Learning DTrace -- Part 2: Scripts and the D Language

Chip Bennett

In Part 1 of this series on DTrace, I covered the fundamentals of using DTrace from the command line. Once DTrace command arguments start getting complex, it's generally easier to write a script rather than make the command line longer. Here is the basic layout of a D program:

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

probe-description[, probe-description[, ...]]
/ predicate /
{
   action; [action; [...]]
}
```

Let's break this down one part at a time:

1. The first line is similar to what you would see in any shell script [1]. The **-s** option must be specified so that Dtrace knows it is being invoked in script mode rather than command mode.

2. A *probe-description* is any 4-tuple probe description, as discussed in Part 1. These specifiers must be of the **-n** type. That is, the D compiler assumes that a single word, or whatever is after the last colon, is the probe name. (i.e., a probe-description of a single colon will match all probes.) You can specify more than one probe-description by using a comma-separated list of probe tuples.

3. The *predicate* is a C-style relational expression. This line is optional. A missing predicate has the same effect as a predicate that is always true.

4. The actions enclosed in braces are the actions to be performed if (1) the instrumented probe fires, and (2) the predicate is true. The action body in braces is also in the style of C, but does not have any flow control statements and has a different set of available functions than ANSI-C [2]. The actions are also optional. An action block that is empty (just an open and close brace) performs only the default action, as described in Part 1.

The probe-descriptions, predicate, and action block, together make what is called a *probe clause*. There can be many probe clauses in one D program, and their probe-descriptions can overlap, (i.e., **syscall:::entry**, **open:entry**, and **open:** will all match the firing of the **syscall::open:entry** probe). When a probe fires, only the clauses that match that probe are executed, and they are executed in the order they appear in the D program. (Keep in mind that the corresponding predicates also have to be true before the clause's action block is executed.)

Example Scripts

So let's take a look at some example scripts to get an idea of what they're like. The absolute simplest script possible is the one that instruments all of the probes on your system.

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

:
{ }
```

DTrace hint: You may only want to try this script on a lab or test system. Internally, this script is activating every available kernel probe, which is going to be on the order of tens of thousands. DTrace is extremely efficient, but even this script is a bit abusive. I've tried it many times without doing any more harm than using up a lot of CPU, but you should be cautious. Another problem you will probably see with this script is dropped data. Every time a probe fires, DTrace records data into a buffer (one for each consumer and each CPU). If the buffer becomes full, the data is dropped. You can see these drops by running ***dtrace -n : > /dev/null*** *on a lab or test system. The normal trace output will be routed to /dev/null, but the drop messages are sent to standard error.*

If you were to actually run this script, you would have to save it into a file, say "intense.d", turn on the execute permissions (**chmod +x intense.d**), and then execute it from the command prompt (**./intense.d**). Like any script, you can pass it as an argument to the interpreter, rather than turn it into an executable [3]. To invoke the script in this manner, you'd enter **dtrace -s intense.d**. If you use this method, then you don't need the preamble (**#!/usr/sbin/dtrace -s**), and you don't need to turn on the execute bits.

In the rest of my examples, I won't show the preamble, but keep in mind that you need it if you are executing your scripts by changing them into commands.

So now let's add a little more to our degenerate program so that it behaves more reasonably.

```
syscall::exec*:entry
{ }
```

Don't forget to invoke the script with the **-l** option initially to verify how many probes it's going to instrument (**dtrace -ls sample.d**). You should have matched two probes: *exec* and *exece* [4]. This is accomplished by using basic shell pattern matching when specifying probe tuples. Now try allowing the script to instrument (**dtrace -s sample.d**). How would you describe what this D program is displaying? (If you're not seeing any activity, try logging in from another window and type a few commands.)

You should be seeing default output for each time the **exece** system call is invoked. However, this script doesn't tell us which process is doing the **exec**. To display this information, we need a built-in variable and a way of printing to standard out.

