L41 - Lecture 3: The Process Model (1)

Dr Robert N. M. Watson

2 March 2015
Reminder: last time

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

I showed you dramatic example of the *probe effect*: 80% overhead for some stack tracing for `malloc()` when running `dd`.

Gosh, that is **really big**, even for the probe effect.
Last time: Probe effect example: `dd` system time

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

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

```
x no-dtrace
+ dtrace
```

```
+------------------------------------------------------------------------------+
| + | | + | | + | | + | | + | | + | | + | | + | | + | | + | | + | | + | | + | + + + + | + + + + | + + + + |
+------------------------------------------------------------------------------+
```

<table>
<thead>
<tr>
<th>N</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Avg</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 10</td>
<td>0.155</td>
<td>0.187</td>
<td>0.179</td>
<td>0.1757</td>
<td>0.011333824</td>
</tr>
<tr>
<td>+ 10</td>
<td>0.47</td>
<td>0.503</td>
<td>0.495</td>
<td>0.4925</td>
<td>0.0087717982</td>
</tr>
</tbody>
</table>

Difference at 95.0% confidence
0.3168 +/- 0.00952196
180.307% +/- 5.41944%
(Student’s t, pooled s = 0.0101341)
So we did a little investigating...

- Number of calls to `malloc` per run: 3

- That’s silly overhead for three traces, even with traps for each `malloc`, DIF execution, walking the stack, writing to a buffer, etc.

- Discovered that FreeBSD/arm DTrace `stack()` cost 10M cycles per frame on the stack! Doh!

- On all other architectures, tracing stacks is really cheap
- On ARM, you must use EABI unwind tables
  - ... and that look up unwind state for every return address
  - ... which turns out to never have been a fast path before DTrace
  - ... which meant it acquired locks and did linear walks

- Tried module-free kernel on BeagleBone Black
- Now it is basically free (no statistically significant difference)
- TODO: fix module lookup in FreeBSD/ARM for next year
Take two: Probe effect example: `dd system time`

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

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

x no-dtrace
+ dtrace
* dtrace-better

+-----------------------------------------------+
|       |       |       |       |
| x     | x *   | x *   | x *   |
| x     | *x**  | *x**  | *x**  |
| x *   | *x**x |       |       |
|       |       |       |       |
+-----------------------------------------------+

N  Min  Max  Median  Avg   Stddev
x  10  0.155 0.187  0.179  0.1757  0.011333824
+ 10  0.47  0.503  0.495  0.4925  0.0087717982

Difference at 95.0% confidence
0.3168 +/- 0.00952196
180.307% +/- 5.41944%
(Student’s t, pooled s = 0.0101341)

x  10  0.165  0.191  0.183  0.1822  0.0088794394
+ 10

No difference proven at 95.0% confidence
```

I will also need a different example of the probe effect next year. Turns out, `stack` from every `lock acquire` will work just fine. Tracing 57,446 mutex acquisitions = $\approx 3.72 \times$ overhead – perfect.
This time: the process model

1. The process model and its evolution
2. Brutal pre-introduction to virtual memory
3. Where do programs come from?
4. Traps and system calls
5. Reading for next time
The process model: 1970s foundations

- **Multics** process model
  - ‘Program in execution’
  - *Process isolation* bridged by *controlled communication* via supervisor (kernel)
- **Hardware foundations**
  - Supervisor mode
  - Memory segmentation
  - Trap mechanism
- **Hardware protection rings**
  (Schroeder and Saltzer, 1972)
The process model: today

- ‘Program in execution’
  - Process $\approx$ address space
  - ‘Threads’ execute code
  - Kernel interaction via *traps*: system calls, page faults, ...
  - Unit of resource accounting: open files, credentials, memory, scheduling, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - MMU virtual addressing
  - Trap mechanism
- Details vary only a little over {UNIX, Linux, Windows, ...}
The UNIX process life cycle

- **fork()**
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)

- **execve()**
  - Kernel replaces address space, loads new binary, starts execution

- **exit()**
  - Process can terminate self (or be terminated)

- **wait4 (et al)**
  - Parent can await exit status
Process model evolution

- 1980s: Code, heap, and stack
- 1990s: Dynamic linking, multithreading
- 2000s: Scalable memory allocators implement multiple arenas (e.g., jemalloc)
Process address space: \texttt{dd}

- Inspect \texttt{dd} process address space with \texttt{procstat -v}.

