Intro & Arch. Support for OS

CS 202: Advanced Operating Systems

Questions to get you thinking

- Is OS always running?
  - If not, how do we make sure it gets to run?
- How do we prevent user programs from directly manipulating hardware?
  - If you run a restaurant, would you want customers to go into your kitchen and mess up?

Basic Idea

- OS is a *sleeping beauty*
  - Technically known as *limited direct execution*
- User programs (when scheduled) run *directly* on the CPU
- But OS adds *limits* on running programs
  - OS runs in response to "events"
  - Only OS can manipulate hardware or critical system state
    - Some operations are only valid in privileged mode
  - Without limits on running programs, the OS wouldn't be in control of anything and thus would be "just a library"
- Most of the time the OS is *sleeping*
  - Good! Less overhead
  - Good! Applications are running directly on the hardware

What do we need from the architecture/CPU?

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Other stuff
  - Mechanisms to handle concurrency, Isolation, virtualization …

Privileged Instructions

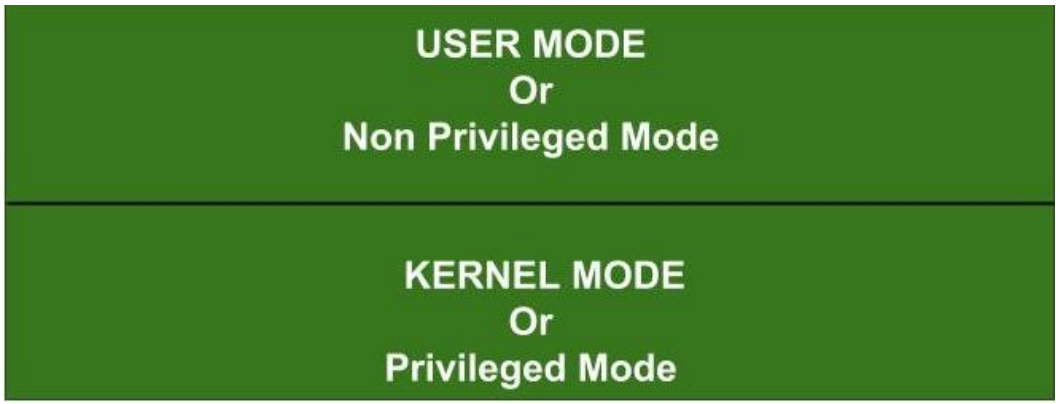
Also called "protected instructions"

- OS must have exclusive access to hardware and critical data structures

- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction

Privilege mode

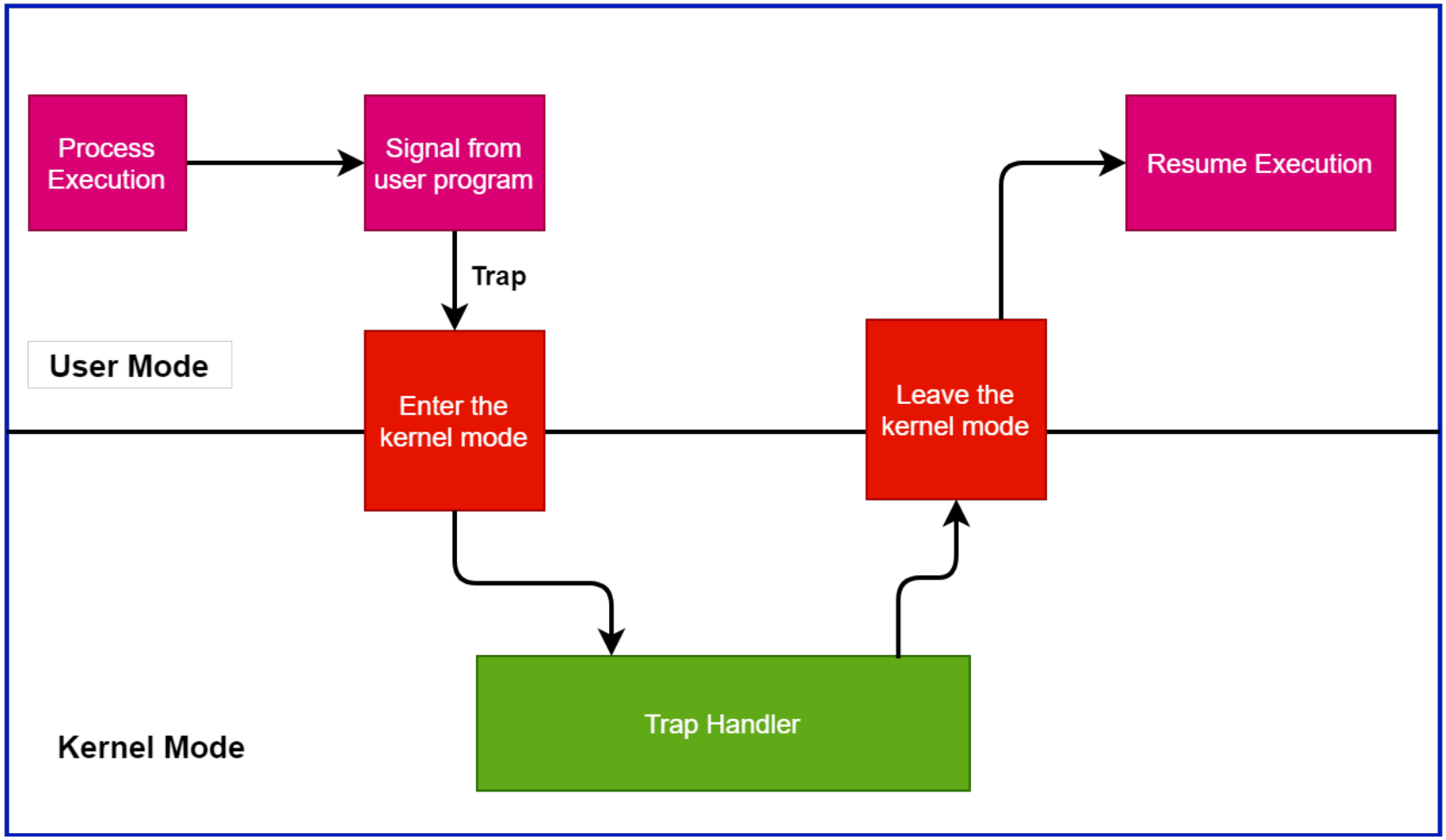
- HW support (at least) two execution modes:



Hardware restricts privileged instructions to OS

- Q: How does the HW know if the executed program is OS?
- "Mode" kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode trap to OS

