Application Development Framework
Application Express
Big Data
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
.NET
Dynamic Scripting Languages
Embedded
Digital Experience
Enterprise Architecture
Enterprise Management
Identity & Security
Java
Linux
Service-Oriented Architecture
Solaris
SQL & PL/SQL
Systems - All Articles
Virtualization

Using DTrace on Oracle Linux

by Richard Friedman

An introduction to DTrace probes and providers for Oracle Linux and how they are different from those in Oracle Solaris. Also covers DTrace commands and scripting.

Published June 2013

DTrace is a comprehensive dynamic tracing facility originally developed for the Oracle Solaris operating system, and it is now available to Oracle Linux customers. DTrace is designed to give operational insights that allow users to tune and troubleshoot the operating system and applications dynamically in real time. DTrace provides Oracle Linux developers with a tool to analyze performance and track down performance problems across the software stack. DTrace enables higher quality applications development, reduced downtime, lower cost, and greater utilization of existing resources. It is available for download from the Unbreakable Linux Network for Oracle Linux Support customers.

Getting Started with DTrace

DTrace is a comprehensive dynamic tracing facility that was first developed for the Oracle Solaris operating system, and subsequently ported to Oracle Linux. DTrace allows you to explore the running system and track down performance problems across many layers of the software stack.

For a little color and background about this article, see the accompanying blog on the OTN Garage. To comment or discuss, join us on Facebook.

Some of the advantages of using DTrace on Oracle Linux are that you can

Observe dynamic runtime performance across the software stack, OS kernel, system libraries, and applications.
Identify performance bottlenecks by defining live probe points at runtime.
Develop scripts that are executed when probes fire under predicate control.
Detect and report memory access errors instead of crashing the system.
This port of DTrace for the Oracle Linux Unbreakable Enterprise Kernel is still evolving technology. The goal is to provide an integrated solution to dynamic tracing that ultimately leads to functional compatibility with Oracle Solaris while bringing new features to Oracle Linux.

A Quick Look

To use DTrace, you specify locations of interest in the kernel, called *probes*, where DTrace can bind a request to perform a set of actions, such as recording a stack trace, a time stamp, or the argument to a function. Probes function like programmable telemetry sensors deep in the OS that record information. When a probe is triggered, DTrace gathers data from it and reports the data back to you. Oracle Linux DTrace probes live within DTrace kernel modules called *providers* that perform the instrumentation to enable the probe.

DTrace is safe to use in production systems and does not require restarting the system or any running applications. Runtime tracing can be invoked using the `dtrace`(1) command or with the DTrace D scripting language. Either way, you can query the system probes to provide immediate, concise answers to arbitrary questions that you formulate.

For example, Listing 1 shows a simple command-line `dtrace` call that names the commands for invoking the `open()` systems call and the files being opened (during a console login process):

```
# dtrace -q -n syscall::open:entry'{ printf("%-16s%-16s\n", execname,copyinstr(arg0)); }'
login           /etc/ld.so.cache
login           /lib64/libpam.so.0
login           /lib64/libpam_misc.so.0
login           /lib64/libselinux.so.1
login           /lib64/libaudit.so.1
login           /lib64/libc.so.6
login           /lib64/libdl.so.2
login           /lib64/libcrypt.so.1
login           /lib64/libfreebl3.so
login           /dev/tty1
login           /etc/pam.d/login
login           /lib64/security/pam_securetty.so
login           /etc/pam.d/system-auth
login           /lib64/security/pam_env.so
login           /lib64/security/pam_fprintd.so
login           /etc/ld.so.cache
login           /usr/lib64/libdbus-glib-1.so.2
login           /lib64/libdbus-1.so.3
login           /lib64/libpthread.so.0
login           /lib64/librt.so.1
login           /lib64/libgobject-2.0.so.0
login           /lib64/libglib-2.0.so.0
login           /lib64/security/pam_unix.so
login           /etc/ld.so.cache
login           /lib64/libnsl.so.1
login           /lib64/security/pam_succeed_if.so
login           /lib64/security/pam_deny.so
login           /lib64/security/pam_nologin.so
login           /etc/pam.d/system-auth
login           /lib64/security/pam_localuser.so
login           /lib64/security/pam_permit.so
login           /etc/pam.d/system-auth
login           /lib64/security/pam_cracklib.so
login           /etc/ld.so.cache
login           /usr/lib64/libcrack.so.2
login           /lib64/security/pam_selinux.so
login           /lib64/security/pam_loginuid.so
login           /lib64/security/pam_console.so
login           /lib64/security/pam_namespace.so
login           /lib64/security/pam_keyinit.so
login           /etc/pam.d/system-auth
login           /lib64/security/pam_limits.so
login            /lib64/security/pam_ck_connector.so
login            /usr/lib64/tls/x86_64/libck-connector.so.0
login            /usr/lib64/tls/libck-connector.so.0
login            /usr/lib64/x86_64/libck-connector.so.0
login            /usr/lib64/libck-connector.so.0
login            /etc/pam.d/other
login            /etc/nsswitch.conf
login            /etc/ld.so.cache
login            /lib64/libnss_files.so.2
login            /etc/passwd
login            /etc/securetty
dbus-daemon      /selinux/access
dbus-daemon      /proc/1464/cmdline
dbus-daemon      /proc/1464/cmdline
dbus-daemon-lau  /etc/ld.so.cache
dbus-daemon-lau  /lib64/libexpat.so.1
dbus-daemon-lau  /lib64/libpthread.so.0
dbus-daemon-lau  /lib64/librt.so.1
dbus-daemon-lau  /lib64/libc.so.6
dbus-daemon-lau  /etc/dbus-1/system.conf
dbus-daemon-lau  /etc/nsswitch.conf
dbus-daemon-lau  /etc/ld.so.cache
dbus-daemon-lau  /lib64/libnss_files.so.2
dbus-daemon-lau  /etc/passwd
dbus-daemon-lau  /usr/share/dbus-1/system-services/net.reactivated.Fprint.service
dbus-daemon-lau  /etc/passwd
dbus-daemon-lau  /proc/sys/kernel/ngroups_max
dbus-daemon-lau  /etc/group
^C
```

