# The Monolithic xv6 Kernel & RISC-V ISA

Computer Science 432 — Lecture 5 — Duane Bailey

*February 16, 2022*

# Announcements

- Lab 2 (xv6) is currently out; due Monday before the next lab
- Small Group meetings for Lab 1 (mex) today and tomorrow (Knuth)
- No office hours on Friday.

- Please follow along with Syllabus: Read Chapter 3 for Monday.

# Kernel Structure

- We recall that the purpose of the kernel is:
  - Multiplexing: Fairly give all processes the same view of the machine
  - Isolation: Protect processes and the kernel from getting "wrecked" by problems & bugs
  - Interaction: Support the intentional interaction of processes

- There are choices to be made:
  - Do we have a kernel? If no: similar to approach of embedded devices.
    - No overhead of kernel.
    - No protection or support, generally.
  - Do we have a *monolithic kernel*? An us-*vs*-them approach. Linux, many unix versions.
    - Kernel encapsulates all code that runs in supervisor mode.
  - Do we have a *microkernel*? Mach and many derivatives, including XNU, Darwin, et al.
    - Kernel is very small, supporting message-passing among user-space resource controllers

# Hardware Abstractions

- Hardware components are provided as *services*.
  - Storage accessed through a *file system* using opaque/abstract *file descriptors*.
  - CPU is shared through transparent *context switching* managed by a *scheduler*.
  - Memory is constructed using *exec* which works with a system of *virtual memory*.
  - Processes communicate through *file descriptors*

- Kernel in xv6: a monolithic kernel that balances *convenience* vs. *isolation*

# Privilege Modes

- RISC-V supports three execution privilege modes:
  - *Machine mode:* a mode for booting and configuring machine.
    - Flat physical memory.
    - No limit on instructions.
    - xv6 uses this mode only during boot process.
  - *Supervisor mode:* a mode for executing a monolithic kernel.
    - Virtual memory, with ability to change memory mapping.
    - Access to privileged instructions.
    - xv6's monolithic kernel runs, for the most part, in Supervisor Mode
  - *User mode:* mode for user-written programs and process execution.
    - Virtual, protected memory.
    - Instructions limited.
    - Processes use this mode. *Userland.*

# Moving Between Modes

- *Isolation* is established by severely constraining movement between modes.
  - Generally, you can move into a more protected modes through a *call mechanism*
    - A single instruction, `ecall`, controls access to Supervisor mode.
    - When supervisor mode is entered, memory layout is modified:
      - Kernel-specific mapping of virtual memory
      - Kernel-specific stack
      - Any communicated values must be manually moved between userland and kernel.
    - Kernel validates call before fully entering into the kernel proper.
    - When kernel is finished, it performs a `sret` to reverse the process
    - Main observation: Kernel must be able to do its work without the user noticing.

  - Similar approach in moving between supervisor and machine modes.

# Kernel Organization

- Kernel is a collaboration of services
  - Most services have source & header
  - All kernel services run in Supervisor
  - No isolation of services within kernel
  - But: easier to communicate within kernel
  - *Mistakes here cause total system failure*
- Our goal: understand motivation for decisions made in this code.

| File | Description |
|---|---|
| bio.c | Disk block cache for the file system. |
| console.c | Connect to the user keyboard and screen. |
| entry.S | Very first boot instructions. |
| exec.c | exec() system call. |
| file.c | File descriptor support. |
| fs.c | File system. |
| kalloc.c | Physical page allocator. |
| kernelvec.S | Handle traps from kernel, and timer interrupts. |
| log.c | File system logging and crash recovery. |
| main.c | Control initialization of other modules during boot. |
| pipe.c | Pipes. |
| plic.c | RISC-V interrupt controller. |
| printf.c | Formatted output to the console. |
| proc.c | Processes and scheduling. |
| sleeplock.c | Locks that yield the CPU. |
| spinlock.c | Locks that don't yield the CPU. |
| start.c | Early machine-mode boot code. |
| string.c | C string and byte-array library. |
| swtch.S | Thread switching. |
| syscall.c | Dispatch system calls to handling function. |
| sysfile.c | File-related system calls. |
| sysproc.c | Process-related system calls. |
| trampoline.S | Assembly code to switch between user and kernel. |
| trap.c | C code to handle and return from traps and interrupts. |
| uart.c | Serial-port console device driver. |
| virtio_disk.c | Disk device driver. |
| vm.c | Manage page tables and address spaces. |

Figure 2.2: Xv6 kernel source files.

# Process Structure

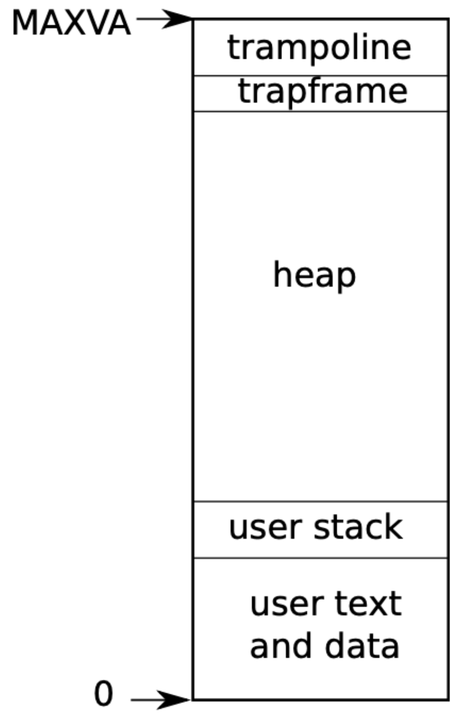

Figure 2.3: Layout of a process's virtual address space

- The main unit of isolation is the *process*
  - Processes represent a thread of execution; a locus of control
  - Have their own virtual memory mapping
  - Dedicated user text (code) and data
  - Dedicated user stack (grows down; note position)
  - Dedicated heap (grows up; note position)
  - A dedicated "trapframe" — writeable area for transition between user & kernel
  - A shared "trampoline" — code used to transition between user & kernel