Privilege mode



Privilege mode

- CPU mode bit
- Mode 0 = untrusted – user domain
  - Privileged instructed and registers are disabled by CPU
- Mode 1 = trusted = kernel domain
  - Enabling all instructions and registers

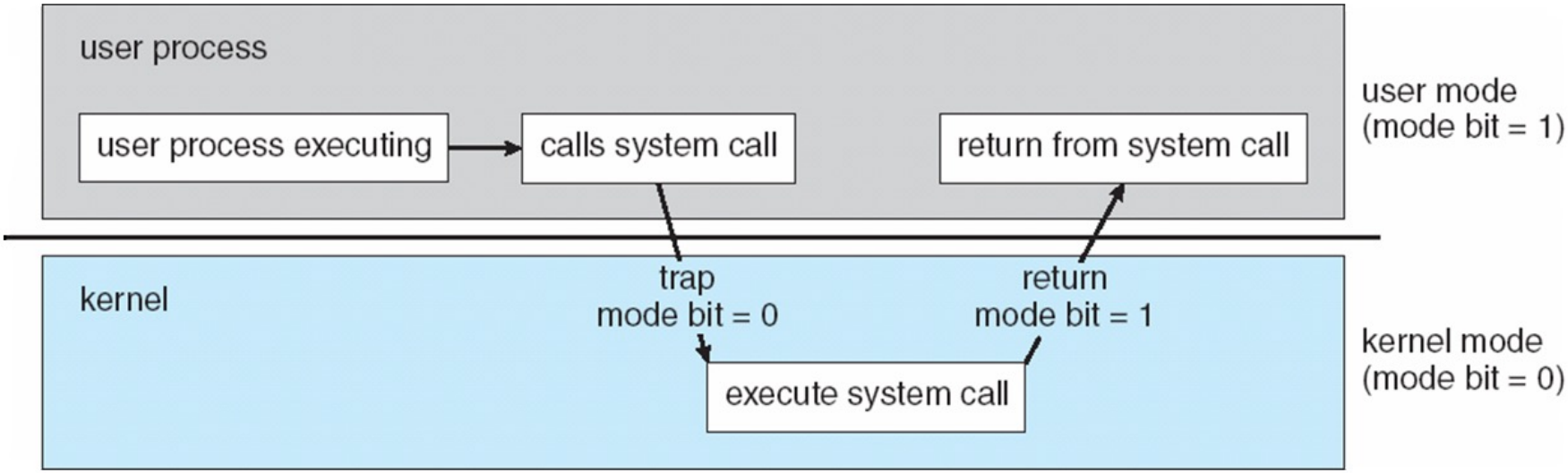
Privilege mode

- Boot sequence
  - Load first sector of disk which contains OS code to predetermined address in memory
  - Mode set to 1; PC set to predetermined address
- OS takes over
  - Initialize devices, MMU, timers, etc.
  - Load programs from disk, set up pagetables, etc.
  - Mode set of 0; PC set to program entry point

Mode Switching

- External Interrupt
  - Timer, I/O interrupts
- Internal Trap
  - Page fault, Invalid operation (division by 0), ...
- System Call
  - I/O operations (file open), fork, …

mode switching



Switching back and forth

- Going from higher privilege to lower privilege
  - Kernel to user
  - Easy: can directly modify the mode register to drop privilege
- But how do we escalate privilege?
  - Special instructions to change mode
    - System calls (`int 0x80, syscall, svc`)
    - Saves context and invokes designated handler
      - You jump to the privileged code; you cannot execute your own
    - OS checks your syscall request and honors it *only if* safe
  - Interrupts, traps

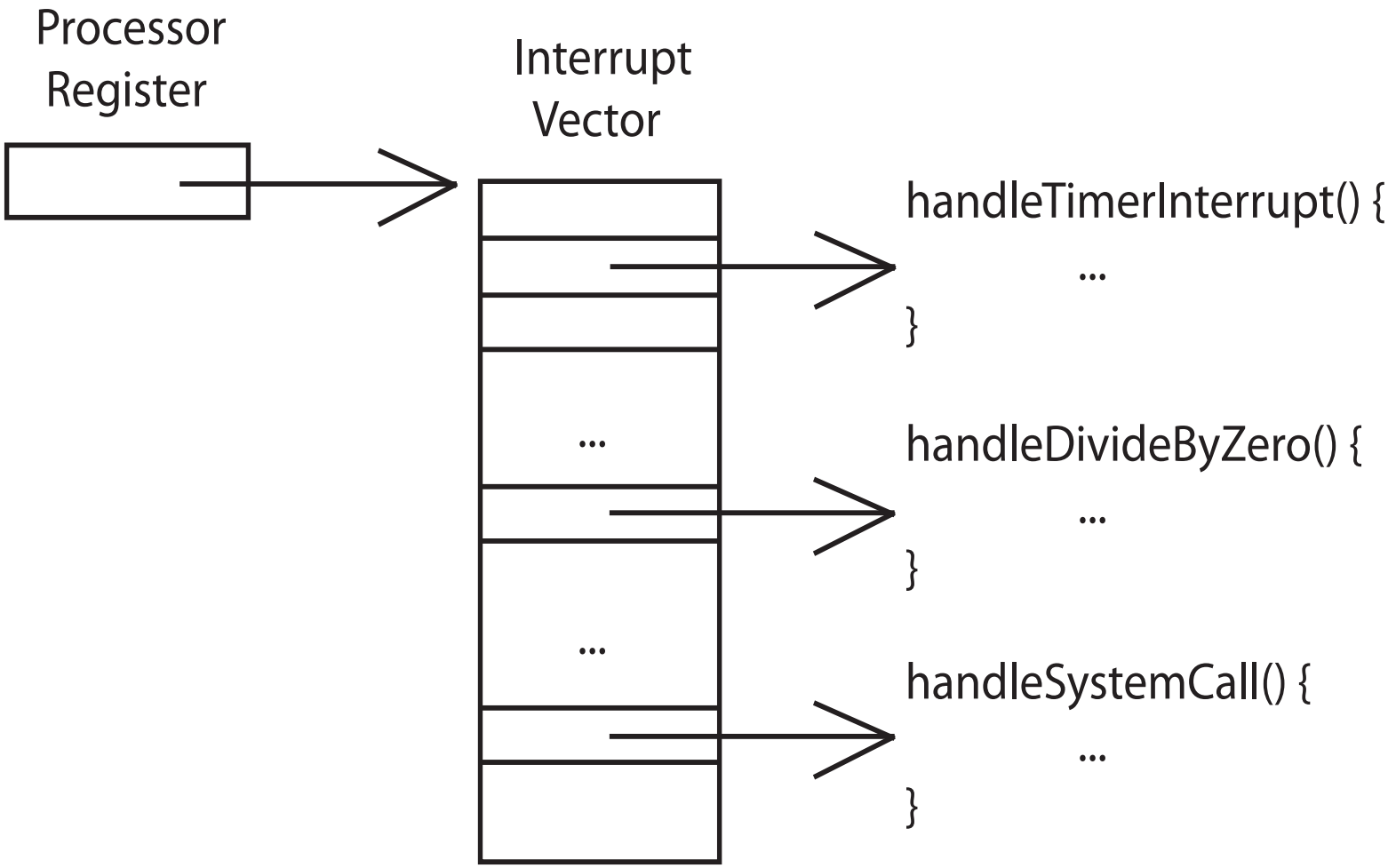
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Other stuff
  - Mechanisms to handle concurrency, Isolation, virtualization …

