

Chapter 1

Introduction



“UNIX is simple. It just takes a genius to understand its simplicity.”

— Dennis Ritchie



Dennis Ritchie

Overview

- ❖ What is an operating system, anyway?
- ❖ Operating systems history
- ❖ The zoo of modern operating systems
- ❖ Review of computer hardware
- ❖ Operating system concepts
- ❖ Operating system structure
 - User interface to the operating system
 - Anatomy of a system call
 - Monolithic vs. microkernel vs. virtual machine

What is an operating system?

- ❖ It's a program that provides useful abstractions of hardware to applications
 - Acts as an intermediary between computer and users
 - Standardizes the interface to the user across different types of hardware
 - Hides the messy details which must be performed
 - Presents user with a virtual machine, easier to use
- ❖ It's a resource manager
 - Each program gets time with the resource
 - Each program gets space on the resource
- ❖ May have potentially conflicting goals:
 - Use hardware efficiently
 - Give maximum performance to each user

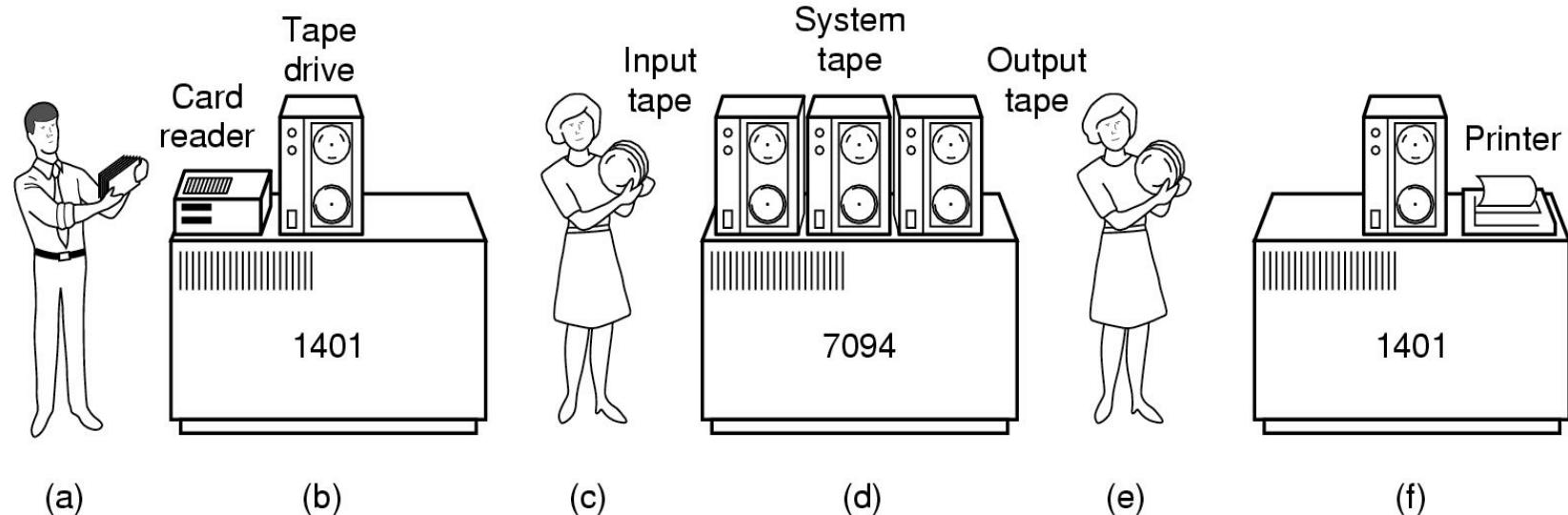
Operating system timeline

- ❖ First generation: 1945 – 1955
 - Vacuum tubes
 - Plug boards
- ❖ Second generation: 1955 – 1965
 - Transistors
 - Batch systems
- ❖ Third generation: 1965 – 1980
 - Integrated circuits
 - Multiprogramming
- ❖ Fourth generation: 1980 – present
 - Large scale integration
 - Personal computers
- ❖ Fifth generation: 1995–???
 - Mobile devices

First generation: direct input

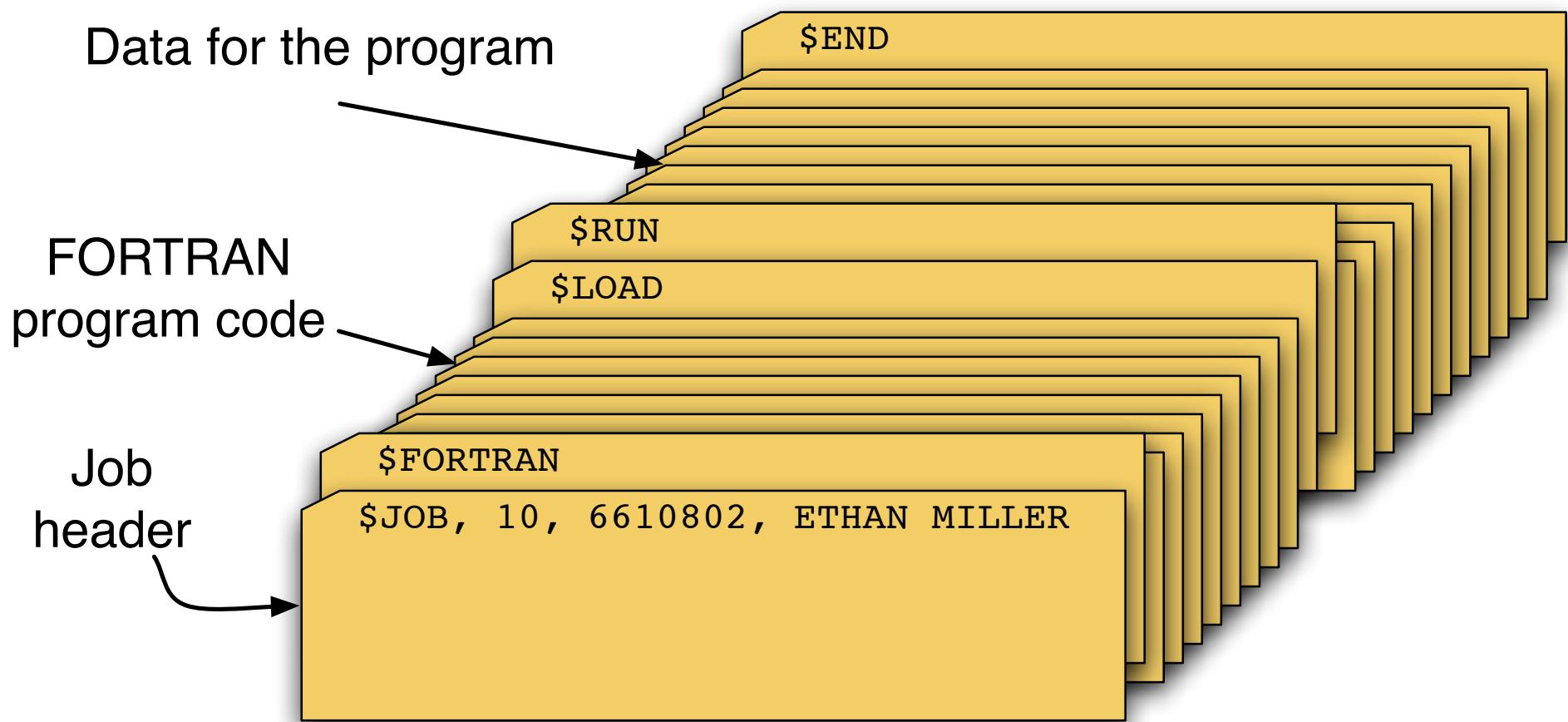
- ❖ Run one job at a time
 - Enter it into the computer (might require rewiring!)
 - Run it
 - Record the results
- ❖ Problem: lots of wasted computer time!
 - Computer was idle during first and last steps
 - Computers were very expensive!
- ❖ Goal: make better use of an expensive commodity: computer time

Second generation: batch systems



- ❖ Bring cards to 1401
- ❖ Read cards onto input tape
- ❖ Put input tape on 7094
- ❖ Perform the computation, writing results to output tape
- ❖ Put output tape on 1401, which prints output

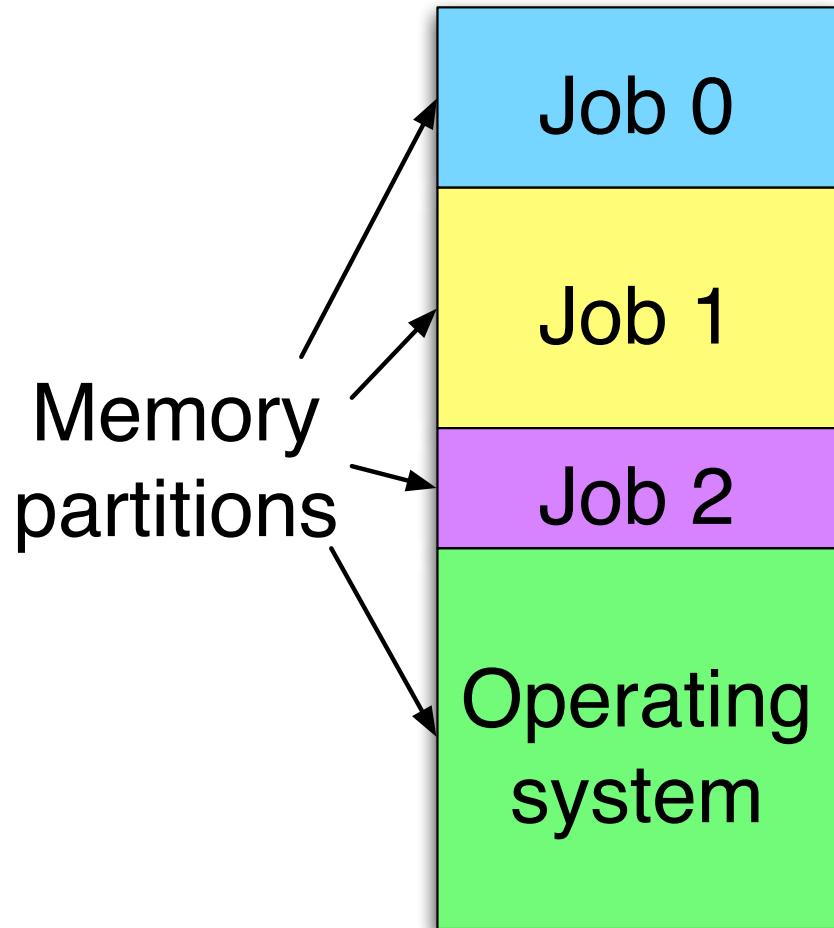
Structure of a typical 2nd generation job



