CSCI 3753
Operating Systems

Introduction

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 01/09/13

Operating Systems

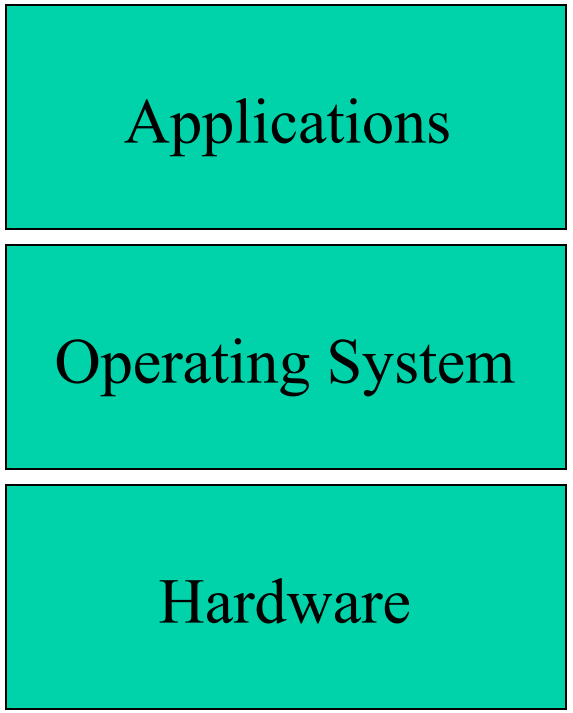
- OS is an essential (and perhaps the most important) part of a computing system.
  - PCs, laptops, supercomputers, cloud, data centers, cell phones, PDAs, tablets, embedded devices, …
  - Controls hardware components.
  - Provides a usable interface.
  - Allows sharing of various computing components.
- One of the largest piece of software.

Windows, Linux, Mac OS X, Google Android, …

> 500 at http://www.operating-system.org/betriebssystem/_english/os-liste.htm

What is an Operating System?

An operating system is a layer of software between applications and hardware that provides useful services to applications

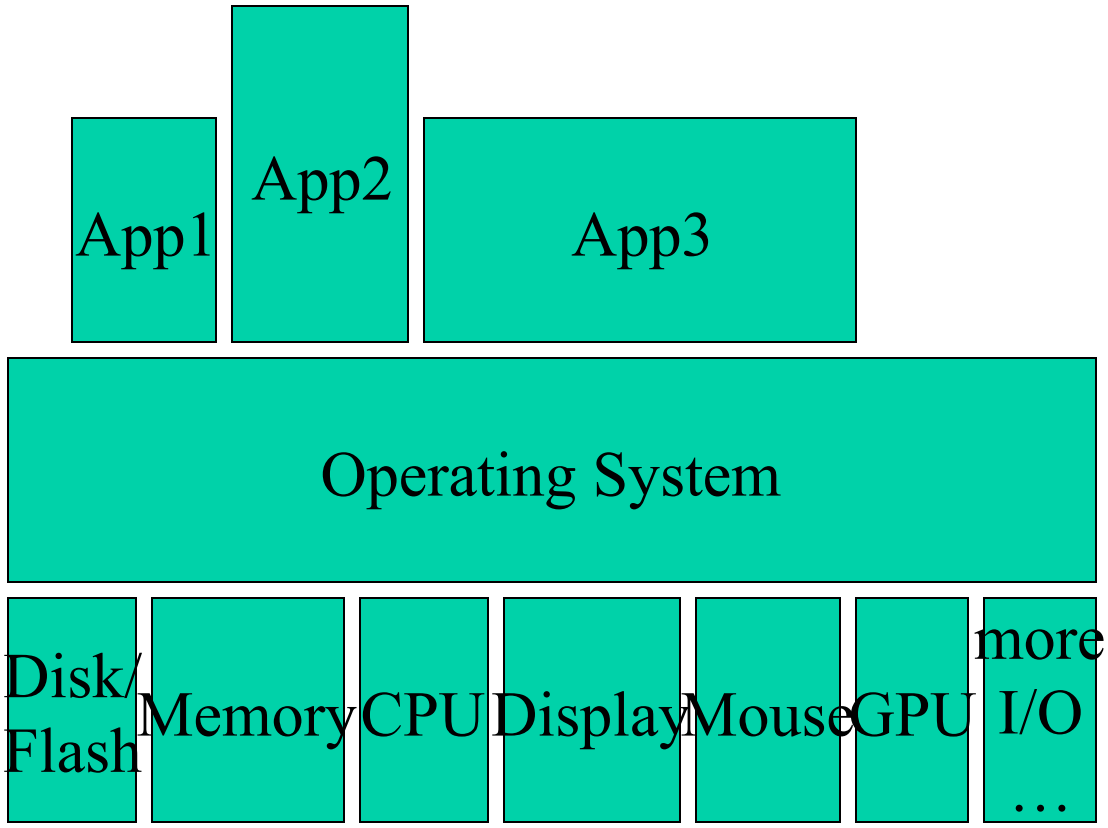


Two views of an OS

- Extended machine
  - Hardware is too complex for most computer users to understand.
  - OS provides users with an equivalent of an extended or virtual machine that is easier to use.
- Resource manager
  - A computing system is comprised of many resources: processors (CPUs), memory, clocks, disks, key board, mouse, monitor, network cards, etc.
  - OS allows sharing and effective utilization of these resources.

Security and protection

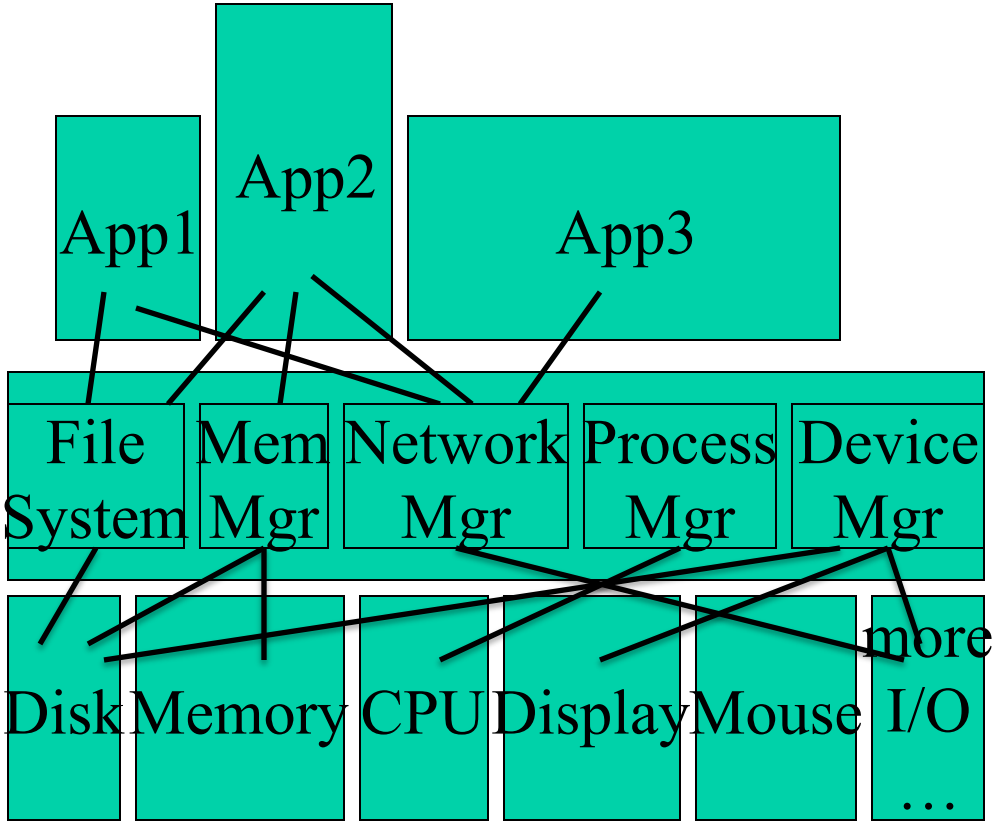
What is an Operating System?



Other devices include: wired network card, WiFi, camera, microphone, audio output, keyboard, DVD/CD, USB, etc.

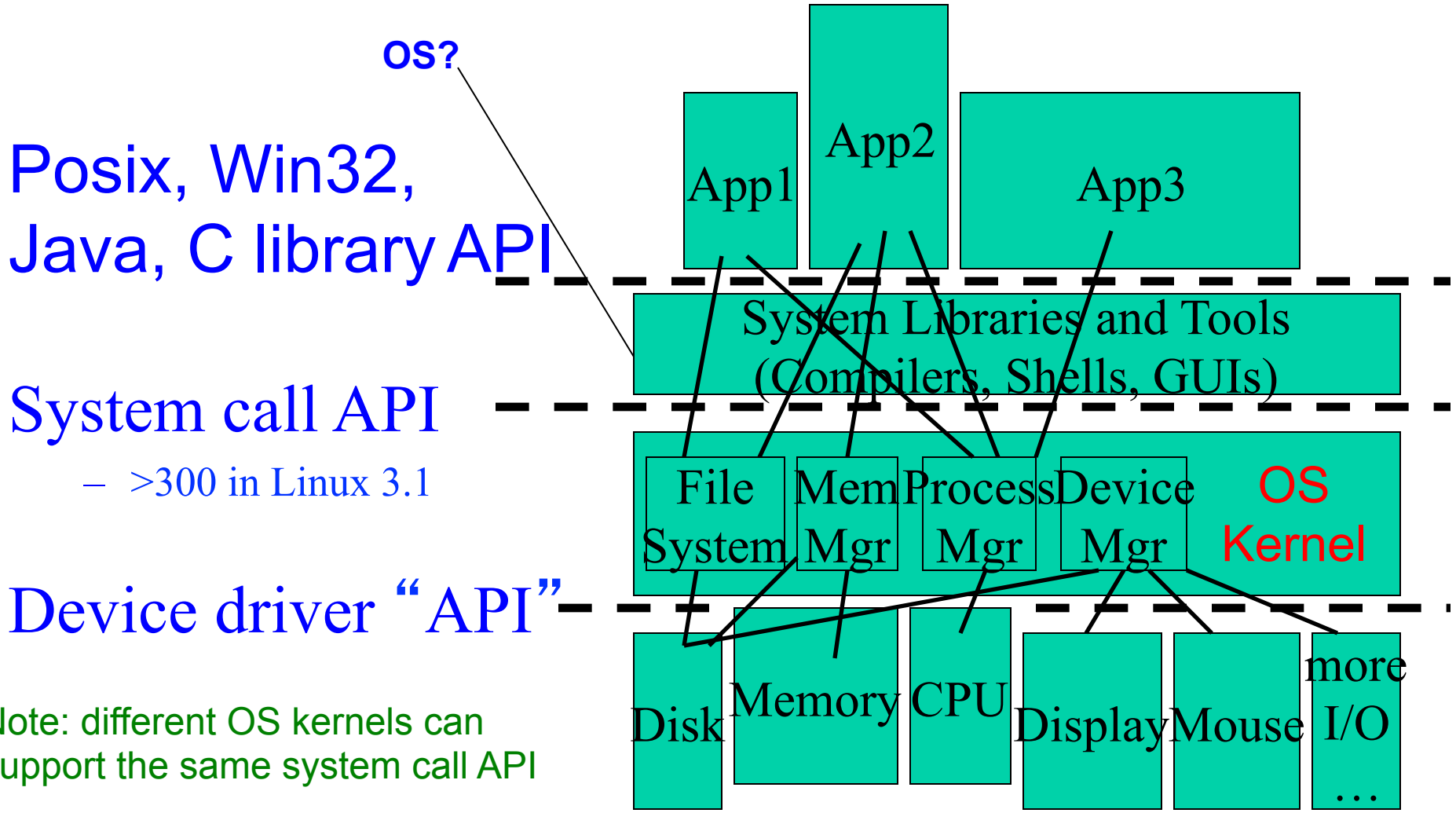
OS as Resource Manager

- Process management
- Memory management
- File system
- Device management
- Network management



Not pictured above in the OS:
the network manager.

Extended Machine View



Note: different OS kernels can support the same system call API

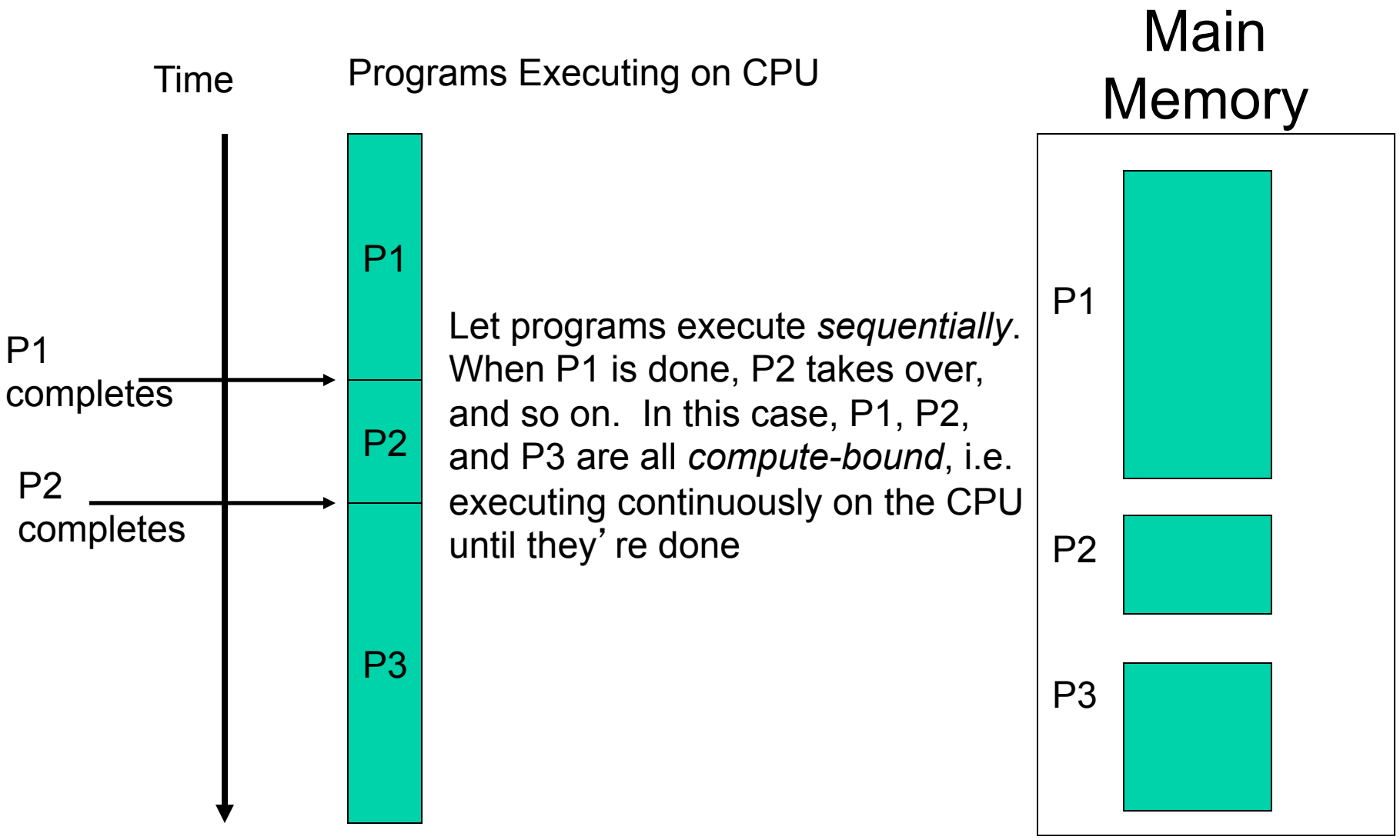
Evolution of OS

- 1940's and 50's
  - Vacuum tubes
  - Single user
  - No programming language: machine language
  - Used for straight-forward numerical calculations
  - Single program
  - No OS
- mid 50's and early 60's
  - Transistors (more reliable, smaller in size, cheaper)
  - Punch cards
  - **Batch processing**
  - Used for scientific and engineering calculations.
  - FORTRAN

Batch Processing

- Execute a pre-defined collection of programs (jobs) called a batch.
- No human interaction