Events

- An event is an exceptional control flow
  - Events immediately stop current execution (think about Ctrl+C)
- The kernel defines a handler for each event type
  - Such event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- Once the system is booted, OS is one big event handler
  - All entry to the kernel occurs as the result of an event

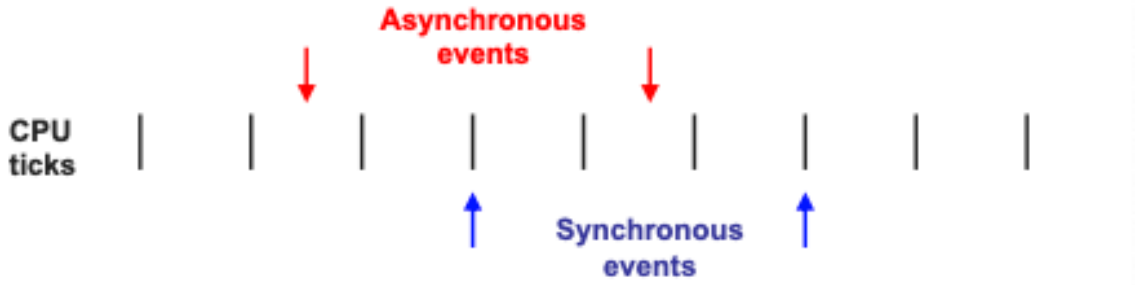
Handling events – Interrupt vector table



Categorizing Events

- This gives us a convenient table:

	Unintentional	Intentional
Synchronous	Fault	Syscall
Asynchronous	Interrupt	Signal



- Terms may be used slightly differently by various OSes, CPU architectures
  - e.g., exceptions include fault and software interrupt

Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault
  - Memory access violation (unaligned, permission, not mapped, bounds…)
  - Illegal instruction
  - Divide by zero

- Upon exception, hardware “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
  - Invokes registered fault handler

Handling Faults

- Some faults are handled by "fixing" the exceptional condition and returning to the faulting context

- Example: page faults in memory management
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault

Handling Faults

- The kernel may handle *unrecoverable* faults by killing the user process
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)

- What about unrecoverable faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows "Blue screen of death"
    - Kernel is halted, state dumped to a core file, machine locked up

Categorizing Events

	Unintentional	Intentional
Synchronous	Fault	**Syscall**
Asynchronous	Interrupt	Signal

