Chapters 1 and 2:
Kernel Mode, Traps, System Calls, Multitasking

CSCI 3753

Instructor: Chris Womack

CSCI 3753 Announcements

- PA #1 due Monday Jun 19th by 2355 (1155pm)
  - Add a system call to Linux
  - Will be discussed in recitation and demos
- Will release a problem set next week
- Read chapters 1-2 and 13 (I/O Systems) in the textbook

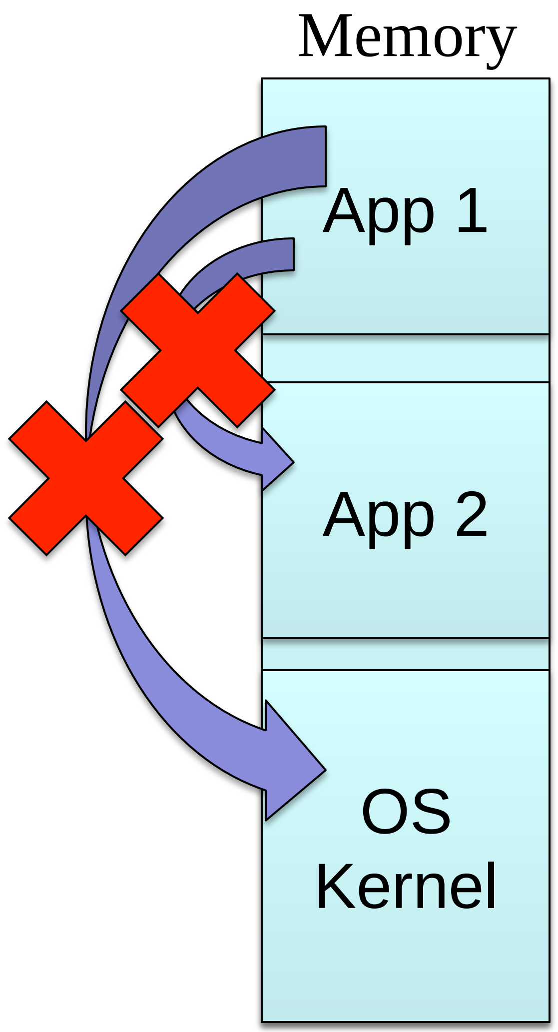
Recap…

- An OS is a software layer that sits between applications and I/O devices
  - Main Goals: Abstraction, Arbitration, & Protection
- An OS consists of many components
  - Memory manager, Scheduler, File System, Device Management, Network Stack, etc.
- Linux is a *monolithic* kernel – complex, contains many components
- Mach OS is a *microkernel* – kernel only contains scheduler, memory manager, and messaging

Protection in Operating Systems

- One of an Operating System's main goals is Protection
  - Protect applications from each other
  - Protect OS from applications

1. Prevent applications from writing into privileged memory
   - e.g. of another app or OS kernel
2. Prevent applications from invoking privileged functions
   - e.g. OS kernel functions



Memory Protection via Virtual Memory

- Recall that an executable only has virtual addresses
- These are translated into physical memory addresses at run time by a page table
- OS controls the page table
- Difficult for a program to write into another program or kernel’s address space
  – Any virtual address given to memory manager is translated into a non-conflicting physical address
  – Access to the “wrong” memory causes a page fault
  – Caveats: shared libraries, …

Protecting the OS via a Mode Bit

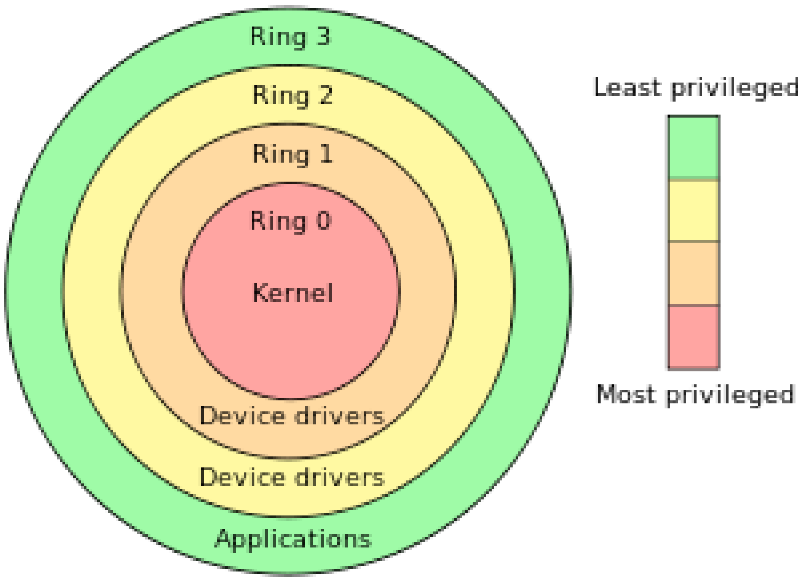
- Processors include a hardware *mode* bit that identifies whether the system is in *user* mode or *supervisor/kernel* mode
  - Requires extra support from the CPU hardware for this OS feature
- Prevents applications from executing privileged instructions
  - Can’t reset time slice register, or change interrupt vector register, …
- Embedded microcontrollers don’t have mode bit
- 80286 added mode bit in 1982

Kernel Mode vs User Mode

- Supervisor or kernel mode (mode bit = 0)
  - Can execute all machine instructions, including privileged instructions
  - Can reference all memory locations
  - Kernel executes in this mode
- User mode (mode bit = 1)
  - Can only execute a subset of non-privileged instructions
  - Can only reference a subset of memory locations
  - All applications run in user mode

