The Process Model (1)

Lecture 3, Part 2: Processes In Practice

Prof. Robert N. M. Watson

2026-2027

Process address space: dd(1)

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

```
root@rpi4-000:~ # procstat -v 20921
  PID              START                END PRT   RES PRES REF SHD FLAG  TP PATH
20921            0x200000           0x203000 r--     3    8   3   0 CN--- vn /bin/dd
20921            0x212000           0x217000 r-x     5    8   3   0 CN--- vn /bin/dd
20921            0x226000           0x227000 r--     1    0   1   0 C---- vn /bin/dd
20921            0x236000           0x237000 rw-     1    1   1   0 ----- df
20921           0x40236000         0x4023c000 r--     6   27  51   0 CN--- vn /libexec/ld-elf.so.1
20921           0x4024b000         0x40260000 r-x    21   27  51   0 CN--- vn /libexec/ld-elf.so.1
20921           0x4026f000         0x40270000 r--     1    0   2   0 C---- vn /libexec/ld-elf.so.1
20921           0x40270000         0x40271000 rw-     1    0   2   0 C---- vn /libexec/ld-elf.so.1
20921           0x40280000         0x402a3000 rw-    27   27   1   0 ----- df
20921           0x402a3000         0x402ab000 r--     8   19  37   0 CN--- vn /lib/libutil.so.9
20921           0x402ab000         0x402ba000 ---     0    0   0   0 CN--- --
20921           0x402ba000         0x402c5000 r-x    11   19  37   0 CN--- vn /lib/libutil.so.9
20921           0x402c5000         0x402d4000 ---     0    0   0   0 CN--- --
20921           0x402d4000         0x402d5000 r--     1    0   1   0 C---- vn /lib/libutil.so.9
20921           0x402d5000         0x402e4000 ---     0    0   0   0 CN--- --
20921           0x402e4000         0x402e5000 rw-     1    0   1   0 C---- vn /lib/libutil.so.9
20921           0x402e5000         0x402e7000 rw-     0    0   0   0 ----- --
20921           0x402e7000         0x40360000 r--    81  376  54   0 CN--- vn /lib/libc.so.7
20921           0x40360000         0x4036f000 ---     0    0   0   0 CN--- --
20921           0x4036f000         0x404a7000 r-x   272  376  54   0 CN--- vn /lib/libc.so.7
20921           0x404a7000         0x404b6000 ---     0    0   0   0 CN--- --
20921           0x404b6000         0x404c0000 r--    10    0   1   0 C---- vn /lib/libc.so.7
20921           0x404c0000         0x404cf000 ---     0    0   0   0 CN--- --
20921           0x404cf000         0x404d6000 rw-     7    0   1   0 C---- vn /lib/libc.so.7
20921           0x404d6000         0x40700000 rw-    17   17   1   0 ----- df
20921           0x40800000         0x41000000 rw-    48   48   1   0 ----- df
20921     0xffffbffff000     0xfffffffdf000 ---     0    0   0   0 ----- --
20921     0xfffffffdf000     0xffffffffff000 rw-     4    4   1   0 ---D- df
20921     0xffffffffff000 0x1000000000000 r-x     1    1  18   0 ----- ph
```

dd

rtld

libutil

libc

jemalloc heap

stack

vdso / sigcode

r: read x: execute D: Downward growth S: Superpage

w: write C: Copy-on-write N: Needs copy

* Linux equivalent procstat –v is /proc/X/maps

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@rpi4-000:~ # objdump -p /bin/dd
/bin/dd:     file format elf64-littleaarch64

