Learning DTrace -- Part 1

Chip Bennett

One of the most innovative features of Solaris 10 is a facility that allows administrators to dynamically track down problems and performance issues on an active production system with almost no impact on that system. This facility is called DTrace, which stands for Dynamic Tracing. This is the first of a series of articles designed to help you understand how DTrace works and give you the basic knowledge necessary to use DTrace to solve your own system problems.

If you're familiar with C programming and awk programming, you already know half of what you need to know to run DTrace. DTrace uses a scripting language called D, which has a similar syntax to C, except that it doesn't support flow control (if, while, for, etc.). Because of the nature of system tracing, the DTrace framework has its own flow, which is tailored using a pattern match syntax that is much like awk.

DTrace does its job by allowing you to insert probes (also called "instrumenting" probes) at specific points in the kernel or user program. When these probes are reached, data is recorded that provides information regarding the event. The information provided varies according to the type of probe. You tell DTrace, through the D language, where you want probes instrumented and how you want the event data reported.

Listing Probes

Before actually writing scripts, the simplest way to start looking at DTrace is by using it from the command line. The simplest command option in DTrace is the one that lists all of the possible probes than can be instrumented on your system:

```
# dtrace -l
   ID   PROVIDER     MODULE              UNCTION  NAME

    1     dtrace                                 BEGIN
    2     dtrace                                 END
    3     dtrace                                 ERROR
    4    syscall                           nosys  entry
    5    syscall                           nosys  return
    6    syscall                           rexit  entry
    7    syscall                           rexit  return
    8    syscall                          forkall  entry
    9    syscall                          forkall  return
  ...
38328        fbt       zmod          huft_build  return
38329        fbt       zmod               _info  entry
38330        fbt       zmod               _info  return
38331        fbt       zmod          z_strerror  entry
38332        fbt       zmod          z_strerror  return
38333        fbt       zmod        z_uncompress  entry
38334        fbt       zmod        z_uncompress  return
```

Each probe is characterized by a four-part tuple: provider, module, function, and name. The above list shows the four parts of each probe, plus the unique probe identification number. Most of the above list is removed for brevity, but if you look at the last line, you'll see that my system has 38,334 available probes. Your mileage may vary.

The probe provider is a kernel program that performs a specific type of instrumentation. For example, the "syscall" provider provides probes into the kernel system call interface (read, write, fork, etc.). In this case, there are two possible probes for each system call: one for entry into the system call, and one for exit from the system call. Another provider, which instruments in a completely different way, is the "profile" provider. This provider allows for probes that record events at specific time intervals. You'll see more about the "profile" provider later.

The other parts of the probe tuple further define the location of the probe:

- The "module" where the probe is located (if the probe corresponds to a program location).
- The "function" within the module.
- The "name" that generally defines what event the probe is recording, such as "entry" or "return".

Some probes don't have the "module" part of the tuple, and some additionally don't have the "function" part. However, all probes have at least a "provider" and a "name".

We can further define which probes we want to deal with by specifying a colon-delimited tuple. For example, the tuple "lockstat:genunix:mutex_exit:adaptive-release" refers to a probe named "adaptive-release", which is in the "mutex_exit" function in the "genunix" module and is provided by the "lockstat" provider. To list such a probe, we could specify:

```
dtrace -ln lockstat:genunix:mutex_exit:adaptive-release
   ID   PROVIDER       MODULE         FUNCTION  NAME
  470   lockstat      genunix      mutex_exit  adaptive-release
```

The **-n** flag allows us to specify a tuple by name, that is, DTrace assumes that the last component of the tuple specified is the name component.

There are three other flags that allow us to specify the other components as the last one in the tuple:

```
# dtrace -lf lockstat:genunix:mutex_exit
   ID   PROVIDER       MODULE        FUNCTION  NAME
  470   lockstat      genunix      mutex_exit  adaptive-release

# dtrace -lm lockstat:genunix
ID   PROVIDER          MODULE        FUNCTION  NAME
  467   lockstat      genunix     mutex_enter  adaptive-acquire
  468   lockstat      genunix     mutex_enter  adaptive-block
  469   lockstat      genunix     mutex_enter  adaptive-spin
  470   lockstat      genunix      mutex_exit  adaptive-release
  471   lockstat      genunix   mutex_destroy  adaptive-release
...

# dtrace -lP lockstat
   ID   PROVIDER       MODULE        FUNCTION  NAME
  467   lockstat      genunix     mutex_enter  adaptive-acquire
  468   lockstat      genunix     mutex_enter  adaptive-block
  469   lockstat      genunix     mutex_enter  adaptive-spin
  470   lockstat      genunix      mutex_exit  adaptive-release
  471   lockstat      genunix   mutex_destroy  adaptive-release
...
```

When we only partially specify a tuple in this way, we generally match more than one probe. Also, note that specifying all of the lockstat probes is the same as specifying all of the lockstat:genunix probes, because in the case of lockstat, there is only one module.