Listing 1

Note that this is happening in real time on the running system. The trace is terminated by pressing Control + C.

The DTrace scripting language, D, is designed specifically for dynamic tracing. With D scripts, you can dynamically turn on probes, collect the data, and process it. D scripts can be shared with others, and many are published on the Oracle Technology Network.

In the example in Listing 2, the D language script `syscalls.d` is invoked to display the system calls process 5178 is using and their frequency:

```
# cat syscalls.d
#!/usr/sbin/dtrace -qs
syscall:::entry
/pid == $1/
{
  @num[probefunc] = count();
}
# ./syscalls.d 5178
^C
  chdir                                                             2
  clone                                                             2
  pipe                                                              2
  rt_sigreturn                                                      2
  setpgid                                                           2
  access                                                            4
  getegid                                                           4
  geteuid                                                           4
  getgid                                                            4
  getuid                                                            4
  wait4                                                             4
  sendmsg                                                           5
  socket                                                            5
  close                                                             9
  newstat                                                          14
  read                                                             23
  write                                                            29
  ioctl                                                            40
  rt_sigaction                                                    120
  rt_sigprocmask                                                  136
```

Listing 2

Probes and Providers

DTrace probes come from a set of kernel modules called providers, each of which performs a particular kind of instrumentation to create probes. When you use DTrace, each provider is given an opportunity to publish the probes it can provide to the DTrace framework. You can then enable and bind your tracing actions to any of the probes that have been published.

To list all of the available probes on your system, type the command shown in Listing 3. You might observe a different list on your system, because the number of probes varies depending on your Oracle Linux platform, the software you have installed, and the provider modules that you have loaded.

```
# dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4    syscall                                                read entry
    5    syscall                                                read return
    6    syscall                                               write entry
    7    syscall                                               write return
    8    syscall                                                open entry
    9    syscall                                                open return
   10    syscall                                               close entry
   11    syscall                                               close return
   12    syscall                                             newstat entry
   13    syscall                                             newstat return
   14    syscall                                            newfstat entry
   15    syscall                                            newfstat return
   16    syscall                                            newlstat entry
   17    syscall                                            newlstat return
   18    syscall                                                 poll entry
   19    syscall                                                 poll return
   20    syscall                                                lseek entry
   21    syscall                                                lseek return
   22    syscall                                                 mmap entry
   23    syscall                                                 mmap return
   24    syscall                                             mprotect entry
   25    syscall                                             mprotect return
   26    syscall                                               munmap entry
   27    syscall                                               munmap return
   28    syscall                                                  brk entry
...
#
```

Listing 3

Each probe is shown with an integer ID and a human-readable name, which is composed of four parts, shown as separate columns in the `dtrace` output.

