

CS 1550: Introduction to Operating Systems

Prof. Ahmed Amer

amer@cs.pitt.edu

<http://www.cs.pitt.edu/~amer/cs1550>

Chapter 1





Class outline

- Introduction, concepts, review & historical perspective
- Processes
 - Synchronization
 - Scheduling
 - Deadlock
- Memory management, address translation, and virtual memory
- Operating system management of I/O
- File systems
- Security & protection
- Distributed systems (as time permits)





Overview: Chapter 1

- 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





What *is* an operating system?

- A program that runs on the “raw” hardware and supports
 - Resource Abstraction
 - Resource Sharing
- Abstracts and standardizes the interface to the user across different types of hardware
 - Virtual machine hides the messy details which must be performed
- Manages the hardware resources
 - 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
- Next generation: ???
 - Systems connected by high-speed networks?
 - Wide area resource management?



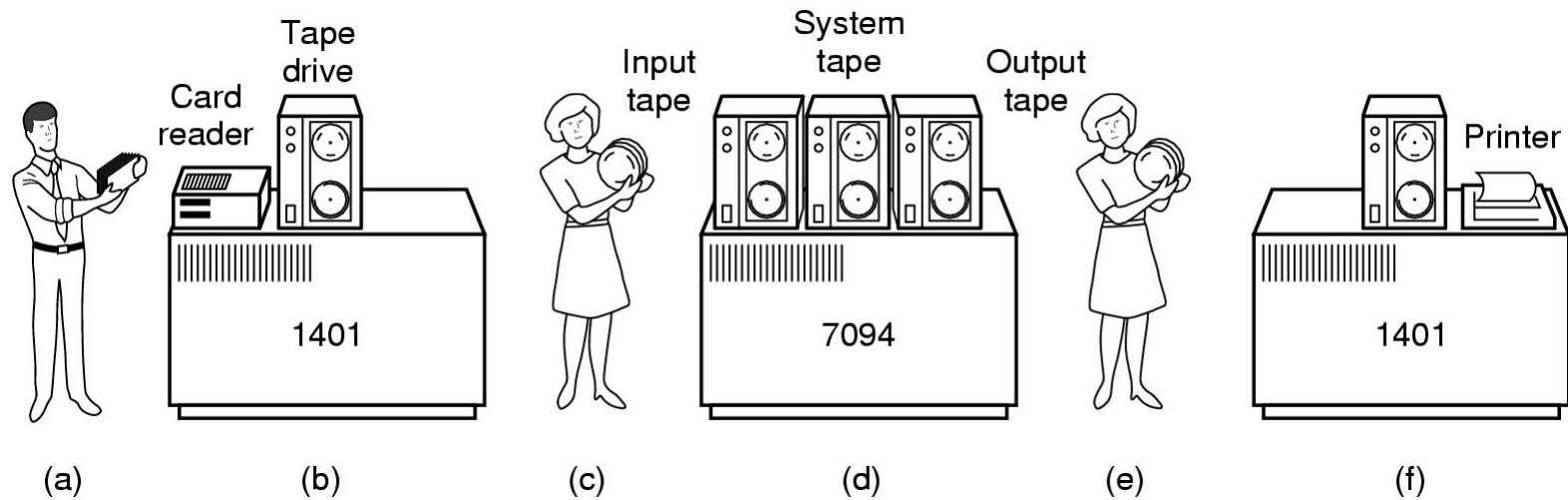


First generation: direct input

- 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

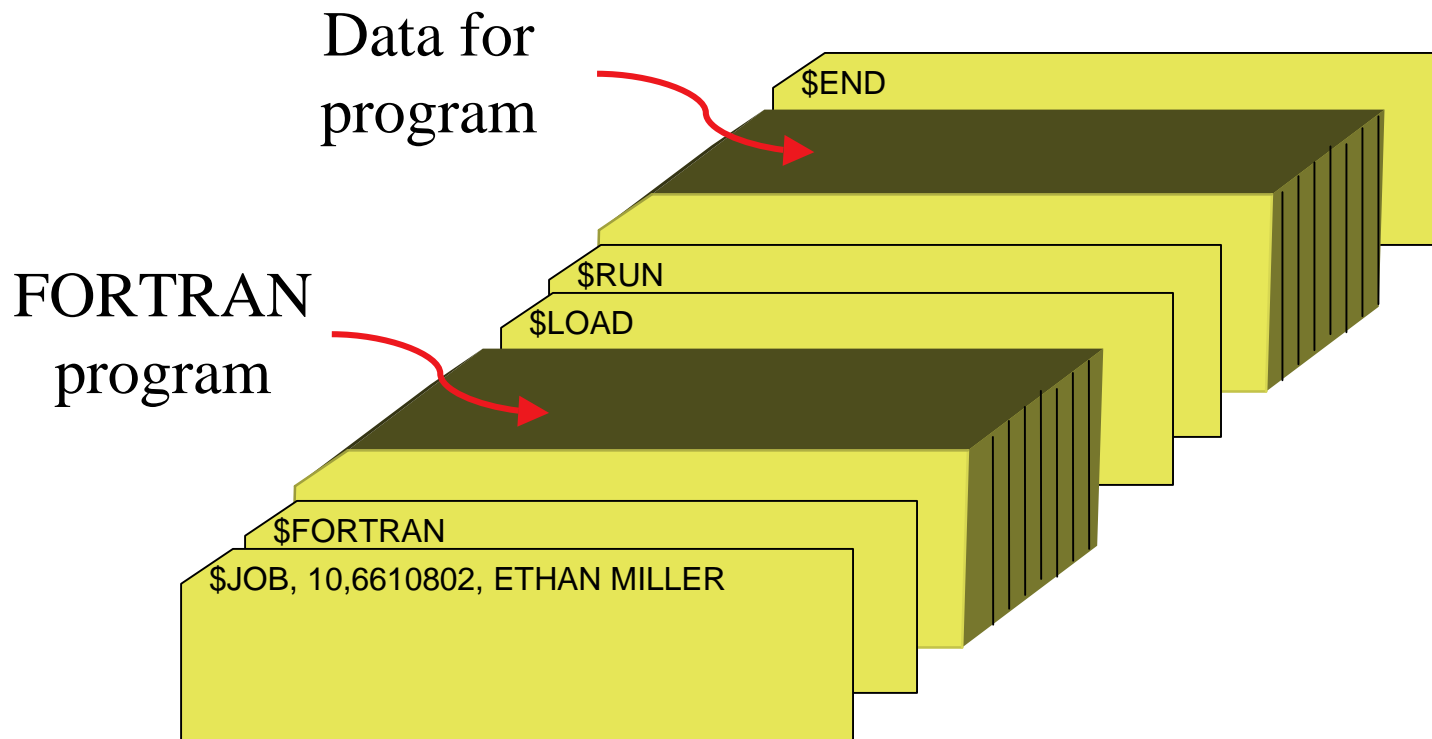


Second generation: batch systems



- 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



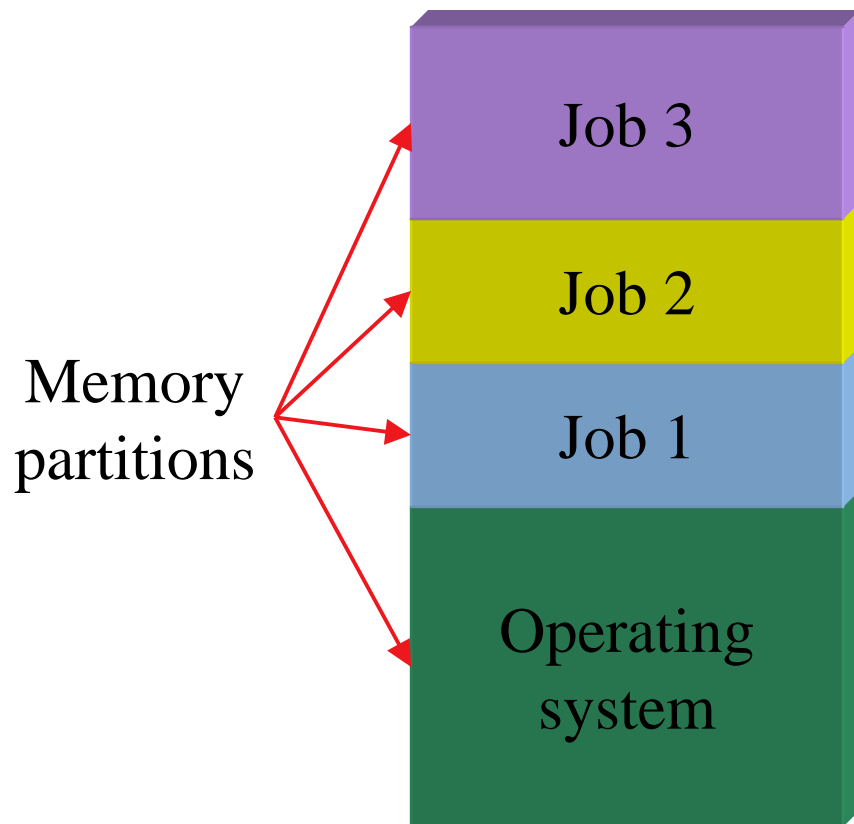


Spooling

- 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
- Operating system protected from each job as well
- Resources (time, hardware) 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