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000200040 paddr 0x0000000000200040 align 2**3
         filesz 0x0000000000000268 memsz 0x0000000000000268 flags r--
  INTERP off    0x00000000000002a8 vaddr 0x00000000002002a8 paddr 0x00000000002002a8 align 2**0
         filesz 0x0000000000000015 memsz 0x0000000000000015 flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000200000 paddr 0x0000000000200000 align 2**16
         filesz 0x0000000000002f3c memsz 0x0000000000002f3c flags r--
    LOAD off    0x0000000000002f3c vaddr 0x0000000000212f3c paddr 0x0000000000212f3c align 2**16
         filesz 0x00000000000034a4 memsz 0x00000000000034a4 flags r-x
    LOAD off    0x00000000000063e0 vaddr 0x00000000002263e0 paddr 0x00000000002263e0 align 2**16
         filesz 0x00000000000001a8 memsz 0x00000000000001a8 flags rw-
    LOAD off    0x0000000000006588 vaddr 0x0000000000236588 paddr 0x0000000000236588 align 2**16
         filesz 0x00000000000001e8 memsz 0x00000000000004d0 flags rw-
 DYNAMIC off    0x00000000000063f0 vaddr 0x00000000002263f0 paddr 0x00000000002263f0 align 2**3
         filesz 0x0000000000000180 memsz 0x0000000000000180 flags rw-
   RELRO off    0x00000000000063e0 vaddr 0x00000000002263e0 paddr 0x00000000002263e0 align 2**0
         filesz 0x00000000000001a8 memsz 0x00000000000001a8 flags r--
...
```

ELF encodes information on the target process environment: 64-bit ELF, little endian, aarch64 ABI

objdump inspects ELF files

Program header lists a set of segments that will be mapped into virtual memory

E.g., code, rodata, globals / BSS, linker metadata

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@rpi4-000:~ # objdump -p /bin/dd
/bin/dd:     file format elf64-littleaarch64


Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000200040 paddr 0x0000000000200040 align 2
         filesz 0x0000000000000268 memsz 0x0000000000000268 flags r--
  INTERP off    0x00000000000002a8 vaddr 0x00000000002002a8 paddr 0x00000000002002a8 align 2**0
         filesz 0x0000000000000015 memsz 0x0000000000000015 flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000200000 paddr 0x0000000000200000 align 2**16
         filesz 0x0000000000002f3c memsz 0x0000000000002f3c flags r--
    LOAD off    0x0000000000002f3c vaddr 0x0000000000212f3c paddr 0x0000000000212f3c align 2**16
         filesz 0x00000000000034a4 memsz 0x00000000000034a4 flags r-x
    LOAD off    0x00000000000063e0 vaddr 0x00000000002263e0 paddr 0x00000000002263e0 align 2**16
         filesz 0x00000000000001a8 memsz 0x00000000000001a8 flags rw-
    LOAD off    0x0000000000006588 vaddr 0x0000000000236588 paddr 0x0000000000236588 align 2**16
         filesz 0x00000000000001e8 memsz 0x00000000000004d0 flags rw-
 DYNAMIC off    0x00000000000063f0 vaddr 0x00000000002263f0 paddr 0x00000000002263f0 align 2**3
         filesz 0x0000000000000180 memsz 0x0000000000000180 flags rw-
   RELRO off    0x00000000000063e0 vaddr 0x00000000002263e0 paddr 0x00000000002263e0 align 2**0
         filesz 0x00000000000001a8 memsz 0x00000000000001a8 flags r--
...
```

ELF interpreter path name (run-time linker)

Memory mapped content in the file including executable code, global variables, linker metadata, etc.

Permissions on ELF segments are implemented using MMU page permissions

Virtual memory (quick but painful primer)

