DTrace Quick Start Guide

Observing Native and Web Applications in Production

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

Part No: 821–0909–10
October 2009

Contents

Preface

The *DTrace Quick Start Guide* shows you how to use DTrace to collect information about your system or application that you can use to improve the performance of your system or application. DTrace is especially well suited for collecting data about applications running on a live system under a real workload. This guide provides example DTrace scripts, including examples that examine an AMP stack and Web 2.0 applications.

DTrace Resources

For more information see the following resources.

- Complete reference information about DTrace: *Solaris Dynamic Tracing Guide* at `http://wikis.sun.com/display/DTrace/Documentation`
- DTrace community and discussion forum on the OpenSolaris™ web site: `http://opensolaris.org/os/community/dtrace/`
- More example D scripts: on your Solaris™ operating system in `/usr/demo/dtrace/`
- Information about observability: Observability community on the OpenSolaris web site at `http://www.opensolaris.org/os/community/observability/` and Solaris observability feature on the Sun™ web site at `http://www.sun.com/software/solaris/observability.jsp`

- Articles on SDN (the Sun Developer Network): "DTrace Quick Reference Guide" at `http://developers.sun.com/solaris/articles/dtrace_quickref/dtrace_quickref.html` and "Tutorial: DTrace by Example" at `http://developers.sun.com/solaris/articles/dtrace_tutorial.html`
- DTrace BigAdmin System Administration Portal at `http://www.sun.com/bigadmin/content/dtrace/`
- Presentations and screencasts: Search for DTrace on the Sun Learning eXchange at `http://slx.sun.com/`
- DTrace Hands on Lab: A step by step guide to learning DTrace at `http://developers.sun.com/learning/javaoneonline/j1lab.jsp?lab=LAB-9400[amp]yr=2008`
- Training: Search for DTrace on the Sun training site at `http://www.sun.com/training/`

CHAPTER 1

Introduction to DTrace

The beginning of this chapter provides enough information to enable you to get useful information using DTrace. The last two sections of this chapter describe using DTrace from both application development and system administration perspectives.

Overview

DTrace is a comprehensive dynamic tracing facility that is built into the Solaris OS and can be used by administrators and developers to examine the behavior of both user programs and of the operating system itself. With DTrace you can explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. DTrace is safe to use on production systems and does not require restarting either the system or applications.

DTrace dynamically modifies the operating system kernel and user processes to record data at locations of interest, called *probes*. A probe is a location or activity to which DTrace can bind a request to perform a set of actions, such as record a stack trace, a timestamp, or the argument to a function. Probes are like programmable sensors in your Solaris system. DTrace probes come from a set of kernel modules called *providers*, each of which performs a particular kind of instrumentation to create probes.

You can access DTrace capabilities by using the `dtrace`(1M) command or by using DTrace scripts. You must be the super user on the system to execute the `dtrace` command or DTrace scripts, or you must have appropriate DTrace privileges.

DTrace includes a new scripting language called D that is designed specifically for dynamic tracing. With D, you can easily write scripts that dynamically turn on probes and collect and process the information. By sharing D scripts, you can easily share your knowledge and troubleshooting methods with others. See the resources in the Preface of this guide for sources of example D scripts.

Introduction to D Scripts

DTrace works on an event/callback model: You register for an event and implement a callback that is executed when the event occurs. Examples of events include executing a particular function, accessing a particular file, executing an SQL statement, garbage collecting in Java™. Examples of data collected in the callback include function arguments, time taken to execute a function, SQL statements. In DTrace the event definition is called a *probe*, the occurrence of the event is called *probe firing*, and the callback is called an *action*. You can use a *predicate* to limit the probes that fire. Predicates are statements that evaluate to a boolean value. The action executes only when the predicate evaluates to true.

