The Power of DTrace

Roy Cecil

Sun Microsystems, Inc.

March 2007

Table of Contents

Overview

Dynamic Tracing (DTrace) is a debugging tool introduced in the Solaris™ 10 Operating System to help debug systemic problems that are difficult to diagnose using traditional debugging tools and mechanisms. This tool takes advantage of points of instrumentation in the Solaris OS to present information useful for debugging errors and investigating performance issues in applications running on the Solaris platform.

This article covers the following topics:

- Key Features of DTrace
- How Does DTrace Work?
- About Probes
- Example That Illustrates the Power of DTrace
- Anatomy of a D Program
- D Program Examples
- Aggregation
- Built-In Variables and Thread Local Variables
- DTrace Resources
- Author Profile

Key Features of DTrace

DTrace is designed to offer these features:

- **No production risk.** DTrace is considered the only debugging tool currently available that is safe to use on production systems. By design, DTrace does not allow constructs that can bring down the system through careless programming, such as loops and pointers. The absence of looping prevents users from leaving an unending loop that results in nightmarish system downtime. The absence of pointers prevents DTrace users from affecting memory allocated to kernel or application processes. Though it is possible to cause problems to production systems with DTrace, a user has to do so intentionally. Casual usage cannot interfere with production activities.

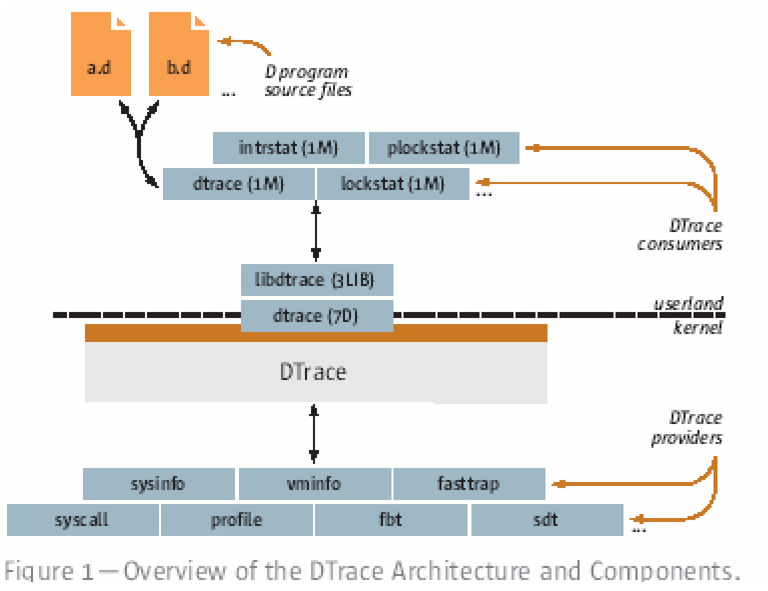
- **No implications.** The Solaris 10 OS includes nearly 40,000 probes that are points of instrumentation in the Solaris kernel. This instrumentation can be turned on and off at will, leaving no overhead when the tracing is turned off. This feature is very important as it helps you troubleshoot live production systems. You can use DTrace to query these probes and develop a picture of how an application is utilizing the kernel.

- **Extensibility.** You can combine queries to create custom probes. A framework is provided to trace user land functions. In addition, you can create custom probes in user applications to provide the ability to use DTrace to obtain information specific to the application during runtime within the semantics of the application.

How Does DTrace Work?

Figure 1 illustrates the DTrace system. The `dtrace`(1M) command uses a library named `libdtrace` as entry points into the various "DTrace providers" within the kernel, each of which gives a logical view of some kernel subsystem. The binary that you can use for this function is named `dtrace`. The binary can be used directly with various command-line options, or it can be used as a shell written in D language just as a developer might use a Korn or Perl script.