- In the kernel, a process is represented by a `struct proc`
  - This structure maintains resources dedicated to process: page table, state, stack, etc.

# Booting and First Process

- Boot code:
  - Bootloader is stored in ROM.  Loads kernel into memory at 0x8000 0000 (devices below)
  - Control transferred to kernel (at `_entry`), in machine mode.
- Kernel's _entry:
  - Sets up a simple 4K stack (one per hardware thread or *hart*), jumps to C code, `start`.
  - This routine sets up initial page tables, initializes timer interrupts.
  - Sets registers to appear as though there had been a supervisor->machine mode call
  - "Returns" to `main`.
- Kernel's main:
  - Initializes devices (eg. console)
  - From `userinit` routine, creates the initial process, `init`, with pid 1.
- Initial process (*userland*!)
  - Calls exec (reentering kernel) and runs `/init`
  - This program creates the `console` (if necessary) and establishes descriptors 0, 1, and 2; forks `sh`.

# The RISC-V Machine

- Reduced Instruction Set Computer (RISC), version V
  - Open source Instruction Set Architecture (ISA)
  - Open source hardware description
  - Developed at Berkeley
  - Early but wide-spread adoption
- 64 bit datapath
  - 32-bit, 3-address instructions.
  - 64-bit pointers (38 bits used)
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit doubles
  - 32 general purpose registers.
  - Many (hundreds) *computer status registers* (CSRs): hart ids, uptime, retired instruction counts, page tables, etc. (Not unusual to have these. Unusual because they're open.)

# Register file

| reg | name | saver | description |
|---|---|---|---|
| x0 | zero | | hardwired zero |
| x1 | ra | caller | return address |
| x2 | sp | callee | stack pointer |
| x3 | gp | | global pointer |
| x4 | tp | | thread pointer |
| x5-7 | t0-2 | caller | temporary registers |
| x8 | s0/fp | callee | saved register / frame pointer |
| x9 | s1 | callee | saved register |
| x10-11 | a0-1 | caller | function arguments / return values |
| x12-17 | a2-7 | caller | function arguments |
| x18-27 | s2-11 | callee | saved registers |
| x28-31 | t3-6 | caller | temporary registers |
| pc | | | program counter |

- 32 registers: x0 through x31, fully symmetric
- x0 is always 0, alias “zero”
- pc is the program counter
- ra typically used as return address register
- sp is typically used as the stack pointer
- fp (also: s0) is typically used for the frame pointer.
- arguments are passed in a0 through a7, then on the stack
- s0-s11 are callee “saved registers”; use after saving then restore
- t0-t6 are caller “temporary registers”

- Observation: process context is *mostly saved registers*.

# Instruction Set Overview

- Details found in the Porter book (see website)
  - Instruction set *Green Card* is 1 page summary (be aware: process layout is not xv6)
- A very small number of instructions
  - Most instructions are 3-address, with destination on the left
  - load and store are only memory instructions, with address on right
    - e.g. “ld  a1, 8(a0)”  load double into a1 from 8 off from base a0
    - e.g. “sd  a1, 8(a0)”  store double from a1 at 8 off from base a0
- Many are synthesized (by assembler) from other instructions
  - e.g. “call routine” is “jalr ra, routine”
  - e.g. “mv a1, a0” is “add a1, a0, x0”
- No condition codes: Conditional branches test and branch in one instruction
  - “Set” instructions conditionally compute 1 or 0 into destination register (wow!)
- Short stores to registers zero remaining bits (there’s a sign-extend for signed work)
- Multiply, divide, and remainder instructions.

# Calling Convention* — Caller

- To call a routine:
  - Put arguments in a0 through a7 (rest on stack, *in reverse order*).
  - Call routine:
    - PC+4 saved in ra.
    - jump to routine.
    - The call instruction is a shorthand macro for common case.
  - Routine does its work.
  - Result found in a0 (and possibly a1).

- Good resource: decoded binary, `user/cat.asm`

# Calling Convention* — Callee

- On entry:
  - sp points to top of stack
  - fp points to base of the caller frame
  - ra is the return address
  - a0..a7 contain arguments
- Typical entry protocol (effectively). Stack is 8-byte aligned.
  - The current stack pointer will become the next frame pointer.
  - Push on the return address.
  - Push on the old frame pointer
  - Push on saved registers
  - Push on locals.
- Actual entry protocol is tricky.

- Typical exit protocol:
  - Store return value in a0.
  - Restore old fp.
  - Restore sp to entry level
  - return, using ra
- On exit:
  - a0 is result
  - Other a-regs garbage