Learning DTrace -- Part 4: Solving Everyday Problems

Chip Bennett

In part three of this series on DTrace, I covered a lot of detail on the power of the D language. In this part, I'll use that power to look at two common uses for DTrace: application profiling and finding system performance problems.

Profiling Applications

Profiling is the process of analyzing an application's characteristics, usually for the purpose of improving its performance. Before you can profile a program, however, you need to give the D program the ability to restrict recorded events to those related to just one program. There are a couple of ways to accomplish this.

First, you can associate your D program with one or more processes by using either the **-c** option to specify a command to execute or the **-p** option to specify the process ID of an already running process. You can have multiple instances of both options, but the first one specified gets its process ID stored in a special macro variable, *$target*. (Macro variables get replaced with their associated value before the D program actually starts, so any place in your D program you use *$target* will get replaced with the process ID number of the first **-c** or **-p** process you specify.) As an example, you might use the *$target* macro in a predicate to restrict probe clause execution to just the one process. After all of the associated processes exit, the D program will exit too, displaying any default ending output or running your *END* probe.

Second, you can simply pass a process ID number as an argument to the D program. Any arguments, after any D program options, are passed to macro variables *$1*, *$2*, *$3*, etc. (The **$0** macro is the name of your D program.) Since DTrace has no way of knowing that your arguments are processes, when using this method your D program doesn't exit when the processes exit.

A good first step, when profiling, is to gather information on where a process is spending its time. For my profiling experiment, I'm going to compare the **cp** command and the **dd** command for copying a file. If I start out with a 50M file called **file1** and run **time cp file1 file2**, I get:

```
real        4.1
user        0.0
sys         0.9
```

However, if instead a copy the file using **time dd if=file1 of=file2**:

```
real        7.5
user        1.0
sys         4.6
```

The **dd** command took almost twice as long to copy the file and used more than four times as much system time [1]. Because the difference in system time is so high, let's do a comparison of the system calls that each command is executing. To gather a system call count, I'll use the following D program (*scount.d*):

```
syscall:::entry
/ $target == pid /
{
   @[probefunc] = count()
}
```

As you may recall from part three of this series, this D program uses the count aggregating function to gather the total number of times each system call is invoked. For the syscall provider, *probefunc* is the system call name, so the count aggregation will be keyed on system calls. The predicate restricts our data collection to system calls made by the target process. Now I'll run each copy command against this D program:

```
# dtrace -s scount.d -c "cp file1 file2"
dtrace: script 'scount.d' matched 226 probes
dtrace: pid 2194 has exited

  creat64                                           1
  fstat64                                           1
  getpid                                            1
  getrlimit                                         1
  open64                                            1
  pathconf                                          1
  rexit                                             1
  acl                                               2
  brk                                               2
  memcntl                                           2
  setcontext                                        2
  open                                              3
  resolvepath                                       3
  stat                                              3
  close                                             5
  stat64                                            5
  mmap64                                            7
  munmap                                            7
  write                                             7
  mmap                                             15

# dtrace -s scount.d -c "dd if=file1 of=file2"
dtrace: script 'scount.d' matched 226 probes
102400+0 records in
102400+0 records out
dtrace: pid 2200 has exited

  creat64                                           1
  fstat64                                           1
  getpid                                            1
  getrlimit                                         1
  open                                              1
  open64                                            1
  resolvepath                                       1
```

```
  rexit                                             1
  stat                                              1
  sysconfig                                         1
  munmap                                            2
  setcontext                                        2
  sigaction                                         2
  close                                             3
  brk                                               4
  mmap                                              4
  read                                         102401
  write                                        102406
```

Two differences between **cp** and **dd** stand out: the **cp** command only uses seven writes, whereas the **dd** command uses 102,406 writes to accomplish the same copy. Also, the **cp** command is suspiciously missing any reads at all. So then how is it reading file1 to copy it?

Clearly, even this high level of analysis gives us some notions as to what might be going on here. Since **dd** is making many more write calls than **cp**, I'm led to believe that it's probably writing smaller blocks than **cp**. Also, the lack of any read calls in **cp** makes me wonder if **cp** has a different, possibly better, way to read files.