Batch Processing

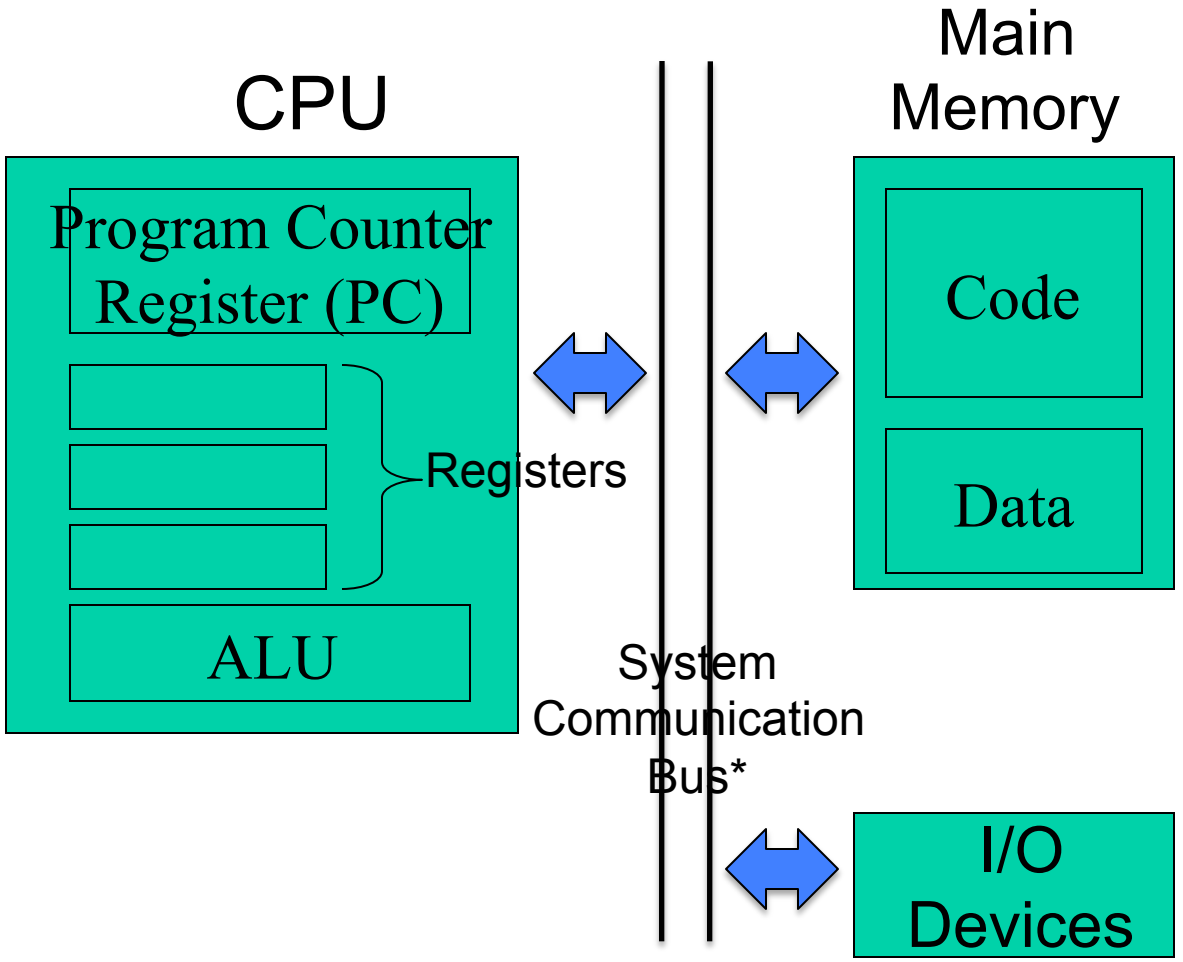


Limitations of Batch Processing

- Batch jobs are very non-interactive
  - Don't support a shell application for example
  - 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

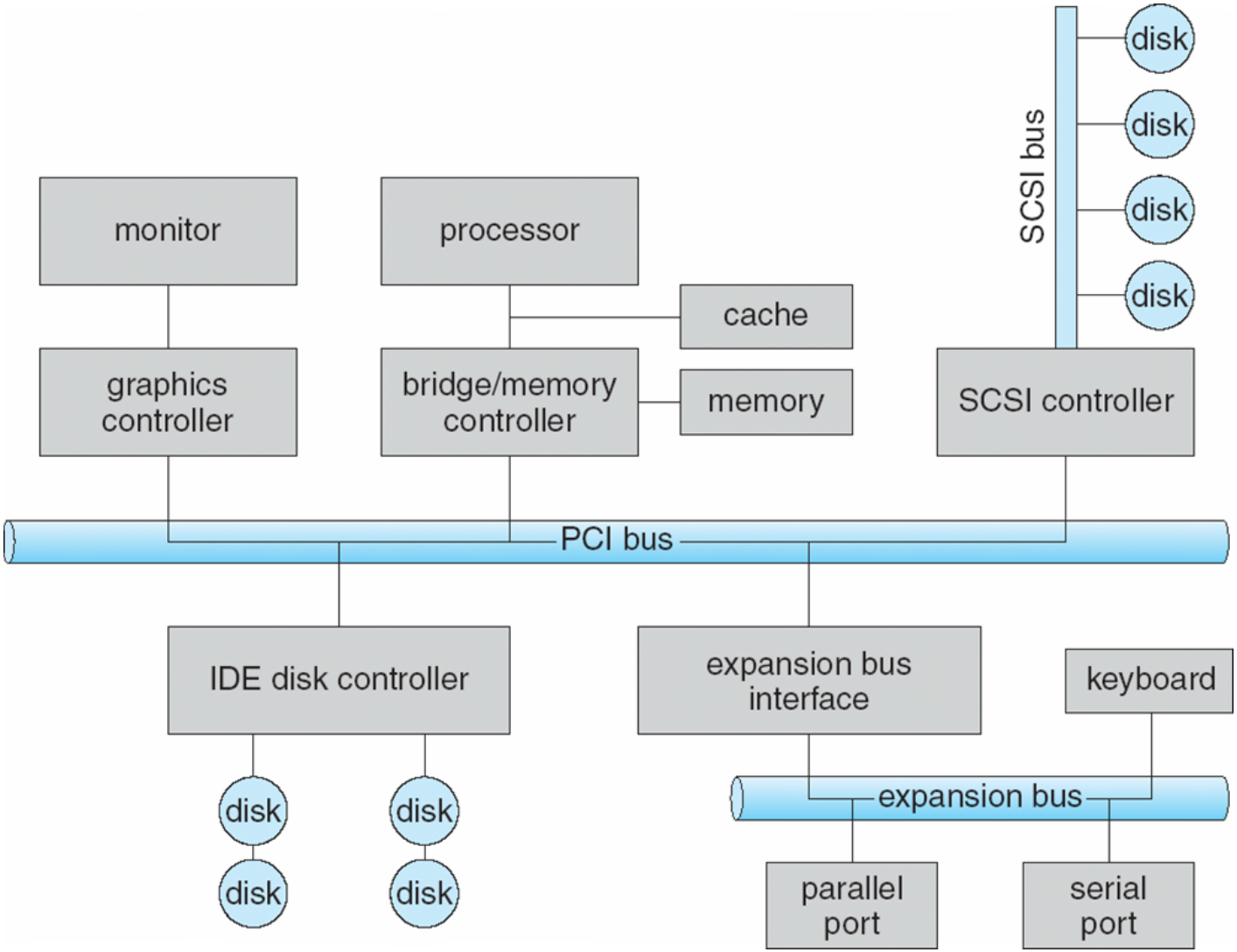
- mid 60's to mid 70's
  - Integrated circuits.
  - **Multiprogramming:** When CPU is idle, e.g. when the running program is blocked for an I/O, start another program.
  - **Multitasking:** Switch CPU between different programs irrespective of whether the running program is blocked.
  - Examples: CTSS, IBM 360, Multics, Unix

The von Neumann Computer's Hardware Architecture and I/O

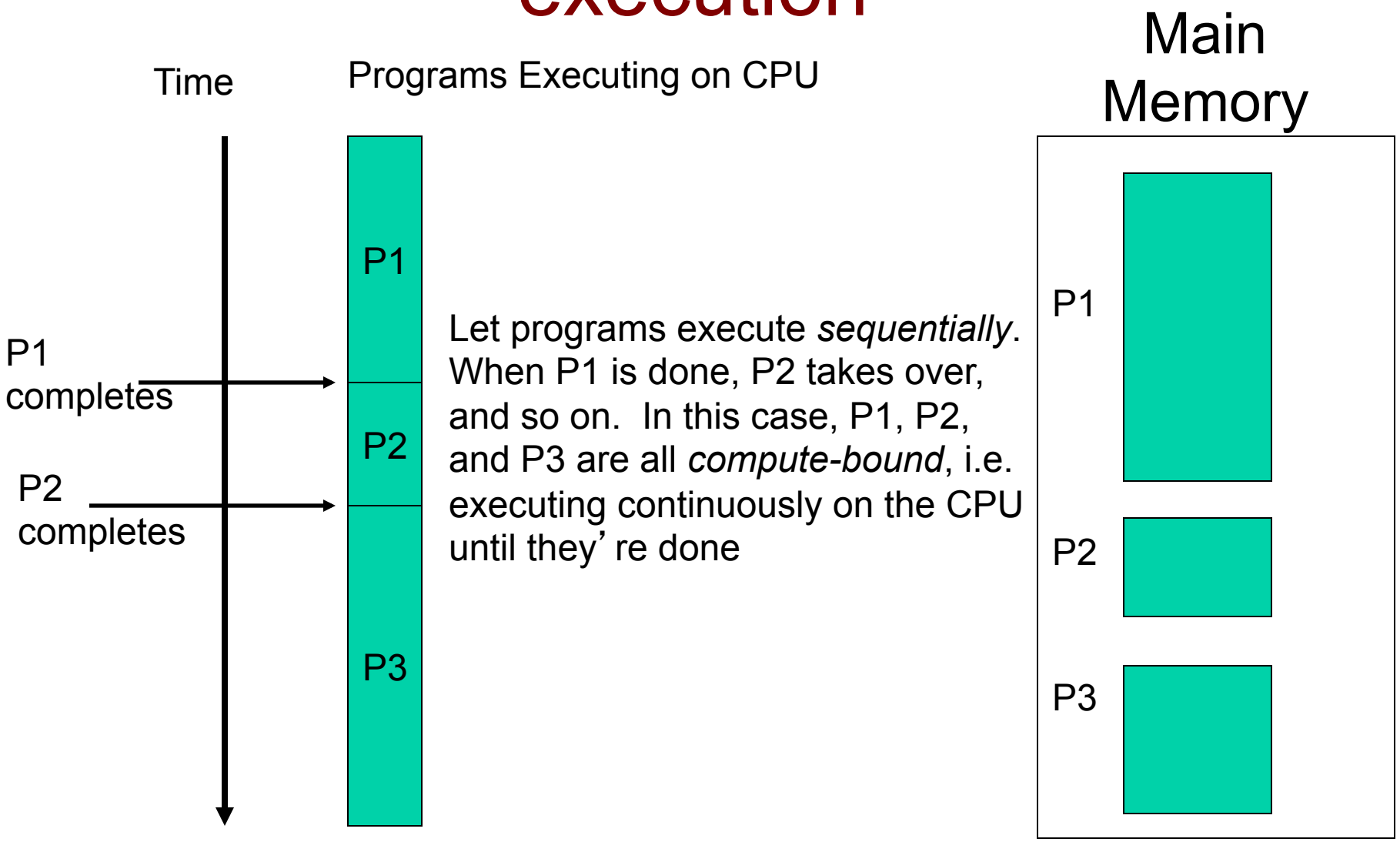


* Includes control, address, and data buses

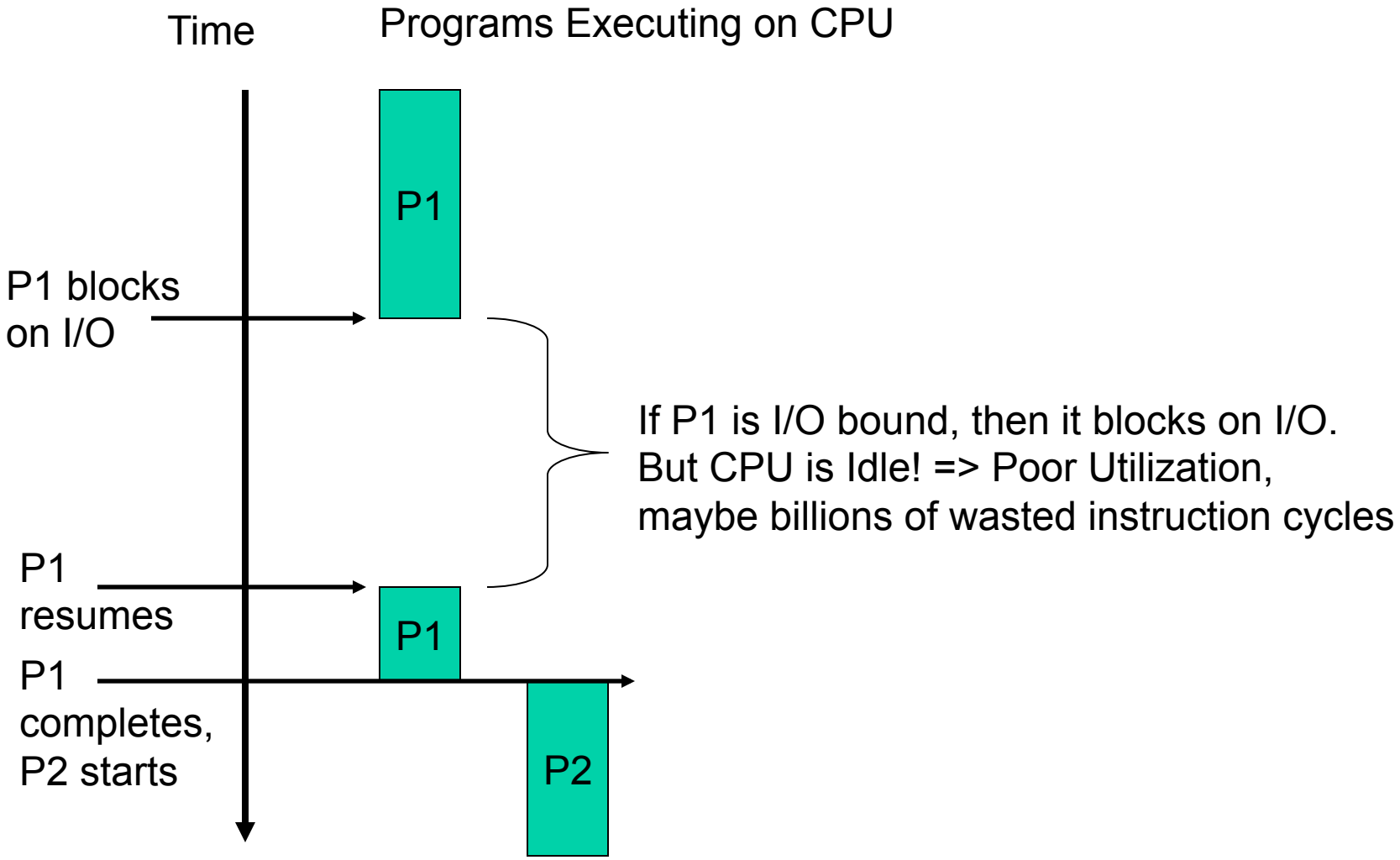
A Typical PC Bus Structure



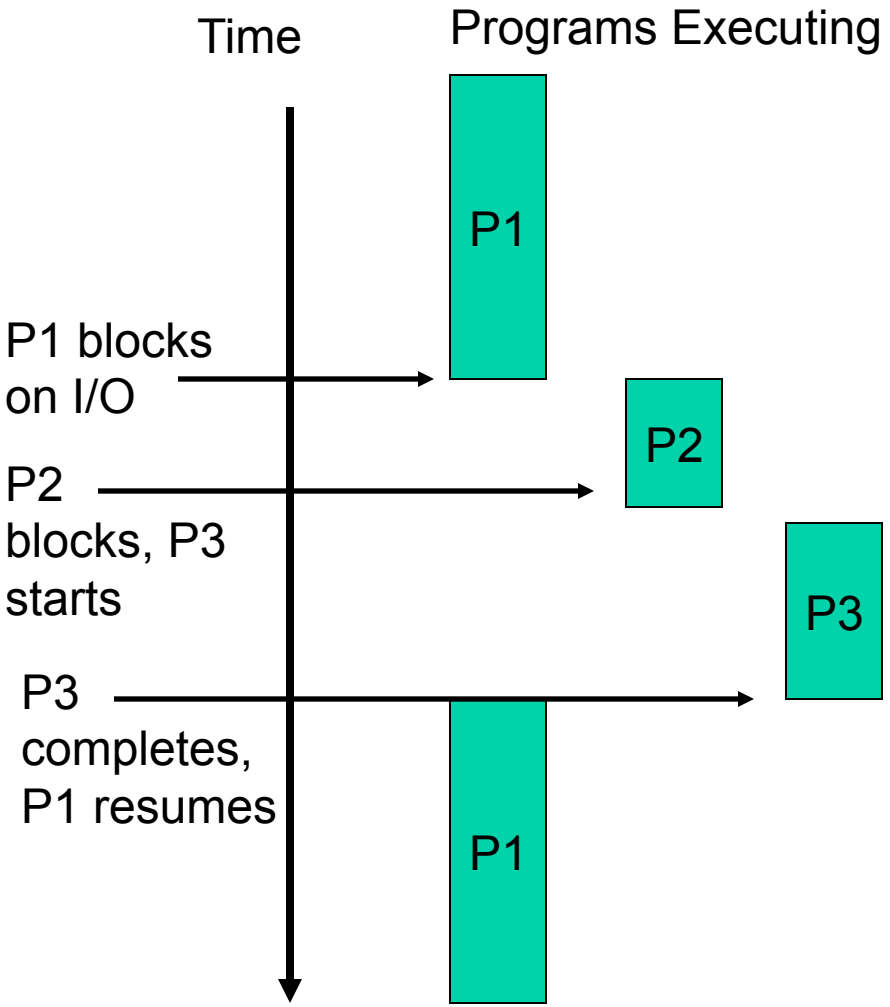
Multiprogramming: sequential execution



Multiprogramming: Problem with sequential execution



Multiprogramming



- OS Scheduler switches CPU between multiple executing programs when a program is blocked, e.g. for I/O
- OS *time-multiplexes* CPU between executable programs
- Context switch

