Advanced Operating Systems:

DTrace Quick Start

Prof. Robert N. M. Watson

2023-2024

DTrace is a dynamic tracing tool originally developed by Sun Microsystems (now Oracle) available in Solaris, Mac OS X, Windows, FreeBSD, and as an optional add-on for Linux. A key design goal of DTrace is that it is always available, possible because it incurs no (serious) performance penalty when not being actively used. Simple tracing may be done using the command line; in practice, however, we write short scripts in DTrace’s D scripting language, which allow much more complex behaviours such as conditional or speculative tracing, data aggregation, and so on. This document provides a short glossary of DTrace terms, information on the dtrace command-line tool, a few sample DTrace scripts to get you started, and a list of commonly used probes and built-in variables.

DTrace principles

DTrace scripts consist of a series of statements identifying the probes, or instrumentation points, that should be enabled, an optional predicate that will further refine the set of behaviours to trace, and an action block that dictates what DTrace should do (e.g., trace) when a matching probe fires and is accepted by the predicate. Probe names include implicit wildcards, making it easy to trace, for example, all system calls, or all NFS client RPCs. Predicates allow environmental context or retained script state to control whether the action executes – e.g., whether the process name is sshd, or if the same thread has previously performed some action during the current system call. Actions can record values in variables, generate trace output, and various other activities that influence both future tracing and also, potentially, system behaviour. Your first DTrace script might read as follows:

```
BEGIN { printf("Hello world"); exit(0); }
```

This script matches the DTrace-internal BEGIN event, corresponding to script start, and has two statements in its action block: one, to print the string Hello world, and a second to terminate the script. A more enlightening script might trace system-call arguments to the open() system call by processes with the name sshd, using a predicate to ensure that only events triggered by sshd will be traced:

```
syscall::open:entry /execname == "sshd"/ { trace(copyinstr(arg0)); }
```

Notice that the system-call argument, arg0, is in fact a pointer to user memory, rather than an in-kernel string. The DTrace copyinstr() function allows the script to copy the string into kernel memory so that its contents can be traced.

DTrace command

The dtrace command-line tool provides a simple user interface to DTrace. It will generally be used in one of three forms: to list available probes, to specify simple probes, predicates, and actions on the command line, or with a script holding a set of probes, predicates, and actions originating in a file. Except for very simple investigations, you will almost always want to use the latter.

You will need to run dtrace as root; you can use su to switch to the root user if you have logged in as another user via SSH. Most filesystems in our teaching setup are mounted read-only in order to avoid wear on the flash, as well as discourage storing files on partitions that we might overwrite if issuing software updates during the module. You are encouraged to store trace data, and any other information of value, such as scripts you write, in /data, and to copy important data off the board to your notebook or workstation for analysis.
Listing available probes

To list all available probes:

dtrace -l

To list all probes in the syscall provider:

dtrace -l -P syscall

Specifying probes, predicates, and actions on the command line

The following examples use the -n argument, which specifies tracing by probe name. To trace return values (file descriptors) from the open system call:

dtrace -n 'syscall::open:return /errno == 0/ { trace(execname); trace(arg0); }'

To track the aggregate distribution of memory sizes requested via the kernel malloc function:

dtrace -n 'fbt:kernel:malloc:entry { @foo = quantize(arg0); }'

Count stack traces sampled using the 99Hz kernel profiling timer:

dtrace -n 'profile-99 { @foo[stack()] = count(); }'

Run a DTrace script named foo.d:

dtrace -s foo.d

It will sometimes be useful to specify the -q flag, for quiet output: this requests that all normal DTrace output be suppressed, with only explicit output from functions such as printf() being displayed.

DTrace scripts

The dtrace command also accepts a -s argument allowing a file to be specified as the source of the script to run. Scripts consist of a series of probes, predicates, and actions. Special probes, BEGIN and END, fire as the script starts up and terminates, and the function exit() can be used to terminate a script from an action.

