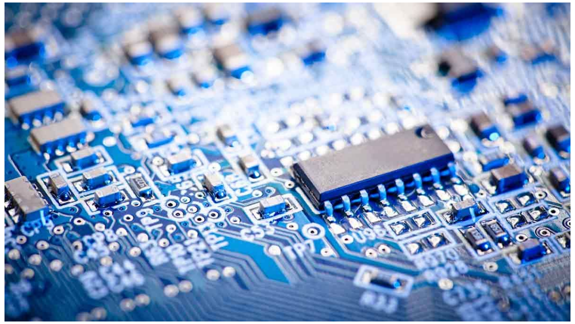
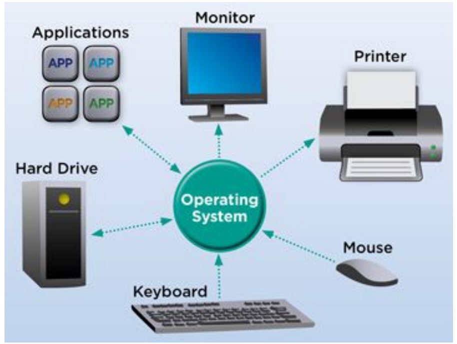
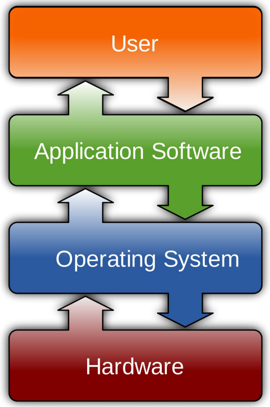
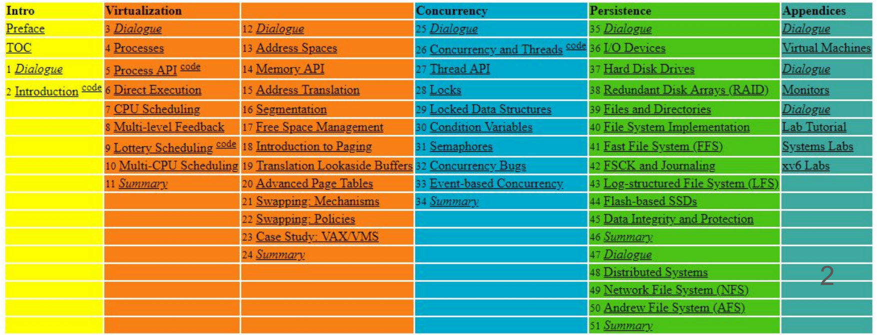




CS 3650 Computer Systems

Dr. Alden Jackson

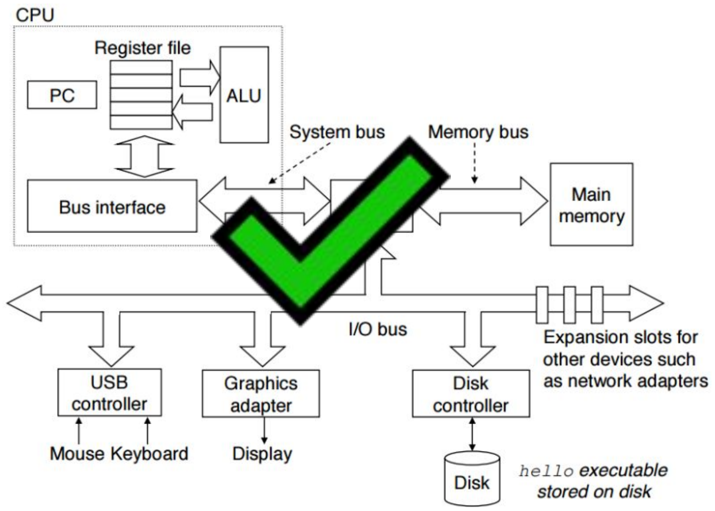


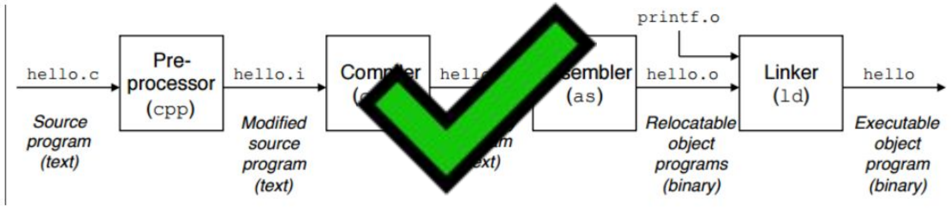


Lecture 10 - xv6

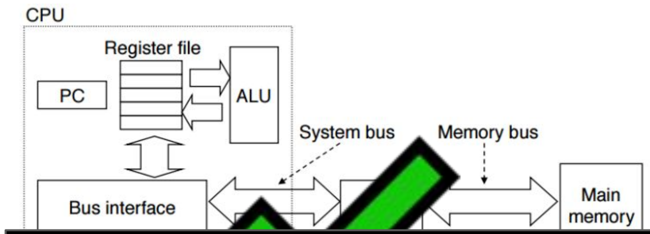
Putting it all together
(One more deep dive)

So far, we have knocked out two pictures



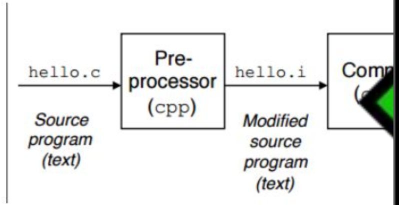


Here are some of the core topics we have looked at



Some core topics

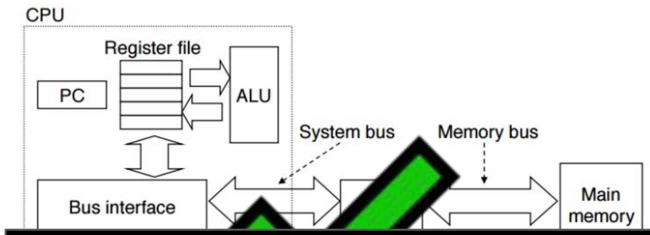
- Assembly
- C
- Memory
- GPU
- General CPU Architecture



Some core topics

Compilers
Linkers

And we have had partial coverage of OS

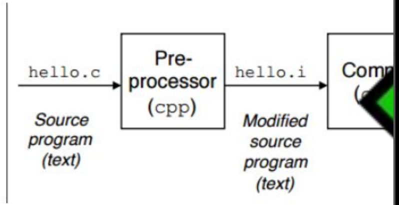


Some core topics

- Assembly
- C
- Memory
- GPU
- General CPU Architecture

Some core topics

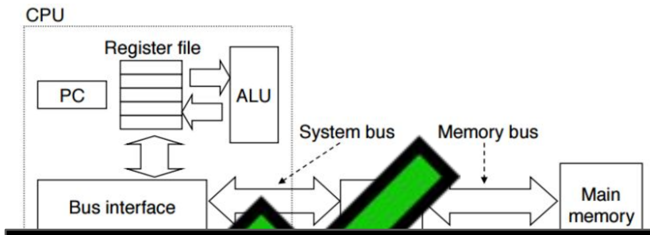
- Context Switching
- System Calls
- Exception Handling
- Virtual Memory

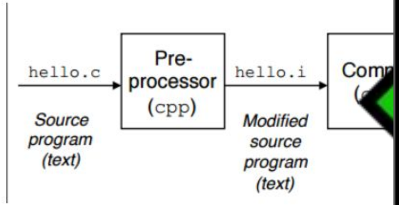


Some core topics

Compilers
Linkers

Now we'll get some experience with a working OS!