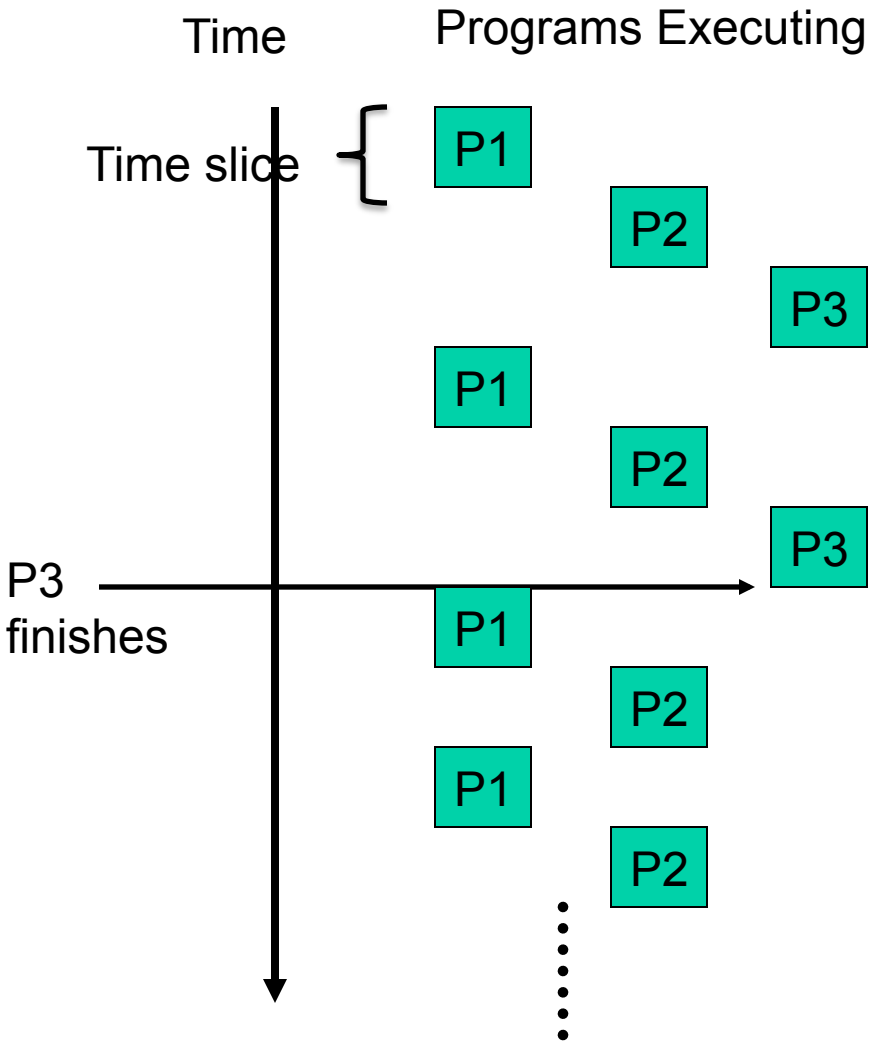
Context Switch

- Switching CPU from one running program to another is called a *context switch*
  - Save the current state (PC, IR, data registers, stack pointers, etc.) of the running application
  - Load the state of the new application
  - there is overhead due to this context switching
- No useful progress occurs for any applications while the OS is context-switching the CPU.

- Problem with pure multiprogramming: A CPU-bound program may delay the execution of other programs
  - A program with an infinite loop
  - A program to calculate the value of pi to the one-billionth decimal place

Multitasking

- CPU rapidly switches between programs



Advantages:

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

Disadvantage: context switch overhead

Cooperative vs Preemptive Multitasking

- In cooperative multitasking, programs quickly and voluntarily yield CPU before they're done
  - Early OSs did this (Windows 3.1, Mac OS 9.*)
  - Poor fault isolation due to misbehaving programs
- In preemptive multitasking, OS forces programs to give up CPU
  - Fault isolation, interactivity, efficient CPU utilization
  - Time slice: time interval for which a program is allowed to run at any time
- All modern OSs are preemptively multitasked
  - Linux, BSD Unix, Windows NT/XP/Vista/7, Mac OS X 10.*

Preemptive Multitasking

- How does the OS force *rapid* switching?
  - 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
- Context switch overhead: If you choose your default time slice too small, then you'll incur a lot of context-switching overhead as a % of your overall CPU utilization.

- mid 70's to 90's
  - LSI/VLSI; LANs
  - PCs and workstations
  - Process control and real-time systems
  - User friendly software
  - Distributed operating systems
  - MS DOS, 4.2 BSD Unix with TCP/IP, Mac OS with GUI, Linux, …

OS Design

- OS design is guided by two factors:
  - Technology: CPU, storage, communication, new I/O devices, …
  - Application needs

Operating System: Current Trends

- Hardware has evolved quickly - OS must adjust
  - Moore's Law roughly applies to CPU speed and/or memory size: doubles every 18 months => exponential!
    - Enables complex modern operating systems like Linux, Windows, UNIX, OS X

Storage Evolution:

Punch card stores code 1950s-80s

Tape Drive 1950s-80s

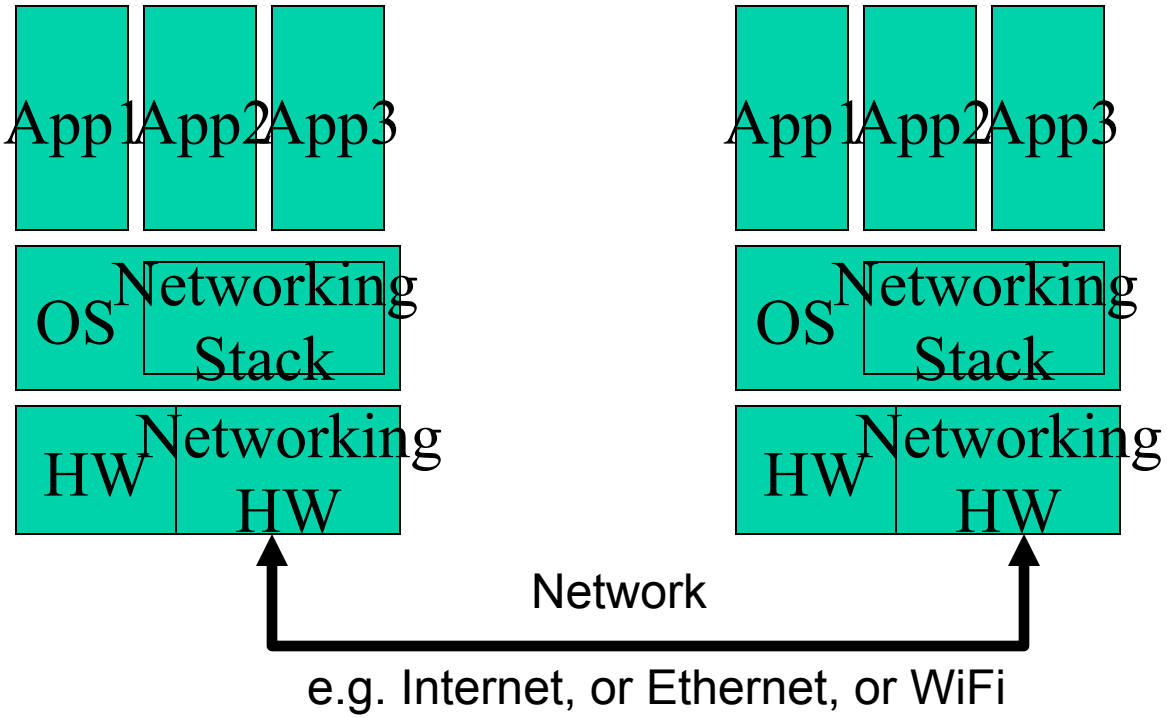
Disk Drive 1960s-2000s

??

Flash Drive already at 16 GB @ $30 => OS disk scheduling obsolete??

But Moore's Law doesn't apply to disk access speed or to battery life

Distributed operating systems



- Networked File System
- OS adds TCP/IP Network Stack
- Device driver support for Networking cards

- Examples:
  - App1 is a distributed client server app, e.g. App1 on left is Web browser, App1 on right is Web server

Operating System: Current Trends

- Diversification of OS's to many different target environments
  - Energy-efficient cell phone OSs - scaling down
    - iPhone's iOS, Google's Android, …
  - Multi-processor OSs - scaling up
    - Adapting Linux and Windows to multiple cores. Massively parallel supercomputers.
  - Real-Time OS for Embedded and Multimedia Systems
    - VXWorks, robotic OSs, …

Operating System: Current Trends

- Virtualization – Virtual Machines (VMs)
  - Running a Windows VM inside a Linux OS, and vice versa.
  - More layers of abstraction
- Cloud computing rents VMs on racks of PCs at a massive scale

Google Data Center in The Dalles, Oregon

CSCI 3753
Operating Systems

Design Issues

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 01/17/13

System Boot

- Operating system manages all programs: where they are stored, when to run them, etc.
- But how does the system know where the operating system is or how to load the kernel?
- *Booting* the system: Procedure od starting a computer by loading the operating system
- Bootstrap program (also called bootstrap loader)
  - Locates the kernel, loads it into main memory, and starts its execution
  - Typically a 2-step process: a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel

System Boot

- When CPU receives a reset event (powered/reboot)
  - IR is loaded with a predefined memory location that contains the initial bootstrap program
  - In ROM: needs no initialization and cannot easily be infected
- Bootstrap program
  - Run diagnostics to determine the state of the machine
  - Initialize registers, main memory, device controllers, etc.
  - Start OS
- Smaller systems: store entire OS in ROM or EPROM (firmware)
- Large systems: bootstrap loader in firmware; OS in disk

Dual Mode Operation

- Processor mode: distinguish between execution on behalf of an OS and execution on behalf of a user.
- Kernel: trusted software module that supports the correct operation of all other software; core part of OS.
- OS interface: how a user interacts with an OS to request OS services.

Protecting the OS via a Mode Bit

- In early CPUs, there was no way to differentiate between the OS and applications:
  - Want to protect OS from being overwritten by app's
  - Want to prevent applications from executing certain privileged instructions, like resetting the time slice register, resetting the interrupt vector, etc.
- 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

Processor mode

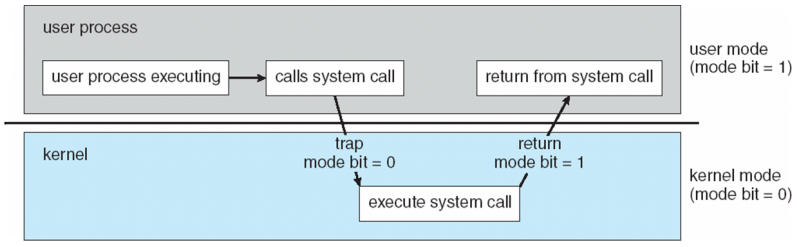
- Supervisor mode or user mode: determined by a mode bit.
  - Supervisor mode (mode bit = 0): processor can execute every instruction available in the instruction set.
  - User mode (mode bit = 1): processor can execute only a subset of instructions available in the instruction set.
- Privileged (protected) instructions:
  - Instructions that can be executed only in supervisor mode.
  - I/O instructions
  - Protection and security: privileged load and store instructions
- Used to define two classes of memory space: user space and system space.

User / Kernel Modes

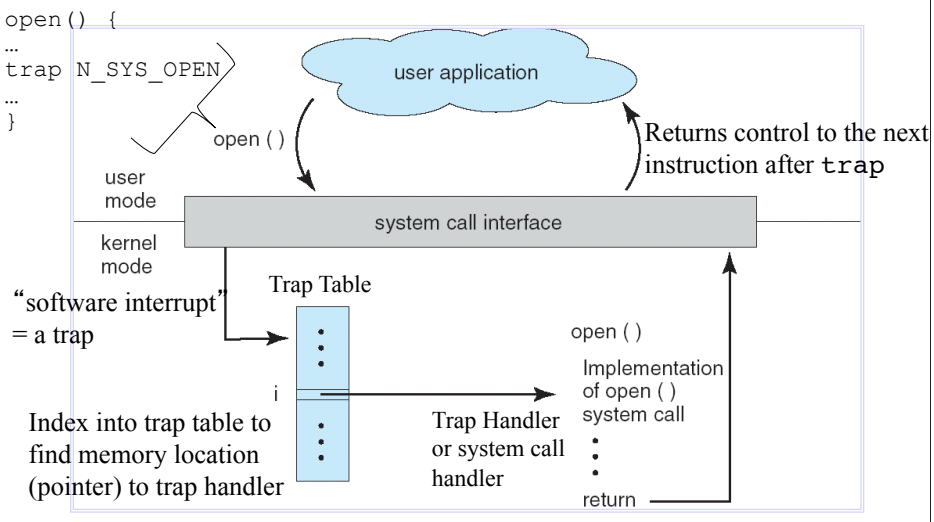
- Most modern CPU's support this mode bit
  - Intel x86 CPUs have four modes or rings, but not all are necessarily used by the OS
    - Example: an OS like Linux or Windows might set itself as ring/mode 0 (highest privilege, can execute any CPU instruction and access any location in memory), while all applications run in ring/mode 3 (lowest privilege, if an app attempts to run a privileged instruction or access restricted memory, it will generate a fault, invoking the higher privileged OS). Rings 1 and 2 unused.
    - Example: a virtual machine monitor (VMM) such as VMWare might set itself as ring 0, while a guest OS VM might run as ring 1 or 2, and user applications would run as ring 3
  - embedded microcontrollers typically don't have a mode bit

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
  - Also called `syscall` in MIPS
  - 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