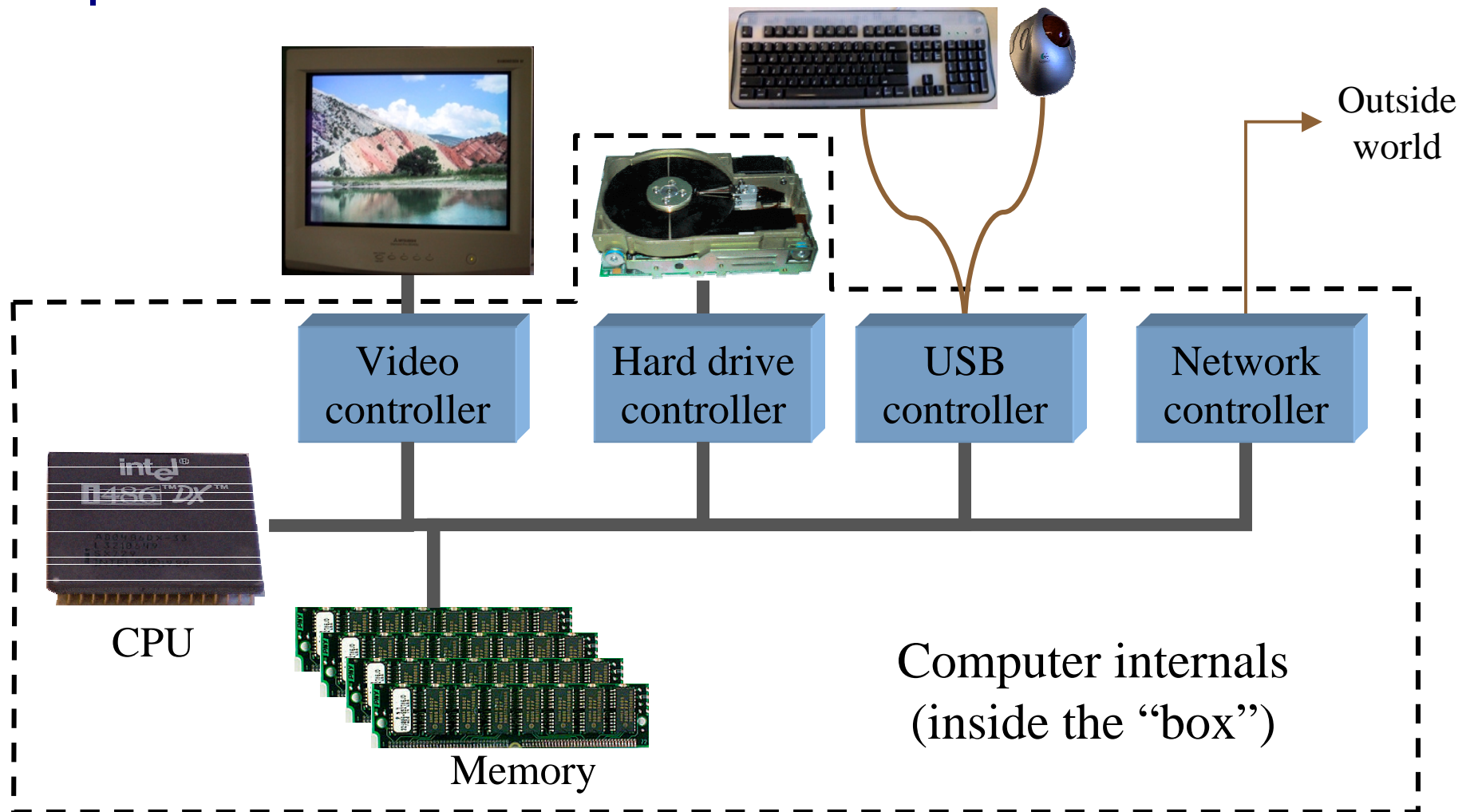


Types of modern operating systems

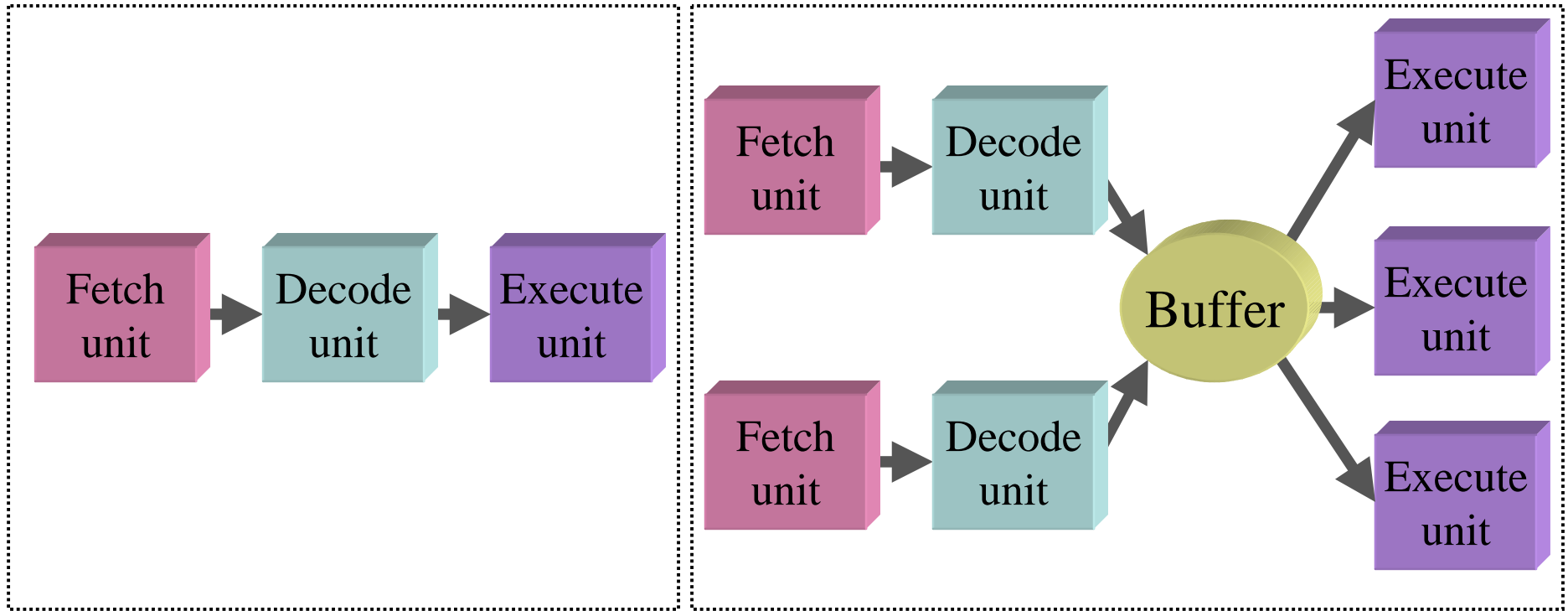
- Mainframe operating systems: MVS
 - Server operating systems: FreeBSD, Solaris
 - Multiprocessor operating systems: Cellular IRIX
 - Personal computer operating systems: Windows, Unix
 - Real-time operating systems: VxWorks
 - Embedded operating systems
 - Smart card operating systems
- ⇒ Some operating systems can fit into more than one category



