```
#!/usr/sbin/dtrace -qs
BEGIN
{
        printf("Solaris Dynamic Tracing");
}
```


Jim Mauro

Senior Staff Engineer
Performance & Availability Engineering
Sun Microsystems, Inc

james.mauro@sun.com

Credits

Most of this material was written by the creators of DTrace: Bryan Cantrill, Mike Shapiro & Adam Leventhal.

The Problem

- As systems have grown more complex, performance problems are increasingly not seen in a system until production deployment
- ...but performance analysis tools, by and large, target *developers* in *development*
- Production environment left with crude, process-centric tools – of little use on *systemic* problems

Why Dynamic Tracing?

- Well-defined techniques for debugging *fatal*, *non-reproducible* failure:
  - Obtain core file or crash dump
  - Debug problem *postmortem* using mdb(1), dbx(1)
- Techniques for debugging *transient* failures are much more ad hoc
  - Typical techniques push traditional tools (e.g. truss(1), mdb(1)) beyond their design centers
  - Many transient problems cannot be debugged at all using extant techniques

Transient failure

- Any unacceptable behavior that does not result in fatal failure of the system
- May be a clear failure:
  - “read(2) is returning EIO on a device that isn't reporting any errors.”
  - “Our application occasionally doesn't receive its timer signal.”
  - “One of our threads is missing a condition variable wakeup.”

Debugging transient failure

- Historically, we have debugged transient failure using process-centric tools: truss(1), pstack(1), prstat(1), etc.
- These tools were not designed to debug *systemic* problems
- But the tools designed for systemic problems (i.e., mdb(1)) are designed for postmortem analysis...

Postmortem techniques

- One technique is to use postmortem analysis to debug transient problems by *inducing* fatal failure during period of transient failure
- Better than nothing, but not by much:
  - Requires inducing fatal failure, which nearly always results in more downtime than the transient failure
  - Requires a keen intuition to be able to sort out a dynamic problem from a static snapshot of state

Solution Constraints

- Performance analysis in production
  - Must have zero probe effect when enabled
  - Must be absolutely, positively, unquestionably, irrefutably, SAFE!
    - Errors and misuse MUST NOT induce system failure
- To have system scope
  - Entire system must be intrumentable – kernel and applications
  - Must be able to easily prune and coalesce data to highlight systemic trends

Introducing DTrace

- New facility in Solaris 10 for dynamic instrumentation of production systems
- Dtrace features:
  - **Dynamic instrumentation:** zero proble effect when disabled
  - **Unified instrumentation:** can instrument both the kernel and running apps such that data and control flow can be followed across boundaries
  - **Arbitrary-context kernel instrumentation:** can instrument even delicate kernel subsystems, like scheduling and synchronization

