Learning DTrace -- Part 5: Completing the Picture

Chip Bennett

This article is the last of a five-part series on DTrace. In part four, I provided examples of how DTrace can be used to iteratively solve performance problems and profile an application. In this final article, I'll cover more probe providers that will help monitor the system and your applications, and I'll show you ways to use DTrace at those times when the DTrace command isn't available.

The fbt Provider

The **fbt** (function boundary trace) provider gives you the ability to establish probes for any kernel function. For this provider, **probemod** specifies the kernel module (i.e., unix, ip, nfs), **probefunc** names the function, and **probename** stipulates whether you're instrumenting the entry probe or the return probe. As an example, suppose I wanted to show all of the kernel function calls for a system call. Let's create a D program called sclist.d:

```
syscall::$1:entry
{
   self->follow = 1;
   printf ("syscall %s begins", probefunc);
}
fbt:::entry
/ self->follow /
{
}
syscall::$1:return
/ self->follow /
{
   printf ("syscall %s ends", probefunc);
   exit(0);
}
```

In this script, I'm using a macro argument to make the script more flexible. When the D program is invoked, $1 will be replaced with the name of a system call to be inspected. Once the **syscall** probe fires, **self->follow** is set to true, which will allow the **fbt** probes to trace the instruction flow. The **fbt** probe specifier instruments all of the **fbt** entry probes, one for every kernel function. Surprisingly, instrumenting that many probes doesn't take very long. (The D program starts in a second or two.) An **fbt** probe will fire for every kernel function that is entered, but nothing gets recorded until **self->follow** becomes true. Since our goal is just to see the kernel functions that are called for a system call, this D program exits after the system call returns:

```
# dtrace -s sclist.d close
dtrace: script 'sclist.d' matched 22461 probes
CPU     ID                    FUNCTION:NAME
0   3028                      close:entry syscall close begins
0  11292                      close:entry
0  12331               closeandsetf:entry
0  14101                     closef:entry
0  13860                  fop_close:entry
0  19126                  ufs_close:entry
0  14961                cleanlocks:entry
```

```
0  16405         flk_get_lock_graph:entry
0  13622                cleanshares:entry
0  17434                    vn_rele:entry
0  14119                     crfree:entry
0  12227            kmem_cache_free:entry
0  11051                       setf:entry
0   8123                 fd_reserve:entry
0  12741               cv_broadcast:entry
0   3029                     close:return syscall close ends
```

Even though only a small portion of the instrumented probes are recording data, all of the **fbt** probes are instrumented, so be cautious where you try this script.

The usability of the **fbt** provider is enhanced by the availability of OpenSolaris source code at:

```
http://opensolaris.org
```

and books like *Solaris Internals*, Second Edition, by Richard McDougall and Jim Mauro. The argument list for each kernel function is available using the **args** array or the **argn** variables. You need to be cautious when writing **fbt**-based scripts that name specific kernel functions. There's no guarantee that a kernel function's argument list will be the same from one Solaris release to the next, or that the function will even still be present.

What would really be nice would be to see the flow as a thread enters and returns from functions. To that end, there is a D option called **flowindent**, which changes the output of the **fbt** provider. In the following example (scflow.d), I'll instrument all of the **fbt:::return** probes in addition to the **entry** probes and add the **flowindent** option:

```
#pragma D option flowindent
syscall::$1:entry
{
   self->follow = 1;
   printf ("syscall %s begins", probefunc);
}
fbt:::entry,
fbt:::return
/ self->follow /
{
}
syscall::$1:return
/ self->follow /
{
   printf ("syscall %s ends", probefunc);
   exit(0);
}
```