`provider`—The name of the DTrace provider that is publishing this probe. The provider name typically corresponds to the name of the DTrace kernel module that performs the instrumentation to enable the probe.
`module`—If this probe corresponds to a specific program location, the name of the kernel module in which the probe is located.
`function`—If this probe corresponds to a specific program location, the name of the program function in which the probe is located.
`name`—The final component of the probe name, which is a name that gives you some idea of the probe's semantic meaning.
When referring to a specific probe, these parts are shown together, separated by colons, as in

`provider:module:function:name`

The providers shown in Table 1 are available in the current implementation of DTrace on Oracle Linux.

Table 1

Provider	Kernel Module	Description
`dtrace`	`dtrace`	Provides probes for DTrace itself: `BEGIN`, `END`, `ERROR`, which are used to optionally initialize DTrace before tracing begins, process after tracing ends, and handle unexpected errors in other probes during execution.
`io`	`io`	Provides monitoring probes related to data input and output.
`proc`	`proc`	Provides probes for monitoring process creation and termination, new program image execution, and sending and handling signals.
`profile`	`profile`	Provides probes associated with timed interrupts. These probes can be used to sample the system's state at fixed intervals.
`sched`	`sdt`	Provides probes related to CPU scheduling.
`sdt`	`sdt`	Provides statically defined tracing probes at several important locations of interest in the kernel.
`syscall`	`systrace`	Provides probes at the entry to and return from every system call. These probes are particularly useful for gaining insight into the interaction between applications and the underlying system.

Wildcards (`*` and `?`) can be used in the probe description, and blank fields are interpreted as wildcards. Here are a few examples.

Entry into the `open()` system call:

```
syscall::open:entry
```

Entry into any `open*()` system call, such as `open64()`:

```
syscall::open*:entry
```

Entry into any system library call:

```
syscall:::entry
```

All calls published by the `syscall` provider:

```
syscall:::
```

Differences Between DTrace in Oracle Linux and Oracle Solaris

Some DTrace providers from Oracle Solaris have not yet been implemented in Oracle Linux. For a current list of providers and any limitations on the Oracle Linux implementations, see the *Oracle Linux Administrator's Solutions Guide for Release 6*.

To list the probes that a specific provider publishes, use the following command:

```
# dtrace -l -P provider
```

To verify that a probe is available, use this command:

```
# dtrace -l -n probe_name
```

The `dtrace`(1) Command Line

As we've seen, many quite useful traces can be run directly from the `dtrace`(1) command line. These are called *one-liners* and many can be found in the documentation and on the internet.

For example, the one-liner shown in Listing 4 performs a directed kernel stack trace:

```
# dtrace -n 'gettimeofday:entry {stack()}'
dtrace: description 'gettimeofday:entry' matched 1 probe
CPU ID 0 196
0 196
...
                FUNCTION:NAME
           gettimeofday:entry
vmlinux`pollwake
vmlinux`dtrace_stacktrace+0x30
vmlinux`__brk_limit+0x1e1832d7
vmlinux`__brk_limit+0x1e1913a1
vmlinux`pollwake
vmlinux`do_gettimeofday+0x1a
vmlinux`ktime_get_ts+0xad
vmlinux`systrace_syscall+0xde
```

```
vmlinux`audit_syscall_entry+0x1d7
vmlinux`system_call_fastpath+0x16
           gettimeofday:entry
vmlinux`dtrace_stacktrace+0x30
vmlinux`__brk_limit+0x1e1832d7
vmlinux`__brk_limit+0x1e1913a1
vmlinux`security_file_permission+0x8b
vmlinux`systrace_syscall+0xde
vmlinux`audit_syscall_entry+0x1d7
vmlinux`system_call_fastpath+0x16
...
```
Listing 4

A complete list of all the `dtrace` command options can be found in the *Oracle Linux Dynamic Tracing Guide for Release 6*. Here are a few of the important option flags:

