solaris™

> **Solaris™ 10** How To Guides

# HOW TO USE DTRACE FROM A SOLARIS™ 10 SYSTEM

Videhi Mallela, Solaris Marketing
Angel Camacho, Solaris Marketing
Angelo Rajadurai, Market Development Engineering

Sun microsystems

## About This DTrace How To Guide

The DTrace How to Guide is intended to help a new user learn how to use DTrace for gathering and using system and application information from a Solaris™ 10 system. It covers how to build a D script and identifies the providers that every System Administrator and developer should know when learning and using DTrace. A few examples to help the reader get started using DTrace are also included.

After reading this guide the user will be able to construct scripts to gather useful information about running applications that in turn can be used to improve the performance on Solaris.

See the Solaris Dynamic Tracing Guide at *http://docs.sun.com/app/docs/doc/817-6223* for more details on the D language and DTrace.

## Contents

# DTrace How To Guide

## Introduction to DTrace

DTrace is a comprehensive dynamic tracing facility that is built into Solaris 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. It 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, like recording a stack trace, a timestamp, or the argument to a function. Probes are like programmable sensors scattered all over your Solaris system in interesting places. DTrace probes come from a set of kernel modules called *providers*, each of which performs a particular kind of instrumentation to create probes.

DTrace includes a new scripting language called *D* which is designed specifically for dynamic tracing. With D it is easy to write scripts that dynamically turn on probes, collect the information, and process it. D scripts make it convenient for users to share their knowledge and troubleshooting methods with others. A number of useful D scripts are included in Solaris 10, and more can be found on Sun's BigAdmin site: *sun.com/bigadmin/content/dtrace/* and on the OpenSolaris project site: *opensolaris.org/os/community/dtrace/*.

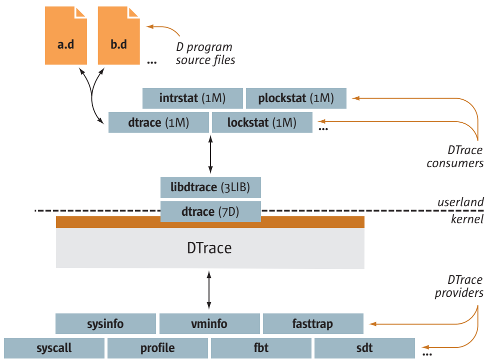

Figure 1 — Overview of the DTrace Architecture and Components.

## Introduction to D Scripts

First, a brief discussion on the construct of a D script. 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 probes fire and if the predicate evaluates to true. Think of probes as events: a probe fires when the event happens. Let's take a simple D script example:

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

Here the probe description is the syscall::write:entry which describes the write system call. The predicate is /execname == “bash”/. This checks to see if 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.

## Probe Description

Now let's look a little deeper. The probe is described using four fields, the provider, module, function, and name.

- *provider*—Specifies the instrumentation method to be used. For example, the syscall provider is used to monitor system calls while the io provider is used to monitor the disk io.
- *module and function*—Describes the module and function you want to observe.
- *name*—Typically represents the location in the function. For example, use entry for name to instrument when you enter the function.

Note that wild cards like * and ? can be used. Blank fields are interpreted as wildcards. Table 1 shows a few examples.

| Probe Description | Explanation |
|---|---|
| syscall::open:entry | entry into open system call |
| syscall::open*:entry | entry into any system call that starts with open (open and open64) |
| syscall:::entry | entry into any system called |
| syscall::: | all probes published by the system call provider |

Table 1—Examples of Probe Descriptions.

## Predicate

A predicate can be any D expression. Table 2 illustrates a few examples. The action is executed only when the predicate evaluates to true.

| 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 |
| 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 first argument is 0 |

Table 2—Examples of Predicates.

## Action

The action section can contain a series of action commands separated by semi-colons (;). Examples are given in Table 3.

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

Table 3—Examples of Actions.

Note that 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 which fired is printed.

With this minimum information you can develop useful D scripts. The next section of this guide focuses on using DTrace from both application development and system administration perspectives.

## DTrace for Developers

The types of information application developers are particularly interested in can be obtained by using the following providers: syscall, proc, pid, sdt, vminfo. Developers can use these to look at running processes as well as process creation and termination, LWP creation and termination, and signal handling. This section focuses on the pid provider only.