Here is where we see the power of DTrace. The D programming language gives you the tools to look at both the forest and the trees, and any zoom level between. To prove or disprove our theories, we'll need to look at the system calls more closely. Let's take a look at the *write* system calls. The *write* system call has three arguments (see **man -s 2 write**). The first argument is the file descriptor, the second is a pointer to the buffer to be written, and the third is the number of bytes to be written. To focus on the *write* system calls only, we'll create a new D program (wcount.d):

```
syscall::write:entry
/ $target == pid /
{
   @[arg0] = quantize(arg2);
}
```

For this example, I've introduced a new aggregating function, *quantize*. By applying *quantize* to the length of the write, it displays a frequency distribution of the write lengths. A separate distribution graph is displayed for each aggregation key, which in this case is the file descriptor of the write:

```
# dtrace -s wcount.d -c "cp file1 file2"
dtrace: script 'wcount.d' matched 1 probe
dtrace: pid 2264 has exited


           4
  value  ------------- Distribution ------------- count
1048576 |                                         0
2097152 |@@@@@@                                   1
4194304 |                                         0
8388608 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 6
16777216 |                                        0

# dtrace -s wcount.d -c "dd if=file1 of=file2"
dtrace: script 'wcount.d' matched 1 probe
102400+0 records in
102400+0 records out
dtrace: pid 2266 has exited

           2
  value  ------------- Distribution ------------- count
      1 |                                         0
      2 |@@@@@@@@@@@@@@@                          2
      4 |@@@@@@@@@@@@@@@                          2
      8 |@@@@@@@@@@@@@@@                          2
     16 |                                         0

           4
  value  ------------- Distribution ------------- count
    256 |                                         0
    512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 102400
   1024 |                                         0
```

Each row in the output represents a "pail". (The *Solaris Dynamic Tracing Guide* [2] calls it a bucket, so I'll use pail.) Each pail contains values that are greater than or equal to the value listed at the left of the row, but smaller than the value on the next row ($\mathbf{r}_n <= x < \mathbf{r}_{n+1}$). The value r for each row increases in powers of two. For example, the second row of the third distribution says that there were 102,400 writes that were greater than or equal to 512 bytes but less than 1024.

In this case, I suspect that most of the writes are the same length, so it really isn't sufficient to just know which pail the write falls in. We need to know exactly how many writes occurred for each buffer size.

Even though *quantize* produces prettier output, to accomplish this I'm going back to using *count* (wcount2.d):

```
syscall::write:entry
/ $target == pid /

{
   @[arg0, arg2] = count();
}
```

Running this script against each copy command produced the following results for me:

```
# dtrace -s wcount2.d -c "cp file1 file2"
dtrace: script 'wcount2.d' matched 1 probe
dtrace: pid 13164 has exited

        4          2097152               1
        4          8388608               6
# dtrace -s wcount2.d -c "dd if=file1 of=file2"
dtrace: script 'wcount2.d' matched 1 probe
102400+0 records in
102400+0 records out
dtrace: pid 13166 has exited

        2               12               1
        2               13               1
        2                2               2
        2                6               2
        4              512               102400
```

The **cp** command only had seven writes: six 8M writes and one 2M write, for a total of 50M. The **dd** command wrote to two file descriptors. File descriptor 2 is standard error, so we conclude that these writes were used to display the record count messages. That leaves file descriptor 4 with 102,400 512 byte writes. These writes also add up to 50M of data. Our conclusion is correct: **dd** is doing a lot more small writes. It turns out that the **dd** command has a blocksize parameter that allows us to make **dd** use an 8M buffer for copying. If I run **time dd if=file1 of=file2 bs= 8192k**, I get the following results:

```
6+1 records in
6+1 records out

real        3.3
user        0.0
sys         1.3
```

The messages from **dd** clearly show that **dd** is now writing six 8M blocks and one 2M block. We could prove that with our D script, but I'll leave that for you to try on your own. The system call time is still almost 50% higher for **dd** than it is for **cp**. Perhaps this has something to do with **cp** not using the read system call. If so, how is **cp** reading the file?

The only other system call, in the call summary, that has to do with file IO, is *mmap*. This system call maps an area of memory over a file segment, so that the file becomes part of virtual memory. As the program accesses the memory, Solaris pages in sections of the file. There are no read system calls. There is still IO, of course, but it's all managed by the system's paging algorithms,

which are very efficient. Sounds like a good theory. How do we prove it? Like we did with write, we can restrict the D program to just look at the *mmap* system call (mcount.d):

```
syscall::mmap*:entry
/ $target == pid /
{
   @[(int)arg4, arg1] = count();
}
```

