CS 3650 Computer Systems – Summer 2025

OS Kernels, Booting, xv6 (1)

Week 10

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Ji-Yong Shin and Alden Jackson's lecture slides for the same course.

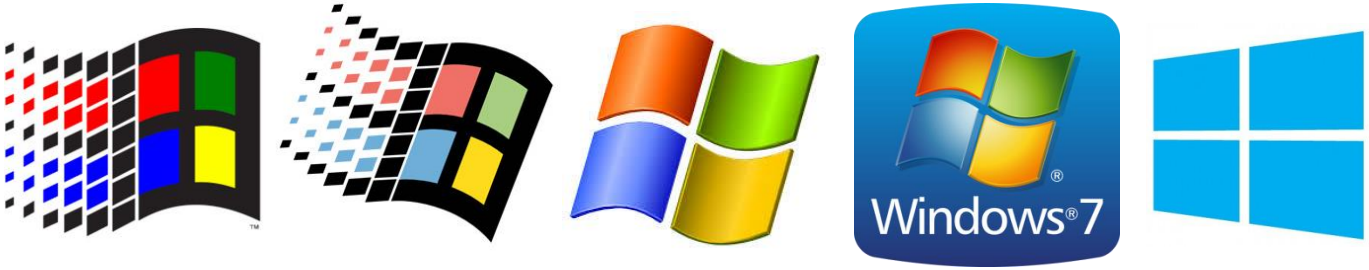
What is an Operating System?

- OS is software that sits between user programs and hardware

- OS provides interfaces to computer hardware
  - User programs do not have to worry about details
- OS is a resource manager and control program
  - Controls execution of user programs
  - Decides between conflicting requests for hardware access
  - Attempts to be efficient and fair
  - Prevents errors and improper use

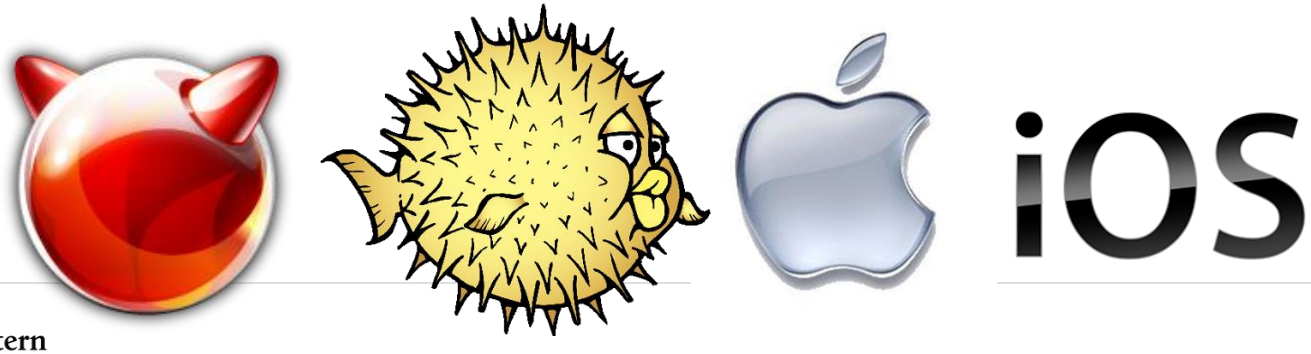
Many Different OSes

Windows



Linux

BSD



Without an operating system

- Life would be hard for us as software engineers having to always directly interface with hardware, and vice versa
- (Typically our computers, would be no better than a box with blinking lights)

Be grateful this isn't your IDE!

Operating System History

Brief Operating System History [link]

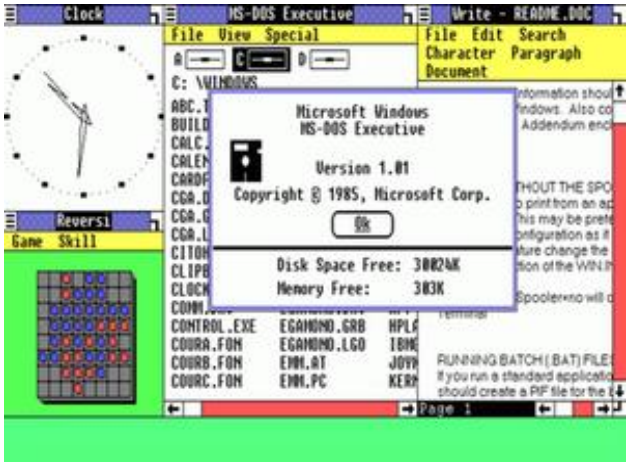
- 1955 and earlier: Very early mainframes have no operating system
- 1956: GM-NAA I/O used for research by General Motors -- first real OS
- 1960s: IBM delivers System/360 OS
  - Details recounted in Mythical Man Month Book

- 1970-80s: Digital Equipment Corporation (DEC) and Data General (DG) lead the minicomputer market
  - Data General's initial design detailed in The Soul of a New Machine
  - There is no reason anyone would want a computer in their home. -- Ken Olsen, Founder and CEO of DEC

Brief Operating System History [link]

- 1981: IBM releases a Personal Computer (PC) to compete with Apple
  - Basic Input/Output System (BIOS) for low-level control
  - Three high-level OSes, including MS-DOS
  - Developers were asked to write software for DOS or BIOS, not bare-metal hardware
- 1982: Compaq and others release IBM-compatible PCs
  - Different hardware implementations (except 808x CPU)
  - Reverse engineered and reimplemented BIOS
  - Relied on customized version of MS-DOS

IBM Eventually Loses Control



- 1985: IBM clones dominated computer sales
  - Used the same underlying CPUs and hardware chips
  - Close to 100% BIOS compatibility
  - MS-DOS was ubiquitous
  - Thus, IBM PC hardware became the de-facto standard
- 1986: Compaq introduces 80386-based PC
- 1990's: Industry is dominated by "WinTel" (Microsoft and Intel)
  - Intel x86 CPU architectures (Pentium 1, 2, and 3)
  - Windows 3.1, NT, 95 software compatibility

Let’s build an operating system!

To build an OS, what tools would we need?

- Potential tools needed:
  - High-level programming languages
    - Assembly, C, …
  - Knowledge of
    - Computer architecture
  - Some idea about
    - How to divide up resources: memory, processes, etc.

- Looks like we have some of these foundations!

- Note this is not a hypothetical question, new Operating Systems are made all of the time
  - e.g., Android, iOS, etc.

First Design Decision: Kernel

(Reminder of the Kernel)

One Program to rule them all,
One Program to find them,

One Program to bring them all,
and in darkness bind them in the
Land of Linux where
programmers code

*Pop Culture reference from Lord of the Rings

Towards a Kernel

- "The one program running at all times on the computer" is the kernel
  - Typically the first program loaded up
    - (loaded by the bootloader--we'll get to this)

- Questions:
  - What are the features that kernels should implement?
  - How should we architect the kernel to support these features?
    - i.e. what feature does our kernel support and what goes into user land?

