Learning DTrace -- Part 3: Advanced Scripting and Aggregations

Chip Bennett

In part two of this series on DTrace, I covered the basics of writing D programs. In this part, I'll go into more detail about the D language and tell you about some additional probe providers. Along the way, I'll show you some examples, expanding on last month's example script and developing a new one.

This leg of our adventure will deal more with the D language, which is based on C. I'll explain some of the common features, but it's beyond the scope of these articles to teach all of the capabilities of C. If you're not familiar with the C language, there are a number of online references and tutorials that can be found by searching the Internet for something like "C programming tutorials."

Arithmetic Expressions

D supports most of the arithmetic expressions that ANSI C supports. These expressions can be used in predicates, recording actions like *printf*, functions like *exit*, or in a statement by themselves. Standalone D expressions, like C expressions, must have a side effect to be useful. The two sets of operators with side effects are the assignment operators and the increment/decrement operators. For example, the expression *3+1* is a valid standalone expression, but it doesn't change anything.

One particular expression operation that D borrowed from C is especially useful in D. It is the conditional expression (expression1 ? expression2 : expression3). This is a powerful operation in C, but it's even more useful in D, because D doesn't have flow control statements [1]. Since this operation is so important in D and often unknown to casual C programmers, I'll touch on it here.

First, expression1 is evaluated. If the result is true (non-zero), then expression2 is evaluated and returned as the overall expression result. Otherwise, the result of expression 3 is returned. For example:

```
  printf ("The value of x is %s\n", x == 0 ? "zero" : "non-zero");
```

I'll tell you more about the usefulness of conditional expressions a little later.

DTrace Provider

DTrace has a number of providers for creating different type of probes. The only provider you've seen, unless you've been reading the manual, is the *syscall* provider, which provides a way to instrument Solaris system call ingress and egress.

There are times when you need to be able to perform actions only once: at the start of the run, and at the end of the run. DTrace has a provider, actually called the *dtrace* provider, that gives you this capability. The *dtrace* provider only has three probes. (Note that capitalization is important. These probe names are in all uppercase.):

- *BEGIN* -- Fires once at the start of the script, before any other probes fire
- *END* -- Fires once at the end of the script, after all the other probes fire
- *ERROR* -- Fires when a clause execution error occurs

To expand on the D program shown in part two (*execs.d*), suppose you'd like to add a header and a trailer to the output. You can use the *BEGIN* probe to print a header before any of the other probes fire. It also might be useful to keep some kind of tally of probe events, to be printed at the conclusion of the run. The *dtrace* provider's *END* probe can be used for this purpose:

```
#pragma D option quiet
dtrace:::BEGIN
{
   printf ("%-20s %s\n", "===============", "==============");
   printf ("%-20s %s\n", "Execing Process", "Execed Process");
   printf ("%-20s %s\n", "===============", "==============");
   total = 0;
}
syscall::exec*:entry
{
   self->exn = execname;
}
syscall::exec*:return
/ self->exn != NULL /
{
   printf ("%-20s %s\n",
   self->exn, execname);
   self->exn = 0;
   total++;
}
dtrace:::END
{
   printf ("%-20s %s\n", "===============", "==============");
   printf ("Total # of execs = %d\n", total);
}
```

In practice, you usually only specify *BEGIN* and *END* without the rest of the probe tuple; the *dtrace* provider probes are unique and well known.

You'll notice that the variable *total* is not prefixed with *self->*. This is because *total* is not a thread-local variable (see part two of this series for an explanation of thread-local variables). Since *total* is not one of the built-in variables, the D compiler will make it a global variable. Global variables maintain their value for the entire execution, and there is only one copy shared by all threads. Like other variable types in D, the compiler gets the type of a global variable from an explicit declaration or an assignment statement. All explicit declarations must be outside of the probe clauses.