Spooling

- ❖ Original batch systems used tape drives
- ❖ Later batch systems used disks for buffering
 - Operator read cards onto disk attached to the computer
 - Computer read jobs from disk
 - Computer wrote job results to disk
 - Operator directed that job results be printed from disk
- ❖ Disks enabled simultaneous peripheral operation on-line (spooling)
 - Computer overlapped I/O of one job with execution of another
 - Better utilization of the expensive CPU
 - Still only one job active at any given time

Third generation: multiprogramming



- ❖ Multiple jobs in memory
 - Protected from one another
- ❖ OS protected from each job as well
- ❖ Resources (hardware, time) split between jobs
- ❖ Still not interactive
 - User submits job
 - Computer runs it
 - User gets results minutes (hours, days) later

Timesharing

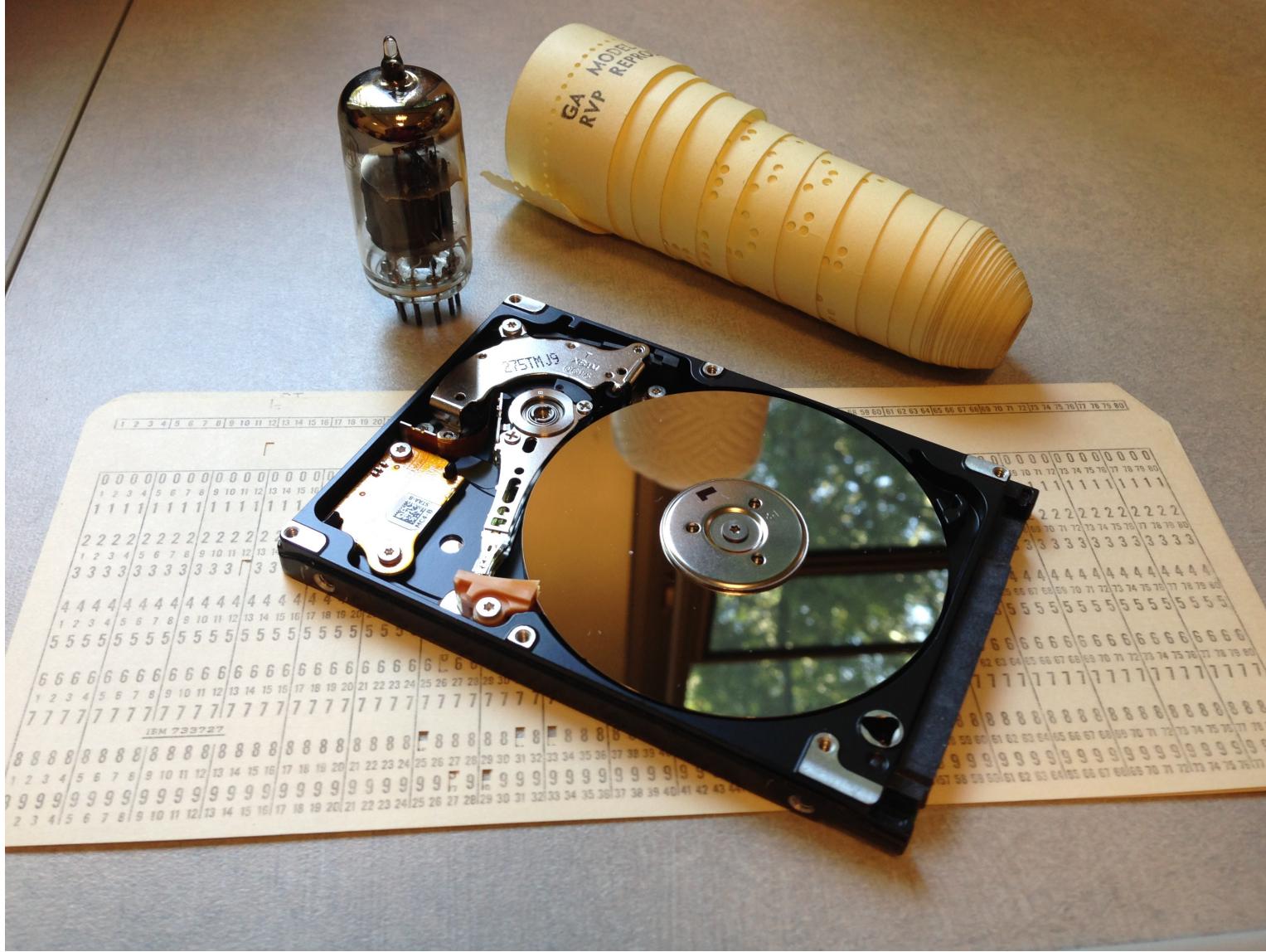
- ❖ Multiprogramming allowed several jobs to be active at one time
 - Initially used for batch systems
 - Cheaper hardware terminals ⇒ interactive use

- ❖ Computer use got much cheaper and easier
 - No more “priesthood”
 - Quick turnaround meant quick fixes for problems

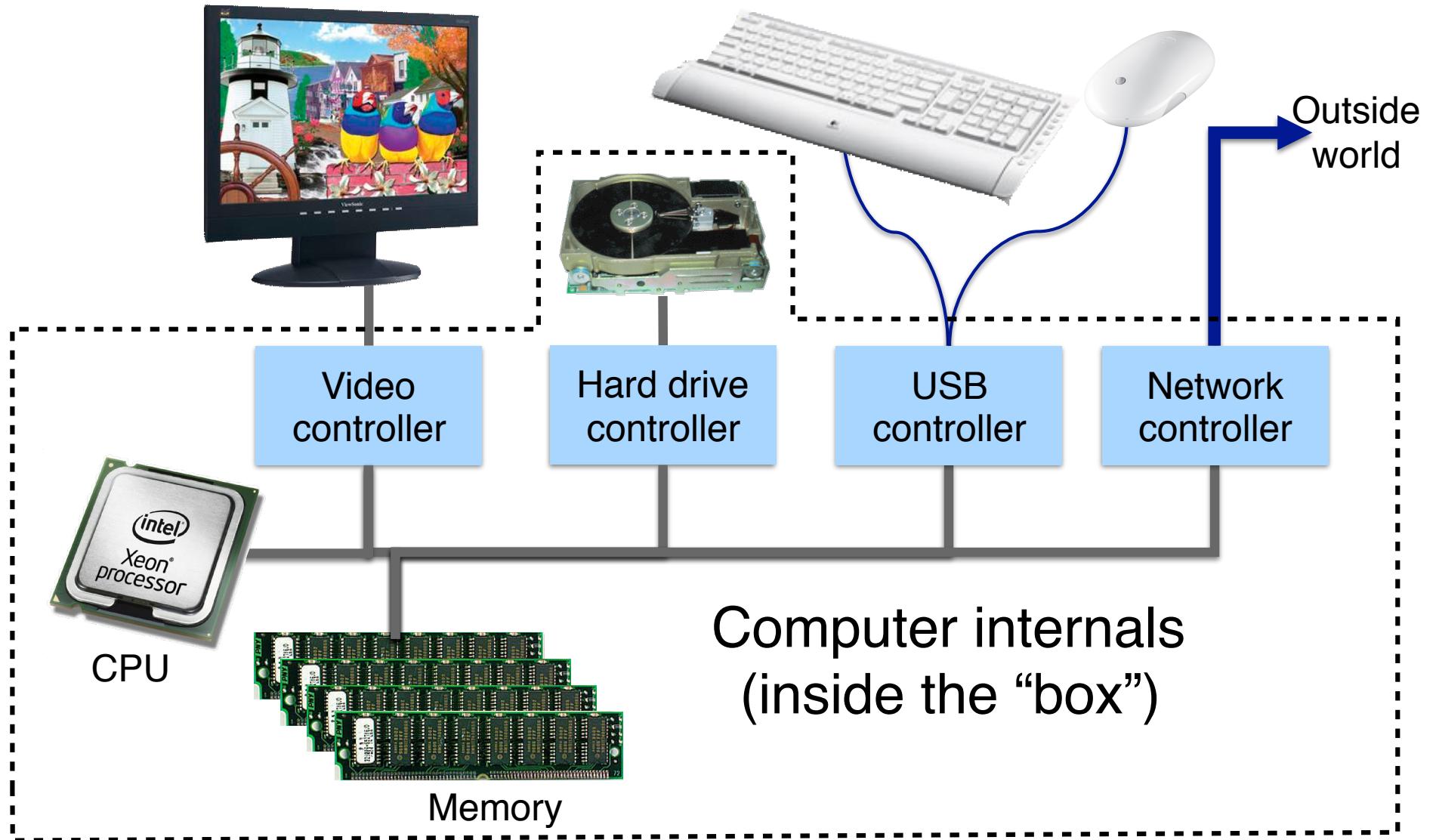
Types of modern operating systems

- ❖ Mainframe operating systems: MVS
- ❖ Server operating systems: FreeBSD, Linux, Solaris
 - Multiprocessor operating systems
- ❖ Personal computer operating systems: MacOS X, Windows 10, Linux (?)
- ❖ Handheld device operating systems: Android (Linux), iOS (OS X), Windows
- ❖ Real-time operating systems: VxWorks, QNX
 - Embedded operating systems
 - Sensor network nodes
- ❖ Smart card operating systems
- ❖ *Some operating systems can fit into more than one category*