A D script consists of a probe description, a predicate, and actions as shown below:

```
probe description
/predicate/
{
        actions
}
```

When the D script is executed, the probes described in the probe description are enabled. The action statements are executed when the probe fires (the probe event occurs) *and* the predicate evaluates to true. Consider the following simple D script:

```
syscall::write:entry
/execname == "bash"/
{
        printf("bash with pid %d called write \
                system call\n",pid);
}
```

In this D script, the probe description is the `syscall::write:entry` which describes the `write` system call. The predicate is `/execname == "bash"/`. This script checks whether the executable that is calling the `write` system call is the `bash` shell. The `printf` statement is the action that executes every time `bash` calls the `write` system call.

Predicates and action statements are optional.

- If the predicate is missing, then the action is always executed.
- If the action is missing, then the name of the probe that fired is printed.

In the following simple D script the predicate is missing, so the action is always executed. This script prints all the processes successfully started in the system.

```
proc:::exec-success
{
        trace(execname)
}
```

The following D script is more complex. This script shows all the files that are opened in the system.

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

Probe Description

The probe consists of four fields separated by colon characters:

provider:*module*:*function*:*name*

provider	Required. Specifies the layer that is instrumented. For example, the `syscall` provider is used to monitor system calls while the `io` provider is used to monitor the disk I/O.
module	Optional. Describes the module that is instrumented.
function	Optional. Describes the function that is instrumented.
name	Optional. Typically represents the location in the function. For example, use `entry` to instrument when you enter the function.

Wild card characters such as `*` and `?` can be used. Leaving a field blank is equivalent to using the `*` wild card. The following table shows a few examples.

TABLE 1–1 Examples of Probe Descriptions

Probe Description	Explanation
`syscall::open:entry`	Entry into the `open()` system call
`syscall::open*:entry`	Entry into any system call that starts with "open" such as `open()` and `open64()`
`syscall:::entry`	Entry into any system call
`syscall:::`	All probes published by the `syscall` provider
`pid1234:libc:malloc:entry`	Entry into the `malloc()` routine in the `libc` library in pid 1234
`pid1234::*open*:entry`	Entry into any function in pid 1234 that has "open" in its name
`pid1234:::entry`	Entry into any function in pid 1234, including the main executable and any library

Predicate

The predicate is any D expression. You can construct predicates using the many built-in variables or arguments provided by the probe. The following table shows a few examples. The action is executed only when the predicate evaluates to true.

TABLE 1–2 Examples of Predicates

Predicate	Explanation
`cpu == 0`	True if the probe executes on `cpu0`
`pid == 1029`	True if the pid of the process that caused the probe to fire is 1029
`uid == 123`	True if the probe is fired by a process owned by userid 123
`execname == "mysql"`	True if the probe is fired by the `mysql` process
`execname != "sched"`	True if the process is not the scheduler (`sched`)
`ppid !=0 && arg0 == 0`	True if the parent process id is not 0 and the first argument is 0

Action

The action section is a series of action commands separated by semicolon characters (;). The following table shows a few examples. The action is executed only when the predicate evaluates to true.

TABLE 1–3 Examples of Actions

Action	Explanation
`printf()`	Print something using C-style `printf()` command
`ustack()`	Print the user level stack
`trace()`	Print the given variable

Developing a D Script

The following `dtrace` command prints all the functions that process id 1234 calls. The `-n` option specifies a probe name to trace.

```
# dtrace -n pid1234:::entry
```

The following example shows how to use the above `dtrace` command in a script. The `-s` option means compile the D program source file.

```
#!/usr/sbin/dtrace -s
/* The above line means that the script that follows
   needs to be interpreted using dtrace.
   D uses C-style comments.
*/
pid1234:::entry
{}
```

Replace the 1234 in the above command or script with the id of a process that you own that you want to examine. Remember to make your script file executable. Use Ctrl-C to return to the shell prompt.

