L41: Kernels and Tracing

Dr Robert N. M. Watson

25 February 2015
Reminder: last time

1. What is an operating system?
2. Systems research
3. About the module
4. Lab reports
This time: Tracing the kernel

1. DTrace
2. The probe effect
3. The kernel source
4. A little on kernel dynamics
Dynamic tracing with DTrace

  - “Facility for dynamic instrumentation of production systems”
  - Unified and safe instrumentation of kernel and userspace
  - Zero probe effect when not enabled
  - Dozens of ‘providers’ representing different trace sources
  - Tens of thousands of instrumentation points
  - C-like high-level control language with predicates and actions
  - User-defined variables, thread-local variables, associative arrays
  - Data aggregation and speculative tracing

- Adopted in Solaris, Mac OS X, and FreeBSD; module for Linux
- Heavy influence on Linux ftrace
- Our tool of choice for this module
DTrace scripts

- Human-facing C-like language
- One or more \{probe name, predicate, action\} tuples
- Expression limited to control side effects (e.g., no loops)
- Specified on command line or via a .d file

```
fbt::malloc:entry /execname == "csh"/ { trace(arg0); }
```

**probe name** Identifies the probe(s) to instrument; wildcards allowed; identifies the **provider** and a provider-specific **probe name**

**predicate** Filters cases where action will execute

**action** Describes tracing operations
D Intermediate Format (DIF)

root@beaglebone:/data # dttrace -Sn 'fbt::malloc:entry /execname == "csh"/ { trace(arg0); }'

DIFO 0x8047d2320 returns D type (integer) (size 4)
OFF OPCODE INSTRUCTION
00: 29011801 ldgs DT_VAR(280), %r1 ! DT_VAR(280) = "execname"
01: 26000102 sets DT_STRING[1], %r2 ! "csh"
02: 27010200 scmp %r1, %r2
03: 12000006 be 6
04: 0e000001 mov %r0, %r1
05: 11000007 ba 7
06: 25000001 setx DT_INTEGER[0], %r1 ! 0x1
07: 23000001 ret %r1

NAME ID KND SCP FLAG TYPE
execname 118 scl glb r string (unknown) by ref (size 256)

DIFO 0x8047d2390 returns D type (integer) (size 8)
OFF OPCODE INSTRUCTION
00: 29010601 ldgs DT_VAR(280), %r1 ! DT_VAR(280) = "execname"
01: 23000001 ret %r1

NAME ID KND SCP FLAG TYPE
arg0 106 scl glb r D type (integer) (size 8)
Some kernel DTrace providers in FreeBSD

<table>
<thead>
<tr>
<th>Provider</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callout_execute</td>
<td>Timer-driven callouts</td>
</tr>
<tr>
<td>dtmalloc</td>
<td>Kernel malloc()/free()</td>
</tr>
<tr>
<td>dtrace</td>
<td>DTrace script events (BEGIN, END)</td>
</tr>
<tr>
<td>fbt</td>
<td>Function Boundary Tracing</td>
</tr>
<tr>
<td>io</td>
<td>Block I/O</td>
</tr>
<tr>
<td>ip, udp, tcp, sctp</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>lockstat</td>
<td>Locking</td>
</tr>
<tr>
<td>proc, sched</td>
<td>Kernel process/scheduling</td>
</tr>
<tr>
<td>profile</td>
<td>Profiling timers</td>
</tr>
<tr>
<td>syscall</td>
<td>System call entry/return</td>
</tr>
<tr>
<td>vfs</td>
<td>Virtual filesystem</td>
</tr>
</tbody>
</table>

- Providers represent data sources – types of instrumentation
- Apparent duplication: FBT vs. event-class providers?  
  - Efficiency, expressivity, interface stability, portability
Tracing kernel `malloc()` calls

- Trace first argument to kernel `malloc()` for `csh`
- Note: captures both successful and failed allocations

```
root@beaglebone:/data # dtrace -n
  'fbt::malloc:entry /execname=="csh"/ { trace(arg0); }'
```

**Probe** Use FBT to instrument `malloc()` prologue

**Predicate** Limit actions to processes executing `csh`

**Action** Trace the first argument (`arg0`)

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>64</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>2748</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>392</td>
</tr>
</tbody>
</table>

^C
## Aggregations

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>Number of times called</td>
</tr>
<tr>
<td><code>sum()</code></td>
<td>Sum of arguments</td>
</tr>
<tr>
<td><code>avg()</code></td>
<td>Average of arguments</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>Minimum of arguments</td>
</tr>
<tr>
<td><code>max()</code></td>
<td>Maximum of arguments</td>
</tr>
<tr>
<td><code>stddev()</code></td>
<td>Standard deviation of arguments</td>
</tr>
<tr>
<td><code>lquantize()</code></td>
<td>Linear frequency distribution (histogram)</td>
</tr>
<tr>
<td><code>quantize()</code></td>
<td>Log frequency distribution (histogram)</td>
</tr>
</tbody>
</table>

- Often we want summaries of events, not detailed traces
- DTrace allows early, efficient *reduction* using aggregations
- Scalable multicore implementations (i.e., commutative)
- `@variable = function(); printa()` to print
Profiling kernel `malloc()` calls by `csh`

root@beaglebone:/data # dtrace -n 'fbt::malloc:entry /execname=="csh"/ { @traces[stack()] = count(); }'