Recording Actions and Built-in Variables

The D language provides an output action called *trace()*. It takes, as a single argument, a D expression. Additionally, D provides a set of built-in variables, one of which is *execname*, the name of the process that caused the probe to fire. In this case, that would be the process that called *exec*. The following modification to the script will show us this information, in addition to the default output:

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

To make the script complete, it would be nice to also print the name of the program being exec'd. Each type of system call has specific arguments passed to it. DTrace has a set of built-in variables (*arg0* through *arg9*) that contain the system call arguments. In the case of *exec*, *arg0* will contain the name of the program being exec'd. However, what's actually passed is a pointer to a string, so the *arg0* variable has to be treated as a pointer. (The Solaris reference manual lists the arguments for each system call.)

Subroutines and String Variables

Our pointer has a problem: it refers to a string that is not in the same address space [5] where the D program is running. The D program runs in the kernel, while the address in *arg0* points to a string in the process that called *exec*.

DTrace hint: Even though the ***dtrace*** *command runs in your address space, the D program gets compiled into an internal form called DIF (D Intermediate Format). When a probe fires, any DIF programs associated with that probe are executed in the kernel. This is why the program can't access the pointer in arg0 directly. (The DIF program records data in the buffer. Later, the* ***dtrace*** *command gets the recorded data from the buffer.)*

To access the string from userland, we need some kind of built-in function to get the data for us. DTrace has a limited set of built-in functions. One of these, **copyinstr**, takes a pointer as an argument and returns the null terminated string from the other address space. It does this by making a copy of the string in the probe clause scratch memory, which is automatically reclaimed when the probe clause ends.

But *copyinstr* doesn't just return a pointer to an array of characters. Unlike C, the D language has a real string type. This means that if you declare a variable as type string, wherever you reference this variable, your D program will use it as a string (not a pointer, a single character, or an array of characters). This also means that strings can be assigned using the assignment operator (=), and can be compared using the relational operators. The built-in variable *execname* is type string as is the returned value from *copyinstr()*.

With the changes I've described so far, here is the new D program:

```
syscall::exec*:entry
{
   trace(execname);
   trace(copyinstr(arg0));
}
```

The above program will work but will not produce very pretty output:

```
CPU     ID       FUNCTION:NAME
  0    106       exece:entry   ksh      /usr/bin/ls
  0    106       exece:entry   ksh      /usr/bin/who
```

Let's go over a couple of more features to make things look a little nicer. First, instead of using the **trace** action, we'll use **printf**, which allows formatting of the output. For the most part, D's **printf** works like its counterpart in C. Second, to make the output even cleaner, we'll get rid of the default output. D has a quiet option, which can be specified with **-q** on the command line or using a pragma in the D program. I've used pragma in my example below:

```
#pragma D option quiet

syscall::exec*:entry
{
   printf ("%-20s %s\n", execname, copyinstr(arg0));
}
```

Thread-Local Variables

We are almost finished with the D program, but there is one more pointer issue that we need to deal with. It is possible that the address in *arg0* points to a location in memory that is not in real memory (just on disk). This could happen because the string is a constant that has never been accessed or because the string got paged out at some point. DTrace can't access such addresses, because the kernel resident probe clause runs with interrupts disabled and can't wait around for the disk I/O to complete. The trick is that we need to defer access until the system call has completed; that way we know the string is in memory. This technique requires that we have another probe clause that gets executed when the *exec* system call returns.

However, when the *return* probe fires, *arg0* will have a different value. What are we going to do with the address in *arg0* from the *entry* probe? We need to save *arg0* so that we can reference the address later in the *return* probe clause. But where should we save it? Since Solaris is a multi-threaded operating system, there could be many *exec* system calls going on at the same time. We need to save the pointer in such a way that will allow a unique save location for each thread.

DTrace has a variable type called *thread-local*. When a *thread-local* variable is declared, a copy gets created for each new thread that fires the probe. You reference it by placing *self->* in front of the variable name. Thus, one thread, that fires the *exec:entry* probe and then later fires the *exec:return* probe, is accessing the same thread-local variables. Over the lifetime of a DTrace run, there could be hundreds of thousands of threads, so there has to be a way of getting rid of the variables when you know you don't need them anymore (i.e., when you don't care about the thread anymore).