Components of a simple PC



CPU internals

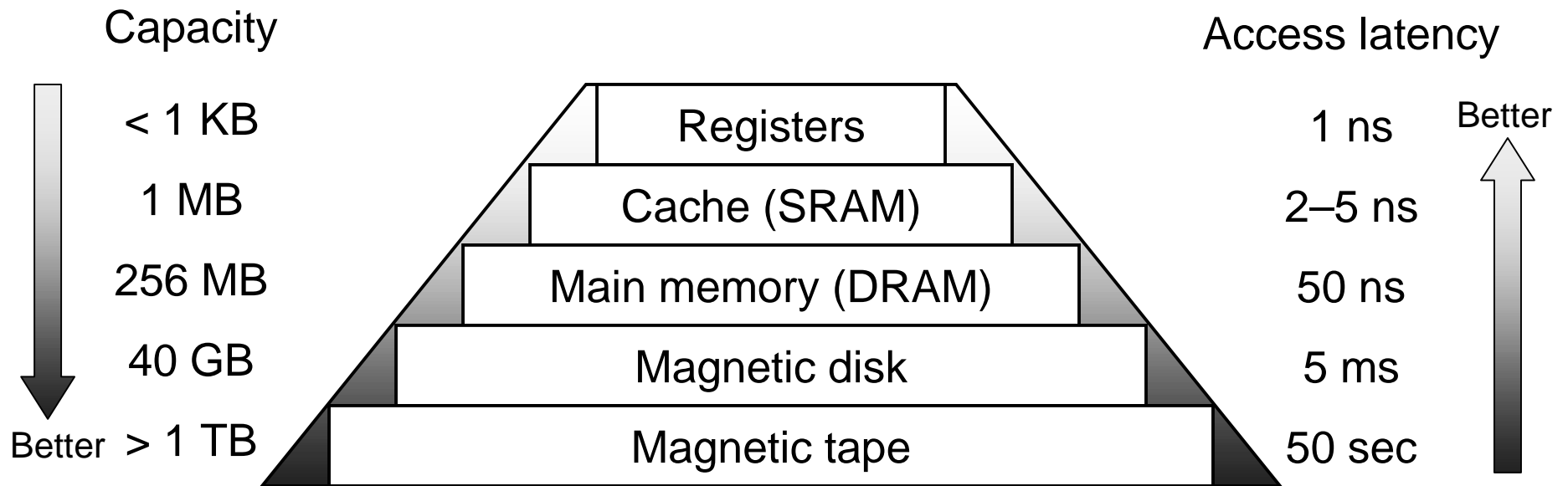


Pipelined CPU

Superscalar CPU



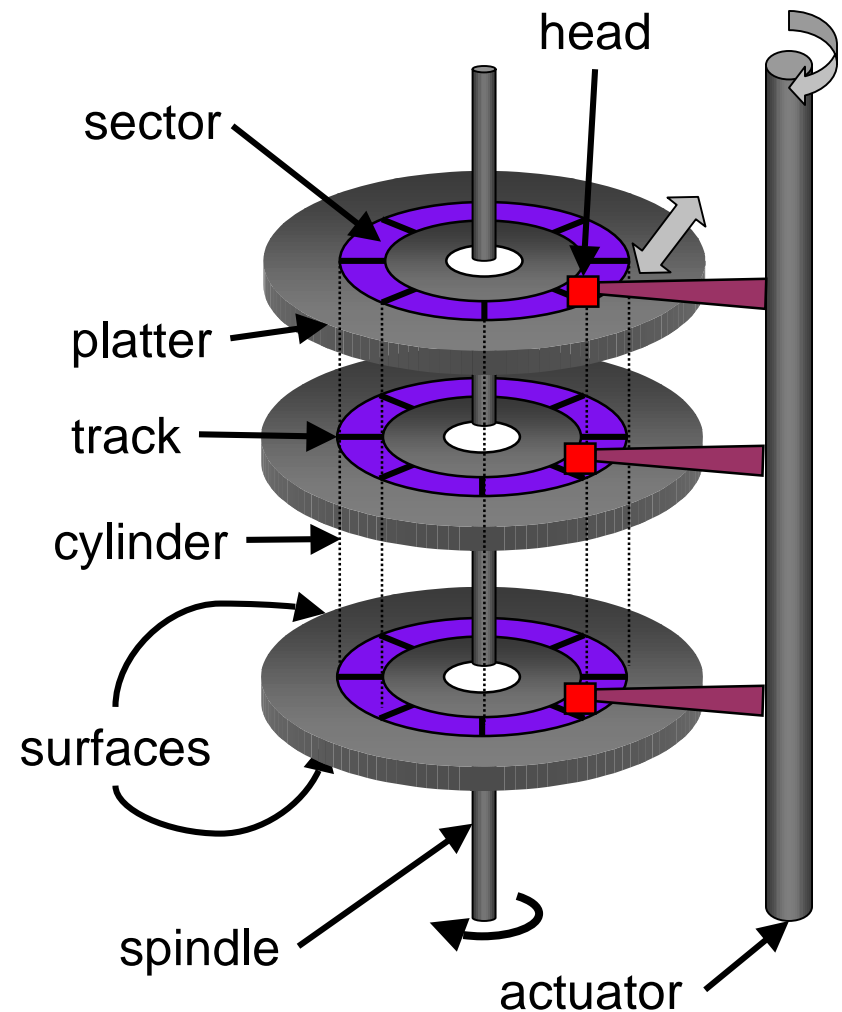
Storage pyramid



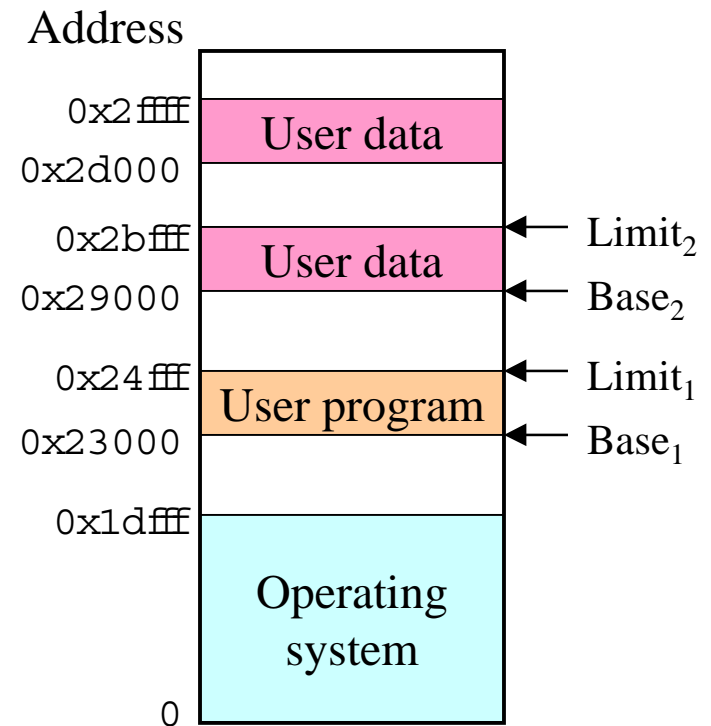
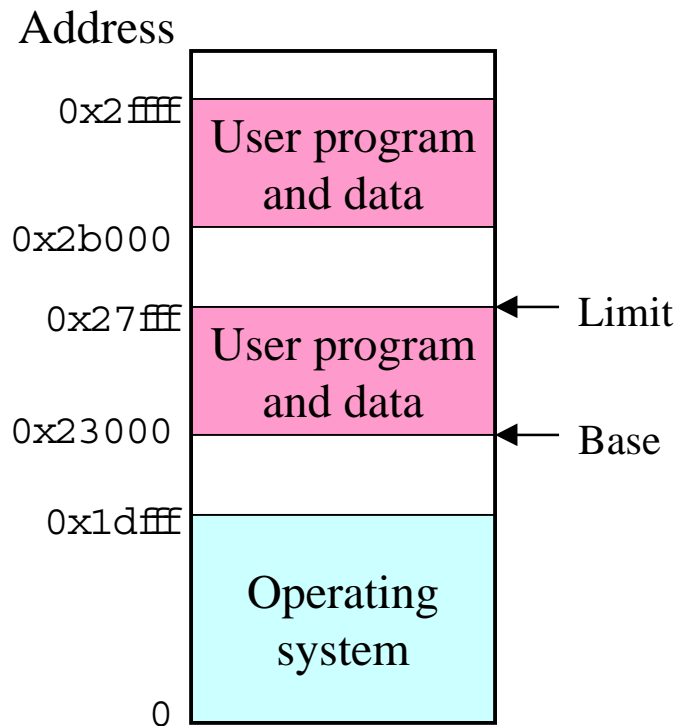
- Goal: really large memory with very low latency
 - Latencies are smaller at the top of the hierarchy
 - Capacities are larger at the bottom of the hierarchy
- Solution: move data between levels to create illusion of large memory with low latency