You should see output similar to the following:

```
# dtrace -n pid23:::entry
dtrace: description 'pid23:::entry' matched 7954 probes
CPU     ID                    FUNCTION:NAME
  1 105673          set_parking_flag:entry
  1 105912                queue_lock:entry
^C
#
```

Note – On systems other than Solaris systems, you must be the super user on the system to execute the `dtrace` command or D scripts. On Solaris systems, you must be the super user on the system to use DTrace, or you must have appropriate DTrace privileges. You must be the root user to profile or trace processes owned by root. To grant DTrace privileges, become the super user and use the `usermod`(1M) command for the specified non-root user or edit the `/etc/user_attr` file directly. You should only need `dtrace_user` and `dtrace_proc` privileges.

Using Script Parameters

To make the above script easier to reuse, modify the script to take the process id as a parameter, as shown below.

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

You must provide one and only one argument when you invoke this script. Otherwise, `dtrace` will not execute.

DTrace is dynamic: It shows events as they happen. If you probe a process that is not busy, you see a blank line until a matching event occurs. Try the command with a busy process such as a web browser process.

Using Aggregate Functions

If you specified a busy process, you probably received a very large volume of output. The D language aggregate construct helps you manage this output. Aggregations collect the output in tables in memory and output a summary. Aggregations have the following form:

```
@name[table index(es)] =aggregate_function()
```

The following is an example of an aggregate construct:

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

Add this construct to the action area of your script

```
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        @count_table[probefunc] = count() ;
}
```

When you run the modified script, you see output similar to the following. Most of the data is omitted from this sample output.

```
# ./myDscript.d 23
dtrace: script './myDscript' matched 7954 probes
^C
  __clock_gettime             5
  ioctl                      20
  mutex_lock                767
  sigon                    1554
```

This script collects information into a table until you enter Ctrl-C. When you enter Ctrl-C, DTrace outputs the data that was collected during that time.

The output table lists each function only one time, with the number of times that function was entered, from fewer to greater number of times entered. The variable `probefunc` is a built-in variable that holds the name of the function in each event. The aggregation function `count()` increments the number of times the function was called. Other popular aggregation functions include `average()`, `min()`, `max()`, and `sum()`.

The probes are created dynamically when the script runs. The probes are disabled automatically when the script stops. DTrace also automatically deallocates any memory it allocated.

The following script uses the `probemod,probefunc` table index to collect a table that shows both function name and library name:

```
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        @count_table[probemod,probefunc]=count();
}
```

The library name is in the first column as shown in the following sample:

```
# ./myDscript.d 23
dtrace: script './myDscript' matched 7954 probes
^C
  libc.so.1            __clock_gettime            5
  libc.so.1            ioctl                     20
  libc.so.1            mutex_lock               767
  libc.so.1            sigon                   1554
```

Calculating Time Spent in Each Function

To determine how much time is spent in each function, use the DTrace built-in variable `timestamp`. Create probes in the entry and return of each function and then calculate the difference between the two `timestamp` values. The `timestamp` variable reports time in nanoseconds from epoch.

```
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        ts[probefunc] = timestamp;
}
pid$1:::return
{
        @func_time[probefunc] = sum(timestamp \
                - ts[probefunc]);
```

```
        ts[probefunc] = 0;
}
```

Note that the `ts[]` is an array, and D has automatically declared and initialized it for you. Setting the value of a variable to 0 instructs `dtrace` to recover the memory of the variable.

The output gives the function name and the total time spent in that function during the time the script ran.

```
# ./myDscript.d 23
dtrace: script './myDscript' matched 15887 probes
^C
  enqueue                                  2952
  mutex_lock                            3151262
  cond_timedwait                 31282360711332
```

Ignoring Functions Already Entered

Notice that the `cond_timedwait` line in the previous example has a very large time number. This is because the `cond_timedwait()` function was already executing when the D script was started. To avoid this condition, add the following predicate to the return probe section. This predicate ignores a function return if there was no enter for that function.

```
pid$1:::return
/ts[probefunc] != 0/
{
        @func_time[probefunc] = sum(timestamp \
                - ts[probefunc]);
        ts[probefunc] = 0;
}
```

As in C, you can omit the `!= 0` part and just use `/ts[probefunc]/` as the predicate.