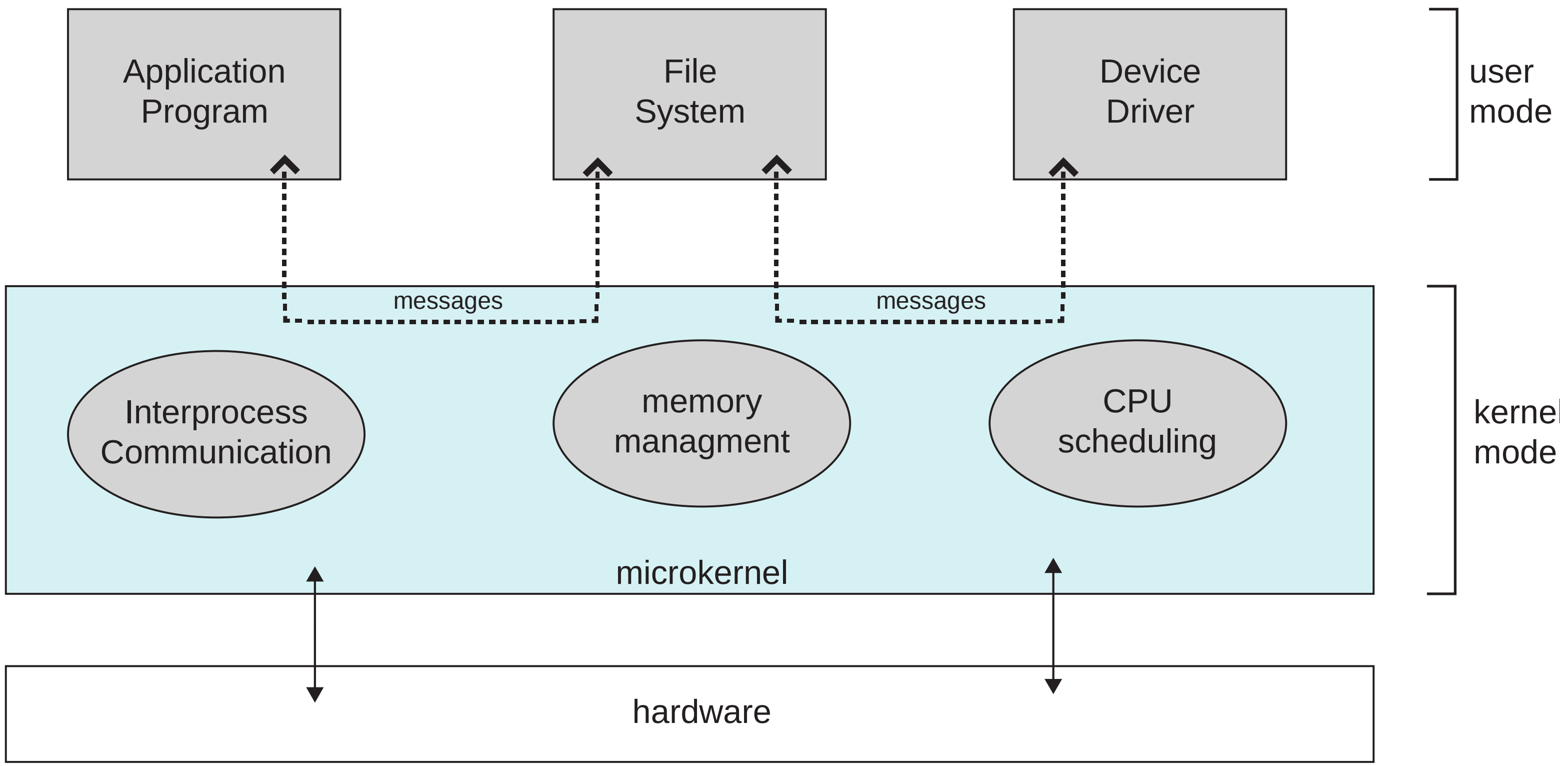
Multiple Rings/Modes of Privilege

- Intel x86 CPUs support four modes or rings of privilege
- Common configuration:
  - OS like Linux or Windows runs in ring 0 (highest privilege), Apps run in ring 3, and rings 1-2 are unused



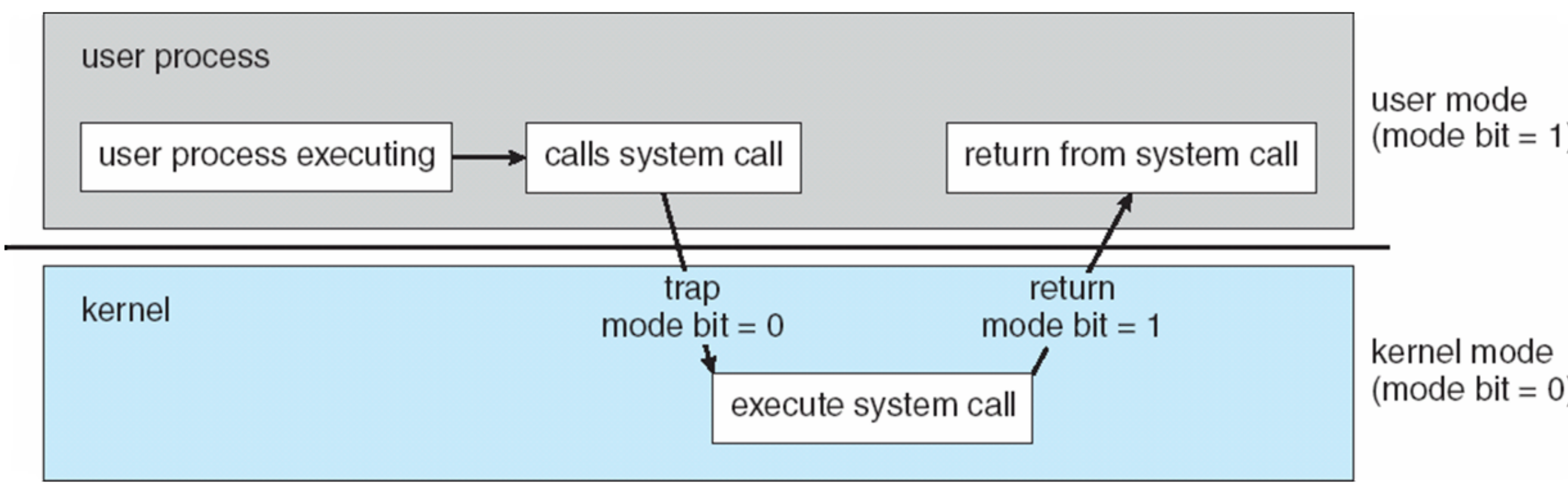
- Virtual machines (one possible configuration)
  - VM's hypervisor runs in ring 0, guest OS runs in ring 1 or 2, Apps run in ring 3