The *mmap* system call has six arguments (see **man -s 2 mmap**), but only the size of the allocated area of memory (*arg1*) and the file descriptor (*arg4*) are of interest to us right now. I used a metacharacter on the probefunc name because there are two mmap system calls: *mmap* and *mmap64*. Also, note that I cast the aggregation key, *arg4*, into an int. The argn parameters are typed *unsigned*, and the file descriptor argument to mmap can be a **-1**, when the allocated area is anonymous, that is, with no existing file mapped behind it. Here's the output from the command **dtrace -s mcount.d -c "cp file1 file2"**:

```
dtrace: script 'mcount.d' matched 2 probes
dtrace: pid 14500 has exited

        3              296             1
        3             3260             1
        3            13512             1
        3            13613             1
        3            55381             1
        3          2097152             1
       -1             5880             1
       -1            16384             1
       -1            24576             1
       -1            81920             1
       -1           147456             1
       -1             8192             2
        3             8192             3
        3          8388608             6
```

There is a distribution around some anonymous mmaps (file descriptor -1), and a distribution around file descriptor 3. (Note that the counts for file descriptor 3 are split into two groups. This is because count sorts its output by the count value [3]). It's file descriptor 3 that is of more interest. However, there are several different memory size allocations. How do we know which ones have to do with reading file1? I'm suspicious that the mmaps of size 8M and 2M are the ones we're looking for, but we should prove it.

To do this, we need to restrict our output to mmap calls that are only associated with reading "file1". In our D program, we can set up a control variable that gets set once file1 is opened (mcount2.d):

```
BEGIN
{
   fd = -1;  /* -1 means file1 not open */
}

/* If file1 not open, save the file name pointer */

syscall::open*:entry
/ $target == pid && fd == -1 /
{
   fnp = arg0;
}

/* If file1 not open, open succeeded,
      and filename is file1, save fd */

syscall::open*:return
/ $target == pid && fd == -1 && arg0 != -1 /
{
   fd = basename(copyinstr(fnp)) == "file1" ? arg0 : -1;
}

/* If file1 is open and the mmap fildes is file1's,
      collect size data */

syscall::mmap*:entry
/ $target == pid && fd != -1 && fd == arg4 /
{
   @[arg4, arg1] = count();
}

/* If close is for file1, show that file1
      no longer open */

syscall::close*:entry
/ $target == pid && arg0 == fd /
{
   fd = -1;
}
```

(Note that I used global variables rather than thread-local variables, since the cp command and the dd command are single threaded.) Once the D program sees an open system call for file1, it sets *fd* to the file descriptor from that open. The probe clause for the mmap probe is predicated on file1 being open, so we only collect data for mmaps associated with file1. (For information on why we need to handle the file name pointer from the open: entry in the open:return probe clause, see part 2 of this series.)

Here's what we get from running the command **dtrace -s mcount2.d -c "cp file1 file2"**:

```
dtrace: script 'mcount2.d' matched 8 probes
dtrace: pid 14530 has exited

        3          2097152                1
        3          8388608                6
```

This clearly shows us that the **cp** command maps 8M of memory to file1 six times and 2M of memory to file1 one time. But just because file1 is being mmapped, does this prove that this is how file1 is being read? Because there are no other file IO system calls present, probably; but there is one more step we might take to further prove it. It would be difficult to trace the **cp** program actually accessing the mmapped memory locations, but we don't really need to. All we need to see is the kernel paging in the file data from memory. To this end, here is a change to mcount2.d (mcount3.d):

```
#pragma D option quiet
BEGIN
{
   fd = -1;  /* -1 means file1 not open */
   pc = 0;   /* Page-in count per mmap  */
}
syscall::open*:entry
/ $target == pid && fd == -1 /
{
   fnp = arg0;
}
syscall::open*:return
/ $target == pid && fd == -1 && arg0 != -1 /
{
   fd = basename(copyinstr(fnp)) == "file1" ? arg0 : -1;
}
syscall::mmap*:entry
/ $target == pid && fd != -1 && fd == arg4 /
{
   printf ("Last mmap page-in count %d\n", pc);
   pc = 0;
}
vminfo:::fspgin
/ $target == pid && fd != -1 /
{
   pc++;
}
syscall::close*:entry
/ $target == pid && arg0 == fd /
{
   fd = -1;
}
END
{
   printf ("Last mmap page-in count %d\n", pc);
}
```