Here is what I captured using this script, but your output may be different depending on the type of file being closed:

```
# dtrace -s scflow.d close
dtrace: script 'scflow.d' matched 44876 probes
CPU FUNCTION
0  => close                                 syscall close begins
0    -> close
0      -> closeandsetf
0        -> closef
0          -> fop_close
0            -> ufs_close
0              -> cleanlocks
0                -> flk_get_lock_graph
0                <- flk_get_lock_graph
0              <- cleanlocks
0              -> cleanshares
0              <- cleanshares
0            <- ufs_close
0          <- fop_close
0          -> vn_rele
0          <- vn_rele
0          -> crfree
0          <- crfree
0          -> kmem_cache_free
0          <- kmem_cache_free
0        <- closef
0        -> setf
0          -> fd_reserve
0          <- fd_reserve
0          -> cv_broadcast
0          <- cv_broadcast
0        <- setf
0      <- closeandsetf
0    <- close
0  <= close

         syscall close ends
```

The pid Provider

The **pid** provider does for user code what the **fbt** provider does for kernel code. Any user program that still has its external symbol list, (linked with symbols included, and not stripped) can have **entry** and **return probes** instrumented. Let's start with an example that displays all the calls made within a library function (libflow.d). Notice the similarities to the **fbt** example:

```
#pragma D option flowindent
pid$target:libc:$1:entry
{
   self->follow = 1;
}
pid$target:::entry,
pid$target:::return
/ self->follow /
{
}
pid$target:libc:$1:return
/ self->follow /
{
   exit (0);
}
```

The **pid** provider is similar in function to the **fbt** provider. The main difference, from our perspective, is that there is only one kernel, but many processes. Similarly to how the profile provider creates new probes as needed, the pid provider allows for the creation of new providers. The actual provider name is the word **pid** followed by the process ID number of the process being traced. So, the probe specification **pid123::abc:entry** would instrument an entry probe at the beginning of function abc in process 123.

Our example script makes use of the **$target** macro to specify the process ID number for a **pid** provider. We'll use the **-c** flag later to set the macro to the process ID of a process that we want to trace. The first probe specification instruments a probe at the entry to a function from the libc library. The function, specified with the **$1** argument macro, will also come from the DTrace command line. This is our trigger probe. Similar to the **fbt** probe example, once the function we specify has been called, the script will trace all of the user code calls and returns until we return from the function with which we started:

```
# dtrace -s libflow.d -c ls closedir
dtrace: script 'libflow.d' matched 6150 probes
...
dtrace: pid 138 has exited
CPU FUNCTION
0  -> closedir
0    -> lfree
0      -> getbucketnum
0      <- getbucketnum
0      -> memset
0      <- memset
0      -> lmutex_lock
0      <- lmutex_lock
0    <- lfree
0    -> lmutex_unlock
0    <- lmutex_unlock
0    -> lfree
0      -> getbucketnum
0      <- getbucketnum
0      -> memset
0      <- memset
0      -> lmutex_lock
0      <- lmutex_lock
0    <- lfree
0    -> lmutex_unlock
0    <- lmutex_unlock
0   | closedir:return
0  <- closedir
```

In a second example, I've written a C program that calls a recursive function to calculate factorial. We can use DTrace to monitor the function calls and the values being passed. Here is the C program (factorial.c) that we'll monitor:

```
main()
{
   int f;
   f = fac(6);
   return 0;
}
int fac (int n)
{
   if (n <= 1)
      return 1;
   else
      return n * fac (n - 1);
}
```

This is a traditional recursive algorithm, but it can sometimes be a little confusing. DTrace allows us to see exactly what's going on in the function without adding any debug code to the C program. Our D program is called fac.d:

```
pid$target:factorial:fac:entry
{
   trace (arg0);
}
pid$target:factorial:fac:return
{
   trace (arg1);
}
```

We can choose whether or not to use flowindent, if we like, with the **-F** option on the command line:

```
# dtrace -s fac.d -c ./factorial
dtrace: script 'fac.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
0  46401                      fac:entry                6
0  46401                      fac:entry                5
0  46401                      fac:entry                4
0  46401                      fac:entry                3
0  46401                      fac:entry                2
0  46401                      fac:entry                1
0  46402                      fac:return               1
0  46402                      fac:return               2
0  46402                      fac:return               6
0  46402                      fac:return              24
0  46402                      fac:return             120
0  46402                      fac:return             720
dtrace: pid 258 has exited

# dtrace -Fs fac.d -c ./factorial
dtrace: script 'fac.d' matched 2 probes
CPU FUNCTION
0  -> fac                                              6
0    -> fac                                            5
0      -> fac                                          4
0        -> fac                                        3
0          -> fac                                      2
0            -> fac                                    1
0            <- fac                                    1
```

```
0          <- fac                                          2
0        <- fac                                            6
0      <- fac                                             24
0    <- fac                                              120
0  <- fac                                                720
dtrace: pid 263 has exited
```