Hardware generated

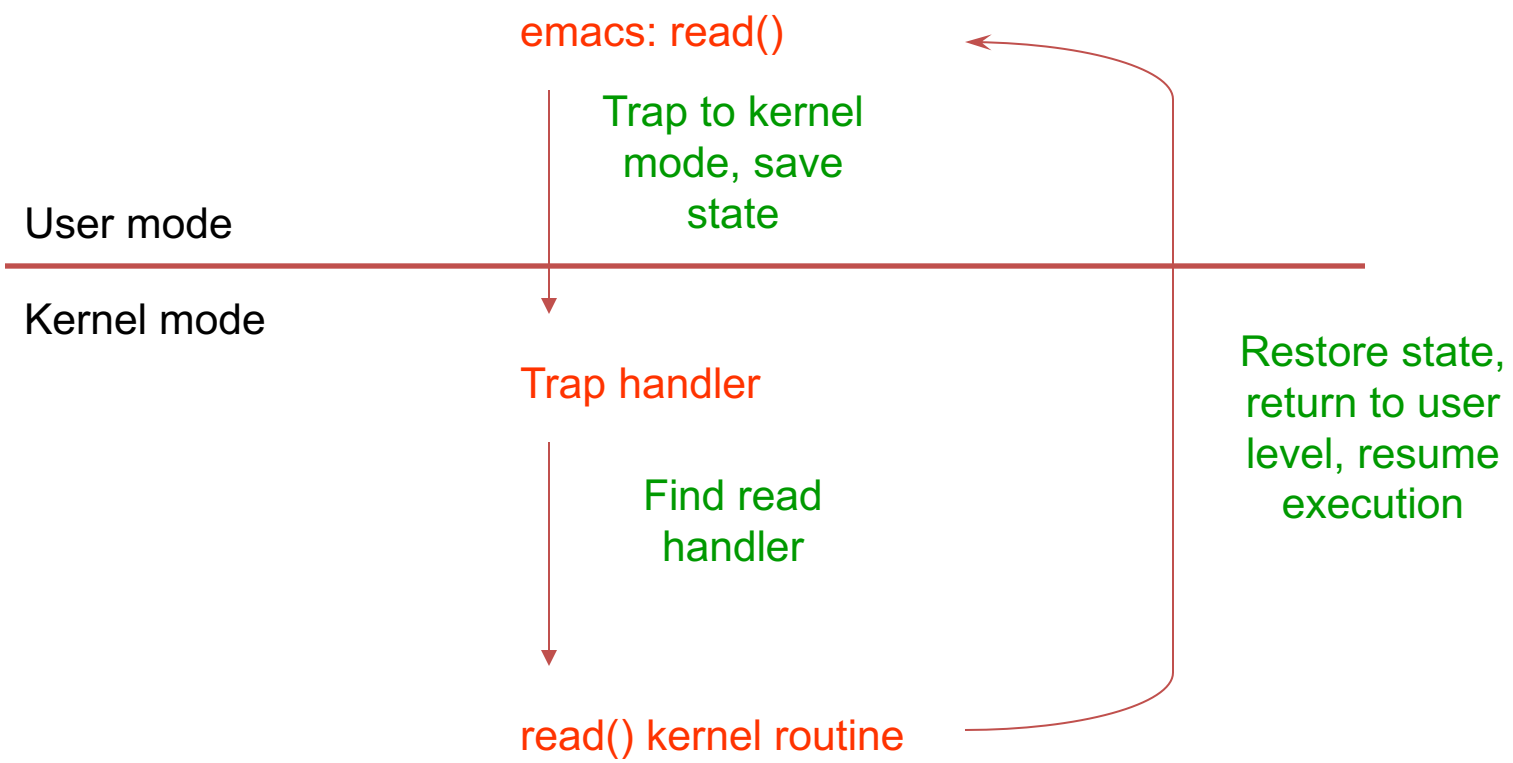
usually Software generated

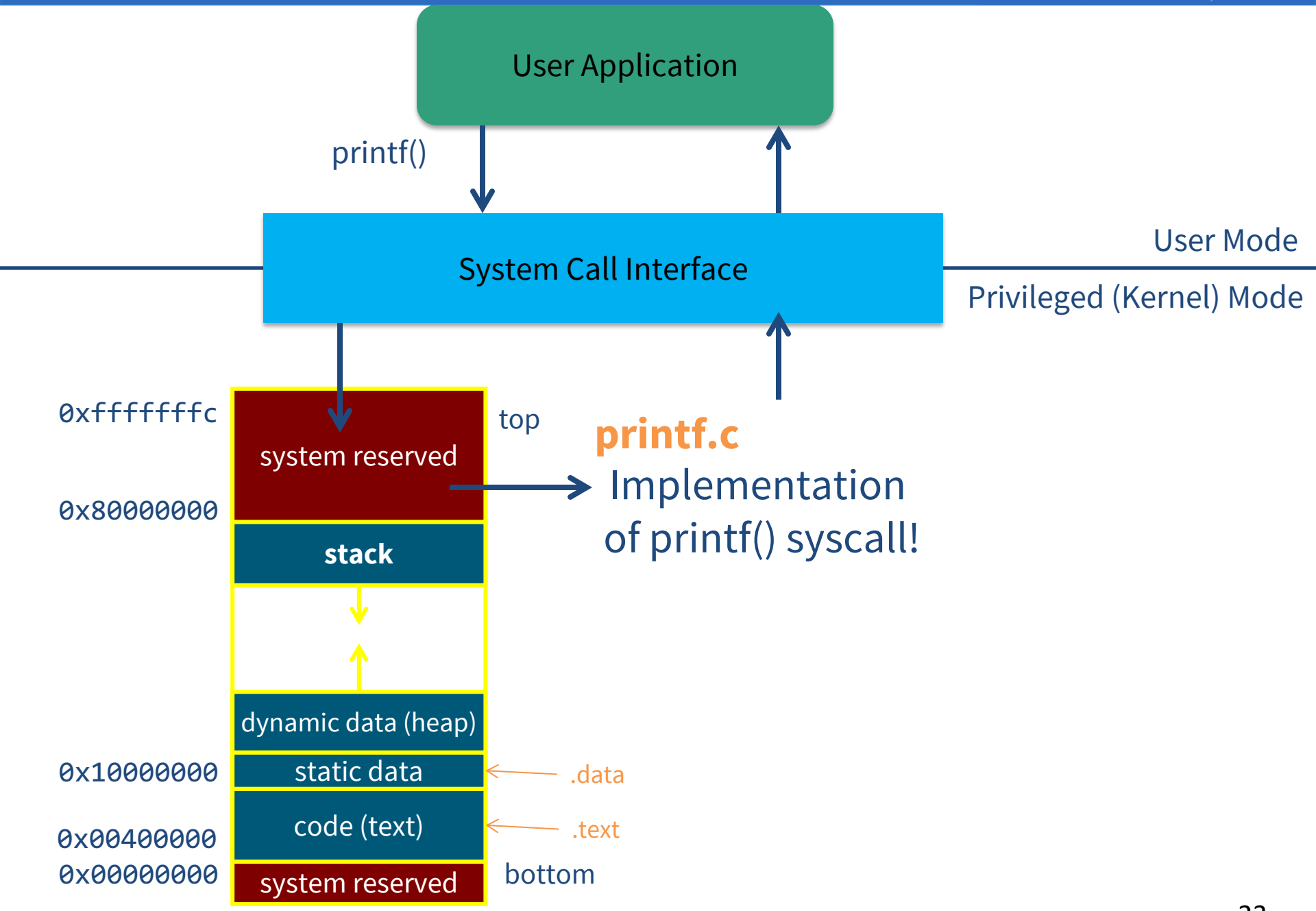
System Calls

- For a user program to do something "privileged" (e.g., I/O), it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call

- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
    - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
  - Returning from system call restores this state

System Call


emacs: read()
Trap to kernel mode, save state
User mode
Kernel mode
Trap handler
Find read handler
read() kernel routine
Restore state, return to user level, resume execution


User Application
printf()
System Call Interface
User Mode
Privileged (Kernel) Mode
0xffffffff
top
system reserved
printf.c
Implementation
of printf() syscall!
0x80000000
stack
dynamic data (heap)
0x10000000
static data
.data
code (text)
.text
0x00400000
0x00000000
system reserved
bottom

Sample system call

- putc(): print character to screen
- send(): send a packet on the network
- sbrk(): allocate a page
- sleep(): put current program to sleep, wake other

System Call Questions

- There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
  - User program cannot specify an exact address to jump to (why?)
  - OS maintains a system call table
  - Before issuing **int $0x80** (x86) or **syscall** (riscv), set **%eax/%rax** or **v0** with the syscall number

- System calls are like function calls, but how to pass parameters?
  - Just like calling convention in syscalls, typically passed through **%ebx, %ecx, %edx, %esi, %edi, %ebp** (x86) or **a0 through a7** (riscv)

Categorizing Events

	Unintentional	Intentional
Synchronous	Fault	Syscall
Asynchronous	**Interrupt**	**Signal**

Software

- Interrupts & signals: asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

Timer – special interrupt

- The key to a time-sharing OS
  - The fallback mechanism by which the OS reclaims control
- Timer is set to generate an interrupt after a period of time
  - Setting timer is a privileged instruction
  - When timer expires, generates an interrupt
    - Handled by the OS, forcing a switch from the user program
  - Basis for OS scheduler *(more later…)*
- Also used for time-based functions (e.g., *sleep()*)

