# History of Operating Systems and Dual Mode Execution

CS439: Principles of Computer Systems

January 26, 2015

## Last Time

Operating Systems are:

- Referees:
  - Manage shared resources
  - Provide protection and communication for processes
- Illusionists:
  - Provide the illusion of infinite resources
- Glue:
  - Provide standard services which the hardware implements

## Today’s Agenda

- History of Operating Systems
  - Batch systems, Asynchronous I/O, Time Slicing
- Dual Mode Execution
  - User vs. Kernel modes

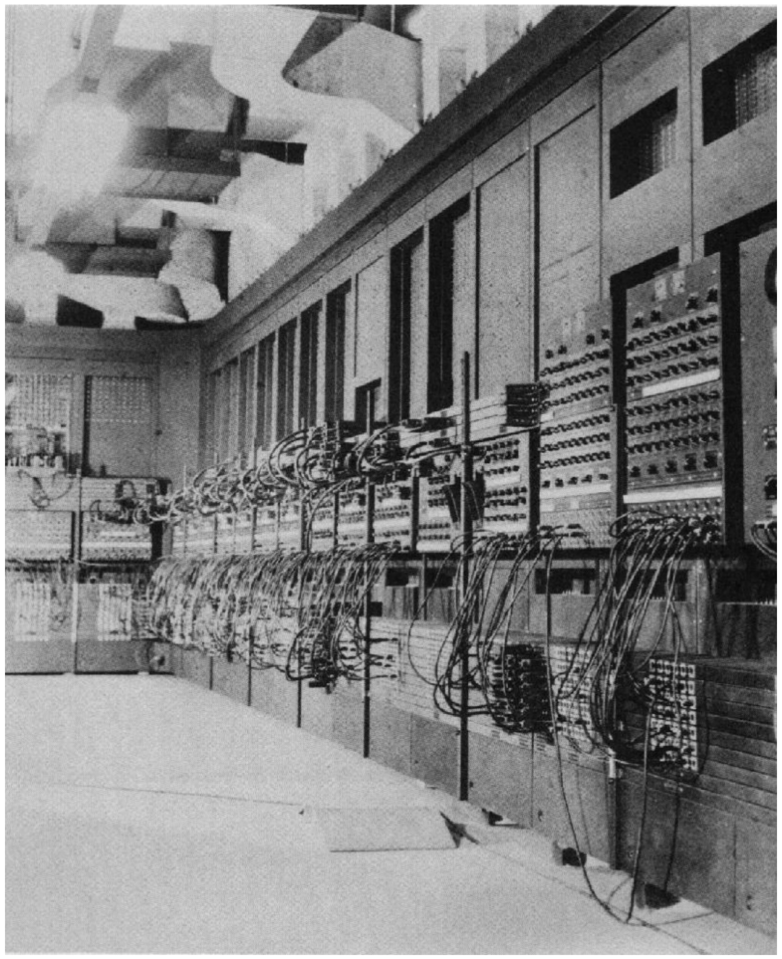
## History of Operating Systems

# Overview

- Phase 1: Hardware expensive, humans cheap
- Phase 2: Hardware cheap, humans expensive
- Phase 3: Hardware very cheap, humans very expensive

## Phase 1: Expensive Hardware, Cheap Humans

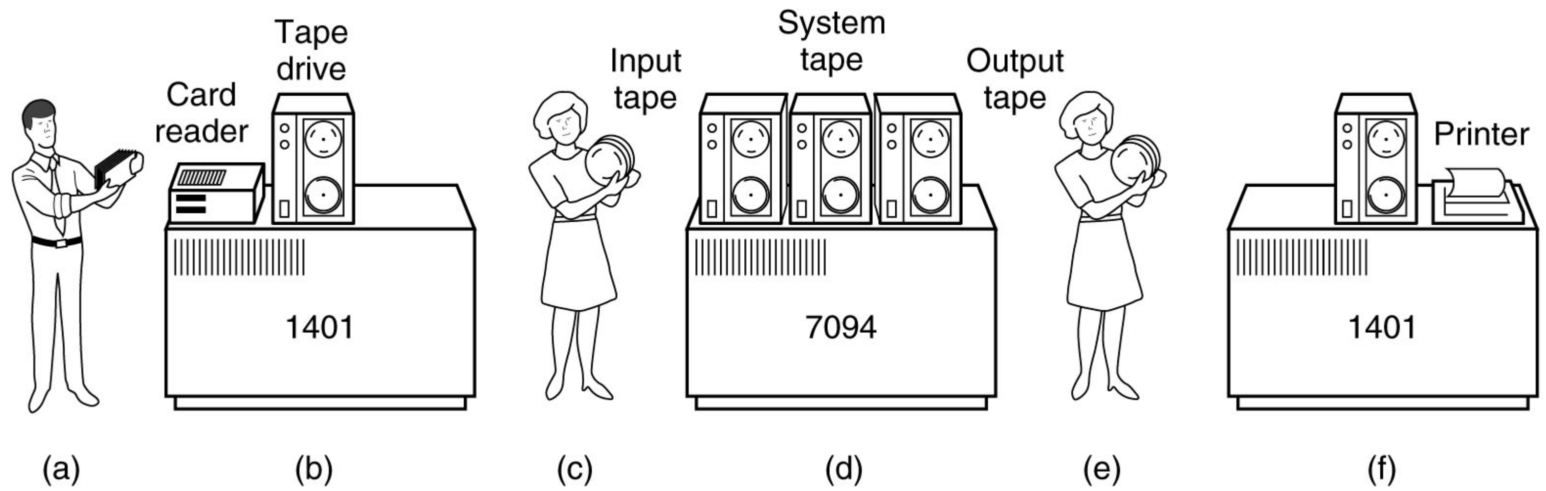
1. One user on the console, one process at a time (1945-1955)
   - Single user system
   - OS is a subroutine library (and a loader)
     - A stack of cards you pull off the shelf to do, for example, a matrix multiply
   - Problem: Low utilization of expensive components



## Phase 1: Expensive Hardware, Cheap Humans

2. Batch processing: load program, run, output to tape, print results, repeat (1955-1965)
   - Users give their program (on punch cards) to a human who schedules the jobs
   - OS loads, runs, and outputs user jobs
   - Advantage: next job can be loaded immediately as previous one finishes
     - Better use of hardware but debugging is much more difficult
   - Disadvantages:
     - No protection---a buggy program can crash the batch monitor
     - Computer is idle during I/O

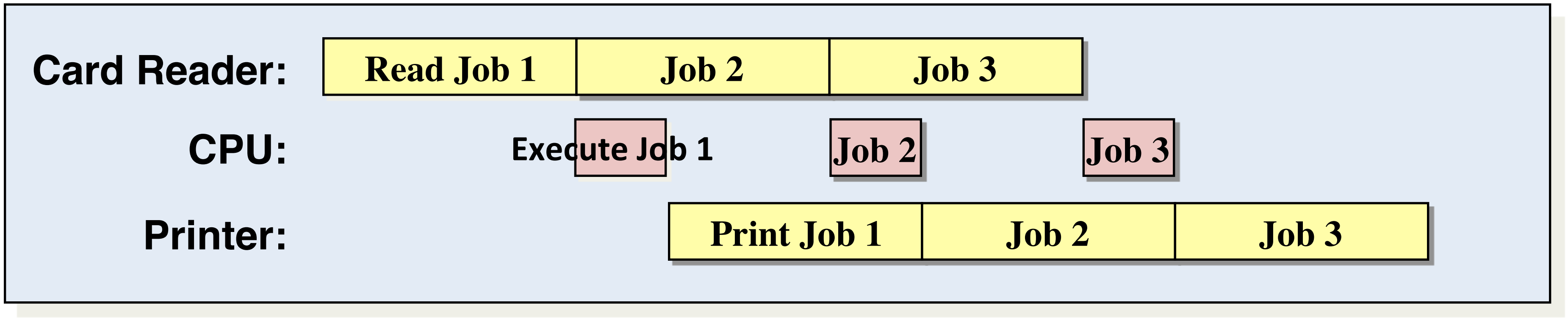
## Batch Processing



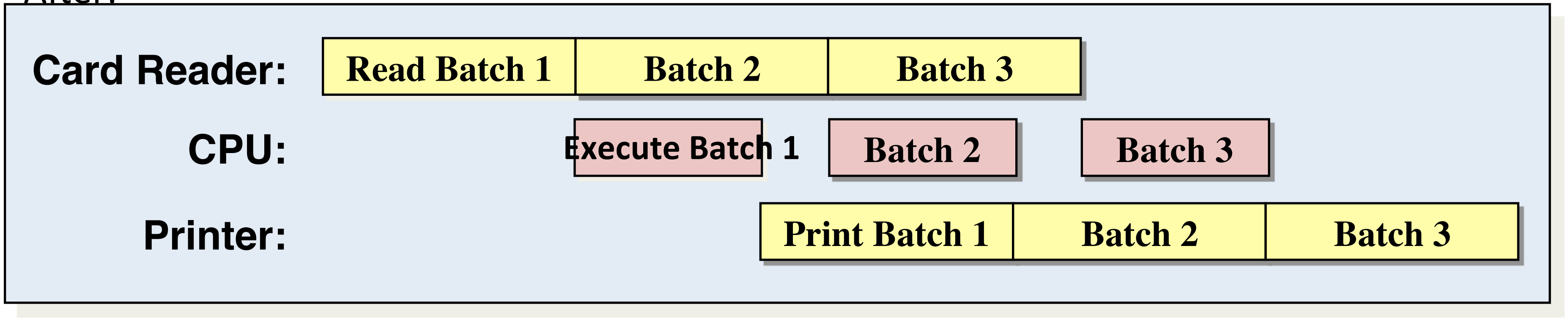
An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