Microkernel System Structure



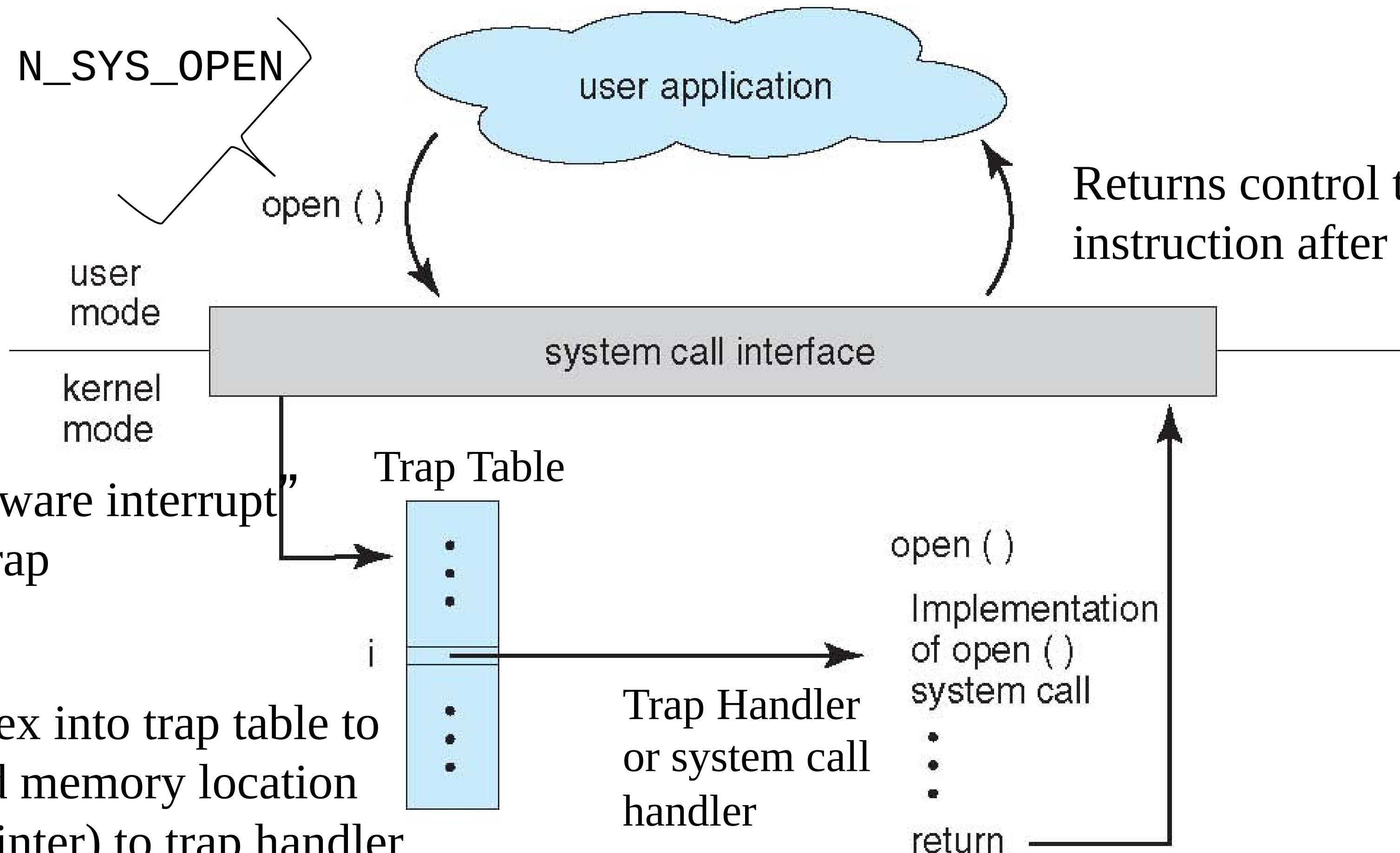
System Calls:
How Apps and the OS Communicate

- The `trap` instruction is used to switch from user to supervisor mode, thereby entering the OS"
  - `trap` sets the mode bit to 0"
  - On x86, use INT assembly instruction (more recently SYSCALL/SYSENTER)"
  - mode bit set back to 1 on return"
- Any instruction that invokes `trap` is called *a system call!*
  - There are many different classes of system calls"



API – System Call – OS Relationship

```
open() {
…
trap N_SYS_OPEN
…
}
```



Returns control to the next instruction after `trap`!

"software interrupt" = a trap

Index into trap table to find memory location (pointer) to trap handler

Trap Handler or system call handler

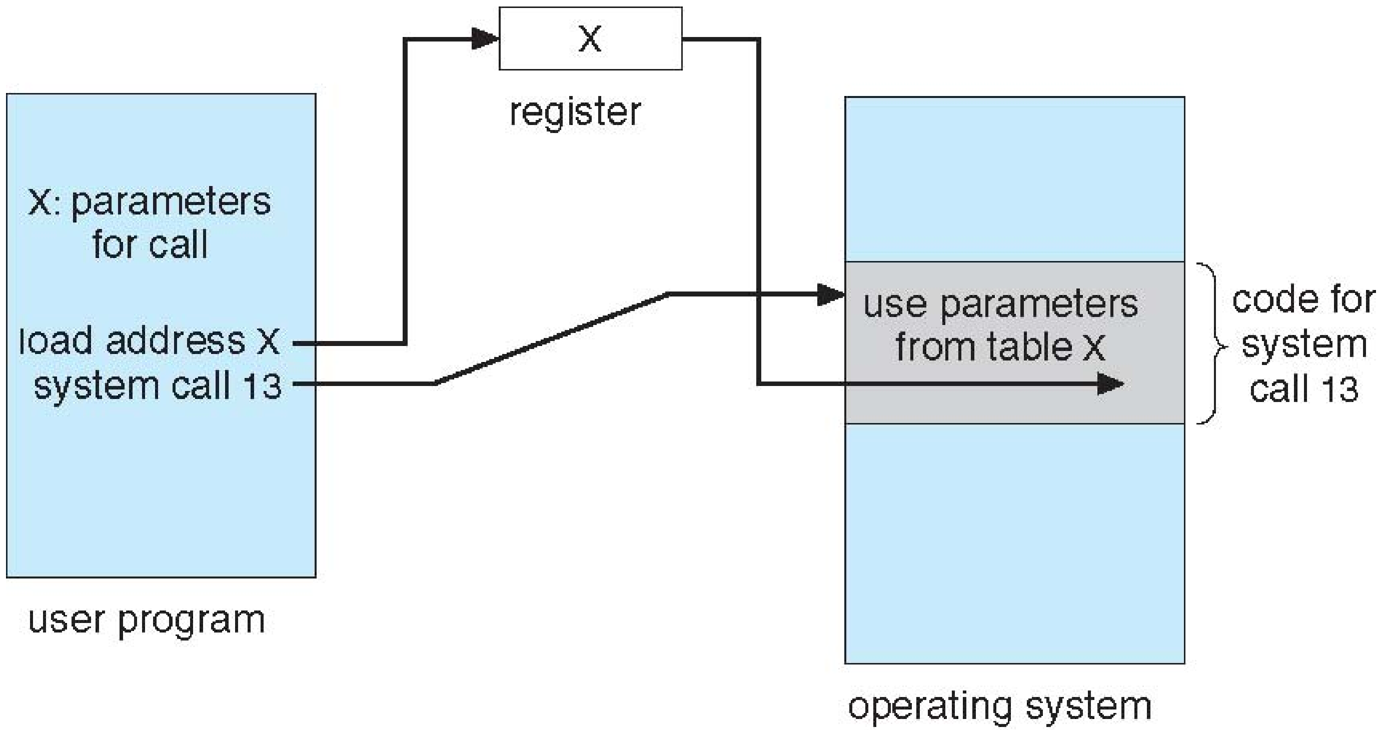
Trap Table

- The process of indexing into the trap table to jump to the trap handler routine is also called *dispatching*
- The trap table is also called a *jump table* or a *branch table*
- "A trap is a *software interrupt*"
- Trap handler (or system call handler) performs the specific processing desired by the system call/trap

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call"
  - Exact type and amount of information vary according to OS and call"
- Three general methods used to pass parameters to the OS"
  1. Simplest: pass the parameters in *registers!*
     - ! In some cases, may be more parameters than registers"
  2. *Pointer:* Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register "
     - ! This approach taken by Linux and Solaris"
  3. Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the *stack* by the operating system"
  - Block and stack methods do not limit the number or length of parameters being passed"

Parameter Passing via Table



Classes of System Calls Invoked by `trap`!

system call interface

Process control	File Management	Device Management	Information Management	Comm-unications

Process control

- end, abort
- load, execute
- fork, create, terminate
- get attributes, set
- wait for time
- wait event, signal event
- allocate memory, free

File Management

- create, delete
- open, close
- read, write, reposition
- get attributes, set

Device Management

- request device, release
- read, write, reposition
- get attributes, set
- logically attach or detach devices

