6.S081: OS Organization

Adam Belay <abelay@mit.edu>
Lecture topics

• OS design
  • System calls
  • Hardware isolation
  • Micro kernels vs. monolithic kernels

• System calls in xv6
OS Organization (from last time)

Kernel

Userspace

vi
gcc
nginx
sshd

System calls

Kernelspace

CPU
RAM
Disk
Net
Multiplexing

• Must handle multiple applications
• Need isolation between them
• But must share resources too
Strawman solution

- Applications use hardware directly
- OS acts as a library
Problem: Can’t multiplex

• Each app must periodically give up hardware
• But weak isolation
  • App forgets to give up -> nothing else runs
  • Apps has end-less loop -> nothing else runs
  • Can’t even kill the misbehaving app from another app
• This scheme is sometimes used in practice
  • Called cooperative scheduling
Bigger problem: Memory isolation

• All apps share physical memory
  • One app can overwrite another app’s memory
  • One app can overwrite the OS library

• No security!
UNIX interface

• Processes (instead of cores): fork()
  • OS transparently allocates cores
    • Save + Restore registers
  • Enforces that processes give them up
    • Periodically re-allocates cores
UNIX interface

• Memory (instead of physical memory): exec(), brk()
  • Each process has its own memory
  • OS decides where to place app in memory
  • OS enforces isolation between apps
  • OS stores image in file system (loaded with exec)
UNIX interface

- Files instead of disk blocks: open(), read(), write()
  - OS provides convenient names
  - OS allows files to be shared by apps and users
- Pipes instead of shared physical memory: pipe()
  - OS buffers data
  - OS stops sender if it transmits too fast
OS must be defensive

• An app shouldn’t be able to crash OS
• An app shouldn’t be able to break out of isolation
• Need strong isolation between apps and OS

Solution: use CPU hardware support
• User/kernel mode (privilege modes)
• Virtual memory
CPUs provide user/kernel mode

• Kernel mode: can execute “privileged” instructions
  • E.g., changing back to user mode
  • E.g., programming a timer chip
  • E.g., controlling virtual memory

• User mode: can’t execute “privileged” instructions
  • If it tries, faults to kernel

Plan: Run kernel is kernel mode, apps in user mode

* (RISC-V M-mode not used in this class)
CPUs provide virtual memory

- Page tables translate virtual address to physical
- Defines what physical memory an app can access
- OS sets up page table so each app can only access its memory
System calls

• Apps need to communicate with kernel
• Solution add instruction to change mode in controlled way
  • ecall on Risc-V
  • Enters kernel at pre-agreed instruction address
System calls

App -> printf() -> write()

... <- sys_write() <- trampoline

Userspace

Kernelspace

CPU
RAM
Disk
Net
Kernel is trusted computing base

• Kernel must be “correct”
  • Bugs could allow user apps to circumvent isolation

• Kernel must treat user apps as suspect
  • Each system call argument must be validated
  • User/kernel mode transitions must be set up correctly

• Requires a security mindset
  • Any bug could be a security exploit!
Aside: is isolation possible without HW support?

• Imagine no kernel/user mode or virtual memory
• Yes! Use a strongly-typed programming language
  • E.g., singularity OS
• The compiler is then the TCB
• But HW support is the most common plan
Monolithic kernels

- OS runs in kernel space
- xv6 does this, so does Linux
- Kernel interface == system call interface
- Kernel is one big program with everything (filesystem, drivers, memory management, etc.)

Pros:
- Easy for subsystems to cooperate
  - E.g., one cache for file system and virtual memory
- Good performance

Cons:
- Interactions are complex, leads to bugs
- No isolation within
Microkernels

• Runs OS services as ordinary user programs
  • E.g., a server provides the file system
• kernel implements minimal mechanism to run services in user space
  • Processes with memory
  • Interprocess communication
• Kernel interface != system call interface
• Pro: More isolation, more fault tolerance
• Con: Hard to get good performance, complexity
Xv6 case study

• Monolithic kernel
  • UNIX system calls are the kernel interface

• Source code reflects OS organization
  • user/    apps in user mode
  • kernel/  kernel’s implementation

• Kernel has several parts
  • kernel/defs.h, kernel/proc.c, kernel/fs.c, etc.

Goal: simple, easily readable/understandable
Building xv6 (e.g., make)

```
.c  gcc  .o
.c  gcc  .o
       ld
       a.out
```
Risc-V’s (emulated) computer

• A very simple board, e.g., no display
• Risc-V processor with N cores
• RAM (128 MB)
• Supports interrupts (PLIC, CLINT)
• Supports UART (serial port)
  • Xv6 uses this to provide a console (out)
  • Xv6 uses this to get keyboard input (in)
• Supports E1000 network card (over PCIe)
Why develop with qemu?

• More convenient than using real hardware
• Qemu emulates several types of computers
  • we use the "virt" one [https://github.com/riscv/riscv-qemu/wiki](https://github.com/riscv/riscv-qemu/wiki)
  • close to the SiFive board (https://www.sifive.com/boards) but with virtio for disk
CPU emulation

• What is it to “emulate”?
• qemu is a C program that faithfully implements a RISC-V processor

```c
for (;;) {
    read next instructions
    decode instruction
    execute instruction (updating processor state)
}
```