I/O using Interrupts

- **Interrupts** are the basis for asynchronous I/O
  - CPU issues read command
  - I/O module gets data from peripheral whilst CPU does other work
  - I/O module interrupts CPU
  - CPU requests data
  - I/O module transfers data
  - CPU issues read command
- Why this way? → Higher Throughput

CPU doesn't wait

I/O Example

Receiving a network packet:

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

Summary

- Protection
  - User/kernel modes
  - Protected instructions
- Faults
  - Unexpected event during execution (e.g., divide by zero)
- System calls
  - Used by user-level processes to access OS functions
  - Access what is "in" the OS
- Interrupts
  - Timer, I/O

OS runs when any of these are triggered

Historic Evolution of OS

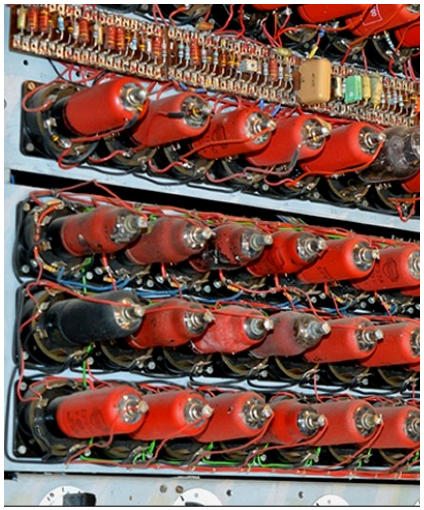
Outline

- Evolution of Operating Systems (and computers!)

- Why study history?
  - Understand why OS's look like they are
  - Appreciate how and why different pieces evolved
  - Explain how external forces also shape OS
  - Provide context for the rest of the quarter
  - It's interesting!

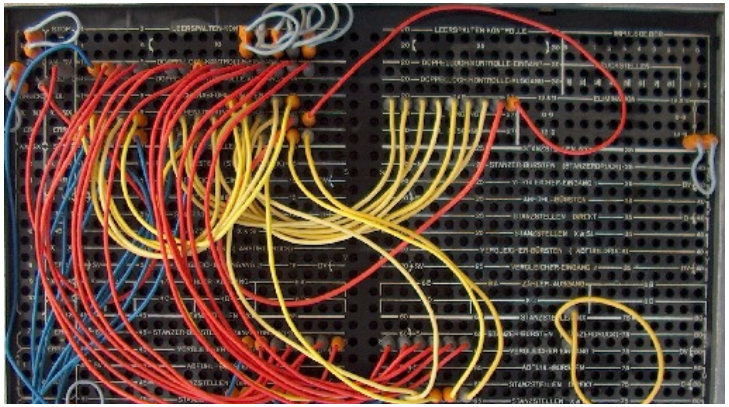
Dawn of computing

- Pre 1950 : very first electronic computers
  - valves and relays
  - single program with dedicated function
- Pre 1960 : stored program valve machines
  - single job at a time; OS is a program loader



Phase 0 of OS Evolution (40s to 1955)

- No OS
  - Computers are exotic, expensive, large, slow experimental equipment
  - Program in machine language and using plugboards
  - User sits at console: no overlap between computation, I/O, user thinking, etc..
    - Program manually by plugging wires in
    - Goal: number crunching for missile computations
- Imagine programming that way
  - Painful and slow



OS progress in this period

- Libraries of routines that are common
  - Including those to talk to I/O devices
  - Punch cards (enabling copying/exchange of these libraries) a big advance!
  - Pre-cursor to OS

Phase 1: 1955-1970

- Computers expensive; people cheap
  - Use computers efficiently – move people away from machine
  - OS becomes a batch monitor
    - Loads a job, runs it, then moves on to next
    - If a program fails, OS records memory contents somewhere
    - More efficient use of hardware but increasingly difficult to debug
- Thomas Watson
  - "I think there is a world market for maybe five computers", 1943
  - Often called "the world's greatest salesman" by the time of his death in 1956

Advances in technology in this stage

- Data channels and interrupts
  - Allow overlap of I/O and computing
  - Buffering and interrupt handling done by OS
  - Spool (buffer) jobs onto “high speed” drum memory

Phase 1: problems

- Utilization is low (one job at a time)
- No protection between jobs
- Short jobs wait behind long jobs
  - So, we can only run one job at a time
- Coordinating concurrent activities
- Still painful and slow (but less so?)

Advances in OS in this period

- Hardware provided memory support (protection and relocation)
- Multiprogramming (not to be confused with time sharing)
- Scheduling: let short jobs run first
- OS must manage interactions between concurrent things
  - Starts emerging as a field/science
- OS/360 from IBM first OS designed to run on a family of machines from small to large

Some important projects

- Atlas computer/OS from Manchester U. (late 50s/early 60s)
  - First recognizable OS
  - Separate address space for kernel
  - Early virtual memory

- THE Multiprogramming system (early 60s)
  - Introduced semaphores
  - Attempt at proving systems correct; interesting software engineering insights

Not all is smooth

- Operating systems didn't really work
- No software development or structuring tools; written in assembly
- OS/360 introduced in 1963 but did not really work until 1968
  - Reported on in mythical man month
- Extremely complicated systems
  - 5-7 years development time typical
  - Written in assembly, with no structured programming
  - Birth of software engineering?

Phase 2: 1970s

- Computers and people are expensive
  - Help people be more productive
  - Interactive time sharing: let many people use the same machine at the same time
  - Emergence of minicomputers
    - Terminals are cheap
  - Keep data online on fancy file systems
  - Attempt to provide reasonable response times (Avoid thrashing)

Important advances and systems

- Compatible Time-Sharing System (CTSS)
  - MIT project (demonstrated in 1961)
  - One of the first time-sharing systems
  - Corbato won Turing award in 1990
  - Pioneered much of the work in scheduling
  - Motivated MULTICS

MULTICS

- Jointly developed by MIT, Bell Labs and GE
- Envisioned one main computer to support everyone
  - People use computing like a utility like electricity – sound familiar? Ideas get recycled
- Many many fundamental ideas: protection rings, hierarchical file systems, devices as files, …
- Building it was more difficult than expected
- Technology caught up

Unix appears

- Ken Thompson, who worked on MULTICS, wanted to use an old PDP-7 laying around in Bell labs
- He and Dennis Richie built a system designed by programmers for programmers
- Originally in assembly. Rewritten in C
  - If you notice for the paper, they are defending this decision
  - However, this is a new and important advance: portable operating systems!
- Shared code with everyone (particularly universities)

Unix (cont'd)

- Berkeley added support for virtual memory for the VAX
- DARPA selected Unix as its networking platform in arpanet
- Unix became commercial
  - …which eventually lead Linus Torvald to develop Linux