Handling Multithreaded Applications

Two threads could execute the same function at the same time and produce a race condition. To handle this case you need one copy of the `ts[]` construct for each thread. DTrace addresses this with the `self` variable. Anything that you add to the `self` variable is made thread local.

The following script ignores functions that were already entered and handles multithreaded applications.

```
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        self->ts[probefunc] = timestamp;
}
pid$1:::return
/self->ts[probefunc]/
{
        @func_time[probefunc] = sum(timestamp \
                - self->ts[probefunc]);
        self->ts[probefunc] = 0;
}
```

Minimizing DTrace Performance Impact

The above script can enable tens of thousands of probes or even hundreds of thousands of probes. Even though each probe is light weight, enabling such a large number of probes on your system can impact the performance of your application.

You can limit the number of probes enabled by modifying the probe description. See the following table for some examples.

TABLE 1–4 Limiting Number of Probes Enabled

Probe Description	Explanation
`pid$1:libc::entry`	Limit probes to one library
`pid$1:a.out::entry`	Limit probes to non-library functions
`pid$1:libc:printf:entry`	Limit probes to one function

Collecting Information About an Application

As an alternative to specifying a process id number to be traced, you can specify a command to be traced. When you specify a command to be traced, DTrace collects data on that command process and all of the child processes of that command. DTrace displays the collected data after the specified command exits.

To collect data about a command, use the `-c` option to specify the command on the `dtrace` command line or when you invoke your D script. If you use a D script, use the DTrace `$target` macro inside the script to capture the process id of the argument to the `-c` option on the command line.

The following script counts the number of times `libc` functions are called from a specified command:

```
#!/usr/sbin/dtrace -s
pid$target:libc::entry
{
        @[probefunc]=count();
}
```

Use the `-c` option to specify the target command:

```
# ./myDscript.d -c "man dtrace"
```

In this example, the first output you see is the `dtrace`(1M) man page. When you press the **q** key, you see output such as the following. Most of the output is omitted in this example.

```
dtrace: script './myDscript.d' matched 2914 probes
dtrace: pid 1111 has exited
  ___lwp_private             1
  strncpy                  125
  strlen                  7740
```

DTrace for Developers

The providers discussed in this section collect types of information that often are most interesting to application developers: `syscall`, `proc`, `pid`, `sdt`, `vminfo`. Use these providers to observe running processes as well as process creation and termination, LWP creation and termination, and signal handling. This section focuses on the `pid` provider. See the resources in the Preface of this guide for sources of more examples.

The pid Provider

The `pid` provider instruments the entry and return from any user level function in a running process. With the `pid` provider you can trace any instruction in any process on the system. The name of the `pid` provider includes the process id of the running process that you want to examine. See "Developing a D Script" on page 13 for examples of probe descriptions using the `pid` provider.

The syscall Provider

The `syscall` provider reports information about system calls made. The following script displays the stack trace when a program makes a `write()` system call.

```
#!/usr/sbin/dtrace -s
syscall::write:entry
```

```
{
        @[ustack()]=count();
}
```

The sysinfo Provider

The `sysinfo` provider reports information about kernel statistics that are classified by the name `sys`. These kernel statistics provide the input for system monitoring utilities such as `mpstat`(1M). The following script counts the number of times various processes get to run in the CPU. The `sysinfo:::pswitch` event occurs when a process is switched to run on the CPU. Enter Ctrl-C to display the results from running this script.

```
#!/usr/sbin/dtrace -s
sysinfo:::pswitch
{
        @[execname] = count();
}
```

The proc Provider

The `proc` provider reports information about processes. The following script displays the process name, process id, and user id when a new process is started in the system. The `proc:::exec-success` event occurs when a new process is started successfully. The `-q` option suppresses output such as column headings and number of probes matched.

```
#!/usr/sbin/dtrace -qs
proc:::exec-success
{
        printf("%s(pid=%d) started by uid \
                - %d\n",execname, pid, uid);
}
```

DTrace for System Administrators

System administrators must handle the behavior and misbehavior of applications running on a predetermined environment. Use the following providers to obtain this type of information: `syscall`, `proc`, `io`, `sched`, `sysinfo`, `vminfo`, `lockstat`, and `profile`. Of these providers, `syscall`, `proc`, `io`, and `sched` are the easiest starting points. These providers report information related to processes, threads, stack status, and many other kernel metrics.