Some core topics

- Context Switching
- System Calls
- Exception Handling
- Virtual Memory

Some core topics

- Assembly
- C
- Memory
- GPU
- General CPU Architecture

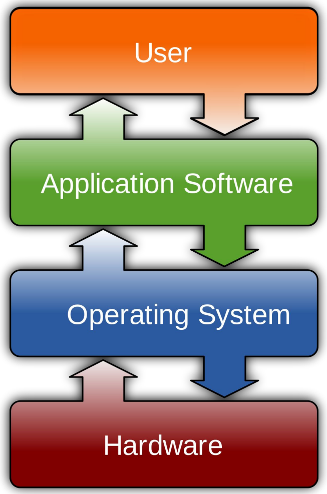
Some core topics

Compilers
Linkers

Operating System Refresher

(First day of class--what is an OS? Answer)

- Any and all software that sits between a user program and hardware
- OS is a resource manager and allocator
  - e.g. Handles conflicts between processes for hardware access
  - And it tries to be as efficient and fair as possible
- Overall an OS is a control program or “conductor”
  - Controls the execution of user programs
  - Prevents errors and improper use to maintain uptime.
- Additionally we discussed that there are many different Operating Systems that exist.
  - We have spent a lot of time in the POSIX environment
    - Our CCIS machines are CentOS
    - POSIX = portable operating system interface (It’s a standard)

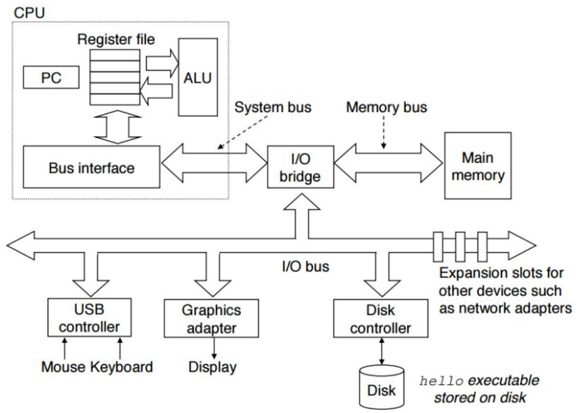


Without an operating system

- 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)

The OS and Computer Architecture

- Okay, great, 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?
  - So far we have some idea about how our OS work with devices?
    - (Interfacing with drivers)
  - We also have a pretty good idea how the OS works with memory at least on a process basis.
    - But we'll want to think even more about how processes are scheduled.



Operating System History

Brief Operating System History [link]

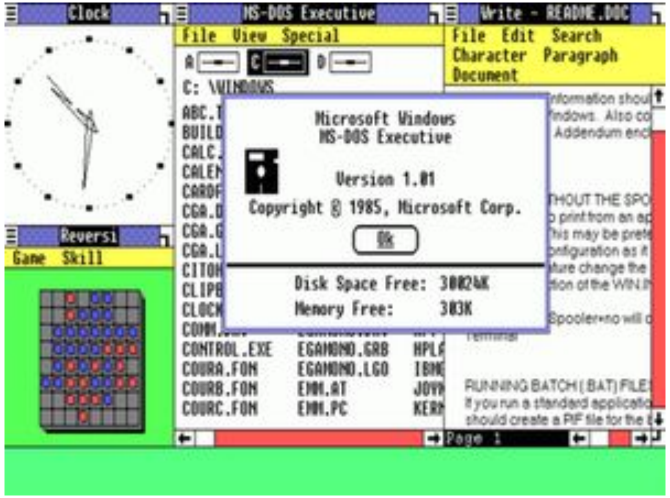
- 1955 and earlier: Very early mainframes have no operating system
- 1956: GM-NAA I/O used for research by General Motors -- first real OS
- 1960s: IBM delivers System/360 OS
  - Details recounted in Mythical Man Month Book
- 1970-80s: Digital Equipment Corporation (DEC) and Data General (DG) lead the 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



Microsoft Windows
MS-DOS Executive

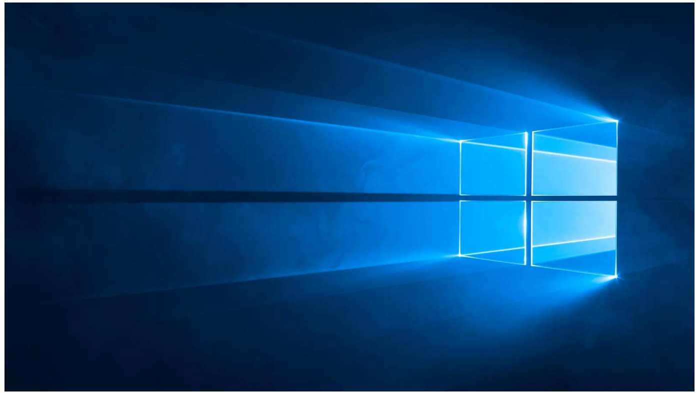
Version 1.01

Copyright © 1985, Microsoft Corp.

Ok

Disk Space Free: 30024K
Memory Free: 303K

Let’s build an operating system!



To build an Operating System, what tools would we need?

- Potential tools needed:
    - A high-level programming language
        - C
    - Knowledge of computer architecture
    - Some idea about how to divide up resources like memory, processes, etc.
- Looks like we have some of these foundations!

To build an Operating System, what tools would we need?

- Potential tools needed:
    - A high-level programming language
        - C
    - Knowledge of computer architecture
    - Some idea about how to divide up resources like 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 features does our kernel support?