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



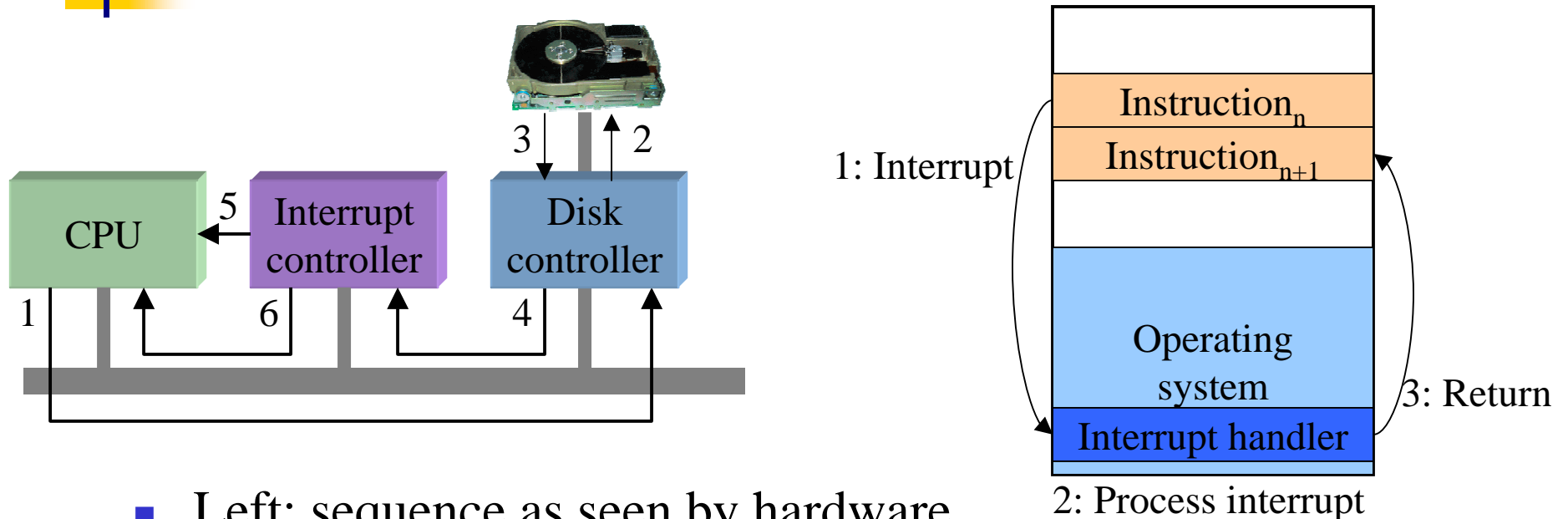
Memory



- 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)

