# The DTrace One-Liner Tutorial

This teaches you DTrace for FreeBSD in 12 easy lessons, where each lesson is a one-liner you can try running. This series of one-liners introduces concepts which are summarized as bullet points. For an online DTrace reference, see the DTrace Guide, which contains a longer tutorial in Chapter 1.

Contributed by Brendan Gregg (2014), primary author of "DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD" (Prentice Hall, 2011).

**Lesson 1. Listing Probes**

```
dtrace -l | grep 'syscall.*read'
```

"dtrace -l" lists all probes.

- A *probe* is an instrumentation point for capturing event data.
- grep(1) can be used with "dtrace -l" to find probes. There are more precise ways to do this (eg, dtrace -n 'syscall::*read*:entry').
- The five fields printed are: DTrace probe ID (ignore it), provider name (a library of related probes), module name (if set), function name, and probe name.
- Pay attention to the provider name and the probe name: these two fields form a stable API. When possible, you'll only use these two fields when specifying probes.

**Lesson 2. Hello World**

```
dtrace -n 'dtrace:::BEGIN { printf("Hello FreeBSD!\n"); }'
```

This prints a welcome message. Run it, then hit Ctrl-C to end.

- Probes are specified by the four columns seen earlier, colon-separated, and where blank fields are wildcards.
- dtrace:::BEGIN is a special probe which fires once when dtrace begins. You can use it to set variables and print headers.
- An *action* can be associated with probes, in { }. This example calls printf() when the probe fires.

**Lesson 3. File Opens**

```
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Now we're tracing file opens as they happen, showing the process name and pathname.

- execname: this is a *builtin variable* that has the current process's name. Other builtins include pid, tid, ppid.
- arg0: this is the first probe argument, the meaning of which is documented by each provider chapter in the DTrace Guide. For the syscall provider entry probe, the syscall arguments can be traced as arg0, arg1, ..., argN, which are all unsigned int's (uint64_t).
- The open() syscall arguments are: const char *pathname, int flags, mode_t mode (see the open(2) man page). So, arg0 is the pathname pointer.
- copyinstr(): this pulls a user-level string into the kernel, where DTrace can read it. Since arg0 is just a number (unsigned int), DTrace needs us to say it's a user-level string.

**Lesson 4. Syscall Counts By Process**

```
dtrace -n 'syscall:::entry { @[execname, probefunc] = count(); }'
```

This summarizes syscalls by process name and syscall type, printing a report on Ctrl-C.

- @: This denotes a special variable type called an *aggregation*, which provide a powerful and efficient way to summarize data. You can add an optional variable name after the @, eg "@num", either to improve readability, or to differentiate between more than one aggregation.
- []: The optional brackets allow a key to be set for the aggregation, much like an associative array.
- probufunc: This is a builtin variable that refers to the function field of the probe name. For the syscall provider, this is the name of the system call.
- count(): This is an *aggregation function* – the way it is populated. count() counts the number of times it is called. Since this is saved by execname, the result is a frequency count of system calls by process name.
- Aggregations are automatically printed when dtrace ends (eg, via Ctrl-C). They can be explicitly printed using printa(), which allows their presentation to be customized.

**Lesson 5. Distribution of read() Bytes**

```
dtrace -n 'syscall::read:return /execname == "sshd"/ { @ = quantize(arg0); }'
```

This summarizes the return value of read() syscalls, printing it as a histogram.

- //: This is a *predicate*, which acts as a filter for the action. The action is only executed if the predicate expression is true, in this case, only for processes named "sshd". Boolean operators are supported ("&&", "||").
- arg0: On the syscall return probe, this has the return value of the system call. For read(), this is either -1 (error) or the number of bytes successfully read.
- @: This is an aggregation similar to the previous lesson, but without any keys ([]) this time.
- quantize(): This is an aggregation function which summarizes the argument as a power-of-2 histogram. On the output, the "value" column shows how many events were at least this size, and the "count" column shows how many events fell into this bucket range. The histogram can be used to study multi-modal distributions.
- Other aggregation functions include lquantize() (linear quantize), avg() (average), min(), and max().

**Lesson 6. Timing read() Syscall**

```
dtrace -n 'syscall::read:entry { self->ts = timestamp; } syscall::read:return /self->ts/ {
    @ = quantize(timestamp - self->ts); self->ts = 0; }'
```

Summarize the time spent in read(), in nanoseconds, as a histogram.

- self->: This denotes a *thread-local variable* – one that is stored with the current thread. In this case, it's used to store the start time of the syscall in a variable named "ts", which can be referenced in a later probe action for the same thread.
- timestamp: Nanoseconds since boot. This is a high resolution timestamp counter than can be used to time events.
- /self->ts/: This predicate checks that "ts" not NULL/0, ensuring that it was populated by an entry probe with a meaningful value, so that the later delta calculation (timestamp - self->ts: ie, now minus start) can be performed.
- self->ts = 0: This frees the "ts" thread-local variable.

**Lesson 7. Measuring CPU Time in read()**