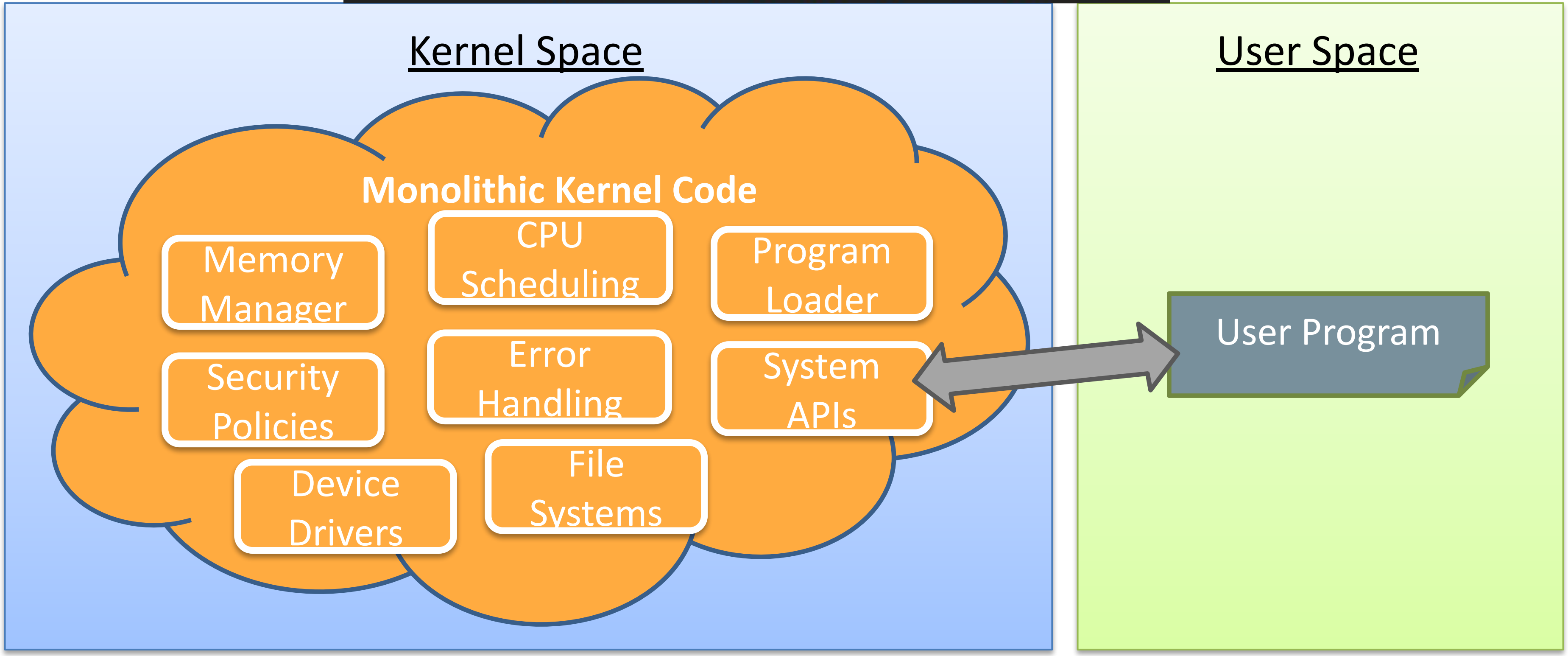
Kernel Features

- 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

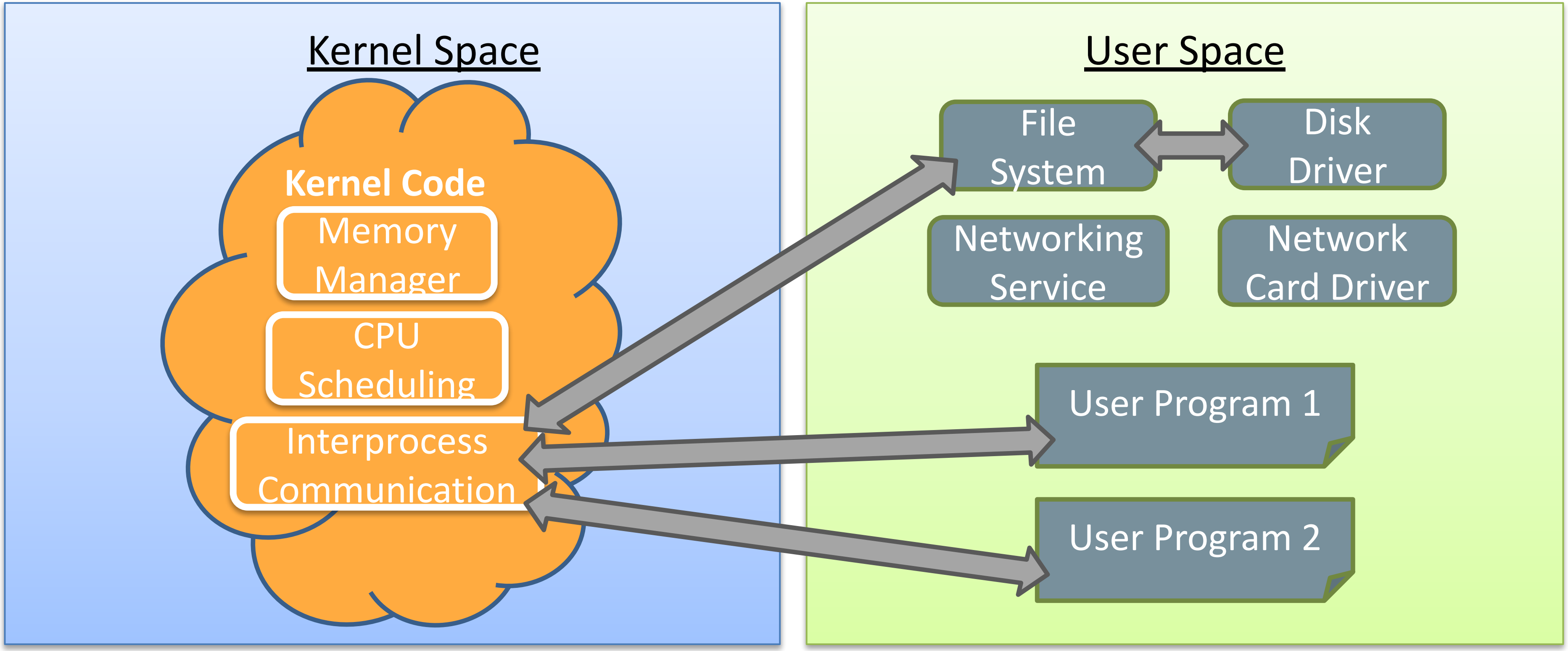
1. Monolithic kernels
   - All functionality is compiled together
   - All code runs in privileged kernel-space
2. Microkernels
   - Only essential functionality is compiled into the kernel
   - All other functionality runs in unprivileged user space
3. Hybrid kernels
   - Most functionality is compiled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

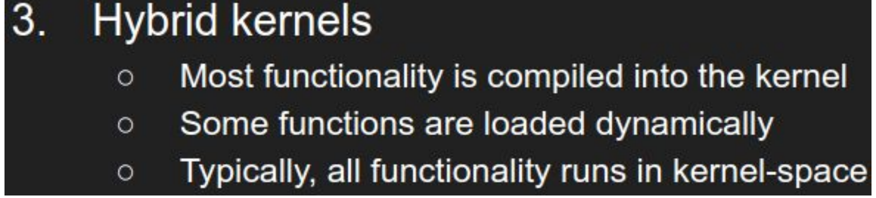
1. Monolithic kernels
   - All functionality is compiled together
   - All code runs in privileged kernel-space