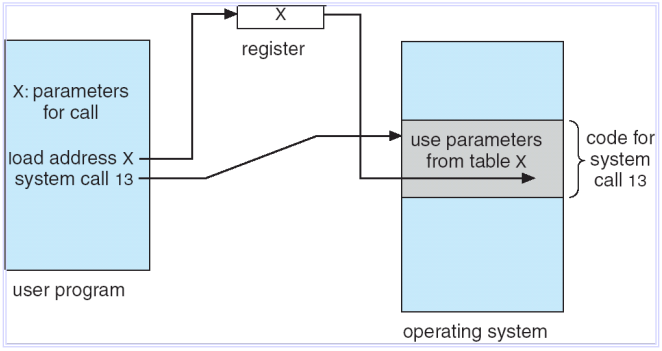
Trap Table

- 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

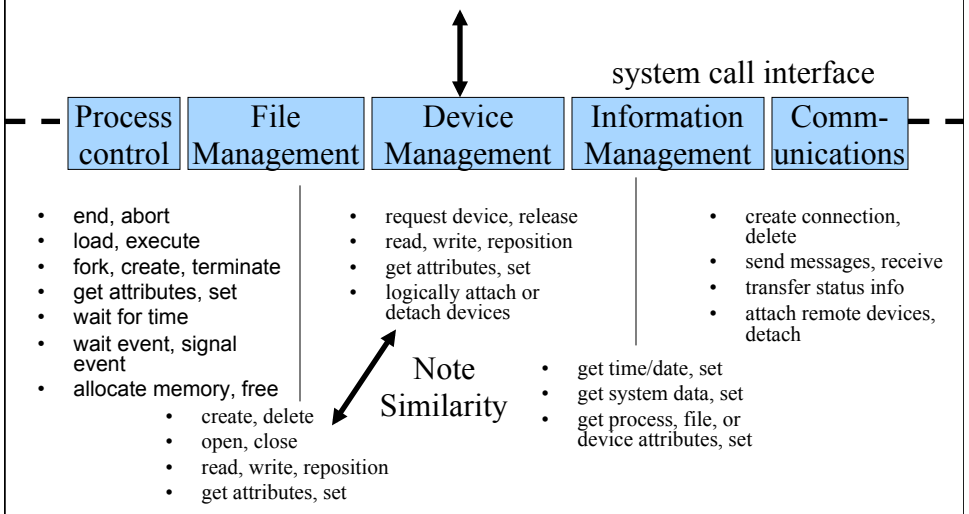
System Call Parameter Passing

- 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
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block* in memory, and block address passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - 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`



Examples of Exceptions in x86 Systems

Class	Cause	Examples	Return behavior
Trap	Intentional exception, i.e. "software interrupt"	System calls	always returns to next instruction, synchronous
Fault	Potentially recoverable error	Divide by 0, stack overflow, invalid opcode, page fault	might return to current instruction, sync
Abort	nonrecover-able error	Hardware bus failure	never returns, sync
Interrupt	signal from I/O device	Disk read finished	always returns to next instruction, async

Course Outline

- Device Management (Chapter 13)
  - Managing I/O devices
- Process Management (Chapters 3 – 7)
  - Processes and threads
  - Process synchronization
  - CPU scheduling
  - Deadlocks
- Memory Management (Chapters 8 – 9)
  - Primary memory management
  - Virtual memory

- Storage Management (Chapters 10 – 12)
  - Mass-storage structure
  - File system interface and implementation
- Protection and security(Chapters 14 – 15)
- If time permits …
  - Virtual machines and distributed systems

CSCI 3753
Operating Systems

Device Management

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 01/22/13

Bootstrapping the OS

- Multi-stage procedure:
  1. Power On Self Test (POST) from ROM
     - Check hardware, e.g. CPU and memory, to make sure it's OK
  2. BIOS (Basic Input/Output System) looks for a device to boot from…
     - May be prioritized to look for a USB flash drive or a CD/DVD-ROM drive before a hard disk drive
     - Can also boot from network

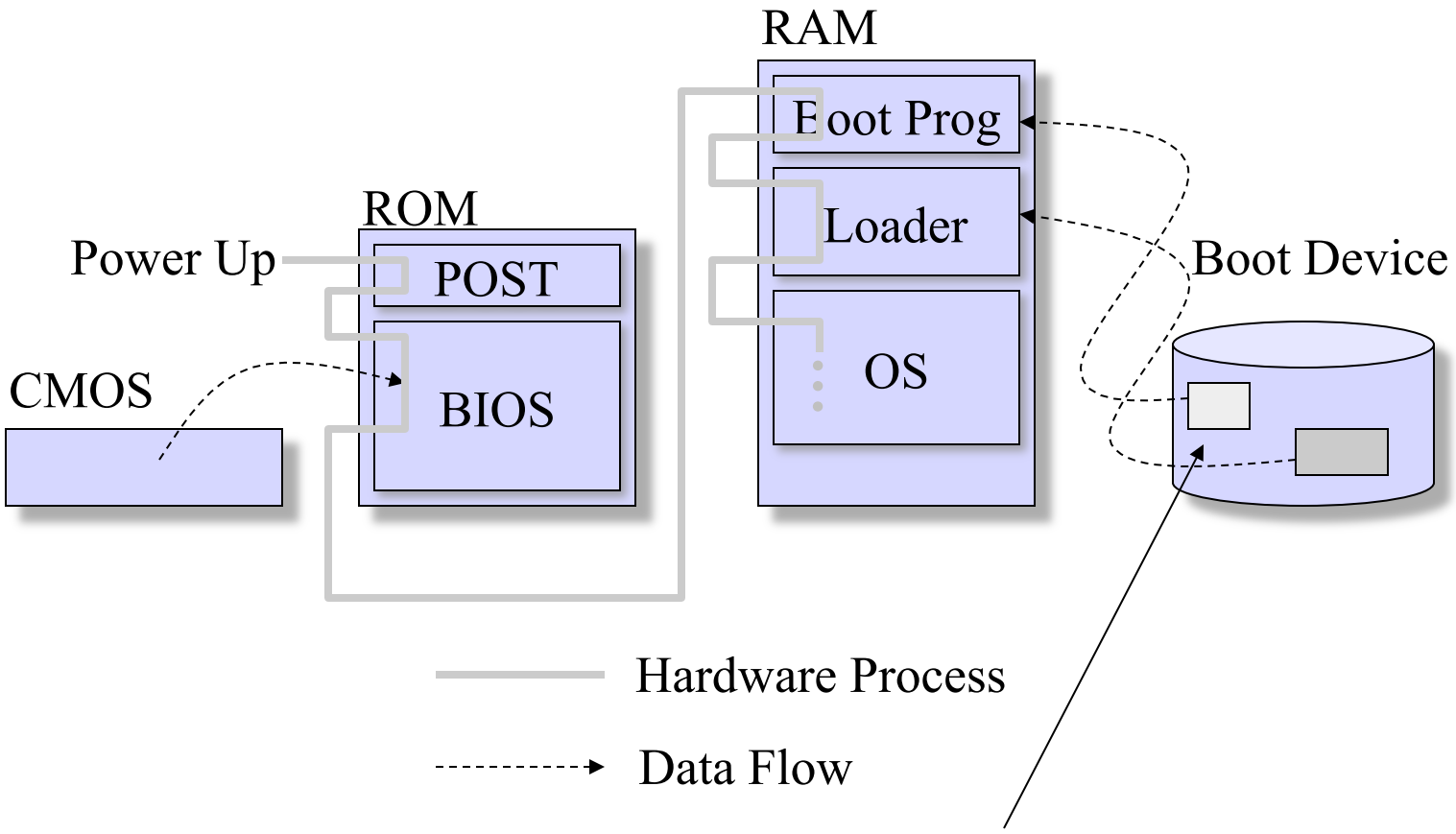
Bootstrapping the OS (2)

- Multi-stage procedure: (continued)
  3. BIOS finds a hard disk drive to boot from
     - Looks at Master Boot Record (MBR) in sector 0 of disk
     - Only 512 bytes long (Intel systems), contains primitive code for later stage loading and a partition table listing an active partition, or the location of the bootloader

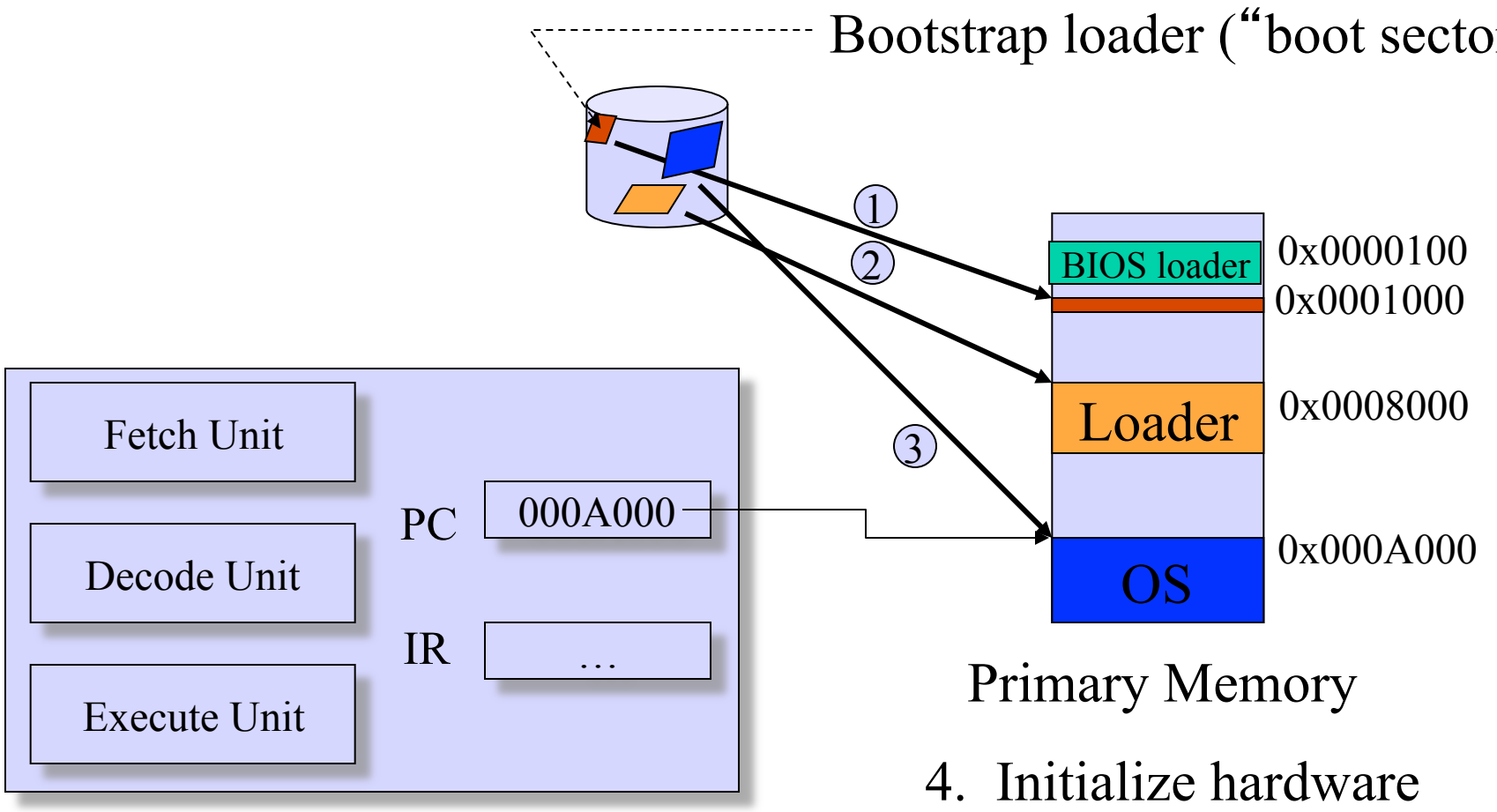
Bootstrapping the OS (3)

- Multi-stage procedure: (continued)
  4. Primitive loader then loads the secondary stage bootloader
     - Examples of this bootloader include LILO (Linux Loader), and GRUB (Grand Unified Bootloader)
     - Can select among multiple OS's (on different partitions) – i.e. dual booting
     - Once OS is selected, the bootloader goes to that OS's partition, finds the boot sector, and starts loading the OS's kernel

Intel System Initialization



Bootstrapping Example



4. Initialize hardware
5. Create user environment
6. …

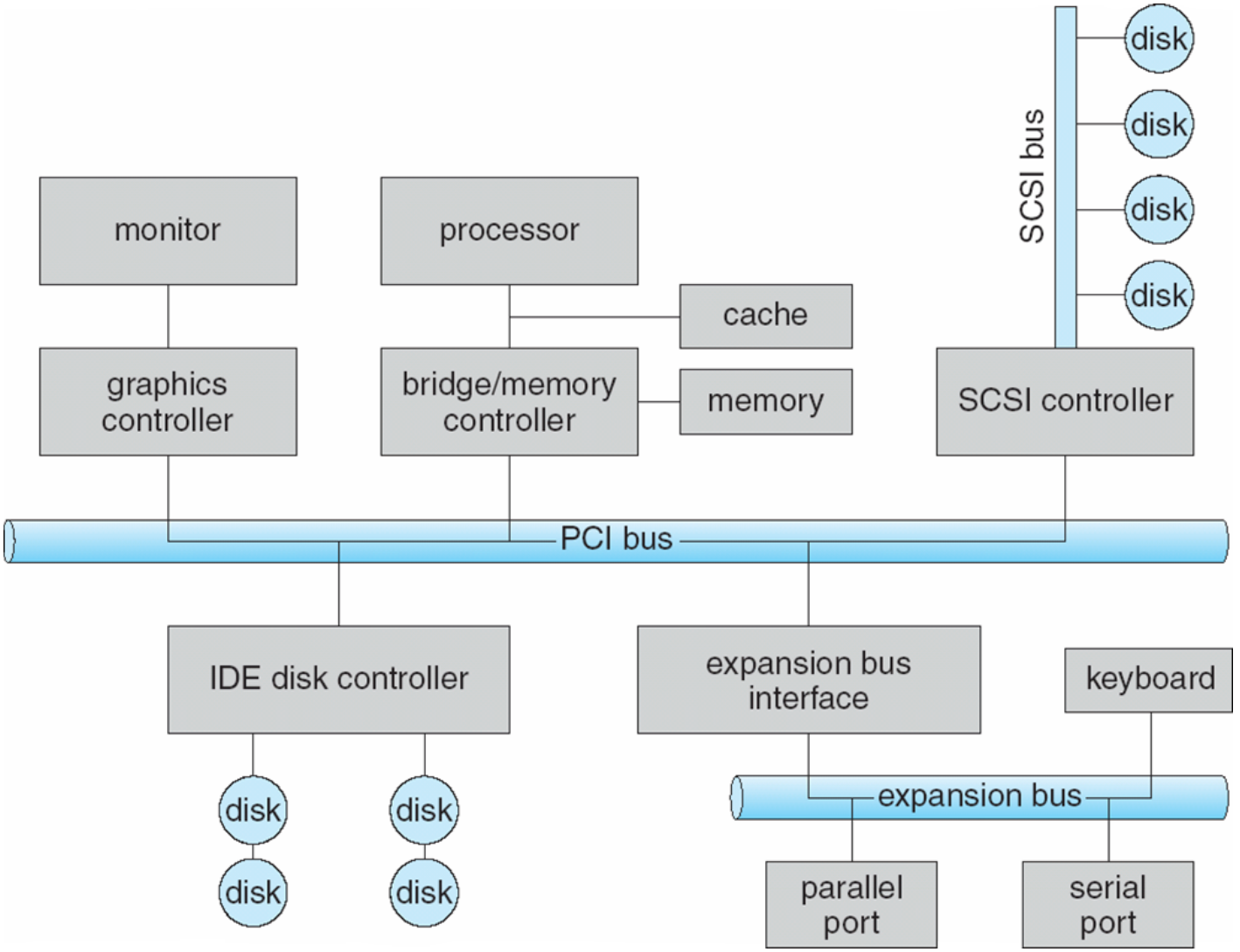
Device Manager

- Controls the operation of I/O devices
  - Issue I/O commands to the devices
  - Catch interrupts
  - Handle errors
  - Provide a simple and easy-to-use interface
    - Device independence: same interface for all devices.