```
dtrace -n 'syscall::read:entry { self->vts = vtimestamp; } syscall::read:return /self->vts/ {
    @["On-CPU us:"] = lquantize((vtimestamp - self->vts) / 1000, 0, 10000, 10); self->vts = 0; }'
```

Summarize the CPU time spent to execute read(), in nanoseconds, as a histogram.

- vtimestamp: This timestamp counter is only incremented when the current thread is on-CPU. Units are nanoseconds of CPU time. It can be used to see if time was spent blocked vs running, when compared to "timestamp" deltas.
- "On-CPU us:": This is a shortcut (hack) for decorating the output with a label. It's a single fixed key for the aggregation, which will be printed with the aggregation on the output.
- lquantize(): Linear quantize. The arguments are: value, min, max, step.

**Lesson 8. Count Process-Level Events**

```
dtrace -n 'proc::: { @[probename] = count(); } tick-5s { exit(0); }'
```

Count process-level events for five seconds, printing a summary.

- proc: The proc provider has high-level process events, such as process and thread creation and destruction. The rest of the probe name (":::") are blank fields, which are wildcards that match all the probes in this provider. You can list the probes using: dtrace -ln 'proc:::'.
- probename: The name of the probe. Remember that this is the other stable name, along with the provider name.
- tick-5s: This is a shortcut for profile:::tick-5s, a profile provider tick probe that will fire every 5 seconds on one CPU only. The "5s" can be customized.
- exit(0): This action exits DTrace, returning 0 to the shell.

**Lesson 9. Profile On-CPU Kernel Stacks**

```
dtrace -x stackframes=100 -n 'profile-99 /arg0/ { @[stack()] = count(); }'
```

Profile kernel stacks at 99 Hertz, printing a frequency count.

- -x stackframes=100: This sets the stackframes tunable to 100. Its default is 20, and this is the limit for the number of stack frames returned by the stack() action. We're setting it higher to avoid truncation.
- profile-99: A probe from the profile provider (short for profile:::profile-99), which fires at 99 Hertz on *all* CPUs.
- arg0: For the profile provider, this is the kernel program counter. Testing it in a predicate ensures that this is sampling when the kernel is on-CPU.

- stack(): Returns the kernel stack trace. This is used as a key for the aggregation, so that it can be frequency counted. The output of this is ideal to be visualized as a Flame Graph.

**Lesson 10. Scheduler Tracing**

```
dtrace -n 'sched:::off-cpu { @[stack(8)] = count(); }'
```

Count kernel stacks that led to blocking (off-CPU) events.

- sched: The sched provider has probes for different kernel CPU scheduler events: on-cpu, off-cpu, enqueue, dequeue, etc. List them using: dtrace -ln 'sched:::'.
- off-cpu: This probe fires when a thread leaves CPU. This will be a blocking event: eg, waiting on I/O, a timer, paging/swapping, or a lock.
- stack(5): A kernel stack trace, truncated to 8 frames. ("-x stackframes=8" would have the same effect.)
- off-cpu fires in thread context, so that the stack() refers to the thread who is leaving. As you use other providers, pay attention to context, as execname, pid, stack(), etc, may not refer to the target of the probe. This is usually noted in the provider documentation from the DTrace guide.

**Lesson 11. TCP Inbound Connections**

```
dtrace -n 'tcp:::accept-established { @[args[3]->tcps_raddr] = count(); }'
```

TCP passive opens by remote IP address.

- tcp: The tcp provider has high-level probes for TCP events, and arguments for protocol inspection.
- accept-established: This probe fires when a connection was successfully established (a TCP passive open).
- args[3]->tcps_raddr: args[0..N] are typed arguments, and arg0..N are unsigned ints. The tcp provider has typed arguments of protocol information. In this case, args[3]->tcps_raddr is a string version of the remote IP address. You can list the typed arguments using: dtrace -lvn tcp:::accept-established.
- The context of this probe is important: it fires when the kernel TCP routine completes the TCP handshake. At this point, the accepting process is not on-CPU, therefore, builtins like execname will not show which you might expect.

**Lesson 12. Raw Kernel Tracing**

```
dtrace -n 'fbt::vmem_alloc:entry { @[curthread->td_name, args[0]->vm_name] = sum(arg1); }'
```

Summarize kernel vmem_alloc() calls by thread name, vmem cache name, and total requested bytes.

- fbt: This is the function boundary tracing provider, which dynamically traces the entry and return of kernel functions. This is an "unstable" provider: since it can trace any kernel function, there is no guarantee that your fbt probe will work between kernel versions, as the function names, arguments, and roles may change. Also, since it is tracing the raw kernel, you'll need to browse the kernel source to understand what these probes, and arguments, mean.
- curthread: This is a bulitin that is a pointer to the thread struct that is currently on-CPU (see sys/sys/proc.h). Members can be dereferenced, like in C. In this case, td_name, the thread name.
- args[0]: This typed argument is the first entry argument to vmem_alloc(), since we traced the "entry" probe. The source tells us it is currently a vmem_t. The vm_name member of this struct has been dereferenced here.
- arg1: The second argument to vmem_alloc() is a vmem_size_t, which is ultimately an unsigned integer. It can be fetched either using args[1] (typed), or arg1 (uint64_t, untyped).

At this point you understand much of DTrace, and can begin to use and write powerful one-liners. You can browse the full list of DTrace One-Liners to pick up more DTrace functionality.