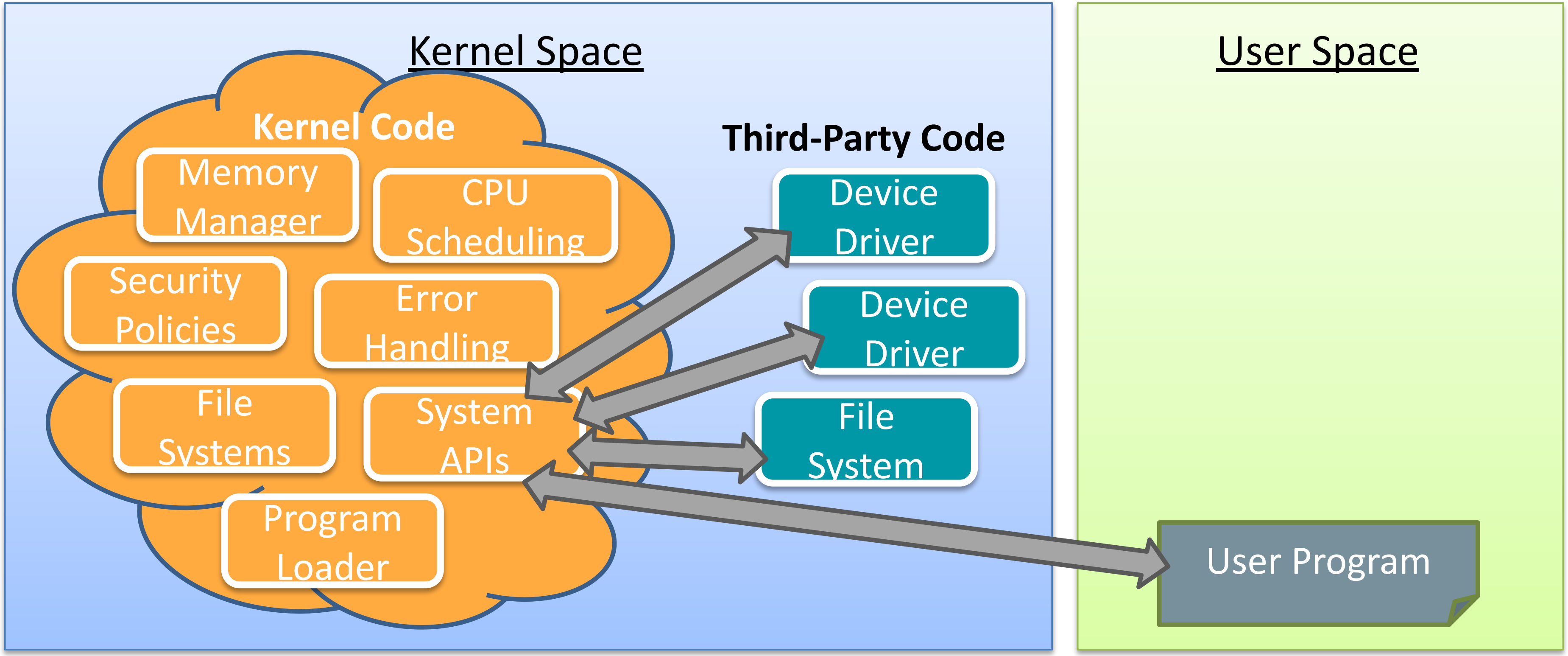
2. Microkernels

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





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

Windows®7

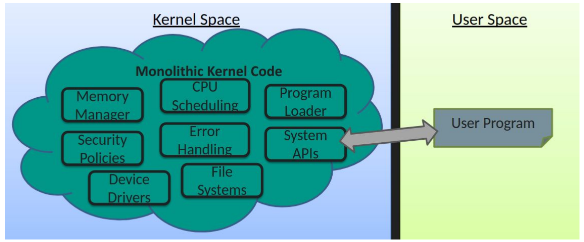
iOS

Microkernels:
Small code base,
Few features

Hybrid Kernels:
Pretty large code base,
Some features delegated

Monolithic Kernels:
Huge code base,
Many features

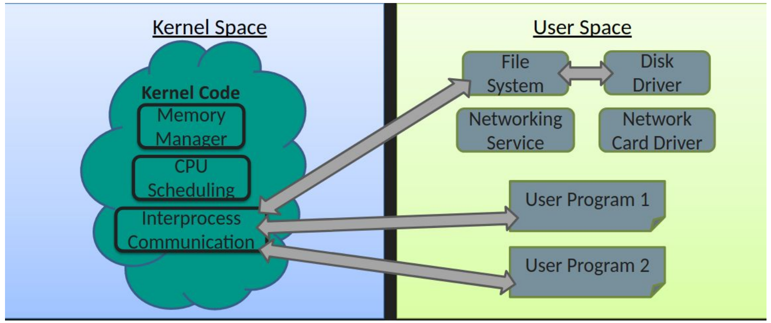
Pros/Cons of Monolithic Kernels



- 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
    - Excellent for high-security systems
    - 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

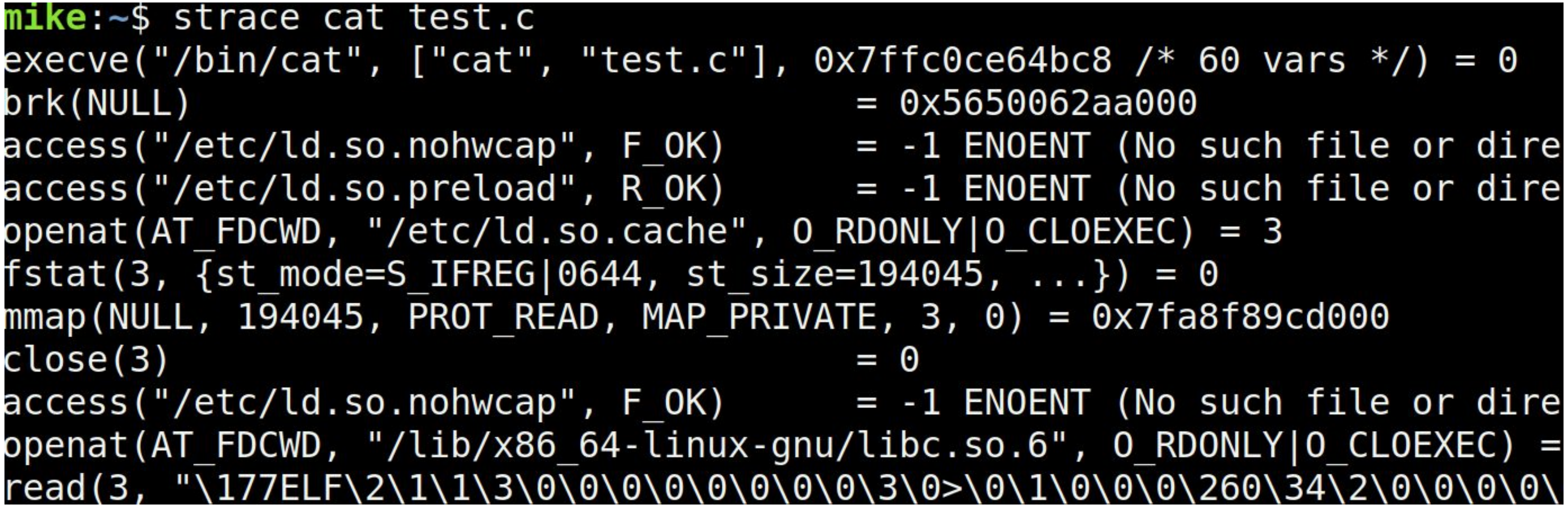
We need to be able to perform some typical OS services

- Memory Management
- Some abstract data types (arrays, strin
- Input and Output functions (printf, s
- 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`

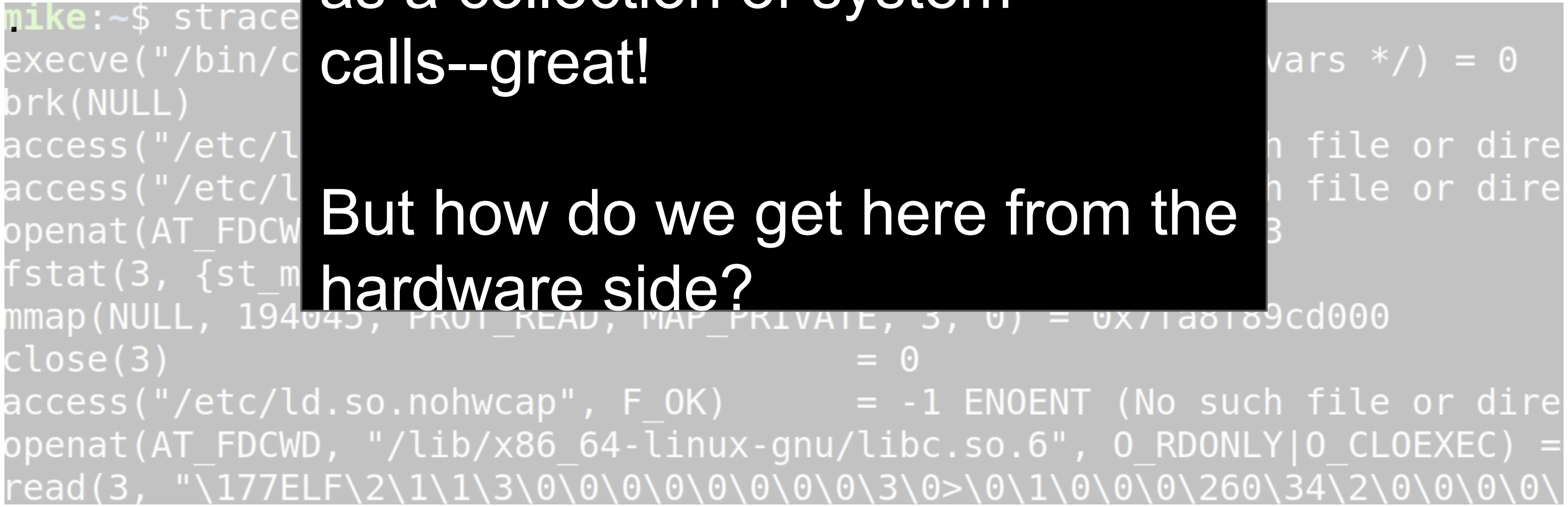
- Remember the ‘strace’ tool?
- Something neat we can do too, is peak into all of these system calls that are being made--again we can see there is no magic



```
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_mode=S_IFREG|0644, st_size=194045, ...}) = 0
mmap(NULL, 194045, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa8f89cd000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or dire
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) =
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\
```