Note Similarity

Information Management

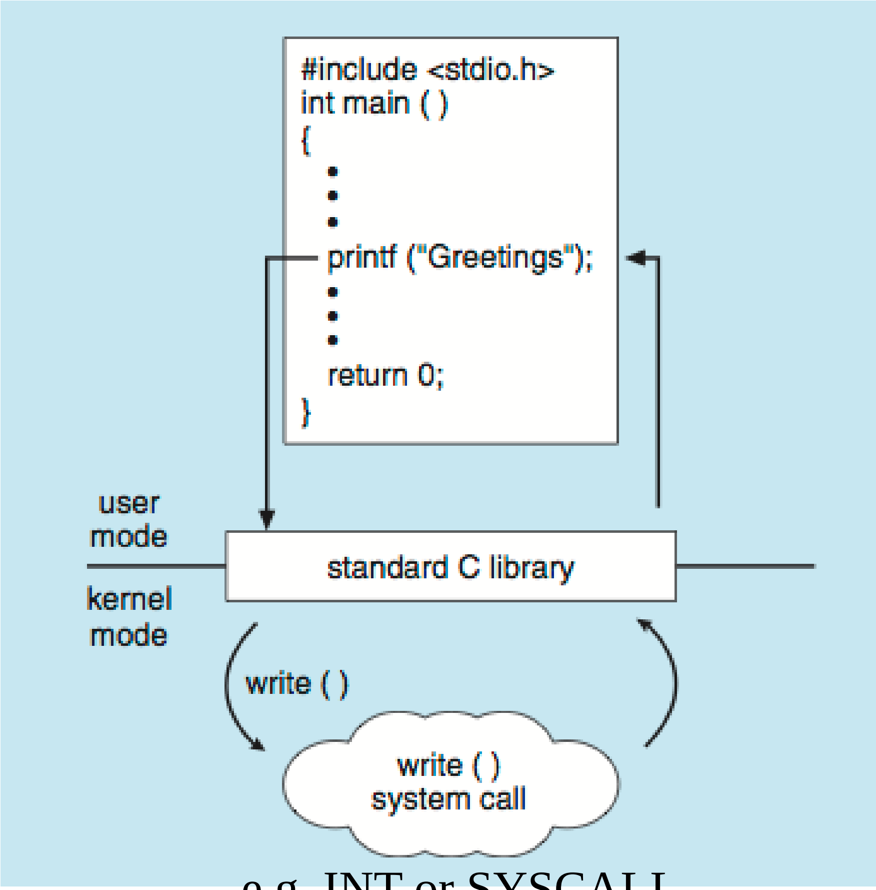
- get time/date, set
- get system data, set
- get process, file, or device attributes, set

Communications

- create connection, delete
- send messages, receive
- transfer status info
- attach remote devices, detach

Standard C Library Example

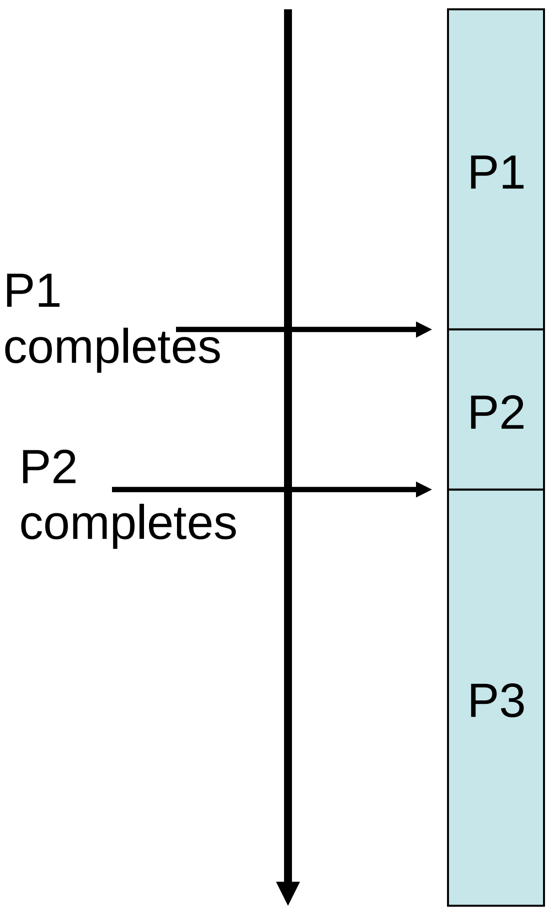
- C program invoking printf() library call, which calls write() system call"



