Introduction

1 Added by jon_haslam, last edited by aleventhal on Aug 07, 2008

Introduction

Welcome to Dynamic Tracing in the Solaris Operating System! If you have ever wanted to understand the behavior of your system, DTrace is the tool for you. DTrace is a comprehensive dynamic tracing facility that is built into Solaris that can be used by administrators and developers on live production systems to examine the behavior of both user programs and of the operating system itself. DTrace enables you to explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. As you'll see, DTrace lets you create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions you can formulate using the DTrace D programming language. The first section of this chapter provides a quick introduction to DTrace and shows you how to write your very first D program. The rest of the chapter introduces the complete set of rules for programming in D as well as tips and techniques for performing in-depth analysis of your system. You can share your DTrace experiences and scripts with the rest of the DTrace community on the web at http://www.opensolaris.org/os/community/dtrace/ and http://www.sun.com/bigadmin/content/dtrace/. All of the example scripts presented in this guide can be found on your Solaris system in the directory /usr/demo/dtrace.

Top

Getting Started

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

Every probe in DTrace has two names: a unique integer ID and a human-readable string name. We're going to start learning DTrace by building some very simple requests using the probe named BEGIN, which fires once each time you start a new tracing request. You can use the dtrace(1M) utility's -n option to enable a probe using its string name. Type the following command:

```
# dtrace -n BEGIN
```

After a brief pause, you will see DTrace tell you that one probe was enabled and you will see a line of output indicating that the BEGIN probe fired. Once you see this output, dtrace remains paused waiting for other probes to fire. Since you haven't enabled any other probes and BEGIN only fires once, press Control-C in your shell to exit dtrace and return to your shell prompt:

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

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

You can construct DTrace requests using arbitrary numbers of probes and actions. Let's create a simple request using two probes by adding the END probe to the previous example command. The END probe fires once when tracing is completed. Type the following command, and then again press Control-C in your shell after you see the line of output for the BEGIN probe:

```
# dtrace -n BEGIN -n END
dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                  :BEGIN
^C
  0      2                    :END
#
```

As you can see, pressing Control-C to exit dtrace triggers the END probe. dtrace reports this probe firing before exiting.

Now that you understand a little bit about naming and enabling probes, you're ready to write the DTrace version of everyone's first program, “Hello, World.” In addition to constructing DTrace experiments on the command line, you can also write them in text files using the D programming language. In a text editor, create a new file called hello.d and type in your first D program:

Example: hello.d: Hello, World from the D Programming Language

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

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

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0         1                  :BEGIN   hello, world
#
```

As you can see, dtrace printed the same output as before followed by the text “hello, world”. Unlike the previous example, you did not have to wait and press Control-C, either. These changes were the result of the *actions* you specified for your BEGIN probe in hello.d. Let's explore the structure of your D program in more detail in order to understand what happened.

Each D program consists of a series of *clauses*, each clause describing one or more probes to enable, and an optional set of actions to perform when the probe fires. The actions are listed as a series of statements enclosed in braces { } following the probe name. Each statement ends with a semicolon (;). Your first statement uses the function trace to indicate that DTrace should record the specified argument, the string “hello, world”, when the BEGIN probe fires, and then print it out. The second statement uses the function exit to indicate that DTrace should cease tracing and exit the dtrace command. DTrace provides a set of useful functions like trace and exit for you to call in your D programs. To call a function, you specify its name followed by a parenthesized list of arguments. The complete set of D functions is described in Chapter 10, Actions and Subroutines.

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

Top

Providers and Probes

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

```
# dtrace -l
  ID   PROVIDER            MODULE          FUNCTION NAME
   1     dtrace                                     BEGIN
   2     dtrace                                     END
   3     dtrace                                     ERROR
   4   lockstat           genunix       mutex_enter adaptive-acquire
   5   lockstat           genunix       mutex_enter adaptive-block
   6   lockstat           genunix       mutex_enter adaptive-spin
   7   lockstat           genunix       mutex_exit  adaptive-release
   ... many lines of output omitted ...
#
```

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