strace | `strace cat test.c`

- Remember th
- Something ne em calls that are being made--a
- .



```
mike:~$ strace
execve("/bin/c                                        vars */) = 0
brk(NULL)
access("/etc/l                                        h file or dire
access("/etc/l                                        h file or dire
openat(AT_FDCW                                        3
fstat(3, {st_m
mmap(NULL, 194045, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa8189cd000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or dire
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) =
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\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?

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?)

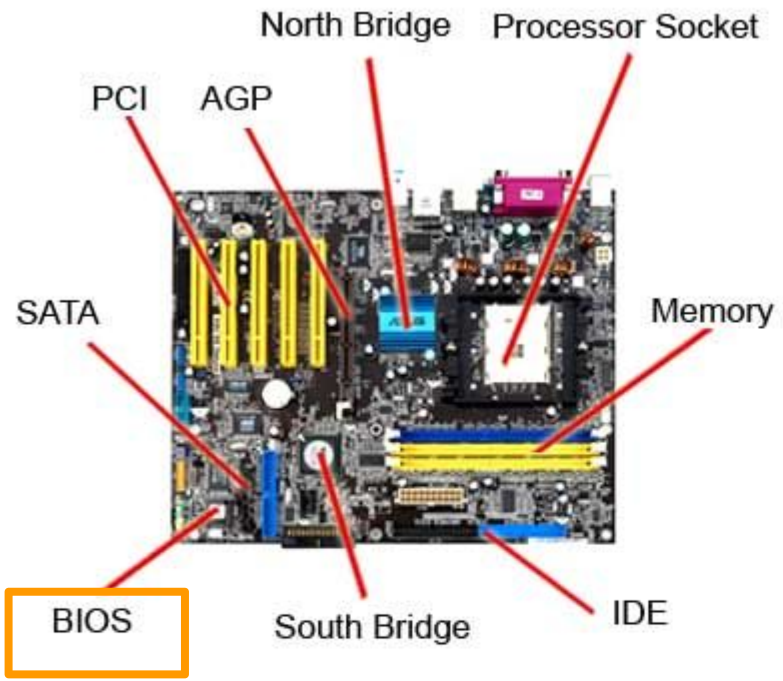
Pop Interview Question

- "What happens after you push the power button on your machine?"
  - (i.e. what happens in software?)
- (True story: Prof Shah had this as an interview question)
  - Understanding operating systems and putting together our hardware knowledge will answer this question!

Boot Process
(Before we get to our Operating System!)

Step 1 | A first program is executed: The BIOS

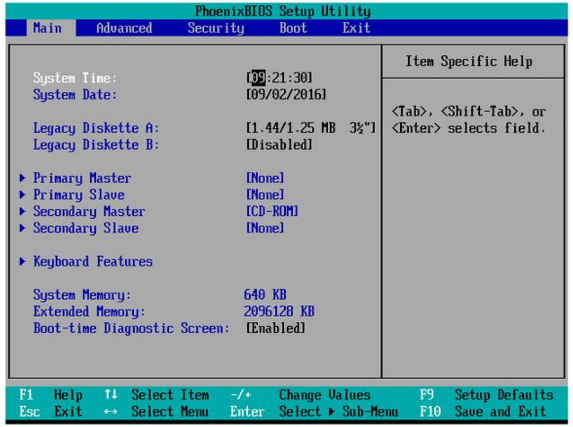
- x86 machines start by executing a program called the BIOS
  - BIOS: Basic Input/Output System
- The BIOS is ‘baked into’ our computers motherboard
  - This means it is stored in non-volatile memory (i.e. memory that persists)
  - (A motherboard is the entirety of the printed circuit you see on the 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’

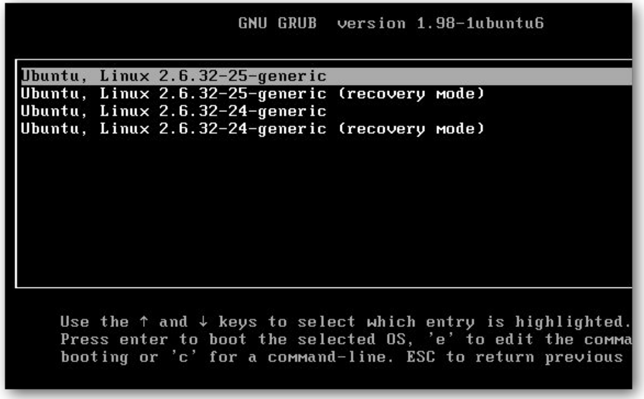


```
                      PhoenixBIOS Setup Utility
  Main     Advanced     Security     Boot     Exit

  System Time:                  [09:21:30]           Item Specific Help
  System Date:                  [09/02/2016]
                                                     <Tab>, <Shift-Tab>, or
  Legacy Diskette A:            [1.44/1.25 MB  3½"]  <Enter> selects field.
  Legacy Diskette B:            [Disabled]

▶ Primary Master                [None]
▶ Primary Slave                 [None]
▶ Secondary Master              [CD-ROM]
▶ Secondary Slave               [None]

▶ Keyboard Features

  System Memory:                640 KB
  Extended Memory:              2096128 KB
  Boot-time Diagnostic Screen:  [Enabled]

F1   Help  ↑↓  Select Item  -/+    Change Values      F9   Setup Defaults
Esc  Exit  ↔   Select Menu  Enter  Select ▶ Sub-Menu  F10  Save and Exit
```

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.

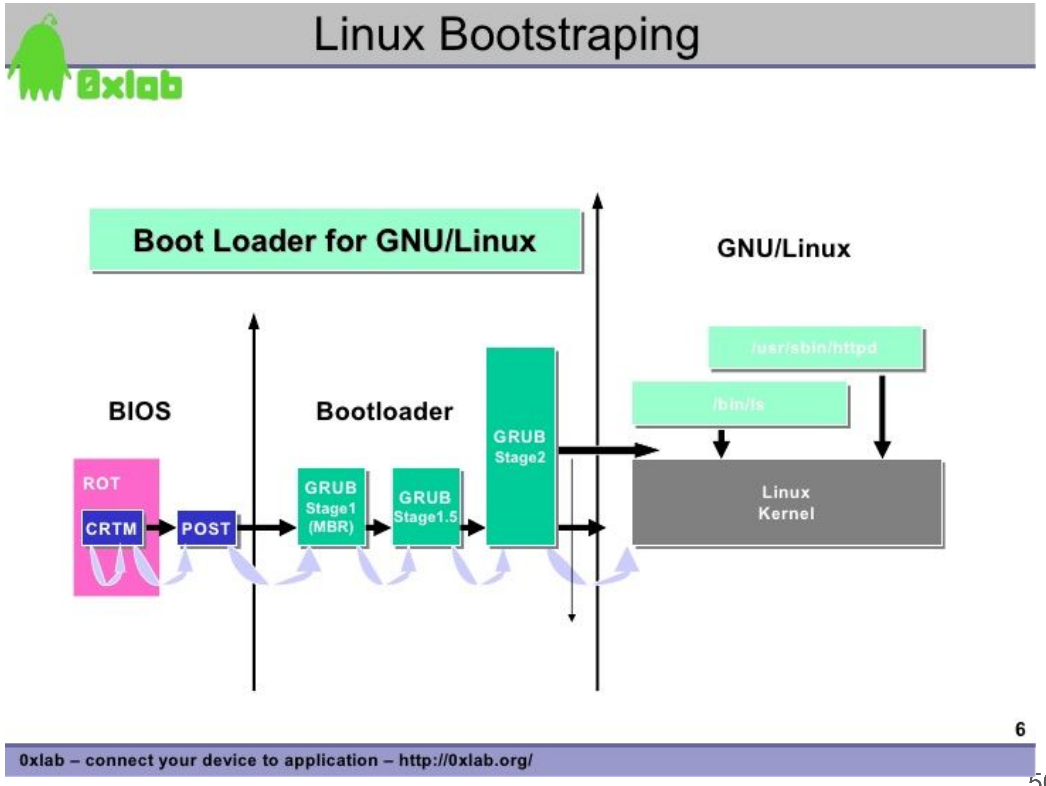