The syscall Provider

The `syscall` provider is probably the most important provider to learn and use because system calls are the primary communication channel between user level applications and the kernel. Knowing which system calls are being used and how much time they take helps to establish metrics of system usage and identify possible misbehavior.

The following command reports information when the system enters the `close`(2) system call:

```
# dtrace -n syscall::close:entry
dtrace: description 'syscall::close:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  61791                      close:entry
```

The following script identifies the process that wrote to a particular process:

```
#!/usr/sbin/dtrace -s
syscall::write:entry
{
        trace(pid);
        trace(execname);
}
```

Partial output looks similar to the following:

```
# ./myDscript.d
dtrace: script './myDscript.d' matched 1 probe
CPU     ID        FUNCTION:NAME
  0  61787          write:entry      903  gnome-terminal
  0  61787          write:entry      805  compiz-bin
```

Use the following script to determine how much time your web server is spending at read(2). You can easily modify this script to collect information about other processes. The BEGIN probe is a place to put actions that you want to perform only once at the beginning of the script.

```
#!/usr/sbin/dtrace -qs
BEGIN
{
        printf("size\ttime\n");
}
syscall::read:entry
/execname == "httpd"/
{
        self->start = timestamp;
}
syscall::read:return
/self->start/
{
        printf("%d\t%d\n", arg0, timestamp - self->start);
        self->start = 0;
}
```

BEGIN, END, and ERROR are special probes. These events occur only when the D script starts, ends, or encounters an error. Use the BEGIN clause for actions such as printing the headings of the output or reporting when the script started. You might use the END clause to summarize data. Note that DTrace by default prints all the data it collects and cleans up after itself, so you do not need to use an END clause to perform those kinds of actions.

The proc Provider

The `proc` event occurs when processes, threads, or signals are created or terminated. The `proc` provider can tell you which user sent a given `signal`(3head) to which process.

The following script traces all signals sent to all processes currently running on the system. The `-w` option permits destructive actions. The `pr_fname` variable is the name of `exec`'ed file in the `psinfo_t` structure of the receiving process. For more information about `proc` arguments and the `psinfo_t` structure, see the "proc Provider" section of the *Solaris Dynamic Tracing Guide* at `http://wikis.sun.com/display/DTrace/proc+Provider`.

```
#!/usr/sbin/dtrace -wqs
proc:::signal-send
{
        printf("%d was sent to %s by ", \
                args[2], args[1]->pr_fname);
        system("getent passwd %d | cut -d: -f5", uid);
}
```

The following is sample partial output:

```
# ./myDscript.d
18 was sent to hald-runner Super-User
18 was sent to init by Super-User
18 was sent to ksh93 by Super-User
^C
2 was sent to dtrace by Me
```

Add the conditional statement (`/args[2] == SIGKILL/`) to the script and send `SIGKILL` signals to different processes from different users.

```
#!/usr/sbin/dtrace -wqs
proc:::signal-send
/args[2] == SIGKILL/
```

```
{
        printf("SIGKILL was sent to %s by ", \
                args[1]->pr_fname);
        system("getent passwd %d | cut -d: -f5", uid);
}
```

The sched Provider

The `sched` provider dynamically traces scheduling events. Use it to understand when and why threads sleep, run, change priority, or wake other threads.