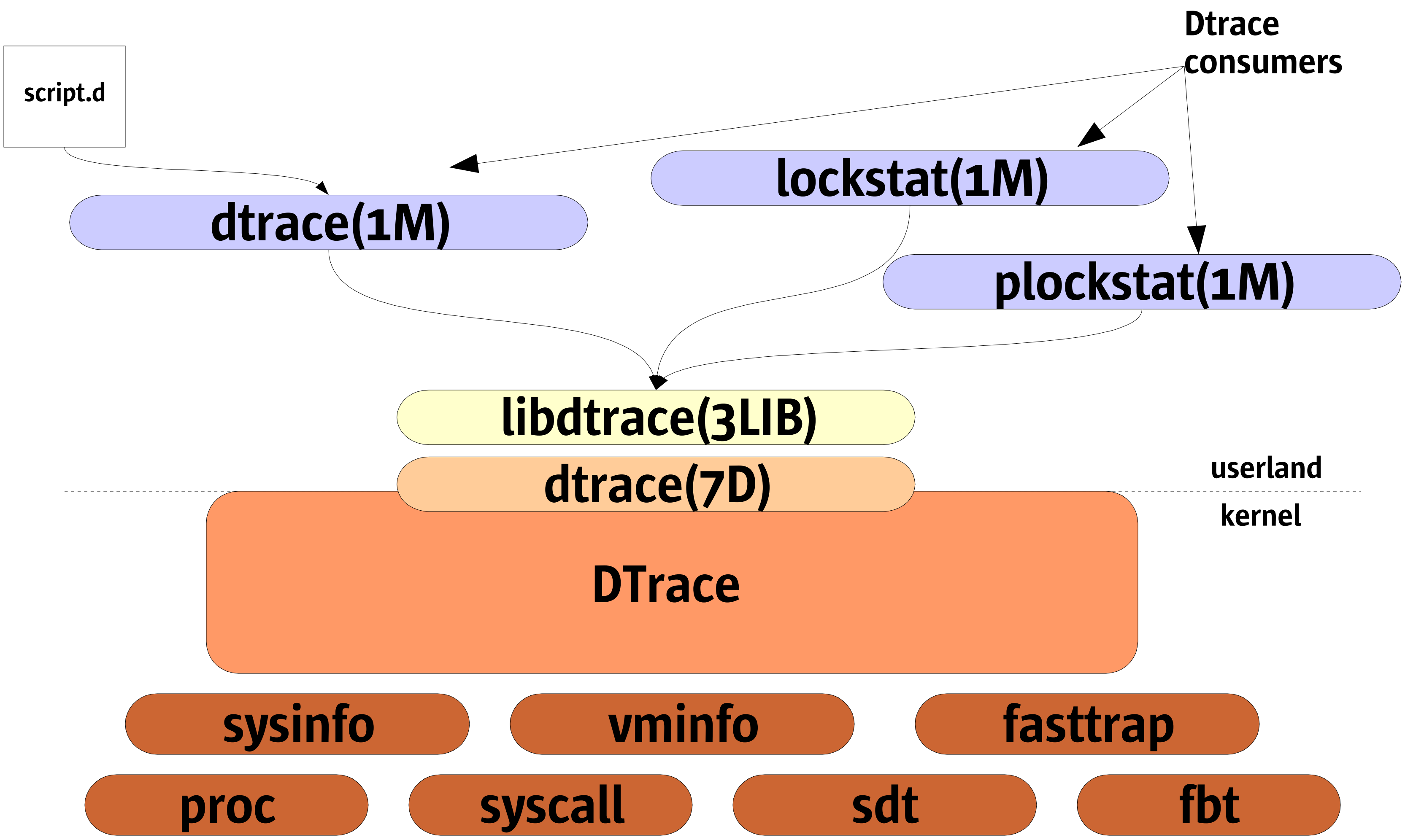
DTrace Features (cont)

- **Data integrity:** if data can not be recorded for any reason, errors are always reported
- **Arbitrary actions:** actions that can be taken at any point of instrumentation are not defined a priori; user can specify arbitrary action
- **Predicates:** predicate mechanism allows actions to be taken only when user-specified conditions are met
- **High level control language:** predicates and actions are specified in a C-like language that supports all ANSI C operators, allows access to kernel variables and types