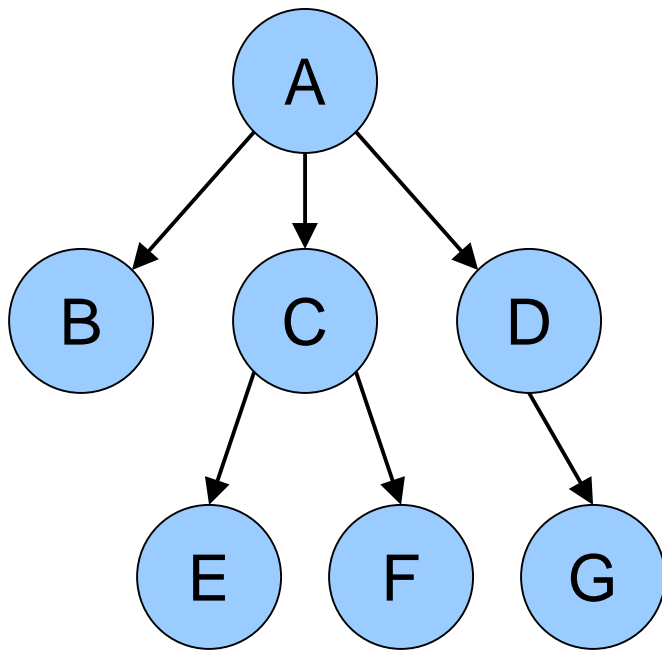


Operating systems concepts

- 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



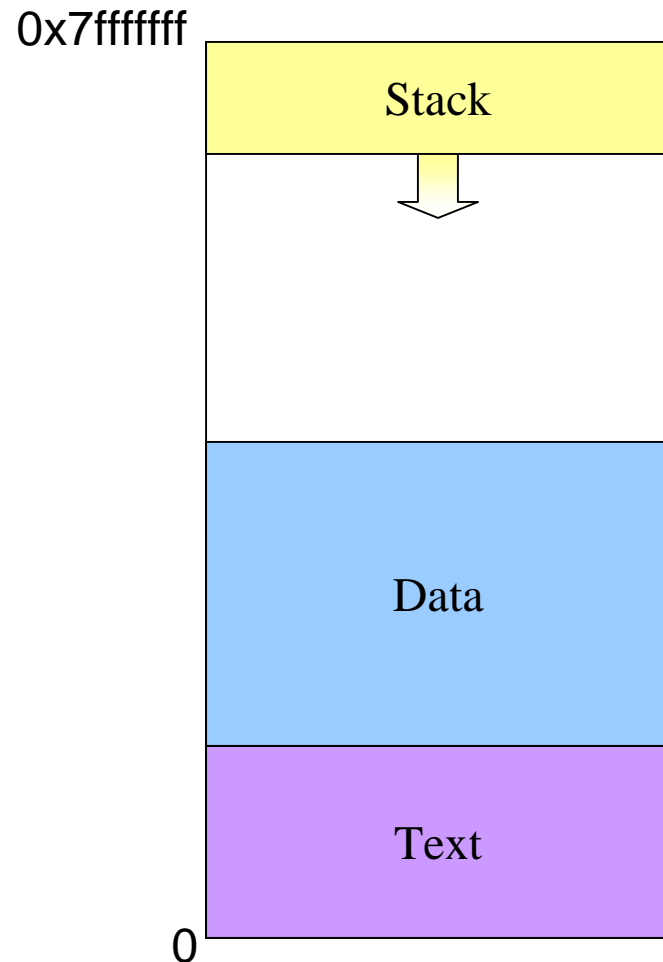
Processes



- 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



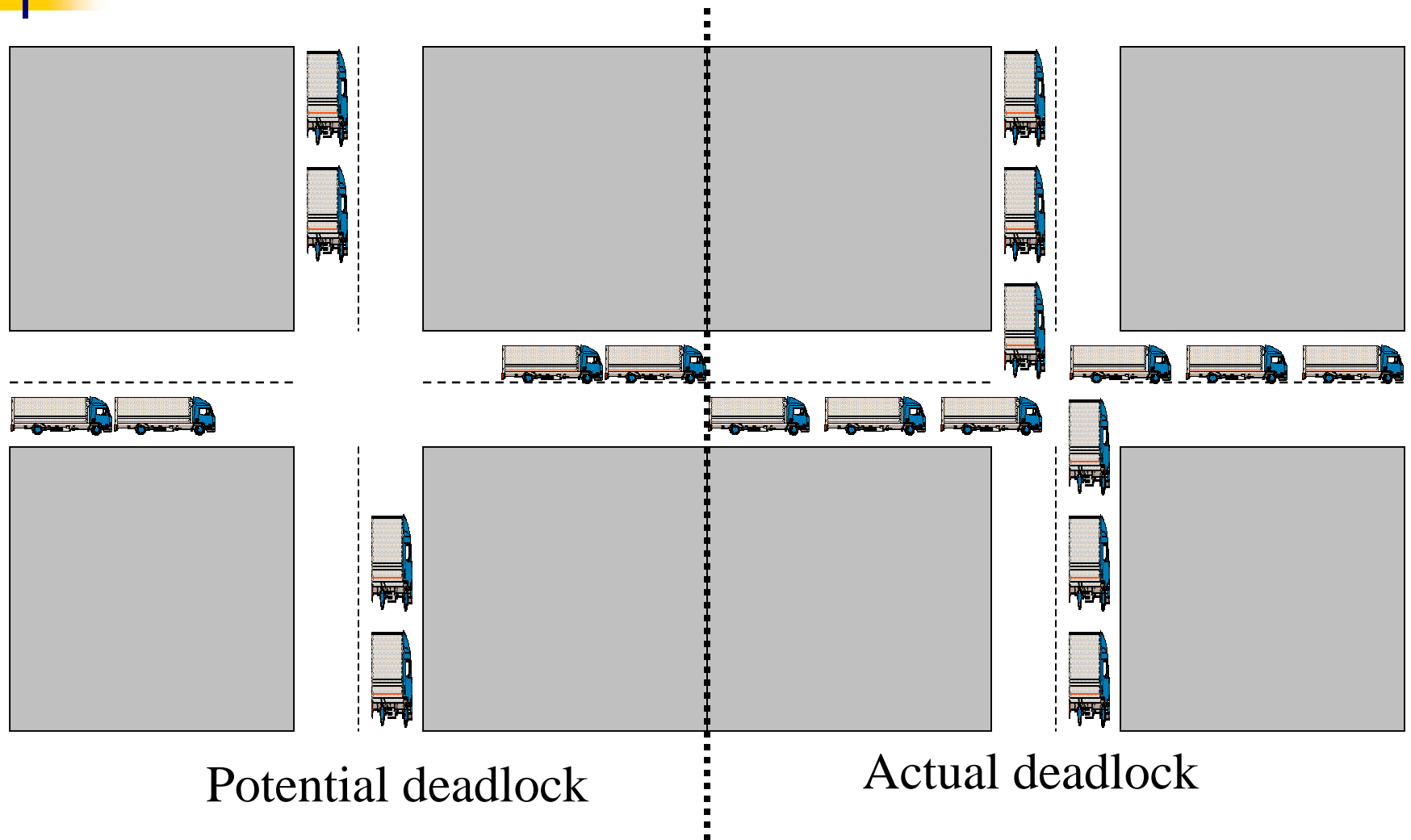
Inside a (Unix) process