```
                 GNU GRUB  version 1.98-1ubuntu6

Ubuntu, Linux 2.6.32-25-generic
Ubuntu, Linux 2.6.32-25-generic (recovery mode)
Ubuntu, Linux 2.6.32-24-generic
Ubuntu, Linux 2.6.32-24-generic (recovery mode)

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, 'e' to edit the comma
    booting or 'c' for a command-line. ESC to return previous
```

Here is the OS loading process [image source]

Here is the high level abstraction--at the very least the steps to remember

1. BIOS
2. Boot loader
3. Operating System

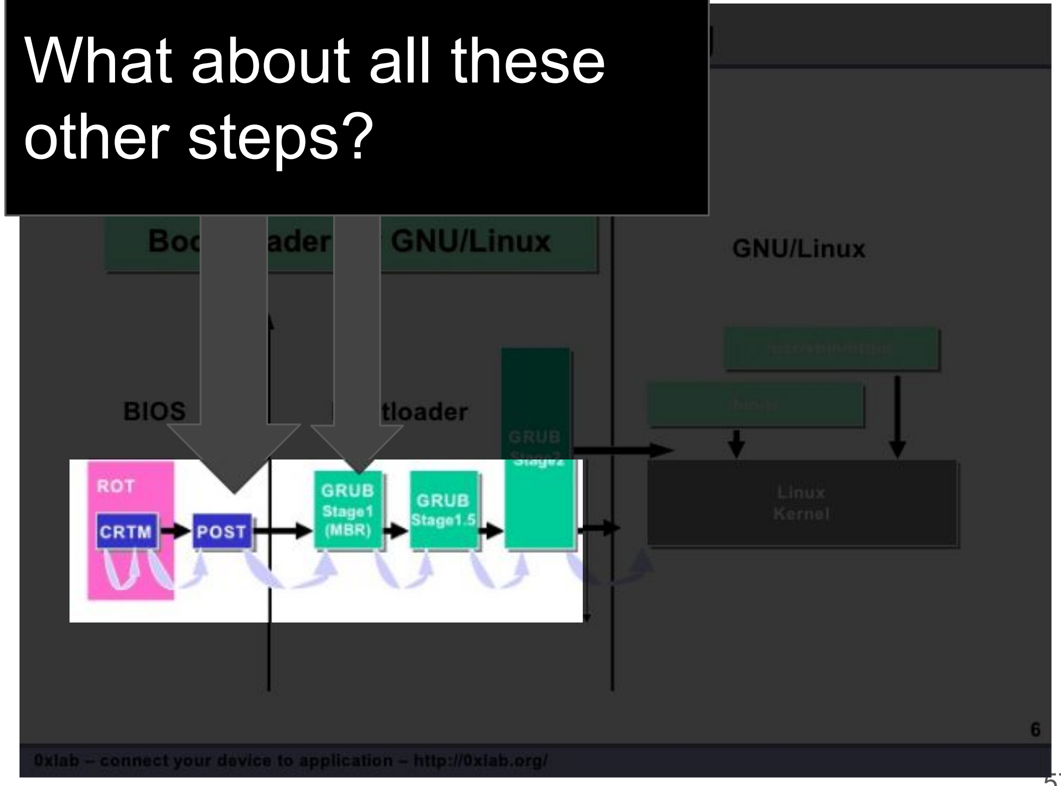


Here is the OS loading process [image source]

Here is the high level abstraction--at the very least the steps to remember

1. BIOS
2. Boot loader
3. Operating System

What about all these other steps?



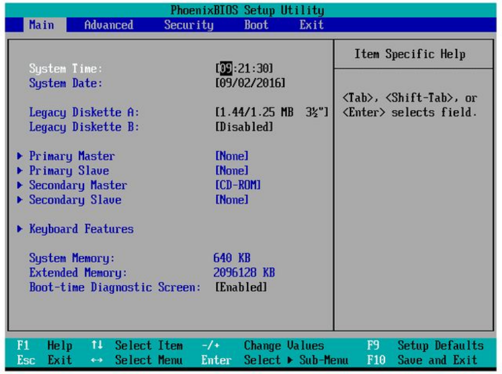
A few more steps

Pushing power

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

Starting the BIOS (1/5)

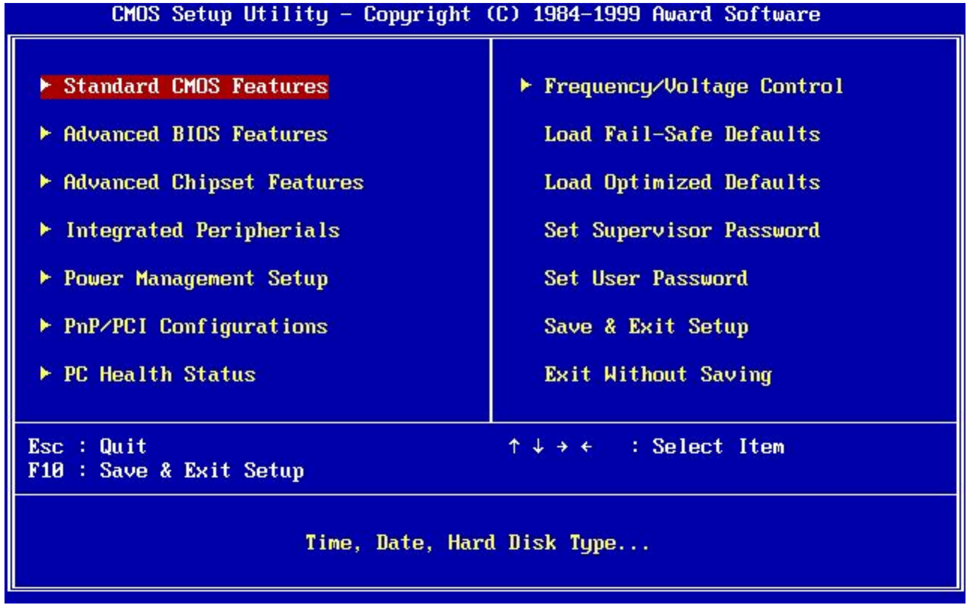
- Basic Input/Output System (BIOS)
  - A mini-OS burned onto a chip
- Begins executing a soon as a PC powers on
  - Code from the BIOS chip gets copied to RAM at a low address (e.g. 0xFF)
  - jmp 0xFF (16 bits) written to RAM at 0xFFFF0 (220-16)
  - x86 CPUs always start with 0xFFFF0 in the EIP register
- 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