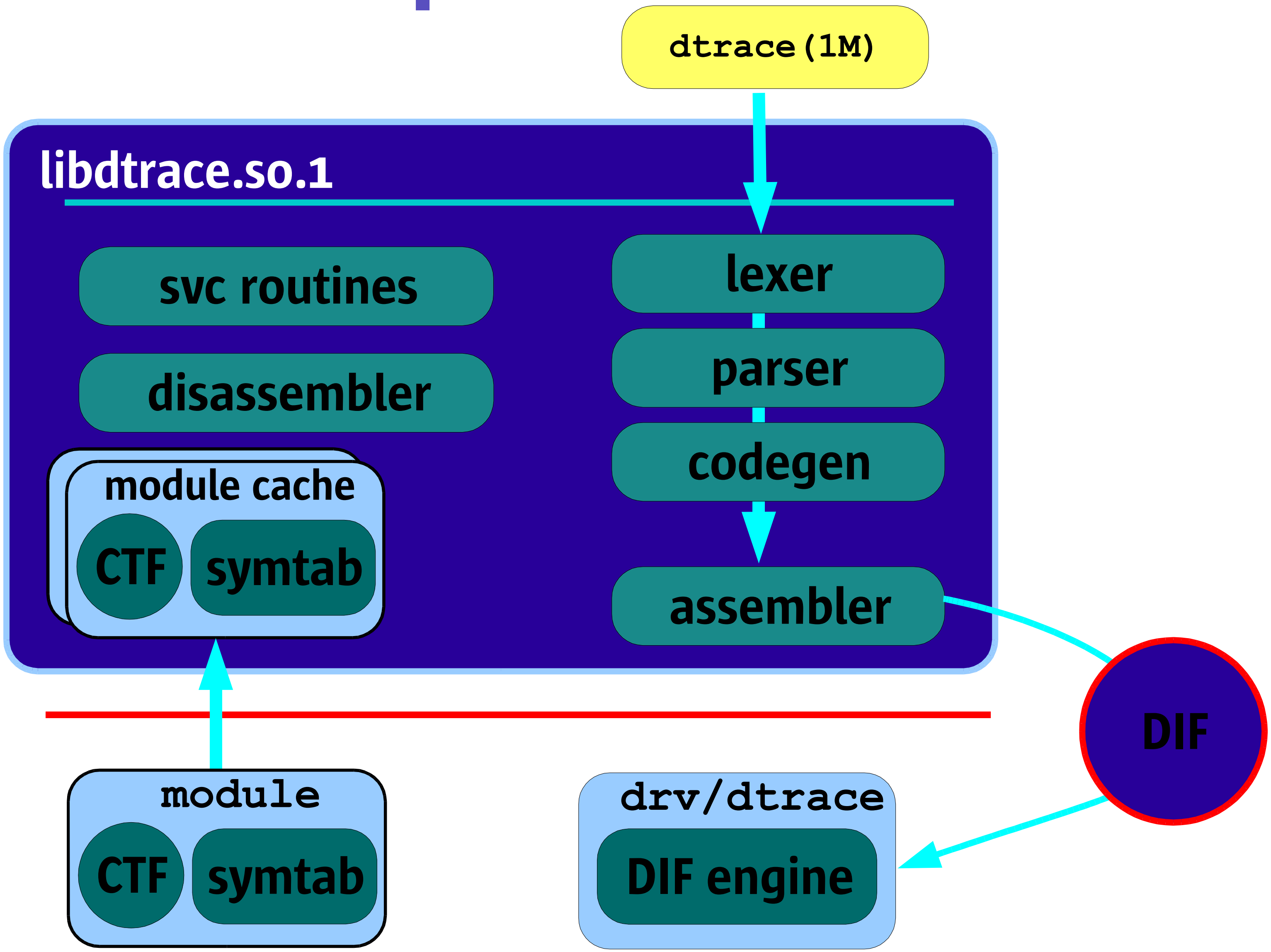
Dtrace Features (cont)

- User-defined variables: support for global and thread-local variables, associative arrays
- Data aggregation: scalable mechanism for aggregating data
- Speculative tracing: mechanism for speculatively recording data and deferring the decision to commit or discard the data
- Scalable architecture: Allows for tens-of-thousands of probes, consumers
- Scripting capability: command line or scripts (the 'D' language)

Dtrace – The Big Picture



Dtrace Components



Probes

- A *probe* is a point of instrumentation
- A probe is made available by a *provider*
- Each probe identifies the *module* and *function* that it instruments
- Each probe has a *name*
- These four attributes define a tuple that uniquely identifies each probe
- Each probe is assigned an integer identifier

Dtrace Probes

```
pae1> dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4   fasttrap                                            fasttrap fasttrap
    5    syscall                                               nosys entry
    6    syscall                                               nosys return
    ...
  838        fbt              unix                  sfmmu_kpm_page_cache return
  839        fbt              unix                         sfmmu_get_ctx entry
  840        fbt              unix                         sfmmu_get_ctx return
  841        fbt              unix                   sfmmu_tlb_all_demap entry
  842        fbt              unix                   sfmmu_tlb_all_demap return
  843        fbt              unix                     sfmmu_replace_tsb entry
  844        fbt              unix                     sfmmu_replace_tsb return
    ...
19958        sdt                ip                       tcp_wput_accept conn-inc-ref
19959        sdt                ip                  tcp_bind_hash_report conn-inc-ref
19960        sdt                ip                              tcp_rsrv conn-inc-ref
19961        sdt                ip                         tcp_rput_data conn-inc-ref
19962        sdt                ip                              tcp_open conn-inc-ref
19963        sdt                ip                     tcp_eager_cleanup conn-inc-ref
19964        sdt                ip                     tcp_eager_blowoff conn-inc-ref
...
```

Providers

- A provider represents a methodology for instrumenting the system
- Providers make probes available to the DTrace framework
- DTrace informs providers when a probe is to be enabled
- Providers transfer control to DTrace when an enabled probe is hit

Providers, cont.