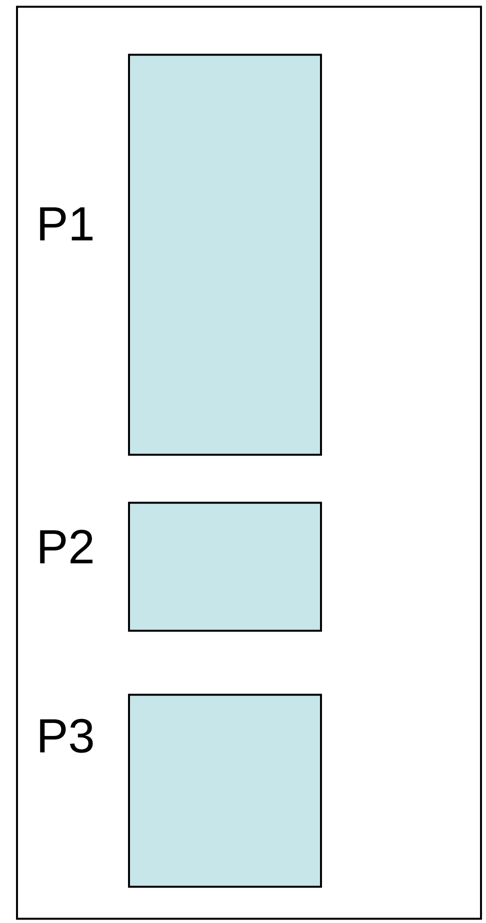
e.g. INT or SYSCALL

How does an OS support multiple applications? Batching of jobs

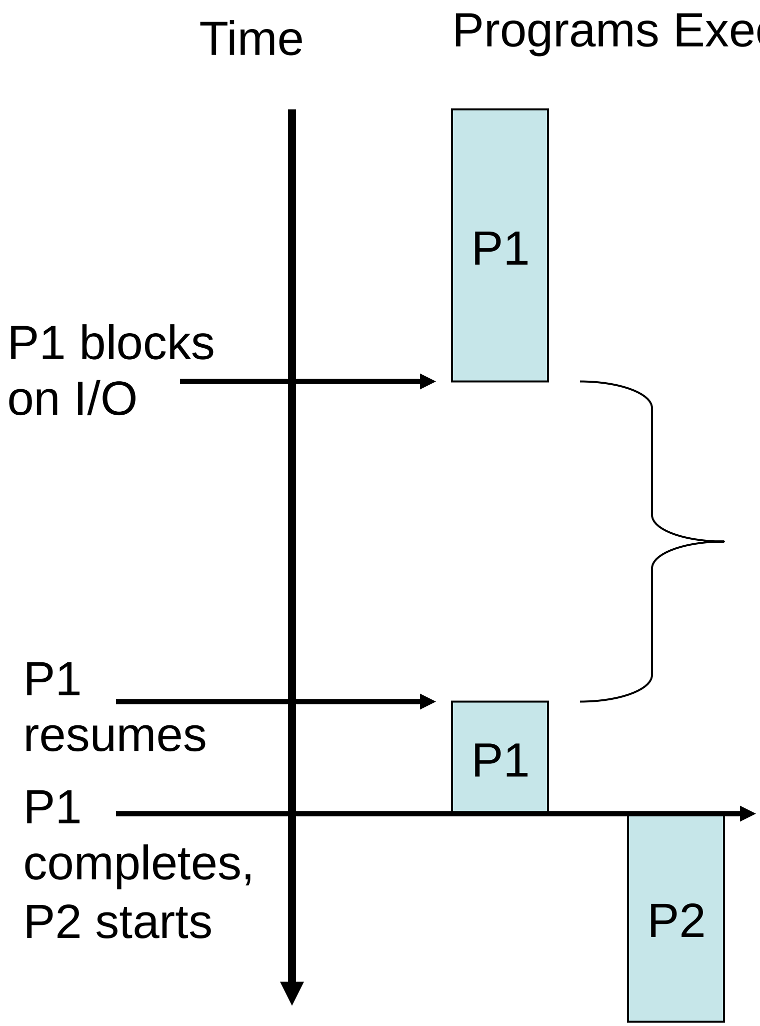
Time Programs Executing on CPU



Let programs execute *sequentially*. When P1 is done, P2 takes over, and so on. In this case, P1, P2, and P3 are all *compute-bound*, i.e. executing continuously on the CPU until they' re done



What happens if some programs are I/O-bound?

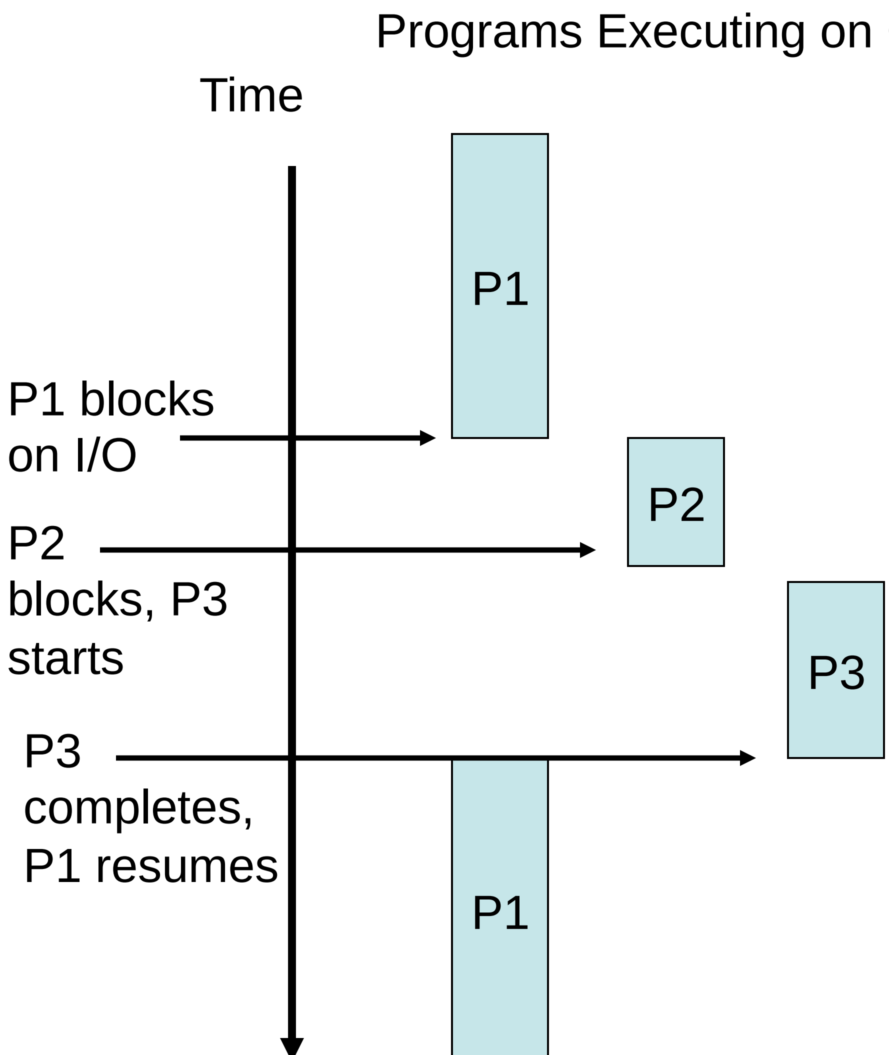


If P1 is I/O bound, then it blocks on I/O.
But CPU is Idle! => Poor Utilization,
maybe billions of wasted instruction cycles

Limitations of Sequential Execution

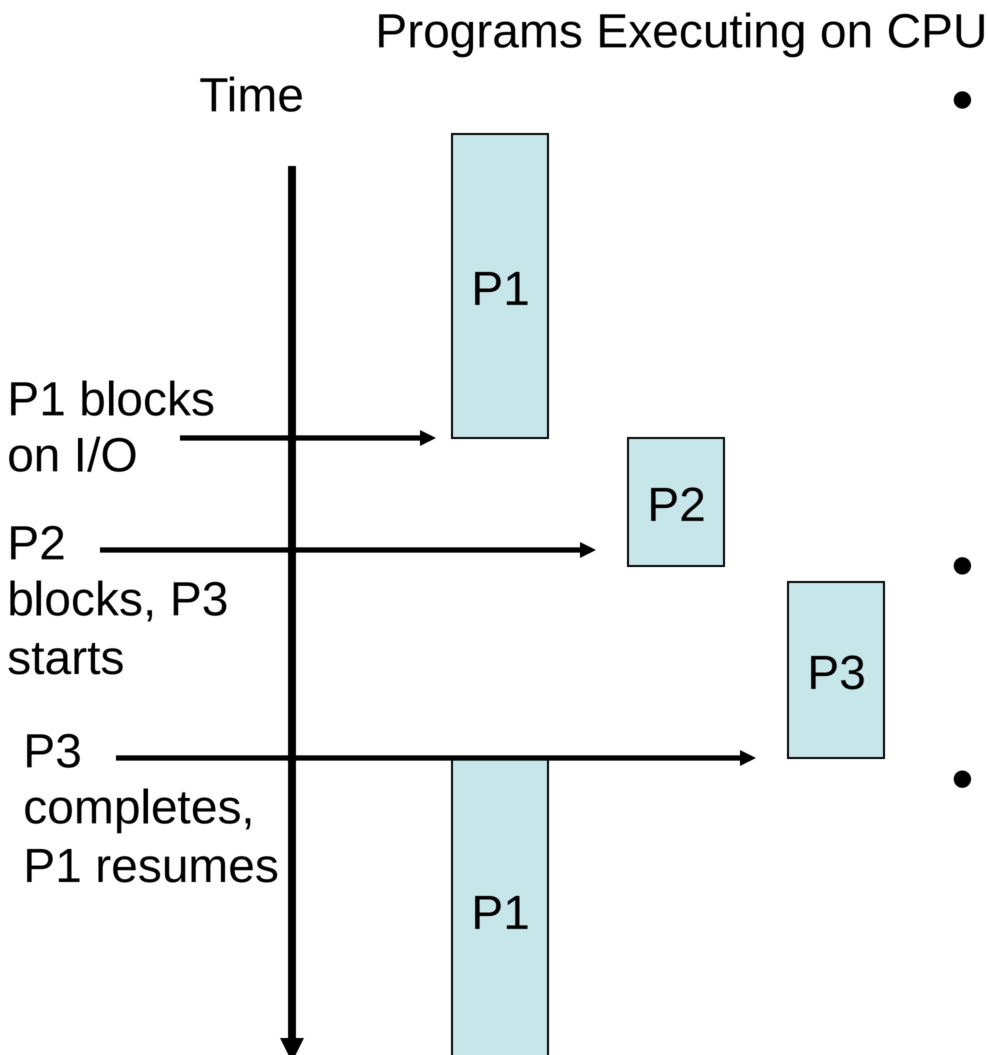
- Program P1 blocks waiting for something to complete
  - waiting on I/O, e.g. waiting for a disk write to complete, or waiting to read data from a keyboard
    - I/O can be very slow compared to CPU speed
  - then CPU is idle for potentially billions of cycles!
- Better if CPU switches to another program P2 and begins executing P2
  - better utilization of the CPU for *I/O-bound* programs, e.g. shells, editors (talk to keyboard and disk)