# Batch Processing (1955-1965)

Before:



After:

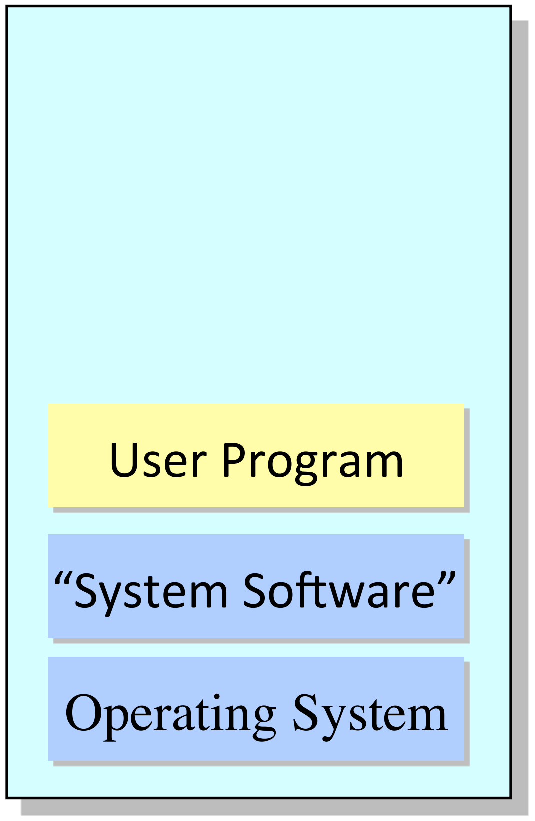


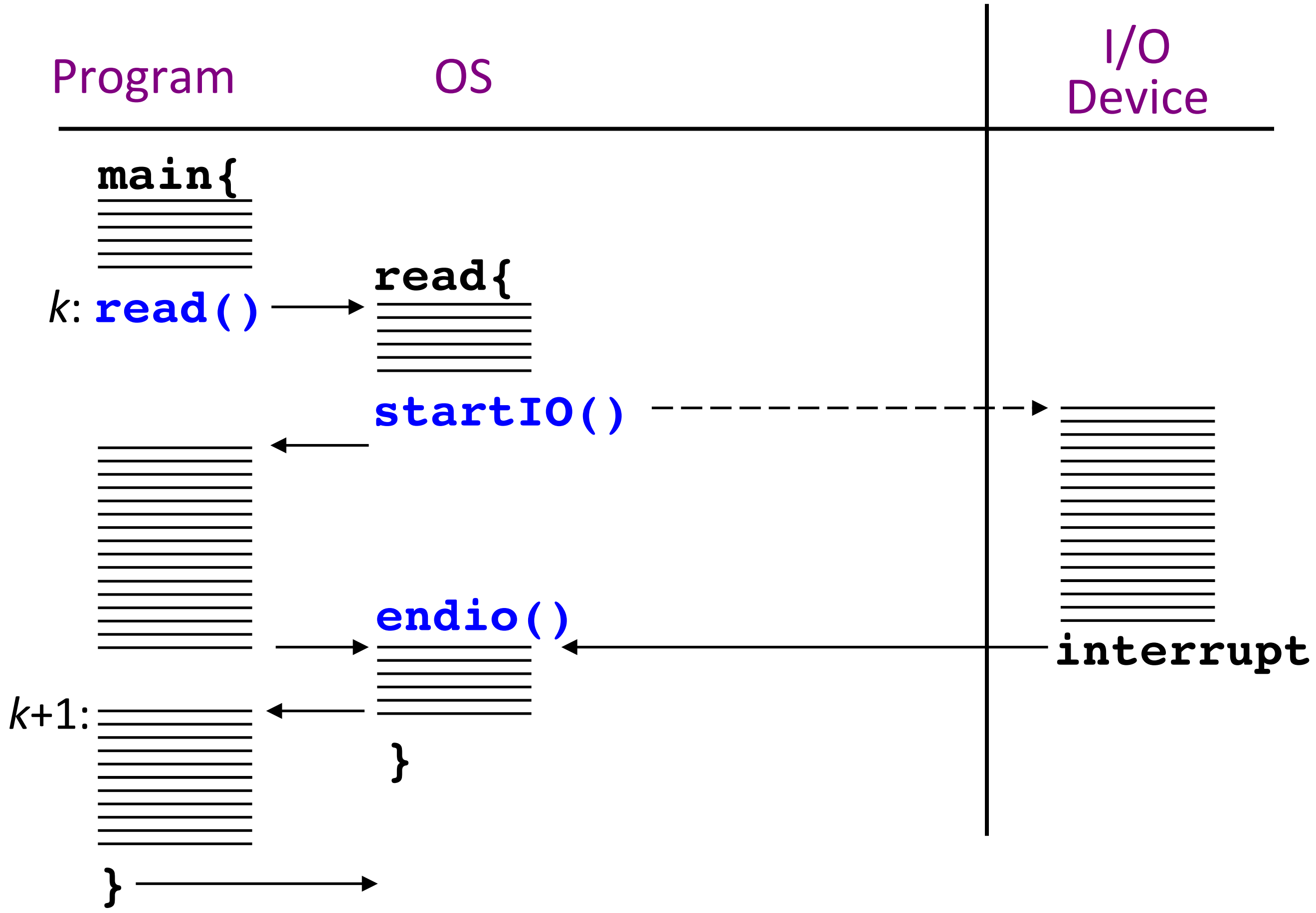
Each batch represents two jobs.

## Phase 1: Expensive Hardware, Cheap Humans

3. Data channels, interrupts, overlap of I/O and computation
   - OS requests I/O, goes back to computation, gets interrupt when I/O device finishes
   - No sharing, only protect OS from applications
   - Add concurrency *within same process*
   - Buffering and interrupt handling in OS
   - Spool jobs on the drum
   - Performance improves because I/O and processing happen concurrently