- DTrace has quite a few providers, e.g.:
  - The *function boundary tracing (FBT)* provider can dynamically instrument every function entry and return in the kernel
  - The *syscall* provider can dynamically instrument the system call table
  - The *lockstat* provider can dynamically instrument the kernel synchronization primitives
  - The *profile* provider can add a configureable-rate profile interrupt of to the system
  - The *plockstat* provider can dynamically instrument user-defined lock primitives
  - ...

Consumers

- A DTrace consumer is a process that interacts with DTrace
- No limit on concurrent consumers; DTrace handles the multiplexing
- Some programs are DTrace consumers only as an implementation detail
- dtrace(1M) is a DTrace consumer that acts as a generic front-end to the DTrace facility

Listing probes

- Probes can be listed with the “-l” option to dtrace(1M)
- Can list probes
  - in a specific function with “-f*function*”
  - in a specific module with “-m*module*”
  - with a specific name with “-n*name*”
  - from a specific provider with “-P*provider*”
- For each probe, provider, module, function and name are displayed

Example – Listing Probes

- How many probes on your system?
- List all the probes in the UFS module

```
# dtrace -l | wc -l
   32743
# dtrace -l -m ufs
   ID   PROVIDER            MODULE                          FUNCTION NAME
14763    sysinfo               ufs                     ufs_idle_free ufsinopage
14764    sysinfo               ufs                     ufs_idle_free ufsipage
14765    sysinfo               ufs                 ufs_iget_internal ufsiget
14766    sysinfo               ufs                          blkatoff ufsdirblk
14767        fbt               ufs                         hashalloc entry
14768        fbt               ufs                         hashalloc return
14769        fbt               ufs                           alloccg entry
14770        fbt               ufs                           alloccg return
...
```

Example – Listing Probes

- List probes in the read function
- List probes with the name xcalls

```
# dtrace -l -f read
   ID   PROVIDER            MODULE                          FUNCTION NAME
   11    syscall                                                read entry
   12    syscall                                                read return
 3821    sysinfo           genunix                              read readch
 3825    sysinfo           genunix                              read sysread
 7384        fbt           genunix                              read entry
 7385        fbt           genunix                              read return
...
# dtrace -l -n xcalls
   ID   PROVIDER            MODULE                          FUNCTION NAME
  492    sysinfo              unix                            xc_all xcalls
  493    sysinfo              unix                           xc_some xcalls
14298    sysinfo SUNW,UltraSPARC-II                    send_one_mondo xcalls
```

Example – Listing Probes

- List probes from the sysinfo provider

```
# dtrace -l -P sysinfo
   ID   PROVIDER            MODULE                          FUNCTION NAME
  492    sysinfo              unix                            xc_all xcalls
  493    sysinfo              unix                           xc_some xcalls
  494    sysinfo              unix                          fpu_trap trap
  495    sysinfo              unix                              trap trap
  498    sysinfo              unix                          swtch_to pswitch
  499    sysinfo              unix                 swtch_from_zombie pswitch
  500    sysinfo              unix                             swtch pswitch
  502    sysinfo              unix                    rw_enter_sleep rw_wrfails
  504    sysinfo              unix                           preempt inv_swtch
```

Fully specifying probes

- To specify multiple components of a probe tuple, separate the components with a colon
- Empty components match anything
- For example, “syscall::open:entry” specifies a probe:
  - from the “syscall” provider
  - in any module
  - in the “open” function
  - named “entry”

Specifying Probes

- Four components to probe

```
provider:module:function:name
e.g.
fbt:genunix:sys_enterclass:entry
```

- dtrace(1M) options for probe components

```
dtrace [ -i id]
    [ -P prov]
    [ -m [prov:] mod ]
    [ -f [[ prov: ] mod: ] func ]
    [ -n [[[ prov: ] mod: ] func: ] name
```

```
pae1> dtrace -n fbt:genunix:sys_enterclass:entry
dtrace: description 'fbt:genunix:sys_enterclass:entry' matched 1 probe
```