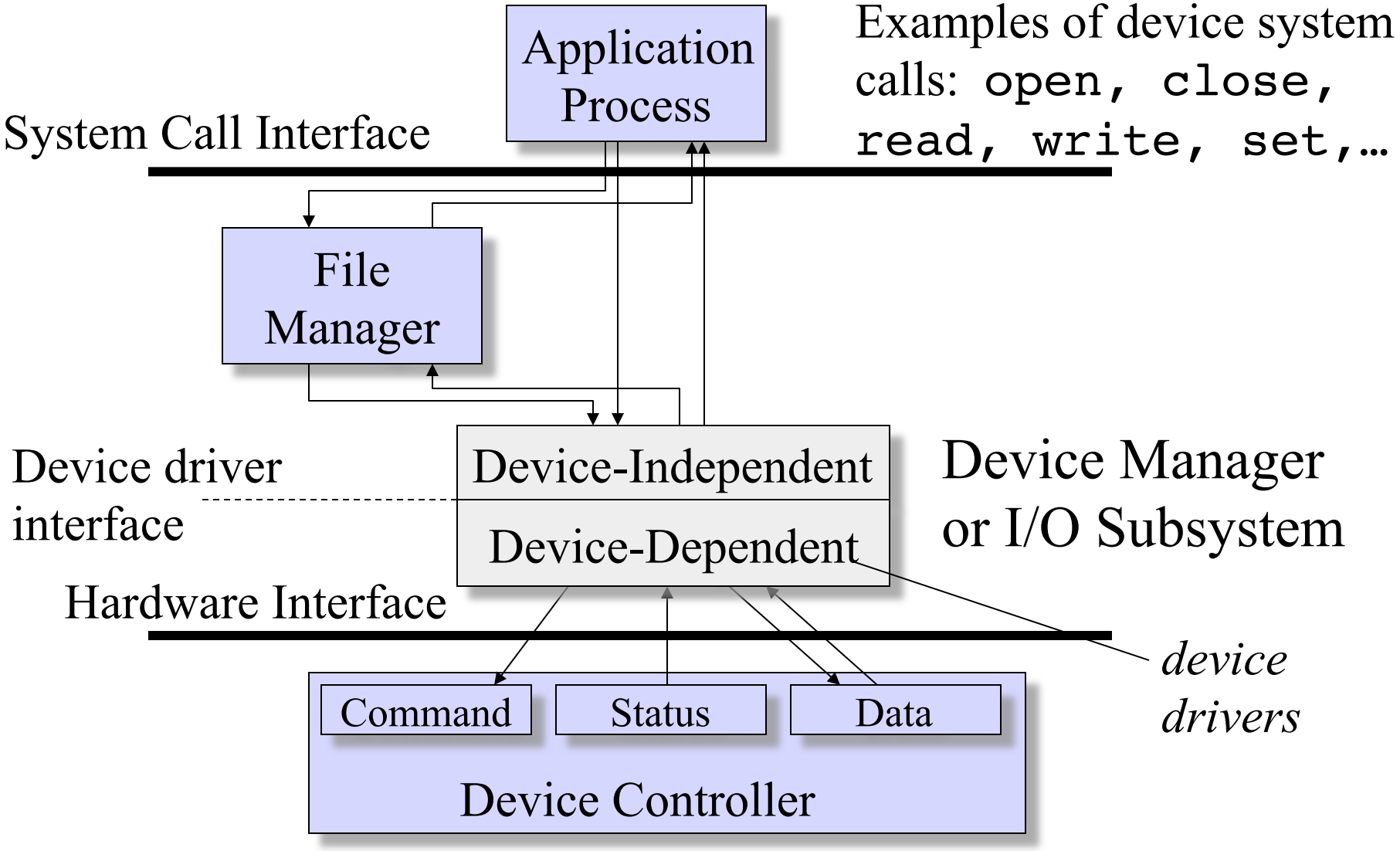
Device Characteristics

- I/O devices consist of two high-level components
  - Mechanical component
  - Electronic component: device controllers
- OS deals with device controllers

A Typical PC Bus Structure



Device Management Organization



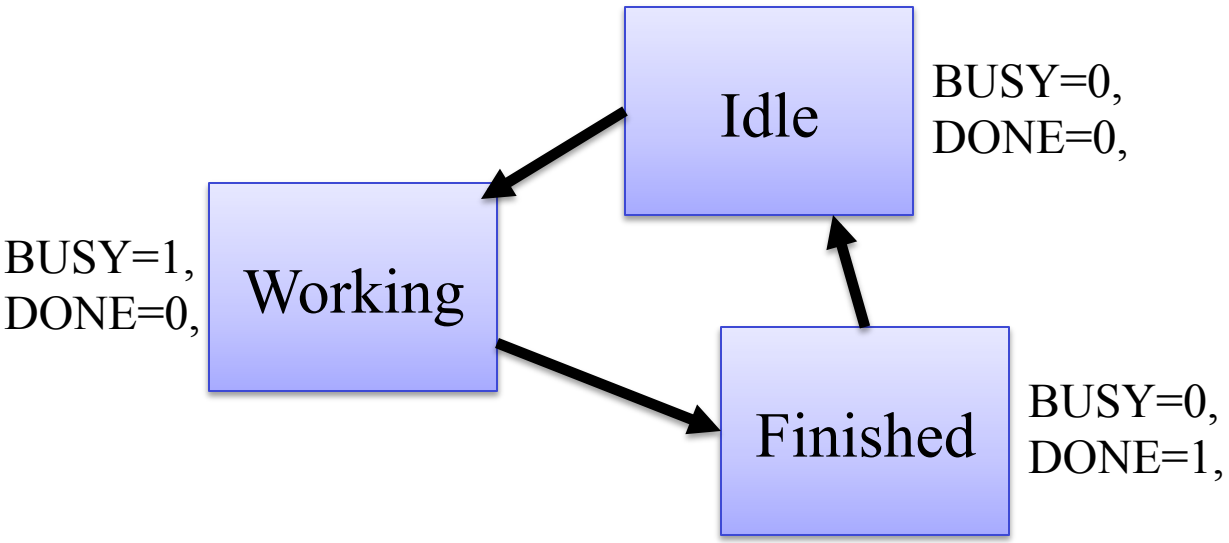
Device System Call Interface

- Create a simple standard interface to access most devices:
  - Every I/O device driver should support the following: `open, close, read, write, set` (`ioctl` in UNIX), `stop,` etc.
  - Block vs character
  - Sequential vs direct/random access
  - Blocking versus Non-Blocking I/O
    - blocking system call: process put on wait queue until I/O completes
    - non-blocking system call: returns immediately with partial number of bytes transferred, e.g. keyboard, mouse, network sockets
  - Synchronous versus asynchronous
    - asynchronous returns immediately, but at some later time, the full number of bytes requested is transferred

Device Drivers

- Support the device system call interface functions `open, read, write`, etc. for that device
- Interact directly with the device controllers
  - Know the details of what commands the device can handle, how to set/get bits in device controller registers, etc.
  - Are part of the device-dependent component of the device manager
- Control flow:
  - An I/O system call traps to the kernel, invoking the trap handler for I/O (the device manager), which indexes into a table using the arguments provided to run the correct device driver

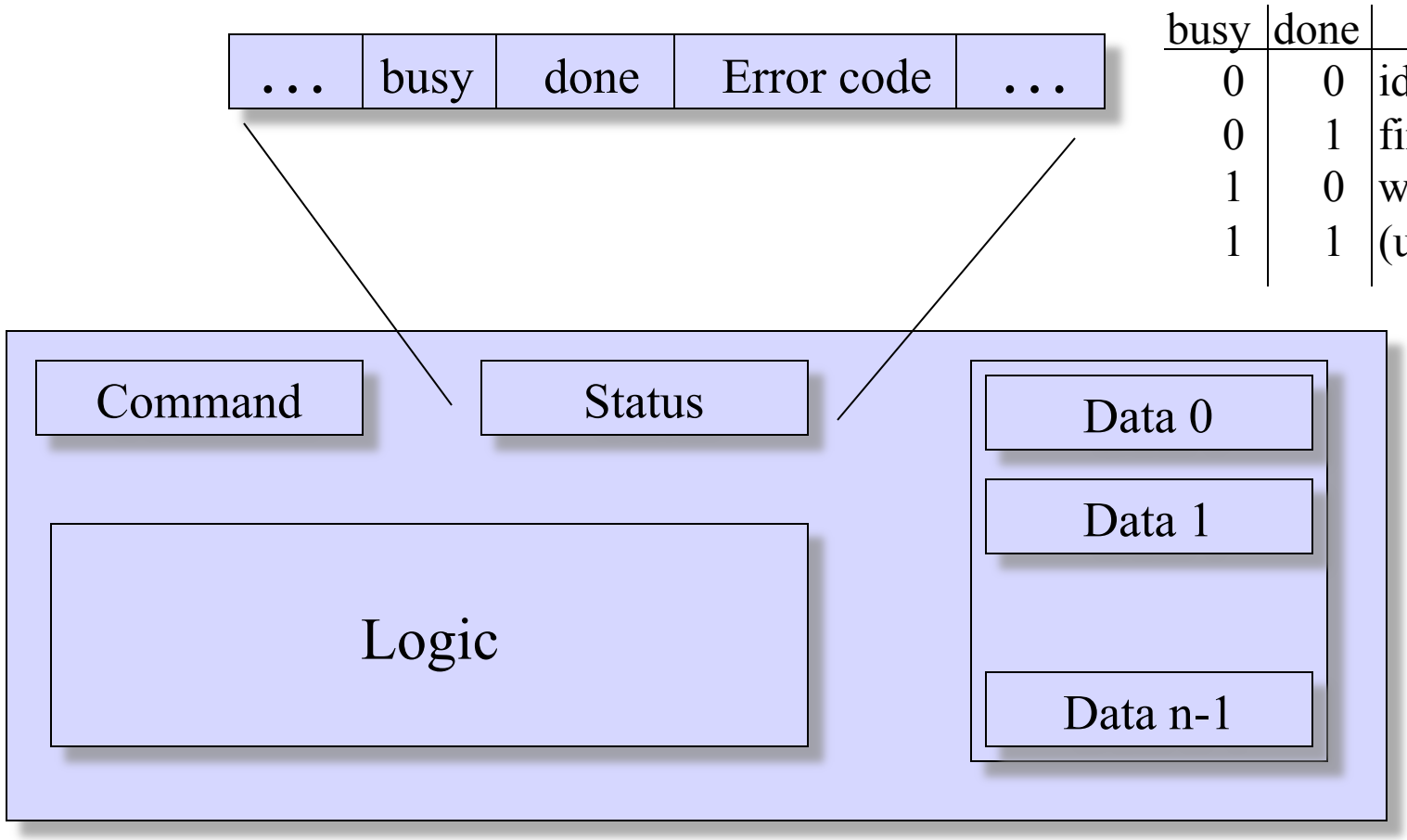
Device Controller States



- Need three states to distinguish the following:
  - Idle: no app is accessing the device
  - Working: one app only is accessing the device
  - Finished: the results are ready for that one app

- Therefore, need 2 bits for 3 states:
  - A BUSY flag and a DONE flag
  - BUSY=0, DONE=0 => Idle
  - BUSY=1, DONE=0 => Working
  - BUSY=0, DONE=1 => Finished
  - BUSY=1, DONE=1 => Undefined

Device Controller Interface



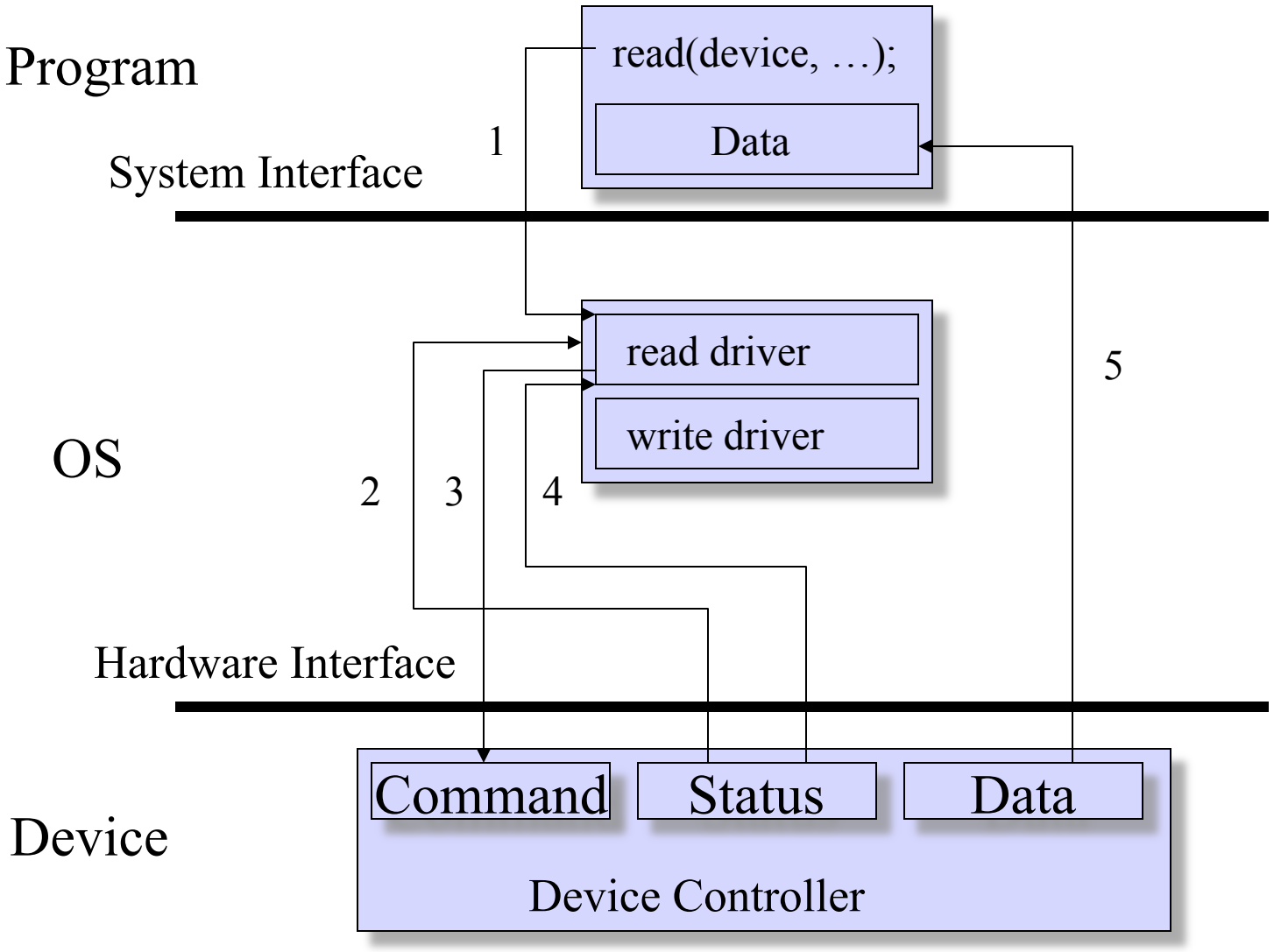
busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)

Polling I/O: A Write Example

```
                                                          BUSY  DONE
                                                            *     *
while(deviceN.busy || deviceN.done) <waiting>;            0     0
deviceN.data[0] = <value to write>
deviceN.command = WRITE;
while(deviceN.busy) <waiting>;                            1     0
/* finished, so read some status bits… */                 0     1
deviceN.done = FALSE;
                                                          0     0
```

- Devices *much* slower than CPU
- CPU waits while device operates
- Would like to multiplex CPU to a different process while I/O is in process

Polling I/O Read Operation



Polling I/O – Busy Waiting

- Note that the OS is spinning in a loop twice:
  - Checking for the device to become idle
  - Checking for the device to finish the I/O request, so the results can be retrieved
  - This wastes CPU cycles that could be devoted to executing applications
- Instead, want to *overlap* CPU and I/O
  - Free up the CPU while the I/O device is processing a read/write