Kernel Features

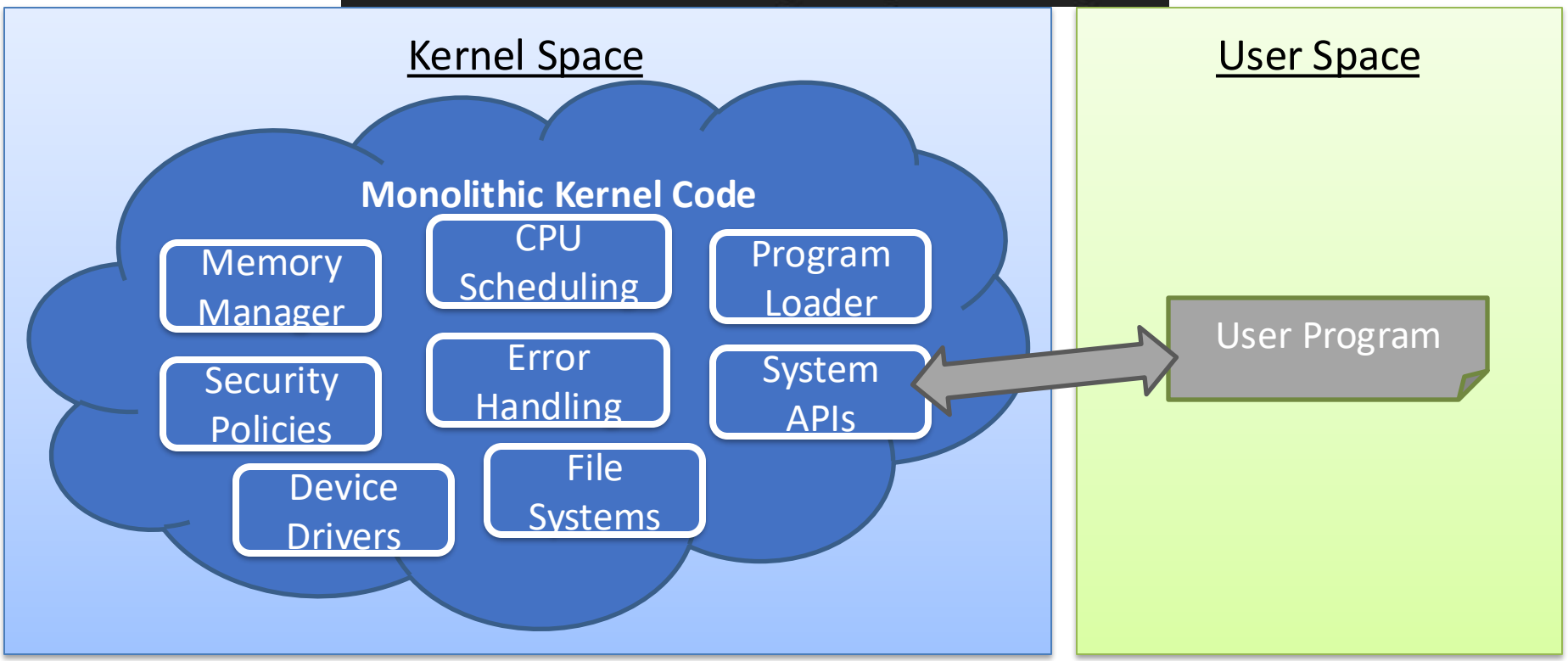
- Device management
  - Required: CPU and memory
  - Optional: disks, keyboards, mice, video, etc.
- Loading and executing programs
- System calls and APIs
- Protection and fault tolerance
  - E.g. a program crash shouldn't crash the computer
- Security
  - E.g. only authorized users should be able to login

Architecting Kernels: Three basic approaches

- Monolithic kernels
  - All functionality is compiled together
  - All code runs in privileged kernel-space
- Microkernels
  - Only essential functionality is compiled into the kernel
  - All other functionality runs in unprivileged user space
- Hybrid kernels
  - Most functionality is compiled into the kernel
  - Some functions are loaded dynamically
  - Typically, all functionality runs in kernel-space