Enabling probes

- Probes are enabled by specifying them without the “-l” option
- When enabled in this way, probes are enabled with the *default action*
- The default action will indicate only that the probe fired; no other data will be recorded
- For example, “dtrace -m nfs” enables every probe in the “nfs” module

DTrace – Enabling Probes

- Enable probes provided by the “syscall” provider, and the “syscall” “open” function

```
pae1> dtrace -P syscall
dtrace: description 'syscall' matched 452 probes
CPU     ID                    FUNCTION:NAME
  0    102                     ioctl:return
  0    101                      ioctl:entry
  0    102                     ioctl:return
  0    101                      ioctl:entry
  0    102                     ioctl:return
pae1> dtrace -f syscall::open
dtrace: description 'syscall::open' matched 2 probes
CPU     ID                    FUNCTION:NAME
 12     15                       open:entry
 12     16                      open:return
 12     15                       open:entry
 12     16                      open:return
 12     15                       open:entry
```

Dtrace – Enabling Probes

- Enable the entry probe in the clock function

```
# dtrace -n clock:entry
dtrace: description 'clock:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0   3967                      clock:entry
  0   3967                      clock:entry
  0   3967                      clock:entry
  0   3967                      clock:entry
```

Actions

- *Actions* are taken when a probe fires
- Actions are completely programmable
- Most actions *record* some specified state in the system
- Some actions *change* the state of the system system in a well-defined way
  - These are called *destructive actions*
  - Disabled by default
- Many actions take as parameters expressions in the *D language*

The D language

- D is a C-like language specific to DTrace, with some constructs similar to awk(1)
- Complete access to kernel C types
- Complete access to statics and globals
- Complete support for ANSI-C operators
- Support for strings as first-class citizen
- We'll introduce D features as we need them...

Built-in D variables

- For now, our D expressions will consist only of built-in variables
- Example of built-in variables:
  - `pid` is the current process ID
  - `execname` is the current executable name
  - `timestamp` is the time since boot, in nanoseconds
  - `curthread` is a pointer to the `kthread_t` structure that represents the current thread
  - `probemod`, `probefunc` and `probename` are the current probe's module, function and name

Actions: “`trace`”

- `trace()` records the result of a D expression to the trace buffer
- For example:
  - `trace(pid)` traces the current process ID
  - `trace(execname)` traces the name of the current executable
  - `trace(curthread->t_pri)` traces the `t_pri` field of the current thread
  - `trace(probefunc)` traces the function name of the probe

Actions, cont.

- Actions are indicated by following a probe specification with "{ *action* }"
- For example:

```
dtrace -n 'readch{trace(pid)}'
dtrace -m 'ufs{trace(execname)}'
dtrace -n 'syscall::entry {trace
  (probefunc)}'
```

- Multiple actions can be specified; they must be separated by semicolons:

```
dtrace -n 'xcalls{trace(pid); trace
  (execname)}'
```

DTrace - Actions

- Trace the executable name in every "*poll*" system call

```
pae1> dtrace -n 'syscall::poll: { trace(execname) }'
dtrace: description 'syscall::poll: ' matched 2 probes
^c
```

DTrace - Actions

- Trace the PID in every entry to the "*pagefault*" function

```
pae1> dtrace -f 'pagefault { trace(pid) }'
dtrace: description 'pagefault ' matched 2 probes
CPU     ID                    FUNCTION:NAME
 12   2554                  pagefault:entry      3979
 12   2555                 pagefault:return      3979
 12   2554                  pagefault:entry      3979
 12   2555                 pagefault:return      3979
 12   2554                  pagefault:entry      3979
^c
...
```

DTrace - Actions

- Trace the timestamp in every entry to the "*clock*" function

```
pae1> dtrace -f 'clock { trace(timestamp) }'
dtrace: description 'clock ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0   4113                     clock:entry  1306863050033058
  0   4114                    clock:return  1306863050128812
  0   4113                     clock:entry  1306863060015632
  0   4114                    clock:return  1306863060094122
  0   4113                     clock:entry  1306863070016883
  0   4114                    clock:return  1306863070094331
...
```

D Scripts

- Complicated DTrace enablings become difficult to manage on the command line
- dtrace(1M) supports *scripts*, specified with the “-s” option
- Alternatively, executable DTrace interpreter files may be created
- Interpreter files always begin with:

```
#!/usr/sbin/dtrace -s
```

D Scripts, cont.