The **pid** provider can also instrument offset points within a function. For example, **pid123::func:4**, instruments a point 4 bytes from the start of the function. Additionally, if no **probename** is specified for the probe, all of the instructions for the function are instrumented, and each instruction execution is traced. Use of this capability, indiscriminately, could make an application execute very slowly.

Variables and Arrays Revisited

We've looked at the different types of user-defined variables as we needed them. Now let's take one more look at all of the data types, just to round out the topic.

D has all of the same data types as C, so much so, that you can even include C system header files (i.e., **#include <stdio.h>**), provided you use the **-C** option [1]. In addition to the C data types, D also has real strings and associative arrays. Strings have already been explained in previous segments, so let's get into more details about arrays.

Scalar arrays are declared and used in D the same way they are in C. The following example uses scalar arrays of varying dimensions and types:

```
int x[5][2];
int y[3];
string z[3];
struct { int a; char b; } s[5];
BEGIN
{
   x[4][1] = 20;
   y[2] = 10;
   z[2] = "abc";
   s[3].b = 'c';
}
```

Associative arrays are indexed by a comma-separated list of keys that can be any scalar expression. Associative arrays are usually defined by their first use. For example, **x["abc", 12*p] = 47**, defines an associative array of integers with a string key and an integer key. Every assignment or reference after that must have the same key types and value type. Unlike scalar arrays, there's no need to declare the size of the array; storage is allocated as each location is assigned. Associative arrays can also be explicitly declared. The aforementioned array could be declared as **int x[string, int]**. As an example, the following D program uses an associative array to keep track of the first time a process calls the write system call:

```
# cat firstwrite.d
syscall::write:entry
/ !first[pid] /
{
   first[pid] = 1;
   printf ("First write for %s(%d)", execname, pid);
}
# dtrace -s firstwrite.d
```

```
dtrace: script 'firstwrite.d' matched 1 probe
CPU   ID                    FUNCTION:NAME
0   3024                    write:entry First write for sshd(5611)
0   3024                    write:entry First write for locale(5613)
0   3024                    write:entry First write for dtrace(5610)
0   3024                    write:entry First write for sshd(5597)
0   3024                    write:entry First write for sshd(5614)
0   3024                    write:entry First write for sshd(5616)
0   3024                    write:entry First write for cat(5618)
0   3024                    write:entry First write for ls(5620)
```

The associative array, **first**, initially has no true (1) values. Previously un-initialized locations return zero. The first time a process calls write, the array location, keyed by the PID of the process, is set to true (1). If that process does any more writes, the predicate will prevent the probe clause from being executed. The end result is that only the first write for each process is recorded.

How would the previous example be accomplished if there were no associative arrays? You certainly couldn't allocate a scalar array with enough elements for every possible PID! You'd have to use a smaller scalar array as a list of PIDs to search or create some kind of linked list. The problem is that you can't search an array or linked list. Do you remember why? That's right -- no flow control statements.

D doesn't allow explicit declarations inside the probe clause, so you can't declare variables with local scope, but you can declare variables that are only associated with one thread or one set of probe clauses. You've already met the former, thread-local variables, which are created for each thread. Clause-local variables, on the other hand, only maintain a value for the life of the clauses associated with the firing of one probe. They are defined by using "**this->**" in front of the variable name. Unlike global and thread-local variables, they don't get initialized, so you have to initialize them in your D program. They're great for accumulating intermediate results for one pass through a set of clauses.