How does DTrace decide when to fire the *END* probe? DTrace has an *exit* function that can be called from any clause. Additionally, you could type ^C to interrupt execution. When the D program calls *exit*, receives a SIGINT, or receives a SIGTERM, it executes the *END* clause(es) before exiting. Your output should look something like this:

```
===============      ==============
Execing Process      Execed Process
===============      ==============
sshd                 sh
sh                   locale
sshd                 pt_chmod
sshd                 sh
sh                   quota
sh                   cat
sh                   mail
^C
===============      ==============
Total # of execs = 7
```

Structures

In D, structures are declared and referenced the same way they are in C. There are two main reasons why you would use structures in D -- to organize your own data and to allow easy access to the myriad of structures in the kernel.

Like other kinds of declarations in D, you may only declare structures outside of probe clauses. The declaration syntax is the same as C, as is the dot operator used to reference a structure member (mystruct.x). Also, like C, if you are dealing with a pointer to a structure, you can access a member using the arrow operator (pmystruct->x). Most kernel structure references you run into will be pointers. They come from basically two sources: built-in variables and external variables.

Some of the built-in variables are pointers to structures. For example, *curpsinfo* is a pointer to the current psinfo structure (see **man -s 4 proc**). An extremely useful built-in variable is the *args* array. This is an array that represents the arguments passed to the probe clause. The *args* array is referenced just like any array in C, using a 0-based index. However, the *args* array has been given a special capability -- the probe provider can make each element of the *args* array a different type. For example, one probe might define *args[0]* an integer and *args[1]* a pointer to a structure. Another probe might define *args[0]*, *args[1]*, and *args[2]* to be pointers to different structures.

This flexibility allows strongly typed access to the most useful data that each probe has to offer. Note that not all probes have the *args* array available. Some probes don't have any arguments. Others have arguments, but the *args* array still isn't available. In those cases, there's another set of built-in variables: *arg0* through *arg9*. However, these variables are just integers. If they contain pointers to structures, the variables must be cast, and the structures to which they point may need to be declared, unless they're already declared by default (i.e., psinfo_t). The DTrace guide details which arguments are available for each provider and probe.

One of the providers that defines types for the *args* array is the *io* provider. It has a set of probes named *start,* which has various function names for different kinds of I/O. All of the *start* probes have three elements in their *args* arrays:

- args[0] -- pointer to a bufinfo structure
- args[1] -- pointer to a devinfo structure
- args[2] -- pointer to a fileinfo structure

Each structure provides information about different aspects of an I/O request. The following are the actual structure definitions. These declarations are automatically provided to any clauses called from an *io:::start* probe: you do not include them in your D program. This information was extracted from the Solaris Dynamic Tracing Guide:

```
typedef struct bufinfo {
    int b_flags;         /* flags */
    size_t b_bcount;     /* number of bytes */
    caddr_t b_addr;      /* buffer address */
    uint64_t b_blkno;    /* expanded block # on device */
    uint64_t b_lblkno;   /* block # on device */
    size_t b_resid;      /* # of bytes not transferred */
    size_t b_bufsize;    /* size of allocated buffer */
    caddr_t b_iodone;    /* I/O  completion routine */
    dev_t b_edev;        /* extended device */
} bufinfo_t;

typedef struct devinfo {
    int dev_major;       /* major number */
    int dev_minor;       /* minor number */
    int dev_instance;    /* instance number */
    string dev_name;     /* name of device */
    string dev_statname; /* name of device + instance/minor */
    string dev_pathname; /* pathname of device */
} devinfo_t;

typedef struct fileinfo {
    string fi_name;      /* name (basename of fi_pathname) */
    string fi_dirname;   /* directory (dirname of fi_pathname) */
    string fi_pathname;  /* full pathname */
    offset_t fi_offset;  /* offset within file */
    string fi_fs;        /* filesystem */
    string fi_mount;     /* mount point of file system */
} fileinfo_t;
```

With all this predefined power at hand, writing a D program to inspect I/O information becomes a much easier task. Here's an (*ios.d*) example that prints some of this information, as each I/O start occurs:

```
#pragma D option quiet
io:::start
{
   printf ("%5d %4d %4d %30s %30s\n", args[0]->b_bcount,
           args[1]->dev_major, args[1]->dev_minor,
           args[1]->dev_pathname, args[2]->fi_pathname);
}
```

