L41 - Lecture 3: The Process Model (1)

Dr Robert N. M. Watson

2 November 2015

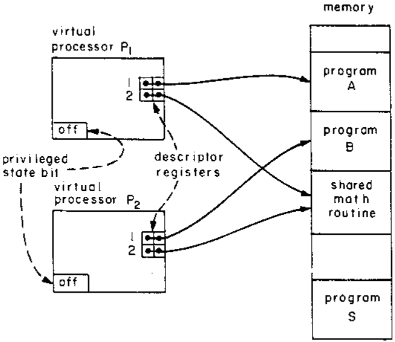
Reminder: last time

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

This time: the process model

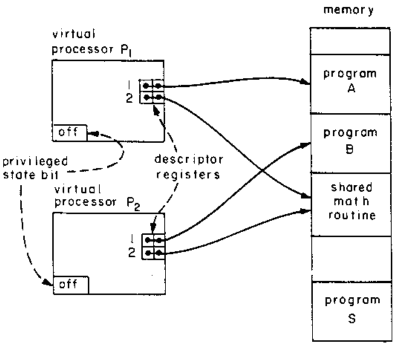
1. The process model and its evolution
2. Brutal (re,pre)-introduction to virtual memory
3. Where do programs come from?
4. Traps and system calls
5. Reading for next time

The process model: 1970s foundations



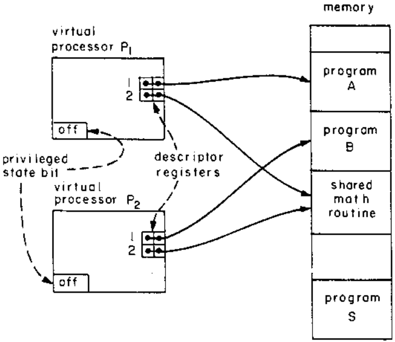
- Saltzer and Schroeder, *The Protection of Information in Computer Systems*, SOSP'73, October 1973. (CACM 1974)

The process model: 1970s foundations



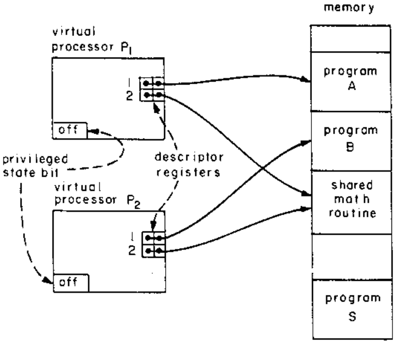
- Saltzer and Schroeder, *The Protection of Information in Computer Systems*, SOSP'73, October 1973. (CACM 1974)
- *Multics* process model
  - 'Program in execution'
  - *Process isolation* bridged by *controlled communication* via supervisor (kernel)

The process model: 1970s foundations



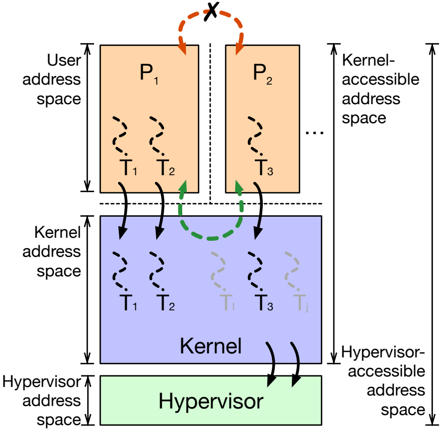
- Saltzer and Schroeder, *The Protection of Information in Computer Systems*, SOSP'73, October 1973. (CACM 1974)
- *Multics* process model
    - 'Program in execution'
    - *Process isolation* bridged by *controlled communication* via supervisor (kernel)
- Hardware foundations
    - Supervisor mode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