Figure 1: Overview of the DTrace Architecture and Components



When executed, D language programs are compiled "on the fly" into byte code that runs on a D byte-code interpreter embedded inside the kernel. The DTrace virtual machine runs the byte code to ensure its safety. If the code is safe and you have sufficient privileges, the code is patched into the kernel dynamically and executed as kernel-level code. This is why probes that are not enabled do not create any overhead.

Inside the kernel, various providers provide a logical view of the kernel subsystems and expose the information reported by DTrace.

About Probes

Probes are points of instrumentation in the Solaris 10 kernel. In short, a probe is a specific point in the kernel's source code. When a program execution passes one of these points, the probe that enabled it is referred to as having fired.

For example, in the `libc` library, in the function that allocates memory (`malloc`), a probe may be defined each time `malloc` is called or whenever `malloc` returns control to the calling routines, which are named `entry` and `return`, respectively. Therefore, if DTrace is being used to watch when `malloc` returns a value, occasions when this happens are said to have fired the `libc:malloc:return` probe.

Tip: *Use the command* `dtrace -l` *to list the available 4-tuples.*

Each probe is uniquely identified by a 4-tuple. This avoids name clashes in the kernel, which is a code base of millions of lines of code. Unique names also help you to more easily recognize the probes that might be of interest. A 4-tuple comprises four components:

- **Provider.** The provider is a logical name used to group probes and can give you a sense of logical subsystems in the kernel. For example, the `fbt` provider puts a probe in each and every function entry and exit point.

- **Module.** The module corresponds to Solaris kernel modules. In the case of custom probes built into applications, the module may be the class or code module where the probe is defined.

- **Function.** The function refers to the function or subprocedure where the probe is defined, regardless of whether the probe is one of the existing kernel probes or a custom probe built into an application.

- **Name.** A name refers to a unique point in the execution of a function. Two ubiquitous names are `entry` and `return`. The `entry` point refers to the first instruction in a function's execution after it is called, and `return` refers to the last point in a function before control goes back to the calling routine. The `entry` probe should make all incoming parameters to the function available, and the `return` probe should make any return values available.

In a DTrace script, a 4-tuple is expressed as follows:

```
provider:module:function:name
```

For example, you can use the `pid` provider to look at the inner workings of user processes. Modules for the `pid` provider are typically the names of libraries a process is using. Functions used with the `pid` provider can be any function in an executable or in the library that a process is using. Finally, the names to use with `pid` are points in the execution of a function where you are interested in observing some useful information. For example, you can define a name `entry` if you are interested in knowing when control passes to the function.

So, to look at the entry information for the `malloc` function found in the `libc` library for a process currently running on a system as PID 456, the 4-tuple to specify in DTrace is:

```
pid456:libc:malloc:entry
```

To examine the return values from the main function in the same process, use:

```
pid456:a.out:main:return
```

Execute the `malloc` probe as follows:

```
# dtrace -n pid456:libc:malloc:entry
```

The output might look like this:

```
dtrace: description 'pid456:libc:malloc:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  61082                    malloc:entry
  0  61082                    malloc:entry
```

__Note:__ DTrace must be used by root unless special privileges are set through Role Based Access Control (RBAC).

An Example to Illustrate the Power of DTrace

DTrace is not a cure for all your problems; rather it is a means to identify the cause of problems. Making that identification requires that you ask the right questions, however. So, DTrace is not a substitute for your knowledge of the system.

If you do not have detailed knowledge of the Solaris OS internals, do not despair. There are many ready-made, useful scripts available on the Web contributed by DTrace enthusiasts. The DTrace Resources section at the end of this guide provides references to script libraries.

This guide illustrates the use of DTrace by example. Everyone who has used a UNIX machine is familiar with the `stat` tools available to monitor system usage. The tool `mpstat` is used to monitor CPU usage. A typical `mpstat` output is as follows:

```
CPU minf mjf xcal  intr ithr csw icsw migr smtx srw syscl usr sys wt idl
0      0  0   524  741  128   3    0    1    0   0     1   4   6  0  90
1      0  0   658  453    6   3    0    1    0   0    21   5   6  0  89
```

Each of the columns supplies valuable information about the system. This data is collected in counters in the operating system and `mpstat` displays the data for the intervals specified. As useful as this data is, it still does not give good insight into which process is responsible for these values.