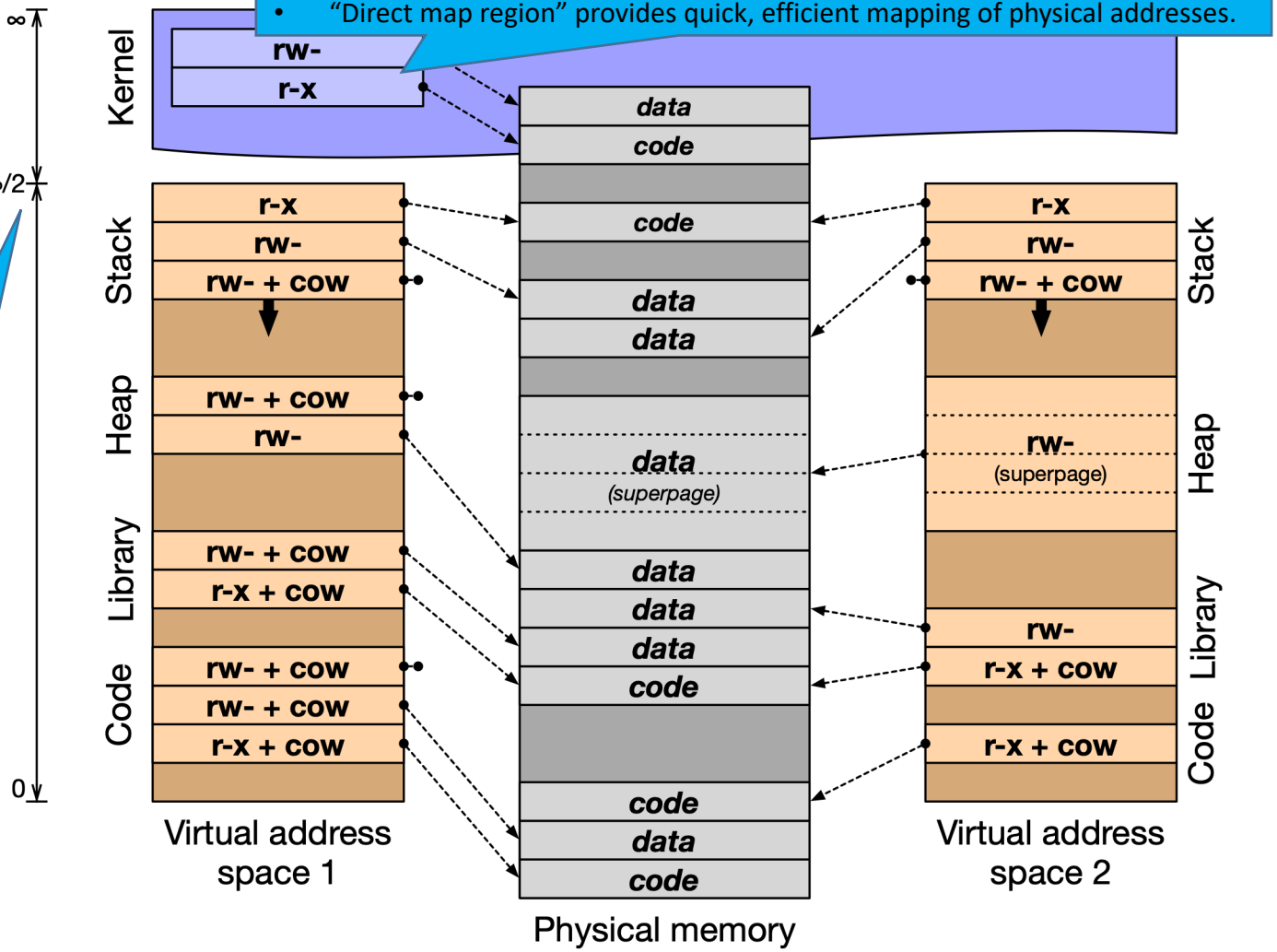
- **Memory Management Unit (MMU)**
  - Transforms **virtual addresses** into **physical addresses**
  - Affects **loads**, **stores**, and **instruction fetches**
  - Memory is laid out in **virtual pages** (4K, 2M, 1G, …)
  - Control available only to the supervisor (historically)
  - Software handles 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

* In some instruction-set architectures – e.g., Armv8-A – these are called ‘exceptions’

Virtual memory (quick but painful primer) (1/3)

- Kernel address space is also managed using the MMU.
- Unified global kernel address space (with certain exceptions).
- Kernel mappings may "borrow" pages from userspace.
- "Direct map region" provides quick, efficient mapping of physical addresses.

- A fixed partition between user and kernel address space makes checks quick and easy to implement.
- On some architectures (e.g., ARMv8-A), this point is configurable
- The kernel also needs substantial address space. It's a squeeze in 32 bits, and fine with 64.