# Overlapping I/O and Computation



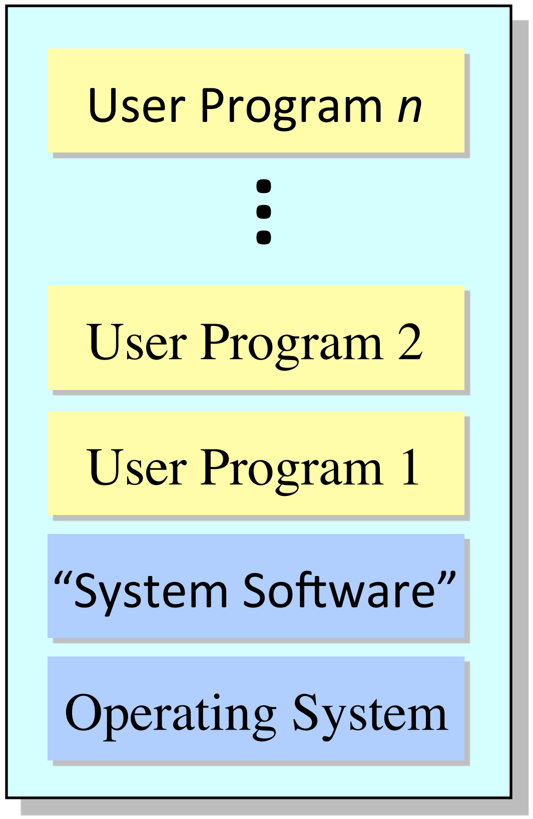


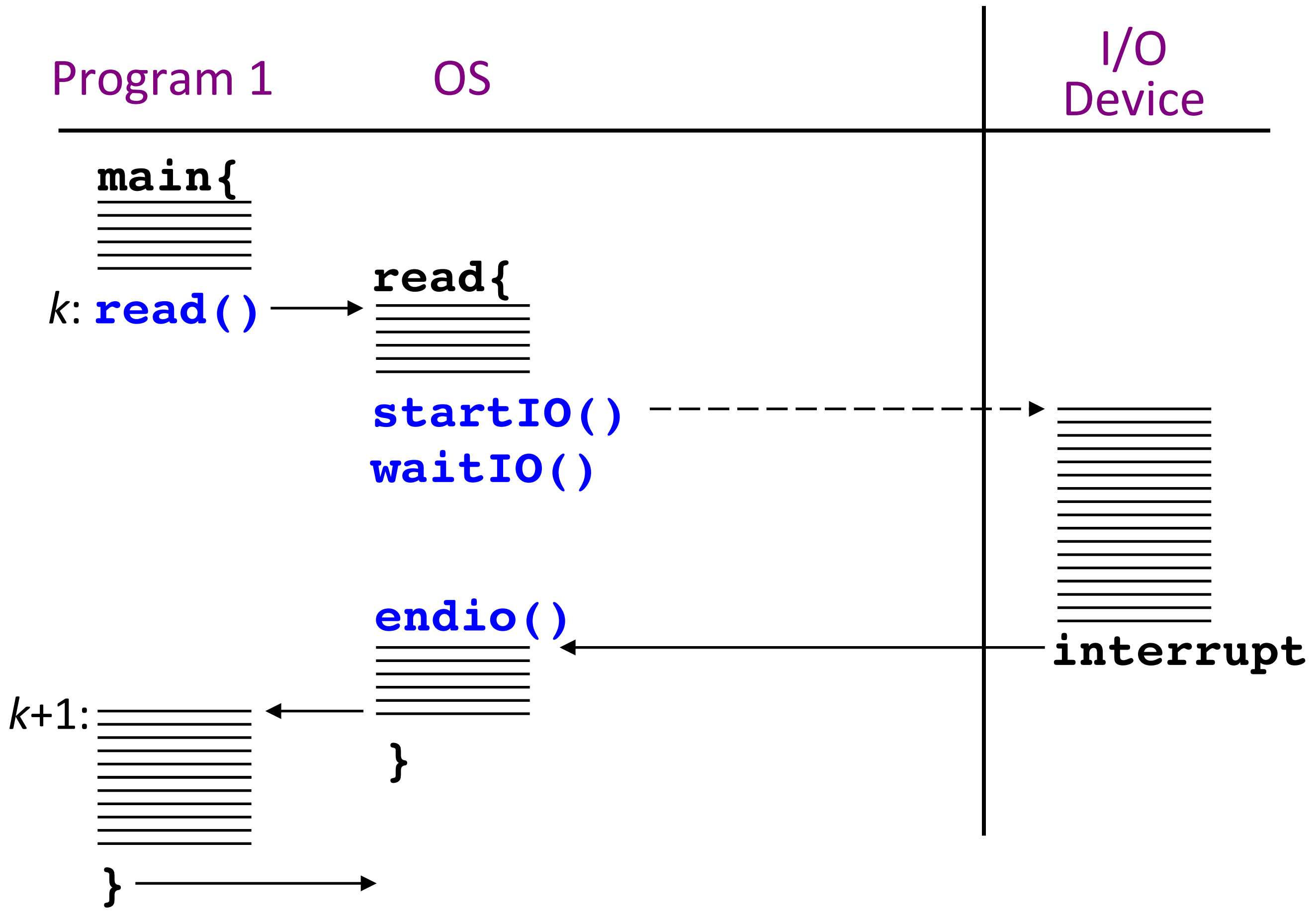
## Phase 1: Expensive Hardware, Cheap Humans

4. Multiprogramming: several programs run at the same time sharing the machine
   - One job runs until it performs I/O, then another job gets the CPU
   - OS manages interactions between concurrent programs (which ones start and execute, provides protection)
   - Requires: Memory protection and relocation

# Multiprogramming (1965-1980)

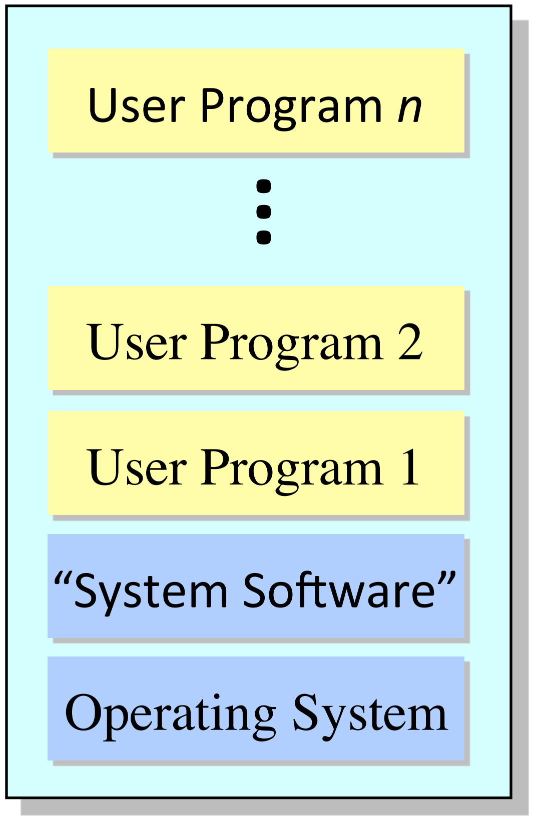
Keep several jobs in memory and multiplex CPU between jobs

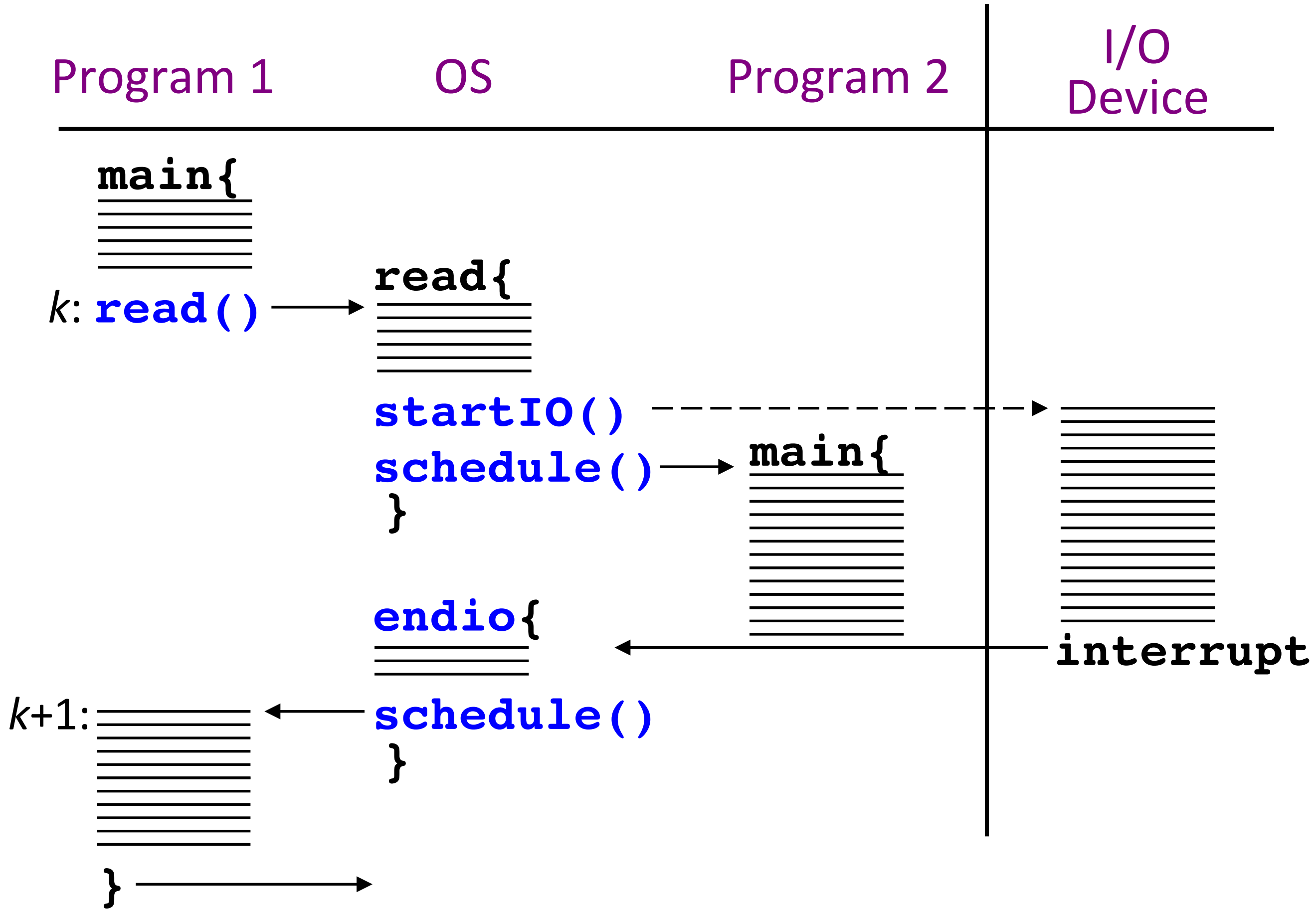




# Multiprogramming (1965-1980)

Keep several jobs in memory and multiplex CPU between jobs





## iClicker Question

In batch systems, each job must completely finish before the next job may begin.