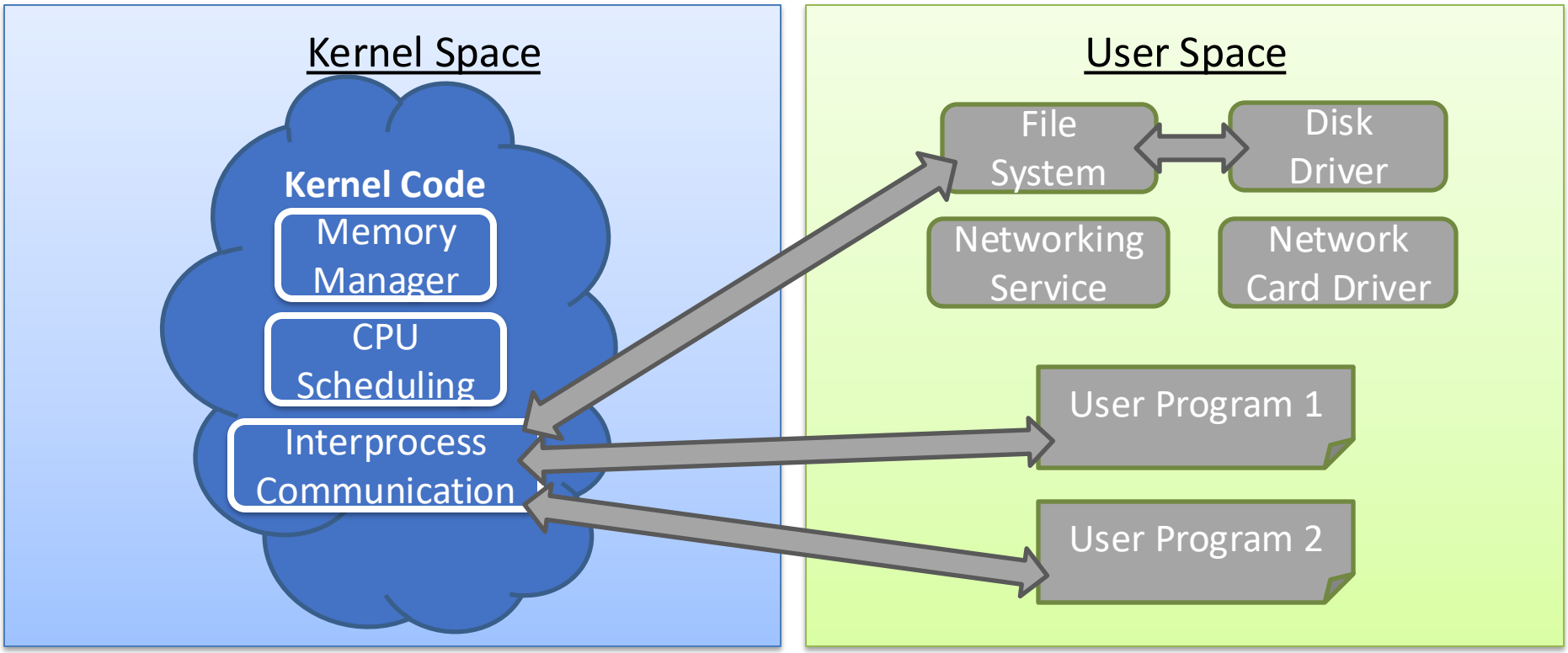
Monolithic Kernel

1. Monolithic kernels
   - All functionality is compiled together
   - All code runs in privileged kernel-space



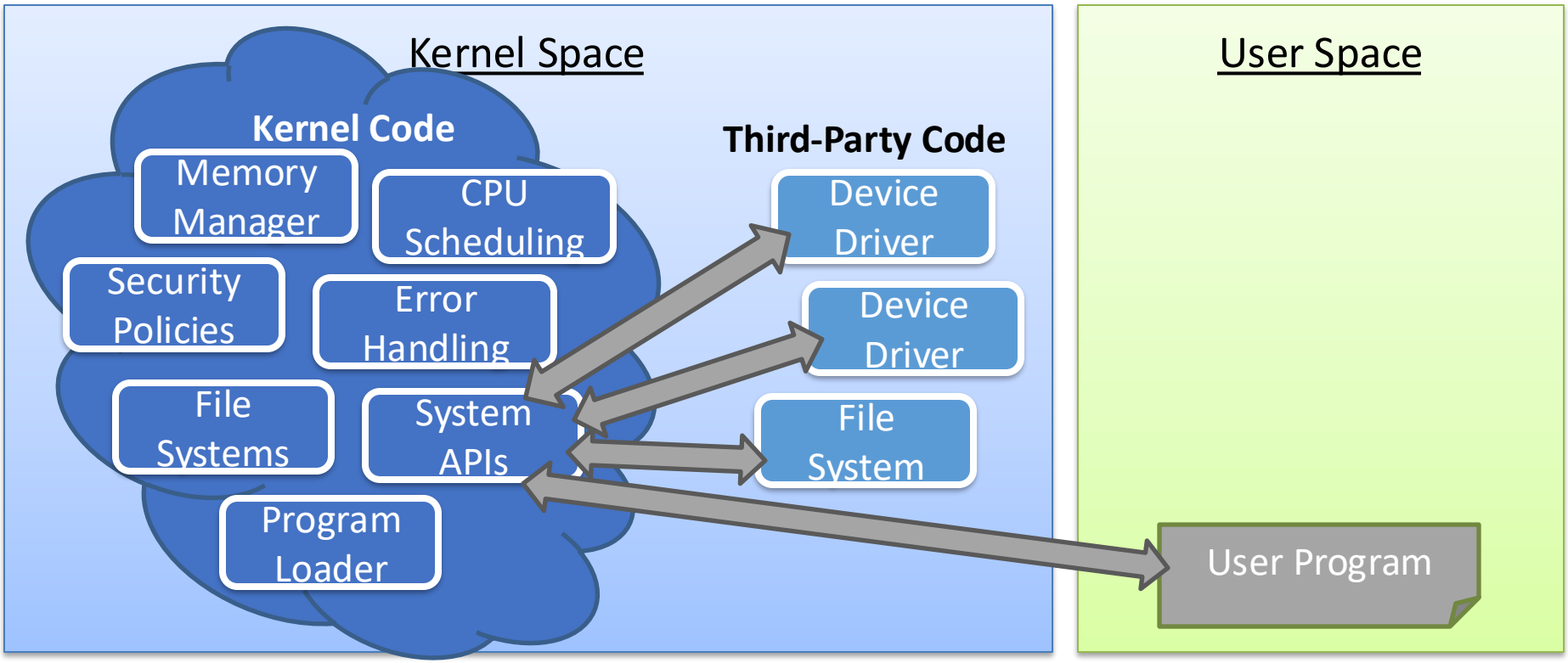
Microkernel

2. Microkernels
   - Only essential functionality is compiled into the kernel
   - All other functionality runs in unprivileged user space



Hybrid Kernel

3. Hybrid kernels
   - Most functionality is compiled into the kernel
   - Some functions are loaded dynamically
   - Typically, all functionality runs in kernel-space



Research Kernels:
Mach
L4
GNU Hurd

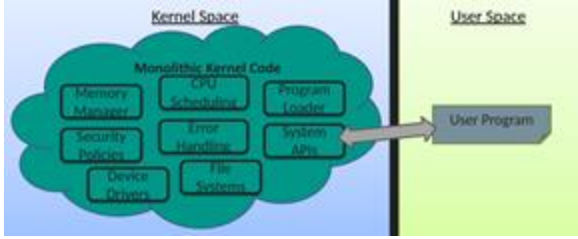
Kernels for
Embedded System:
QNX

Microkernels:
Small code base,
Few features

Hybrid Kernels:
Pretty large code base,
Some features delegated

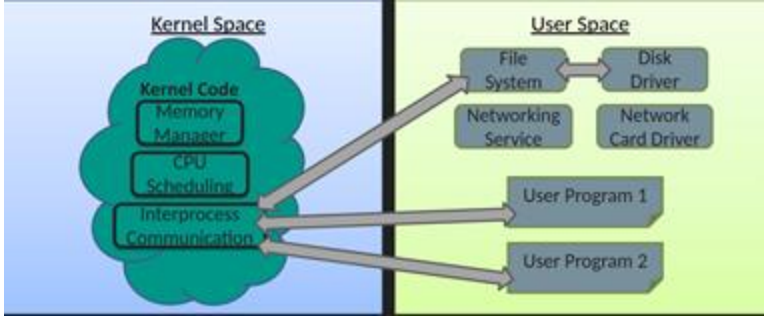
Monolithic Kernels:
Huge code base,
Many features

Pros/Cons of Monolithic Kernels



- Advantages?
  - Single code base eases kernel development
  - Robust APIs for application developers
  - No need to find separate device drivers
  - Fast performance due to tight coupling
- Disadvantages?
  - Large code base, hard to check for correctness
  - Bugs crash the entire kernel (and thus, the machine)

Pros/Cons of Microkernels



- Advantages?
  - Small code base, easy to check for correctness
  - Extremely modular and configurable
  - Choose only the pieces you need for embedded systems
  - Easy to add new functionality (e.g., a new file system)
  - Services may crash, but the system will remain stable

- Disadvantages?
  - Performance is slower: many context switches
  - No stable APIs, more difficult to write applications

Pros/Cons of Hybrid

- Some mix of the tradeoffs taken from the Microkernels and Monolithic kernels

Alright--let's spec out something closer to a hybrid kernel

Pieces of an Operating System

- We need to be able to perform some typical OS services
- Memory Management
- Some abstract data types (arrays, strings, etc.)
- Input and Output functions (printf, scanf, etc.)
- File System
- UI Management
- Textual Output
- Graphics
- Maybe more
  - Security, networking, multi-processing

Pieces of an Operating System

- We need to be able to perform some typical OS services
- Memory Management
- Some abstract data types (arrays, strings, etc.)
- Input and Output functions (printf, sc f, etc.)
- File System
- UI Management
- Textual Output
- Graphics
- Maybe more
  - Security, networking, multi-processing

If you take a close look, you'll notice some of these are starting to look like our 'system calls'

strace | `strace cat test.c`

- Remember the ‘strace’ tool?
- Something neat we can do too, is peak into all of these system calls that are being made--again we can see there is no magic

```
mike:~$ strace cat test.c
execve("/bin/cat", ["cat", "test.c"], 0x7ffc0ce64bc8 /* 60 vars */) = 0
brk(NULL)                                = 0x5650062aa000
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or dire
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or dire
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mo
mmap(NULL, 1940                                                  cd000
close(3)
access("/etc/lo                                                   file or dire
openat(AT_FDCWD                                                  |O_CLOEXEC) =
read(3, "\177EL                                                  34\2\0\0\0\0\
```

So at some level, we can think of an OS on the software side, as a collection of system calls--great!

But how do we get here from the hardware side?

CS 3650 Computer Systems – Summer 2025

OS Kernels, Booting, xv6 (1)

Week 10

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Ji-Yong Shin and Alden Jackson's lecture slides for the same course.

What is an Operating System?

- OS is software that sits between user programs and hardware

- OS provides interfaces to computer hardware
  - User programs do not have to worry about details
- OS is a resource manager and control program
  - Controls execution of user programs
  - Decides between conflicting requests for hardware access
  - Attempts to be efficient and fair
  - Prevents errors and improper use

Kernel Features

- Device management
  - Required: CPU and memory
  - Optional: disks, keyboards, mice, video, etc.
- Loading and executing programs
- System calls and APIs
- Protection and fault tolerance
  - E.g. a program crash shouldn't crash the computer
- Security
  - E.g. only authorized users should be able to login

Architecting Kernels: Three basic approaches

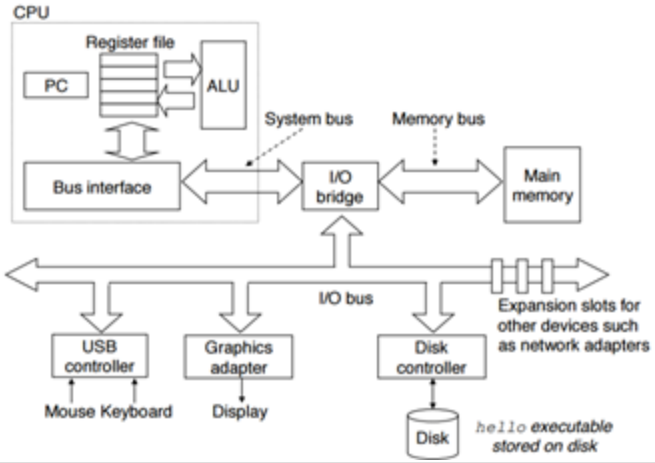
- Monolithic kernels
  - All functionality is compiled together
  - All code runs in privileged kernel-space
- Microkernels
  - Only essential functionality is compiled into the kernel
  - All other functionality runs in unprivileged user space
- Hybrid kernels
  - Most functionality is compiled into the kernel
  - Some functions are loaded dynamically
  - Typically, all functionality runs in kernel-space

Pieces of an Operating System

- We need to be able to perform some typical OS services
- Memory Management
- Some abstract data types (arrays, strings, etc.)
- Input and Output functions (printf, scanf, etc.)
- File System
- UI Management
- Textual Output
- Graphics
- Maybe more
  - Security, networking, multi-processing

The OS and Computer Architecture

- Let us say we have an OS like Linux
  - How does our architecture know what to do with an Operating System or where to load it from?