Your output should look something like this:

```
 8192  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a  /usr/share/lib/termcap
 8192  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a                  <none>
32768  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a       /var/tmp/ExP3aOLg
16384  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a       /var/tmp/ExP3aOLg
16384  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a       /var/tmp/ExP3aOLg
34816  136  8 /devices/pci@1f,0/ide@d/dad@0,0:a                  <none>
```

No Control Constructs: Deal With It

The lack of control constructs in D is not as limiting as you might think. There are several methods for getting around this limitation:

1. Use the C preprocessor to implement static changes.

2. Utilize the ability to have more than one probe clause per probe.

3. Use conditional expressions to assign alternate values.

4. Drive your D script, or multiple D scripts, from a parent script or program.

C Preprocessor

As you may recall, when writing C programs, the C preprocessor (cpp), performs a first pass on the C source file and looks for directives that start with a # in column one. Based on the preprocessor directive, the source code is modified and then handed to the real compiler. For example, *#if*, followed by a logical expression, controls whether or not a block of code gets included in the compilation. You can invoke the C preprocessor for your D program using the **-C** flag on the DTrace command line. One use for this is to make the **printf** format a common definition that can be maintained in one place.

For example, here's how you might modify *execs.d* using the preprocessor directive *#define*:

```
#pragma D option quiet
#define fmt "%-20s %s\n"
BEGIN
{
  printf (fmt, "===============", "==============");
  printf (fmt, "Execing Process", "Execed Process");
  printf (fmt, "===============", "==============");
}
  ...
```

There are other DTrace command-line options available to control the preprocessor, such as **-D** to define a symbol or **-I** to add a path to the *#include* search. Any of these can also be included in the script preamble (#!/usr/sbin/dtrace). The preprocessor can be used to statically control which parts of the D program are given to the compiler.

Multiple Probe Clauses for a Probe

Conditional logic in the probe clause can be accomplished by dividing the clause into multiple clauses. The predicate is used to select which clause is executed when the corresponding probe fires. Also, since the size of a probe clause's trace record is static[1], this is the only way to generate a varying amount of trace data for a probe. For example, suppose I wanted to refine the printing of the header and trailer in *execs.d*.

It would be useful to reprint the header every 20 lines of output, and omit the separator line at the end if there were no execs printed:

```
#pragma D option quiet
BEGIN
{
   total = 0;
}
syscall::exec*:entry
{
   self->exn = execname;
}
syscall::exec*:return
/ self->exn !=NULL && !(total % 20) /
{
   printf ("%-20s %s\n", "===============", "==============");
   printf ("%-20s %s\n", "Execing Process", "Execed Process");
   printf ("%-20s %s\n", "===============", "==============");
}
syscall::exec*:return
/ self->exn != NULL /
{
   printf ("%-20s %s\n",
   self->exn, execname);
   self->exn = 0;
   total++;
}
END
/ total /
{
   printf ("%-20s %s\n", "===============", "==============");
}
END
{
   printf ("Total # of execs = %d\n", total);
}
```

The exec return clauses are executed in order each time the probe fires. The first clause will print the header if *total* is a multiple of 20 (which of course includes 0). The beauty of this method is that if no execs happen, the header will never be printed. When the *END* probe fires, if *total* is zero, the ending separator is not printed.

Another good example exists in the *ios.d* script. In the output, you probably noticed that *args[2]->fi_pathname* is the string **<none>** much of the time. This is because many IO requests are not associated with any particular file.