We can also match more than one probe by using the colon as a placeholder and leaving out one or more parts of the tuple. For example, the following are equivalent probe specifications and will match the same set of probes:

```
dtrace -lP lockstat
dtrace -lm lockstat:
dtrace -lf lockstat::
dtrace -ln lockstat:::
```

These two **dtrace** commands will match all of the available probes:

```
dtrace -l
dtrace -ln :::
```

Instrumenting Probes

Although it's useful to be able to list the probes in which we have interest, it's even more useful to actually instrument them. To instrument the same probes that we have been listing, simply remove the **-l** flag from the command line. For example, the following commands all instrument the same set of probes -- all of the probes provided by lockstat:

```
dtrace -P lockstat
dtrace -m lockstat:
dtrace -f lockstat::
dtrace -n lockstat:::
```

When probes are instrumented, the default behavior is to write (to standard out) as each probe fires, the CPU# where the probe fired, the ID# of the probe, the probe function, and the probe name:

```
# dtrace -P lockstat | head
dtrace: description 'lockstat' matched 22 probes
CPU     ID                    FUNCTION:NAME
  0    470     mutex_exit:adaptive-release
  0    470     mutex_exit:adaptive-release
  0    467    mutex_enter:adaptive-acquire
  0    470     mutex_exit:adaptive-release
  0    467    mutex_enter:adaptive-acquire
  0    470     mutex_exit:adaptive-release
  0    477           lock_try:spin-acquire
  0    478         lock_clear:spin-release
  0    473           lock_set:spin-acquire
```

This **dtrace** session instruments all of the probes provided by lockstat (22 probes). As each probe fires, a line of output is generated. At this point, the purpose of the lockstat provider should be pretty clear: provide a way of tracing lock events (mutexes and spinlocks) in the kernel. The trace above is listing each lock event as it occurs. (Note that if we had not piped the **dtrace** command to head, it would have run forever and would have had to be interrupted with a **kill** command or control-C.)

DTRACE HINT: When running your own DTrace commands and scripts, it's important not to overload the system with too many instrumented probes. You can actually instrument all of the probes on a system and get away with it, but it's rather CPU intensive, plus you end up with a lot of dropped events as the event queues fill up. (DTrace tells you when it drops events.) Most of the time, if you are instrumenting thousands of probes, you are probably collecting too much information. An easy way to safeguard against this is to always run your **dtrace** command or script the first time with the **-l** flag. This will only *list* the probes you are instrumenting, rather than actually instrument them. If you want a count of the number of probes you would have instrumented, pipe the output of the list form of your DTrace command to **wc -l** and subtract one for the header.