Multiprogramming



- when CPU is idle (e.g. blocked on I/O), run another program => improved CPU utilization
- OS Scheduler switches CPU between multiple executing programs
  - programs share CPU
- OS *time-multiplexes* CPU between executable programs

Multiprogrammed Batch Systems



- Submit your program, called a *job*, into a job queue,
  - When CPU is available, OS executes your job, running to completion
  - Great for long-running jobs like simulations
- But turnaround time is long
  - Takes awhile to find out your compilation errors & bugs
- For what kind of programs is multiprogramming not suitable?
  - Interactive applications!

Limitations of Multiprogramming

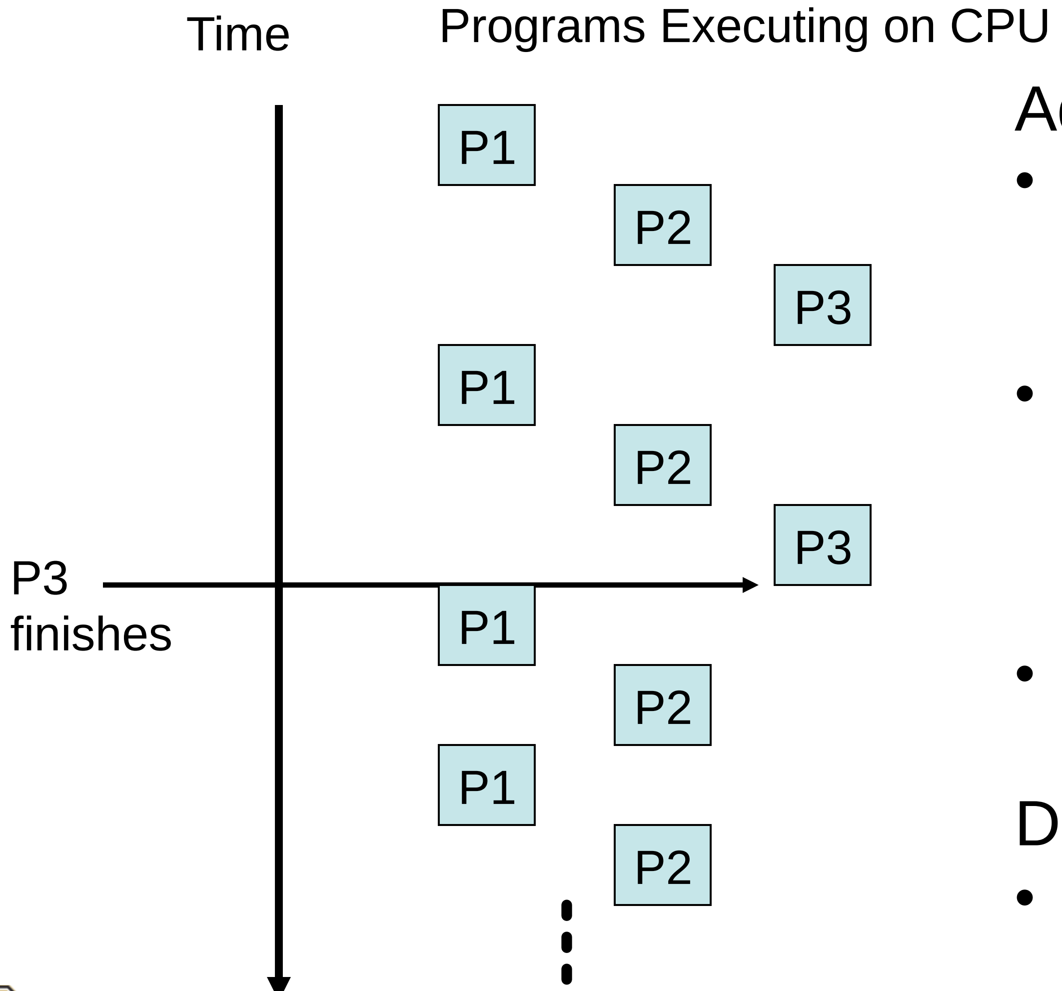
- Batch jobs are very non-interactive
  - Don't support a shell application for example
- What we want to see
  - design jobs to yield much sooner than an I/O block, to give the impression of interactivity
  - If one of the programs was a shell, then it appears to the human user as if the computer is instantly responsive
  - In the small time segment a shell is given, it can draw a character on the screen that you've just typed => appearance of real-time interactivity

Multitasking

- CPU rapidly switches between multiple programs
  - Each program gets a small slice of the CPU, then yields the CPU to another program
  - This switching happens often enough that each program still gets a fair percentage of the CPU, and can still make significant progress
  - At the same time, interactive programs like shells are now supported – **this was a big innovation**

Multitasking

- CPU rapidly switches between programs



Advantages:

- efficient CPU usage, i.e. no idle time if one program blocks
- Better isolation – a misbehaving program can't stop other programs from executing*
- **supports interactivity**

Disadvantage:

