
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1              <- lookuppnvp
1            <- lookuppnat
1          <- lookupnameat
1        <- vn_openat
1        -> setf
1          -> fd_reserve
1            -> mutex_owned
1            <- mutex_owned
1            -> mutex_owned
1            <- mutex_owned
1          <- fd_reserve
1          -> cv_broadcast
```

```
  1          <- cv_broadcast
  1        <- setf
  1        -> unfalloc
  1          -> mutex_owned
  1          <- mutex_owned
  1          -> crfree
  1          <- crfree
  1          -> kmem_cache_free
  1            -> kmem_cache_free_debug
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1              -> file_cache_destructor
  1                -> mutex_destroy
  1                <- mutex_destroy
  1              <- file_cache_destructor
  1              -> copy_pattern
  1              <- copy_pattern
  1            <- kmem_cache_free_debug
  1          <- kmem_cache_free
  1        <- unfalloc
  1        -> set_errno
  1        <- set_errno
  1      <- copen
  1    <- open
  1  <= open                                            2
```

Index