Defining Your Own Application Probes

It's one thing to be able to instrument probes at any function within an application, but this doesn't necessarily carry any meaning to someone who isn't familiar with the internal workings of your application. DTrace has a mechanism called User-land Statically Defined Tracing (USDT) that gives the programmer the capability of creating instrumentation points anywhere in their code. The advantage over the **pid** provider is that the abstraction these probes represent is under control of the programmer.

For example, let's suppose you've written a messaging system and you want to allow other programmers the capability to debug message traffic. If you include the function symbols with the executable, others could use the PID provider, but they would need access to your source code, knowledge of the various functions that are involved in message traffic, and how to interpret the arguments to these functions. They may also need to write complex D clauses to drill down into your code to get to key data elements. And then if you modify the function interfaces and the internal data structures, you may break DTrace scripts that others have written. USDT gives you the tools to provide probe points at higher levels of abstraction (i.e., *msg-received* or *msg-sent*) along with whatever data you deem appropriate. At the same time, you can safely make changes to your code without affecting everyone else's use of DTrace with your program, as long as you don't change the USDT probe points.

Let's look at a simple example. Here is a C program that implements a stack of integers (push and pop), along with a header file for common definitions, and a program to call the stack routines to try them out:

```
# cat stack.c
#include <stdlib.h>
typedef struct stackstruct
{
   int item;
   struct stackstruct *next;
} STACK;

static STACK *head = NULL;

void push (int val)
{
   STACK *temp;
   temp = (STACK *) malloc (sizeof (STACK));
   temp->item = val;
   temp->next = head;
   head = temp;
}
int pop (void)
{
   int val;
   STACK *temp;
   if (head)
   {
      val = head->item;
      temp = head;
      head = head->next;
      free (temp);
   }
   else
   {
      val = 0;
   }
   return val;
}
# cat stack.h
void push (int);
int pop(void);
# cat do-stack.c
#include "stack.h"
main (int argc, char *argv[])
{
   int i, val;
   for (i = 1; i < argc; i++)
      push (atoi(argv[i]));
   while ((val = pop()) != 0)
      printf ("%d\n", val);
   return 0;
}
```

The file stack.c contains routines that allow another C program (in this case do-stack.c) to push and pop integers. The stack.h program provides the push and pop interface definitions to do-stack.c, which is a C main program that takes integer command line arguments and pushes them onto the stack.

Let's suppose you're the author of these programs, and you want others to be able to monitor what the stack is doing, with DTrace. At the same time, you don't want other programmers to have to deal with the details of how the program works. Additionally, it would be better if others could write DTrace scripts that didn't break when you make code changes. To accomplish this, you can add meaningful probe points into stack.c using the sdt.h system header file and DTRACE_PROBE macros. Here is stack.c modified to include the USDT probe points:

```
#include <stdlib.h>
#include <sys/sdt.h>
typedef struct stackstruct
{
   int item;
   struct stackstruct *next;
} STACK;
static STACK *head = NULL;

void push (int val)
{
   STACK *temp;
   DTRACE_PROBE1(mystack,push,val);
   temp = (STACK *) malloc (sizeof (STACK));
   temp->item = val;
   temp->next = head;
   head = temp;
}
int pop (void)
{
   int val;
   STACK *temp;
   if (head)
   {
      val = head->item;
      temp = head;
      head = head->next;
      free (temp);
      DTRACE_PROBE1(mystack,pop,val);
   }
   else
   {
      val = 0;
      DTRACE_PROBE(mystack,empty);
   }
   return val;
}
```

The DTRACE_PROBE macro comes in different flavors (DTRACE_PROBE, ...1, ...2, etc.) depending on how many arguments you want to pass to a probe clause. The first two arguments to the macro are the name of the USDT provider that is being created, and the name of the probe. In this case, the provider will be called **mystack**, and the probe names are **empty**, **push**, and **pop**. Next, you need to create a provider definition file for DTrace (mystack.d):

```
provider mystack {
   probe empty();
   probe push (int);
   probe pop (int);
};
```