Does anything happen before our Operating System is running?

Pop Interview Question

- “What happens after you push the power button on your machine?” (i.e. what happens in software?)

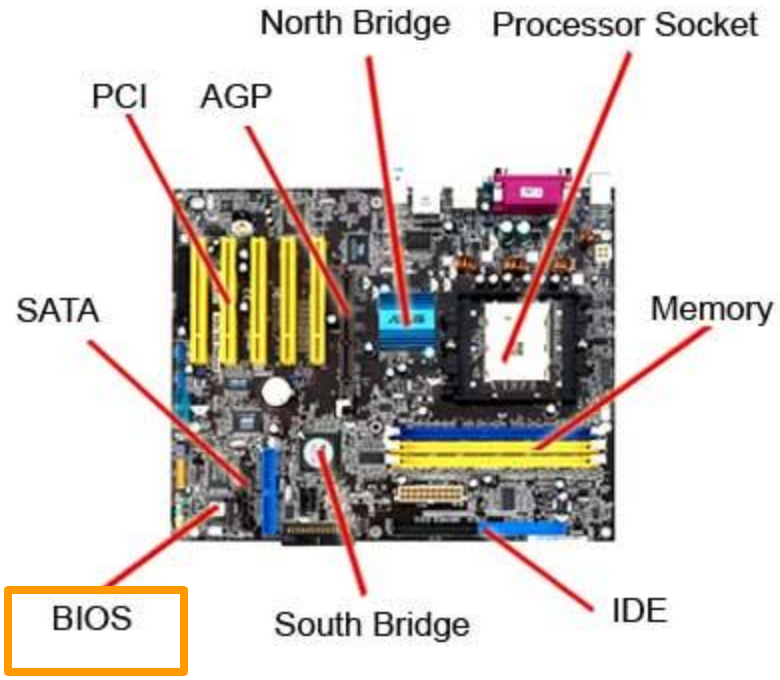
Pop Interview Question

- “What happens after you push the power button on your machine?” (i.e. what happens in software?)
- Understanding operating systems and putting together our hardware knowledge will answer this question!

Boot Process

The first program is executed: The BIOS

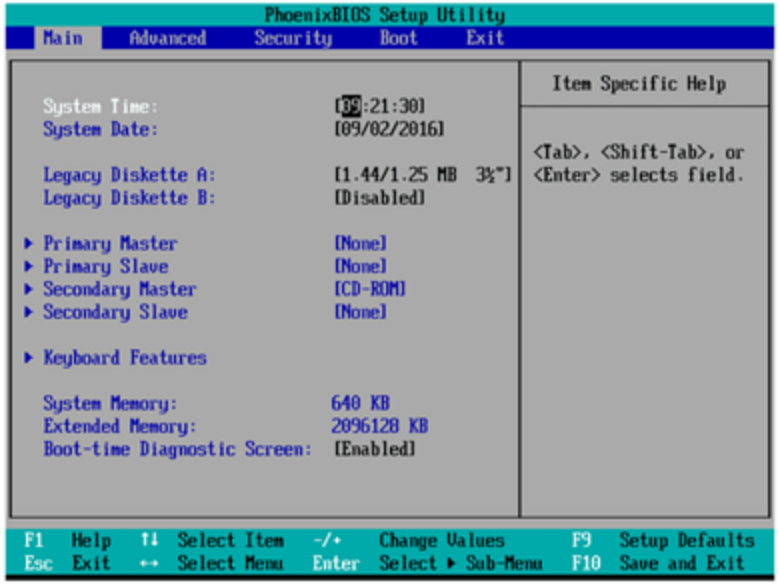
- x86 machines start by executing a program called the BIOS
  - BIOS: Basic Input/Output System
- The BIOS is 'baked into' our computers motherboard
  - This means it is stored in non-volatile memory
    (i.e. memory that persists)
  - (A motherboard is the entirety of the printed circuit you see on the right. It helps organize all of the components that are attached together).



More on BIOS and the ‘boot loader’ (1/2)

- The Basic Input/Output System’s (BIOS) job is to make sure that all of the hardware is ready to go

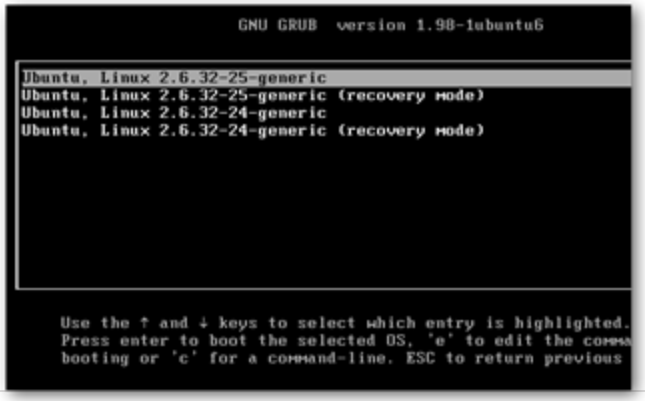
- If all of the components are ready, then control is transferred into what is called the ‘boot loader’



More on BIOS and the ‘boot loader’ (2/2)

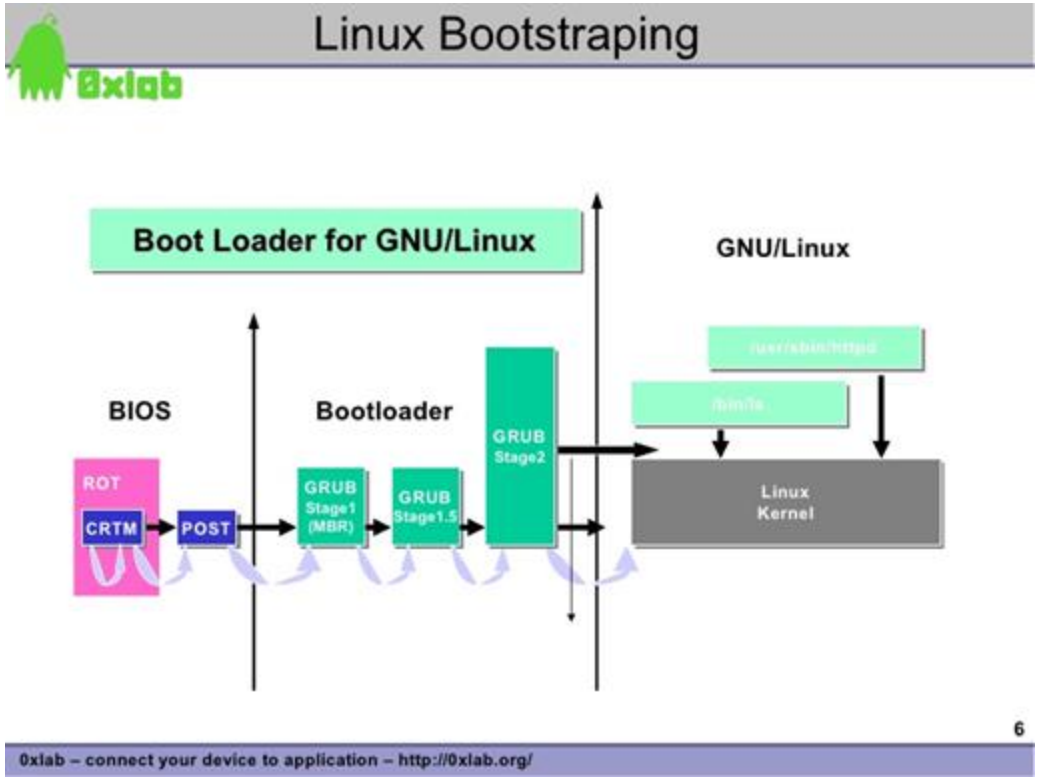
- The BIOS transfers control to the ‘boot loader’ by looking at the ‘boot sector’, which has some amount of bytes (e.g. 512 bytes) that tell us where the boot loader is.
  - You may have seen programs like GRUB which allow you to select which operating system to load.