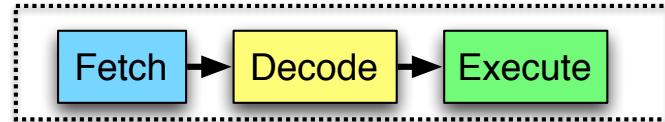
Technology Evolves



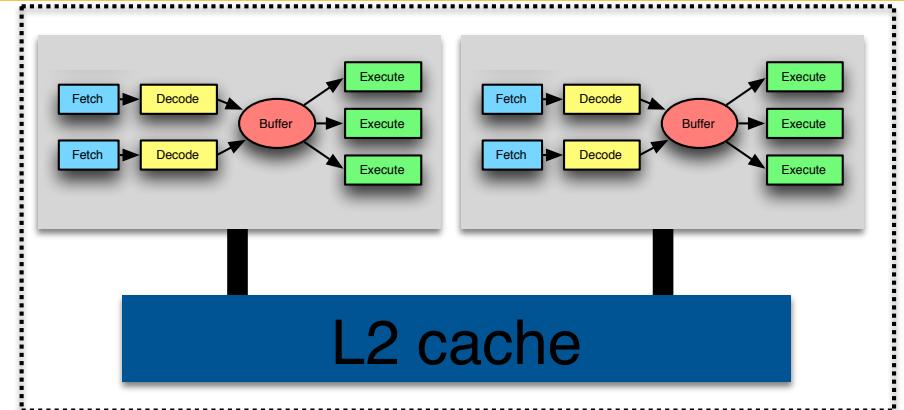
Components of a simple PC



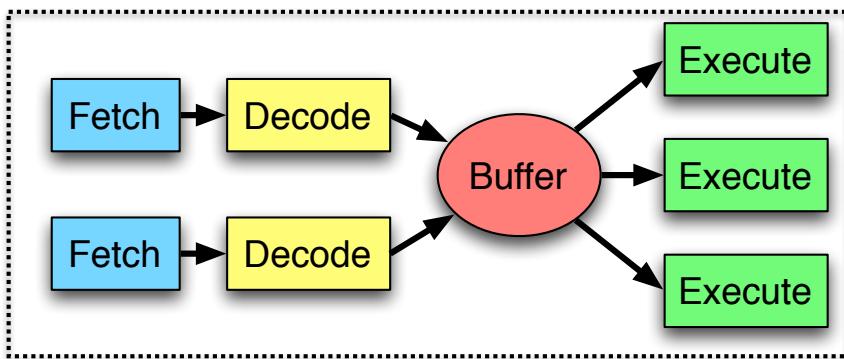
Multicore CPUs



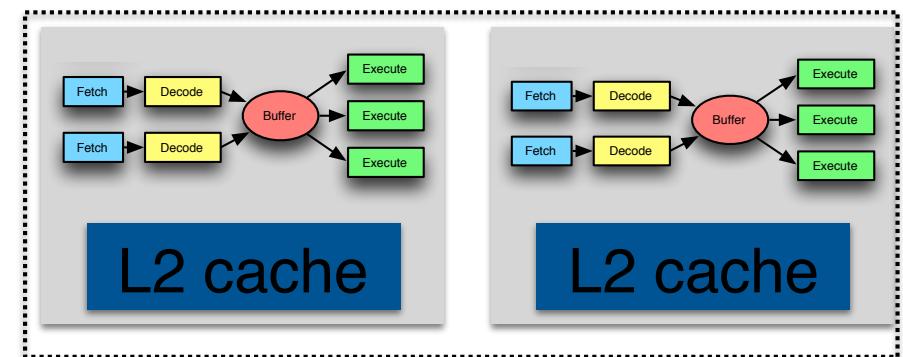
Pipelined CPU



Dual-core CPU
(shared L2 cache)

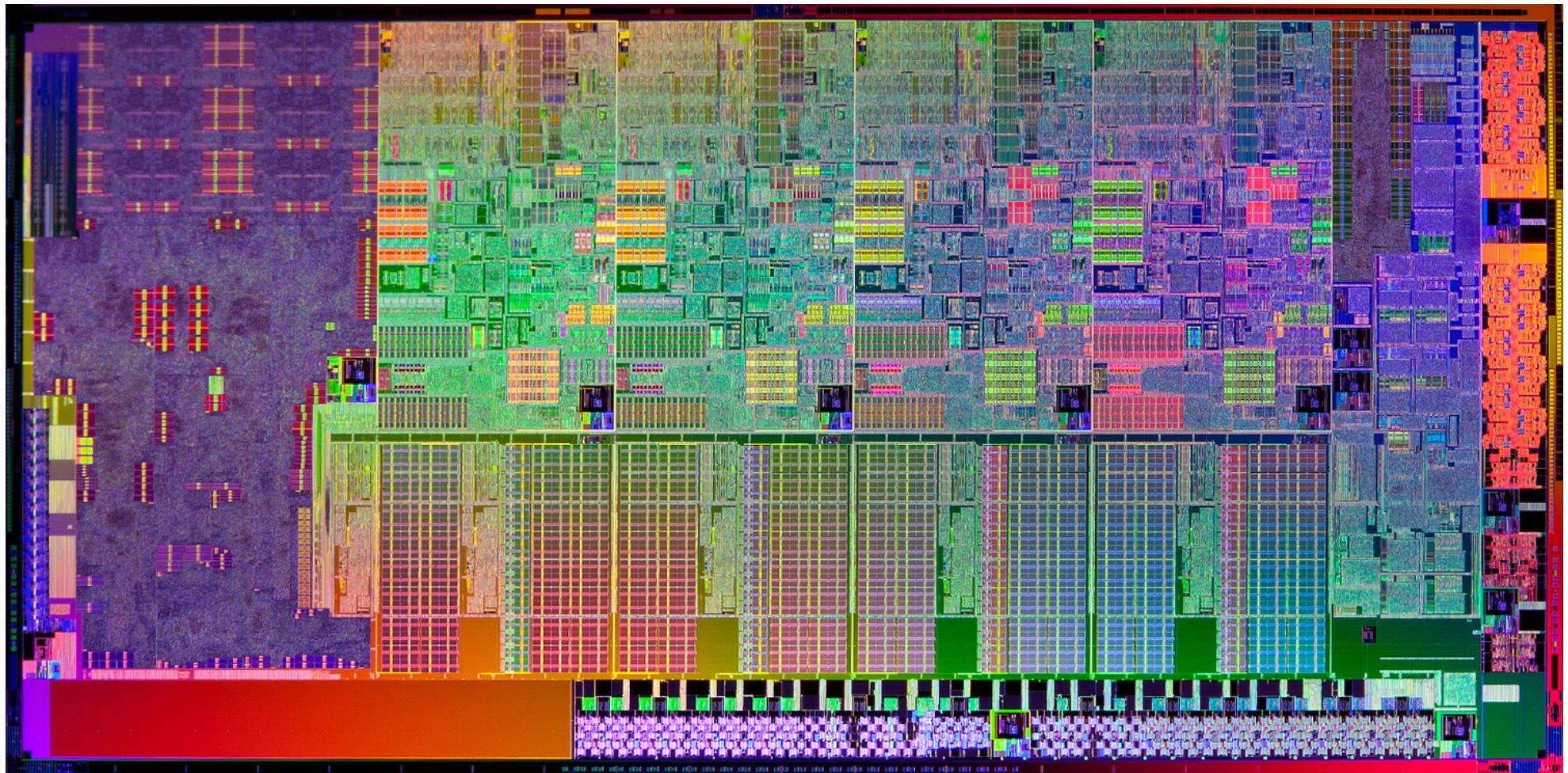


Superscalar CPU

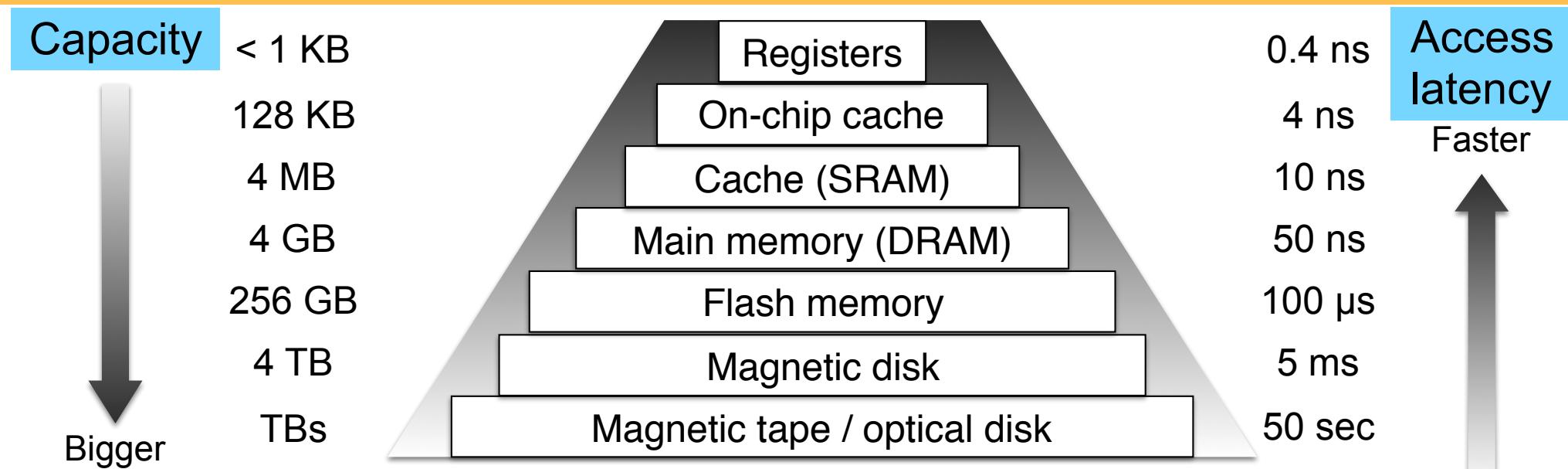


Dual-core CPU
(separate L2 caches)