The definition file lists all of the probes that will be apart of the mystack provider. To aid in the abstraction, you should try to make the probe names as self-explanatory as possible. Next, you

need to add the provider and probe definitions to the build process you use to create the applications:

```
# gcc -c stack.c
# dtrace -G -s mystack.d stack.o
# gcc do-stack.c mystack.o stack.o -o do-stack
```

The first line compiles your applications sources in the usual way. The **dtrace** command uses the **-G** option, which generates an object, in this case called **mystack.o**, which contains DTrace linkage information. The resulting object is then included in the final executable linkage (the third line). If you run the **do-stack** command without DTrace, you get something like this:

```
# ./do-stack 100 200 300
300
200
100
```

Now let's create a DTrace script (do-stack.d) that makes use of this new set of probes. The provider name is going to be **mystack<pid>**, where **<pid>** is the PID of the executing program. In my example, I'm going to use the **-c** flag to start the program with the script, though, as you may recall, there are different ways to associate a DTrace program with a process. Here's the DTrace script that instruments all of the probes provided by mystack:

```
mystack$target:::empty
{
}
mystack$target:::push
{
   trace (arg0);
}
mystack$target:::pop
{
   trace (arg0);
}
```

Now the moment of truth. Here is the output of the DTrace command that runs the D program and the C program together:

```
# dtrace -s do-stack.d -c "./do-stack 100 200 300"
dtrace: script 'do-stack.d' matched 3 probes
300
200
100
CPU     ID                    FUNCTION:NAME
  0  46407                        push:push              100
  0  46407                        push:push              200
  0  46407                        push:push              300
  0  46406                          pop:pop              300
  0  46406                          pop:pop              200
  0  46406                          pop:pop              100
  0  46405                          pop:empty
dtrace: pid 9692 has exited
```

In this example, the highest level of abstraction is not much better than just using the **pid** provider, but I think you can see that in a complex application, there is a lot of power to provide more meaningful information that is directly related to your application.

One other point of interest -- if you want to just list the probes that are being instrumented by your D program, you can do that:

```
# dtrace -ls do-stack.d -c ./do-stack
   ID   PROVIDER            MODULE                  FUNCTION NAME
46405 mystack9611          do-stack                  pop empty
46407 mystack9611          do-stack                  push push
46406 mystack9611          do-stack                    pop pop
```

Anonymous Tracing

Early in the Solaris boot process, it isn't possible for any DTrace consumers to be running, so DTrace provides the capability for trace data to be accumulated without a consumer. This is called "anonymous tracing". The most common use for this is to give device driver developers the ability to monitor and debug their code as it is being executed at system boot.

The two main problems with operating DTrace at boot, is that there is no way to run your DTrace program directly, and the provider modules may not have been loaded yet. These problems are solved by doing an "anonymous enabling". If you run your D program with the **-A** option, DTrace adds information to /kernel/drv/dtrace.conf that will cause all of the probes and actions in your D program to execute at system boot, even though there is no consumer present. Additionally, **forceload** statements are added to /etc/system for each provider that you'll need.

As an example, let's suppose you wanted to see a **count** aggregation of all of the **pts** (pseudo-tty slave) function calls as the **pts** driver is being loaded. To set this up, do the following:

```
# dtrace -A -n 'fbt:pts::entry { @[probefunc]=count() }'
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
```

When the system is rebooted, you'll see messages showing that the appropriate probes are being enabled:

```
NOTICE: enabling probe 0 (fbt:pts::entry)
NOTICE: enabling probe 1 (dtrace:::ERROR)
```

Trace data has now been collected, but it has been done so anonymously, without any consumer. After the system is booted, to claim the anonymous data, run the **dtrace** command with the **-a** option. (You also need to use the **-e** option to cause the command to exist after consuming the anonymous trace data. Otherwise, it will continue to run, waiting for new data.)

```
# dtrace -ae

  pts_detach                                        1
  pts_attach                                        2
  ptsclose                                          5
  ptsopen                                           5
  ptswsrv                                          15
  ptswput                                          20
```

Postmortem Tracing

DTrace can also be useful in determining the root cause of a system panic. If the system crashes while a DTrace program is running, there will be DTrace data left in the primary buffer that has not been consumed and reported by the **dtrace** command. You can use this data to help find the root cause of the crash. This methodology, called postmortem tracing, doesn't actually involve

the DTrace command, but instead uses the modular debugger (**mdb**) to extract the data. The **mdb** command has DTrace extensions to display trace data and buffer information.