- Our goal at this stage, is to use this very primitive ‘boot loader’ program, to launch and execute a more modern operating system.
  - e.g. Windows, MacOS, Ubuntu, CentOS, etc.



Here is the OS loading process

- Here is the high-level abstraction--at the very least the steps to remember
  - BIOS
  - Boot loader
  - Operating System



[image source]

Digging deeper

Pushing power

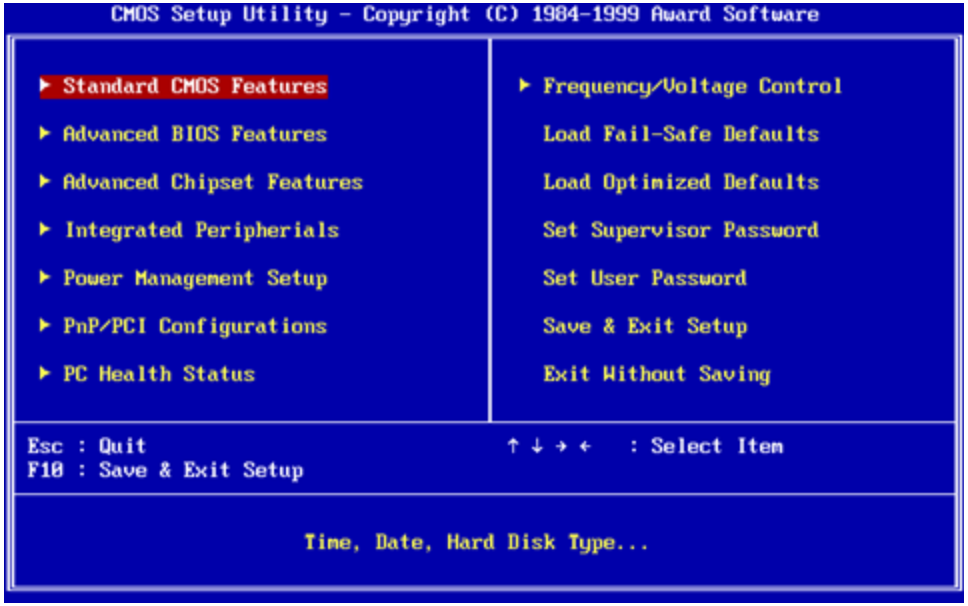
1. Start the BIOS
2. Load settings from CMOS (complementary metal-oxide semiconductor)
3. Initialize any attached devices
4. Run POST (Power on self-test)
5. Initiate the bootstrap sequence

Starting the BIOS (1/5)

- Basic Input/Output System (BIOS)
  - A mini-OS burned onto a chip
- Begins executing as soon as a PC powers on
  - Code from the BIOS chip gets copied to RAM at a low address (e.g. 0xFF)
  - jmp 0xFF (16 bits) written to RAM at 0xFFFF0
  - x86 CPUs always start with 0xFFFF0 in the EIP register
- Essential goals of the BIOS
  - Check hardware to make sure its functional
  - Install simple, low-level device drivers
  - Scan storage media for a Master Boot Record (MBR)
    - Load the boot record into RAM
    - Tells the CPU to execute the loaded code

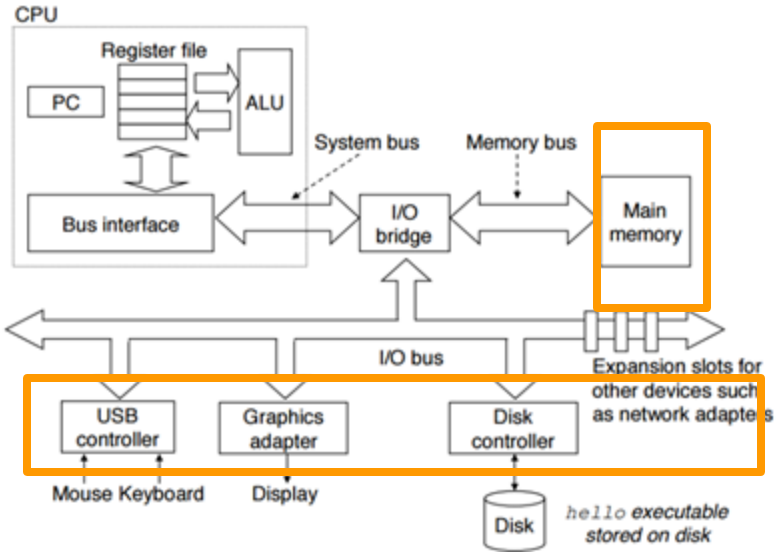
Load settings from CMOS (2/5)

- BIOS often has configurable options
  - Values are stored in a special battery-backed CMOS memory
  - These values are then read in by the BIOS, often containing information about how devices have been configured.



Initialize any attached devices (3/5)

- Scans and initializes hardware
  - CPU and memory
  - Keyboard and mouse
  - Video
  - Bootable storage devices
- Installs interrupt handlers in memory
  - Builds the Interrupt Vector Table
- Runs additional BIOSes on expansion cards
  - Video cards and SCSI cards often have their own BIOS



Run Power On Self-Test (POST) test (4/5)

- This is a diagnostic test to make sure all of the devices that are connected and initialized in the previous steps are working.
- POST Test
  - Check RAM by read/write to each address
  - Check to make sure keyboard is working
  - Check to make sure connected hard drives are working
  - etc.

Bootstrap in an operating system (5/5)

- Finally we need to find and load a real OS

- BIOS identifies all potentially bootable devices
  - Tries to locate Master Boot Record (MBR) on each device
  - Order in which devices are tried is configurable

- Master Boot Record (MBR) has code that can load the actual OS
  - Code is known as a bootloader

- Example bootable devices:
  - Hard drive, SSD, floppy disk, CD/DVD/Bluray, USB flash drive, network interface card (NIC)

The Master Boot Record (MBR)

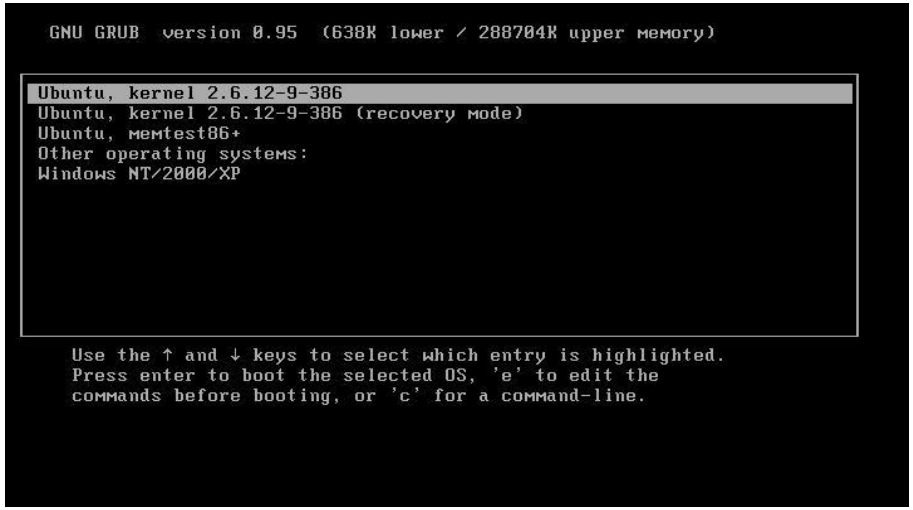
- Special 512-byte file in sector 1 (address 0) of a storage device
- Contains
  - 446 bytes of executable code
  - Entries for 4 partitions