A. True

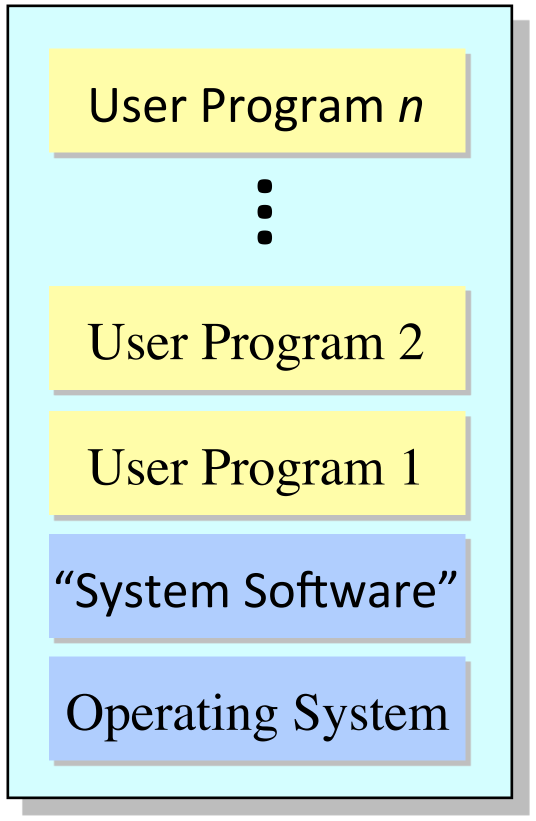
B. False

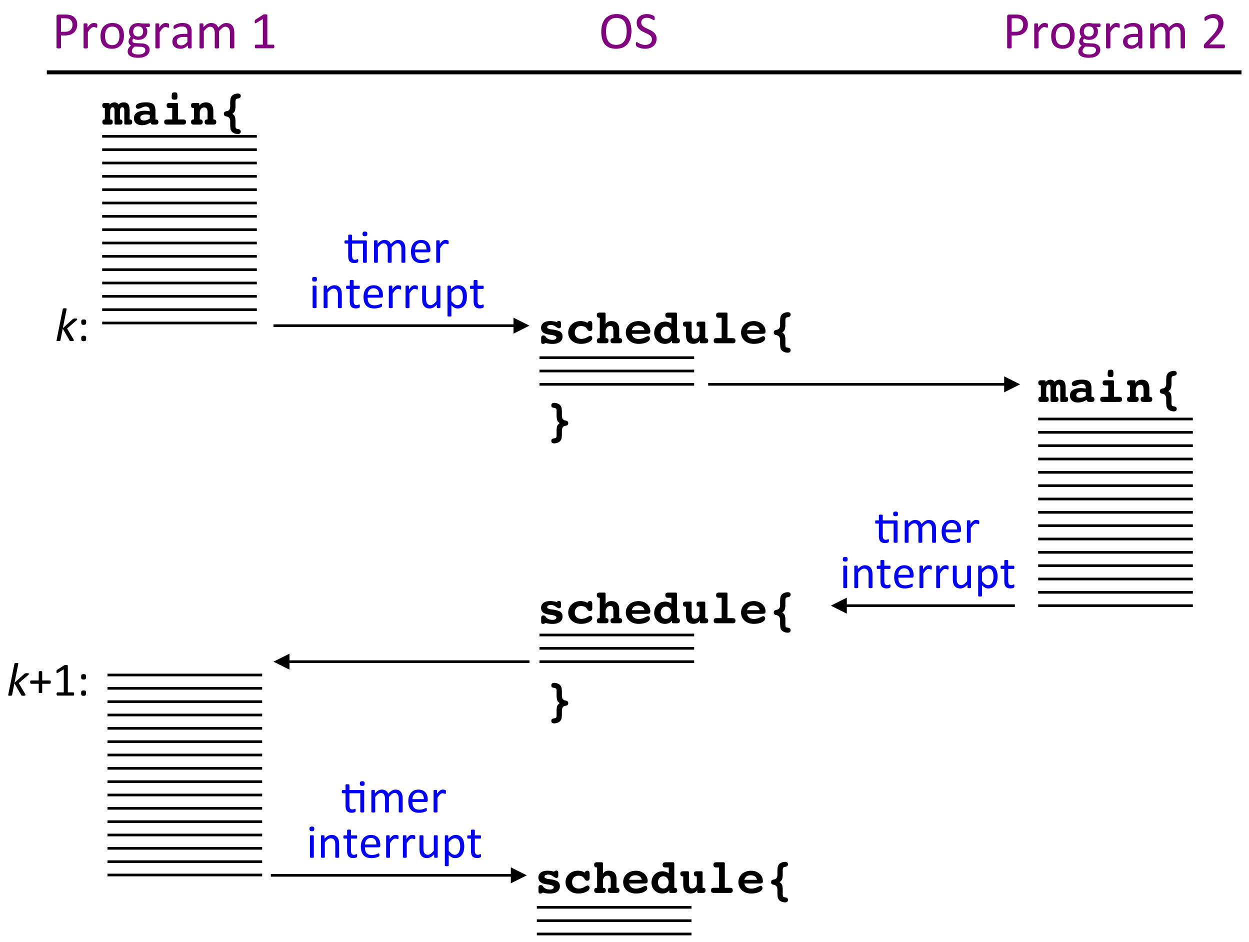
## Phase 2: Cheap Hardware, Expensive Humans

5. Interactive timesharing (1970-)
   - Use cheap terminals to let multiple users interact with the system at the same time
     - Debugging is a lot easier
     - Process switching occurs much more frequently
   - Requires: more sharing, more protection, more concurrency
   - New OS services: shell, file system, rapid process switching (users can interact!), virtual memory (processes running simultaneously!)
   - New problems: response time, thrashing

# Timesharing (1970-  )

A timer interrupt is used to multiplex CPU among jobs





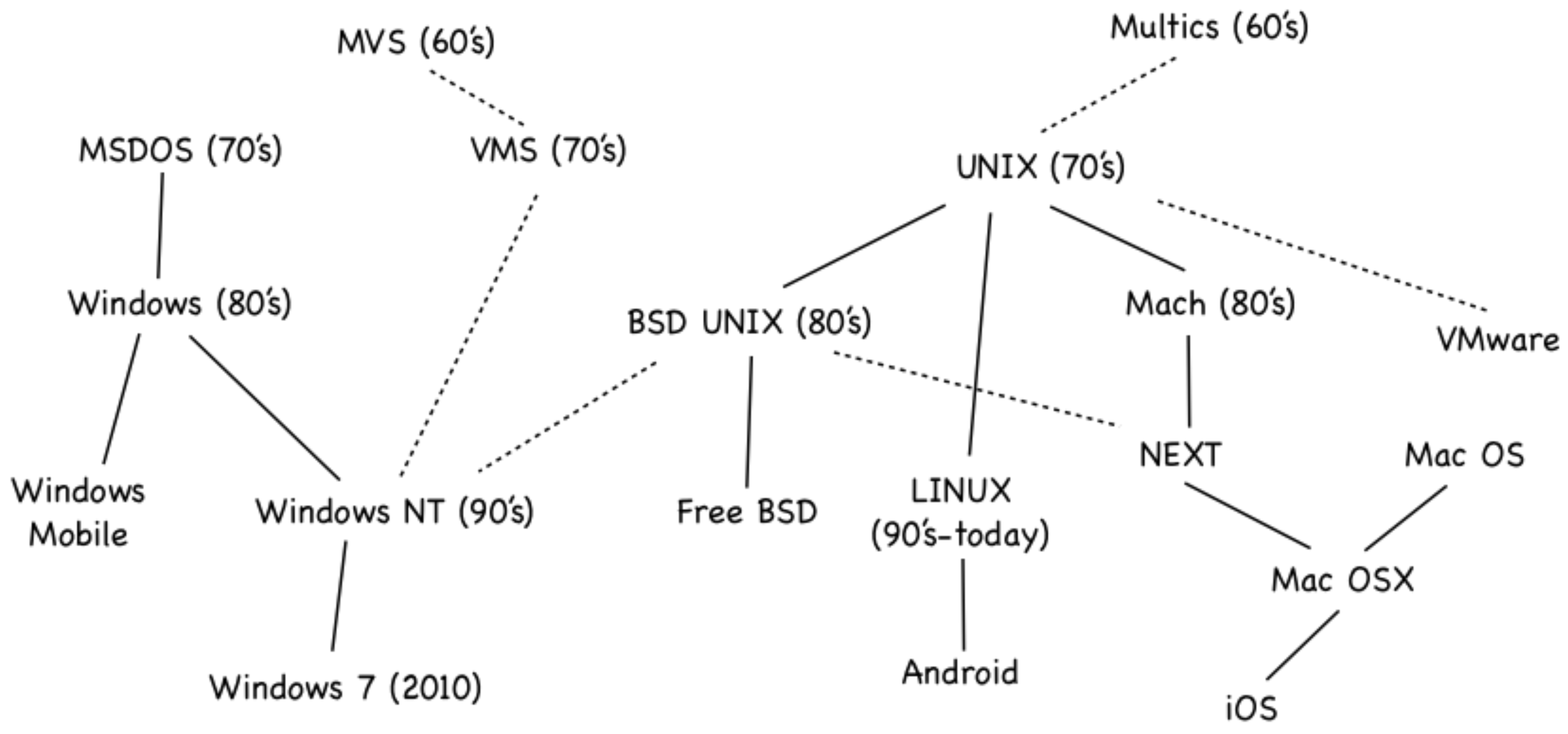
## Phase 3: Very Cheap Hardware, Very Expensive Humans

6. Personal computing
   - Computers are cheap, so give everyone a computer
   - Simplify OS by eliminating multiprogramming, concurrency, and protection
     - A subroutine library again!  (MSDos, MacOS)
     - Failed: humans are expensive, so don't waste their time letting programs crash each other!