### 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 pid of the running process in which you are interested. A few examples of probe descriptions using the pid provider are given in Table 4.

| Example | Explanation |
|---|---|
| pid2439:libc:malloc:entry | entry into the malloc function in libc for process id 2439 |
| pid1234:a.out:main:return | return from main for process id 1234 |
| pid1234:a.out::entry | entry into any function in 1234 that is main executable |
| pid1234:::entry | entry into any function in any library for process id 1234 |

Table 4—Examples of Pid Provider and Associated Probes.

Here is the command you can run to print all the functions that process id 1234 calls:

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

The same command can be used in a script as follows:

```
#!/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
{}
```

You can use this command or script by replacing the 1234 with the pid of the process in which you are interested.

The following steps can be used to build a D script using the pid provider:

1. Try the above command or script right now. As you see, this is not easily configurable.
2. Modify the script to take the process id as a parameter. Your script will now look like:

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

3. Now you can provide the process id as an argument to your script. Note that DTrace will complain and not execute if you do not provide one and only argument to this script. Remember to add execution rights to your scripts.

   You may have noticed that the output from this script went by too fast to be useful. D language has a wonderful construct called *aggregate* to collect all the detail in memory and print out a summary. Aggregations allow you to collect tables of information in memory. Aggregations have the following construct:

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

   For example:

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

   This aggregation will collect information into a table with name of the function (probefunc is a built-in variable that has the name of the function). The aggregation function count() keeps track of the number of times the function was called. The other popular aggregation functions are average, min, max, and sum.
4. The following example adds aggregates in the script you are developing to see a summary table of user functions:

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

   This script will collect information into a table and will continue to run until you press ^c (Control c). Once you stop the script, DTrace will print out the table of information. Notice that you do not need to write any code to print the table. DTrace automatically does this for you.

   You may have noticed that the probes were created dynamically when running this script. The probes are disabled as soon as the script is stopped. You do not need to do anything special for disabling these probes. In addition, DTrace will automatically clean up any memory it allocated. You do need to write any code for cleanup.

5. This script can easily be modified to collect a table with function and library name by changing the index to probemod,probefunc:

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

6. Next, find how much time is being spent in each function. This can be done through the built in variable timestamp. You can create probes in the entry and return of the functions and calculate the time spent by the difference in timestamp from entry to return. The timestamp variable reports time in nanoseconds. Here is the modified script:

```
#!/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. It is good practice to save space by setting the variable to 0.

7. This script works for most cases; however, there are a few corner-case exceptions:
   *Exception 1*—Monitor function entry and return
   Because you are creating the probes on a live running application it is very likely that you were executing a function when you ran the D script. Therefore, it is possible to see the return for a function for which you did not see an enter. This case is easily handled by adding a predicate **/ts[probefunc] != 0/** to the return probe section. This predicate will allow you to ignore the above corner case.

   *Exception 2*—Multi-threaded applications
   The next corner case is a little more tricky and involves multi-threaded applications. There is a likely race condition where two threads could execute the same function at the same time. In this case you need one copy of the ts[] for each thread. DTrace addresses this via the self variable. Anything that you add to the self variable will be made thread local.

8. The following script has been modified to handle these two corner cases:

```
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        self->ts[probefunc] = timestamp;
}

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

**Note:** /self->ts[probefunc]/ is the same as /self->ts[probefunc] != 0/

For very large applications the above script enables a large number of probes (possibly hundreds of thousands). Even though each probe is fairly light weight, adding that many probes to a live running system will impact the performance of your application.

9. You can limit the number of probes enabled by modifying the probe description. See Table 5 for some examples.

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

Table 5 — Modifying Probe Descriptions.

10. Next, you can see how to monitor a process from the time it starts until it ends. DTrace allows you to do this using the **$target** variable and the **-c** option. This script will count the number of times libc functions are called from a given application.

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

pid$target:libc::entry
{
        @[probefunc]=count();
}
```

11. Save this script as libc_func.d and run it.