I've added a new probe, vminfo:::fspgin. The vminfo probes fire when virtual memory (paging and swapping) events occur. The event we are interested in is the page in of filesystem data between mmap calls. To this end, the D program has a page counter (pc), which gets bumped

every time the fspgin probe fires. When the mmap probe clause executes, indicating a new memory segment was mapped over file1, pc gets displayed, and then set back to zero. The first time we run **dtrace -s mcount3.d -c "cp file1 file2"**, we see the following:

```
Last mmap page-in count 0
Last mmap page-in count 67
Last mmap page-in count 64
Last mmap page-in count 69
Last mmap page-in count 64
Last mmap page-in count 65
Last mmap page-in count 64
Last mmap page-in count 18
```

The first count is zero, because this is the first mmap and no mmapped memory has been accessed yet. Every mmap call after that shows that 60 to 70 page-ins have occurred after each mmap call. The paging activity is fairly consistent and would seem to be a result of the **cp** command accessing the memory that was just mapped in. The last count is a lot smaller because the last mmapped file area was only 2M. With cache now loaded with the contents of file1, if we run the command again, we get this instead:

```
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
Last mmap page-in count 0
```

Since the contents of file1 was still in the cache, no page-ins were needed to do the copy a second time [4].

Solving a System Performance Problem -- Anonymous CPU Time

Another common use of DTrace is to discover system problems that conventional tools have trouble with. In this example, I seem to be using a lot of CPU time, but there seem to be no processes to blame. **vmstat 5** gives me the following output:

```
kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr dd dd s1 --   in   sy   cs us sy id
0 0 21 2773568 815736 2  8  3  3  4  0 29  0  0  0  0  463   26  100  0  1 99
0 0 37 2641992 768704 2474 5583 0 0 0 0 0  0 19  0  0  499 4493  268 55 45  0
0 0 37 2641968 768928 2493 5616 0 0 0 0 0  0 19  0  0  499 4494  259 55 45  0
```

The system has no idle time left and is spending about half its time in system calls and about half in user code. Let's take a look at **prstat** to determine which processes are using up all this time:

```
  PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
27494 root     1576K 1120K sleep   49    0   0:00:00 0.7% callfast/1
28891 root     4048K 2072K run     29    0   0:00:00 0.2% ld/1
28041 root     4792K 4424K cpu0    59    0   0:00:00 0.2% prstat/1
28887 root     1352K  960K sleep   19    0   0:00:00 0.1% gcc/1
28890 root     1320K  928K run     19    0   0:00:00 0.0% collect2/1
```

The process time leaders don't seem to show any process using this kind of CPU time. Quite a bit of the CPU time is being spent on system calls, so let's take a closer look at exactly which system calls are being invoked. We'll need to modify our original scount.d script, since this time we're not looking at a specific target. Also, even though I could just collect data for a while and then interrupt my D program, this time I'd like for it to run exactly 5 seconds and exit on its own. To do this, I'll need to be creative with the profile provider. I'll call this new script sc.d:

```
#pragma D option quiet
BEGIN
{
   tick2 = 0;
}
tick-5s
/ tick2 /
{
   exit (0);
}
tick-5s
/ !tick2 /
{
   trace ("Collecting data\n");
   tick2 = 1;
}
syscall:::entry
/ tick2 /
{
   @[probefunc] = count();
}
```

The control variable, *tick2*, allows us to collect data on the second firing of the tick-5s probe. As you may recall, we can't rely on the profile probes waiting the specified time the first time they fire, so we want to collect data for 5 seconds on the first firing of tick-5s. The first time tick-5s fires, the probe clause sets tick2 = 1, which makes the predicate true for the sycall:::entry probe clause. The D program collects data for 5 seconds and then exits on the next firing of tick-5s. When I ran this D program against my busy system, I got the following output:

```
Collecting data

  lwp_continue                                    1
  lwp_create                                      1
  lwp_exit                                        1
  lwp_kill                                        1
  nanosleep                                       1
  stat64                                          1
  pset                                            3
  pollsys                                         4
  gtime                                           7
  lwp_park                                       10
  vfork                                          25
  chmod                                          26
  rusagesys                                      26
  open64                                         50
  getgid                                         52
  getuid                                         52
  umask                                          52
  sysconfig                                      75
  times                                          75
```

```
  schedctl                                          77
  fork1                                            103
  lwp_self                                         103
  unlink                                           125
  rexit                                            127
  exece                                            128
  fcntl                                            129
  read                                             155
  waitsys                                          178
  fstat64                                          206
  write                                            210
  getrlimit                                        228
  memcntl                                          231
  setcontext                                       257
  llseek                                           259
  ioctl                                            281
  lseek                                            333
  dup                                              377
  fstat                                            384
  resolvepath                                      640
  munmap                                           665
  getpid                                           860
  lwp_sigmask                                      865
  access                                           869
  sigaction                                       1173
  close                                           1375
  open                                            1793
  mmap                                            2441
  stat                                            2479
  brk                                             2950
```