The following script determines the amount of time the CPU spends on I/O wait and working. It also breaks down the I/O process and indicates the data that was retrieved during the I/O wait time by StarOffice. You can easily modify this script to fit your particular needs.

```
#!/usr/sbin/dtrace -qs
sched:::on-cpu
/execname == "soffice.bin"/
{
  self->on = vtimestamp;
}
sched:::off-cpu
/self->on/
{
  @time["<on cpu>"] = sum(vtimestamp - self->on);
  self->on = 0;
}
io:::wait-start
/execname == "soffice.bin"/
{
  self->wait = timestamp;
}
io:::wait-done
/self->wait/
{
```

```
  @io[args[2]->fi_name] = sum(timestamp - self->wait);
  @time["<I/O wait>"] = sum(timestamp - self->wait);
  self->wait = 0;
}
END
{
  printf("Time breakdown (milliseconds):\n");
  normalize(@time, 1000000);
  printa("  %-50s %15@d\n", @time);

  printf("\nI/O wait breakdown (milliseconds):\n");
  normalize(@io, 1000000);
  printa("  %-50s %15@d\n", @io);
}
```

The io Provider

The `io` provider examines the disk input and output (I/O) subsystem. With the `io` provider you can get an in-depth understanding of `iostat(1M)` output.

The `io` events occur for all the disk requests, including NFS services, except on metadata requests.

The following example is based on the `/usr/demo/dtrace/iosnoop.d` script. Use this script to trace which files are being accessed on which device and to determine whether the task being performed is a read or a write. This example script is for UFS file systems, not for ZFS file systems.

```
#!/usr/sbin/dtrace -qs
BEGIN
{
        printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}
io:::start
{
        printf("%10s %58s %2s\n", args[1]->dev_statname,
```

```
        args[2]->fi_pathname, args[0]->b_flags & B_READ \
                ? "R" : "W");
}
```

The `fi_pathname` variable is the full path name in the `fileinfo_t` structure. The `dev_statname` variable is the name of the device and instance in the `devinfo_t` structure. For more information about `io` arguments and the `fileinfo_t` and `devinfo_t` structures, see the “io Provider” section of the *Solaris Dynamic Tracing Guide* at http://wikis.sun.com/display/DTrace/io+Provider.

CHAPTER 2

DTrace for Web 2.0 Applications

This chapter first explains why DTrace is the best tool to use to debug and performance tune your Web 2.0 applications. This chapter then shows several examples, including a complete system example, a MySQL™ example, and Drupal examples.

Why Use DTrace?

The deployment stack for a typical Web 2.0 application is becoming increasingly complicated. The stack has JavaScript™ on the browser, multiple layers of API (for example, OpenSocial), an application server, a web server, a database, native code such as C/C++ or Java, and the operating system. A click on a web page can exercise multiple layers. Experts in one area might not be experts in another area, making performance tuning even more difficult.

Each layer listed above has good debugging tools. Database administrators have GUI and command line tools to observe every detail of the database. Language debuggers such as `gdb` do a good job of providing visibility into both scripting and native languages in the stack. Tools such as JProbe, JMeter, and VisualVM provide visibility into Java code. Tools such as `vmstat`, `iostat`, `mpstat`, and `truss` provide visibility into the operating system.

These tools are important in performance tuning and understanding each layer in isolation. However, these tools do not provide insight into the entire system and interaction between the different layers. For example, these tools cannot help find the database queries that are executed when a user clicks a JavaScript button on a browser. Also, some of these tools are not usable in production. For example, `truss` can slow down your application too much to be usable in production.

Developers typically resort to creating custom debug code to address these issues. These pieces of code can be seen in debug enabled versions of applications. These debug versions of applications need special flags and reduce performance of the applications. Therefore, debug enabled applications are not normally used in production.

DTrace addresses this problem with dynamic instrumentation. How do you wish you could debug and performance tune your Web 2.0 applications? What if you could dynamically insert code into a live running application and collect data at the point of instrumentation? What if you could turn these instrumentations on and off dynamically? What if you could use the same instrumentation at every layer of your application? What if this instrumentation supports popular languages such as C, C++, Java, PHP, Python, Ruby, JavaScript?

These capabilities are exactly what DTrace provides for you. DTrace enables you to observe every layer of your application infrastructure. It enables you to collect data for a few seconds (incurring a smaller performance penalty) and turn off data collection dynamically – without restarting applications. You do not need to put new code into your application to use DTrace. DTrace can observe fully performance tuned code – no `-g` option is needed.