- context switch overhead

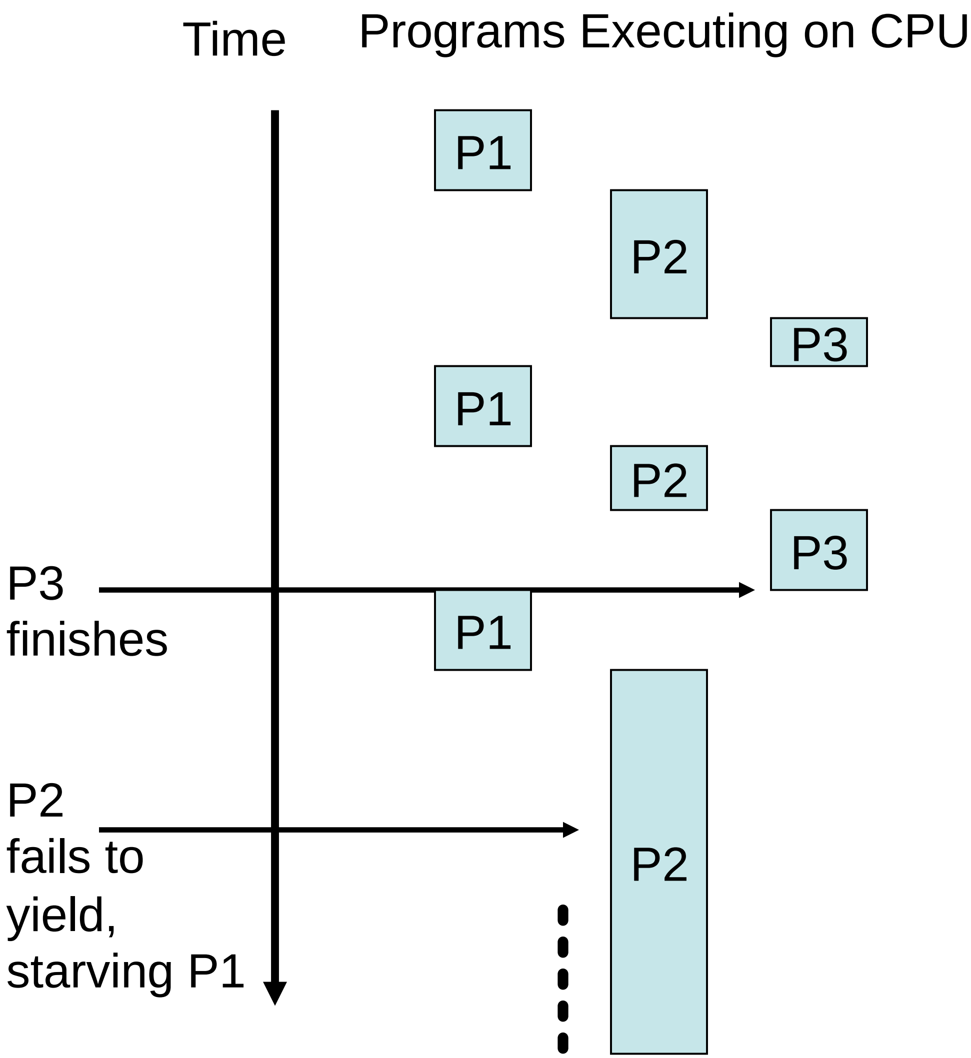
Context Switch Overhead

- Switching from one program to another is called a context switch
- there is overhead due to this context switching
  - With each context switch, the CPU has to *save* the current state of application 1 (its PC, IR, data registers, stack pointer, etc.),
  - and then *load* the state of the new application 2 when app 2 was last switched out (new PC, new IR, new data registers, new stack pointer, etc.)
- All of this takes time - typical overhead = 1 µs,
  - No useful work can be done by program during a context switch

Cooperative Multitasking

- How does an OS achieve Multitasking?
  - Cooperative multitasking
  - Preemptive multitasking
- In cooperative multitasking, programs quickly and *voluntarily* yield CPU before they' re done
  - Like batch mode multiprogramming, except it' s more fine-grained than jobs and yielding is more explicit than opportunistic
  - Early OSs did this (Windows 3.1, Mac OS 9.*)

Cooperative Multitasking



Programs rapidly and voluntarily yield CPU

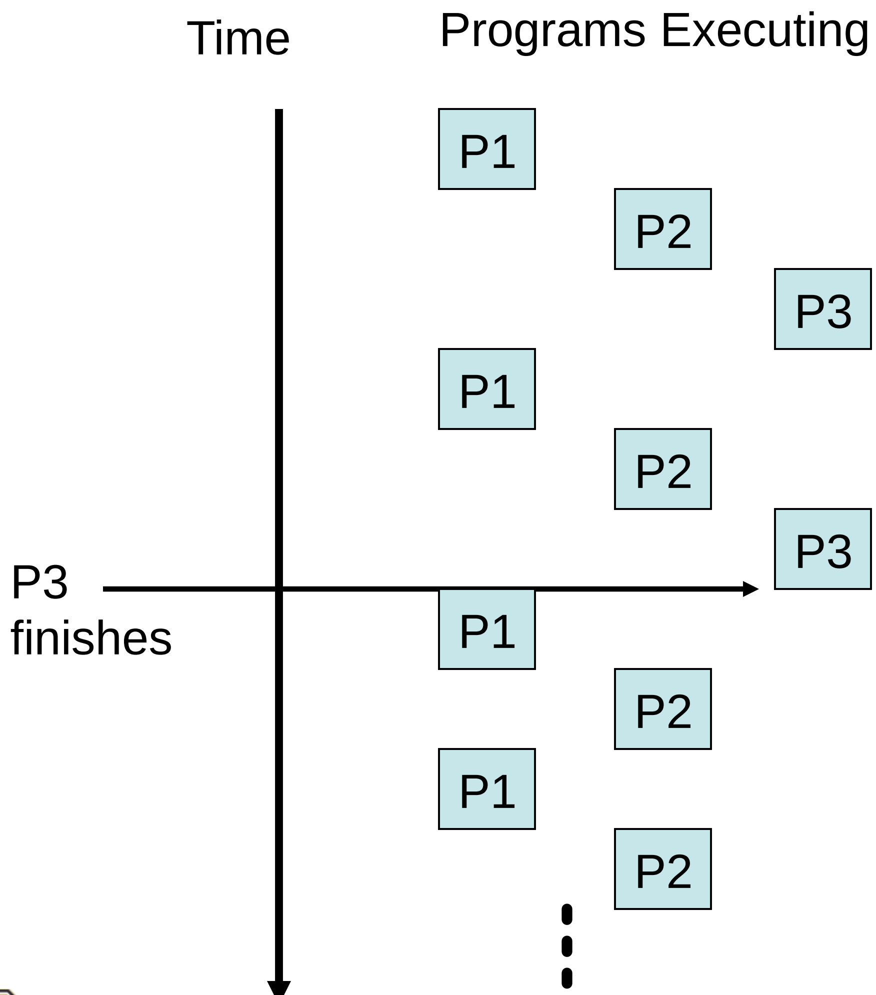
So OS *rapidly* switches between multiple executing programs, giving the appearance of interactivity

Main problems?

- Problem: *poor fault isolation*, i.e. if a program fails to yield (unintentionally or deliberately), other programs can’t run and will be starved.
- Programs have to be rewritten to manually yield control throughout the code.

Preemptive Multitasking

- Force programs to release CPU – better isolation



OS forces rapid switching between multiple executing programs by setting a hardware *timer*

The timer periodically interrupts CPU execution, invoking OS Scheduler to schedule the next program

Preemptive Multitasking Time Slices

- Each program is given a short interval on the CPU called a *time slice*
  - Typical time slice is 30 ms
  - The length of the default time slice is a compile time option for the OS kernel
  - Also some schedulers vary the time slice according to the priority of the process, e.g. higher priority processes get a longer time slice.