There is a lot of system call activity here, so we need to pick one of the systems calls to get more detail. There were 1793 open calls over a 5-second period. Let's find out which files the system was so busy opening (oc.d):

```
syscall::open:entry
{
   self->fnp = arg0;
}
syscall::open:return
/ self->fnp /
{
   @[copyinstr(self->fnp)] = count();
   self->fnp = 0;
}
```

The output I received from executing this script was rather lengthy, so I'll just show you the more important last 10 lines:

```
...

  /usr/sfw/lib/gcc/sparc-sun-solaris2.10/3.4.3/libgcc.a       54
  /usr/sfw/lib/gcc/sparc-sun-solaris2.10/3.4.3/libgcc.so      54
  /usr/sfw/lib/gcc/sparc-sun-solaris2.10/3.4.3/libgcc_eh.a    54
  /usr/sfw/lib/gcc/sparc-sun-solaris2.10/3.4.3/libgcc_eh.so   54
  /usr/sfw/lib/gcc/sparc-sun-solaris2.10/3.4.3/crt1.o         55
  /usr/ccs/lib/libc.a                                        108
  /usr/ccs/lib/libc.so                                       108
  /lib/libc.so.1                                             110
```

```
  /platform/SUNW,UltraAX-i2/lib/libc_psr.so.1                138
  /var/ld/ld.config                                          138
```

This file names appear to be related to the GNU C compiler. This is a lot of open calls for just one invocation of the C compiler. I wonder how many times the compiler is being invoked. Let's check that out (exc.d):

```
syscall::exec*:return
{
   @[execname] = count();
}
```

Running this script for five seconds on my busy system produced the following output (make sure you wait until after DTrace tells you how many probes were matched before you start the 5 second count):

```
  cc1                                                        30
  collect2                                                   30
  gas                                                        30
  gcc                                                        30
  ld                                                         30
```

This is a lot of calls to the C compiler for a 5-second period. We need to find out who is initiating all this activity. The easiest way to do that is to list the execs and who is making them (exlist.d):

```
#pragma D option quiet
syscall::exec*:entry
{
   self->exn = execname;
}
syscall::exec*:return
/ self->exn != "" /
{
   printf ("%s -> %s\n", self->exn, execname);
   self->exn = 0;
}
```

Running this D program made it clear who was starting all the compiles:

```
callfast -> gcc
gcc -> cc1
gcc -> gas
gcc -> collect2
collect2 -> ld
callfast -> gcc
gcc -> cc1
gcc -> gas
gcc -> collect2
collect2 -> ld
...
```

A process named callfast is calling the C compiler repeatedly. In fact, callfast is a script that I wrote to do exactly that. The short processes generated by the C compiler were so short-lived, they weren't around long enough to make a significant impact on **prstat**. However, DTrace was able to find the problem easily.

Summary

There are a variety of problems that are difficult to solve using traditional tools and methods previously available. DTrace makes getting to the bottom of these issues much easier. In the next

and final part to this series, I'll show some other ways to use DTrace and tie up some loose ends about other options and probe providers.

Endnotes

1. You need to know that I cheated a little here in order to get consistent results. The first time you run a command that reads a file, the file gets read into cache. Every time you read the file after that, the data comes from the cache rather than disk. Obviously, this would taint the results. To compensate for this, I wrote a C program to allocate enough heap space to use all available real memory. (I don't advise using this technique on a production system. Using up that much swap space could impact other applications.) Before a timed run, I ran this program to flush the file cache:

```
main()
{
   int i = 0;
   while ((char *)malloc(1024*1024) != 0)
   {
      printf ("%d ", ++i);
   }
   exit(0);
}
```

You don't actually need to run the program until it exits (runs out of virtual memory). Once it starts allocating very slowly, you've invalidated all of the real memory pages that were being used for file cache.

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

3. In a future release of Solaris, there will be options to change the sort order for count. These options are already available in Nevada, the OpenSolaris release (see **http://opensolaris.org**).

4. Before you get all excited about memory mapping being so much better than regular file reads, you need to know that the designers of Solaris are aware of this. That is why Solaris already implements a regular file read using the system paging algorithms. Memory mapping the file uses a little less system time, but the IO for both ways is just as efficient.

*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).*