Observing Multiple Layers

D scripts can span multiple layers. The following script reports how much time you are spending on the different layers in an AMP stack (Apache/MySQL/PHP). The output is `%s` of time spent in Apache, Java, MySQL, the browser, and the operating system. This script is like a TOP tool for AMP.

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

BEGIN
{
    total=mysqlcnt=httpcnt=phpcnt=javacnt=ffxcnt=othercnt=0;
    printf("%10s %10s %10s %10s %10s %10s\n","% MYSQL","% APACHE",\
        "% FIREFOX","% PHP","% Java","% OTHER");
}

php*:::request-startup
{
    inphp[pid,tid]=1;
}

php*:::request-shutdown
{
    inphp[pid,tid]=0;
}

profile-1001
{
    total++;
    (execname=="mysqld")?mysqlcnt++:\
        (execname=="httpd")?(inphp[pid,tid]==1?phpcnt++:httpcnt++):\
        (execname=="java")?javacnt++:\
        (execname=="firefox-bin")?ffxcnt++:othercnt++;
}

tick-30s
{
    printf("%10s %10s %10s %10s %10s %10s\n","% MYSQL","% APACHE",\
```

```
        "% FIREFOX","% PHP","% Java","% OTHER");
}

tick-2s
{
    printf("%10d %10d %10d %10d %10d %10d\n",mysqlcnt*100/total,\
        httpcnt*100/total,ffxcnt*100/total, phpcnt*100/total,\
        javacnt*100/total,othercnt*100/total);
    total=mysqlcnt=httpcnt=phpcnt=ffxcnt=javacnt=othercnt=0;
}
```

Observing One Application

In the previous example you saw how to use DTrace to observe many layers of the stack at production. The examples in this section show how to use DTrace to observe one particular application. With a little knowledge of the application, you can develop D scripts that provide important information about the application while the application is running.

Observing the MySQL Database

Consider the database MySQL, for example. MySQL is open source, and you can easily discover that the name of the function that is called when a particular SQL statement is executed is `dispatch_command()`. You can also easily determine that the SQL statement is passed as a string in the third argument. With only this knowledge you can write the following very simple D script to print out the SQL that are executed in a live running instance of MySQL.

```
#!/usr/sbin/dtrace -qs
#pragma D option strsize=1024

pid$1::*dispatch_command*:entry
```

```
{
        printf("%d::%s\n",tid,copyinstr(arg2));
}
```

The option `strsize` is used to increase the size of strings in D to handle longer SQL statements.

Observing Drupal in Production

Drupal is the popular extensible open source content management system. Many open source modules are available for Drupal. Some of these modules might take too much time. You want to know how much time each module is taking so that you can quickly isolate the slow module. The following D script shows the amount of time taken by each module.

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

#pragma D option quiet

BEGIN
{
        start_time = timestamp;
        printf("Collecting data, press ^c to end...\n");
}

php*:::function-entry
/arg0/
{
        self->pfunc[arg0]=timestamp;
}

php*:::function-return
/arg0 && self->pfunc[arg0]/
{
@php_times[dirname(copyinstr(arg1))]=sum(timestamp - self->pfunc[arg0]);
@php_calls[dirname(copyinstr(arg1))]=count();
}
```

```
END
{
printf("\n====================================\n");
printf("---Elapsed time (usec): %d\n", (timestamp - start_time) / 1000);
printf("====================================\n\n");
normalize(@php_times, 1000);
printf("%-40.40s %12.12s %12.12s\n", "DIR", "TOTAL USEC", "CALLS");
printa("%-40.40s %@12d %@12d\n", @php_times, @php_calls);
printf("----------------\n");
}
```