Profiling callouts

Here is a sample script you can use to profile time spent in kernel callouts (timed asynchronous events). At the start of the callout, a per-thread variable cstart records the timestamp that execution begins. When the callout ends, the quantize aggregation is used to insert the difference between the current time and the start time into a histogram.

callout_execute:::callout-start
{
    self->cstart = vtimestamp;
}

callout_execute:::callout-end
{
    @length = quantize(vtimestamp - self->cstart);
}

Exercise: What would happen if we used timestamp or walltimestamp instead of vtimestamp?
Profiling callouts over a 1-second interval

This script extends our previous script to keep track of time spent in particular callouts, identified by their function name (arg0->c_func) over a one-second interval. Note the use of an associative array indexed by function name, the sum aggregation, and the use of a one-second timer tick to print the aggregation using printa() and clear to reset the array for the next interval.

```d
#pragma D option quiet

callout_execute:::callout-start
{
    self->cstart = vtimestamp;
}

callout_execute:::callout-end
{
    @callouts[{(struct callout *)arg0)->c_func} = sum(vtimestamp - self->cstart);
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
    printf("\n");
}

BEGIN
{
    printf("%40s | %s\n", "function", "nanoseconds per second");
}
```

Exercise: How would we extend this script to not just add up all time spent by a particular callout function, but instead record a histogram of the execution times of each function over each second?

Collect stack traces to privilege check failures

This script counts the number of unique stack traces leading to privilege failures, detected using the priv provider.

```d
priv::priv_check:priv-err
{
    @traces[stack()] = count();
}
```

Exercise: How would we modify this script to instead keep a count of privilege-check failures by system call?

DTrace providers

The following DTrace providers are available on most FreeBSD systems; others may be available when specific modules are loaded. To list probes available from a provider, use dtrace -l -P provider, where provider is the provider name.

- **callout_execute** The kernel’s callout (timer) subsystem; used to schedule asynchronous (and often recurring) events such as I/O timeouts and TCP retransmission. The two probes are callout-start and callout-end.
- **dtmalloc** DTrace kernel memory-allocation provider, which exposes probes for allocation and freeing of various memory types. The per-type probe names are malloc and free. Note that this provider does not cover UMA (slab) allocations, only those by the general kernel malloc.
DTrace provides a large number of built-in variables that can be referenced by scripts; a detailed list can be found in the DTrace documentation (see *Advanced Operating Systems: Readings*).

**arg0..arg9, args[]** Arguments to the DTrace probe: for FBT entry, the function’s actual arguments; for FBT return, its return value as arg0; for system calls, likewise arguments and return value; for other probes, typically data-structure pointers such as a pointer to the pertinent callout structure for callout_execute probes. Use arg0..arg9 for untyped integer values, and args[] for typed structures and pointers that can be dereferenced.

**caller** The program counter where the probe fired.

**cpu** The CPU number that the probe fired on.

**cwd** The current thread’s working directory.

**errno** The error value of the last system call executed by the thread.
**execname** The name of the executing binary.

**pid** The process ID of the current process.

**probefunc** The function name portion of the current probe’s description.

**probemod** The module name portion of the current probe’s description.

**probename** The name portion of the current probe’s description.

**tid** The thread ID of the current thread.

**timestamp** The current time in nanoseconds, from an arbitrary point.

**vtimestamp** The current virtual time of the thread, in nanoseconds, from an arbitrary point.

**walltimestamp** The number of nanoseconds since Epoch.

**DTrace glossary**

**provider** DTrace providers implement classes of events that can be instrumented using DTrace; for example, the Function Boundary Tracing provider allows the prologues and epilogues of all kernel functions to be instrumented, and the System Call provider similarly allows all system-call entry and return events to be instrumented.

**probe** A probe is an event that can be instrumented and traced – e.g., a particular function calling or returning, the profiler firing, a specific named system call entering. Probes have uniquely identifying names in the form `provider:module:function:name`. Fields in probe names may be omitted to match multiple probes.