The process model: 1970s foundations

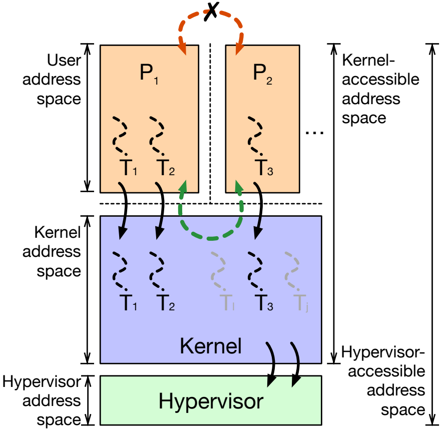


- Saltzer and Schroeder, *The Protection of Information in Computer Systems*, SOSP'73, October 1973. (CACM 1974)
- *Multics* process model
  - 'Program in execution'
  - *Process isolation* bridged by *controlled communication* via supervisor (kernel)
- Hardware foundations
  - Supervisor mode
  - Memory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

The process model: today

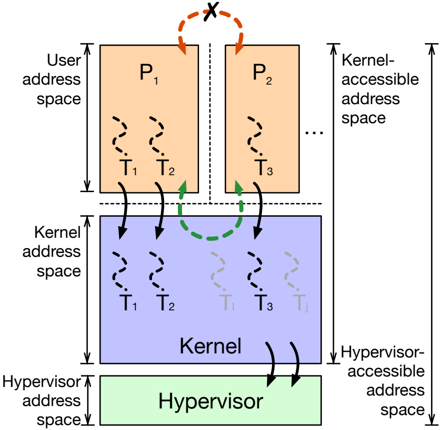


The process model: today



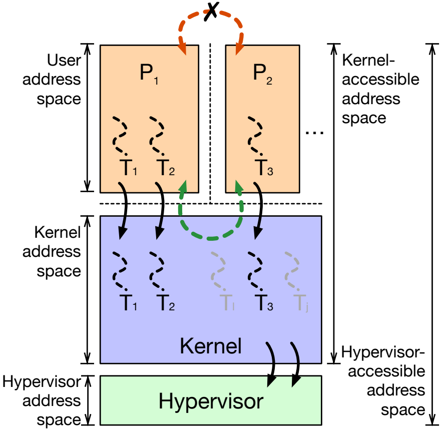
- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code

The process model: today



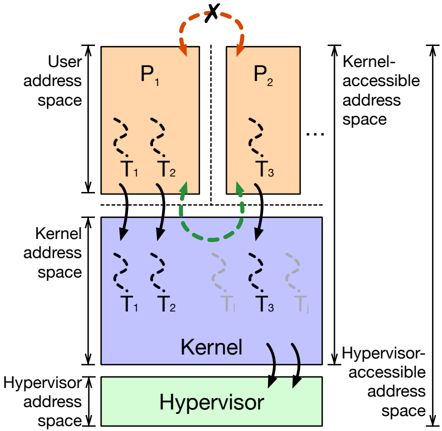
- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code
- Unit of resource accounting
  - Open files, memory, ...

The process model: today



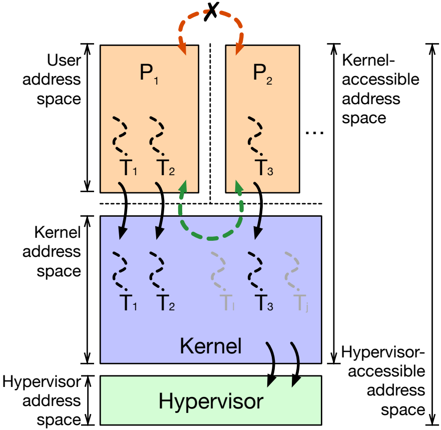
- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via *traps*: system calls, page faults, ...