Preemptive Multitasking Interrupts

- Timer interrupt fires periodically
- This suspends execution of the currently executing program and returns control to the OS scheduler
- The scheduler decides the next program to execute and loads it, then passes control to it
  - Switching from one program to another is called a *context switch*
  - there is overhead due to this context switching
  - Overhead is only 1 µs per time slice of 30 ms, so overhead % = 1/30000 = .003 %

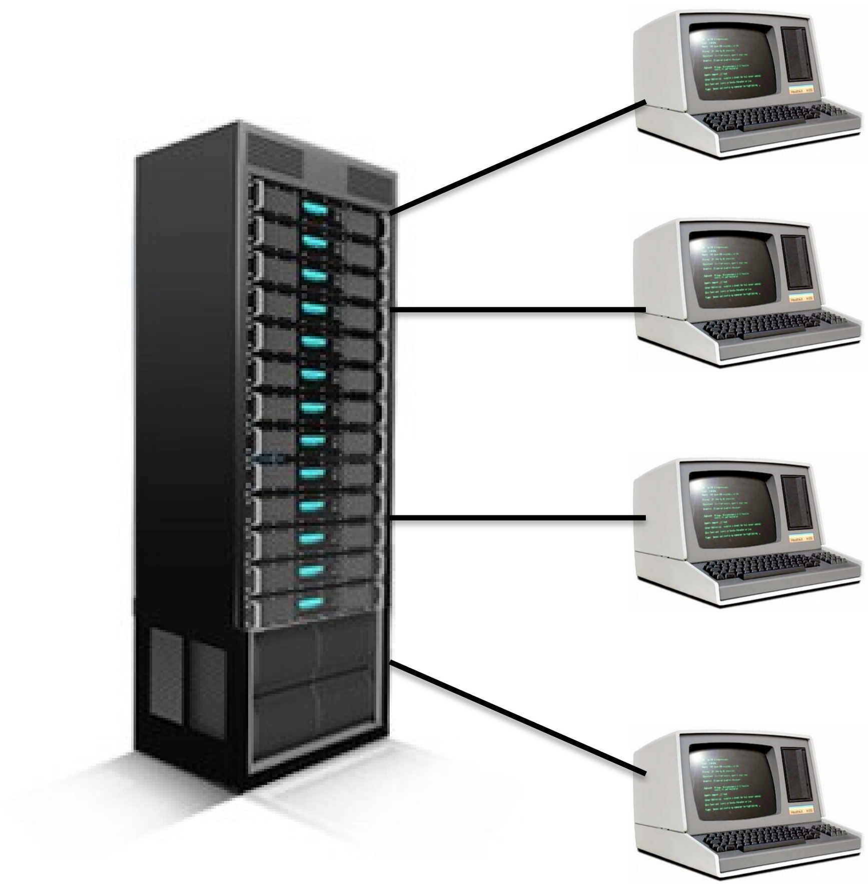
Preemptive Multitasking Benefits

- Efficient sharing of CPU
  - Programs blocked on I/O don't block other programs
- Fault isolation
  - Programs are forced to yield, so can't block other programs
- Support for long-running jobs
- Support for interactive programs

Preemptive Multitasking History

- Early computers were big mainframes
  - But wanted to share the CPU of a mainframe not just between different batch jobs, but also between different *human users*
  - Interactive time sharing systems were developed
- Time-sharing examples
  - multiple processes sharing time locally on a CPU
  - multiple user terminals remotely sharing processing time with a central server
    - keystroke delay during heavy loads (before a class assignment was due) could be significant and non-interactive
- Basically all modern operating systems are preemptively multitasked
  - Linux, BSD Unix, Windows NT/XP/Vista, Mac OS X 10.*,

Time-Sharing Computers



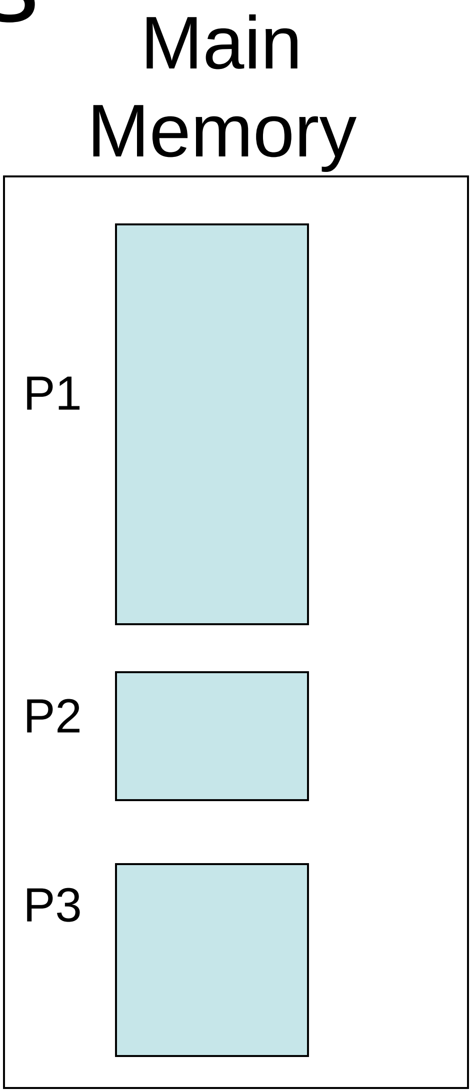
A set of “dumb” terminals are slaves to the master computer

The master multiplexes the CPU rapidly between the different terminals, including rendering characters on their screens!

Delay between keystrokes, so type ahead!

Multitasking & Abstract Machines

- CPU is time-multiplexed between multiple programs
  - Programs share the CPU
- Memory is space-multiplexed between multiple programs
  - programs share RAM
- Each program thus sees its own *abstract machine* (provided by OS)
  - it has its own "private" (slower) CPU
  - it has its own "private" (smaller) memory



Supplementary Slides

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Types of System Calls

- Process control"
  - create process, terminate process"
  - end, abort"
  - load, execute"
  - get process attributes, set process attributes"
  - wait for time"
  - wait event, signal event"
  - allocate and free memory"
  - Dump memory if error"
  - **Debugger** for determining **bugs, single step** execution"
  - **Locks** for managing access to shared data between processes"

Types of System Calls

- File management"
  - create file, delete file"
  - open, close file"
  - read, write, reposition"
  - get and set file attributes"
- Device management"
  - request device, release device"
  - read, write, reposition"
  - get device attributes, set device attributes"
  - logically attach or detach devices"

Types of System Calls (Cont.)

- Information maintenance"
  - get time or date, set time or date"
  - get system data, set system data"
  - get and set process, file, or device attributes"
- Communications"
  - create, delete communication connection"
  - send, receive messages if **message passing model** to **host name** or **process name!**
    - From **client** to **server!**
  - **Shared-memory model** create and gain access to memory regions"
  - transfer status information"
  - attach and detach remote devices"

Types of System Calls (Cont.)

- Protection"
  - Control access to resources"
  - Get and set permissions"
  - Allow and deny user access"