```
# dtrace -l | wc -l
        30122
```

You might observe a different total on your machine, as the number of probes varies depending on your operating platform and the software you have installed. As you can see, there are a very large number of probes available to you so you can peer into every previously dark corner of the system. In fact, even this output isn't the complete list because, as you'll see later, some providers offer the ability to create new probes on-the-fly based on your tracing requests, making the actual number of DTrace probes virtually unlimited.

Now look back at the output from **dtrace -l** in your terminal window. Notice that each probe has the two names we mentioned earlier, an integer ID and a human-readable name. The human readable name is composed of four parts, shown as separate columns in the dtrace output. The four parts of a probe name are:

Provider	The name of the DTrace provider that is publishing this probe. The provider name typically corresponds to the name of the DTrace kernel module that performs the instrumentation to enable the probe.
Module	If this probe corresponds to a specific program location, the name of the module in which the probe is located. This name is either the name of a kernel module or the name of a user library.
Function	If this probe corresponds to a specific program location, the name of the program function in which the probe is located.
Name	The final component of the probe name is a name that gives you some idea of the probe's semantic meaning, such as BEGIN or END.

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

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

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

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

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

Now that you understand where probes originate from and how they are named, we're going to learn a little more about what happens when you enable probes and ask DTrace to do something, and then we'll return to our whirlwind tour of D.

Top

Compilation and Instrumentation

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

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

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

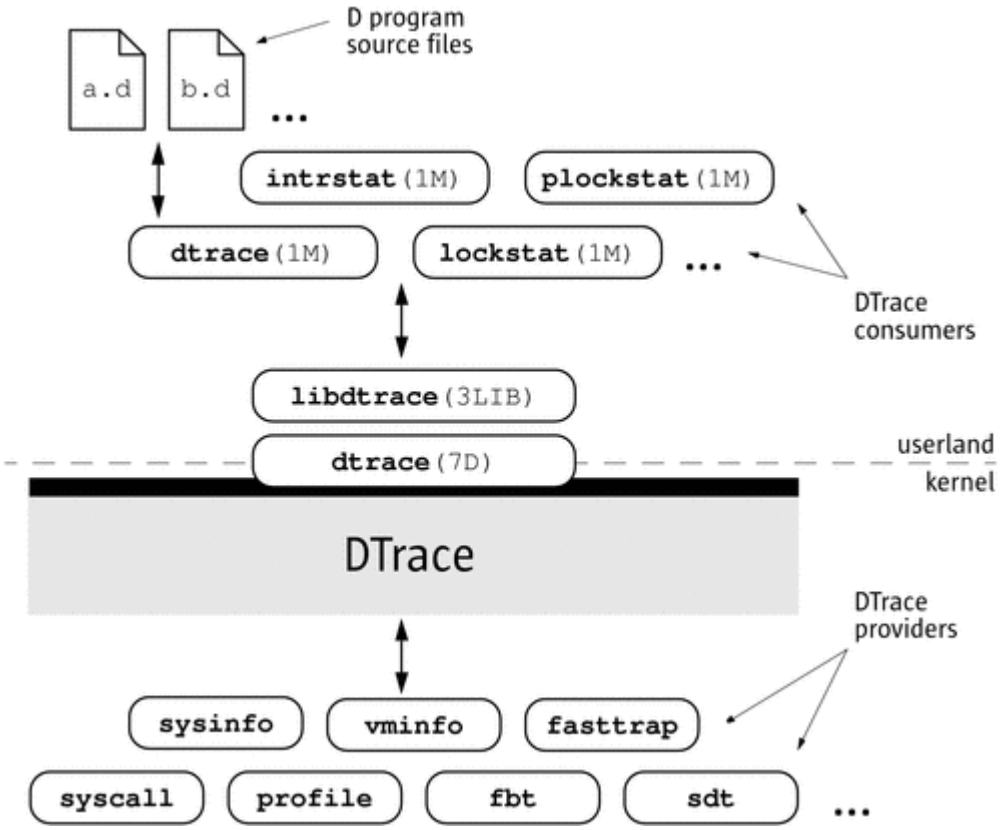
The DTrace framework also provides support for an arbitrary number of virtual clients. You can run as many simultaneous DTrace experiments and commands as you like, limited only by your system's memory capacity, and the commands all operate independently using the same underlying instrumentation. This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

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

reporting and debugging features are described later in this book.

The following diagram shows the different components of the DTrace architecture, including providers, probes, the DTrace kernel software, and the dtrace command.

Overview of the DTrace Architecture and Components



Now that you understand how DTrace works, let's return to the tour of the D programming language and start writing some more interesting programs.

Top

Variables and Arithmetic Expressions

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