Multicore Processor Die



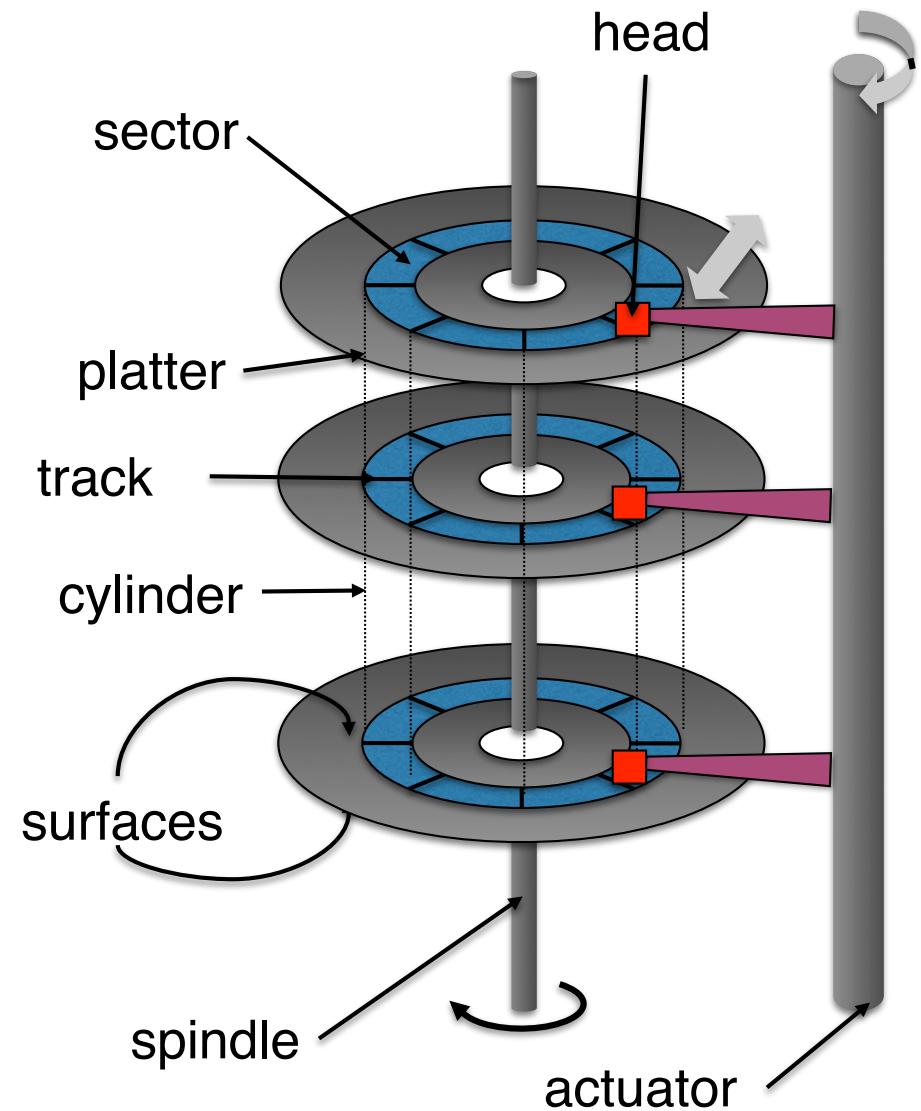
Storage pyramid



- ❖ Goal: really large memory with very low latency
 - Latencies are smaller at the top of the hierarchy
 - Capacities are larger at the bottom of the hierarchy
- ❖ Solution: move data between levels to create illusion of large memory with low latency
 - Some movement is done in hardware (cache ↔ registers)
 - Most is coordinated by software, typically the OS

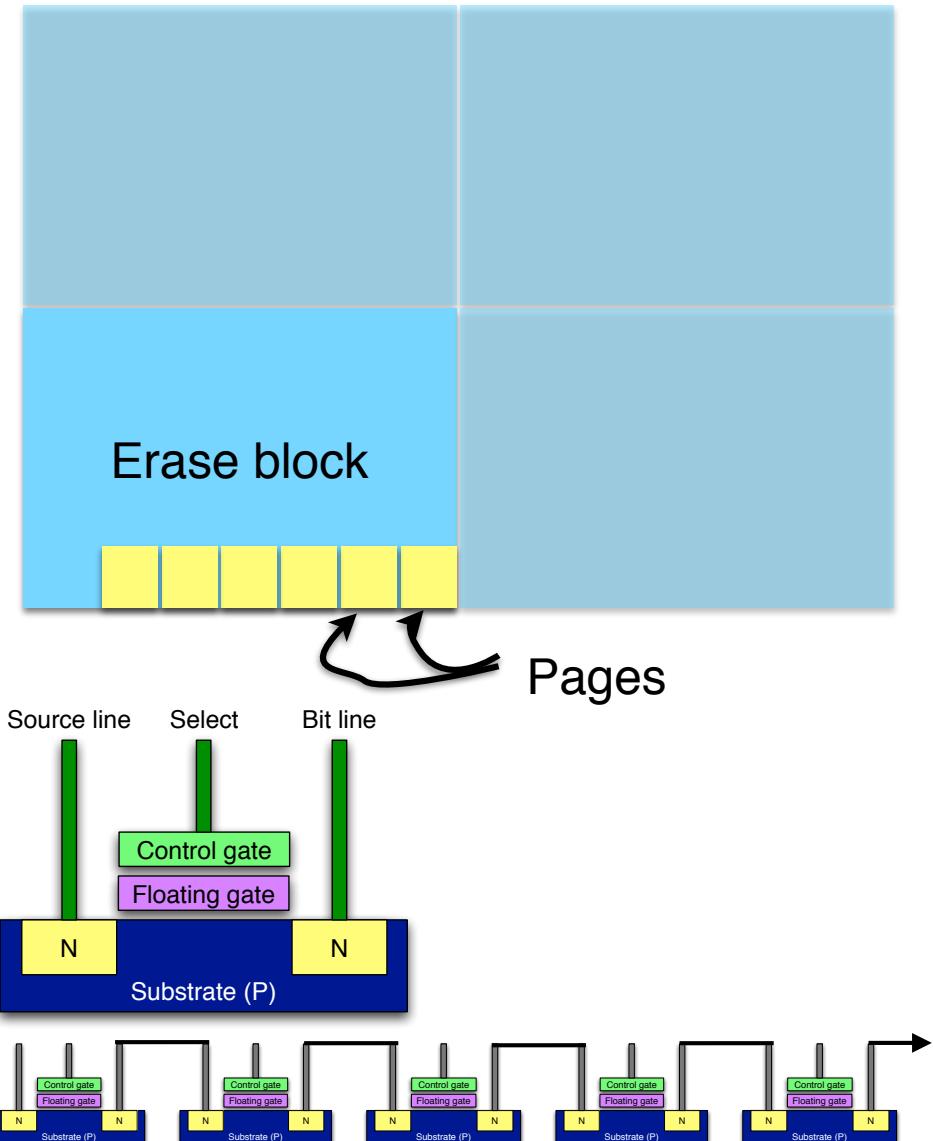
Disk drive structure

- ❖ Data stored on surfaces
 - Up to two surfaces per platter
 - One or more platters per disk
- ❖ Data in concentric tracks
 - Tracks broken into sectors
 - 256B–1KB per sector
 - Cylinder: corresponding tracks on all surfaces
- ❖ Data read and written by heads
 - Actuator moves heads
 - Heads move in unison

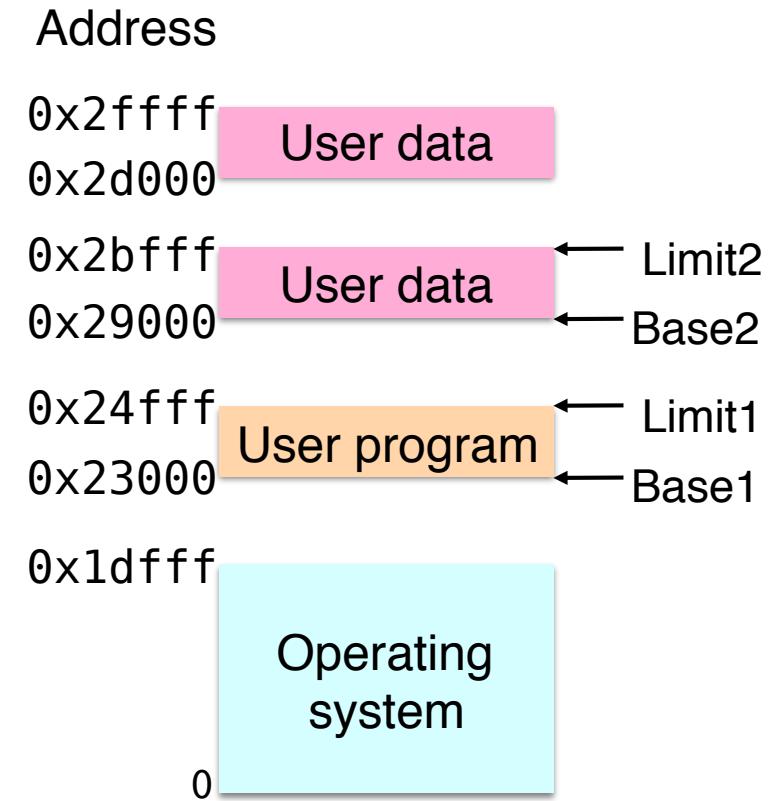
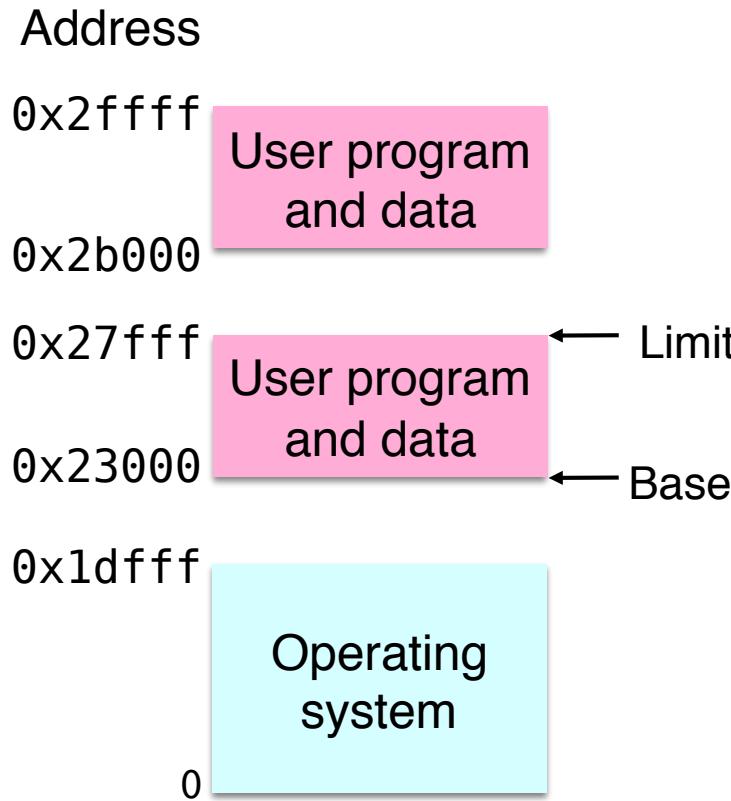