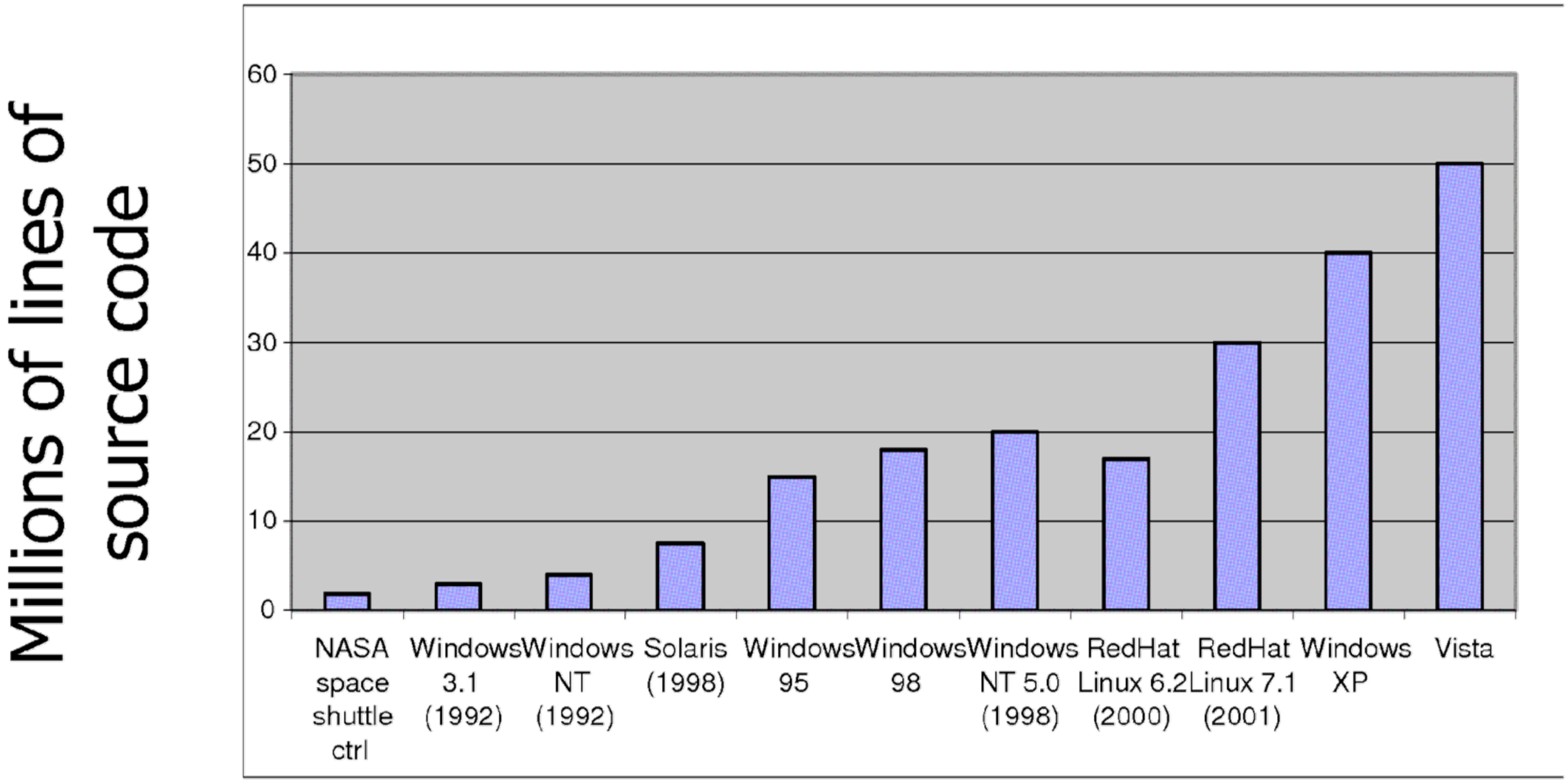
## Phase 3: Very Cheap Hardware, Very Expensive Humans

7. Parallel and Distributed Computing
   - Computers are SO cheap, give people a bunch of them!
   - In parallel systems, multiple processors are in the same machine, sharing memory, I/O devices, …
   - In distributed systems, multiple processors communicate via a network
   - Advantages: increased performance, increased reliability, sharing of specialized resources

# Genealogy of Modern Operating Systems



# Operating System Complexity



From MIT's 6.033 course
(I took it from John Kubiatowicz's CS162 course at Berkeley)