Device Manager I/O Strategies

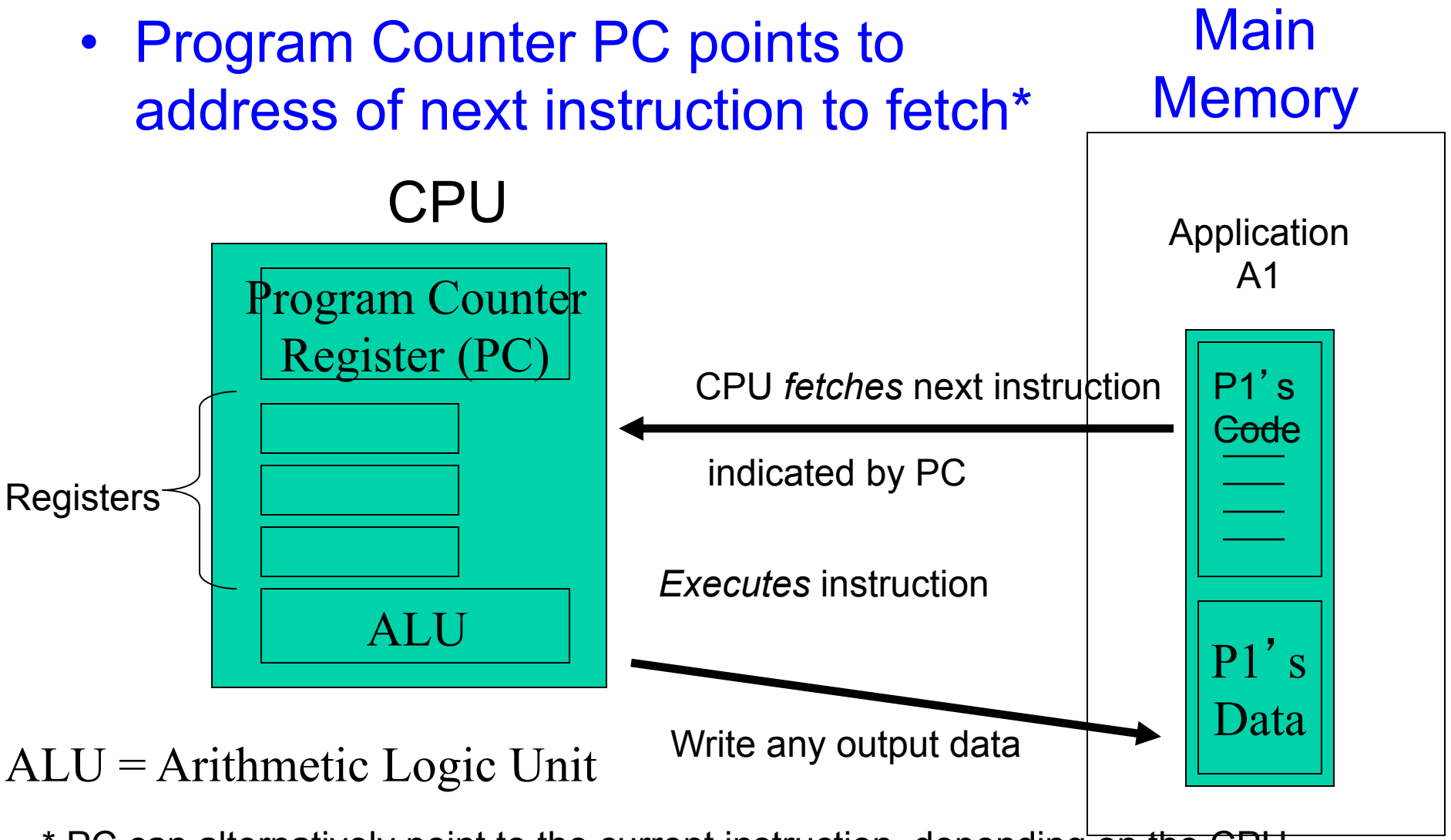
- Underneath the blocking/non-blocking synchronous/asynchronous system call API, OS can implement several strategies for I/O with devices
  - direct I/O with polling
    - the OS device manager busy-waits, we've already seen this
  - direct I/O with *interrupts*
    - More efficient than busy waiting
  - DMA with interrupts

Hardware Interrupts

- CPU incorporates a *hardware interrupt flag*
- Whenever a device is finished with a read/write, it communicates to the CPU and raises the flag
  - Frees up CPU to execute other tasks without having to keep polling devices
- Upon an interrupt, the CPU interrupts normal execution, and invokes the OS's *interrupt handler*
  - Eventually, after the interrupt is handled and the I/O results processed, the OS resumes normal execution

CPU Execution of a Program

- Program Counter PC points to address of next instruction to fetch*



* PC can alternatively point to the current instruction, depending on the CPU

CPU Checks Interrupt Flag Every Fetch/Execute Cycle

CPU Pseudocode

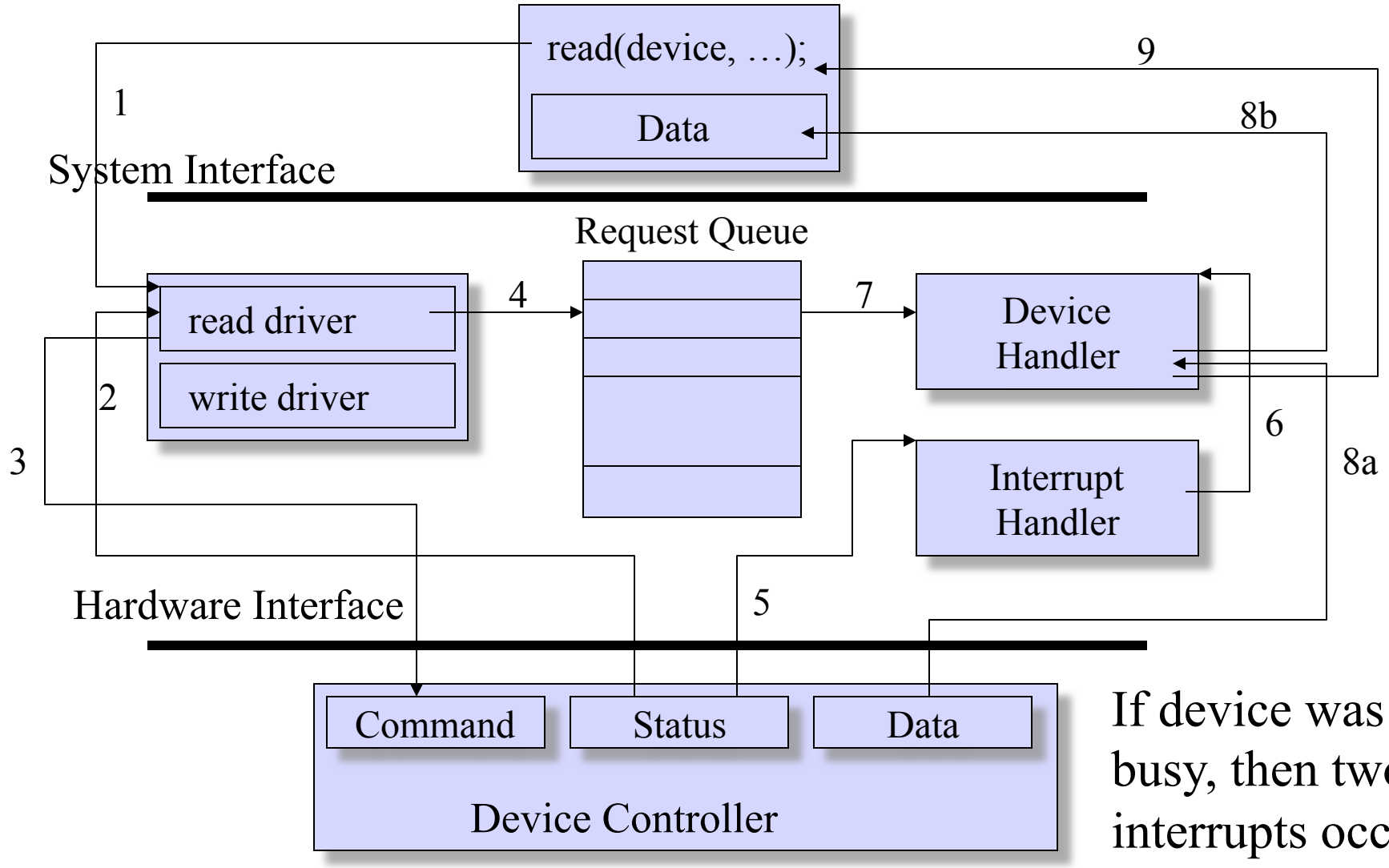
- While (no hardware failure)
  - Fetch next instruction, put in instruction register
  - Execute instruction
  - Check for interrupt: If interrupt flag enabled,
    - Save PC*
    - Jump to interrupt handler

* insight from Nutt's text

Interrupt Handler

- First, save the processor state
  - Save the executing app's program counter (PC) and CPU register data
- Next, find the device causing the interrupt
  - Consult interrupt controller to find the interrupt offset, or poll the devices
- Then, jump to the appropriate device handler
  - Index into the Interrupt Vector using the interrupt offset
  - An Interrupt Service Routine (ISR) either refers to the interrupt handler, or the device handler
- Finally, reenable interrupts

Interrupt-Driven I/O Operation

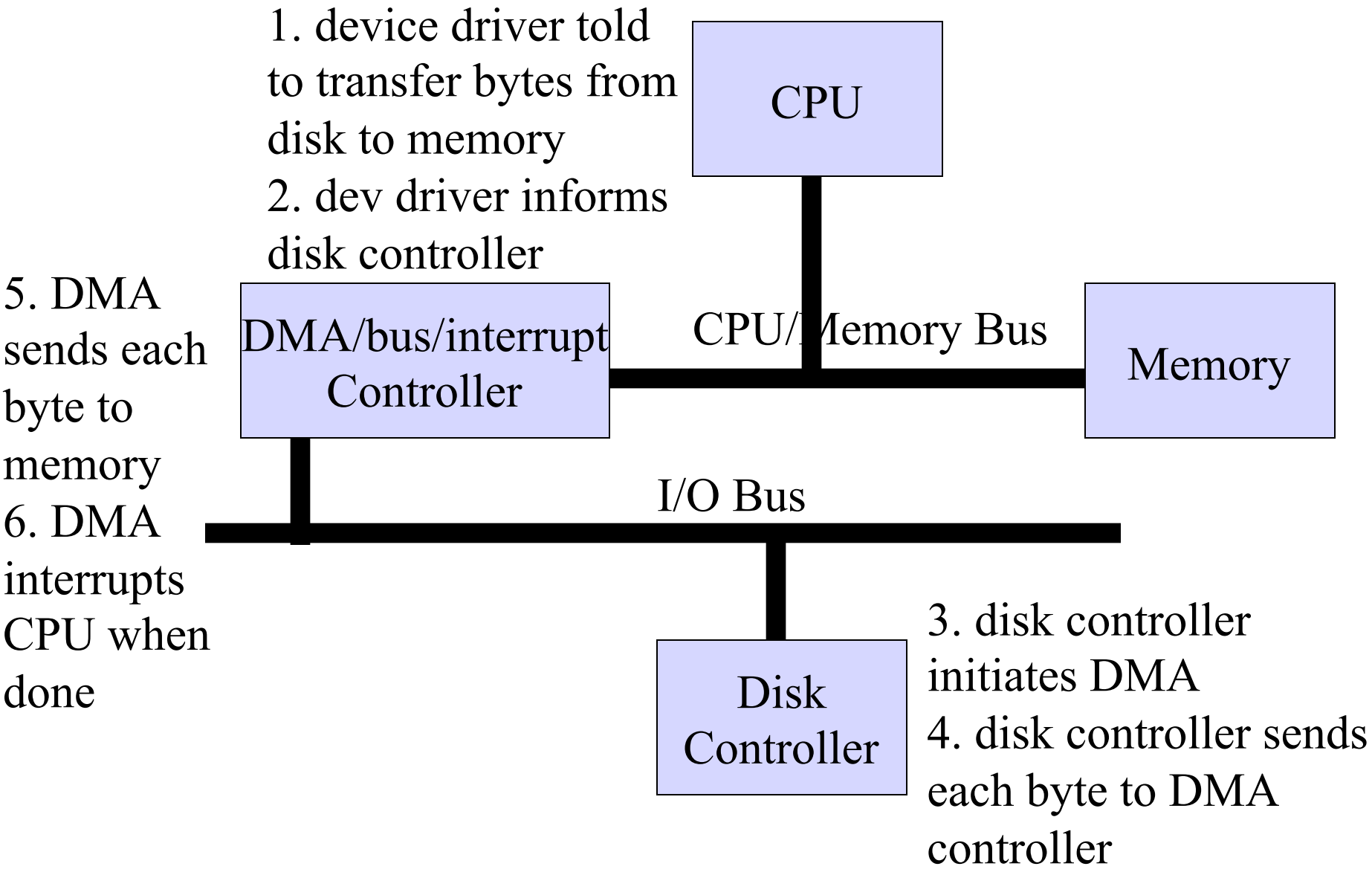


If device was busy, then two interrupts occur

Direct Memory Access (DMA)

- The CPU can become a bottleneck if there is a lot of I/O copying data back and forth between memory and devices
  - Example: want to copy a 1 MB file from disk into memory. The disk is only capable of delivering memory in say 1 KB blocks. So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU. This will slow down execution of normal programs and the OS.
  - Worst cases: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card
- DMA solution: Bypass the CPU for large data copies, and only raise an interrupt at the very end of the data transfer, instead of at every intermediate block

DMA with Interrupts Example



Direct Memory Access (DMA)

- Since both CPU and the DMA controller have to move data to/from main memory, how do they share main memory?
  - Burst mode
    - While DMA is transferring, CPU is blocked from accessing memory
  - Interleaved mode or "cycle stealing"
    - DMA transfers one word to/from memory, then CPU accesses memory, then DMA, then CPU, etc… - interleaved
  - Transparent mode – DMA only transfers when CPU is not using the system bus
    - Most efficient but difficult to detect

Memory-Mapped I/O

- Non-memory mapped (port or port-mapped) I/O typically requires special I/O machine instructions to read/write from/to device controller registers
  - e.g. on Intel x86 CPUs, have IN, OUT
    - Example: OUT dest, src (using Intel syntax, not Gnu syntax)
      - Writes to a device port dest from CPU register src
    - Example: IN dest, src
      - Reads from a device port src to CPU register src
    - Only OS in kernel mode can execute these instructions
    - Later Intel introduced INS, OUTS (for strings), and INSB/INSW/INSD (different word widths), etc.

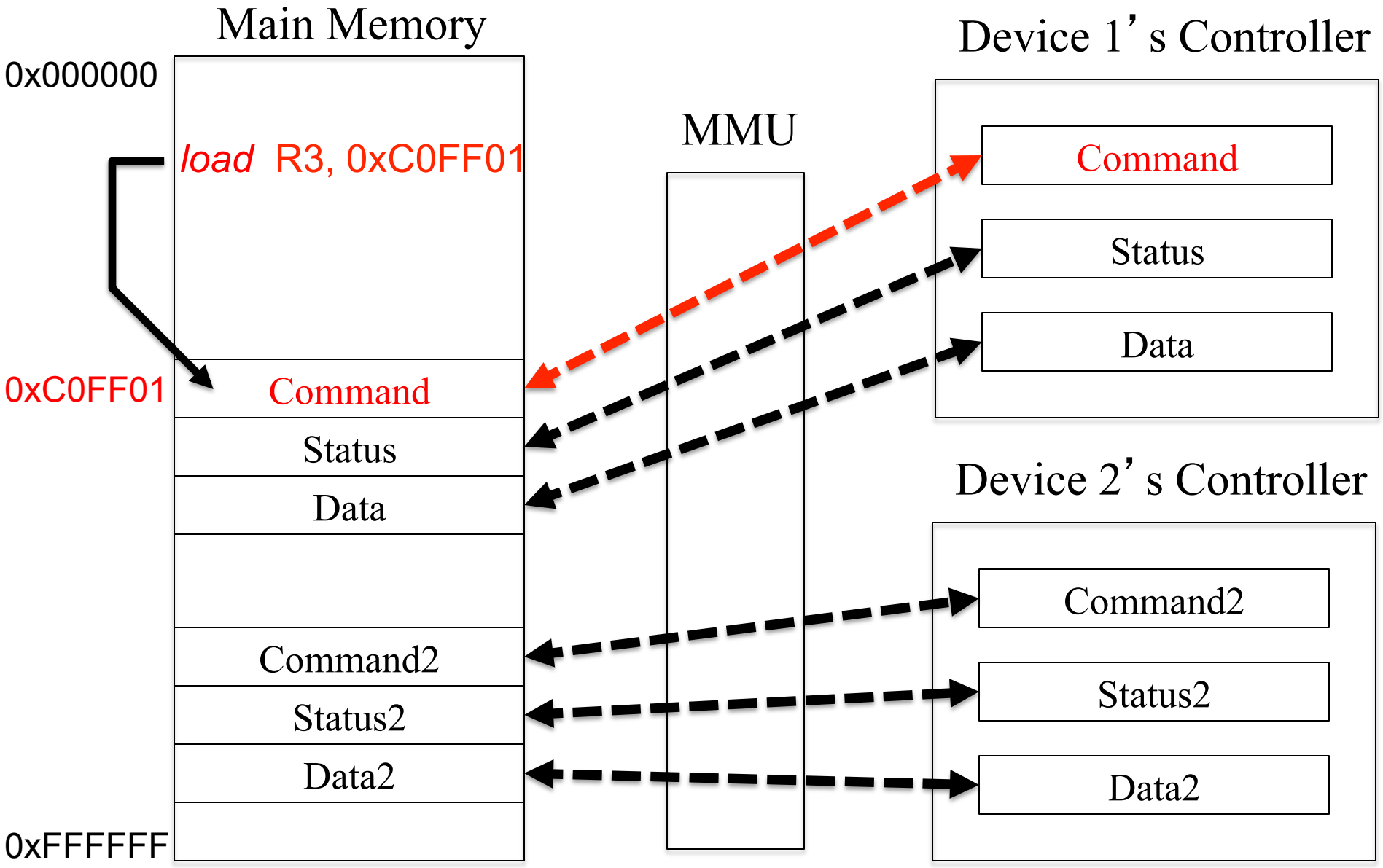
Memory-Mapped I/O (2)