To pretty this up a bit, we could put the tracing of the file path in a different clause, and bypass it altogether if its value is <**none**>:

```
#pragma D option quiet
io:::start
{
   printf ("%5d %4d %4d %s\n", args[0]->b_bcount,
           args[1]->dev_major, args[1]->dev_minor,
           args[1]->dev_pathname);
}
io:::start
/ args[2]->fi_pathname != "<none>" /
{
   printf ("                %s\n", args[2]->fi_pathname);
}
```

The output should look something like this:

```
 1024  136    8 /devices/pci@1f,0/ide@d/dad@0,0:a
 1024  136    8 /devices/pci@1f,0/ide@d/dad@0,0:a
24576  136    8 /devices/pci@1f,0/ide@d/dad@0,0:a
                /var/tmp/Extya4Mg
 8192  136    8 /devices/pci@1f,0/ide@d/dad@0,0:a
                /var/tmp/Extya4Mg
```

Using Conditional Expressions

This technique allows your D program to choose different calculations to perform based on the probe data. As an alternative to the previous example (*ios.d*) of splitting the printed output into two probe clauses, a conditional expression can do the same thing in one probe clause:

```
#pragma D option quiet
io:::start
{
   printf ("%5d %4d %4d %s%s\n",
      args[0]->b_bcount, args[1]->dev_major,
      args[1]->dev_minor, args[1]->dev_pathname,
      args[2]->fi_pathname == "<none>" ?
        "" : strjoin("\n             ", args[2]->fi_pathname));
}
```

For this modification, I've also introduced a string function, *strjoin*. The function returns a string with the two argument strings concatenated. The conditional expression tests *args[2]->fi_pathname*. If it's equal to <**none**>, it prints a null string. If it's not equal to <**none**>, it prints the pathname, but it does so with a newline and some extra white space. This formats the pathname, so that it's lined up on the next line, just like the split probe clause method [2]. The output should look just like the previous example.

Another use for conditional expressions is to present information in a more readable form. The kernel usually stores an enumerated list of choices as a group of bits. For example, the *bufinfo* structure has a *b_flags* field that contains a single bit that represents whether an I/O is a read or a write. In the *ios.d* script, we can mask out this bit and then use a conditional expression to print a value that is more descriptive than 1 or 0:

```
#pragma D option quiet
io:::start
{
   printf ("%c %5d %4d %4d %s%s\n",
      args[0]->b_flags & B_READ ? 'R' : 'W', args[0]->b_bcount,
      args[1]->dev_major, args[1]->dev_minor, args[1]->dev_pathname,
      args[2]->fi_pathname == "<none>" ?
         "" : strjoin("\n               ", args[2]->fi_pathname));
}
```

Like C, the D language uses single quotes for character constants and double quotes for string constants. I chose to use a %c format item with character constants 'R' and 'W'. You could also have used %s with string constants "R" and "W". The choice is somewhat arbitrary [3]. This output should look like the previous examples, but with an R or W in the first column.

Driving the D Script from a Parent Script

In this method, you invoke the D script from another program, such as a shell script. In the parent script, you make all the decisions about how the D script is going to be invoked and then tailor the D script accordingly. The D program source can be contained within the shell script as one long command, or it can be invoked from a separate file. If it's embedded in the shell script, you can use shell variables to tailor the DTrace command line before it's invoked. There are some excellent examples of this technique in the DTrace Toolkit developed by Brendan Gregg at:

```
http://www.opensolaris.org/os/community/dtrace/dtracetoolkit/
```

If the D script is in a separate file, you can use D macro variables on the DTrace command line to tailor the D program. See the Solaris Dynamic Tracing Guide for more information about using macros.

Aggregations

One of the most powerful features of DTrace is aggregations, which provides the ability to automatically collect raw data and print statistical reports. An aggregating function is executed using the following general form:

```
@aggname[key1, key2, ...] = aggfunc(arg1, arg2, ...);
```

where *aggname* is a variable used to uniquely identify this aggregation, each key is an expression that defines how the aggregation is grouped, *aggfunc* is the name of one of D's built-in aggregating functions, and each arg is an expression that *aggfunc* acts on. By default, an aggregating function collects data each time it's executed, and the resulting statistics are printed when the D program exits (either via the *exit* function or ^C).