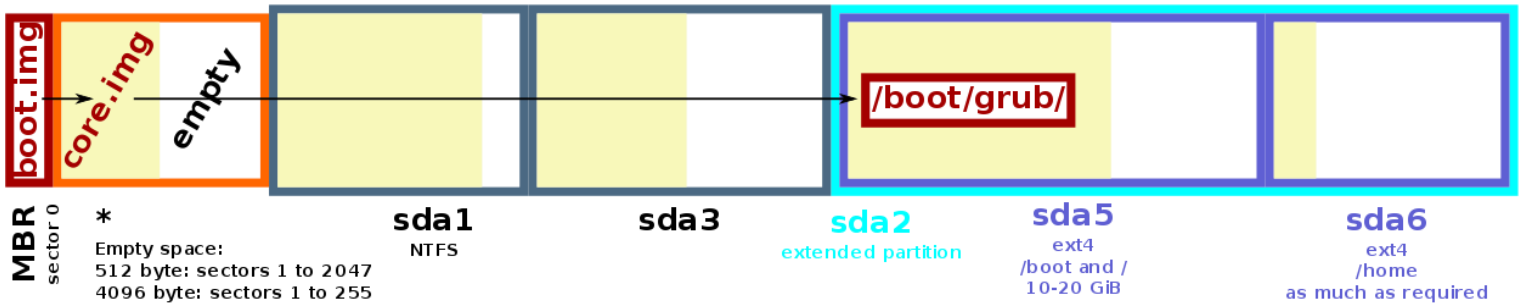
Address		Description	Size (Bytes)
Hex	Dec.		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2
		Total:	**512**

- Too small to hold an entire OS
  - Starts a sequence of chain-loading

Example Bootloader: GRUB

- Grand Unified Bootloader
  - Used with Unix, Linux, Solaris, etc.





*Source: https://en.wikipedia.org/wiki/GNU_GRUB

We need to find and load a real OS now (xv6)

But now lets really see it in action

- We will actually work with a small operating system so we can see exactly what the code looks like.

Introducing xv6!

Goal: Figure out the boot process from a programmer's perspective

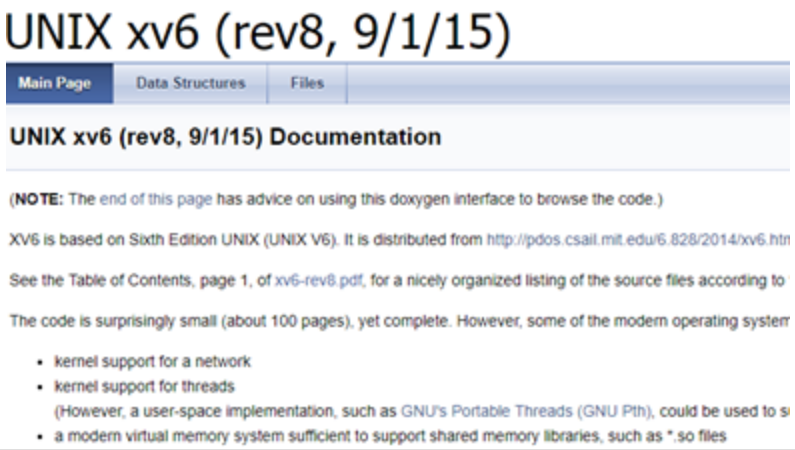
- Our tool is going to be to use the xv6 operating system.
  - xv6 is yet another Unix inspired variant--although much more lightweight (Several thousands of lines of code versus millions).

Our tool xv6 | https://pdos.csail.mit.edu/6.828/2017/xv6.html


Xv6, a simple Unix-like teaching operating system

Introduction

- Not something your instructor developed
- But some folks at MIT have been working on this for long
  - You can and certainly should browse this link for a deeper dive.
  - There is some handy documentation if you want to browse online from NEU faculty (be warned, this is 2 revisions old) https://course.ccs.neu.edu/cs3650/unix-xv6/


UNIX xv6 (rev8, 9/1/15)

Main Page | Data Structures | Files

UNIX xv6 (rev8, 9/1/15) Documentation

(NOTE: The end of this page has advice on using this doxygen interface to browse the code.)

XV6 is based on Sixth Edition UNIX (UNIX V6). It is distributed from http://pdos.csail.mit.edu/6.828/2014/xv6.htm

See the Table of Contents, page 1, of xv6-rev8.pdf, for a nicely organized listing of the source files according to

The code is surprisingly small (about 100 pages), yet complete. However, some of the modern operating system

- kernel support for a network
- kernel support for threads
  (However, a user-space implementation, such as GNU's Portable Threads (GNU Pth), could be used to s
- a modern virtual memory system sufficient to support shared memory libraries, such as *.so files

xv6

- Monolithic kernel
- Runs on x86 processors
  - Note that x86-based versions are no longer maintained
  - xv6 development has moved on to RISC-V

- Refer to the course webpage for useful resources
  - https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf
  - https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf

Boot process in xv6

Files we will look at

- Bootasm.S
  - Real mode -> protected mode
  - Calls bootmain.c
- Bootmain.c
  - Reads main from disk
- Main.c
  - Initializes the kernel
- Proc.c
  - Process creation and scheduling
- Initcode.S
  - Starter code for init process
- Init.c
  - Init process

bootasm.S - real mode to protected mode

- Real mode
  - x86 machine starts with real mode
  - Simulates the old Intel 8088 (1979)
  - 16 bit registers
  - 20 bit memory address (1MB memory)
  - No virtual memory support
  - No memory protection
  - No paging support

- (32bit) Protected mode (CR0 register)
  - Virtual address space enabled
  - Max 4GB memory
  - Protected ring support (recall ring 0 to 3)

bootasm.S - Where our bootstrapping process begins

```
# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.
.code16                     # Assemble for 16-bit mode
.globl start
start:
  cli                       # BIOS enabled interrupts; disable
  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax           # Set %ax to zero
  movw    %ax,%ds           # -> Data Segment
  movw    %ax,%es           # -> Extra Segment
  movw    %ax,%ss           # -> Stack Segment
```

Bootmain.c: loads ELF kernel from disk

```
// Boot loader.
// Part of the boot block, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"
#define SECTSIZE  512
void readseg(uchar*, uint, uint);
void bootmain(void)
```

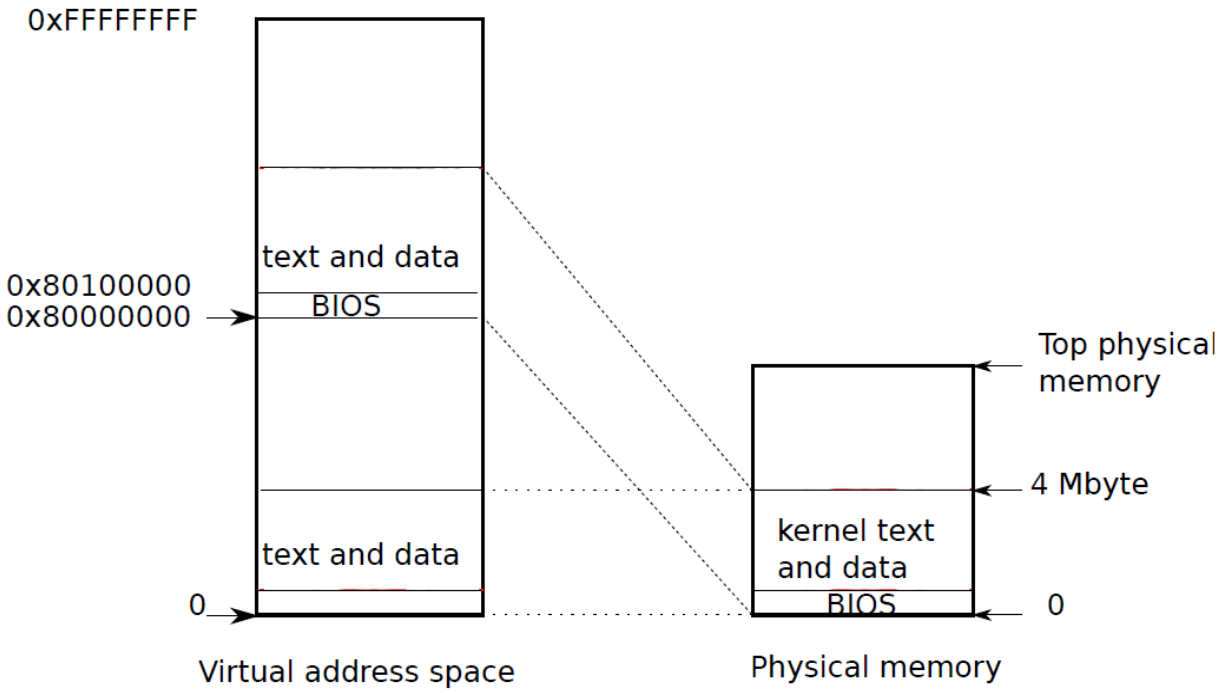
main.c

- After we have successfully bootstrapped, we can begin executing main

- We can actually see various parts of the OS that get setup!
  - Handling files, working with disk, setting up processes, etc.