Before showing you an example, I need to explain something about buffer policies. By default, DTrace creates two buffers per DTrace consumer and per CPU. This allows the DIF programs to write trace data to one buffer, while your **dtrace** command (the consumer) reads the other buffer. This buffer usage policy is called the "switch" policy. There is an alternate policy, the "ring" policy, which only uses one buffer. When this buffer becomes full, DTrace wraps around and starts writing over the oldest data. If you use a ring buffer with DTrace, when a panic occurs, the most current data will be left in the buffer.

As an example, I ran the the command **dtrace -n 'syscall:::entry' -b 16k -x bufpolicy=ring** to generate system call data in a 16K ring buffer. After starting this command, I panicked the OS and collected a crash dump. Then I invoked **mdb** with the crash dump. If more than one **dtrace** command was running at the moment of the crash, there might be several buffers available. The following mdb extensions will be helpful in determining which buffer is the right one:

```
# mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix specfs dtrace ufs ip sctp usba
fctl nca random nfs sppp crypto ptm sd cpc fcip ]
> ::dtrace_state
                ADDR MINOR             PROC NAME                    FILE
         300051d4ac0     2      30000315350 dtrace          3000780dc00
> 30000315350::ps
S    PID   PPID   PGID  SID  UID      FLAGS       ADDR NAME
R    659    627    627  627    0 0x4a004000 0000030000315350 dtrace
>
```

The **::dtrace_state** command displays the existing DTrace buffers. In my case, there is only one. The name of the consumer is listed, but of course, it's usually called "dtrace". You can further inspect the properties of the consumer process with the **::ps** command applied to the process structure pointer. Using the PID of the consumer process, you should be able to narrow in on the correct buffer.

Using the ADDR field from the **::dtrace_state** output, we can display the trace data:

```
> 300051d4ac0::dtrace
CPU     ID                    FUNCTION:NAME
0    119                       ioctl:entry
0    119                       ioctl:entry
0    253                  sysconfig:entry
0    253                  sysconfig:entry
0    189                  sigaction:entry
0    189                  sigaction:entry
0    119                      ioctl:entry
0    211                       mmap:entry
0    371                   schedctl:entry
0    147                   lwp_park:entry
0    305                lwp_sigmask:entry
0    305                lwp_sigmask:entry
0     33                       read:entry
0    329                   pollsys:entry
0    305                lwp_sigmask:entry
0    305                lwp_sigmask:entry
0     33                       read:entry
0    329                   pollsys:entry
```

```
0    305                lwp_sigmask:entry
0    305                lwp_sigmask:entry
0     33                       read:entry
0    329                    pollsys:entry
>> More [<space>, <cr>, q, n, c, a] ?
```

In my example, I've only collected entry data for each system call. Depending on the problem you are tracking down, you will want to tailor the trace data to something specific to the problem.

Summary

In this final article in the DTrace series, I've covered the use of DTrace with applications in more detail and showed you some methods for using DTrace at times when the DTrace command isn't available. As you look back at all five parts to this series, I think you can see that DTrace has a lot of power and flexibility to allow you to drill down into a system or an application, monitor what's going on, and get to the root cause of an issue more quickly. I hope you agree, as I stated at the beginning of the series, that DTrace is one of the most innovative features of Solaris 10.

Endnotes

1. You might ask why you would need to include system header files. The header files provide all of the structure definitions, typedefs, and #defines for inspecting system data. You get some declarations automatically when the probe fires, but not all. The problem is that the header files were written for C, so they sometimes contain D reserved words (like "string"). When this happens, the D compiler generates an error. If you really need the header file for your D program, the only solution is to make a private copy of it, change the word that's reserved, and add the directory of your private header files to an include list on the DTrace command line (**-I**).

*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. He 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).*