As an example, let's suppose I want to modify *ios.d* to include a grand total of both the number of I/O starts and the number of bytes read and written. Also, I want to segregate the totals into read totals and write totals. To get a total number of I/O starts, I'll use the *count* function, and to get a total number of bytes read and written, I'll use the *sum* function:

```
#pragma D option quiet
io:::start
{
   self->rw = args[0]->b_flags & B_READ ? "R" : "W";
   printf ("%s %5d %4d %4d %s%s\n", self->rw, args[0]->b_bcount, args[1]-
>dev_major,
      args[1]->dev_minor, args[1]->dev_pathname,
      args[2]->fi_pathname == "<none>" ?
         "" : strjoin("\n             ", args[2]->fi_pathname));

   @blcktotal[self->rw] = count();
   @bytetotal[self->rw] = sum(args[0]->b_bcount);

   self->rw = 0;
}
```

You'll note that I've added a thread local variable, *self->rw*, so that the conditional expression doesn't have to be repeated. The keys for these aggregations are the same as what is printed in the first column of the detail lines: "R" or "W". I had to make them strings instead of chars: keys with the char type, by default, are displayed as integers (which wouldn't look as nice). It's also worth noting that *count* and *sum(1)* yield the same results. Upon exit, the new *ios.d* will display four lines that look something like this:

```
  R                 107
  W                 129
  R              317440
  W              418816
```

We also have the ability to control the final output using the *printa* data recording action. This action works much like the *printf* action. The format string for *printa* is used to position and restrict how each result line is printed. Each regular format item corresponds to one of the keys in the aggregation. A special format item prefix, %@, corresponds to the aggregation result.

As an example, to get almost the same output as the default (except for the amount of white space), you could use the following in an *END* clause:

```
printa ("%s %@30d\n", @blcktotal);
printa ("%s %@30d\n", @bytetotal);
```

However, to make the output easier to read, you can change the field widths, add other constant data to the format, and surround the *printa* actions with *printf* actions:

```
#pragma D option quiet
io:::start
{
   self->rw = args[0]->b_flags & B_READ ? 'R' : 'W';
   printf ("%c %5d %4d %4d %s%s\n", self->rw, args[0]->b_bcount,
      args[1]->dev_major, args[1]->dev_minor, args[1]->dev_pathname,
      args[2]->fi_pathname == "<none>" ?
         "" : strjoin("\n              ", args[2]->fi_pathname));

   @blcktotal[self->rw] = count();
   @bytetotal[self->rw] = sum(args[0]->b_bcount);

   self->rw = 0;
}
END
{
   printf ("Blocks: R/W     Total\n");
   printf ("=====================\n");
   printa ("        %c %@11d\n", @blcktotal);
   printf ("\n");
   printf (" Bytes: R/W     Total\n");
   printf ("=====================\n");
   printa ("        %c %@11d\n", @bytetotal);
}
```

Note that I changed the result of the conditional expression back to a *char*, rather than a *string*. Now that I have control over the output format of the aggregations, I can use the *char* data type and actually print a character rather than its integer representation. The output at the end, after you enter ^C, should look something like this:

```
Blocks: R/W     Total
=====================
        R           4
        W          36
 Bytes: R/W     Total
=====================
        R        4096
        W       50176
```

Profile Provider

The *syscall* provider is an example of an anchored provider, because its probes are triggered by events in the code that you are observing. DTrace also has unanchored probes, which are asynchronous to the observed code. The *dtrace* provider's probes are examples of unanchored probes. Another unanchored probe provider is the *profile* provider. The *profile* provider's probes fire on a time interval. In *ios.d*, suppose you want to print the aggregation results on a 15-second interval. You could use the profile provider's *tick-15s* probe, which fires every 15 seconds. I've added this probe to *ios.d* below:

```
   ...

END,tick-15s
{
   printf ("\n");
   printf ("Blocks: R/W     Total\n");
   printf ("=====================\n");
   printa ("       %c %@11d\n", @blcktotal);
   printf ("\n");
   printf (" Bytes: R/W     Total\n");
   printf ("=====================\n");
   printa ("       %c %@11d\n", @bytetotal);
   clear (@blcktotal);
   clear (@bytetotal);
}
```

The *io:::start* probe clause hasn't changed, but I added the *tick-15s* probe to the *END* clause. This will cause the aggregations to be printed every 15 seconds, in addition to after the script is interrupted. The *clear* functions zero out the result values for each aggregation, making the output the total for the last 15-second period. If, instead, you want a running total, just remove the *clear* functions.