Flash memory structure

- ❖ Flash is divided into erase blocks
 - Blocks must be erased before they can be written
 - Erase blocks are typically 256KB–1MB or larger
- ❖ Flash is read and written in pages
 - Typically 64–256 pages per erase block
- ❖ Flash Translation Layer (FTL) manages the device
 - Necessary to make it look like a disk
- ❖ More details when we cover I/O

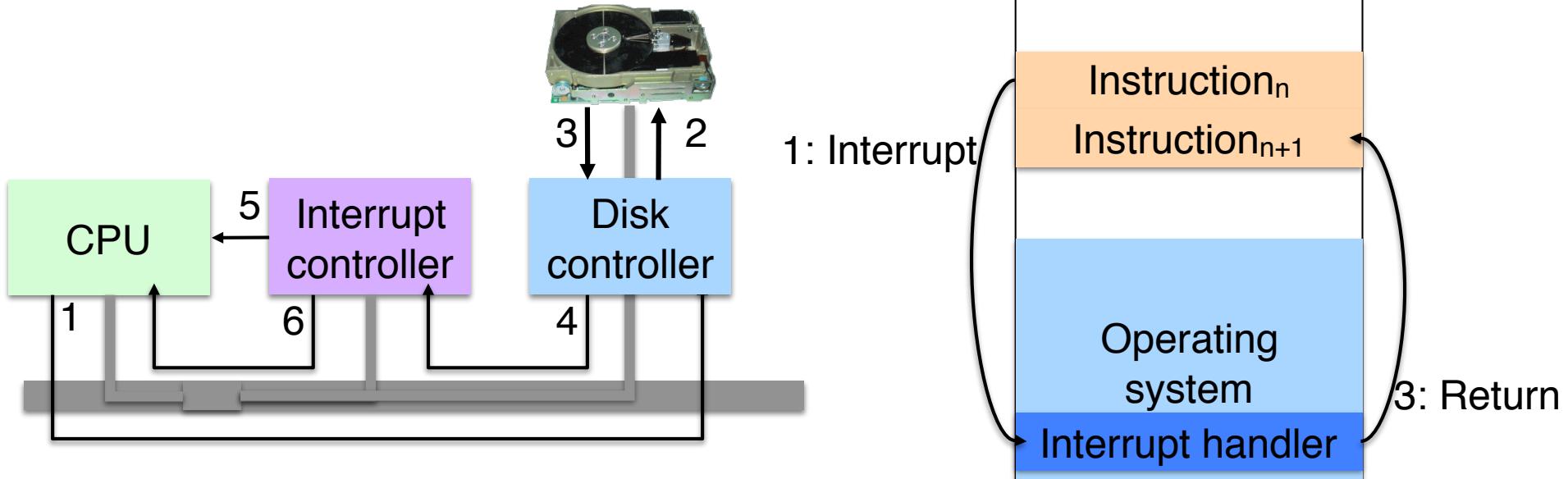


Memory



- ❖ Single base/limit pair: set for each process
- ❖ Two base/limit registers: one for program, one for data

Anatomy of a device request

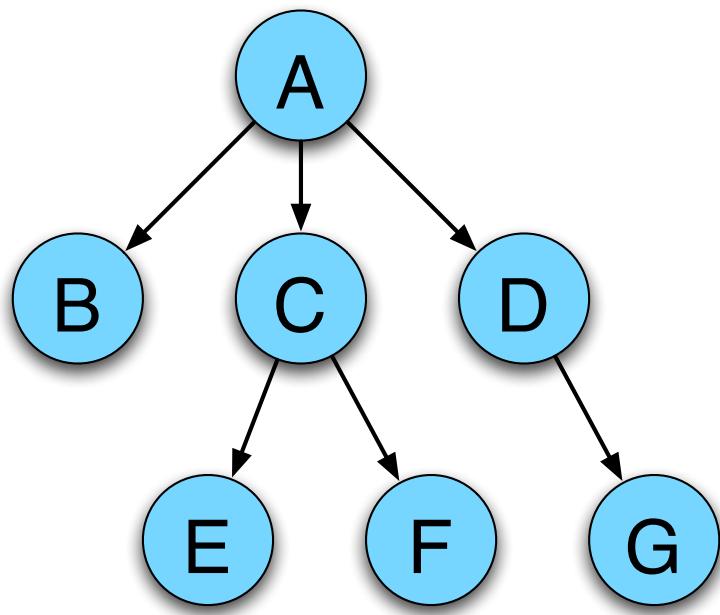


- ❖ Left: sequence as seen by hardware
 - Request sent to controller, then to disk
 - Disk responds, signals disk controller which tells interrupt controller
 - Interrupt controller notifies CPU
- ❖ Right: interrupt handling (software point of view)

Operating systems concepts

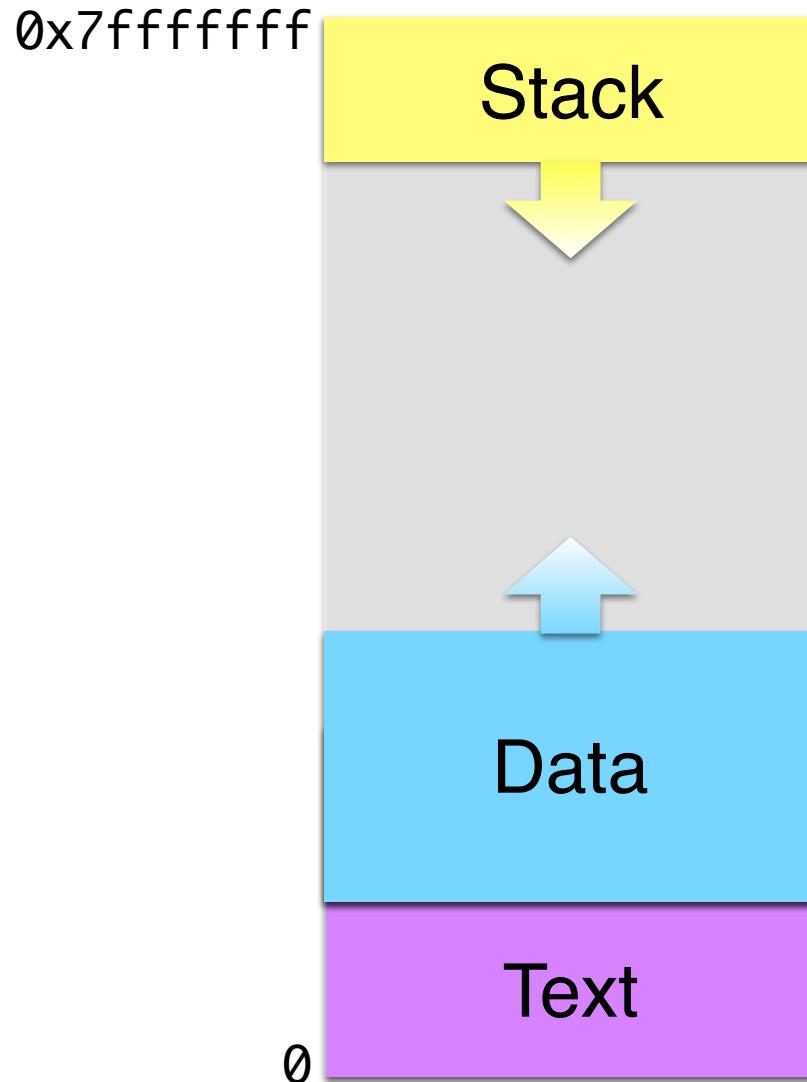
- ❖ Many of these should be familiar to UNIX users...
- ❖ Processes (and trees of processes)
- ❖ Deadlock
- ❖ File systems & directory trees
- ❖ Pipes
- ❖ We'll cover all of these in more depth later on, but it's useful to have some basic definitions now

Processes



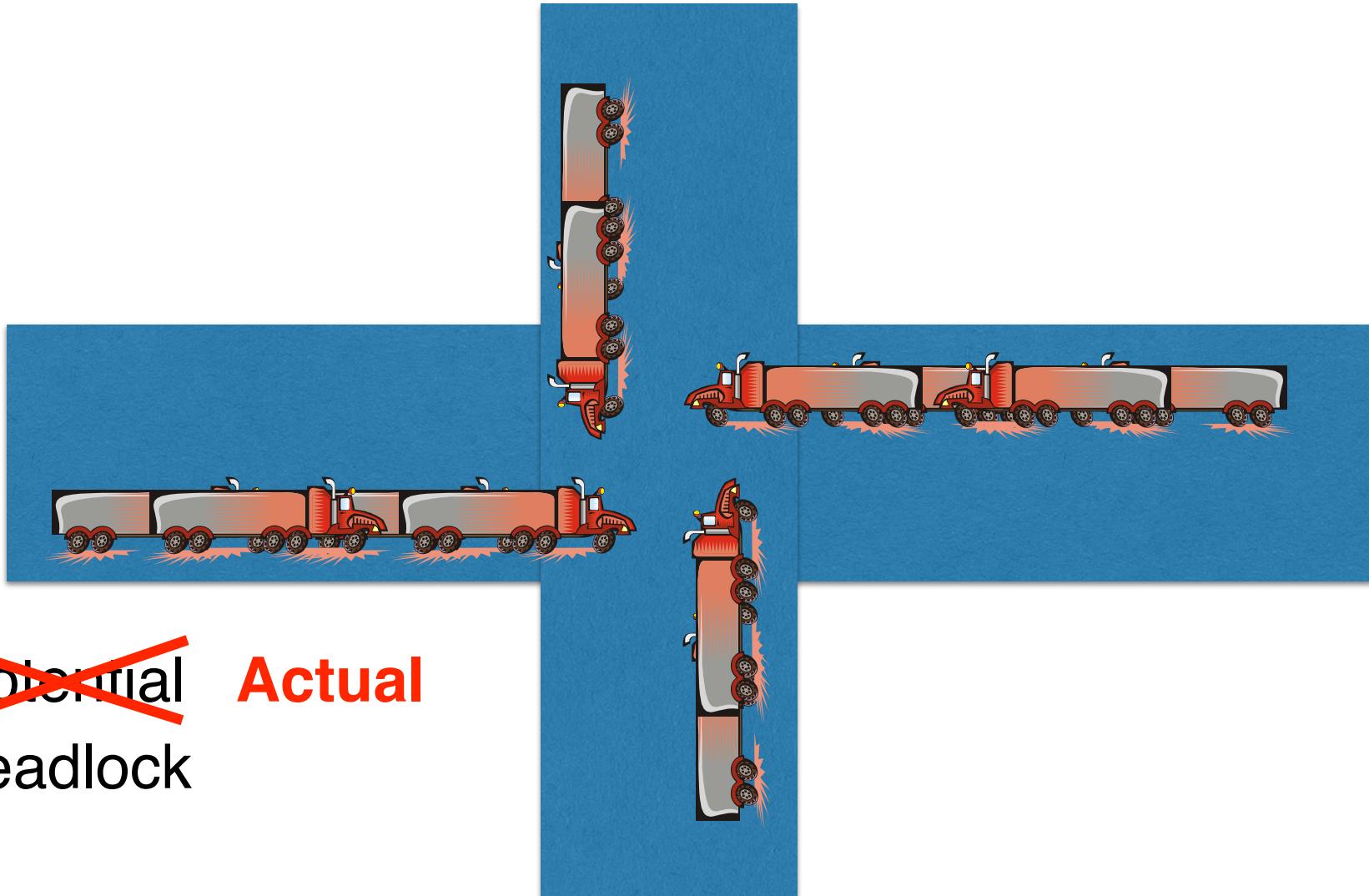
- ❖ Process: program in execution
 - Address space (memory) the program can use
 - State (registers, including program counter & stack pointer)
- ❖ OS keeps track of all processes in a process table
- ❖ Processes can create other processes
 - Process tree tracks these relationships
 - A is the root of the tree
 - A created three child processes: B, C, and D
 - C created two child processes: E and F
 - D created one child process: G