**module** Some DTrace providers further include the name of a kernel module or component in the module field, making it easier to narrow down the set of probes to a particular subsystem. For example, the Network File System provider exposes the NFS version number as the module component of its probe names, and the System Call provider includes the ABI name.

**function** DTrace probes likewise include a function name that further identifies the probe that will fire; in the case of the Function Boundary Tracing provider, the function will be the name of the function instrumented, whereas the actual probe name will be `entry` or `return`.

**consumer** A DTrace consumer is any program that interacts with DTrace to monitor system behaviour. For our purposes, this is the `dtrace` command-line tool, but others also exist, such as the `lockstat` tool.

**predicate** DTrace predicates allow a script to inspect the run-time environment to make a determination as to whether an action should be executed as a result of a probe firing. This allows tracing of an event to be predicated on environmental factors such as what process is executing or prior events that have occurred in the same system call.

**action** DTrace script actions are sequences of tracing operations to perform when a probe fires and is accepted by a predicate. This might include writing data to a trace buffer, but can also include updating variables or, for certain classes of scripts, changing the execution of the kernel.

**variables** DTrace variables allow state to be recorded and acted on within scripts; variables can be global, affecting all instances of execution predicates and actions, thread-local (self), visible only to the current thread, and clause-local (this), visible only to the current action.

Thread-local variables are especially useful, as they can be used to record sequential operations by a thread that determine later tracing activities. For example, a per-thread variable might be used to store the name of the system call in execution so that later, when other operations are recorded, the system-call name is available to be included in a trace entry, or a decision to trace information might be conditioned on which system call is in flight.
**scalar variables**  *Scalar variables* are simple values associated with names specified by the programmer. Scalars can hold values of various types, including integers, pointers, and strings, but also instances of data structures originating in the program being traced. For example, \( x = 5 \) or `self->syscall = probefunc`.

**associative arrays**  *Associative arrays* allow sets of keys and associated values to be stored in a single variable. For example, `self->functions[probefunc]++` or `functions[execname, probefunc]++`.

**built-in variables**  DTrace includes a number of *built-in variables* that hold information about the context for a probe, such as the current process ID, processor ID, executable name, user ID, etc.

**external variables**  It is possible for a DTrace script to refer to kernel variables, which can be done by prefixing the variable name with the `.` character. For example, `bootflags`.

**aggregations**  *Aggregations* are special global variables that efficiently collect and reduce large volumes in data in a scalable way. For example, the `@count` aggregating function allows counters of events to be efficiently implemented in the presence of multiple cores: each core maintains its own instance of the counter, avoiding locking when updating the counter on any particular core, and DTrace will combine those per-core values into a single global value before presenting it to the user. Other aggregating functions include `sum`, `avg`, `min`, `max`, `lquantize`, and `quantize`. The latter two aggregations record distribution information, allowing histograms to be displayed by DTrace.

**speculation**  DTrace scripts may include *speculation*, which allows a set of operations to be performed conditional on some later event that will cause them to *commit* (or *abort*). For example, a script might speculatively collect argument data during a system call, but only commit those recorded arguments to the trace if the system call returns a specific error.

**functions**  DTrace *functions* perform a variety of activities, including appending to the trace (`trace()`), printing out values using format strings (`printf()`, or aggregations using `printa()`), or gathering stack traces (`stack()`).

**destructive scripts**  Most DTrace scripts will simply inspect execution; however, it is also possible to modify kernel execution – e.g., to change scheduling, or overwrite in-memory values. Such scripts are normally used in testing – e.g., to trigger kernel edge cases that are otherwise hard to reach. This includes stopping the running process, sending signals, overwriting memory in user processes, or panicking the kernel to allow further debugging using a kernel debugger. For the purposes of this course, use of destructive scripts is not recommended, as incautious use can easily lead to system crashes or data corruption.

### Further information

See the module reading list for further information; the `dtrace` man page, FreeBSD handbook, and World Wide Web are all useful reference material. The DTrace book by Gregg and Mauro has a more tutorial-like structure, and is strongly recommended.