```
/*
 * Count off and report the number of seconds elapsed
 */
dtrace:::BEGIN
{
        i = 0;
}

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

dtrace:::END
{
        trace(i);
}
```

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

```
# dtrace -s counter.d
dtrace: script 'counter.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0  25499                       :tick-1sec         1
  0  25499                       :tick-1sec         2
  0  25499                       :tick-1sec         3
  0  25499                       :tick-1sec         4
  0  25499                       :tick-1sec         5
  0  25499                       :tick-1sec         6
^C
  0      2                             :END         6
#
```

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

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

```
i = 0;
```

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

char	Character or single byte integer
int	Default integer
short	Short integer
long	Long integer
long long	Extended long integer

The sizes of these types are dependent on the operating system kernel's data model, described in Chapter 2, Types, Operators, and Expressions. D also provides built-in friendly names for signed and unsigned integer types of various fixed sizes, as well as thousands of other types that are defined by the operating system.

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

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

This clause names the probe profile:::tick-1sec, which tells the profile provider to create a new probe which fires once per second on an available processor. The clause contains two statements, the first assigning i to the previous value plus one, and the second tracing the new value of i. All the usual C arithmetic operators are available in D; the complete list is found in Chapter 2, Types, Operators, and Expressions. Also as in C, the ++ operator can be used as shorthand for incrementing the corresponding variable by one. The trace function takes any D expression as its argument, so you could write counter.d more concisely as follows:

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

If you want to explicitly control the type of the variable i, you can surround the desired type in parentheses when you assign it in order to *cast* the integer zero to a specific type. For example, if you wanted to determine the maximum size of a char in D, you could change the BEGIN clause as follows:

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

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

Top

Predicates

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

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

```
dtrace:::BEGIN
{
        i = 10;
}

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

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

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

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
        0  25499                         :tick-1sec        10
        0  25499                         :tick-1sec         9
        0  25499                         :tick-1sec         8
        0  25499                         :tick-1sec         7
        0  25499                         :tick-1sec         6
        0  25499                         :tick-1sec         5
        0  25499                         :tick-1sec         4
        0  25499                         :tick-1sec         3
        0  25499                         :tick-1sec         2
        0  25499                         :tick-1sec         1
        0  25499                         :tick-1sec   blastoff!
#
```

This example uses the BEGIN probe to initialize an integer i to 10 to begin the countdown. Next, as in the previous example, the program uses the tick-1sec probe to implement a timer that fires once per second. Notice that in countdown.d, the tick-1sec probe description is used in two different clauses, each with a different predicate and action list. The predicate is a logical expression surrounded by enclosing slashes / / that appears after the probe name and before the braces { } that surround the clause statement list.

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

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

The relational operator > means *greater than* and returns the integer value zero for false and one for true. All of the C relational operators are supported in D; the complete list is found in Chapter 2, Types, Operators, and Expressions. If i is not yet zero, the script traces i and then decrements it by one using the -- operator.

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

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

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

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

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

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

Now let's combine predicates with a new provider, the syscall provider, and create our first real D tracing program. The syscall provider permits you to enable probes on entry to or return from any Solaris system call. The next example uses DTrace to observe every time your shell performs a read(2) or write(2) system call. First, open two terminal windows, one to use for DTrace and the other containing the shell process you're going to watch. In the second window, type the following command to obtain the process ID of this shell:

```
# echo $$
12345
```

Now go back to your first terminal window and type the following D program and save it in a file named rw.d. As you type in the program, replace the integer constant 12345 with the process ID of the shell that was printed in response to your echo command.

```
syscall::read:entry,
syscall::write:entry
/pid == 12345/
{
}
```

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

```
# dtrace -s rw.d
dtrace: script 'rw.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
        0     34                   write:entry
        0     32                    read:entry
        0     34                   write:entry
        0     32                    read:entry
        0     34                   write:entry
        0     32                    read:entry
        0     34                   write:entry
        0     32                    read:entry
...
```

You are now watching your shell perform read(2) and write(2) system calls to read a character from your terminal window and echo back the result! This example includes many of the concepts described so far and a few new ones as well. First, to instrument read(2) and write(2) in the same manner, the script uses a single probe clause with multiple probe descriptions by separating the descriptions with commas like this:

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

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