Inside a (UNIX) process

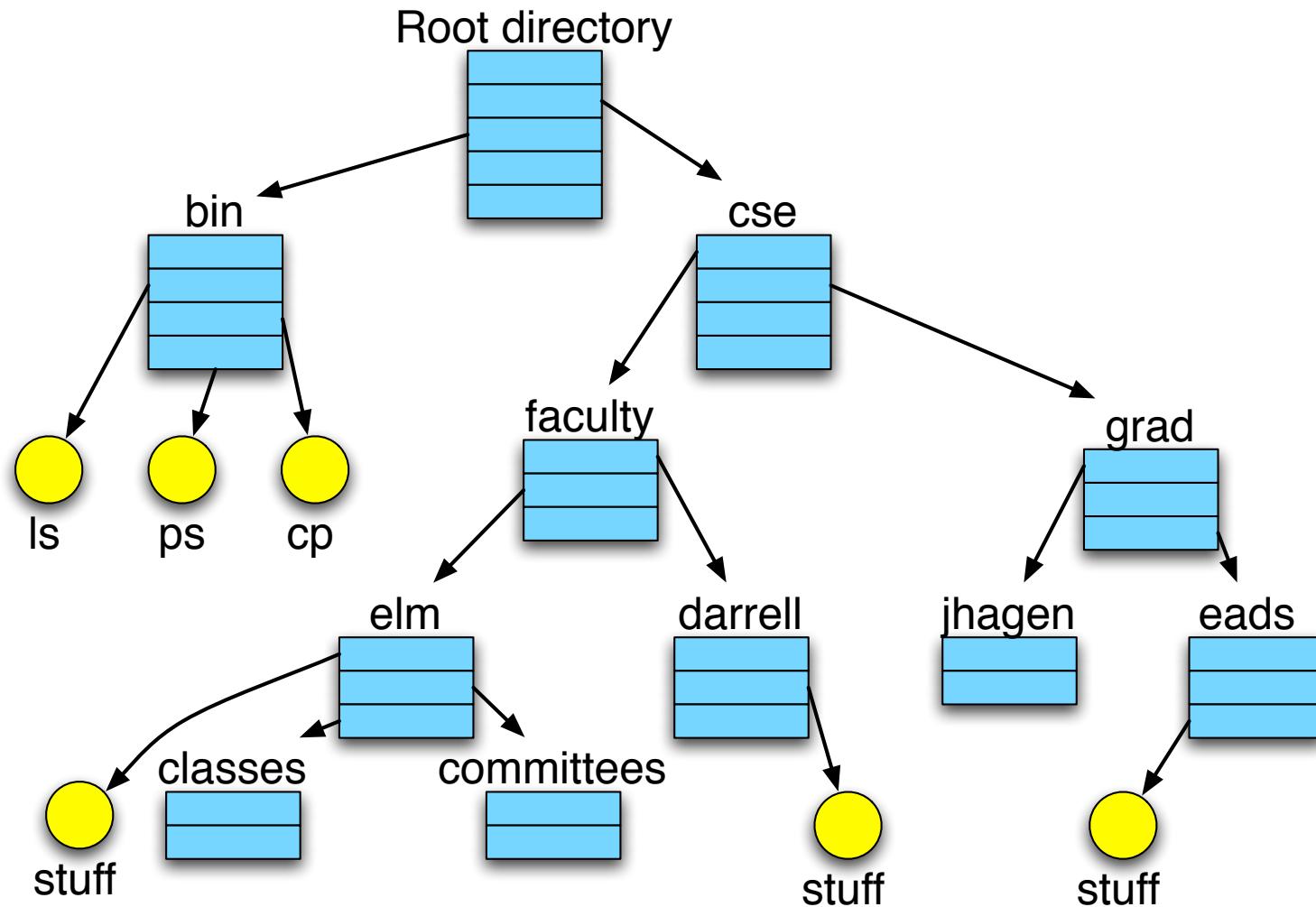


- ❖ Processes have three segments
 - Text: program code
 - Data: program data
 - Statically declared variables
 - Areas allocated by malloc() or new
 - Stack
 - Automatic variables
 - Procedure call information
- ❖ Address space growth
 - Text: doesn't grow
 - Data: grows "up"
 - Stack: grows "down"

Deadlock



Hierarchical file systems



Interprocess communication

- ❖ Processes want to exchange information with each other
- ❖ Many ways to do this, including
 - Network
 - Pipe (special file): A writes into pipe, and B reads from it



System calls

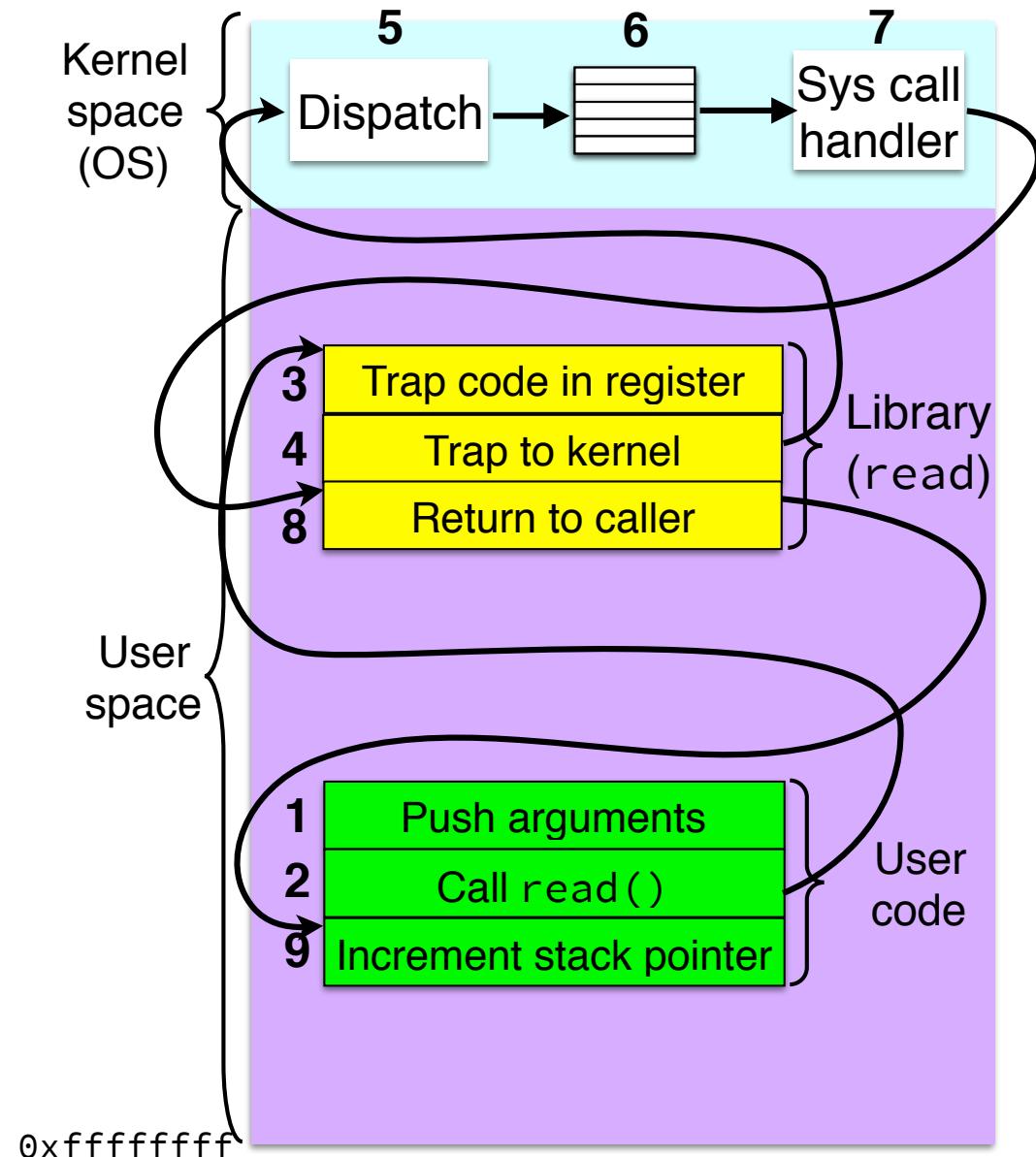
- ❖ OS runs in privileged mode
 - Some operations are permitted only in privileged (also called supervisor or system) mode
 - Example: access a device like a disk or network card
 - Example: change allocation of memory to processes
 - User programs run in user mode and can't do the operations
- ❖ Programs want the OS to perform a service
 - Access a file
 - Create a process
 - Others...
- ❖ Accomplished by system call

How system calls work

- ❖ User program enters supervisor mode
 - Must enter via well-defined entry point
- ❖ Program passes relevant information to OS
- ❖ OS performs the service if
 - The OS is able to do so
 - The service is permitted for this program at this time
- ❖ OS checks information passed to make sure it's OK
 - Don't want programs reading data into other programs' memory!
- ❖ OS needs to be paranoid!
 - Users do the darnedest things....