- port-mapped I/O is quite limited
  - IN and OUT can only store and load
  - don't have full range of memory operations for normal CPU instructions
    - Example: to increment the value in say a device's data register, have to copy register value into memory, add one, and copy it back to device register.
      - With memory-mapped I/O, can increment value in memory directly, and it gets reflected into the device controller's data register automatically
    - AMD did not extend the port I/O instructions when defining the x86-64

Memory-Mapped I/O (3)

- With memory-mapped I/O, just address memory directly using normal instructions to speak to an I/O address
  - e.g. load R3, 0xC0FF01
  - the memory address 0xC0FF01 is mapped to the I/O device's registers
  - Memory Management Unit (MMU) maps memory values and data to/from device registers
    - Device registers are assigned to a block of memory
    - When a value is written into that I/O-mapped memory, the device sees the value, loads the appropriate value and executes the appropriate command

Memory-Mapped I/O (4)



Memory-Mapped I/O (5)

- Typically, devices are mapped into lower memory
  - frame buffers for displays take the most memory, since most other devices have smaller buffers
  - Even a large display might take only 10 MB of memory, which in modern address spaces of tens-hundreds of GBs is quite modest – so memory-mapped I/O is a small penalty

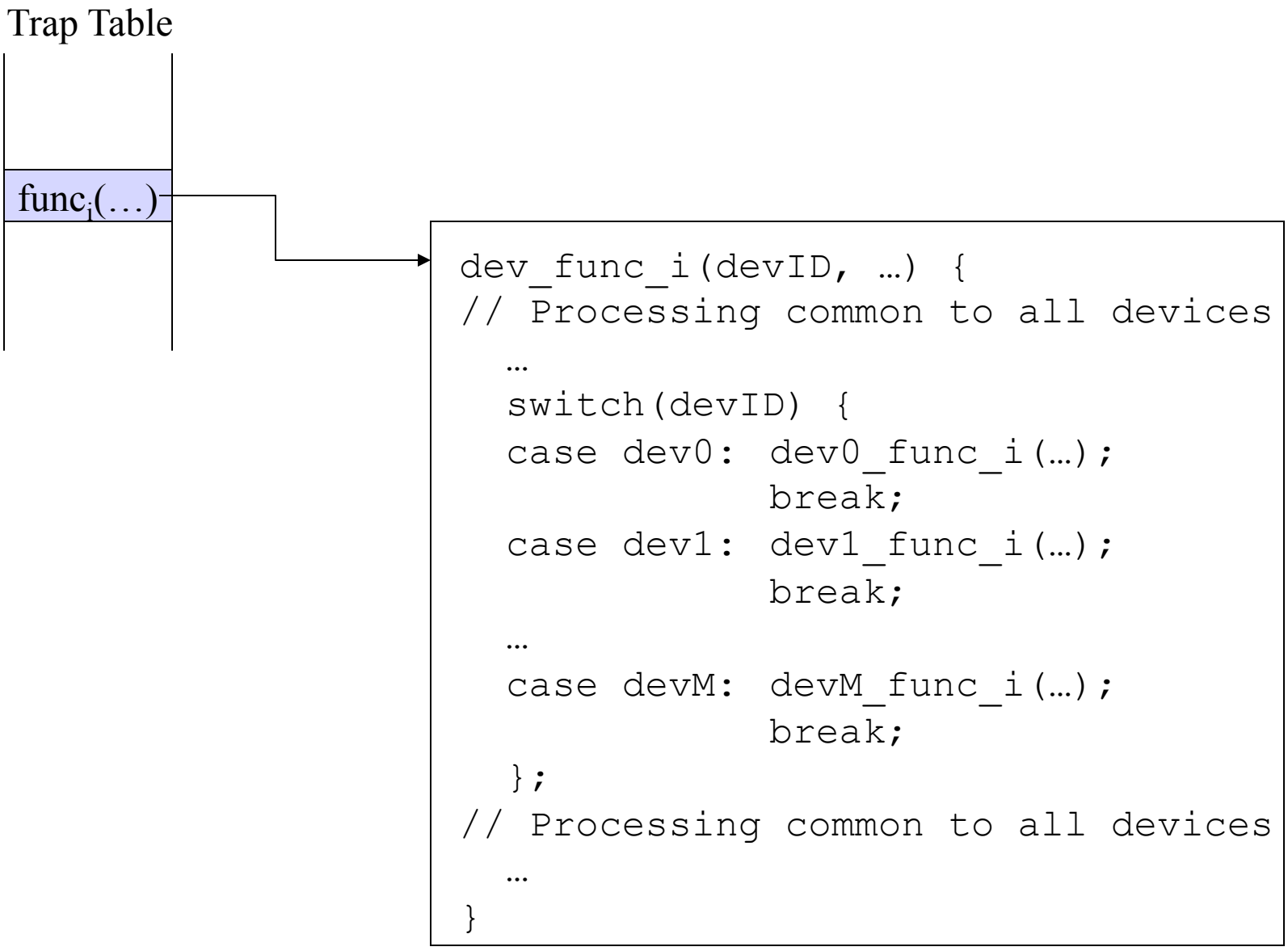
Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device Independent Part

- A set of system calls that an application program can use to invoke I/O operations
- A particular device will respond to only a subset of these system calls
  - A keyboard does not respond to *write( )* system call
- POSIX set: *open*( ), *close*( ), *read*( ), *write*( ), *lseek*( ) and *ioctl*( ).

Device Independent Function Call



Adding a New Device

- Write device-specific functions for each I/O system call
- For each system call, add a new *case* clause to the *switch* statement in device independent function call
- Compile the kernel and new drivers

Problem: Need to compile the kernel, every time a new device or a new driver is added

Loadable Kernel Modules

- A loadable kernel module (LKM) is an object file that contains code to extend a running kernel
- Windows, Linux, OS X …
- Without loadable kernel modules, an OS would have to include all possible anticipated functionality already compiled directly into the base kernel
- Linux (kernel object: *.ko* extension)
  - *modprobe ( )* high level handling of LKMs (add or remove)
  - *lsmod* to list all loaded LKMs
  - *Insmod ( )* to insert and LKM
  - *rmmod ( )* to remove an LKM
  - See */lib/modules* for the all the LKMs

insmod ( ) command

- *insmod* makes an *init_module* system call to load the LKM into kernel memory
- *init_module* system call invokes the LKM's initialization routine (also called *init_module*) right after it loads the LKM
- The LKM author sets up the initialization routine to call a kernel function that registers the subroutines that the LKM contains

LKM example: Reconfigurable Device Drivers

- Allows system administrators to add a device driver to the OS without recompiling the OS
- The new driver is first stored as a *.ko* file
  - Contains an initialization routine
- The initialization routine calls a kernel function to register the device
  - e.g. *register_chrdev*, *register_blkdev*

- An entry table stores the actual function pointers for each device specific funcion call

  dev_func_i[N]

- Replace *switch* statement with

  dev_func_i[j] (…);

- Device registration: Fill appropriate function pointers in the entry table

Universal Serial Bus (USB)

- USB is an industry standard that defines the cables, connectors and communication protocols used in a bus for connection, communication and power supply between computers and electronic devices
  - Keyboards, mouse, printers, digital cameras, etc.
- USB has effectively replaced a variety of earlier interfaces such as serial and parallel ports as well as chargers for portable devices

Firewire (IEEE 1394)

- A serial bus interface for high speed communication and real-time data transfer
- Comparable to USB

CSCI 3753
Operating Systems

Processes

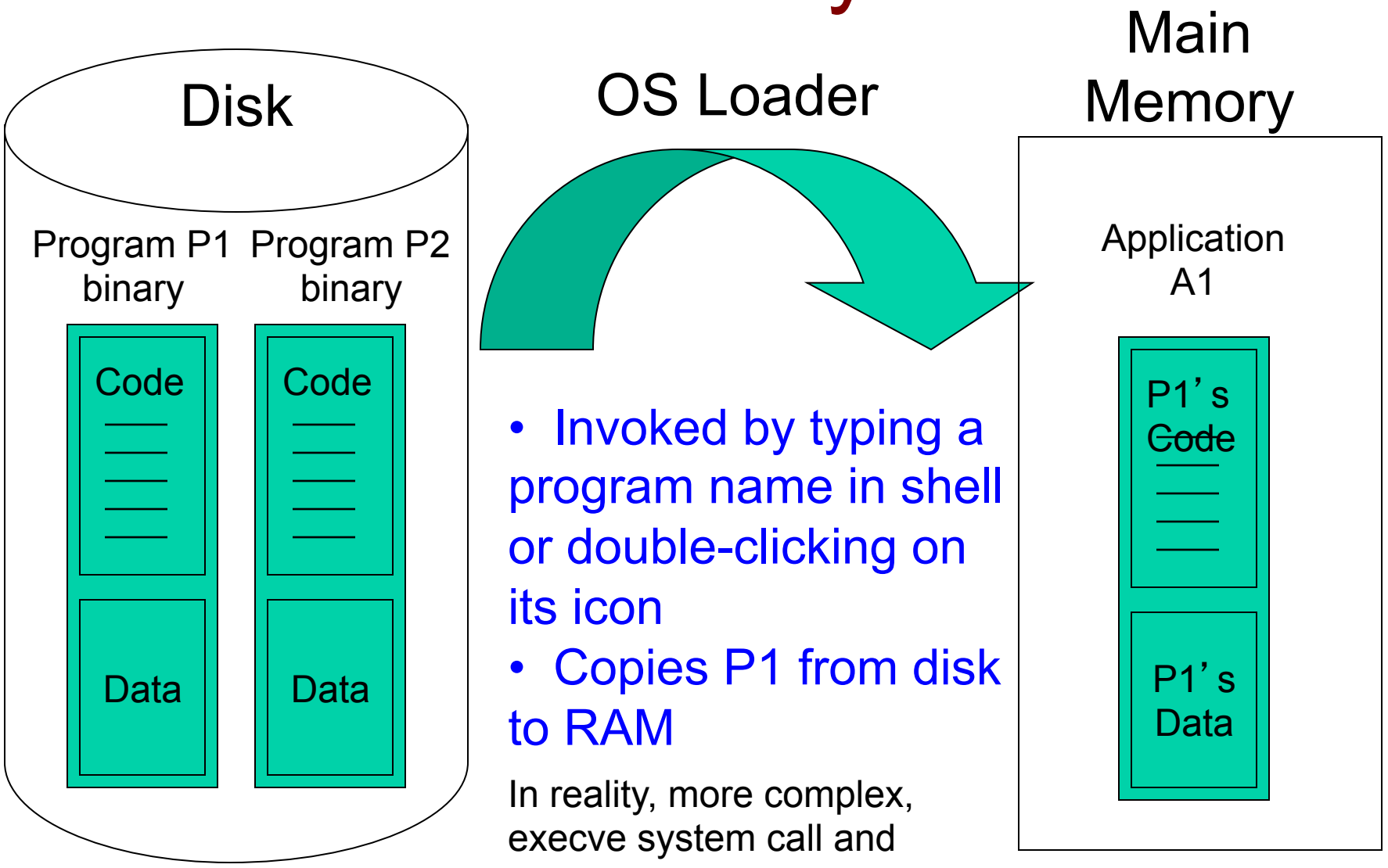
Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 01/31/13