## Dual Mode Execution
## (and some Processes)

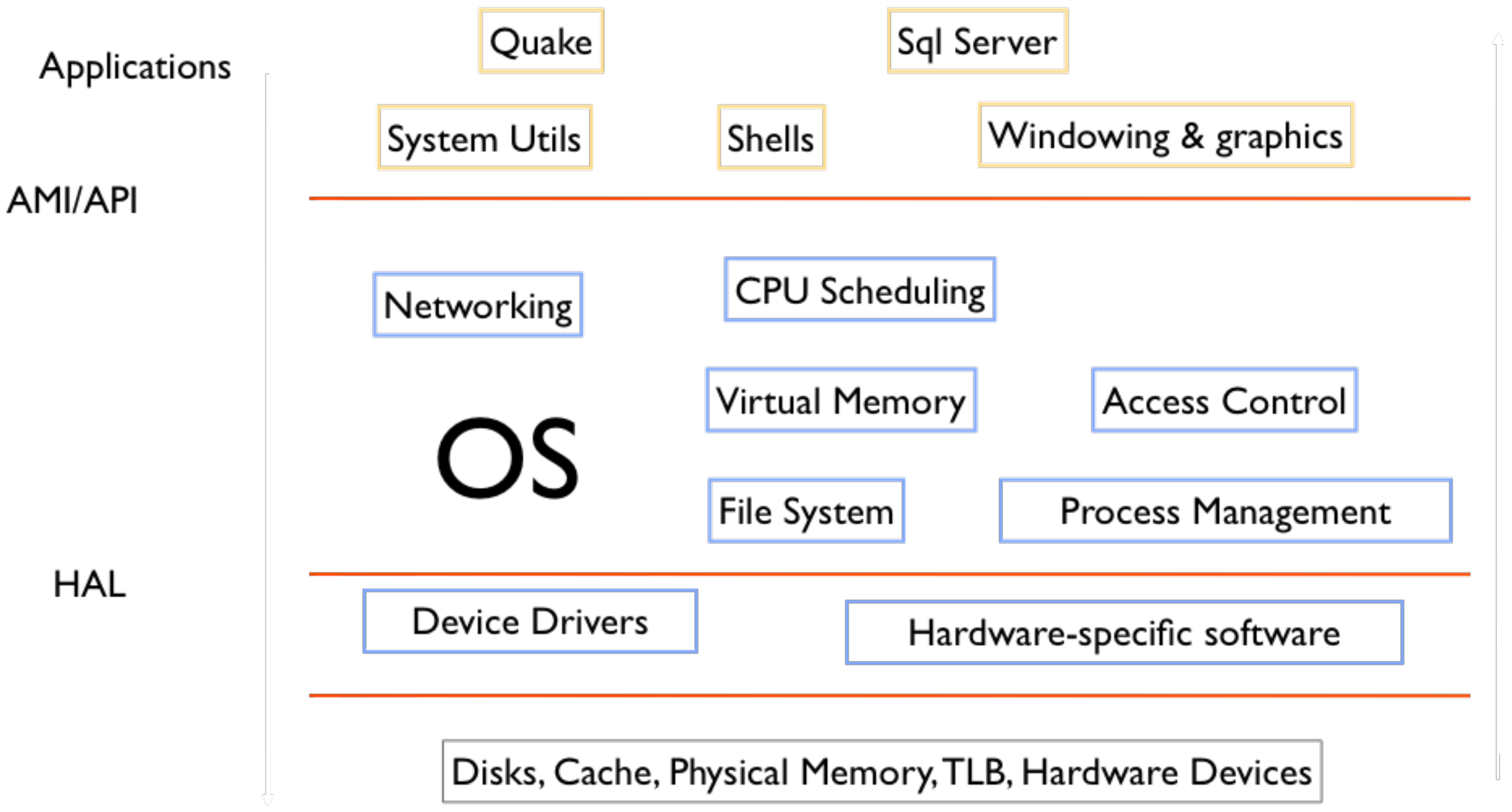
## OS Interfaces

Last time, we saw that the OS has three interfaces

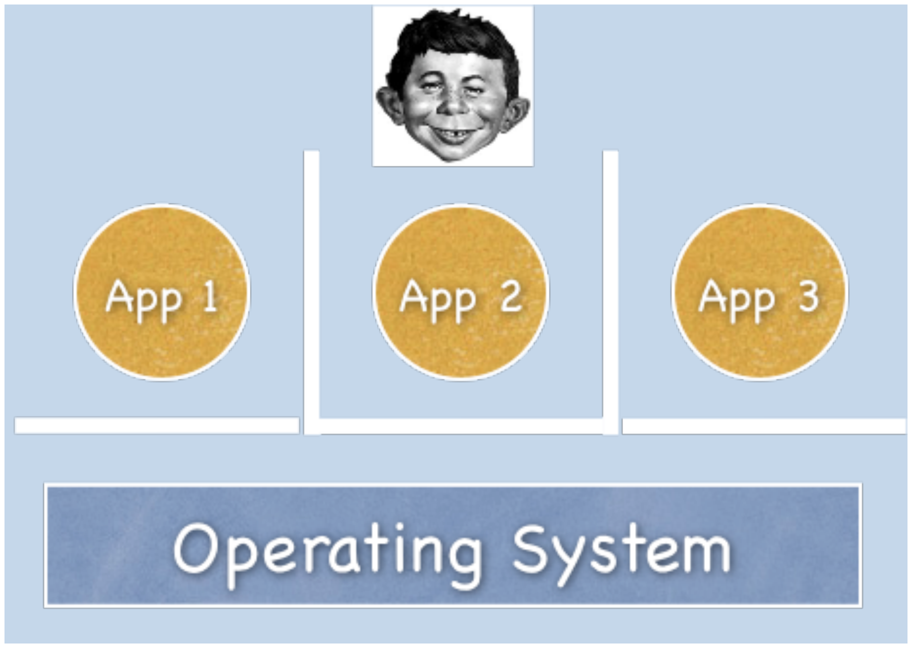
- Abstract Machine Interface (AMI)
  - between OS and apps: API + memory access model + legally executable instructions
- Application Programming Interface (API)
  - function calls provided to apps
- Hardware Abstraction Layer (HAL)
  - abstracts hardware *internally to the OS*

*Why?*

# Logical OS Structure



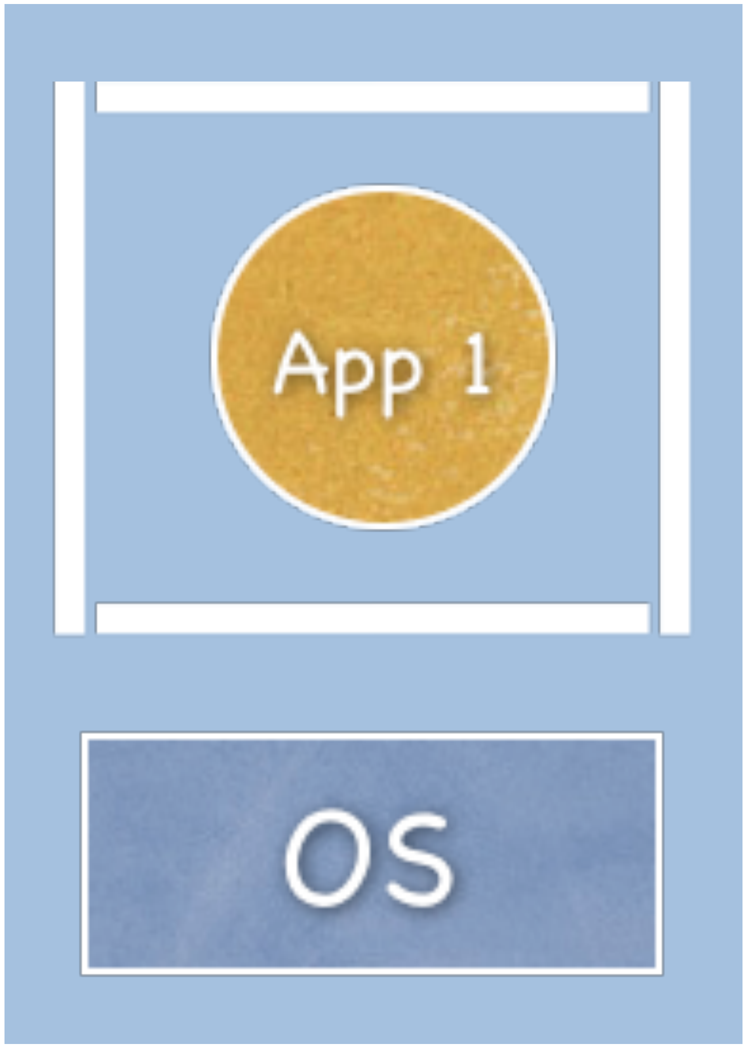
## If Applications Had Free Rein…



- Reading and writing memory, managing resources, accessing I/O... would you trust it all to him?

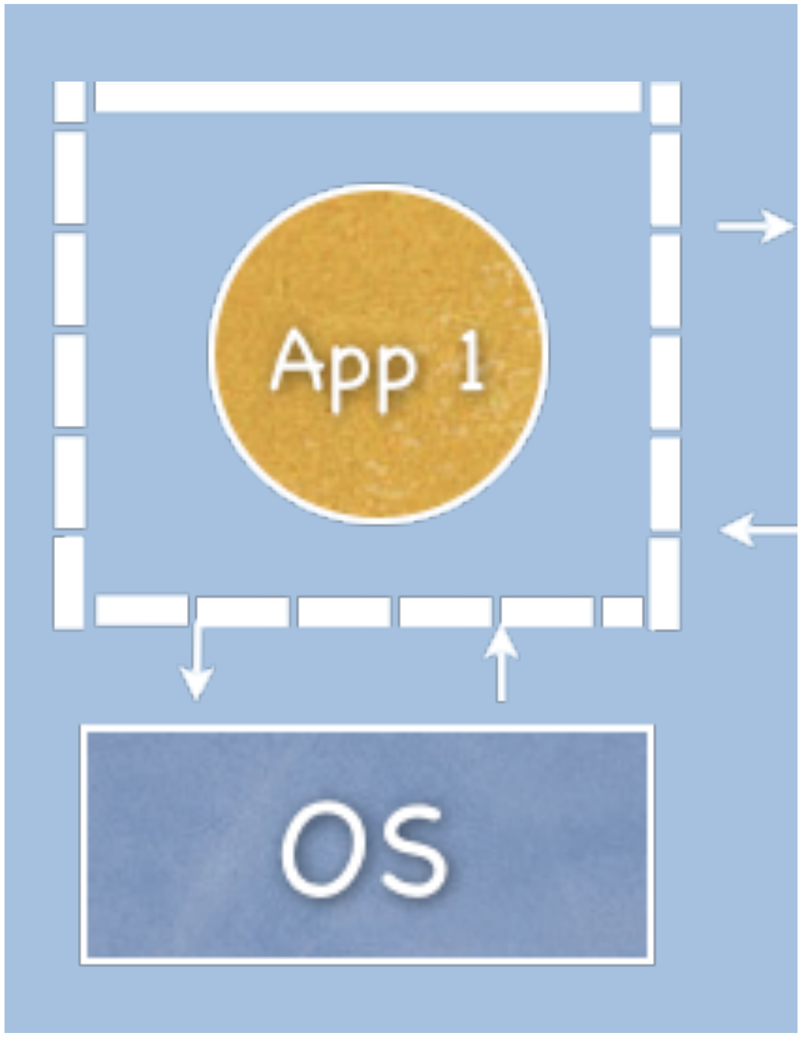
- Buggy apps can crash other apps
- Buggy apps can crash the OS
- Buggy apps can hog all resources
- Malicious apps can violate privacy of other apps
- Malicious apps can change the OS

## The Process: Boxes in the Application



- An abstraction for protection
  - the execution of an application program with restricted rights
- Restricting rights must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