```
/pid == 12345/
```

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

Variable Name	Data Type	Meaning
errno	int	Current errno value for system calls
execname	string	Name of the current process's executable file
pid	pid_t	Process ID of the current process
tid	id_t	Thread ID of the current thread
probeprov	string	Current probe description's provider field
probemod	string	Current probe description's module field
probefunc	string	Current probe description's function field
probename	string	Current probe description's name field

Now that you've written a real instrumentation program, try experimenting with it on different processes running on your system by changing the process ID and the system call probes that are instrumented. Then, you can make one more simple change and turn rw.d into a very simple version of a system call tracing tool like truss(1). An empty probe description field acts as a wildcard, matching any probe, so change your program to the following new source code to trace *any* system call executed by your shell:

```
syscall:::entry
/pid == 12345/
{
}
```

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

Top

Output Formatting

System call tracing is a powerful way to observe the behavior of most user processes. If you've used the Solaris truss(1) utility before as an administrator or developer, you've probably learned that it's a useful tool to keep around for whenever there is a problem. If you've never used truss before, give it a try right now by typing this command into one of your shells:

```
$ truss date
```

You will see a formatted trace of all the system calls executed by date(1) followed by its one line of output at the end. The following example improves upon the earlier rw.d program by formatting its output to look more like truss(1) so you can more easily understand the output. Type the following program and save it in a file called trussrw.d:

Example: trussrw.d: Trace System Calls with truss(1) Output Format

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

syscall::read:return,
syscall::write:return
/pid == $1/
{
        printf("\t\t = %d\n", arg1);
}
```

In this example, the constant 12345 is replaced with the label $1 in each predicate. This label allows you to specify the process of interest as an *argument* to the script: $1 is replaced by the value of the first argument when the script is compiled. To execute trussrw.d, use the dtrace options -q and -s, followed by the process ID of your shell as the final argument. The -q option indicates that dtrace should be quiet and suppress the header line and the CPU and ID columns shown in the preceding examples. As a result, you will only see the output for the data that you explicitly traced. Type the following command (replacing 12345 with the process ID of a shell process) and then press return a few times in the specified shell:

```
# dtrace -q -s trussrw.d 12345
                          = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)^C
#
```

Now let's examine your D program and its output in more detail. First, a clause similar to the earlier program instruments each of the shell's calls to read(2) and write(2). But for this example, a new function, printf, is used to trace data and print it out in a specific format:

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

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

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

%d	Print the corresponding value as a decimal integer
%s	Print the corresponding value as a string
%x	Print the corresponding value as a hexadecimal integer

DTrace printf works just like the C printf(3C) library routine or the shell printf(1) utility. If you've never seen printf before, the formats and options are explained in detail in Chapter 12, Output Formatting. You should read this chapter carefully even if you're already familiar with printf from another language. In D, printf is provided as a built-in and some new format conversions are available to you designed specifically for DTrace.

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

```
# dtrace -q -s trussrw.d
dtrace: failed to compile script trussrw.d: line 4: printf( )
           argument #2 is incompatible with conversion #1 prototype:
                conversion: %s
                 prototype: char [] or string (or use stringof)
                  argument: int
#
```

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

```
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
```

to trace the name of the current probe function and the first three integer arguments to the system call, available in the DTrace variables arg0, arg1, and arg2. For more information about probe arguments, see Chapter 3, Variables. The first argument to read(2) and write(2) is a file descriptor, printed in decimal. The second argument is a buffer address, formatted as a hexadecimal value. The final argument is the buffer size, formatted as a decimal value. The format specifier %4d is used for the third argument to indicate that the value should be printed using the %d format conversion with a minimum field width of 4 characters. If the integer is less than 4 characters wide, printf will insert extra blanks to align the output.

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

```
syscall::read:return,
syscall::write:return
/pid == $1/
{
        printf("\t\t = %d\n", arg1);
}
```

Notice that the syscall provider also publishes a probe named return for each system call in addition to entry. The DTrace variable arg1 for the syscall return probes evaluates to the system call's return value. The return value is formatted as a decimal integer. The character sequences beginning with backwards slashes in the format string expand to tab (\t) and newline (\n) respectively. These *escape sequences* help you print or record characters that are difficult to type. D supports the same set of escape sequences as C, C++, and the Java programming language. The complete list of escape sequences is found in Chapter 2, Types, Operators, and Expressions.