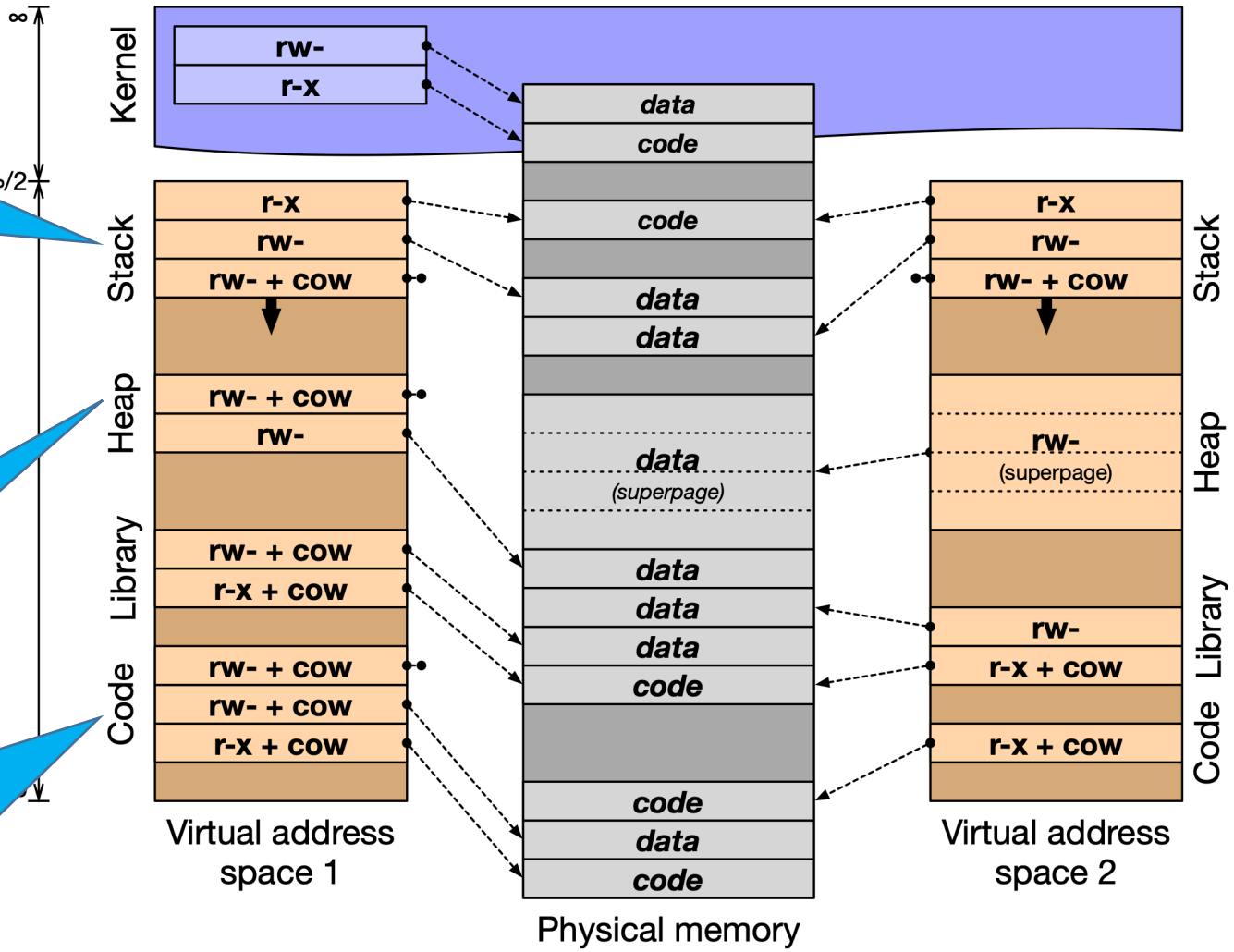


Virtual memory (quick but painful primer) (2/3)