Making a system call

- ❖ System call:
`read(fd, buffer, length)`
- ❖ Program pushes arguments,
calls library
- ❖ Library sets up trap, calls OS
- ❖ OS handles system call
- ❖ Control returns to library
- ❖ Library returns to user program



System calls for files & directories

Assignment 1 uses these!

Call	Description
<code>fd = open(name, how)</code>	Open a file for reading and/or writing
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, size)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, size)</code>	Write data from a buffer into a file
<code>s = lseek(fd, offset, whence)</code>	Move the “current” pointer for a file
<code>s = stat(name, &buffer)</code>	Get a file’s status information (in buffer)
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove a directory (must be empty)
<code>s = link(name1, name2)</code>	Create a new entry (<i>name2</i>) that points to the same object as <i>name1</i>
<code>s = unlink(name)</code>	Remove name as a link to an object (deletes the object if name was the only link to it)

More system calls

Call	Description
pid = fork()	Create a child process identical to the parent
pid=waitpid(pid,&statloc,options)	Wait for a child to terminate
s = execve(name,argv,environp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = chdir(dirname)	Change the working directory
s = chmod(name,mode)	Change a file's protection bits
s = kill(pid,signal)	Send a signal to a process
seconds = time(&seconds)	Get the current time

A simple shell

```
while (TRUE) {                                /* repeat forever */
    print_prompt( );                          /* display prompt */
    read_command (command, parameters) /* input from terminal */

    if (fork() != 0) {                        /* fork off child process */
        /* Parent code */
        waitpid( -1, &status, 0);           /* wait for child to exit */
    } else {
        /* Child code */
        execve (command, parameters, 0); /* execute command */
    }
}
```

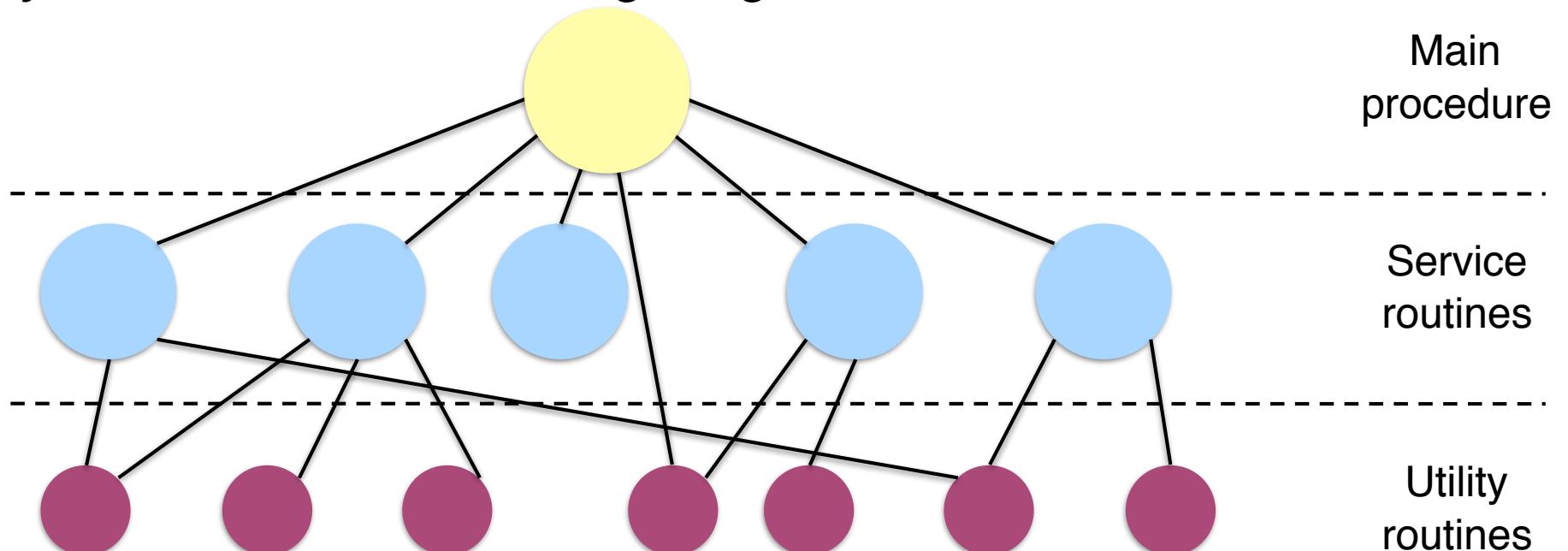
- ❖ Shell runs forever, accepting user commands
- ❖ You'll be writing a (slightly) more complex shell in Assignment 1

Operating system structure

- ❖ OS is composed of lots of pieces
 - Memory management
 - Process management
 - Device drivers
 - File system
- ❖ How do the pieces of the operating system fit together and communicate with each other?
- ❖ Different ways to structure an operating system
 - Monolithic
 - Modular is similar, but more extensible
 - Layered
 - Microkernel
 - Virtual machines

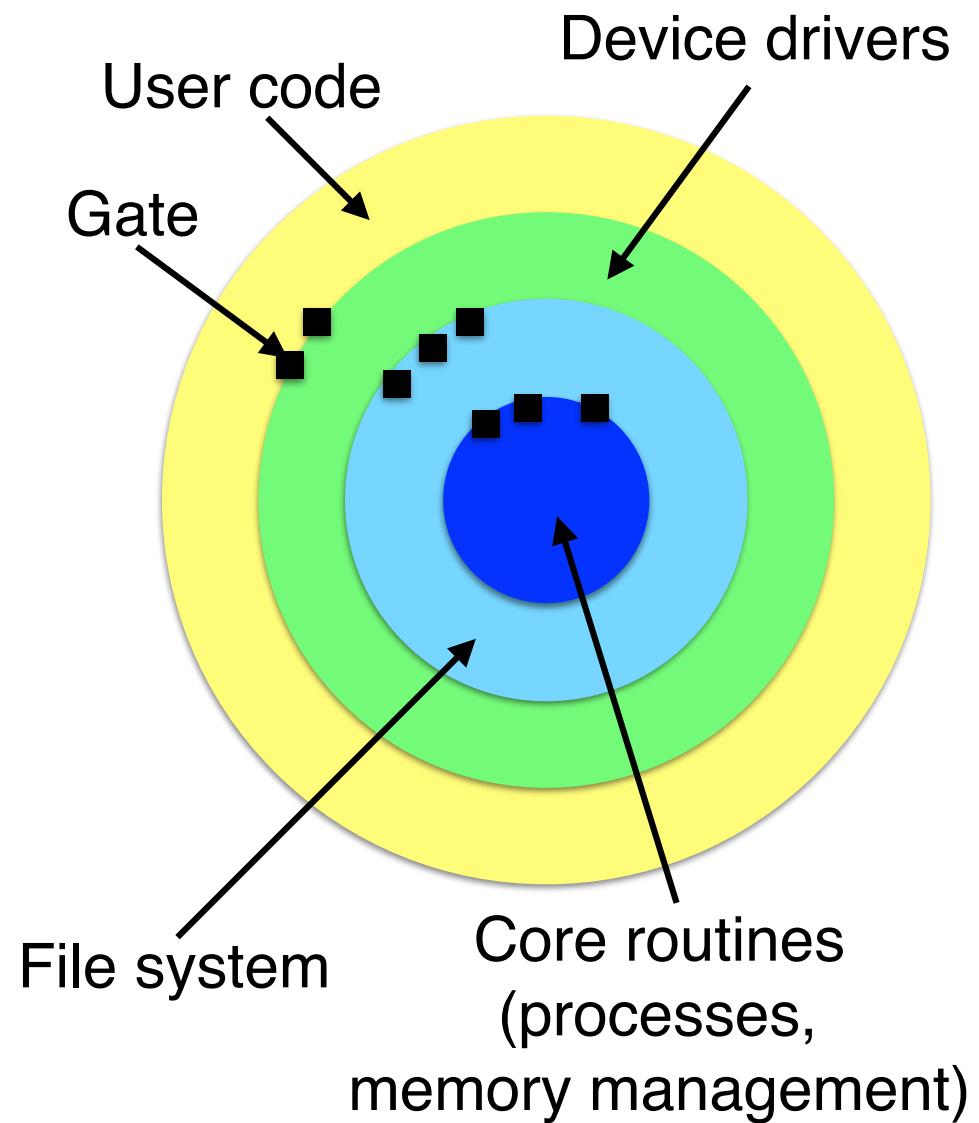
Monolithic OS structure

- ❖ All of the OS is one big “program”
 - Any piece can access any other piece
- ❖ Sometimes modular (as with Linux and *FreeBSD*)
 - Extra pieces can be dynamically added
 - Extra pieces become part of the whole
- ❖ Easy to write, but harder to get right...

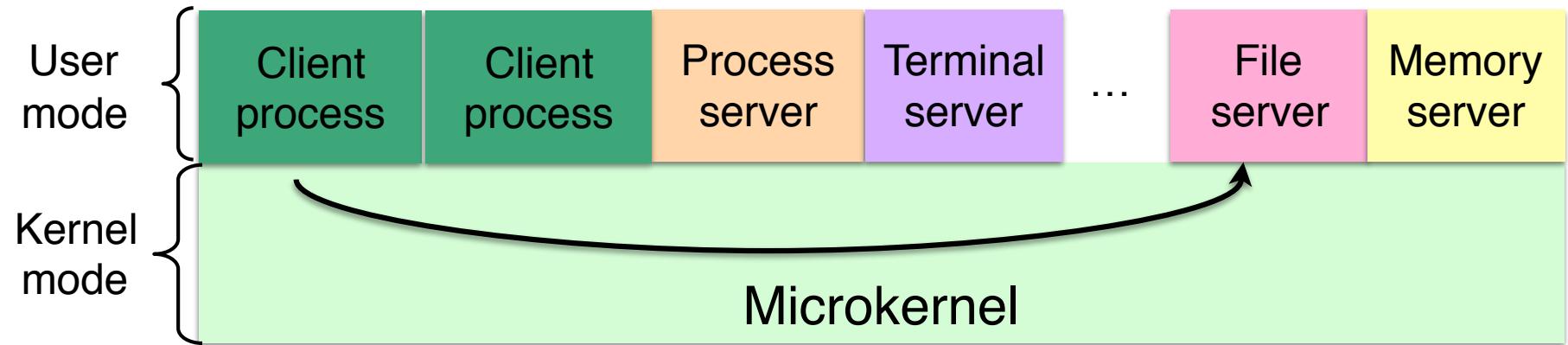


Layered OS

- ❖ CPU supports “layers” of privilege
 - Intel supports 4
 - User code at outermost layer
- ❖ Use privilege layers to enforce separation & provide security
 - Outer layers can only move inward at specific code points (often called *gates*)
- ❖ Not widely used today, but might be a good idea...