Some important ideas in Unix

- OS written in a high level language
- OS portable across hardware platforms
  - Computing is no longer a pipe stove/vertical system
- Pipes
  - combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on
- Mountable file systems
- Many more (we'll talk about unix later)
- 1983 Turing Award

Ken Thompson

Dennis M. Ritchie

Phase 3: 1980s

- Computers are cheap, people expensive
  - Put a computer in each terminal
  - CP/M from DEC first personal computer OS (for 8080/85) processors
  - IBM needed software for their PCs, but CP/M was behind schedule
  - Approached Bill Gates to see if he can build one
  - Gates approached Seattle computer products, bought 86-DOS and created MS-DOS
  - Goal: finish quickly and run existing CP/M software
  - OS becomes subroutine library and command executive

New technologies in Phase 3

- Personal workstations
  - The PERQ
  - Xerox Alto
  - SUN workstation
- Personal computers
  - Apple II
  - IBM PC
  - Macintosh

New technologies (cont'd)

- Business applications!
  - Word processors
  - Spreadsheets
  - Databases
- Marketplace is broken up horizontally
  - Hardware
  - OS
  - Applications

New advances in OS

- PC OS was a regression for OS
  - Stepped back to primitive phase 1 style OS leaving the cool developments that occurred in phase 2

- Academia was still active, and some developments still occurred in mainframe and workstation space

Phase 4: Networked systems 1990s to 2010s

- Machines can talk to each other
  - its all about connectivity
- We want to share *data* not hardware
- Networked applications drive everything
  - Web, email, messaging, social networks, …
- Protection and multiprogramming less important for personal machines
  - But more important for servers

Phase 4, continued

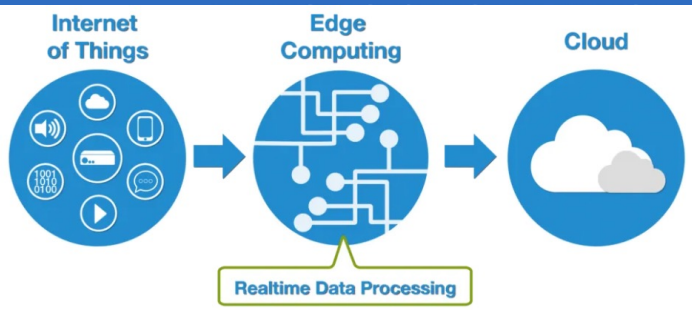
- Market place continued horizontal stratification
  - ISPs (service between OS and applications)
  - Information is a commodity
  - Advertising a new marketplace

- New network-based architectures
  - Client server
  - Clusters
  - Grids
  - Distributed operating systems
  - Cloud computing (or is that phase 5?)

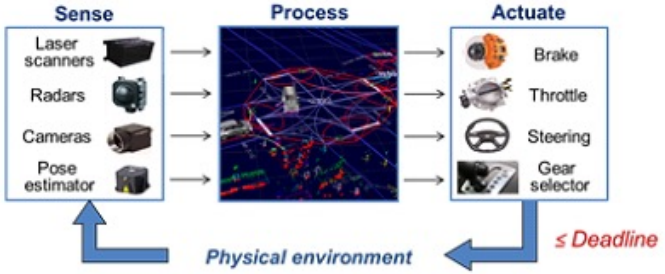
New problems

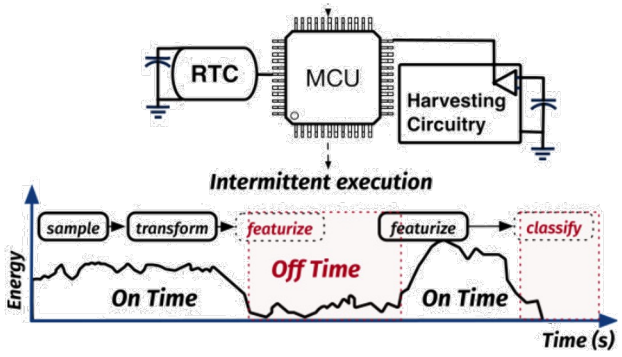
- Large scale
  - Google file system, map-reduce, …
- Concurrency at large scale
  - ACID (Atomicity, Consistency, Isolation and Durability) in Internet Scale systems
    - Very large delays
    - Partitioning
- Security and Privacy
- Tail Latency and Edge Computing

Phase 5: 2010s -- ??



- New generation?
- Mobile devices that are powerful
- Cyber-physical systems & IoT
  - Sensing: location, motion, …
  - Self-driving cars, UAVs, ...
- Intermittently-powered devices
- Computing evolving beyond networked systems
  - But OS for them looks largely the same
  - Is that a good idea?





Archaeology

- Multics -> AT&T Unix -> BSD Unix -> Ultrix, SunOS, NetBSD, …
- Mach (micro-kernel) + BSD -> NextStep -> XNU -> Apple OSX, iOS
- CP/M -> QDOS -> MS-DOS -> Windows 3.1 -> NT -> 95 -> 98 -> 2000 -> XP -> Vista -> 7 -> 8 -> phone -> 10 -> 11
- Linux -> Android
- Linux -> RedHat, Ubuntu, Fedora, Debian, …

xv6 Overview

CS 202: Advanced Operating Systems

xv6

- xv6 is MIT's re-implementation of Unix v6
  - Written in ANSI C
  - Runs on RISC-V and x86
    - We will use the RISC-V version with the QEMU simulator
  - Smaller than v6
  - Preserve basic structure (processes, files, pipes. etc.)
  - Runs on multicores
  - Got paging support in 2011

Ken Thompson & Dennis Ritchie, 1975

xv6

- To understand it, you'll need to read its source code
  - It's not that hard
  - Source code:
    - https://github.com/rtenlab/xv6-riscv (for our projects)
      - Forked from https://github.com/mit-pdos/xv6-riscv; to avoid unexpected updates during this course
  - Book/commentary
    - xv6: a simple, Unix-like teaching operating system
    - https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

Why xv6?

- Why study an old OS instead of Linux, Solaris, or Windows?

1. Big enough
   - To illustrate basic OS design & implementation
2. Small enough
   - To be (relatively) easily understandable
3. Similar enough
   - To modern OSes
   - Once you've explored xv6, you will find your way inside kernels such as Linux

Why RISC-V?