```
# libc_func.d -c "cat /etc/hosts"
```

12. You can easily replace "cat /etc/hosts" with the command of interest.

## Other Useful Scripts

Here are a few more sample scripts which you can use to get additional information about your system. Knowing the number of system calls, how deep the stack is, and knowing who is doing what will help you to understand the workload and to identify possible bottlenecks.

1. The following is a script to find the stack trace when the program makes the write system call.
   Note that you need to run this with the **-c** option.

```
#!/usr/sbin/dtrace -s
syscall::write:entry
{
        @[ustack()]=count();
}
```

2. The following script counts the number of times various processes get to run in the CPU.
   Note that the sysinfo:::pswitch probe fires when a process is switched to run on the CPU.
   Remember to press ^c (Control c) after a few minutes.

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

3. The following script prints out the process name, pid and uid when a new process is started in the system.
   Note that proc:::exec-success fires when a new process is started successfully.

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

## DTrace for System Administrators

Among the tasks that every System Administrator faces are those related to the behavior or misbehavior of the applications running on a pre-determined environment. This type of information can be obtained through the following providers: syscall, proc, io, sched, sysinfo, vminfo, lockstat and profile. Of these providers, syscall, proc, io and sched are the easiest starting points. By using these providers, system administrators get information related to processes, threads, stack status, and many other kernel metrics. It is easier to start with the syscall, proc, io, *and* sched providers.

### The syscall Provider

This is probably the most important provider to learn and use because system calls are the main communication channel between user level applications and the kernel. Knowing which system calls are being used among other information establishes metrics of system usage and identifies possible misbehavior. With the syscall provider you can easily identify who executes what and how much time a certain operation takes, helping you to identify the root cause of system misbehavior.

1. To list all the occurrences of the probe when it was fired and give information about the system calls at entry into the system that are performing a close(2) system call, use the following script:

```
# dtrace -n syscall::close:entry
```

2. To start to identify the process which sent a kill(2) signal to a particular process, use the following script:

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

3. To determine how much time your webserver is spending at read(2), use the following script. Note that this script can easily be modified for other processes.

```
#!/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;
}
```

## The proc Provider

This provider fires at processes and thread creation and termination as well as signals. It is a more sophisticated approach to the simple kill probe and will tell you which user sent a given signal (3head) to which process.

1. Trace all the signals sent to all the processes currently running on the system:

```
#!/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);
}
```

2. Add the conditional statement **(/args[2] == SIGKILL/)** into 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);
}
```

Here you can see the introduction of pr_fname, which is part of the structure of psinfo_t of the receiving process. For more information please refer to the Solaris Dynamic Tracing User Guide.

## The sched Provider

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

1. The following script determines the amount of time the CPU spends on I/O wait and working. It also breaks the I/O process and indicates that the data that was retrieved during the I/O wait time by StarOffice. This script can easily be modified to fit your particular situation.

```
#!/usr/bin/dtrace -sq
sched:::on-cpu
/execname == "soffice.bin"/
{       self->on = vtimestamp;
}

sched:::off-cpu
/self->on/
{       @time["<on cpu>"] = sum(vtimestamp - self->on);
        self->on = 0;
}
```

## The io Provider

This provider looks into 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 probes fire for all the disk requests, including NFS services, except on metadata requests.

1. The following example is based on the /usr/demo/dtrace/iosnoop.d script. Use it to trace which files are being accessed on which device and to determine if the task being performed is a read or a write.

```
#!/usr/bin/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");
}
```

## For More Information

The information in this guide is just a brief introduction to DTrace.
For more information on DTrace visit the following websites:
• *http://sun.com/solaris/observability.jsp*
• *http://opensolaris.org/os/community/dtrace*
Join the DTrace discussion group and participate in the ongoing discussion and development of this innovative new facility in Solaris.

For those interested in classroom training, SunU is offering the following courses:

| **Available Courses** | |
|---|---|
| Dynamic Performance Tuning and Troubleshooting With DTrace (3 days) (SA-327-S10) | http://www.sun.com/training/catalog/courses/SA-327-S10.xml |
| DTrace Facility Live Virtual Classroom Training (3 days) (VC-SA-327-S10) | http://www.sun.com/training/catalog/courses/VC-SA-327-S10.xml |

under "Experienced System Administrators."

sun.com/solaris

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA Phone 1-650-960-1300 or 1-800-555-9SUN Web sun.com