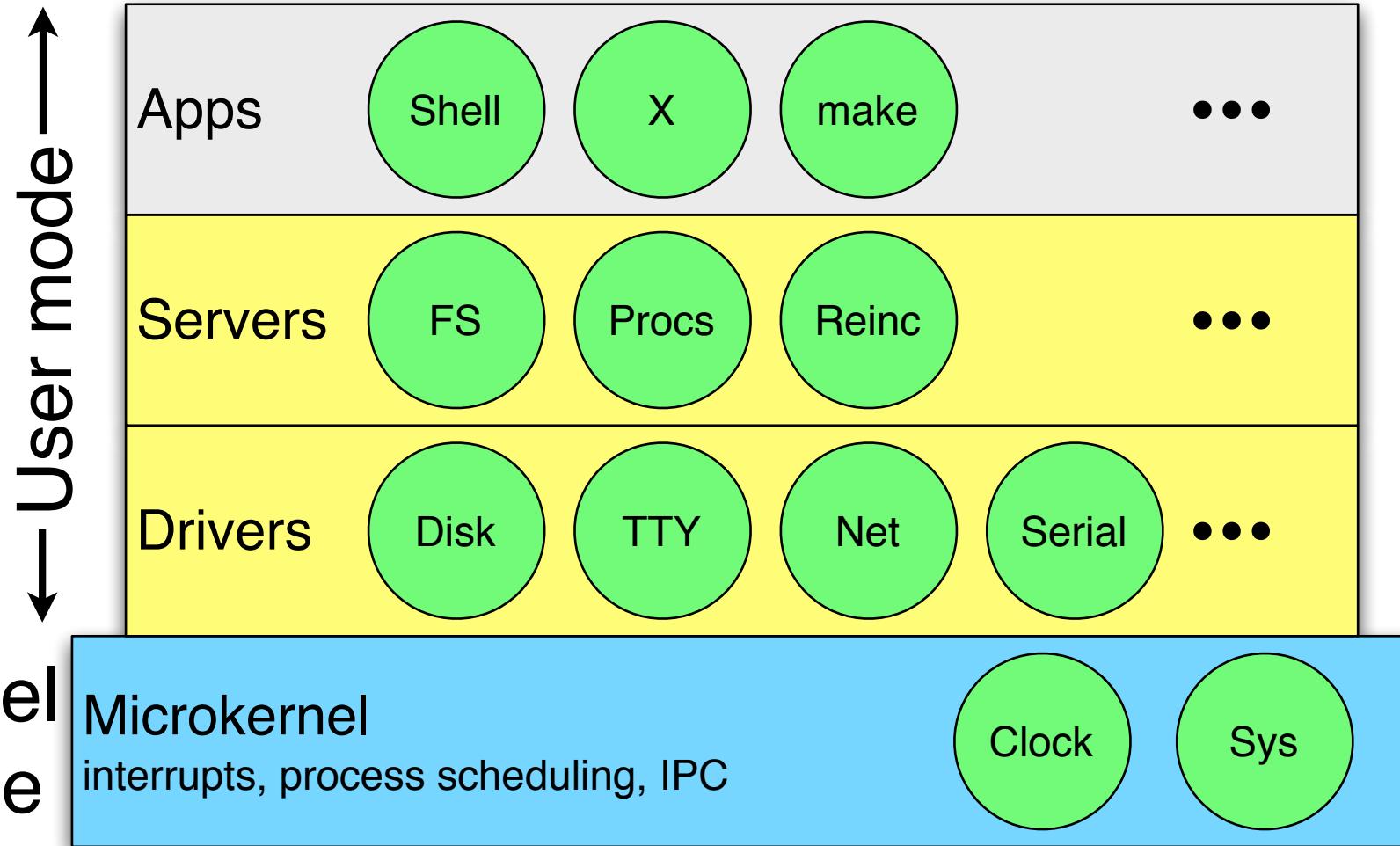


Microkernels (client-server)

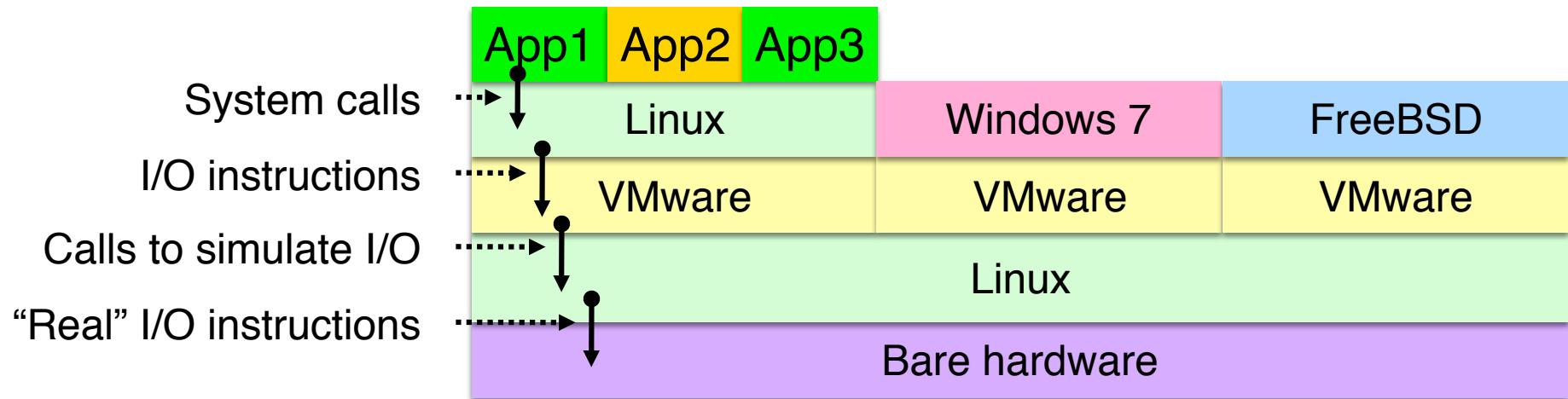


- ❖ Processes (clients and OS servers) don't share memory
 - Communication via message-passing
 - Separation reduces risk of “byzantine” failures
- ❖ Examples include
 - Mach (used by MacOS X)
 - MINIX

MINIX3 microkernel

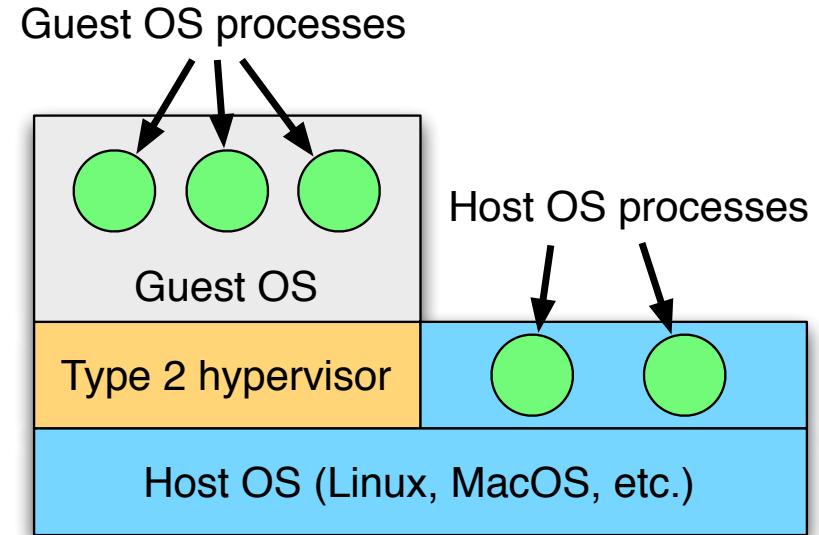
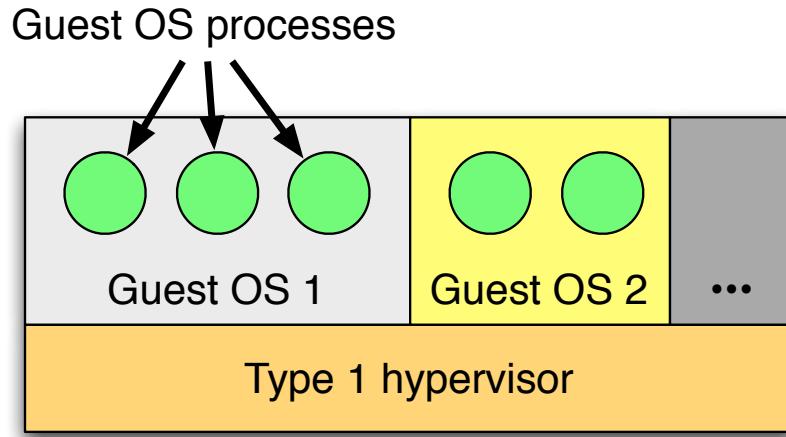


Virtual machines



- ❖ First widely used in VM/370 with CMS
- ❖ Available today in VMware and VirtualBox
 - Allows users to run any x86-based OS on top of Linux, Windows, MacOS
- ❖ “Guest” OS can crash without harming underlying OS
 - Only virtual machine fails—rest of underlying OS is fine
- ❖ “Guest” OS can even use raw hardware
 - Virtual machine keeps things separated

Two types of VM managers



❖ Type 1 hypervisor

- Runs on bare hardware
- Manages resources between guest operating systems
- May provide some simple OS-like services (FS, scheduling)

❖ Type 2 hypervisor

- Runs as a process in the “host” operating system
- May use host OS services (file system, etc.)

Why use virtual machines?

- ❖ Virtual machines encapsulate all of a computer's state in a single file
 - Easy to store and relocate “computers” when packaged as virtual machines
 - Keep your environment in the cloud
 - Put it on any convenient computer
 - Make systems more reliable: use any computer to run your environment
- ❖ Virtual machines allow users low-level access to “hardware”
 - Shared hardware with the illusion of having your own computer
- ❖ Is there a difference between a VM manager and an OS?