For example, the `xcal` column in the `mpstat` output shows the number of xcalls, or cross calls, made over an interval of 5 seconds. Cross calls in a multiprocessor system are special interrupts made by one CPU to another CPU when the first CPU wants the second CPU to do some work on its behalf. This process can result is significant performance problems.

Now, assume a production system is experiencing a serious performance degradation yet the test or development system is not. Using `mpstat`, it is determined that the only difference is that the production system is experiencing more cross calls than the development system. It is clear that something is wrong in the production system, but it is not clear which process or processes are responsible for the extra cross calls.

Prior to DTrace, there was no way to tell in any UNIX system which process was the culprit. However, using DTrace, a simple script can make it clear which process is causing the problem.

Create a file named `xcall.d` and enter the following text into the file:

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

sysinfo:::xcalls

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

This script, when executed, uses the `dtrace` binary to query the `sysinfo` provider on all its modules and functions and any specific probe named `xcalls`. The requested output is a count by executable name of all of the values associated with the probes matched.

> ***Note:*** *Depending on your PATH variable setting, the example command execution may not work. Specify any path component necessary to run the script.*

Execute this script by setting the file privileges such that the file `xcall.d` is executable (mode 700, 750, or 755) and run the file at the command line:

```
# ./xcall.d
```

The initial output of the script will indicate how many probes the specified 4-tuple (`sysinfo:::xcalls`) matched.

```
dtrace: description 'sysinfo:::xcalls' matched 2 probes
```

DTrace will begin to execute and gather information on the cross calls being issued on behalf of the processes running on the system. Wait as long as desired and break the DTrace execution with an interrupt (typically by pressing Ctrl-D). DTrace will then report its findings:

```
bash                                       2
cron                                       9
uname                                     24
sched                                    423
dtrace                                  7415
java                                   50345
```

Based on this output, it is clear that the `java` process is causing the cross calls. Since a `java` process can run for a variety of reasons, it is necessary now to use the `pid` provider to investigate each `java` process to determine which specific process is causing the problem. Once that is determined, the affected Java^{TM} applications can be individually debugged to determine which functions are causing the problem. Once you are armed with this information, the Java code can be adjusted or the vendor of the code can be told precisely where to find and fix the problem.

Anatomy of a D Program

To realize the full power of DTrace, you should understand the structure and syntax of D-language scripts. D language is a powerful combination of the syntax of awk and C++. The general structure of a D program is as follows.

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

probe-description_1
/ predicate_1 /
{
        action_1
        action_2
        .
        .
        .
        action_n
}

probe-description_2
/ predicate_2 /
{
        action_1
        action_2
        .
        .
        .
        action_n
}

.
.
.
probe-description_n
/ predicate_n /
{
        action_1
        action_2
        .
        .
        .
        action_n
}
```

The previous pseudo code illustrates how a D program is laid out:

- The first line forces the program to execute using the `dtrace` binary as the command interpreter.
- The D language itself is organized into several clauses of probe descriptions followed by actions requested when the probes are fired.
- Results are captured whenever events occur that match the probe description.
- Finally, you can use predicates to limit the capture of data based on conditions that must be true when a probe event fires.

For example, suppose you want to know the system calls on a machine but only for a process with a PID of 3456. The D-language probe description for this would be as follows:

