The Process Model (1)

L41 Lecture 3

Dr Robert N. M. Watson

13 November 2017

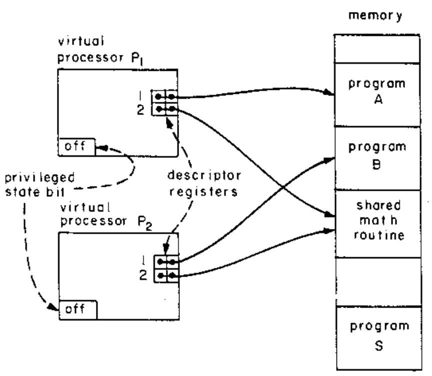
Reminder: last time

- DTrace
- The **probe effect**
- The kernel: Just a C program?
- A little on kernel dynamics: How work happens

This time: The process model

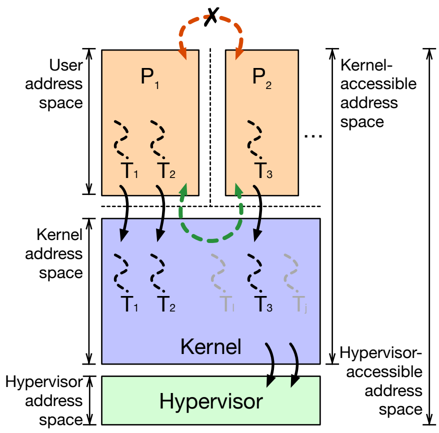
- The process model and its evolution
- Brutal (re, pre)-introduction to virtual memory
- Where do programs come from?
- Traps and system calls
- Reading for next time

The *Process Model*: 1970s foundations



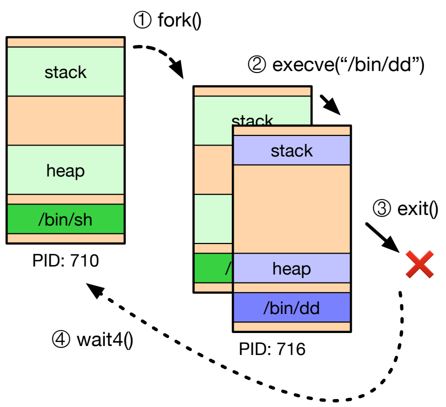
- Saltzer and Schroeder, **The Protection of Information in Computer Systems**, SOSP'73, October 1973. (CACM 1974)
- **Multics process model**
  - 'Program in execution'
  - **Process isolation** bridged by **controlled communication** via **supervisor** (kernel)
- Hardware foundations
  - Supervisor mode
  - Memory segmentation
  - Trap mechanism
- Hardware protection rings (Schroeder and Saltzer, 1972)

The **process model**: today



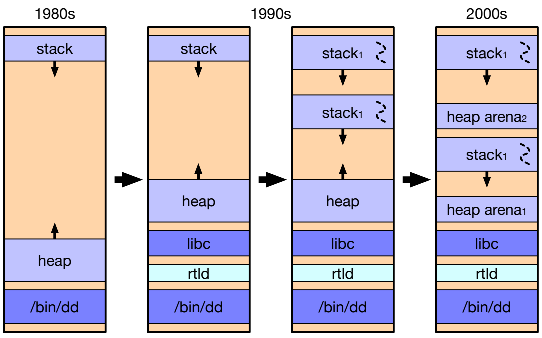
- ‘Program in execution’
  - **Process** ≈ address space
  - **Threads** execute code
- Unit of **resource accounting**
  - Open files, memory, …
- Kernel interaction via **traps**: system calls, page faults, …
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU) to construct **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
  - Acetta, et al: *Mach* microkernel (1986)
  - Nararro, et al: *Superpages* (2002)

Process address space: dd(1)

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

```
root@beaglebone:/data # procstat -v 734
  PID      START        END PRT  RES PRES REF SHD FLAG TP PATH
  734     0x8000     0xd000 r-x    5    5   1   0 CN-- vn /bin/dd
  734    0x14000    0x16000 rw-    2    2   1   0 ---- df
  734 0x20014000 0x20031000 r-x   29   32  31  14 CN-- vn /libexec/ld-elf.so.1
  734 0x20038000 0x20039000 rw-    1    0   1   0 C--- vn /libexec/ld-elf.so.1
  734 0x20039000 0x20052000 rw-   16   16   1   0 ---- df
  734 0x20100000 0x2025f000 r-x  351  360  31  14 CN-- vn /lib/libc.so.7
  734 0x2025f000 0x20266000 ---    0    0   1   0 ---- df
  734 0x20266000 0x2026e000 rw-    8    0   1   0 C--- vn /lib/libc.so.7
  734 0x2026e000 0x20285000 rw-    7  533   2   0 ---- df
  734 0x20400000 0x20c00000 rw-  526  533   2   0 --S- df
  734 0xbffe0000 0xc0000000 rwx    3    3   1   0 ---D df
```

r: read   C: Copy-on-write

w: write   D: Downward growth

x: execute   S: Superpage

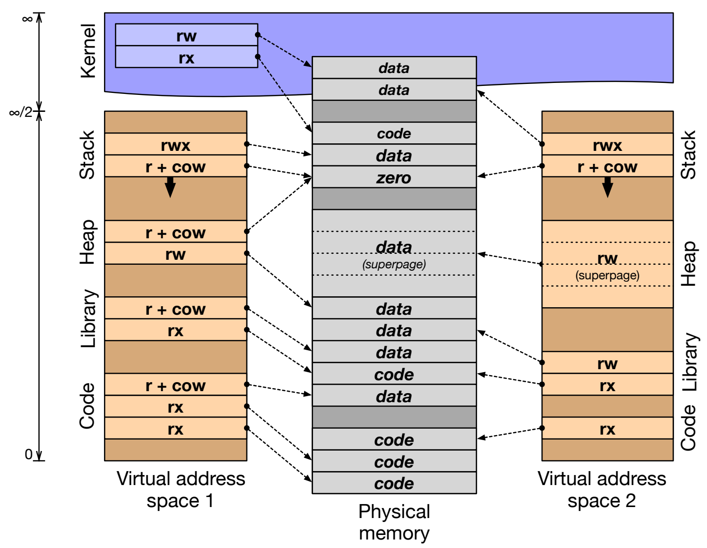
ELF binaries

- 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 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)



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 addres 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)

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

- 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[]`:

```
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_EXECPATH         0xbfffffc4
  716 dd               AT_OSRELDATE        1100062
  716 dd               AT_NCPUS            1
  716 dd               AT_PAGESIZES        0xbfffff9c
  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, securely

RISC	User $pc saved, handler $pc installed, control coprocessor (MMU, …) Kernel address space becomes available for fetch/load/store Reserved registers in ABI ($k0, $k1) or banking ($pc, $sp, …) Software must save other state (i.e., other registers)
CISC	HW saves context to in-memory trap frame (variably sized?)

- 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?)