`-c` *`command`*—Run the specified command and exit upon its completion. If you specify more than one `-c` option, `dtrace` exits when all the commands have exited, and reports the exit status for each child process as it terminates.
`-f` *`function`*—Specify a function (optionally specifying the provider and module) that you want to trace or list. You can append an optional D-probe clause. You can specify the `-f` option multiple times to the command. Optionally, *`function`* can be in the form `[[`*`provider`*`:]`*`module`*`:]` *`function`* `[[`*`predicate`*`]`*`action`*`]`.
`-l`—List probes instead of enabling them. `dtrace` filters the list of probes based on the arguments to the `-f`, `-i`, `-m`, `-n`, `-P`, and `-s` options. If no options are specified, `dtrace` lists all probes.
`-m` *`module`*—Specify a module (optionally specifying the provider) that you want to trace or list. You can append an optional D-probe clause. You can specify the `-m` option multiple times to the command. Optionally, *`module`* can be in the form `[[`*`provider`*`:]`*`module`* `[[`*`predicate`*`]`*`action`*`]]`.
`-n` *`name`*—Specify a probe name (optionally specifying the provider, module, and function) that you want to trace or list. You can append an optional D-probe clause. You can specify the `-n` option multiple times to the command. Optionally, *`name`* can be in the form `[[[`*`provider`*`:]`*`module`*`:]` *`function`*`:]`*`name`* `[[`*`predicate`*`]`*`action`*`]]`.
`-p` *`PID`*—Grab a process specified by its process ID, cache its symbol tables, and exit upon its completion. If you specify more than one `-p` option, `dtrace` exits when all the processes have exited and reports the exit status for each process as it terminates.
`-P` *`provider`*—Specify a provider that you want to trace or list. You can append an optional D-probe clause. You can specify the `-P` option multiple times to the command. Optionally, *`provider`* can be in the form *`provider`*`['D-` *`probe_clause`*`']`.
`-s` *`sourcefile`*—Compile the specified D program source file and begin tracing.
A *predicate* is a logical expression in the D language that is used to conditionally trace data. A predicate appears optionally on the command line enclosed in slashes (`//`) with an action list of D statements enclosed in braces (`{ }`). (More about this in the next section when we talk about the D language.) However, the optional predicate and action text must be appropriately quoted when used on the command line to avoid interpretation by the shell. In Listing 4, the `-n` option flag is of the form `-n '`*`function`*`:`*`name`* `{`*`action`*`}'`.

Here are some examples of predicates.

True if the probe executes on cpu0:

```
cpu == 0
```

True if the PID of the process that fired the probe is 1029:

```
pid == 1029
```

True if the process is not the scheduler (`sched`):

```
execname != "sched"
```

True if the parent PID is not 0 and the first argument is 0:

```
ppid != 0 && arg0 == 0
```

If you only want to note that a particular probe fired on a particular CPU without tracing any data or performing any additional actions, you can specify an empty set of braces with no statements inside and DTrace will print just the name of the probe that fired.

Here are some examples of actions.

Print something using C-style `printf()` commands:

```
printf()
```

Print the user-level stack:

```
ustack()
```

Print a given variable:

```
trace
```

Scripting DTrace in D

DTrace scripts written in the D language consist of a set of clauses that describe probes to enable and predicates and actions to bind to these probes. D programs can also contain pragmas, declarations of variables, pointers, and arrays, and definitions of new types, much like a C program. And they can get pretty complex.

However, unlike a C or C++ program, but similar to a Java program, DTrace compiles your D program into a safe intermediate form that it executes when a probe fires. DTrace validates whether this intermediate form can run safely, reporting any runtime errors that might occur during the execution of your D program, such as dividing by zero or dereferencing invalid memory. As a result, you cannot construct an unsafe D program. You can use DTrace in a production environment without worrying about crashing or corrupting your system. If you make a programming mistake, DTrace disables the instrumentation and reports the error to you.

Every probe clause begins with a list of one or more *probe descriptions*, each taking the usual form *`provider`*`:`*`module`*`:`*`function`*`:`*`name`* followed by a predicate clause and a set of actions to be taken:

```
probe descriptions / predicate / {
  action statements
}
```

One major difference between D and other programming languages such as C, C++, and Java is the absence of control-flow constructs such as `if` statements and loops. D program clauses are written as single straight-line statement lists that trace an optional fixed amount of data. D does provide the ability to conditionally trace data and modify control flow using logical expressions called *predicates* that can be used to prefix program clauses.

Predicate expressions are enclosed in a pair of slashes (`//`) and are evaluated at probe-firing time prior to executing any of the statements associated with the corresponding clause. If the predicate evaluates to true, represented by any non-zero value, the statement list is executed. If the predicate is false, represented by a zero value, none of the action statements is executed and the probe firing is ignored.

Predicates are the primary conditional construct used for building more complex control flow in a D program. You can omit the predicate section of the probe clause entirely for any probe, in which case the actions are always executed when the probe fires.

Probe actions are described by a list of statements separated by semicolons (`;`) and enclosed in braces (`{}`). If you only want to note that a particular probe fired on a particular CPU without tracing any data or performing any additional actions, you can specify an empty set of braces with no statements inside.

DTrace scripts can be run either through the `dtrace` command directly by giving the path to the script in the `-s pathname` option or by creating a standalone script that begins with the invocation `#!/usr/sbin/dtrace -s`.