```
syscall:::entry
/ pid == 3456 /
{
        action_1;
        action_2;
        .
        .
        .
        action_n
}
```

This clause would execute actions 1 through *n* each time a probe named `entry` in any module and a function organized in the `syscall` provider were passed during the execution of the process identified as number 3456. DTrace will not account for the system calls made by other processes in the system because the predicate has the effect of filtering the result set. But DTrace is "smarter" -- for system calls made in other processes, the probes do not fire at all.

As part of a D program, various actions can be performed when a probe fires. The actions are grouped by curly braces ({ }) and separated by semi-colons (;). These actions may aggregate results, print messages, or other perform behaviors that help structure the program output.

Other than this, D language does not use any flow control code. In particular, D language does not have syntactic structures, such as loops (for example, `while` and `for`) or branches (for example, `if`, `switch/case`, and so on).

D Program Examples

Now that you are armed with the knowledge of the D language, here are a few simple D programs.

Hello World

This example displays a simple string as a result of execution.

```
#!/usr/sbin/dtrace -qs
# Hello World! The DTrace way!

BEGIN
{
        printf("Hello World!\n") ;
        exit(0) ;
}

END
{
        printf("Goodbye Cruel World!\n");
}
```

This program uses two special probes called `BEGIN` and `END` that fire whenever a D-script is executed and terminated, respectively. One of the actions, `exit(0)` in the `BEGIN` probe clause, causes the script to terminate, thereby causing the `END` probe to fire.

This program produces the following output:

```
$ ./hello.d

Hello World!

Goodbye Cruel World!
```

All Calls

Next, suppose there is an interest to know the system calls used by bash. The following script will display all the system calls made by bash while the script is executing.

```
#!/usr/sbin/dtrace -qs
#Show me the system calls!

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

Tip: *Leaving an element of the 4-tuple blank is the same as using a wildcard.*

This script uses the `syscall` provider and enables any probe named `entry` for any module or function in the `syscall` provider. The predicate limits the data capture to cases where bash is the executable that causes the probe to fire. This script will generate output as follows:

```
$ ./syscall.d

bash with pid 109 called write
bash with pid 112 called open
.
.
.
```

Aggregation

Formatted strings are printed via the `printf()` function. However, data can be collected and processed before outputting as well. There is no need in D language to declare the type of a variable before using it. In this respect, D language is more like Perl. D language provides integer, floating point, char, and string data types that are all automatically determined at runtime.

In addition, D language provides arithmetic operators, such as + (add), - (subtract), * (multiply), / (divide), and % (mod). Logical operators, such as && (AND), || (OR), and ^^ (XOR), are also available. But the real power of D language is with aggregation. D language treats aggregation like a table, which is very useful when summarizing query results.

Aggregation is a special case with specific syntax. An aggregation is declared in D language as follows, where:

- `name` stands for the name of the aggregation.
- `@` indicates that `name` is really an aggregation (imagine a table).
- `aggfunc` is an aggregation function such as, `count()`, `sum()`, `avg()`, `max()`, `min()`, and so on.
- `args` are the arguments to the aggregation function.

Here's an example (note: keys are the indices upon which we are aggregating the data):

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

The Solaris OS is a time-sharing system. In other words, the Solaris OS attempts to allocate time to work in small chunks in an effort to get as much work done as possible system-wide. Specific tasks or processes do not occupy all of the system's resources at any one time.

Processes are scheduled on a CPU for a specified time slot and moved off the CPU at the end of that time slot. A process can also go off CPU when the process needs a certain resource but that resource is blocked for some reason. Aggregation allows you to obtain and summarize pertinent data about a process over time, which might be more important than knowing how the process is doing in any one time slot.

It is possible to write a small D program to show which processes are going off the CPU. Using aggregation, this information can be printed in the form of a table. A two-column table with the process name in one column and the number of times it moved off the CPU in another column is a good way to use aggregation to provide a visual idea of how the process is doing in competition with other processes. In this case, we aggregate or summarize the count() of times when processes move off of CPU.

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

sysinfo:::pswitch
{
        @myTable[execname] = count() ;
}

END
{
        printa(@myTable) ;
}
```

The data is collected into the table called `myTable` and printed using the `printa()` function. Because there are no loops in D language, output from this program is displayed in its entirety when the program ends. The `printa()` function is a special function that loops through an aggregation and prints the values.