```
                         PhoenixBIOS Setup Utility
 Main    Advanced    Security    Boot    Exit

   System Time:                  [09:21:30]          Item Specific Help
   System Date:                  [09/02/2016]
                                                     <Tab>, <Shift-Tab>, or
   Legacy Diskette A:            [1.44/1.25 MB  3½"] <Enter> selects field.
   Legacy Diskette B:            [Disabled]

 ▸ Primary Master                [None]
 ▸ Primary Slave                 [None]
 ▸ Secondary Master              [CD-ROM]
 ▸ Secondary Slave               [None]

 ▸ Keyboard Features

   System Memory:                640 KB
   Extended Memory:              2096128 KB
   Boot-time Diagnostic Screen:  [Enabled]

 F1  Help  ↑↓ Select Item  -/+   Change Values       F9   Setup Defaults
 Esc Exit  ↔  Select Menu  Enter Select ▸ Sub-Menu   F10  Save and Exit
```

Load settings from CMOS (2/5)

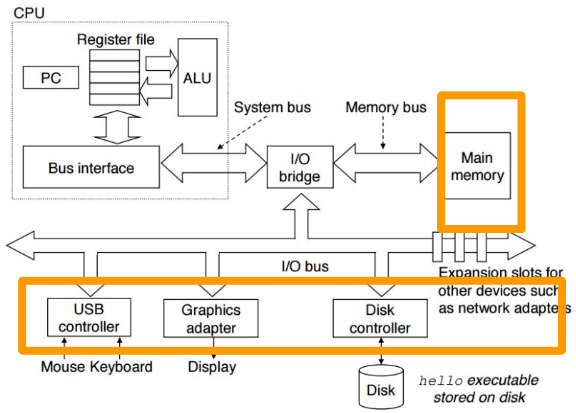
- BIOS often has configurable options
  - Values are stored in a special battery-backed <u>CMOS</u> memory
  - These values are then read in by the BIOS, often containing information about how devices have been configured.



```
CMOS Setup Utility - Copyright (C) 1984-1999 Award Software

▸ Standard CMOS Features          ▸ Frequency/Voltage Control
▸ Advanced BIOS Features            Load Fail-Safe Defaults
▸ Advanced Chipset Features         Load Optimized Defaults
▸ Integrated Peripherials           Set Supervisor Password
▸ Power Management Setup            Set User Password
▸ PnP/PCI Configurations            Save & Exit Setup
▸ PC Health Status                  Exit Without Saving

Esc : Quit                          ↑ ↓ → ←   : Select Item
F10 : Save & Exit Setup

              Time, Date, Hard Disk Type...
```

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)

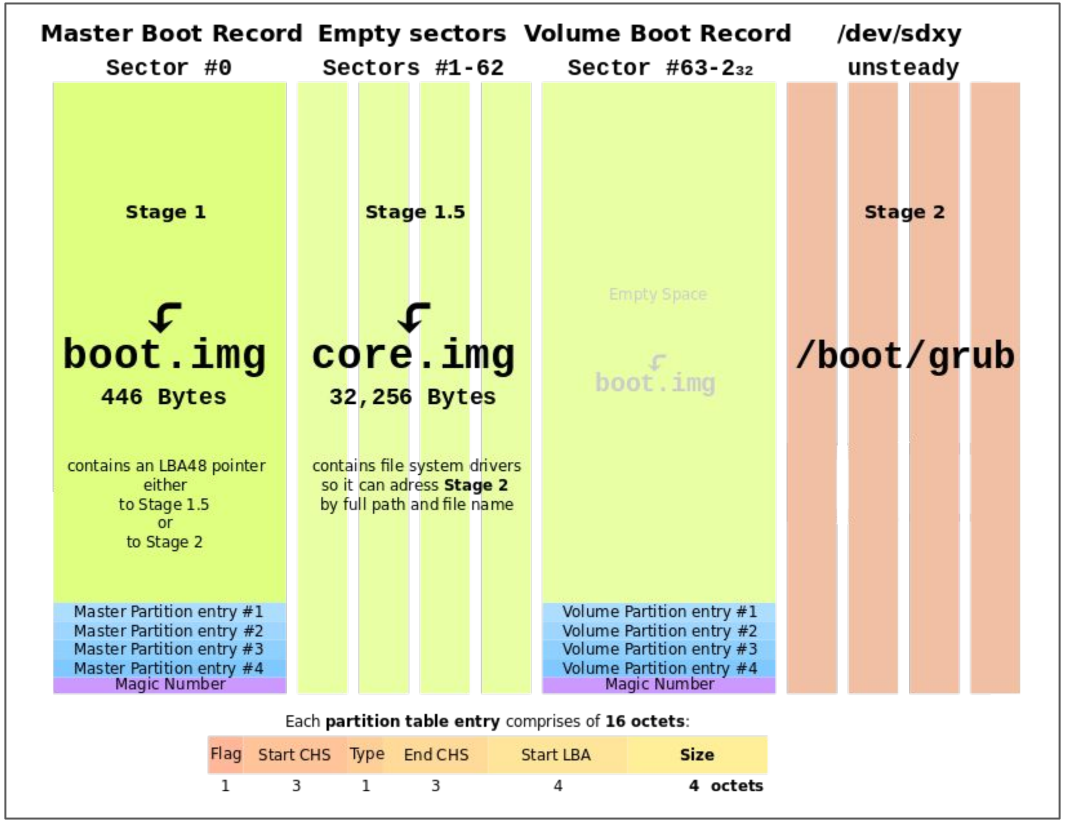
- ***Finally*** we need to find and load a real OS
- BIOS identifies all potentially bootable devices
  - Tries to locate Master Boot Record (MBR) on each device
  - Order in which devices are tried is configurable
- Master Boot Record (MBR) has code that can load the actual OS
  - Code is known as a 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 written to sector 1 (address 0) of a storage device
- Contains 446 bytes of executable code
- Entries for 4 partitions
- Too small to hold an entire OS
  - Starts a sequence of chain-loading

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**

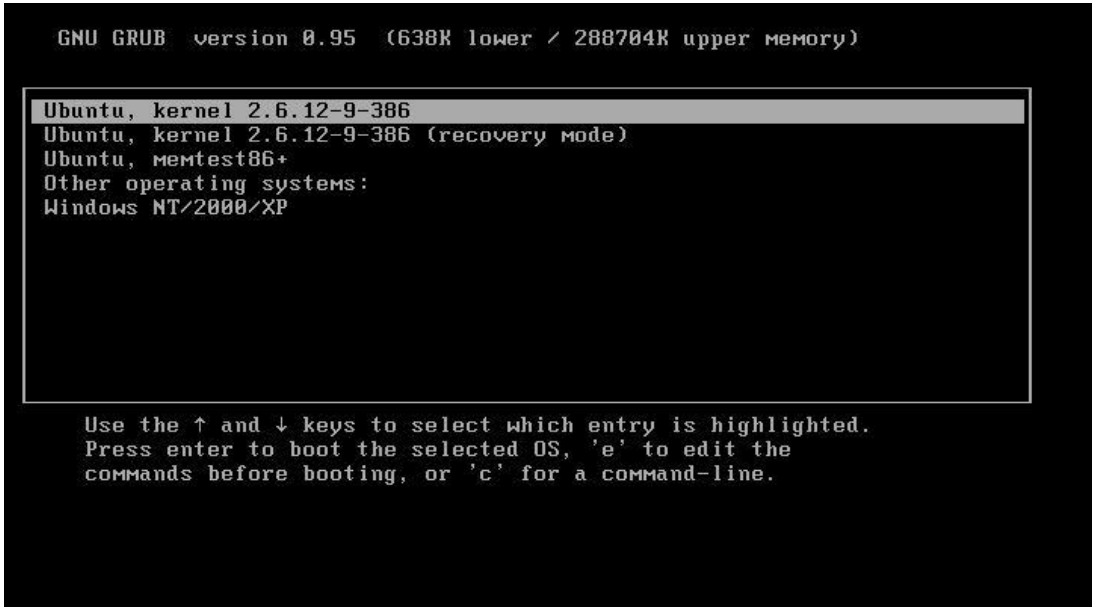
Visualization of Master Boot Record


Master Boot Record
Sector #0
Empty sectors
Sectors #1-62
Volume Boot Record
Sector #63-2³²
/dev/sdxy
unsteady
Stage 1
Stage 1.5
Stage 2
Empty Space
boot.img
446 Bytes
core.img
32,256 Bytes
boot.img
/boot/grub
contains an LBA48 pointer either to Stage 1.5 or to Stage 2
contains file system drivers so it can adress Stage 2 by full path and file name
Master Partition entry #1
Master Partition entry #2
Master Partition entry #3
Master Partition entry #4
Magic Number
Volume Partition entry #1
Volume Partition entry #2
Volume Partition entry #3
Volume Partition entry #4
Magic Number
Each partition table entry comprises of 16 octets:
Flag
Start CHS
Type
End CHS
Start LBA
Size
1
3
1
3
4
4 octets

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

Example Bootloader: GRUB

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