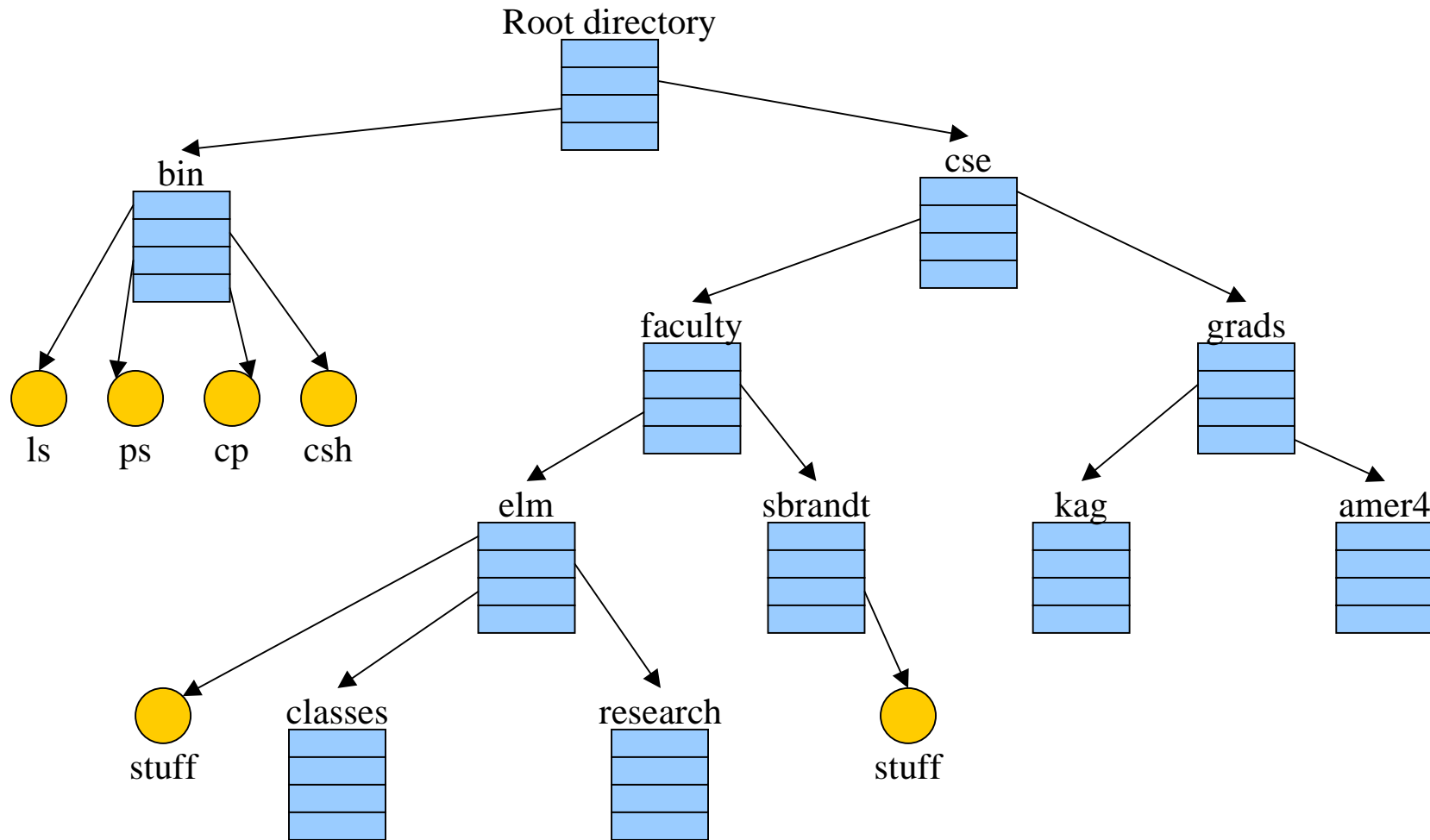
- Processes have three segments
 - Text: program code
 - Data: program data
 - Statically declared variables
 - Areas allocated by `malloc()` or `new`
 - Stack
 - Automatic variables
 - Procedure call information
- Address space growth
 - Text: doesn't grow
 - Data: grows "up"
 - Stack: grows "down"