The output of this script is similar to the following. This output shows you the total time spent in each module and the number of calls to the module.

```
====================================
---Elapsed time (usec): 9006797
====================================

DIR                                        TOTAL USEC        CALLS
/htfs/htdocs/drupal-6-10                         2049          393
/htfs/htdocs/drupal-6-10/modules/help            6377          784
/htfs/htdocs/drupal-6-10/modules/upload         11231         1177
/htfs/htdocs/drupal-6-10/modules/taxonom        21546         1177
/htfs/htdocs/drupal-6-10/modules/menu           26118         1570
/htfs/htdocs/drupal-6-10/modules/dblog          35474         1567
/htfs/htdocs/drupal-6-10/modules/color          36645         1567
/htfs/htdocs/drupal-6-10/modules/book           41030         1569
/htfs/htdocs/drupal-6-10/modules/aggrega        50758         1176
/htfs/htdocs/drupal-6-10/modules/blog           67835         1957
/htfs/htdocs/drupal-6-10/modules/update        150881         4321
/htfs/htdocs/drupal-6-10/sites/default         233673         4704
/htfs/htdocs/drupal-6-10/modules/comment       301382        10191
/htfs/htdocs/drupal-6-10/modules/blogapi       367443         1177
/htfs/htdocs/drupal-6-10/themes/garland        639091         4301
/htfs/htdocs/drupal-6-10/themes/engines/       940378         1568
/htfs/htdocs/drupal-6-10/modules/filter       2198530        28607
/htfs/htdocs/drupal-6-10/modules/system       3582767         7060
```

```
/htfs/htdocs/drupal-6-10/modules/node          9758935      32487
/htfs/htdocs/drupal-6-10/modules/block        17829797      11735
/htfs/htdocs/drupal-6-10/modules/user         20965812      41833
/htfs/htdocs/drupal-6-10/includes            431530085    1353555
```

The following D script more closely observes a particular module. In Drupal, each module has hooks. The following D script shows you how much time is spent on each hook of a given module.

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

#pragma D option quiet

BEGIN
{
        start_time = timestamp;
        printf("Collecting data, press ^c to end...\n");
}

php*:::function-entry
/arg0 && dirname(copyinstr(arg1))==$$1/
{
        self->pfunc[arg0]=timestamp;
}

php*:::function-return
/arg0 && self->pfunc[arg0]/
{
@php_times[copyinstr(arg0)]=sum(timestamp - self->pfunc[arg0]);
@php_calls[copyinstr(arg0)]=count();
}

END
{
printf("\n====================================\n");
printf("---Elapsed time (usec): %d\n", (timestamp - start_time) / 1000);
printf("====================================\n\n");
normalize(@php_times, 1000);
```

```
printf("%-40.40s %12.12s %12.12s\n", "HOOK", "TOTAL USEC", "CALLS");
printa("%-40.40s %@12d %@12d\n", @php_times, @php_calls);
printf("----------------\n");
}
```

The output of this script is similar to the following.

```
# ./php_hook_times.d /htfs/htdocs/drupal-6-10/modules/user

====================================
---Elapsed time (usec): 12009356
====================================

DIR                                        TOTAL USEC        CALLS
user_login_default_validators                    3840          678
user_uid_optional_to_arg                         4140          679
user_view_access                                 5371          680
user_user                                        6449          680
user_help                                        7184          680
user_elements                                    8442          678
user_is_logged_in                               16664          679
user_init                                       36201          678
user_module_invoke                             367110          680
user_access                                   1386425        23098
user_login_block                              1413852          678
user_load                                     1745926          679
user_uid_optional_load                        1761082          679
user_block                                   20279110         1356
is_null                                     167035670        35306
array_flip                                  177718668         4054
array_keys                                  533607874        12842
is_numeric                                 1482022968        18166
explode                                    1998298196        30326
implode                                    3231829375        40369
main                                       3668568438        28093
define                                    11510486049        82838
function_exists                           20233875771       215888
----------------
```

These examples are just some of what DTrace can do for you. You can find many more uses for DTrace. See the resources listed in the Preface of this book. You might want to subscribe to the DTrace forum on `http://www.opensolaris.org/`.