RISC-V: The Free and Open RISC Instruction Set Architecture

- RISC-V: open standard instruction set architecture (ISA) based on RISC principles
  - High quality, loyalty free, license free
  - Multiple proprietary and open-source core implementations
  - Supported by growing software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
- Fun to use toolchains for the new architecture

Apple shows interest in RISC-V chips, a competitor to iPhones' Arm tech

RISC-V chip technology could be used for tasks like AI and computer vision.

Intel Will Offer SiFive RISC-V CPUs on 7nm, Plans Own Dev Platform

By Joel Hruska on June 24, 2021 at 8:36 am | Comments

xv6 Structure

- Monolithic kernel
  - Provides services to running programs
- Processes uses system calls to access system services
- When a process call a system call
  - Execution will enter the kernel space
  - Perform the service
  - Return to the user space

xv6 System Calls

System call	Description
`fork()`	Create process
`exit()`	Terminate current process
`wait()`	Wait for a child process to exit
`kill(pid)`	Terminate process pid
`getpid()`	Return current process's id
`sleep(n)`	Sleep for n seconds
`exec(filename, *argv)`	Load a file and execute it
`sbrk(n)`	Grow process's memory by n bytes
`open(filename, flags)`	Open a file; flags indicate read-/write

xv6 System Calls (2)

System call	Description
`read(fd, buf, n)`	Read n bytes from an open file into buf
`write(fd, buf, n)`	Write n bytes to an open file
`close(fd)`	Release open file fd
`dup(fd)`	Duplicate fd
`pipe(p)`	Create a pipe and return fd's in p
`chdir(dirname)`	Change the current directory
`mkdir(dirname)`	Create a new directory
`mknod(name, major, minor)`	Create a device file
`fstat(fd)`	Return info about an open file
`link(f1, f2)`	Create another name (f2) for the file f1
`unlink(filename)`	Remove a file

xv6 kernel source files

- /kernel directory

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.

Setup

- Toolchain
  - You need a RISC-V tool chain and QEMU for RISC-V
- Linux (Ubuntu 20.04)

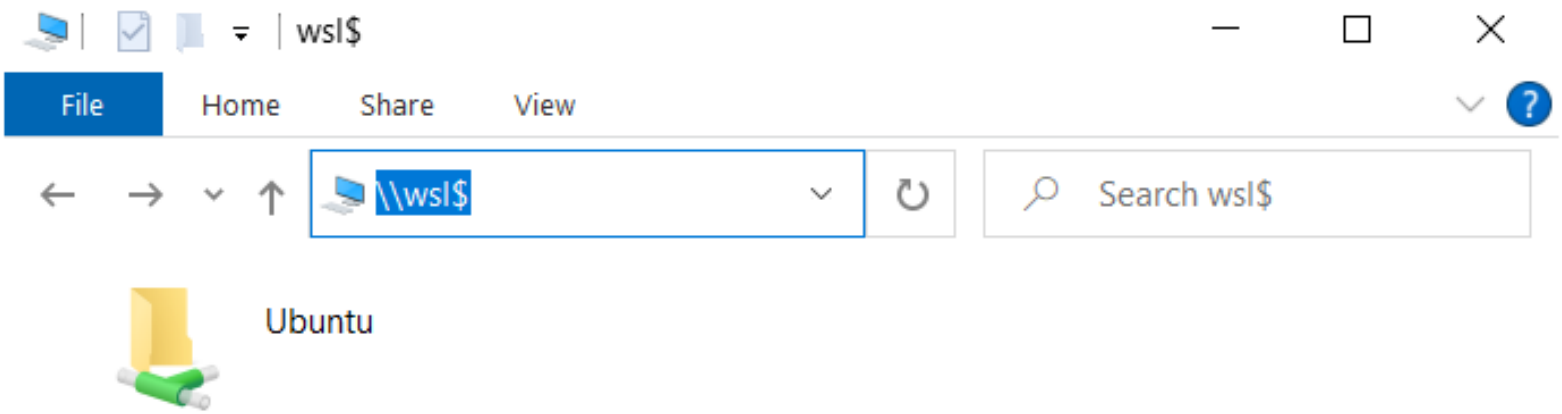
```
$ sudo apt update
$ sudo apt install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

- Windows
  - You can use Windows Subsystem for Linux (WSL) with Ubuntu 20.04
  - Unsure what version of Ubuntu you have? Open WSL terminal and type "lsb_release -a"
  - Follow the above Linux instruction for package installation

```
~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04 LTS
Release:        20.04
Codename:       focal
```

Setup

- Windows (cont')
  - All your WSL Linux files are accessible as \\wsl$ in File Explorer
    - Exposed as network shared files
  - Your home directory is \\wsl$\home\<username>



Setup

- macOS
  - Install developer tools:

```
$ xcode-select --install
```

  - Install Homebrew (package manager)

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

  - Install the RISC-V compiler toolchain:

```
$ brew tap riscv/riscv
$ brew install riscv-tools
```

  - Update path; open ~/.bashrc and add the following line

```
PATH=$PATH:/usr/local/opt/riscv-gnu-toolchain/bin
```

  - Install QEMU

```
$ brew install qemu
```

Setup

- **Download xv6:**

  $ git clone https://github.com/rtenlab/xv6-riscv

  $ cd xv6-riscv

- **Compile and run xv6:**

  $ make qemu

  (“make” to compile only)

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

(xv6 shell)

What does “hart” mean?

- In RISC-V, hart refers to a *hardware thread*
- xv6 boots on hard 0 and turns on other harts

- **Exit from QEMU**
  - Press Ctrl+a and then press c to get the QEMU console
  - Then type “quit” to exit

Create a System Call

- Goal: create a system call "sys_hello" that call a kernel function that displays: "Hello from the kernel space!"

- To do that, open the following files and add the lines with "// hello" comment:

Create a System Call (2)

- kernel/syscall.h: define new syscall number

```
16  #define SYS_open   15
17  #define SYS_write  16
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_hello  22 // hello
```

Create a System Call (3)

- kernel/syscall.c: update system call table

```
105  extern uint64 sys_write(void);
106  extern uint64 sys_uptime(void);
107  extern uint64 sys_hello(void); // hello: declaration
108
109  static uint64 (*syscalls[])(void) = {
110  [SYS_fork]    sys_fork,
111  [SYS_exit]    sys_exit,
112  [SYS_wait]    sys_wait,
113  [SYS_pipe]    sys_pipe,
114  [SYS_read]    sys_read,
115  [SYS_kill]    sys_kill,
116  [SYS_exec]    sys_exec,
117  [SYS_fstat]   sys_fstat,
118  [SYS_chdir]   sys_chdir,
119  [SYS_dup]     sys_dup,
120  [SYS_getpid]  sys_getpid,
121  [SYS_sbrk]    sys_sbrk,
122  [SYS_sleep]   sys_sleep,
123  [SYS_uptime]  sys_uptime,
124  [SYS_open]    sys_open,
125  [SYS_write]   sys_write,
126  [SYS_mknod]   sys_mknod,
127  [SYS_unlink]  sys_unlink,
128  [SYS_link]    sys_link,
129  [SYS_mkdir]   sys_mkdir,
130  [SYS_close]   sys_close,
131  [SYS_hello]   sys_hello, // hello: syscall entry
132  };
133
```

Create a System Call (4)

- kernel/sysproc.c: define syscall function