Deadlock

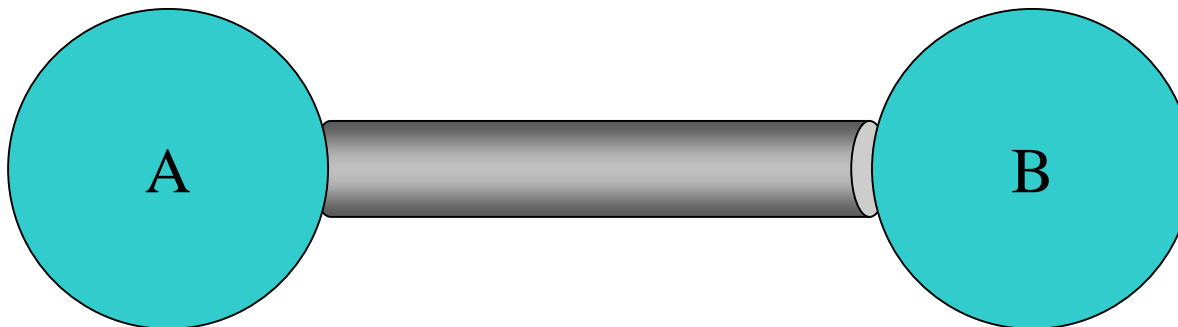


Hierarchical file systems



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



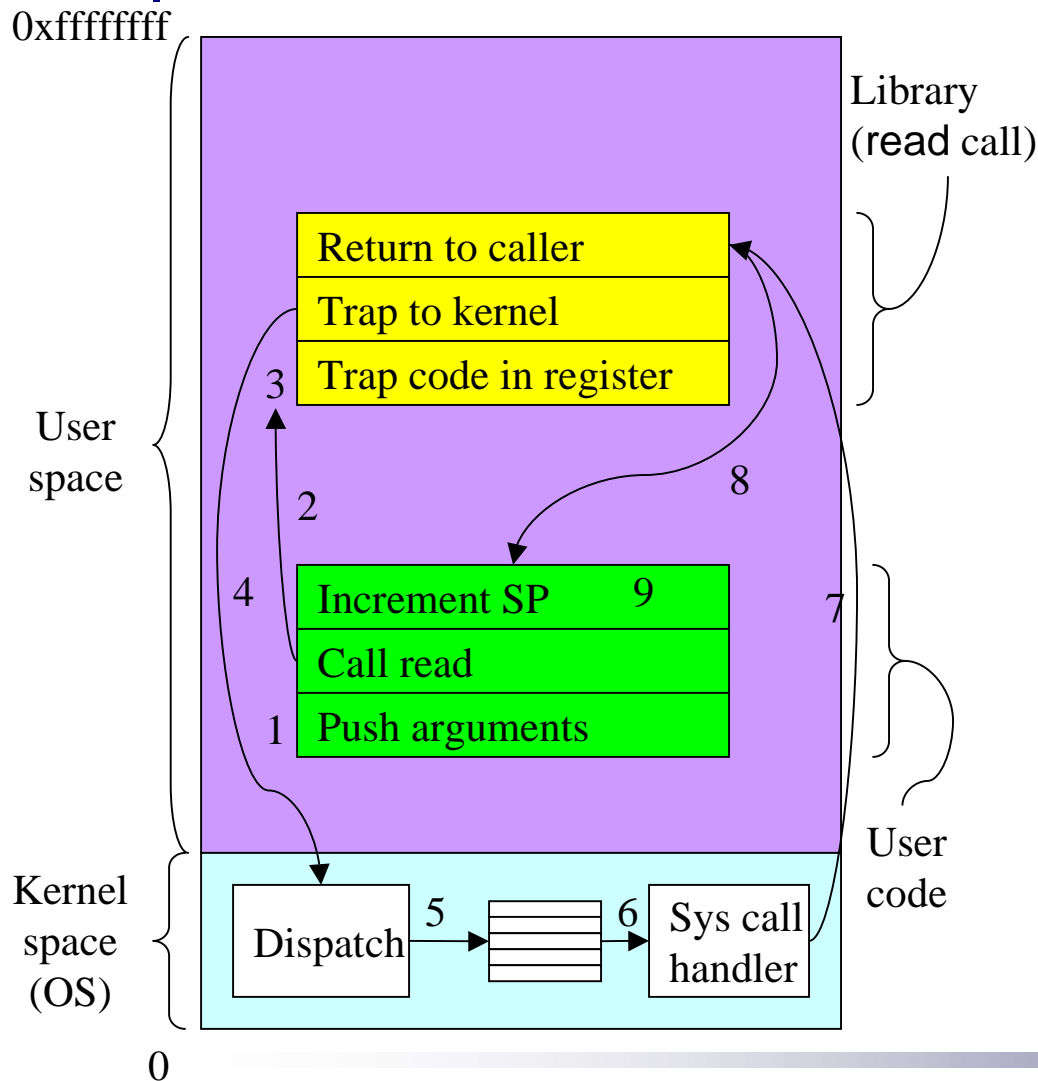


System calls

- Programs want the OS to perform a service
 - Access a file
 - Create a process
 - Others...
- Accomplished by system call
 - 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!



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

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 <i>buffer</i>)
<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 <i>name</i> as a link to an object (deletes the object if <i>name</i> was the only link to it)





More system calls

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid=waitpid(pid,&statloc,options)</code>	Wait for a child to terminate
<code>s = execve(name,argv,environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name,mode)</code>	Change a file's protection bits
<code>s = kill(pid,signal)</code>	Send a signal to a process
<code>seconds = time(&seconds)</code>	Get the elapsed time since 1 Jan 1970