- For example, a script to trace the executable name upon entry of each system call:

```
#!/usr/sbin/dtrace -s

syscall:::entry
{
        trace(execname);
}
```

Predicates

- *Predicates* allow actions to only be taken when certain conditions are met
- A predicate is a D expression
- Actions will only be taken if the predicate expression evaluates to true
- A predicate takes the form "*/expression/*" and is placed between the probe description and the action

Predicates, cont.

- For example, tracing the pid of every process named “date” that performs an open(2):

```
#!/usr/sbin/dtrace -s

syscall::open:entry
/execname == “date”/
{
        trace(pid);
}
```

Example - Predicates

- Trace the timestamp in every ioctl(2) from processes named dtrace

```
pae1> cat dioctl.d
#!/usr/sbin/dtrace -s

::ioctl:
/ execname == "dtrace" /
{
    trace(pid);
}


pae1> ./dioctl.d
dtrace: script './dioctl.d' matched 4 probes
CPU     ID                    FUNCTION:NAME
  8   7984                 ioctl:return      3994
  8    102                 ioctl:return      3994
  8    101                  ioctl:entry      3994
  8   7983                  ioctl:entry      3994
```

Example - Predicates

- Use the `arg0` variable to trace the executable name of every process read (2)'ing from file descriptor 0

```
pae1> cat pred2.d
#!/usr/sbin/dtrace -s
::read:
/ arg0 == 0 /
{
    trace(execname);
}

pae1> ./pred2.d
dtrace: script './pred2.d' matched 6 probes
CPU     ID                    FUNCTION:NAME
  8   3956                      read:readch   ksh
  8     12                      read:return   ksh
 13   3956                      read:readch   ksh
 13     12                      read:return   ksh
```

Example - Predicates

- Use the `arg2` variable to trace the executable name of every processing write(2)'ing more than 100 bytes

```
pae1> cat pred3.d
#!/usr/sbin/dtrace -s

::write:entry
/ arg2 > 100 /
{
        trace(execname);
        printf("Wrote: %d bytes\n",arg0);
}
::write:return
{
        printf("Wrote: %d bytes\n",arg0);
}
pae1> ./pred3.d
CPU     ID    FUNCTION:NAME
  4   3952    write:syswrite    automountd     Wrote: 1 bytes

  4   3948    write:writech     automountd     Wrote: 1 bytes

  0   8069    write:return      dtrace         Wrote: 936 bytes

 13   8069    write:return      ls             Wrote: 936 bytes

...
```

Actions: More actions

- `tracemem()` records memory at a specified location for a specified length
- `stack()` records the current *kernel* stack trace
- `ustack()` records the current *user* stack trace
- `exit()` tells the DTrace consumer to exit with the specified status

Actions: Destructive actions

- Must specify “-w” option to DTrace
- `stop()` stops the current process
- `raise()` sends a specified signal to the current process
- `breakpoint()` triggers a kernel breakpoint
- `panic()` induces a kernel panic
- `chill()` spins for a specified number of nanoseconds

Output formatting

- The `printf()` function combines the `trace` action with the ability to precisely control output
- `printf` takes a printf(3C)-like format string as an argument, followed by corresponding arguments to print
- e.g.:

```
printf(“%d was here”, pid);
printf(“I am %s”, execname);
```

Output formatting, cont.

- Normally, dtrace(1M) provides details on the firing probe, plus any explicitly traced data
- Use the quiet option ("-q") to dtrace (1M) to supress the probe details
- The quiet option may also be set in a D script by embedding:

```
#pragma D option quiet
```

Global D variables

- D allows you to define your own variables that are global to your D program
- Like awk(1), D tries to infer variable type upon instantiation, obviating an explicit variable declaration

Global D variables, cont.

- Example:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

sysinfo:::zfod
{
        zfods++;
}

profile:::tick-1sec
{
        printf(“%d zfods\n”,  zfods);
        zfods = 0;
}
```

Thread-local D variables

- D allows for *thread-local* variables
- A thread-local variable has the same name – but disjoint data storage – for each thread
- By definition, thread-local variables elminate the race conditions that are endemic to global variables
- Denoted by prepending "`self->`" to the variable name

Thread-local D variables, cont

- Thread-local variables that have never been assigned in the current thread have the value zero
- Underlying thread-local storage for a thread-local variable is deallocated by assigning zero to it

Thread-local D variables, cont.

- Example 1:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall::poll:entry
{
        self->ts = timestamp;
}

syscall::poll:return
/self->ts && timestamp - self->ts > 1000000000/
{
        printf("%s polled for %d seconds\n", execname,
            (timestamp - self->ts) / 1000000000);
        self->ts = 0;
}
```

Thread-local D variables, cont.