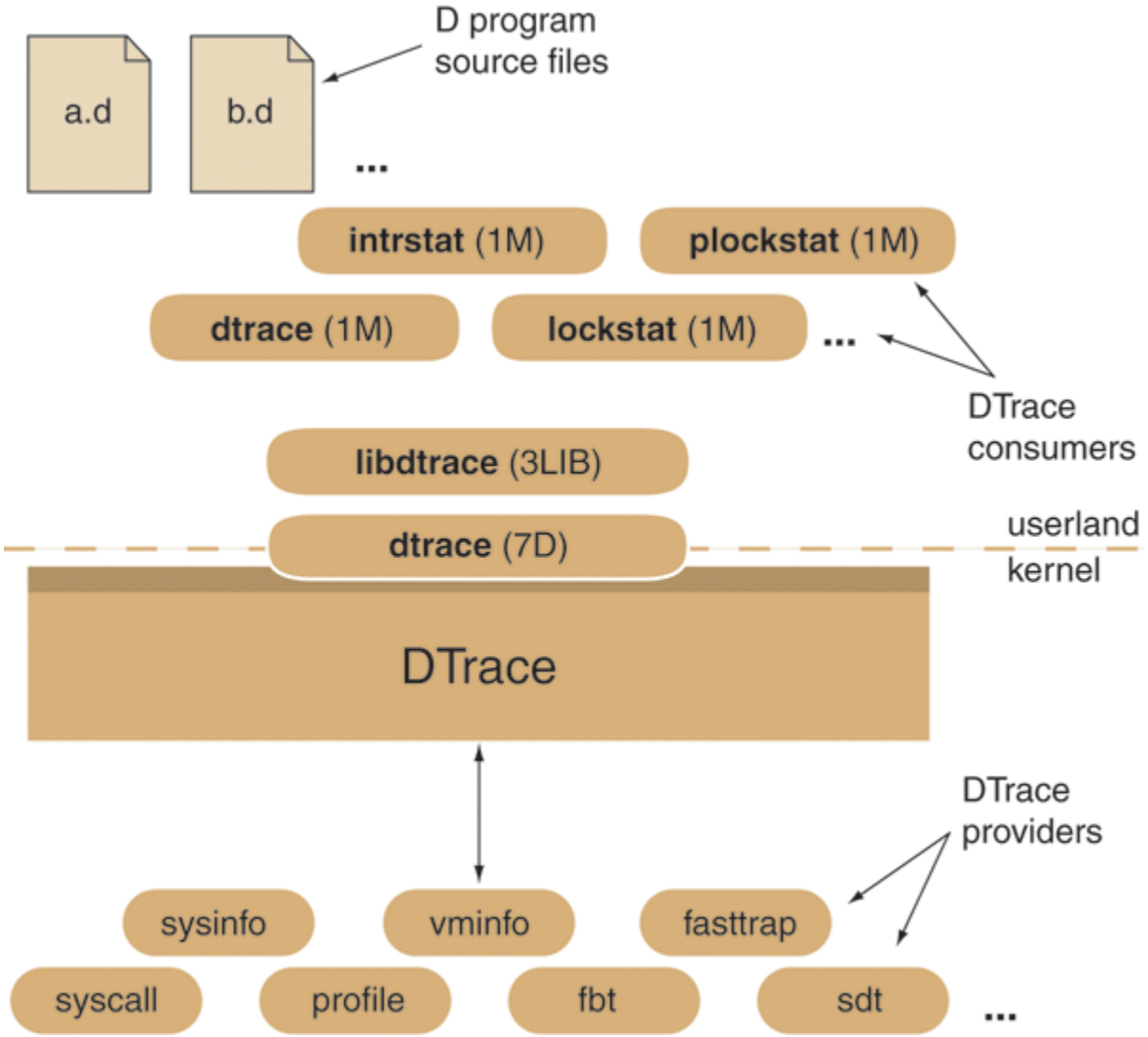
Inside DTrace

At this point, you may be asking how Solaris does all this without impacting the rest of the system [1]. The DTrace facility is a collection of producers and consumers. The DTrace probes, when they fire, produce events that are placed into buffers (one per processor per DTrace

consumer). The DTrace consumers (i.e., the dtrace command) pull events out of the buffers and process them. Figure 1 shows the relationship beteen the DTrace producers and consumers [2]. For additional information on the internals of DTrace, see "Dynamic Instrumentation of Production Systems" by Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal: http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf

Since you only instrument the probes of interest, the amount of data being collected is kept at a minimum. Additionally, DTrace limits the number of events that have to be post-processed, using predicates, which are relational expressions based on data from the probes.

We're not limited to the default action of writing the CPU, probe ID, function, and name. We can tailor the actions that occur when a probe fires and perform different actions based on which probes fire and the contents of the event data. These actions make use of the D language, which I'll cover in part two of this series.



Solaris Dynamic Tracing Guide", Sun Microsystems, Inc., Part No. 817-6223-10

Summary

In part 1 of this series, we've only looked at the preliminaries of using DTrace. In segments to come, I'll explain the D programming language and show you how to use DTrace to solve system problems. DTrace allows you to peer a lot deeper into a running system than previous tools allowed. By the time we're finished with this series, I think you'll agree that DTrace is indeed one of the most innovative features of Solaris 10.

Endnotes

1. Being a fan of "Star Trek"(R), I enjoy the correlation between "system monitoring impact" and "transporter technology". Werner Heisenberg (1901-1976), renowned physicist, proposed what is referred to as the Heisenberg Uncertainty Principle. In the area of system monitoring, the principle loosely defines how the monitor impacts the system. That is, the more in depth and detailed you monitor and describe what is happening on your system, the more the monitor itself changes and impacts the system.

The fun correlation to "Star Trek" is that the principle is also used to describe the impact of transporter technology on the thing being transported: the more you scan to copy an object, the more you disrupt it (see "Quantum Teleportation": **http://www.research.ibm.com/quantuminfo/teleportation/**). Of course, because the show was based on a certain amount of real science, the engineers in "Star Trek" had at their disposal a device called the "Heisenberg Compensator", which not only added a safety mechanism to the transporter, but also gave the writers something accurate sounding to toss around in the script (i.e., "we had to modify the Heisenberg Compensator", or "the transporter failed because of a problem with the Heisenberg Compensator"). ("Star Trek" is a registered trademark of CBS Studios Inc.)

2. From "Solaris Dynamic Tracing Guide", Sun Microsystems, Inc., Part No. 817-6223-10 -- **http://docs.sun.com/app/docs/doc/817-6223/6mlkidlf1?a=view**

*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).*