```
$./aggr.d

soffice.bin     4
dtrace          12
java            28
sched           56
```

Built-In Variables and Thread-Local Variables

Often, it is useful to determine the time spent in a certain function, for example, when profiling an application. When profiling, knowing which functions took the most time is the most interesting aspect. In the following example, the goal is to find out how much time the system spends reading data from the disk for a given application. If the application is multithreaded, and each thread reads data from the disk, a mechanism is required to pull that information together.

Therefore, it is necessary to obtain the timestamp when a `read` system call is issued and to ensure that subsequent reads by the same thread do not change that timestamp. D language provides a mechanism to do this through the built-in variable `timestamp`. This variable holds the number of nanoseconds since the epoch time (1/1/1970).

The value of `timestamp` can be captured whenever a `read` system call is entered using a thread-local variable. This ensures that cases where the probe fires later do not cause the timestamp to be overwritten. The next two scripts demonstrate this concept. The first script, `error.d`, illustrates *erroneous* usage and the second script, `correct.d`, illustrates *correct* usage.

```
#!/usr/bin/dtrace -qs
# error.d: This is the wrong way to use built-in variables

syscall::read:entry
/ execname == "java" /
{
        startTime = timestamp ;
}

syscall::read:return
/execname == "java" /
{
        @[pid] = sum ( startTime - timestamp) ;
}

END
{
        printa(@) ;
}
```

Assume there are two threads, `thread1` and `thread2`. After `thread1` issues a `read` call and before it returns, `thread2` issues a `read` call. So before `syscall::read:return` ever fires, `syscall::read:entry` is fired twice. This can cause the data in `startTime` to be overwritten and result in an erroneous calculation. This can be rectified as follows:

```
#!/usr/bin/dtrace -qs
#correct.d: This is the correct way to use built-in variables

syscall::read:entry
/ execname == "java" /
{
        self->startTime = timestamp ;
}

syscall::read:return
/ execname == "java" /
{
        @[pid]  = sum ( self->startTime - timestamp ) ;
        self->startTime = 0 ;
}

END
{
        printa(@) ;
}
```

The keyword `self` indicates that the variable `startTime` is a thread-local variable. A list of other useful built-in variables is as follows:

- `arg0...arg9` -- Arguments to functions represented in the `int_64` format
- `cpu` -- Current CPU ID
- `pid, ppid, tid` -- Process ID, parent process ID, and thread ID
- `probeprov` -- Probe provider
- `probemod` -- Probe module
- `probefunc` -- Probe function
- `timestamp` -- Timestamp in nanoseconds since epoch

DTrace Resources

- Using DTrace from a Solaris 10 System is a helpful how-to guide on sun.com.
- The Solaris Dynamic Tracing Guide is a complete reference for DTrace available on docs.sun.com.
- The DTrace web page on the BigAdmin sys admin portal has a lot of useful information.
- The OpenSolaris Community: DTrace offers several useful scripts, tools, and a vibrant community of users.
- The DTraceToolkit at OpenSolaris.org contains several useful scripts.
- For a better understanding of the internal workings of the Solaris OS, read the excellent book Solaris Internals by Richard McDougall and Jim Mauro.
- Chime is an open source DTrace visualization tool written in the Java programming language. You can find more details on the OpenSolaris site.
- The Java DTrace API is a work in progress to write user-defined probes in the Java Virtual Machine (JVM); for more information, refer to the OpenSolaris web site.

If you have any doubts about DTrace, participate in the OpenSolaris discussion forums and ask questions. The people are very friendly, they love questions, and they are extremely helpful.

Author Profile

Roy Cecil is a Member of Technical Staff with the Market Development Engineering Department at Sun Microsystems. He holds a Masters Degree in Computer Applications and works with Sun partners to ensure that their products run best on Sun platforms.

Licensing Information