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
 kinit1(end, P2V(4*1024*1024)); // phys page allocator
 kvmalloc();      // kernel page table
 mpinit();        // detect other processors
 lapicinit();     // interrupt controller
 seginit();       // segment descriptors
 picinit();       // disable pic
 ioapicinit();    // another interrupt controller
 consoleinit();   // console hardware
 uartinit();      // serial port
 pinit();         // process table
 tvinit();        // trap vectors
 binit();         // buffer cache
 fileinit();      // file table
 ideinit();       // disk
 startothers();   // start other processors
 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
 userinit();      // first user process
 mpmain();        // finish this processor's setup
}
```

Memory


0xFFFFFFFF
text and data
0x80100000
0x80000000
BIOS
text and data
0
Virtual address space
Top physical memory
4 Mbyte
kernel text and data
BIOS
0
Physical memory

User init

- Userinit
- Creates a process from process table
- Run initcode.S which the compiled binary is part of the kernel
- Initcode.S code “exec” compiled binary at /init (i.e., init.c)
- Init.c opens console for stdin, stdout, sterr and forks shell

proc.c

- Once our OS is running, proc schedules different processes from a table to run
  - See ‘scheduler’ in proc.c

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      ...
```

Walkthrough of xv6 Scheduler

- Thinking about some of these trade-offs, it will be beneficial to look at things from an xv6 perspective.
  - Investigating ‘scheduler’ within xv6 will show how scheduling is done.

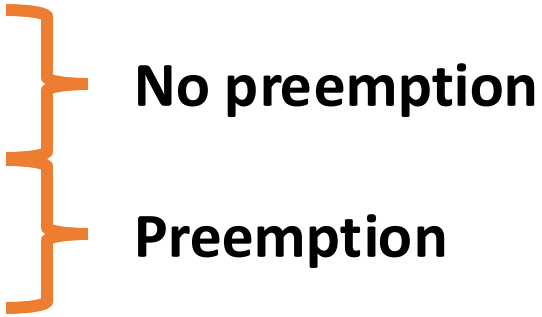
Operating System Scheduler

- The scheduler in an Operating system is responsible for picking which process runs.
- The OS gives each process a ‘time slice’ to execute.
- The OS tries to be fair in making sure every process can make some progress
- However, there are some trade-offs
  - Should a long running process using lots of resources get more time?
  - Or would we rather have short running processes just finish and be done?
  - How does the Operating System even know or estimate time spent?

Basic Scheduler Architecture

- Scheduler selects from *ready* processes, and assigns them to a CPU
  - System may have >1 CPU
  - Various different approaches for selecting processes

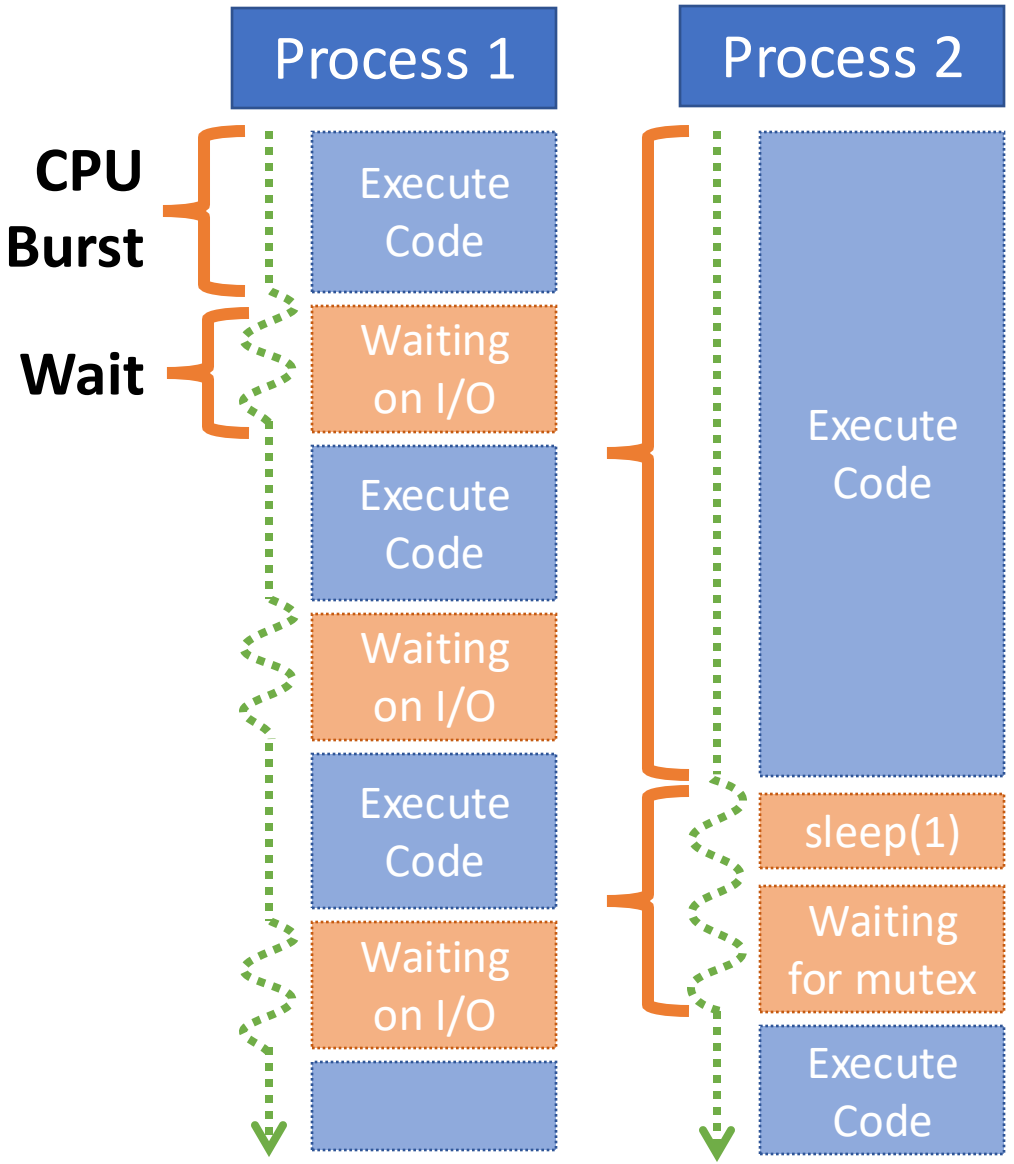
- Scheduling decisions are made when a process:
  1. Switches from *running* to *waiting*
  2. Terminates
  3. Switches from *running* to *ready*
  4. Switches from *waiting* to *ready*



- Scheduler may have access to additional information
  - Process deadlines, data in shared memory, etc.

Basic Process Behavior

- Processes alternate between doing work and waiting
  - Work → CPU Burst
- Process behavior varies
  - I/O bound
  - CPU bound
- Expected CPU burst distribution is important for scheduler design
  - Do you expect more CPU or I/O bound processes?