Listing 5 is DTrace script run using the `-s` option that displays the system calls made by the command specified in the `-c` option:

```
$ cat syscalls.d
syscall:::entry
/pid == $target/
{
        @[probefunc] = count();
}
$ dtrace -s syscalls.d -c date
dtrace: script 'syscalls.d' matched 296 probes
Thu May 23 00:40:52 CEST 2013

  access                                                            1
  arch_prctl                                                        1
  exit_group                                                        1
  getrlimit                                                         1
  lseek                                                             1
  rt_sigprocmask                                                    1
  set_robust_list                                                   1
  set_tid_address                                                   1
  write                                                             1
  futex                                                             2
  rt_sigaction                                                      2
  brk                                                               3
  munmap                                                            3
  read                                                              5
  open                                                              6
  mprotect                                                          7
  close                                                             8
  newfstat                                                          8
  mmap                                                             16
```

Listing 5

In the `syscalls.d` script, the `$target` macro takes on the value of the PID of the command (`/bin/date`), and the predicate traces only the system call entry probes that fire for that PID. There are many useful macros, such as `$target`, that give the script access to features of the running process.

The actions within the braces are executed only when the predicate evaluates to true. Actions enable your DTrace programs to interact with system outside of DTrace. The most common actions record data to a DTrace buffer.

DTrace provides a set of built-in functions for aggregating the data that individual probes gather. In Listing 5, the `count()` function returns the number of times the function has been called. DTrace stores the results of aggregation functions in objects called *aggregations* that are indexed using a tuple of expressions. In D, the syntax for an aggregation is as follows, where *`name`* is the user-defined name for the aggregation, *`keys`* is a comma-separated list of D expressions, *`aggfunc`* is one of the DTrace aggregation functions, and *`args`* is a comma-separated list of arguments appropriate for the aggregating function. The aggregation name is a D identifier that is prefixed with the special character `@`.

```
@name[ keys ] = aggfunc ( args );
```

Here are some aggregations provided by DTrace.

Returns the arithmetic mean of its arguments:

```
avg(expressions)
```

Returns the number of times the function has been called:

```
count()
```

Returns the standard deviation of its arguments:

```
stddev(expressions)
```

The D language is rich in C-style objects and syntax, and it allows the definition of variables, arrays, pointers, strings, structs, and unions. A library of subroutines adds to the functionality of the language. The complete language with examples is described in the *Oracle Linux Dynamic Tracing Guide for Release 6*.

Installing and Configuring DTrace on Oracle Linux
The DTrace package and the Unbreakable Enterprise Kernel (UEK) package are available on the Unbreakable Linux Network (ULN), but not on the public yum server. You must register your system with ULN before you can download the required packages. The Oracle Linux Administrator's Solutions Guide for Release 6 lists the packages you need to download from ULN and shows how to install them.

The *Administrator's Solutions Guide* also describes how to use the `modprobe` command to load the modules that support the DTrace probes that you want to use. For example, if you wanted to use the probes that the `proc` provider publishes, you would load the `systrace` module:

```
# modprobe systrace
```

The Value of DTrace
DTrace gives systems administrators and application developers the ability to watch the workings of the operating system and running applications from within, through system libraries and system calls, and deep into the kernel.

Use DTrace as a tool to discover and quantify the root cause of issues in the running system, even if the issue is internal to a kernel device driver or something else anywhere in the software stack. And, you can do this without disturbing the system, rebooting, or doing anything destructive. Consider it the Swiss Army knife in your toolkit.

See Also
Dynamic Tracing Guide for Oracle Linux Release 6 and the *Oracle Linux Administrator's Solutions Guide for Release 6*, both available from the

Oracle Linux 6 product documentation Website at http://docs.oracle.com/cd/E37670_01/
"Tutorial: DTrace by Example": http://www.oracle.com/technetwork/server-storage/solaris/dtrace-tutorial-142317.html
DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD, by Brendan Gregg and Jim Mauro; Prentice Hall 2011: http://www.dtracebook.com/index.php/Main_Page
DTrace forum: https://forums.oracle.com/forums/forum.jspa?forumID=1017

About the Author

Richard Friedman is a freelance technical writer developing content and documentation for various high-tech companies and organizations. Richard worked over 15 years for Sun Microsystems as lead documentation writer for compilers and developer tools. Long before that he was a staff systems and applications programmer at New York University Courant Institute, the Lawrence Berkeley Laboratory computer centers, and various software consulting organizations. He lives in Oakland, California.

Revision 1.0, 06/06/2013