```
root@beaglebone:/data # procstat -v 734

<table>
<thead>
<tr>
<th>PID</th>
<th>START</th>
<th>END</th>
<th>PRT</th>
<th>RES</th>
<th>PRES</th>
<th>REF</th>
<th>SHD</th>
<th>FLAG</th>
<th>TP</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>734</td>
<td>0x8000</td>
<td>0xd000</td>
<td>r-x</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>CN--</td>
<td>vn</td>
<td>/bin/dd</td>
</tr>
<tr>
<td>734</td>
<td>0x14000</td>
<td>0x16000</td>
<td>rw-</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x2001400</td>
<td>0x20031000</td>
<td>r-x</td>
<td>29</td>
<td>32</td>
<td>31</td>
<td>14</td>
<td>CN--</td>
<td>vn</td>
<td>/libexec/ld-elf.so.1</td>
</tr>
<tr>
<td>734</td>
<td>0x20038000</td>
<td>0x20039000</td>
<td>rw-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C---</td>
<td>vn</td>
<td>/libexec/ld-elf.so.1</td>
</tr>
<tr>
<td>734</td>
<td>0x20039000</td>
<td>0x20052000</td>
<td>rw-</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20100000</td>
<td>0x2025f000</td>
<td>r-x</td>
<td>351</td>
<td>360</td>
<td>31</td>
<td>14</td>
<td>CN--</td>
<td>vn</td>
<td>/lib/libc.so.7</td>
</tr>
<tr>
<td>734</td>
<td>0x2025f000</td>
<td>0x20266000</td>
<td>---</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20266000</td>
<td>0x2026e000</td>
<td>rw-</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C---</td>
<td>vn</td>
<td>/lib/libc.so.7</td>
</tr>
<tr>
<td>734</td>
<td>0x2026e000</td>
<td>0x20285000</td>
<td>rw-</td>
<td>7</td>
<td>533</td>
<td>2</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20400000</td>
<td>0x20c00000</td>
<td>rw-</td>
<td>526</td>
<td>533</td>
<td>2</td>
<td>0</td>
<td>--S-</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0xbffe0000</td>
<td>0xc0000000</td>
<td>rwx</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>D</td>
<td>df</td>
</tr>
</tbody>
</table>
```

Program binary, \texttt{r}: read
Run-time linker, \texttt{w}: write
Shared library, \texttt{x}: execute
BSS, Heap, Stack \texttt{C}: Copy-on-write
\texttt{D}: Downward growth
\texttt{S}: Superpage
ELF binaries

- **UNIX**: Executable and Linkable Format (ELF)
- **Mac OS X/iOS**: Mach-O; Windows: PE/COFF; same ideas
- **Inspect** *dd* ELF program headers using `objdump -p`:

```
root@beaglebone:~ # objdump -p /bin/dd
/bin/dd: file format elf32-littlearm

Program Header:
0x70000001 off 0x0000469c vaddr 0x0000c69c paddr 0x0000c69c align 2**2
  filesz 0x00000158 memsz 0x00000158 flags r--
PHDR off 0x00000034 vaddr 0x00008034 paddr 0x00008034 align 2**2
  filesz 0x000000e0 memsz 0x000000e0 flags r-x
INTERP off 0x00000114 vaddr 0x00008114 paddr 0x00008114 align 2**0
  filesz 0x00000015 memsz 0x00000015 flags r--
LOAD off 0x00000000 vaddr 0x00008000 paddr 0x00008000 align 2**15
  filesz 0x000047f8 memsz 0x000047f8 flags r-x
LOAD off 0x000047f8 vaddr 0x000147f8 paddr 0x000147f8 align 2**15
  filesz 0x000001b8 memsz 0x00001020 flags rw-
DYNAMIC off 0x00004804 vaddr 0x00014804 paddr 0x00014804 align 2**2
  filesz 0x000000f0 memsz 0x000000f0 flags rw-
NOTE off 0x0000012c vaddr 0x0000812c paddr 0x0000812c align 2**2
  filesz 0x0000004c memsz 0x0000004c flags r--
```
Virtual memory (quick but painful primer)

- Memory Management Unit (MMU)
  - Control available only to the supervisor
  - Mediates/transforms memory access (instruction fetch, load, store)
  - Trap fired on failure (e.g., permissions) to be handled by software

- Page tables
  - Memory is laid out in pages (4K)
  - OS-managed Page tables map virtual pages into physical frames
  - Access permissions, page attributes (e.g., caching)
  - Variable-size pages (now) supported

- The Translation Look-aside Buffer (TLB)
  - Hardware cache of entries – avoid walking tables
  - Content Addressable Memory (CAM); 48? 1024? entries
  - TLB entry tagging: entries global or for a specific process
  - Software- vs. hardware-managed

- Virtual address spaces
  - Isolation vs. sharing
  - BSS, Copy-on-Write, Superpages

- Hypervisors and I/O MMUs
Virtual memory (quick but painful primer) (cont)
Role of the run-time linker (\texttt{rtld})

- Static linking: program and libraries linked into a single binary
- Dynamic linking: binary contains only the application, no libraries
  - Shared libraries conserve memory by avoiding code duplication
  - The run-time linker (\texttt{rtld}) loads and links libraries
  - Program binaries contain a list of their library dependencies
  - Also used for plug-ins via \texttt{dlopen()}, \texttt{dlsym()}

- \textit{Loading, relocating, and run-time linking}

```
root@beaglebone:~ # ldd /bin/dd
/bin/dd:
  libc.so.7 => /lib/libc.so.7 (0x20100000)
```

- When the \texttt{execve} system call starts the new program:
  - ELF binaries name their \textit{interpreter} in an ELF segment
  - \texttt{execve} maps \texttt{rtld} and the application into memory
  - The kernel starts userspace execution in \texttt{rtld}
  - \texttt{rtld} loads and links dynamic libraries, runs constructors
  - \texttt{rtld} calls \texttt{main()}

- Slightly complicating: \textit{lazy binding}
Arguments and ELF auxiliary arguments

- C-program arguments are \texttt{argc}, \texttt{argv[]}, and \texttt{envv[]}:

```
root@beaglebone:/data # procstat -c 716
    PID  COMM        ARGS
  716  dd           dd if=/dev/zero of=/dev/null bs=1m
```

- The run-time linker also accepts arguments from the kernel:

```
root@beaglebone:/data # procstat -x 716
    PID  COMM        AUXV      VALUE
  716  dd           AT_PHDR   0x8034
  716  dd           AT_PHENT  32
  716  dd           AT_PHNUM  7
  716  dd           AT_PAGESZ 4096
  716  dd           AT_FLAGS  0
  716  dd           AT_ENTRY 0x8cc8
  716  dd           AT_BASE  0x20014000
  716  dd           AT_EXEC PATH 0xbfffffff4
  716  dd           AT_OSRELDATE 1100062
  716  dd           AT_NCPUS  1
  716  dd           AT_PAGESIZES 0xbfffffff9c
  716  dd           AT_PAGESIZESLEN 8
...
```
Traps and system calls

- Asymmetric domain transition, *trap*, shifts control to kernel
- Asynchronous traps: e.g., timer, peripheral interrupts, Inter-Processor Interrupts (IPIs)
- Synchronous traps: e.g., system calls, divide-by-zero, page faults

- $pc$ to *interrupt vector*: dedicated OS code to handle trap
- Key challenge: kernel must gain control safely, reliably, securely

**RISC**
- $pc$ saved, $epc$ installed, control coprocessor (MMU, ...) made available, kernel memory access enabled, reserved exception registers in ABI.
- Software must save other state (e.g., registers)

**CISC**
- All that and: context saved to in-memory trap frame

- NB: User context switch = trap to kernel, restore a different context
For next time

- We will continue with system calls and traps
- Then more on virtual memory
- Threading models: the great debate

- McKusick, et al: Chapter 6 (*Memory Management*)
- Optional: Anderson, et al, on *Scheduler Activations*