**Probe** Use FBT to instrument `malloc()` prologue

**Predicate** Limit actions to processes executing `csh`

**Action** Keys of associative array are stack traces (`stack()`); values are aggregated counters (`count()`)
DTrace: implementation

```
dtrace -n 'fbt::malloc:entry { trace(execname); trace(arg0); }'
```

```
dtrace -n 'dtmalloc::temp:malloc /execname="csh"/ { trace(execname); trace(arg3); }'
```

```
CPU ID FUNCTION:NAME
0 30408 malloc:entry dtrace 608
0 30408 malloc:entry dtrace 608
3 30408 malloc:entry dtrace 120
3 30408 malloc:entry dtrace 120
3 30408 malloc:entry dtrace 324
0 30408 malloc:entry intr 1232
0 30408 malloc:entry csh 64
0 30408 malloc:entry csh 3272
2 30408 malloc:entry csh 80
2 30408 malloc:entry csh 560
```

```
CPU ID FUNCTION:NAME
1 54297 temp:malloc csh 1024
1 54297 temp:malloc csh 64
```
The ‘probe effect’

- The *probe effect* is the unintended alteration in system behaviour that arises from measurement
- Why? Software instrumentation is *active*: the code is changed
- Potential perturbations:
  - Execution speed relative to other cores: e.g., lock hold times
  - Execution speed relative to external events: e.g., timer ticks
  - Microarchitectural effects: e.g., cache footprint, branch predictor

- DTrace minimises *probe effect* when not being used...
  - ... but has a very significant impact when it is
  - Disproportionate effect on probed events

- What does this mean for us?
  - Don’t benchmark while running DTrace ...
  - ... unless benchmarking DTrace
  - Be aware that traced application may behave differently
  - E.g., more timer ticks will fire, I/O will “seem faster”
Probe effect example: `dd system time`

```bash
# dd if=/dev/zero of=/dev/null bs=10m count=1 status=none

fbt::malloc::entry { @traces[stack()] = count(); }
```

**NB:** `ministat` is an incredibly useful tool – try it!
“Just a C program”

I claimed that the kernel was mostly “just a C program”. This is mostly true, especially if you look at high-level subsystems.

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>crt/csu</td>
<td>locore</td>
</tr>
<tr>
<td>rtld</td>
<td>Kernel linker</td>
</tr>
<tr>
<td>Shared objects</td>
<td>Kernel modules</td>
</tr>
<tr>
<td>main()</td>
<td>main(),platform_start</td>
</tr>
<tr>
<td>libc</td>
<td>libkern</td>
</tr>
<tr>
<td>POSIX threads API</td>
<td>kthread KPI</td>
</tr>
<tr>
<td>POSIX filesystem API</td>
<td>VFS KPI</td>
</tr>
<tr>
<td>POSIX socket API</td>
<td>socket KPI</td>
</tr>
<tr>
<td>DTrace</td>
<td>DTrace</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Kernel dynamics

But not just *any* C program

- Core kernel: \(\approx 3.4\)M LoC in \(\approx 6,450\) files
  - Kernel foundation: Built-in linker, object model, scheduler, memory allocator, threading, debugger, tracing, I/O routines, timekeeping
  - Base kernel: VM, process model, IPC, VFS w/20+, filesystems, network stack (IPv4/IPv6, 802.11, ATM, ...), crypto framework
  - Includes roughly \(\approx 70K\) lines of assembly over \(\approx 6\) architectures

- Alternative C runtime – e.g., SYSINIT, curthread
- Highly concurrent – really, very, very concurrent
- Virtual memory makes pointers .. odd
- Debugging features such as WITNESS lock order verifier

- Device drivers: \(\approx 3.0\)M LoC in \(\approx 3,500\) files
  - \(\approx 415\) device drivers (may support multiple devices)
Spelunking the kernel

Kernel source lives in `/usr/src/sys/sys`:
- kern/ - core kernel features
- sys/ - core kernel headers

Useful resource: [http://fxr.watson.org/](http://fxr.watson.org/)
How work happens in the kernel

- Kernel code executes concurrently in multiple threads
  - User threads in kernel (e.g., system call)
  - Shared worker threads (e.g., callouts)
  - Subsystem worker threads (e.g., network-stack worker)
  - Interrupt threads (e.g., clock ticks)
  - Idle threads

```
root@beaglebone:/data # procstat -at

PID   TID  COMM     TDNAME      CPU  PRI  STATE  WCHAN
0    100000 kernel swapper     -1   84  sleep  swapin
0    100006 kernel dtrace_taskq -1   84  sleep  

10   100002 idle          -     -1  255  run   
11   100003 intr          swi3: vm    0  36  wait  
11   100004 intr          swi4: clock (0) -1  40  wait  
11   100005 intr          swi1: netisr 0 -1  28  wait  

11   100018 intr          intr16: ti_adc0 0  20  wait  
11   100019 intr          intr91: ti_wdt0 0  20  wait  
11   100020 intr          swi0: uart  -1  24  wait  

739  100064 login          -     -1  108  sleep  wait
740  100079 csh             -     -1  140  sleep  ttyin
751  100089 procstat       -     0  140  run   
```
Deferred work

- Many operations begin with system calls in a user thread
- But they may trigger work in many other threads; for example:
  - Triggering a callback in an interrupt thread when I/O is complete
  - Eventual write back of data to disk from the cache
  - Delayed transmission if TCP isn’t able to send

- Several major subsystems provide this:
  - `callout` Closure called after wall-clock delay
  - `eventhandler` Closure called for key global events
  - `task` Closure called eventually
  - `SYSINIT` Function called when module loads/unloads

- (Where closure in C means: function pointer, opaque data pointer)

- We will need to care about these things, as not all the work we are analysing will be in the user thread performing a system call.
For next time

- Read Ellard and Seltzer, *NFS Tricks and Benchmarking Traps*
- Skim handout, *L41: DTrace Quick Start*
- Be prepared to try out DTrace on a real system