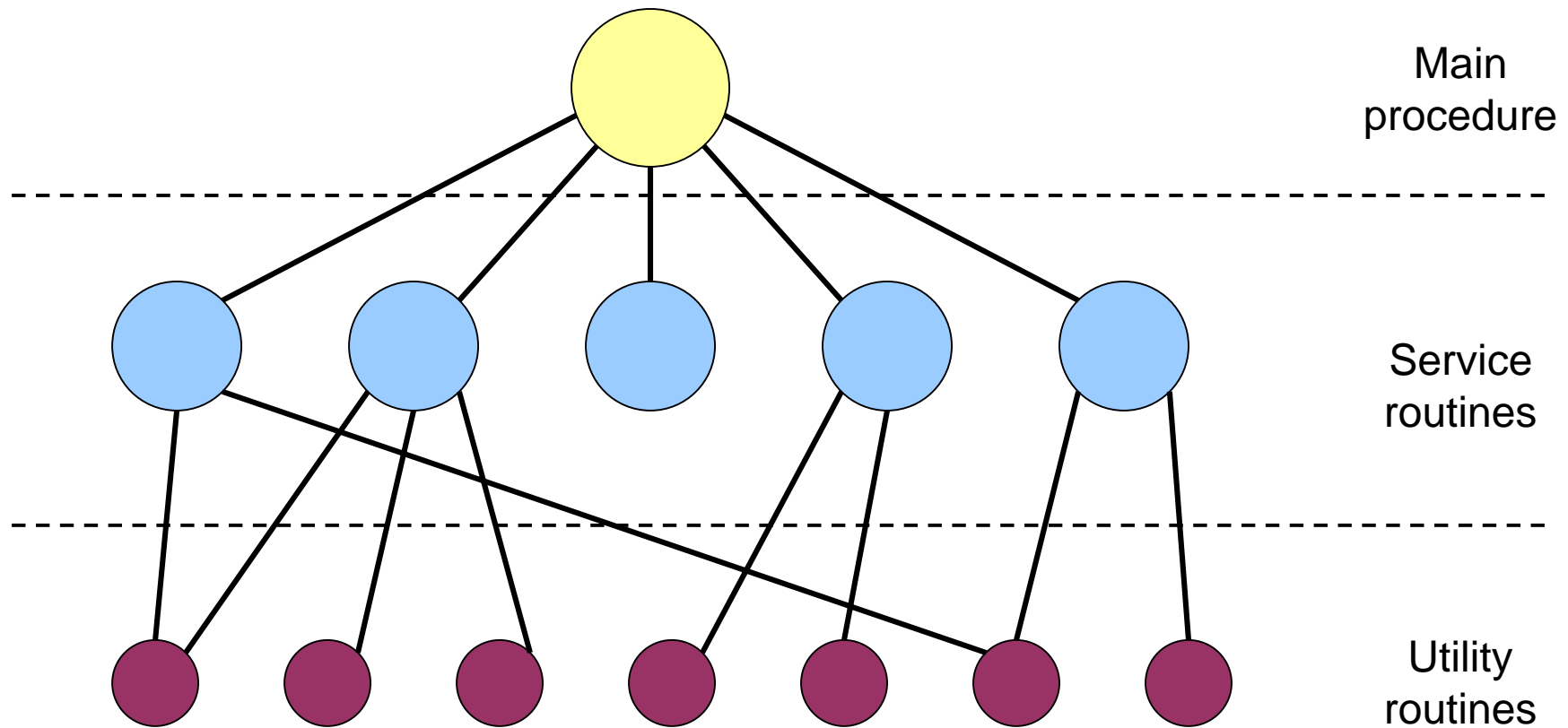
A simple shell

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

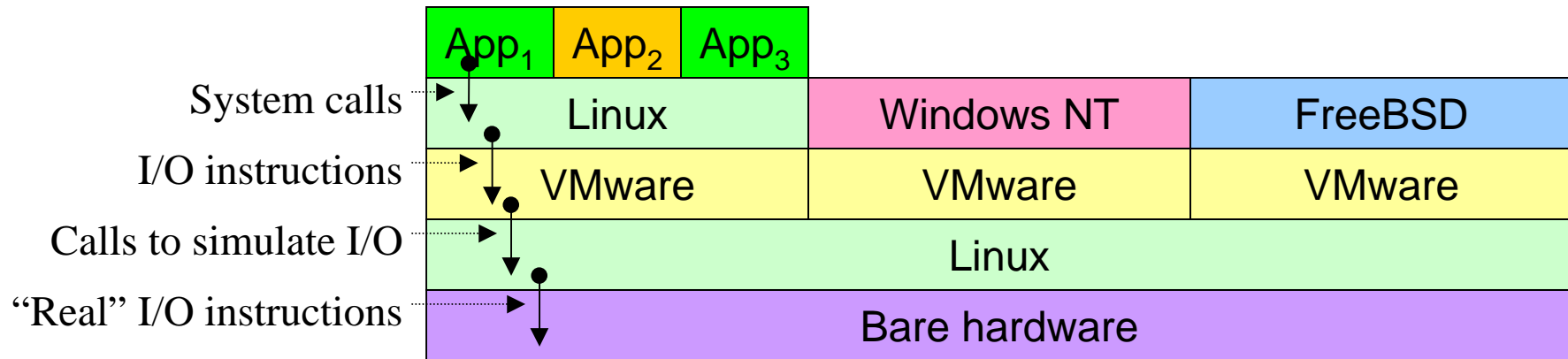
    if (fork() != 0) {                        /* fork off child process */
        /* Parent code */
        waitpid( -1, &status, 0);            /* wait for child to exit */
    } else {
        /* Child code */
        execve (command, parameters, 0);     /* execute command */
    }
}
```



Monolithic OS structure



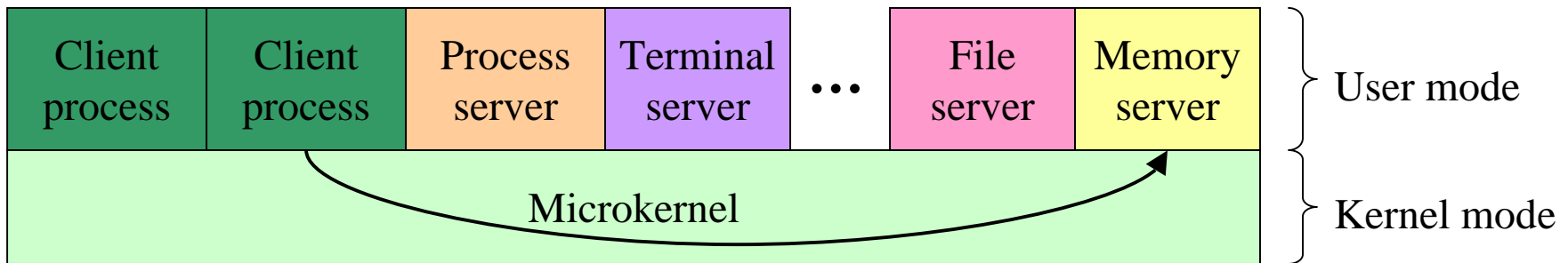
Virtual machines



- First widely used in VM/370 with CMS
- Available today in VMware
 - Allows users to run any x86-based OS on top of Linux or NT
- "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



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





Metric units

Exp.	Number	Prefix	Exp.	Number	Prefix
10^{-3}	0.001	milli	10^3	1,000	Kilo
10^{-6}	0.000001	micro	10^6	1,000,000	Mega
10^{-9}	0.000000001	nano	10^9	1,000,000,000	Giga
10^{-12}	0.0000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.00000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.0000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa