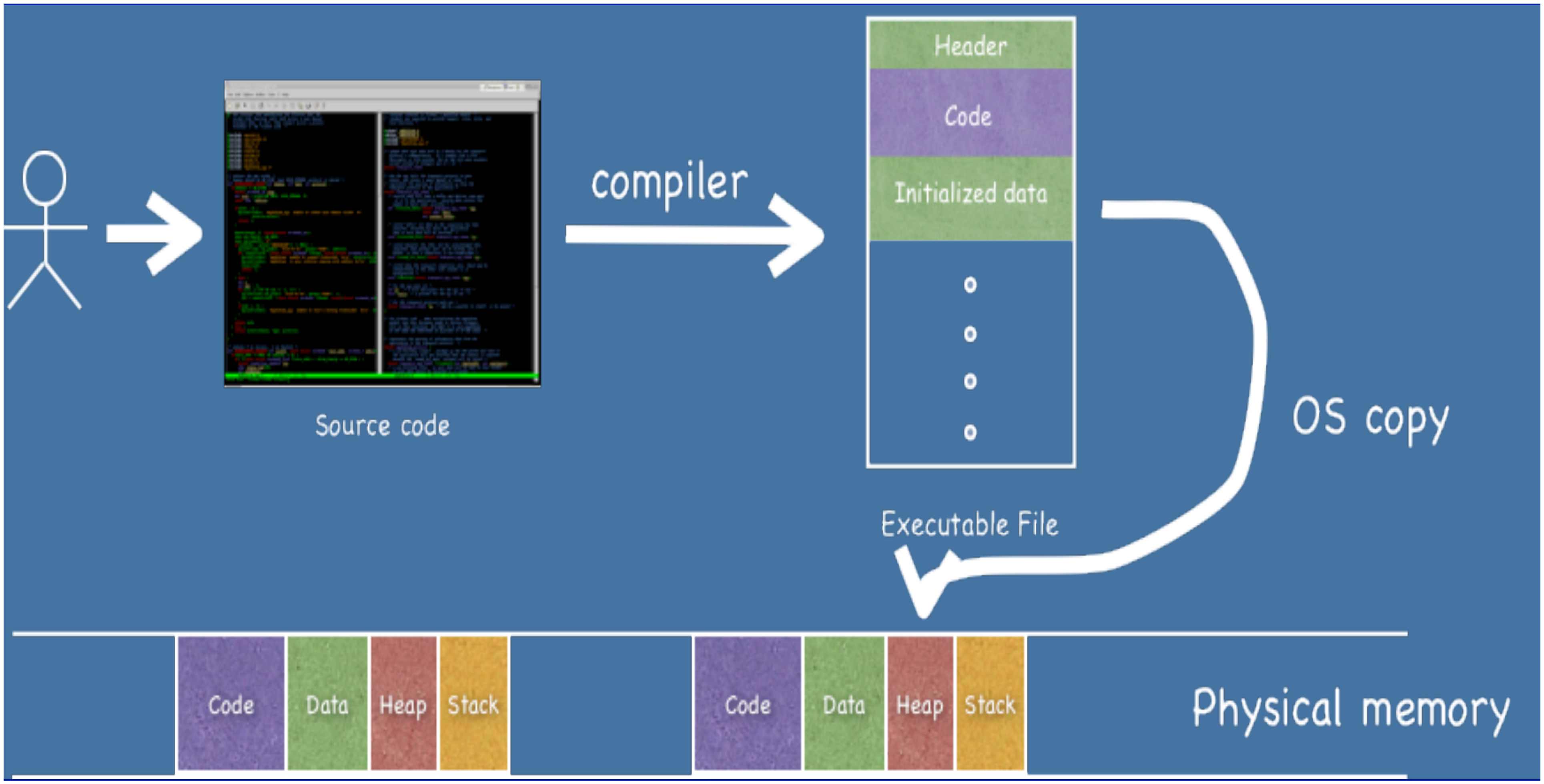
## The Process: Boxes in the Application



- An abstraction for protection
  - the execution of an application program with restricted rights
- Restricting rights must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

## What is a Process?

- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state
- A process is the basic unit of execution in an operating system
- Different processes may run different instances of the same program
  - e.g., my gcc and your gcc process both run the GNU C compiler
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution


compiler
Source code
Header
Code
Initialized data
Executable File
OS copy
Code
Data
Heap
Stack
Code
Data
Heap
Stack
Physical memory

## How can the OS enforce restricted rights?

- Easy: OS interprets each instruction!
  - Good solution?
    - No! Slow
  - Most instructions are safe: can we just run them in hardware?
- *Dual Mode Execution*
  - User mode: access is restricted
  - Kernel mode: access is unrestricted

# Dual Mode Execution

- Supported by the hardware
- Mode is indicated by a bit
  - in the processor status register
- In kernel mode, unrestricted rights
- In user mode, processor checks every instruction
  - Why?

## Possible Ways User Applications Could Compromise Other Applications

- Multiple Processes
  - Each process has its own memory
  - *Could access another process's memory*
- A Process, An OS, A CPU
  - CPU can only execute one process at a time
  - *Could take control of CPU and execute forever*
- Operating System has special privileges
  - Such as?
  - *Could perform an instruction that requires OS privilege*

- Solutions?

## Efficient Protection in Dual Mode Execution

Hardware must support at least three features:

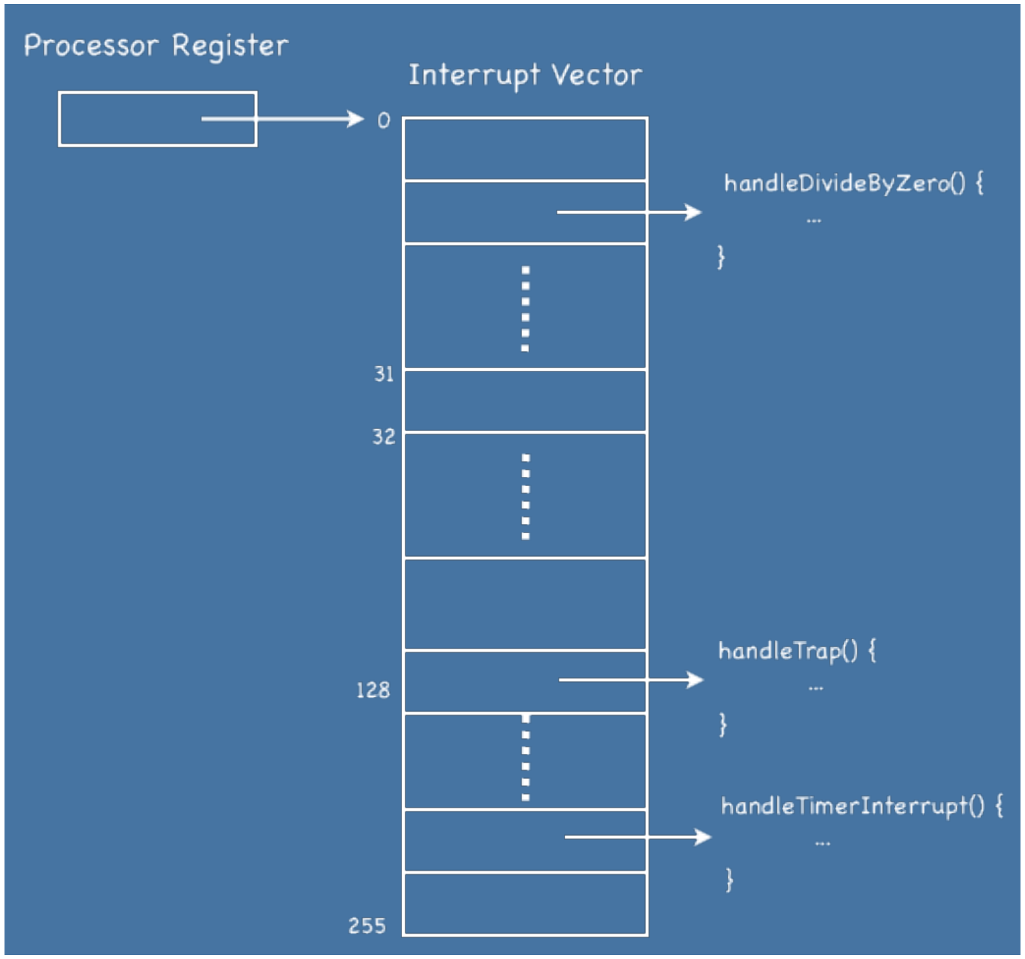
- Memory protection
  - In user mode, memory accesses outside a process' memory region are prohibited
  - We'll return to this later in the course
- Timer interrupts
  - Kernel must be able to periodically regain control from running process
  - Hardware timer can be set to expire after a delay and pass control back to the kernel
- Privileged instructions
  - Instructions only available in kernel mode
  - In user mode, no way to execute potentially unsafe instructions

# From User Mode to Kernel Mode…

- Exceptions
  - user program acts silly (e.g. division by zero)
  - or attempts to perform a privileged instruction
    - sometimes on purpose! (breakpoints)
  - synchronous (related to instruction that just executed)
- Interrupts (asynchronous exceptions)
  - something interrupts the currently executing process
    - timer, HW device requires OS service, …
  - asynchronous (not related to instruction that just executed)
- System calls/Traps
  - user program requests OS service
  - looks like a function call
  - synchronous

# User Mode to Kernel Mode: Details

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - system call?
  - interrupt? then which hardware device?
  - which exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



## Saving the State of the Interrupted Process

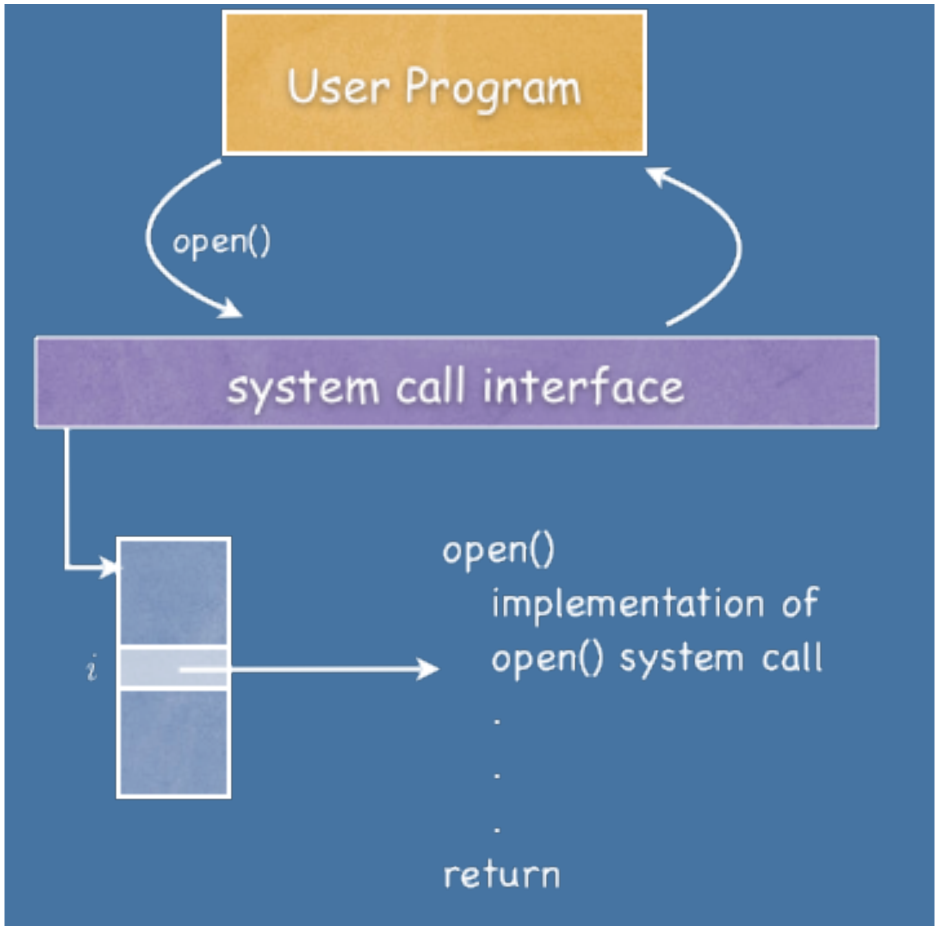
- Privileged hw register points to Exception Stack
  - on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack <u>before </u>handler runs. Why?
  - then handler pushes the rest (pushad on x86)
  - On return, do the reverse (popad on x86)
- Why not use user-level stack?
  - reliability: even if user's stack points to invalid address, handlers continue to work
  - security: kernel state should not be stored in user space (or could be read/written)
- One interrupt stack per processor/process/thread

## Interrupt Masking

- What happens if an interrupt occurs while we are running an interrupt handler?
  - can’t reset kernel stack pointer to point to base of kernel’s interrupt stack… it’s in use!
- Privileged instruction is used to disable (defer) interrupts

# System Calls

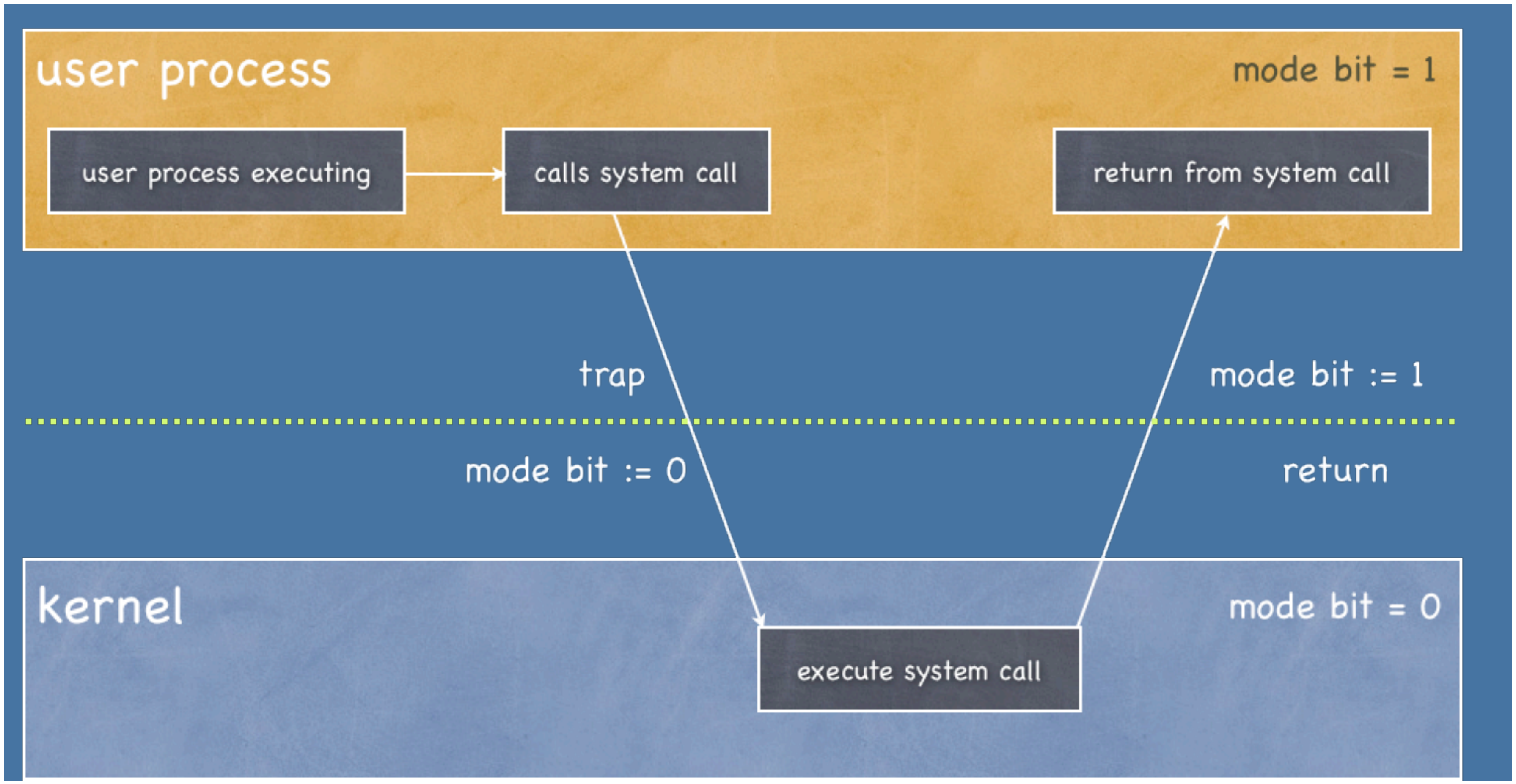
- A request by a user-level process to call a function in the kernel is a *system call*
  - Examples: read(), write(), exit()
- The interface between the application and the operating system (API)
- Mostly accessed through *system-level libraries*
- Parameters passed according to calling convention
  - registers, stack, etc



## System Calls: A Closer Look

- User process executes a trap instruction
- Hardware calls the OS at a pre-specified location
- OS then:
  - identifies the required service and parameters (e.g. open(filename, O_RDONLY))
  - executes the required service
  - sets a register to contain the result of call
  - Executes an RTI instruction to return to the user program
- User program receives the result and continues

## System Calls



## iClicker Question

The interrupt vector is used to determine the action taken by the OS when:

A. An exception occurs
B. An interrupt occurs
C. A system call is executed
D. All of the above
E. None of the above

## Switching Back!

- From an interrupt, just reverse all steps!
  - asynchronous, so not related to executing instruction
- From exception and system call, increment PC on return
  - synchronous, so you want to execute the *next* instruction, not the same one again!
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw

## Switch to User Mode Also Occurs When…

- A process is created
  - OS copies program in memory, sets PC and SP, toggles mode
- Switch to different process
  - OS loads PC, SP, registers and other process information; toggles mode

## Kernel vs. User Mode: Privileged Instructions

User processes may not:

- address I/O directly
- use instructions that manipulate OS memory (e.g., page tables)
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

But in kernel mode, the OS does all these things.

Executing a privileged instruction while in user mode causes a processor exception…

...which passes control to the kernel

# Summary

- Application of ideas has changed over time
  - Every design decision has driven change in other designs
- Operating System provides protection through dual-mode execution
  - Mode changes through interrupts (e.g., time slice), exceptions, or system calls.
  - A status bit in a protected processor register indicates the mode
  - Privileged instructions can only be executed in kernel mode

## Announcements

- TAs, contact information, etc are in the online syllabus
- Discussion sections begin this week! (Remember, you must attend your own!)
- Homework 1 due Friday