```
6.S081 2020 Lecture 3: OS organization

Lecture Topic:
  OS design
    system calls
    micro/monolithic kernel
  First system call in xv6

OS picture
  apps: sh, echo, ...
  system call interface (open, close,...)
        OS

Goal of OS
  run multiple applications
  isolate them
  multiplex them
  share

Strawman design: No OS
  Application directly interacts with hardware
    CPU cores & registers
    DRAM chips
    Disk blocks
    ...
  OS library perhaps abstracts some of it

Strawman design not conducive to multiplexing
  each app periodically must give up hardware
  BUT, weak isolation
    app forgets to give up, no other app runs
    apps has end-less loop, no other app runs
    you cannot even kill the badly app from another app
  but used by real-time OSes
    "cooperative scheduling"

Strawman design not conducive to memory isolation
  all apps share physical memory
  one app can overwrites another apps memory
  one app can overwrite OS library

Unix interface conducive to OS goals
  abstracts the hardware in way that achieves goals
  processes (instead of cores): fork
     OS transparently allocates cores to processes
       Saves and restore registers
     Enforces that processes give them up
       Periodically re-allocates cores
  memory (instead of physical memory): exec
     Each process has its "own" memory
     OS can decide where to place app in memory
     OS can enforce isolation between memory of different apps
     OS allows storing image in file system
  files (instead of disk blocks)
     OS can provide convenient names
     OS can allow sharing of files between processes/users
  pipes (instead of shared physical mem)
     OS can stop sender/receiver

OS must be defensive
  an application shouldn't be able to crash OS
  an application shouldn't be able to break out of its isolation
  => need strong isolation between apps and OS
  approach: hardware support
```

```
    - user/kernel mode
    - virtual memory

  Processors provide user/kernel mode
    kernel mode: can execute "privileged" instructions
      e.g., setting kernel/user bit
      e.g., reprogramming timer chip
    user mode: cannot execute privileged instructions
    Run OS in kernel mode, applications in user mode
    [RISC-V has also an M mode, which we mostly ignore]

  Processors provide virtual memory
    Hardware provides page tables that translate virtual address to physical
    Define what physical memory an application can access
    OS sets up page tables so that each application can access only its memory

  Apps must be able to communicate with kernel
    Write to storage device, which is shared => must be protected => in kernel
    Exit app
    ...

  Solution: add instruction to change mode in controlled way
    ecall <n>
    enters kernel mode at a pre-agreed entry point

  Modify OS picture
    user / kernel (redline)
    app -> printf() -> write() -> SYSTEM CALL -> sys_write() -> ...
      user-level libraries are app's private business
    kernel internal functions are not callable by user

    other way of drawing picture:
    syscall 1  -> system call stub -> kernel entry -> syscall -> fs
    syscall 2                                                 -> proc

    system call stub executes special instruction to enter kernel
      hardware switches to kernel mode
      but only at an entry point specified by the kernel

    syscall need some way to get at arguments of syscall

    [syscalls the topic of this week's lab]

  Kernel is the Trusted Computing Base (TCB)
    Kernel must be "correct"
      Bugs in kernel could allow user apps to circumvent kernel/user
        Happens often in practice, because kernels are complex
        See CVEs
    Kernel must treat user apps as suspect
      User app may trick kernel to do the wrong thing
      Kernel must check arguments carefully
      Setup user/kernel correctly
      Etc.
    Kernel in charge of separating applications too
      One app may try to read/write another app's memory
    => Requires a security mindset
      Any bug in kernel may be a security exploit

  Aside: can one have process isolation WITHOUT h/w-supported
    kernel/user mode and virtual memory?
    yes! use a strongly-typed programming language
    - For example, see Singularity O/S
    the compiler is then the trust computing base (TCB)
    but h/w user/kernel mode is the most popular plan
```

```
Monolothic kernel
  OS runs in kernel space
  Xv6 does this.  Linux etc. too.
  kernel interface == system call interface
  one big program with file system, drivers, &c
  - good: easy for subsystems to cooperate
    one cache shared by file system and virtual memory
  - bad: interactions are complex
    leads to bugs
    no isolation within

Microkernel design
  many OS services run as ordinary user programs
    file system in a file server
  kernel implements minimal mechanism to run services in user space
    processes with memory
    inter-process communication (IPC)
  kernel interface != system call interface
  - good: more isolation
  - bad: may be hard to get good performance
  both monolithic and microkernel designs widely used

Xv6 case study
  Monolithic kernel
    Unix system calls == kernel interface
  Source code reflects OS organization (by convention)
    user/    apps in user mode
    kernel/  code in kernel mode
  Kernel has several parts
    kernel/defs.h
       proc
       fs
       ..
  Goal: read source code and understand it (without consulting book)

Using xv6
  Makefile builds
    kernel program
    user programs
    mkfs
  $ make qemu
    runs xv6 on qemu
    emulates a RISC-V computer

Building kernel
  .c -> gcc -> .s -> .o  \
  ....                    ld -> a.out
  .c -> gcc -> .s -> .o  /
  makefile keeps .asm file around for binary
  see for example, kernel/kernel.asm

The RISC-V computer
  A very simple board (e.g., no display)
  - RISC-V processor with 4 cores
  - RAM (128 MB)
  - support for interrupts (PLIC, CLINT)
  - support for UART
    allows xv6 to talk to console
    allows xv6 to read from keyboard
  - support for e1000 network card (through PCIe)

Development using Qemu
  More convenient than using the real hardware
  Qemu emulates several RISC-V computers
  - we use the "virt" one
```

```
    https://github.com/riscv/riscv-qemu/wiki
  - close to the SiFive board (https://www.sifive.com/boards)
    but with virtio for disk

What is "to emulate"?
  Qemu is a C program that faithfully implements a RISC-V processor
  for (;;) {
    read next instructions
    decode instruction
    execute instruction (updating processor state)
  }
  [big idea: software = hardware]

Boot xv6 (under gdb)
  $ make CPUS=1 qemu-gdb
    runs xv6 under gdb (with 1 core)
  Qemu starts xv6 in kernel/entry.S (see kernel/kernel.ld)
    set breakpoint at _entry
      look at instruction
      info reg
    set breakpoint at main
      Walk through main
    single step into userinit
      Walk through userinit
      show proc.h
      show allocproc()
      show initcode.S/initcode.asm
    break forkret()
      walk to userret
    break syscall
      print num
      syscalls[num]
      exec "/init"
```