Scheduling Optimization Criteria

- Max CPU utilization – keep the CPU as busy as possible
- Max throughput – # of processes that finish over time
  - Min turnaround time – amount of time to finish a process
  - Min waiting time – amount of time a *ready* process waits until it runs
- Min response time – amount time between submitting a request and receiving a response
  - E.g. time between clicking a button and seeing a response
- Fairness – all processes receive fair CPU resources

- **No scheduler can meet all these criteria**
- Which criteria are most important depend on types of processes and expectations of the system
  - E.g. response time is key on the desktop
  - Throughput is more important for MapReduce

First Come, First Serve (FCFS)

- Simple scheduler
  - Processes stored in a FIFO queue
  - Served in order of arrival

Process	Burst Time	Arrival Time
P1	24	0.000
P2	3	0.001
P3	3	0.002

- Turnaround time = completion time - arrival time
  - P1 = 24; P2 = 27; P3 = 30
  - Average turnaround time: (24 + 27 + 30) / 3 = 27

The Convoy Effect

- FCFS scheduler, but the arrival order has changed

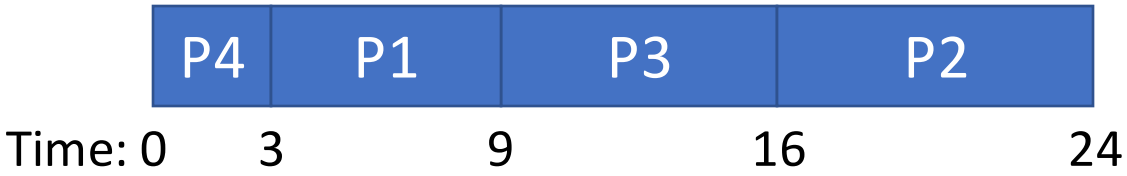
Process	Burst Time	Arrival Time
P1	24	0.002
P2	3	0.000
P3	3	0.001

- Turnaround time: P1 = 30; P2 =3; P3 = 6
  - Average turnaround time: (30 + 3 + 6) / 3 = 13
  - Much better than the previous arrival order!
- Convoy effect (a.k.a. head-of-line blocking)
  - Long process can impede short processes
  - E.g.: CPU bound process followed by I/O bound process

Shortest Job First (SJF)

- Schedule processes based on the length of their next CPU burst time
  - Shortest processes go first

Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	7	0
P4	3	0



- Average turnaround time: (3 + 9 + 16 + 24) / 4 = 13
- SJF is optimal: guarantees minimum average wait time (if all jobs arrive at the same time)

We already have an optimal solution … Do you see any problem?

Predicting Next CPU Burst Length

- Problem: future CPU burst times may be unknown
- Solution: estimate the next burst time based on previous burst lengths
  - Assumes process behavior is not highly variable
  - Use exponential averaging
    - t_n – measured length of the n^{th} CPU burst
    - τ_{n+1} – predicted value for $n+1^{th}$ CPU burst
    - α – weight of current and previous measurements ($0 \leq \alpha \leq 1$)
    - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\, \tau_n$
  - Typically, $\alpha = 0.5$

What About Arrival Time?

- SJF scheduler, CPU burst lengths are known

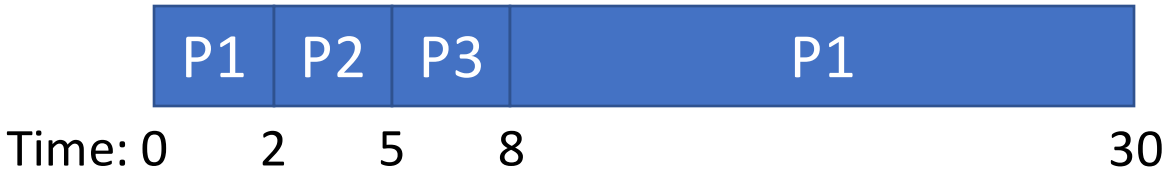
Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

- Scheduler must choose from available processes
  - Can lead to head-of-line blocking
  - Average turnaround time: (24 + 25 + 27) / 3 = 25.3

Shortest Time-To-Completion First (STCF)

- Also known as Preemptive SJF (PSJF)
  - Processes with long bursts can be context switched out in favor or short processes

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3



- Turnaround time: P1 = 30; P2 = 3; P3 = 5
  - Average turnaround time: (30 + 3 + 5) / 3 = 12.7
- STCF is also optimal
  - Assuming you know future CPU burst times

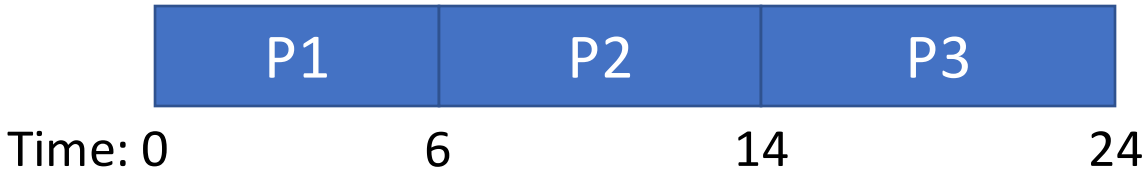
Interactive Systems

- Imagine you are typing/clicking in a desktop app
  - You don't care about turnaround time
  - What you care about is responsiveness
    - E.g. if you start typing but the app doesn't show the text for 10 seconds, you'll become frustrated
- Response time = first run time – arrival time

Response vs. Turnaround

- Assume an STCF scheduler

Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	10	0



- Avg. turnaround time: (6 + 14 + 24) / 3 = 14.7
- Avg. response time: (0 + 6 + 14) / 3 = 6.7

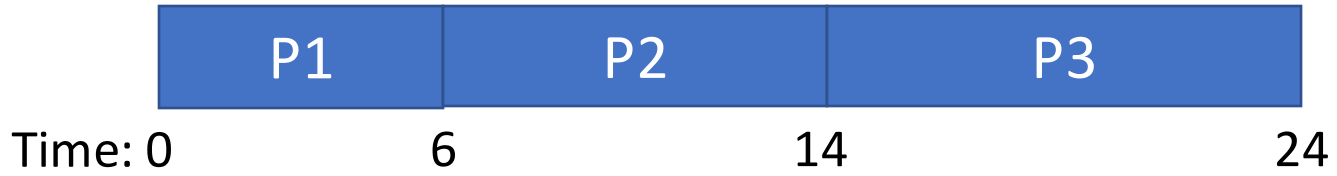
Round Robin (RR)

- Round robin (a.k.a time slicing) scheduler is designed to reduce response times
  - RR runs jobs for a time slice (a.k.a. scheduling quantum)
  - Size of time slice is some multiple of the timer-interrupt period

RR vs. STCF

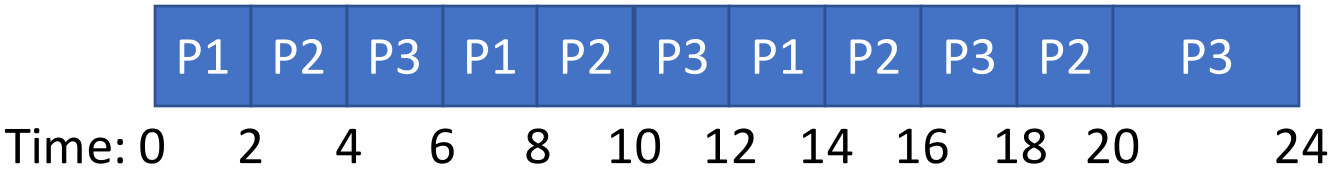
Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	10	0

STCF



- Avg. turnaround time: (6 + 14 + 24) / 3 = **14.7**
- Avg. response time: (0 + 6 + 14) / 3 = **6.7**

RR



- 2 second time slices
- Avg. turnaround time: (14 + 20 + 24) / 3 = **19.3**
- Avg. response time: (0 + 2 + 4) / 3 = **2**

Tradeoffs

RR

+ Excellent response times
  + With *N* process and time slice of *Q*…
  + No process waits more than *N*-1 time slices
+ Achieves fairness
  + Each process receives 1/*N* CPU time
- Worst possible turnaround times
  - If *Q* is large → FIFO behavior

STCF

+ Achieves optimal, low turnaround times
- Bad response times
- Inherently unfair
  - Short jobs finish first

- Optimizing for turnaround or response time is a trade-off
- Achieving both requires more sophisticated algorithms

Selecting the Time Slice

- Smaller time slices = faster response times
- So why not select a very tiny time slice?
  - E.g. 1µs
- Context switching overhead
  - Each context switch wastes CPU time (~10µs)
  - If time slice is too short, context switch overhead will dominate overall performance
- This results in another tradeoff
  - Typical time slices are between 1ms and 100ms