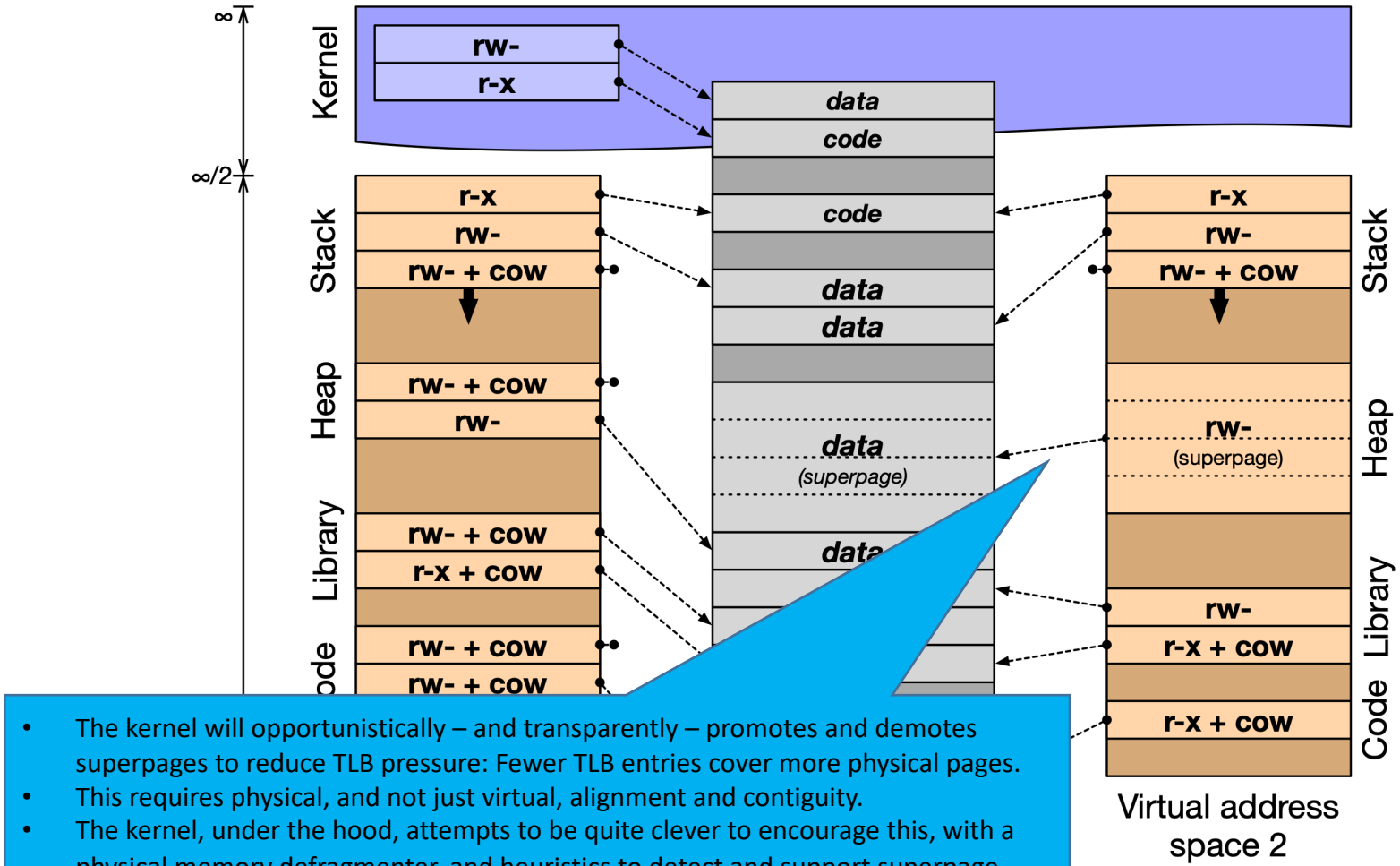
- The userspace stack "grows down" on demand

- Pages will be zero filled on demand – e.g., for BSS or heap memory

Memory mappings from program binaries include:
- Read-write (COW) demand-zeroed pages (BSS)
- Read-write (COW) mappings of data
- Read-execute mappings of program text (COW)



Virtual memory (quick but painful primer (3/3)



- The kernel will opportunistically – and transparently – promotes and demotes superpages to reduce TLB pressure: Fewer TLB entries cover more physical pages.
- This requires physical, and not just virtual, alignment and contiguity.
- The kernel, under the hood, attempts to be quite clever to encourage this, with a physical memory defragmenter, and heuristics to detect and support superpage mappings of non-superpage-aligned data in files, such as for program text.

Role of the run-time loader (rtld) (1/2)

- **Static linking**: program + libraries linked into a single binary
  - Process address space laid out (+ 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 and shared libraries both contains lists of their own **library dependencies**
  - The run-time linker (`rtld`) loads and links libraries
  - Also used for plug-ins via `dlopen()`, `dlsym()`

Role of the run-time loader (rtld) (2/2)

- Five separate but related activities:
  - **Load**: Load ELF segments at suitable virtual addresses
  - **Relocate**: Update GOT, PLT entries to reflect dynamic load addresses of external dependencies – e.g., other libraries
  - **Resolve symbols**: Determine where GOT/PLT entries should point in other libraries
  - **Call constructors**, the **main function**, and **destructors**
  - **Enable dynamic extensions** through dlopen(), dlsym(), etc., that allow dynamic specification of dependencies + looking up symbols in a target library (e.g., a plugin)
- The run-time loader also plays a role in debugging
  - Its internal state is understood by the debugger

Starting a binary (and dependencies)

```
root@rpi4-000:~ # ldd /bin/dd
/bin/dd:
        libutil.so.9 => /lib/libutil.so.9 (0x402a3000)
        libc.so.7 => /lib/libc.so.7 (0x402e7000)
```

- When the `execve` system call starts the new program:
  - ELF binaries specify their **interpreter** in ELF metadata
  - Kernel maps `rtld` and the application binary into memory
  - Userspace starts execution in `rtld`
  - `rtld` loads, links, and initialises dynamic libraries
  - `rtld` runs library and application binary constructors
  - `rtld` calls binary's `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@rpi4-000:~ # procstat -c 20921
  PID COMM                ARGS
20921 dd                  dd if=/dev/zero of=/dev/null bs=1k
```

- The run-time loader accepts additional arguments from the kernel:

```
root@rpi4-000:~ # procstat -x 20921
  PID COMM             AUXV             VALUE
20921 dd               AT_PHDR          0x200040
20921 dd               AT_PHENT         56
20921 dd               AT_PHNUM         11
20921 dd               AT_PAGESZ        4096
20921 dd               AT_FLAGS         0
20921 dd               AT_ENTRY         0x213148
20921 dd               AT_BASE          0x40236000
20921 dd               AT_EHDRFLAGS     0
20921 dd               AT_EXECPATH      0xffffffffefd8
20921 dd               AT_OSRELDATE     1300138
20921 dd               AT_CANARY        0xffffffffef98
20921 dd               AT_CANARYLEN     64
20921 dd               AT_NCPUS         4
20921 dd               AT_PAGESIZES     0xffffffffef80
20921 dd               AT_PAGESIZESLEN  24
20921 dd               AT_TIMEKEEP      0xffffffffc0
20921 dd               AT_STACKPROT     NONEXECUTABLE
20921 dd               AT_HWCAP         0x83
20921 dd               AT_HWCAP2        0
20921 dd               AT_BSDFLAGS      0x1
20921 dd               AT_ARGC          4
20921 dd               AT_ARGV          0xffffffffea68
20921 dd               AT_ENVC          24
20921 dd               AT_ENVV          0xffffffffea90
20921 dd               AT_PS_STRINGS    0xffffffffefe0
```

Address of binary's ELF program header

Entry address for binary

Base address of binary (or rtld if used)

Many system parameters

Command-line arguments and environment above stack

Writable command line visible in ps(1)

Wrapping up

- In this lecture, we have talked about:
  - The basics and history of the process model
  - A few gory implementation details

- Our next lecture, also on the process model, will explore:
  - Traps and system calls
  - Ideas about isolation, security, and reliability
  - More gory details of the VM system