- Example 2:

```
syscall::ioctl:entry
/execname == “date”/
{
        self->follow = 1;
}

fbt:::
/self->follow/
{}

syscall::ioctl:return
/self->follow/
{
       self->follow = 0;
}
```

D Variables

- Write a D script to trace the executable name and amount of time spent in every open(2)

```
#!/usr/sbin/dtrace -qs

syscall::open:entry
{
     self->st = timestamp;
}
syscall::open:return
/ self->st /
{
     tt = timestamp - self->st;
     printf("%s, %d nsecs in open\n",execname, tt);
}
pae1> ./open.d
ls, 64700 nsecs in open
ls, 24870 nsecs in open
date, 71220 nsecs in open
date, 62120 nsecs in open
ls, 58583 nsecs in open
ls, 24758 nsecs in open
ls, 71976 nsecs in open
^C
```

D Variables

- Write a D script to follow a brk(2) system call through the kernel when called by a date(1) command

D Variables

- Add "`#pragma D option flowindent`" to the above and observe the change in output

Aggregations

- When trying to understand suboptimal performance, one often looks for *patterns* that point to bottlenecks
- When looking for patterns, one often doesn't want to study each datum – one wishes to *aggregate* the data and look for larger trends
- Traditionally, one has had to use conventional tools (e.g. awk(1), perl(1))

Aggregations, cont.

- DTrace supports the aggregation of data as a first class operation
- An *aggregating function* is a function f(x), where x is a set of data, such that:

  $f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)$
- E.g., COUNT, SUM, MAXIMUM, and MINIMUM are aggregating functions; MEDIAN, and MODE are not

Aggregations, cont.

- An *aggregation* is the result of an aggregating function keyed by an arbitrary tuple
- For example, to count all system calls on a system by system call name:

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

- By default, aggregation results are printed when dtrace(1M) exits

Aggregations, cont.

- Aggregations need not be named
- Aggregations can be keyed by more than one expression
- For example, to count all ioctl system calls by both executable name and file descriptor:

```
dtrace -n 'syscall::ioctl:entry \
  { @[execname, arg0] = count(); }'
```

Aggregations, cont.

- Some other aggregating functions:
  - `avg()`: the average of specified expressions
  - `min()`: the minimum of specified expressions
  - `max()`: the maximum of specified expressions
  - `quantize()`: power-of-two distribution of specified expressions
- For example, distribution of write(2) sizes by executable name:

```
dtrace -n 'syscall::write:entry \
  { @[execname] = quantize(arg2); }'
```

Exploring DTrace

- This has been just an introduction to DTrace – there's much, much more:
  - BEGIN, END probes
  - Normalization
  - Associative arrays
  - User-level tracing
  - Speculative tracing
  - Postmortem tracing
  - Explicit versioning
  - Aggregation formatting
  - Provider specifics
  - Clause-local variables
  - Ring buffering
  - Anonymous tracing
  - Privilege model
  - Well-defined stability

Exploring DTrace, cont.

- http://docs.sun.com
  - Solaris 10 documentation online
  - “Solaris Dynamic Tracing Guide”

    Written by the engineers that designed and built DTrace
- BigAdmin has a page and discussion forum dedicated to DTrace:

  http://www.sun.com/bigadmin/content/dtrace

The DTrace Revolution

- DTrace tightens the diagnosis loop: *hypothesis* → *instrumentation* → *data gathering* → *analysis* → *hypothesis*
- Tightened loop effects a revolution in the way we diagnose transient failure
- Focus can shift from *instrumentation* stage to *hypothesis* stage:
  - Much *less* labor intensive, less error prone
  - Much *more* brain intensive
  - *Much* more effective! (And a *lot* more fun)

```
#!/usr/sbin/dtrace -s

END
{
    printf("Solaris Dynamic Tracing\n");
}
```

james.mauro@sun.com