```
  GNU GRUB  version 0.95  (638K lower / 288704K upper memory)

 Ubuntu, kernel 2.6.12-9-386
 Ubuntu, kernel 2.6.12-9-386 (recovery mode)
 Ubuntu, memtest86+
 Other operating systems:
 Windows NT/2000/XP

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, 'e' to edit the
    commands before booting, or 'c' for a command-line.
```

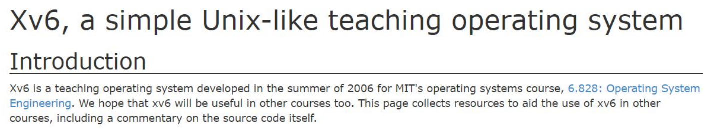
But now lets really see it in action

- 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).



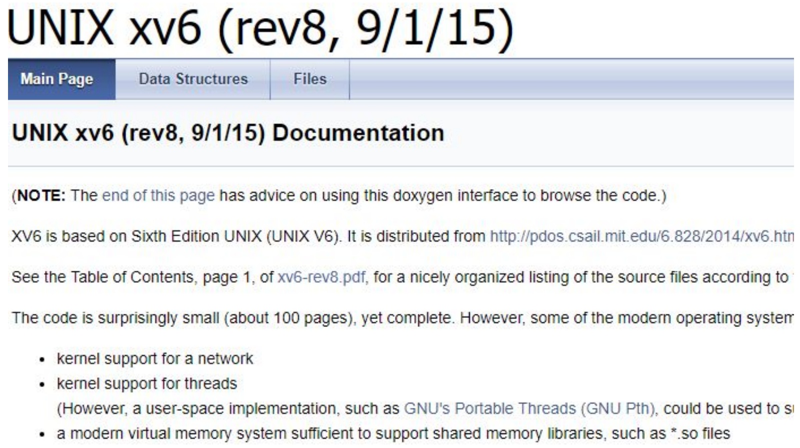
Xv6, a simple Unix-like teaching operating system

Introduction

Xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course, 6.828: Operating System Engineering. We hope that xv6 will be useful in other courses too. This page collects resources to aid the use of xv6 in other courses, including a commentary on the source code itself.

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

- Not something your instructor developed
- But some smart folks at MIT have been working on this for a long while.
  - 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)

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

Boot process in xv6

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
}
```

proc.c

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

```
// 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.
      …
```

(Reminder) Operating System Scheduler

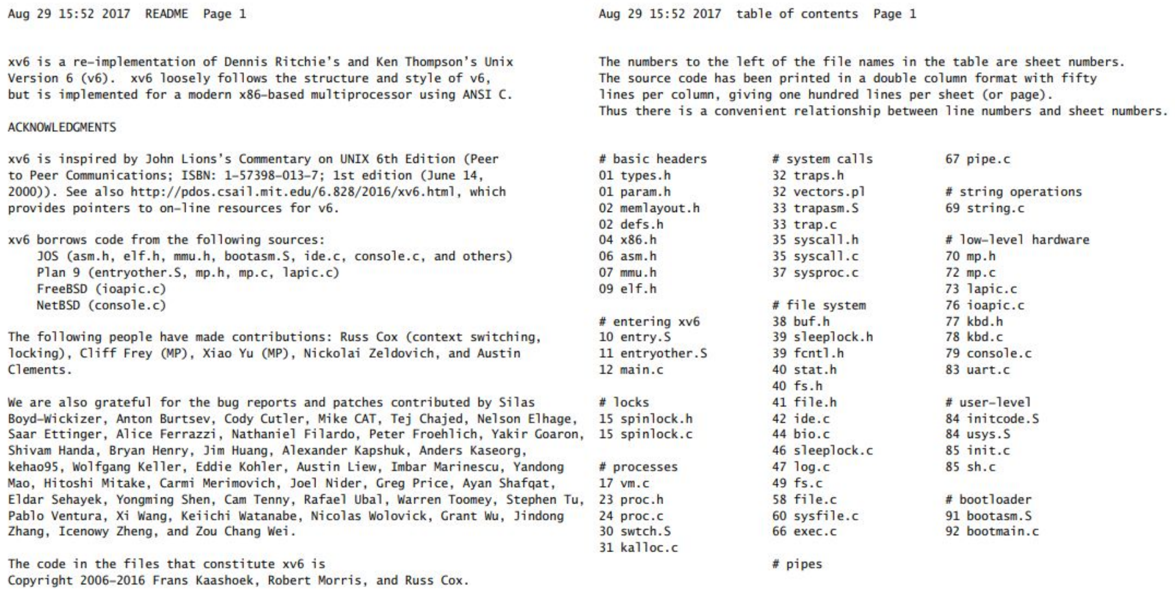
- The scheduler in an Operating system is responsible for picking which process runs.
- The OS gives each process a ‘time slice’ to execute.
- The OS tries to be fair in making sure every process can make some progress
- However, there are some trade-offs
    - Should a long running process using lots 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?

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.
    - (We may jump deeper into this in lab!)

A handy reference card of the OS

- https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf
- Just reading the source file names, some topics should feel familiar
  - proc, trap, file, spinlock, string, syscall, etc.



Aug 29 15:52 2017 README Page 1

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also http://pdos.csail.mit.edu/6.828/2016/xv6.html, which provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

The following people have made contributions: Russ Cox (context switching, locking), Cliff Frey (MP), Xiao Yu (MP), Nickolai Zeldovich, and Austin Clements.

We are also grateful for the bug reports and patches contributed by Silas Boyd-Wickizer, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, Nelson Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter Froehlich, Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Alexander Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin Liew, Imbar Marinescu, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny, Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas Wolovick, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2016 Frans Kaashoek, Robert Morris, and Russ Cox.

Aug 29 15:52 2017 table of contents Page 1

The numbers to the left of the file names in the table are sheet numbers. The source code has been printed in a double column format with fifty lines per column, giving one hundred lines per sheet (or page). Thus there is a convenient relationship between line numbers and sheet numbers.

So far...

- We know what an operating system is
- We know when we press the power button
- An OS needs to be bootstrapped from the BIOS
- We have seen a quick example of this in xv6
- (As well as identified a few important files in xv6)
- Now let's figure out how to get xv6 running
- Friday’s lab will address this…