The Process Model (1)

L41 Lecture 3
Dr Robert N. M. Watson
13 November 2017

Reminder: last time

- DTrace
- The probe effect
- The kernel: Just a C program?
- A little on kernel dynamics: How work happens
This time: The process model

- The process model and its evolution
- Brutal (re, pre)-introduction to virtual memory
- Where do programs come from?
- Traps and system calls
- 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 = address space
  - Threads execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via traps:
  - system calls, page faults, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU) to construct virtual address spaces
  - Trap mechanism
- Details vary little across {BSD, OS X, Linux, Windows, ...}
- Recently: OS-Application trust model inverted due to untrustworthy operating systems – e.g., Trustzone, SGX, ...

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
- NB: `posix_spawn()`?
Evolution of the process model

- **1980s**: Code, heap, and stack
- **1990s**: Dynamic linking, threading
- **2000s**: Scalable memory allocators implement multiple arenas (e.g., as in jemalloc)
- Co-evolution with virtual memory (VM) research

Process address space: dd(1)

- Inspect dd process address space with `procstat -v`

```
root@beaglebone:/data # procstat -v 734
              PID    START       END    PR  RES  PRS  SHD  FLAG   TP    PATH
  734 0x8000 0x14000 0x16000 0x20031000 0x200339000 r-x 5 5 1 0 Cn-- vm /bin/dd
  734 0x200339000 0x200352000 0x20038000 0x200390000 rw- 1 0 1 0 Cn-- vm /libexec/ld-elf.so.1
  734 0x200390000 0x200390000 0x200390000 0x200390000 rw- 1 0 1 0 Cn-- vm /libexec/ld-elf.so.1
  734 0x200390000 0x200390000 0x200390000 0x200390000 rw- 1 0 1 0 Cn-- vm /libexec/ld-elf.so.1
  734 0x201000000 0x201600000 0x2025f8000 0x2025f8000 r-x 29 32 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x2025f8000 0x202660000 0x2025f0000 0x202660000 rw- 0 0 1 0 Cn-- df
  734 0x202660000 0x2026e60000 0x202660000 0x2026e60000 rw- 16 16 1 0 Cn-- df
  734 0x2026e60000 0x202858000 0x2026e60000 0x2026e60000 rw- 16 16 1 0 Cn-- df
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1
  734 0x202858000 0x202858000 0x202858000 0x202858000 rw- 351 360 31 14 Ch-- vm /libexec/ld-elf.so.1

r: read   C: Copy-on-write
w: write  D: Downward growth
x: execute 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 header using objdump -p:

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

Program Header:

```
0x70000001 off 0x0000469c vaddr 0x0000469c paddr 0x00000469c flags r--
filesz 0x00000158 memsz 0x00000158 align 2**2
PHDR off 0x00000034 paddr 0x00004034 paddr 0x00008034 flags r--
filesz 0x00000014 memsz 0x00000014 paddr 0x00008014 flags r-x
INTERP off 0x0000001e paddr 0x0000801e paddr 0x0000801e flags r--
filesz 0x00000015 memsz 0x00000015 paddr 0x00008015 flags r--
LOAD off 0x00000000 paddr 0x00004000 paddr 0x00008000 flags r-x
filesz 0x000000e0 memsz 0x000000e0 paddr 0x000080e0 flags r-x
LOAD off 0x0000047f paddr 0x0000147f paddr 0x0000147f flags r-x
filesz 0x0000047f memsz 0x0000047f paddr 0x0000847f flags r-x
LOAD off 0x00000018 paddr 0x00000818 paddr 0x00000818 flags r--
filesz 0x00000018 memsz 0x00000018 paddr 0x00008018 flags r--
LOAD off 0x00000480 paddr 0x00001480 paddr 0x00001480 flags r--
filesz 0x00000480 memsz 0x00000480 paddr 0x00008480 flags r--
NOTE off 0x00000012 paddr 0x00000812 paddr 0x00000812 flags r--
filesz 0x00000012 memsz 0x00000012 paddr 0x00008012 flags r--
```

Virtual memory (quick but painful primer)
Virtual memory (quick but painful primer)

- **Memory Management Unit (MMU)**
  - Transforms virtual addresses into physical addresses
  - Memory is laid out in virtual pages (4K, 2M, 1G, ...)
  - Control available only to the supervisor (historically)
  - Software handles failures (e.g., store to read-only page) via traps

- **Page tables**
  - SW-managed page tables provide virtual-physical mappings
  - Access permissions, page attributes (e.g., caching), dirty bit
  - Various configurations + traps implement BSS, COW, sharing, ...

- **Translation Look-aside Buffer (TLB)**
  - Hardware cache of entries – avoid walking pagetables
  - Content Addressable Memory (CAM): 48? 1024? entries
  - TLB tags: entries global or for a specific address-space ID (ASID)
  - Software- vs. hardware-managed TLBs

- **Hypervisors and IOMMUs:**
  - I/O performs direct memory access (DMA) via virtual address space

Role of the run-time linker (rtld)

- **Static linking:** program, libraries linked into one binary
  - Process address space laid out (and fixed) at compile time

- **Dynamic linking:** program, libraries in separate binaries
  - Shared libraries avoid code duplication, conserving memory
  - Shared libraries allow different update cycles, ABI ownership
  - Program binaries contain a list of their library dependencies
  - The run-time linker (rtld) loads and links libraries
  - Also used for plug-ins via dlopen(), dlsym()

- Three separate but related activities:
  - **Load:** Load ELF segments at suitable virtual addresses
  - **Relocate:** Rewrite position-dependent code to load address
  - **Resolve symbols:** Rewrite inline/PLT addresses to other code
Role of the run-time linker (rtld)

- When the `execve` system call starts the new program:
  - ELF binaries name their interpreter in ELF metadata
  - Kernel maps rtld and the application binary into memory
  - Userspace starts execution in rtld
  - rtld loads and links dynamic libraries, runs constructors
  - rtld calls `main()`

- Optimisations:
  - Lazy binding: don’t resolve all function symbols at load time
  - Prelinking: relocate, link in advance of execution
  - Difference is invisible – but surprising to many programmers

Arguments and ELF auxiliary arguments

- C-program arguments are `argc`, `argv[]`, and `envv[]`:

- The run-time linker also accepts arguments from the kernel:
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
- $\text{pc}$ to **interrupt vector**: dedicated OS code to handle trap
- Key challenge: kernel must gain control safely, securely

| RISC | User $\text{pc}$ saved, handler $\text{pc}$ installed, control coprocessor (MMU, ...)  
Kernel address space becomes available for fetch/load/store  
Reserved registers in ABI ($k0, k1$) or banking ($\text{spc, ssP, ...}$)  
Software must save other state (i.e., other registers) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CISC</td>
<td>HW saves context to in-memory trap frame (variably sized?)</td>
</tr>
</tbody>
</table>

- User context switch:
  - (1) trap to kernel, (2) save register context; (3) optionally change address space, (4) restore another register context; (5) trap return

For next time

- More on traps and system calls
- Virtual memory support for the process model
- Review ideas from the first lab report
- McKusick, et al: Chapter 6 *(Memory Management)*
- Optional: Anderson, et al, on *Scheduler Activations*
  - (Exercise: where can we find scheduler-activation-based concurrent programming models today?)