The process model: today



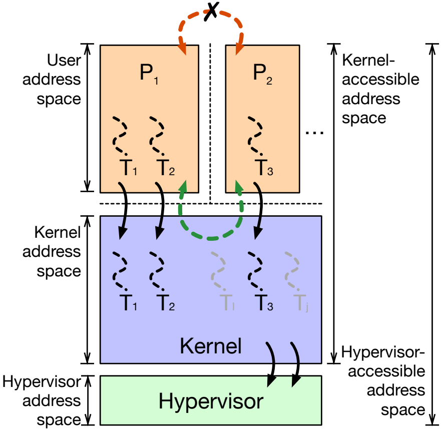
- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via *traps*: system calls, page faults, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU)
  - Trap mechanism

The process model: today



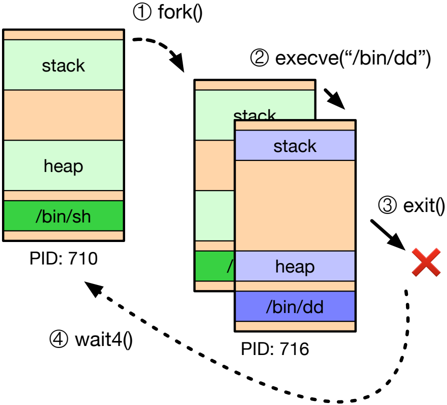
- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via *traps*: system calls, page faults, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU)
  - Trap mechanism
- Details vary little across {BSD, OS X, Linux, Windows, ...}

The process model: today

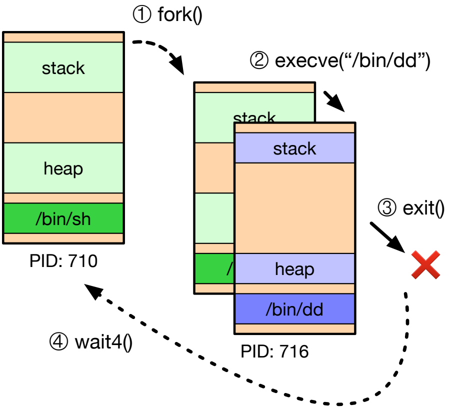


- ‘Program in execution’
  - Process ≈ address space
  - ‘Threads’ execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via *traps*: system calls, page faults, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU)
  - Trap mechanism
- Details vary little across {BSD, OS X, Linux, Windows, ...}
- Recently: OS-App trust model inverted: Trustzone, SGX

The UNIX process life cycle

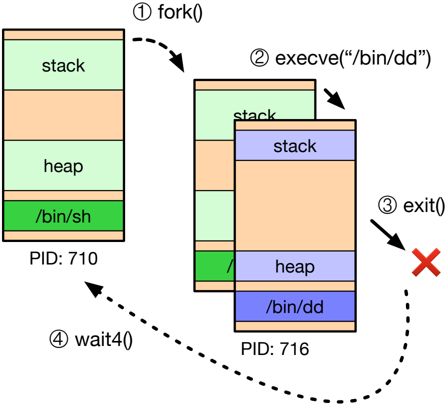


The UNIX process life cycle



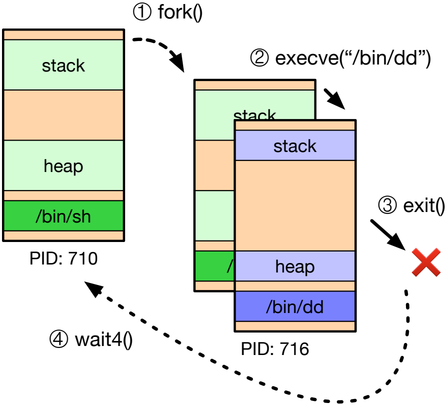
- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)

The UNIX process life cycle



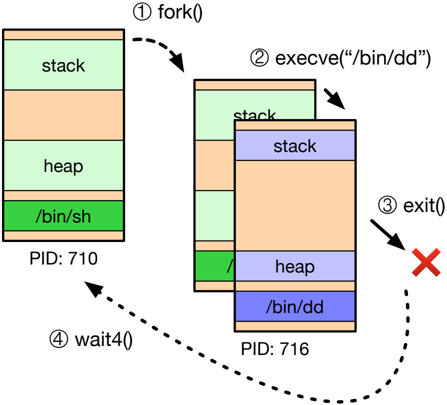
- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution

The UNIX process life cycle



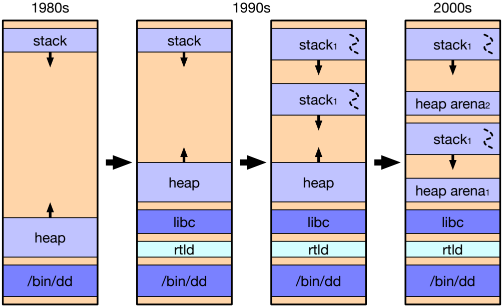
- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution
- `exit()`
  - Process can terminate self (or be terminated)

The UNIX process life cycle

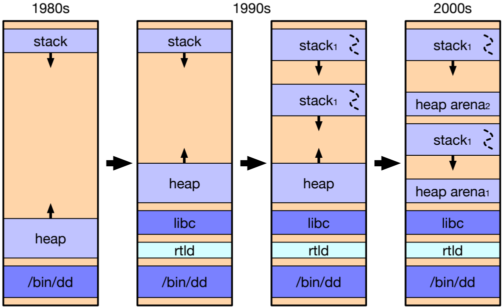


- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution
- `exit()`
  - Process can terminate self (or be terminated)
- `wait4` (et al)
  - Parent can await exit status

Process model evolution

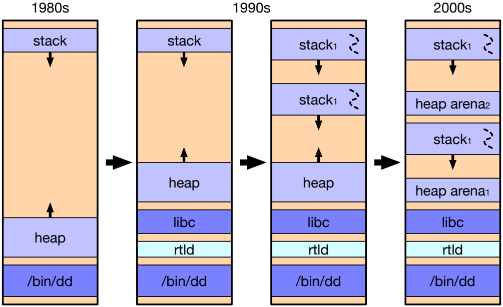


Process model evolution



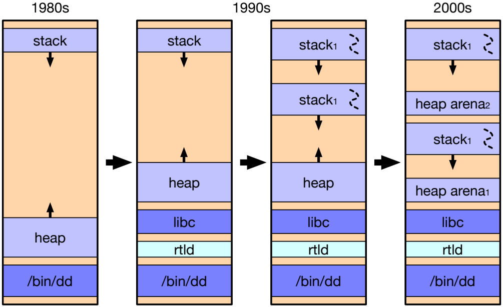
- 1980s: Code, heap, and stack

Process model evolution



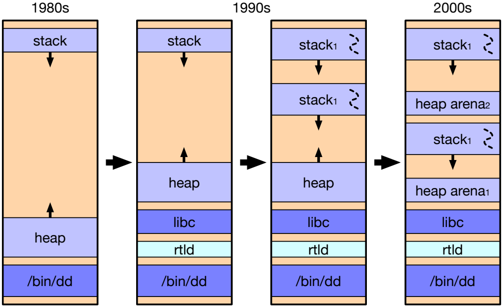
- 1980s: Code, heap, and stack
- 1990s: Dynamic linking, multithreading

Process model evolution



- 1980s: Code, heap, and stack
- 1990s: Dynamic linking, multithreading
- 2000s: Scalable memory allocators implement multiple *arenas* (e.g., `jemalloc`)

Process model evolution



- 1980s: Code, heap, and stack
- 1990s: Dynamic linking, multithreading
- 2000s: Scalable memory allocators implement multiple *arenas* (e.g., `jemalloc`)
- Coevolution with virtual memory research (Acetta, et al: *Mach* microkernel (1986); Navarro, et al *Superpages* (2002))

Process address space: `dd`

Process address space: `dd`

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

Process address space: dd

- 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.
  734 0x20038000 0x20039000 rw-    1    0   1   0 C--- vn /libexec/ld-elf.
  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
```

Process address space: dd

- 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.
  734 0x20038000 0x20039000 rw-    1    0   1   0 C--- vn /libexec/ld-elf.
  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
w: write
x: execute

C: Copy-on-write
D: Downward growth
S: Superpage

ELF binaries

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`:

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
```