```
93  uint64 sys_hello(void) // hello syscall definition
94  {
95    int n;
96    argint(0, &n);
97    print_hello(n);
98    return 0;
99  }
```

- kernel/proc.c: new kernel function

```
685 // hello: printing hello msg
686 void print_hello(int n)
687 {
688   printf("Hello from the kernel space %d\n", n);
689 }
```

Create a System Call (5)

- kernel/defs.h

```
84  // proc.c
85  int             cpuid(void);
86  void            exit(int);
87  int             fork(void);
88  int             growproc(int);
89  void            proc_mapstacks(pagetable_t);
90  pagetable_t     proc_pagetable(struct proc *);
91  void            proc_freepagetable(pagetable_t, uint64);
92  int             kill(int);
93  int             killed(struct proc*);
94  void            setkilled(struct proc*);
95  struct cpu*     mycpu(void);
96  struct cpu*     getmycpu(void);
97  struct proc*    myproc();
98  void            procinit(void);
99  void            scheduler(void) __attribute__((noreturn));
100 void            sched(void);
101 void            sleep(void*, struct spinlock*);
102 void            userinit(void);
103 int             wait(uint64);
104 void            wakeup(void*);
105 void            yield(void);
106 int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
107 int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
108 void            procdump(void);
109 void            print_hello(int); // hello
```

Create a System Call (6)

- Update user-space syscall interface
- user/usys.pl

```
36  entry("sbrk");
37  entry("sleep");
38  entry("uptime");
39  # hello syscall for user
40  entry("hello");
41
```

- user/user.h

```
4   // system calls
5   int fork(void);
6   int exit(int) __attribute__((noreturn));
7   int wait(int*);
8   int pipe(int*);
9   int write(int, const void*, int);
10  int read(int, void*, int);
11  int close(int);
12  int kill(int);
13  int exec(char*, char**);
14  int open(const char*, int);
15  int mknod(const char*, short, short);
16  int unlink(const char*);
17  int fstat(int fd, struct stat*);
18  int link(const char*, const char*);
19  int mkdir(const char*);
20  int chdir(const char*);
21  int dup(int);
22  int getpid(void);
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int hello(int); // hello
```

Test a System Call

1. Write a user program: Create " test.c " file in the user directory of "xv6-riscv" (user/test.c)

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
  int n = 0;
  if (argc >= 2) n = atoi(argv[1]);

  printf("Say hello to kernel %d\n", n);
  hello(n);
  exit(0);
}
```

Test a System Call (2)

2. Edit “ Makefile ” and append “$U/_test\” to UPROGS

```
118 UPROGS=\
119     $U/_cat\
120     $U/_echo\
121     $U/_forktest\
122     $U/_grep\
123     $U/_init\
124     $U/_kill\
125     $U/_ln\
126     $U/_ls\
127     $U/_mkdir\
128     $U/_rm\
129     $U/_sh\
130     $U/_stressfs\
131     $U/_usertests\
132     $U/_grind\
133     $U/_wc\
134     $U/_zombie\
135     $U/_test\
```

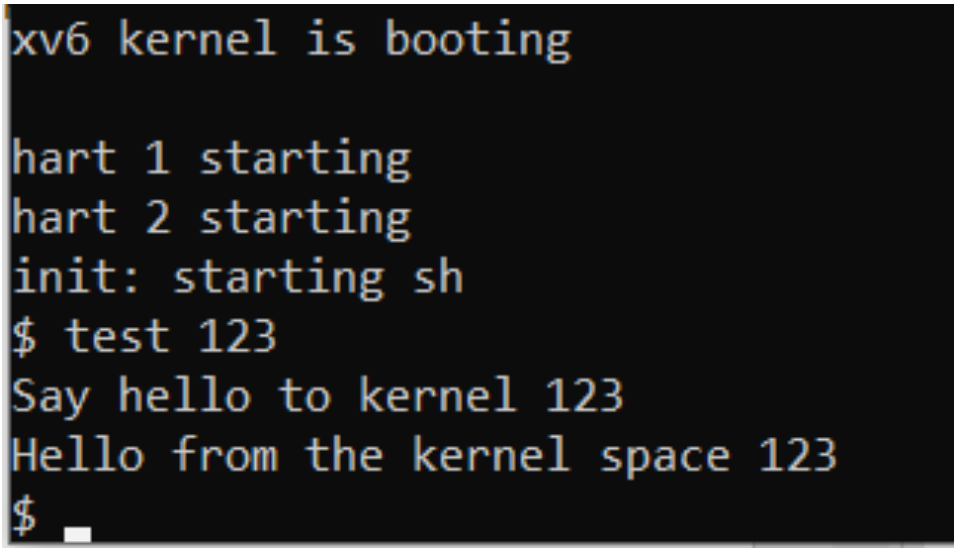
Test a System Call (3)

3. Type:

   $ make qemu

4. After xv6 boots, type:

   $ test



How to use GDB

- To run Qemu with GDB, you need to open another terminal at the same xv6-riscv folder.
- In the first terminal, type:

  $ make qemu-gdb

- In the second second terminal, type:

  $ gdb-multiarch -q -iex "set auto-load safe-path . "

```
~/xv6-riscv$ gdb-multiarch -q -iex "set auto-load safe-path . "
The target architecture is assumed to be riscv:rv64
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb) continue
```

Use “ break <address> ” to set a breakpoint
Type “ continue ” to run until breakpoint

MacOS: If gdb-multiarch doesn’t exist, try "riscv64-unknown-elf-gdb"

Change # of CPUs

- By default, xv6 is compiled for three CPUs. To change the number of CPUs, edit Makefile:

```
156  ifndef CPUS
157  # CPUS := 3
158  CPUS := 1
159  endif
```

- Unless otherwise mentioned, we will use a single-core system, so change to " CPUS := 1 "

```
xv6 kernel is booting

init: starting sh
$
```

No other harts (cores) after this change