The units that can be specified for the *tick* probes are: nanoseconds (nsec or ns), microseconds (usec or us), milliseconds (msec or ms), seconds (sec or s), minutes (min or m), hours (hour or h), days, (day or d), and hertz (hz). Hertz (the default, if no suffix is supplied) specifies the number of times per second the probe will fire.

The numeric value of the probe is completely arbitrary. You could have just as easily used *tick-14s* or *tick-2s*. One of the interesting things about *profile* probes is that they don't exist until they are used. If you run **dtrace -lP profile** before using the *profile* provider the first time, you'll see a handful of probes that get created by default, but probably not *tick-15s*. When you run a D program that uses this probe, it gets created. Once the probe is created, it remains on the probe list until the system is rebooted or the *profile* module is unloaded [4]. On top of that, if five different users are running a D program that happens to use the *tick-15s* probe, they will all be using the same probe.

As you may recall, each DTrace consumer has its own trace buffer. When the *tick-15s* probe fires, that event gets shared with all the buffers for all the consumers who have clauses associated with that probe. This also means that each DTrace consumer sees the firing of the probe at about the same time. (There is a slight delay because of the buffer.) As a result, you shouldn't write a D program that depends on the first firing of a profile probe to be a certain amount of time from the start of the program. If the same probe has been instrumented by another consumer, your timing will vary according to when you enter the cycle. This is easiest to see in an example. Try running the following command in two different windows at the same time. Once the command is entered in the first window, no matter when you start the second instance, you'll see that each consumer shows *tick-15s* firing at about the same time:

```
dtrace -n tick-1s -n tick-15s
```

Summary

In this article, I've taken you a lot deeper into the D language than I did in part two. I think you can begin to see the extreme flexibility of D, and the power of observability that DTrace has on your system. In part four, I'll present some real examples of problems and show you how to solve them using DTrace. Of course, along the way, I'll explain more about the D language and some other probe providers.

Endnotes

[1] There's a couple of very good reasons why the D language doesn't have flow control statements like if, while, or for. In part two, I mentioned that a D program is compiled into a format called DIF (D Intermediate Format). The DIF program for each probe clause is executed in the kernel, with interrupts disabled. If a DIF program got into a long or infinite loop, a processor would now be running a long-lasting, kernel resident, uninterruptible thread. To prevent this possibility, the D compiler doesn't allow backward branches in DIF code. This precludes any kind of loop construct.

Additionally, DTrace requires that each probe clause produce a fixed-length trace record. This means that the total number of bytes generated by all of the recording actions (trace, printf, etc.), for one clause, must be constant. Each clause can generate a different size record, but one clause must generate the same size record each time it is invoked. If the designers allowed flow control statements, the size of the trace record could vary, depending on whether the if condition was true.

[2] You may be asking how we are cheating the static trace record limitation with the conditional expression method. When the D program alternates between the two possible results, isn't the DIF program in the kernel sending a different-sized record to the buffer? Actually it isn't (as you've probably guessed by my leading question). In D, strings are always allocated at a pre-defined length. By default, this pre-defined length is 256. The intermediate string generated by the conditional expression is 256 bytes, regardless of how much data is shoved into it. In this example, three integers and two 256-byte strings are actually put into the buffer. When the DTrace consumer pulls the data out of the buffer, it formats it using the *printf* format string, and varies the length of the output based on the position of the terminating null character.

[3] I say "somewhat" because there is a distinct difference in how much data is put in the recording buffer, considering that strings are always recorded at their full length, which by default is 256 bytes.

[4] DTrace providers are Solaris-loadable modules that can be listed using *modinfo*. Any default probes for a provider are created when the module is loaded. The provider's probes become

undefined when the module is unloaded (modunload -i *module_id*), or when the system is rebooted. So theoretically, profile provider probes, that you've created, could be removed by unloading the profile module, but I've never had success with this -- the module is always busy.

*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).*