Assigning 0 to a thread-local variable de-allocates its storage. So, here's the final script:

```
#pragma D option quiet

syscall::exec*:entry
{
   self->prog = copyinstr(arg0);
   self->exn = execname;
}

syscall::exec*:return
/ self->prog != NULL /
{
   printf ("%-20s %s\n", self->exn, self->prog);
   self->prog = 0;
   self->exn = 0;
}
```

DTrace gets the type of a thread-local variable implied from an assignment statement or explicitly from a declaration. If the type is implied, the assignment statement must be ahead of any use of the variable in your D code; however, if your probes fire in an order such that you reference a thread-local variable before it's initialized, the reference will return zero.

So, why do I check that **self->prog** has been set in the predicate of the second clause? If we didn't have this predicate, what would occur if I happen to start the DTrace program at the moment an exec call was in progress? For that thread, the first probe to fire would be the return probe. Since the entry probe hasn't fired yet, the thread-local variables have not been initialized, and the output would contain erroneous results (in this case, zeros). This type of check, where you don't know for sure which probe is going to fire first, is a common construction in DTrace.

Your output should look something like this:

```
sh                 /usr/bin/ls
sh                 /usr/bin/csh
csh                /usr/bin/pwd
```

There's usually more than one way to solve a given problem. Here's an alternative method that produces almost the same output:

```
#pragma D option quiet

syscall::exec*:entry
{
   self->exn = execname;
}

syscall::exec*:return
/ self->exn != NULL /
{
   printf ("%-20s %s\n", self->exn, execname);
   self->exn = 0;
}
```

On return from *exec*, *execname* contains the name of the program that was being exec'd, so I really don't need to save *arg0* in the entry probe. However, if you try this script, you'll see the output is a little different from the output of the first script.

Summary

In these two articles, I've really just scratched the surface of the capabilities of DTrace and the D language. In the future, I'll be covering other probe providers, and I'll introduce you to an extremely useful statistical feature called aggregations. Ultimately, we'll look at some ways to use DTrace to solve system problems. In the meantime, if you can't wait, or would just like more information on other built-in variables, subroutines, probe providers, etc., I refer you to the Solaris Dynamic Tracing Guide at:

```
http://docs.sun.com/app/docs/doc/817-6223
```

Endnotes

1. The rationale behind this construction is to allow the *exec* system call to determine how to invoke an executable. To *exec*, everything is an executable binary with a two-byte magic number at the beginning telling it what kind of executable. Normal executable binaries have codes that indicate whether the file is "ELF" format, whether the symbols have been stripped, etc. The **#!** two-byte code at the beginning of a script tells exec that this is a script and to actually invoke the executable whose path appears just after the **#!**, passing along any arguments and the rest of the script as parameters and data. This is the standard way all scripts are handled in Unix (sh, bash, csh, ksh, perl, etc.)

2. In the latest version of Solaris 10, or OpenSolaris, there have been some new functions added that are similar to the standard C string manipulation functions. However, documentation on these new features is slow in coming.

3. Invoking the script as an argument to the **dtrace** command actually makes more sense if you are going to invoke a new script with the **-l** option initially to verify how many probes you matched. The **-s** flag must be last, because **dtrace** expects anything after the **-s** flag to be the filename of your script. For example, to list the probes that the "intense.d" script would have instrumented, try **dtrace -ls intense.d**.

4. The **exec** system call is the mechanism by which one process calls another one. When you type an external command at a shell prompt, the shell forks (or replicates itself) to make two processes, and then the second (or child) process transforms itself into a different program by exec'ing that program. There are two basic exec system calls: **exec** and **exece**. The first form doesn't pass the environment variables to the new executable, and the second form does. You need to specify both in the probe tuple in order to make sure you capture all execs.

5. Address spaces work a lot like street addresses. Let's say you're looking for address 201. There's a 201 Main Street and a 201 Oak Street, but they're not the same location. So, when your probe clause receives a pointer as one of its arguments, the address is in the context of the process that caused the probe to fire. It may be address 201, but it's not an address on your street.

*Chip Bennett (**cbennett@laurustech.com**) is a systems engineer for Laurus Technologies (**http://www.laurustech.com**), a Sun Microsystems partner. He has more than 20 years experience with UNIX systems, and holds a B.S. and M.S. in Computer Science. Chip consults with Laurus and Sun customers on a variety of Solaris related topics, and he co-chairs OS-GLUG, the OpenSolaris Great Lakes User Group (Chicago).*