Top

Arrays

D permits you to define variables that are integers, as well as other types to represent strings and composite types called *structs* and *unions*. If you are familiar with C programming, you'll be happy to know you can use any type in D that you can in C. If you're not a C expert, don't worry: the different kinds of data types are all described in Chapter 2, Types, Operators, and Expressions. D also supports a special kind of variable called an *associative array*. An associative array is similar to a normal array in that it associates a set of keys with a set of values, but in an associative array the keys are not limited to integers of a fixed range.

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

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

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

```
a["hello", 123]++;
```

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

Example: rwtime.d: Time read(2) and write(2) Calls

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
        ts[probefunc] = timestamp;
}

syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
        printf("%d nsecs", timestamp - ts[probefunc]);
}
```

As with trussrw.d, specify the ID of shell process when you execute rwtime.d. If you type a few shell commands, you'll

see the amount time elapsed during each system call. Type in the following command and then press return a few times in your other shell:

```
# dtrace -s rwtime.d `pgrep -n ksh`
dtrace: script 'rwtime.d' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0     33                      read:return 22644 nsecs
  0     33                      read:return 3382 nsecs
  0     35                     write:return 25952 nsecs
  0     33                      read:return 916875239 nsecs
  0     35                     write:return 27320 nsecs
  0     33                      read:return 9022 nsecs
  0     33                      read:return 3776 nsecs
  0     35                     write:return 17164 nsecs
...
^C
#
```

To trace the elapsed time for each system call, you must instrument both the entry to and return from read(2) and write(2) and sample the time at each point. Then, on return from a given system call, you must compute the difference between our first and second timestamp. You could use separate variables for each system call, but this would make the program annoying to extend to additional system calls. Instead, it's easier to use an associative array indexed by the probe function name. Here is the first probe clause:

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
        ts[probefunc] = timestamp;
}
```

This clause defines an array named ts and assigns the appropriate member the value of the DTrace variable timestamp. This variable returns the value of an always-incrementing nanosecond counter, similar to the Solaris library routine gethrtime(3C). Once the entry timestamp is saved, the corresponding return probe samples timestamp again and reports the difference between the current time and the saved value:

```
syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
        printf("%d nsecs", timestamp - ts[probefunc]);
}
```

The predicate on the return probe requires that DTrace is tracing the appropriate process and that the corresponding entry probe has already fired and assigned ts[probefunc] a non-zero value. This trick eliminates invalid output when DTrace first starts. If your shell is already waiting in a read(2) system call for input when you execute dtrace, the read:return probe will fire without a preceding read:entry for this first read(2) and ts[probefunc] will evaluate to zero because it has not yet been assigned.

Top

External Symbols and Types

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

D uses the backquote character (`) as a special scoping operator for accessing symbols that are defined in the operating system and not in your D program. For example, the Solaris kernel contains a C declaration of a system

tunable named kmem_flags for enabling memory allocator debugging features. See the *Solaris Tunable Parameters Reference Manual* for more information about kmem_flags. This tunable is declared in C in the kernel source code as follows:

```
int kmem_flags;
```

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

```
trace(`kmem_flags);
```

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

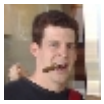
You have now completed a whirlwind tour of DTrace and you've learned many of the basic DTrace building blocks necessary to build larger and more complex D programs. The following chapters describe the complete set of rules for D and demonstrate how DTrace can make complex performance measurements and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

You've only just begun!

Comments (2)

Kedar.MhaswadeATSun.Com says: Jan 15, 2009

Can I go ahead and put Prev, TOC, Next links on each chapter?
I think these links are badly missed (by me, at least).



aleventhal says: Jan 15, 2009

Please do. Thanks.

The individuals who post here are part of the extended Sun Microsystems community and they might not be employed or in any way formally affiliated with Sun Microsystems. The opinions expressed here are their own, are not necessarily reviewed in advance by anyone but the individual authors, and neither Sun nor any other party necessarily agrees with them.

Sun Guidelines on Public Discourse | Privacy Policy | Terms of Use | Trademarks | Site Map | Employment | Investor Relations | Contact

Copyright 1994-2009 Sun Microsystems, Inc.
Powered by Atlassian Confluence