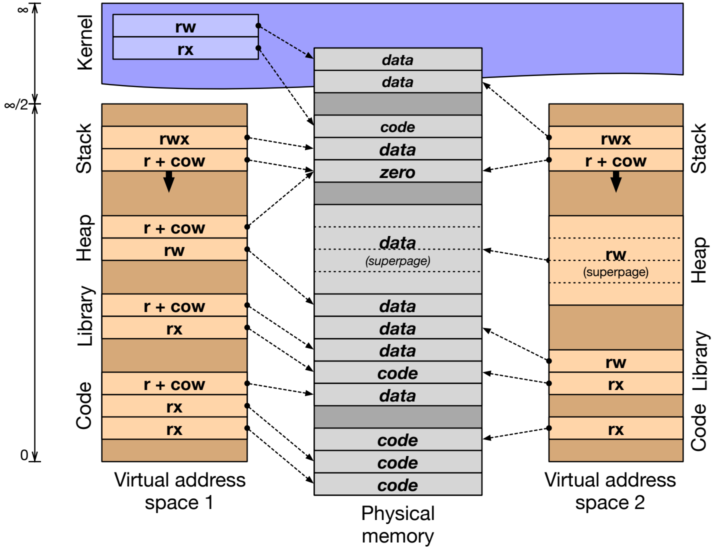
ELF binaries

- 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)



Virtual memory (quick but painful primer) (cont)

Virtual memory (quick but painful primer) (cont)

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

Virtual memory (quick but painful primer) (cont)

- Memory Management Unit (MMU)
  - Transforms *virtual addresses* into *physical addresses*
  - Memory is laid out in *pages* (4K, 2M, 1G...)
  - Control available only to the supervisor
  - Software handles failures (e.g., permissions) via traps
- Page tables
  - SW-managed *page tables* provide *virtual-physical mappings*
  - Access permissions, page attributes (e.g., caching)
  - Various configurations + traps implement BSS, COW, sharing, ...

Virtual memory (quick but painful primer) (cont)

- Memory Management Unit (MMU)
  - Transforms *virtual addresses* into *physical addresses*
  - Memory is laid out in *pages* (4K, 2M, 1G...)
  - Control available only to the supervisor
  - Software handles failures (e.g., permissions) via traps
- Page tables
  - SW-managed *page tables* provide *virtual-physical mappings*
  - Access permissions, page attributes (e.g., caching)
  - Various configurations + traps implement BSS, COW, sharing, ...
- The 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 process
  - Software- vs. hardware-managed TLBs

Virtual memory (quick but painful primer) (cont)

- Memory Management Unit (MMU)
  - Transforms *virtual addresses* into *physical addresses*
  - Memory is laid out in *pages* (4K, 2M, 1G...)
  - Control available only to the supervisor
  - Software handles failures (e.g., permissions) via traps
- Page tables
  - SW-managed *page tables* provide *virtual-physical mappings*
  - Access permissions, page attributes (e.g., caching)
  - Various configurations + traps implement BSS, COW, sharing, ...
- The 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 process
  - Software- vs. hardware-managed TLBs
- Hypervisors and *I/O MMUs*: I/O sources as ‘processes’

Role of the run-time linker (`rtld`)

Role of the run-time linker (`rtld`)

- Static linking: program and libraries linked into a single binary

Role of the run-time linker (`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
  - 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()`

Role of the run-time linker (`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
  - 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:
  - *Loading*: Load ELF segments at suitable virtual addresses
  - *Relocating*: Rewrite position-dependent code to load address
  - *Symbol resolution*: Rewrite inline addresses to other loaded code

Role of the run-time linker (`rtld`) (cont)

```
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 execution in `rtld`
  - `rtld` loads and links dynamic libraries, runs constructors
  - `rtld` calls `main()`

Role of the run-time linker (`rtld`) (cont)

```
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 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

Arguments and ELF auxiliary arguments

Arguments and ELF auxiliary arguments

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

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
```

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:

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

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

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

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*