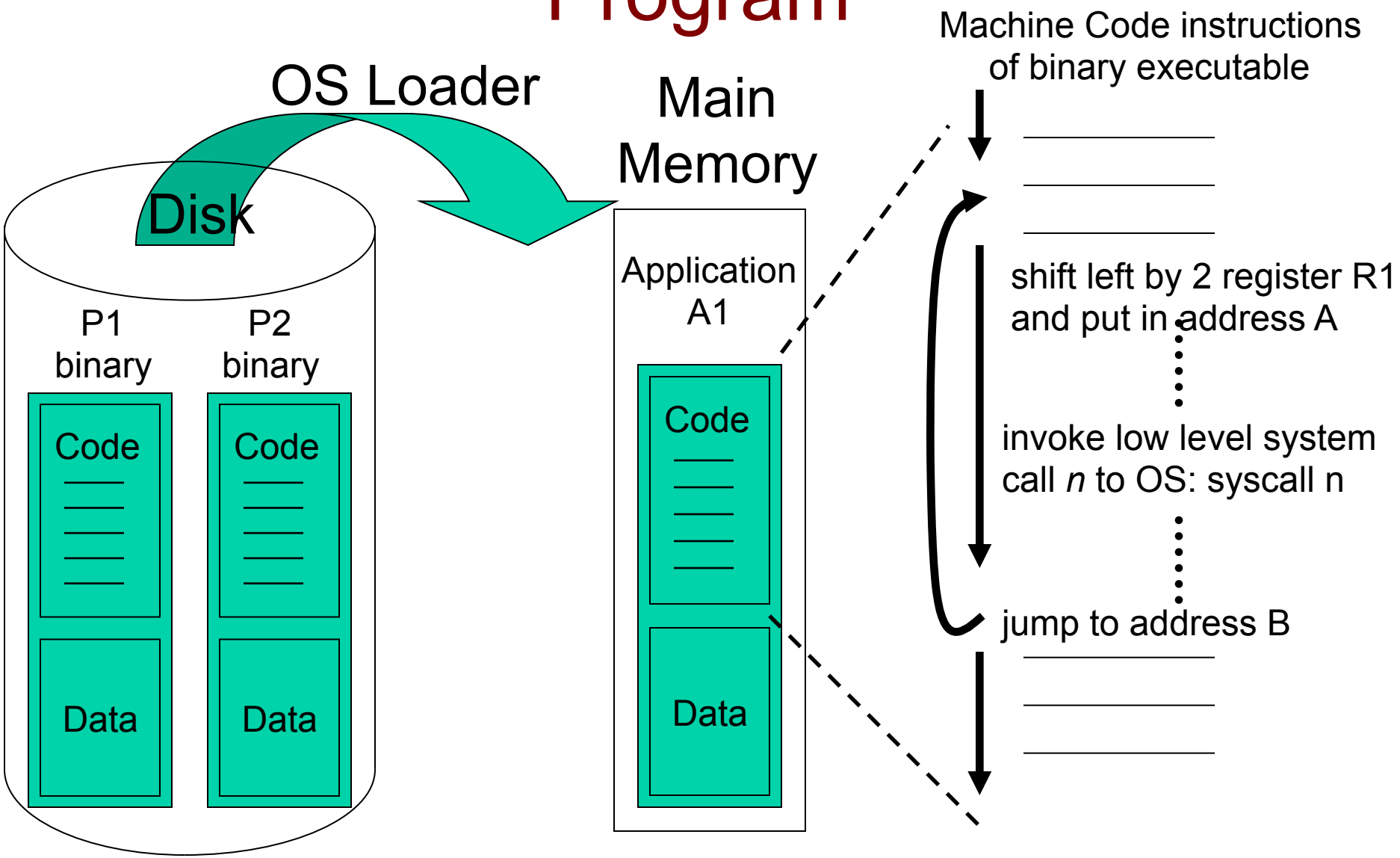
Loading a Program into Memory



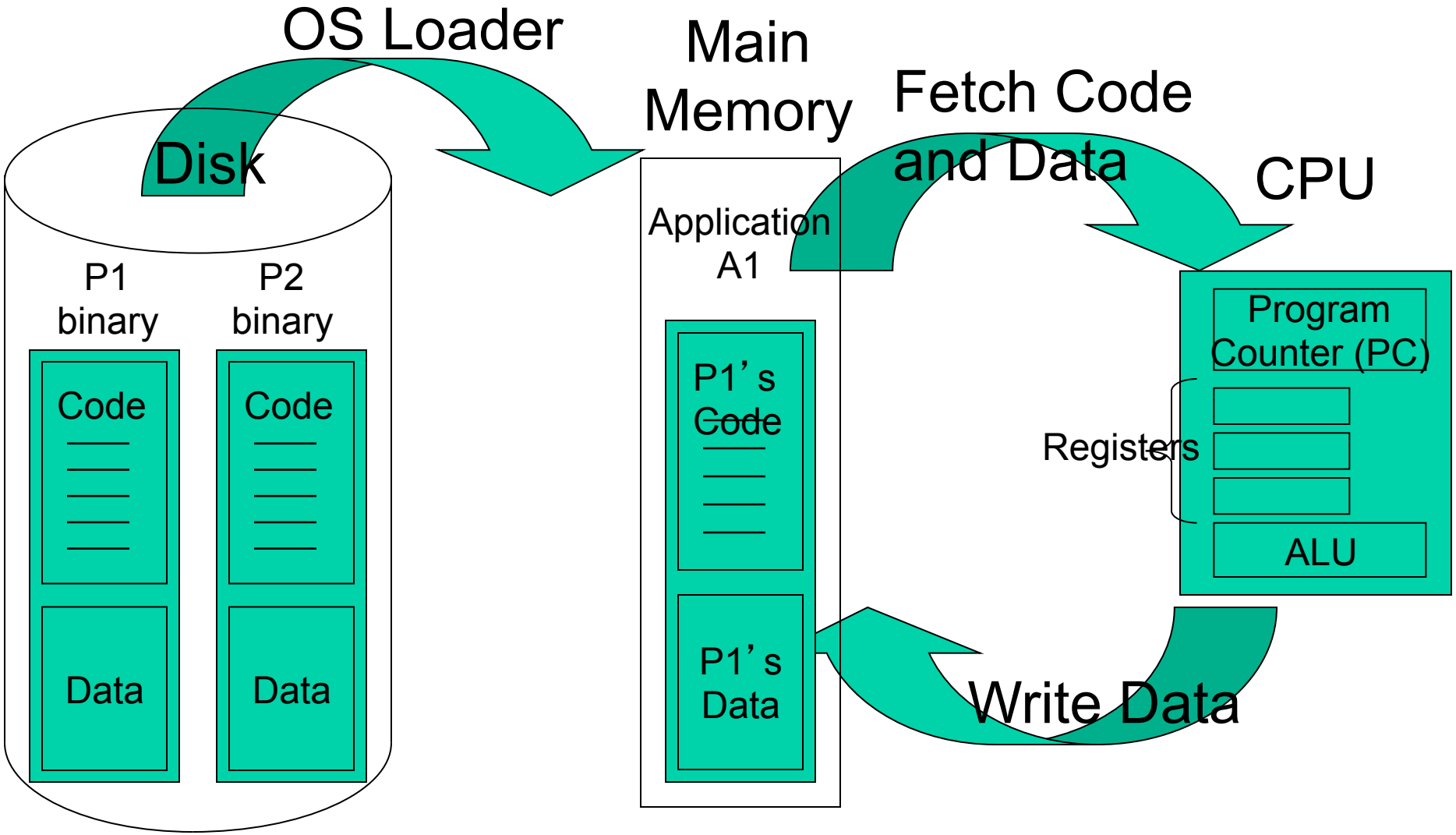
- Invoked by typing a program name in shell or double-clicking on its icon
- Copies P1 from disk to RAM

In reality, more complex, execve system call and paging are involved

Loading and Executing a Program

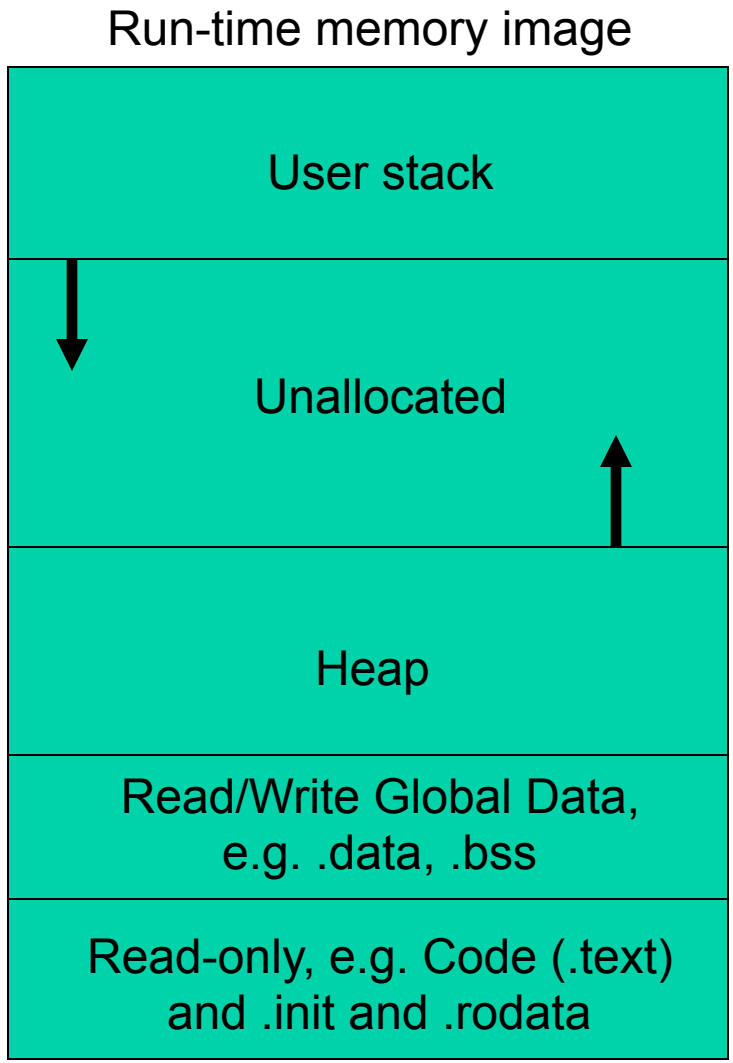


Loading and Executing a Program



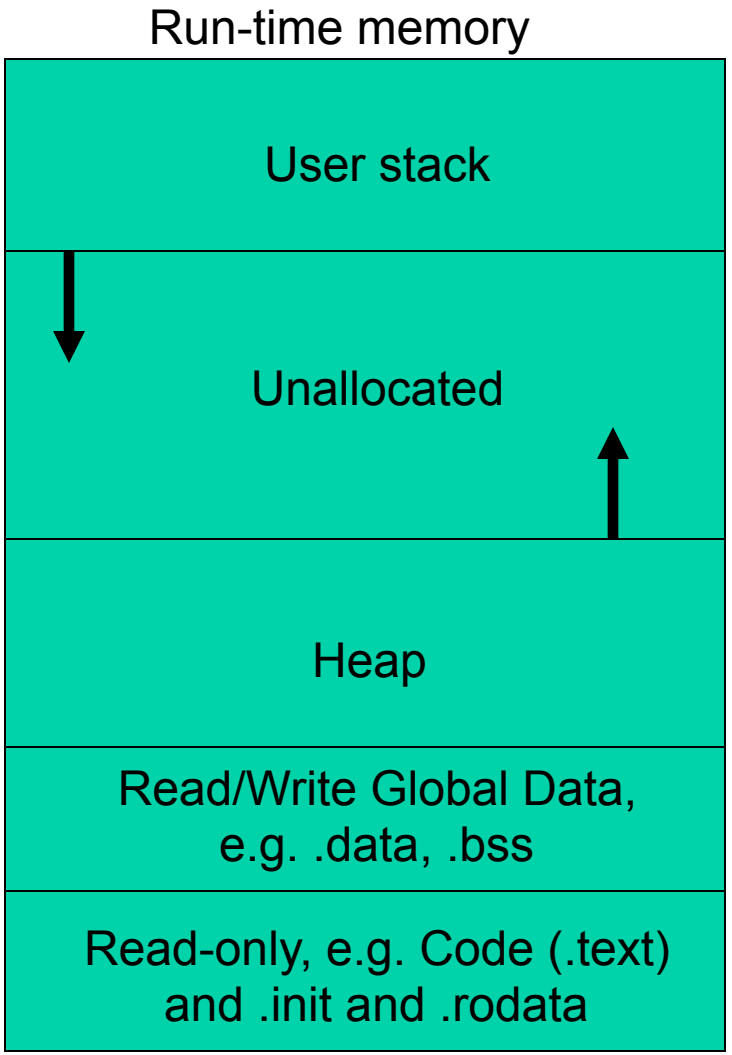
Loading Executable Object Files

- When a program is loaded into RAM, it becomes an actively executing application
- The OS allocates a stack and heap to the app in addition to code and global data.
  - A call stack is for local variables, function parameters and return addresses
  - A heap is for dynamic variables, e.g. *malloc(), new*
  - Usually, stack grows downward from high memory, heap grows upward from low memory, but this architecture-specific

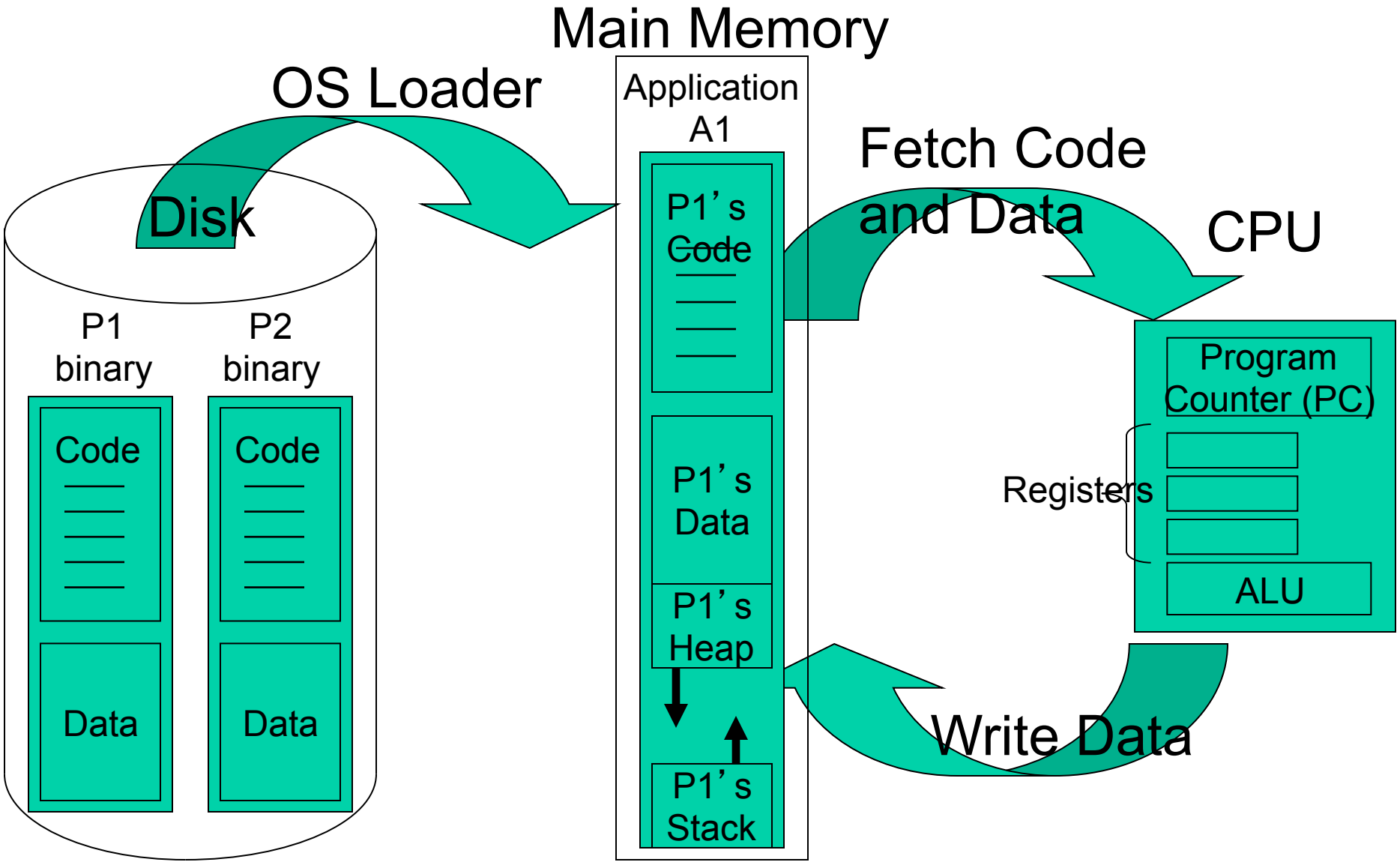


Running Executable Object Files

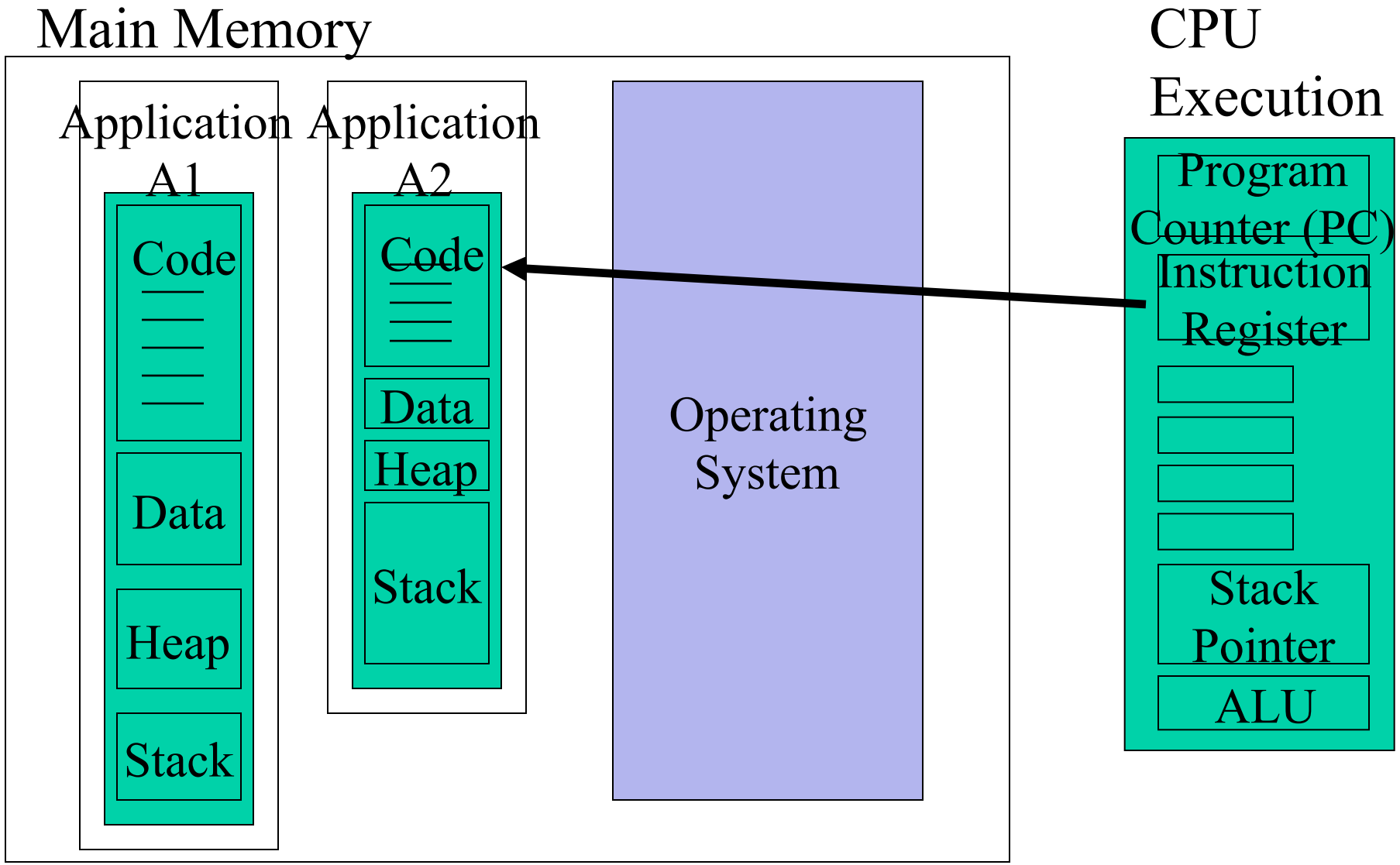
- Stack contains local variables
  - As main() calls function f1, we allocate f1's local variables on the stack
  - If f1 calls f2, we allocate f2's variables on the stack below f1's, thereby growing the stack, etc…
  - When f2 is done, we deallocate f2's local variables, popping them off the stack, and return to f1
- Stack dynamically expands and contracts as program runs and different levels of nested functions are called
- Heap contains run-time variables/ buffers
  - Obtained from malloc()
  - Program should free() the malloc'ed memory
- Heap can also expand and contract during program execution



Loading and Executing a Program – a more complete picture

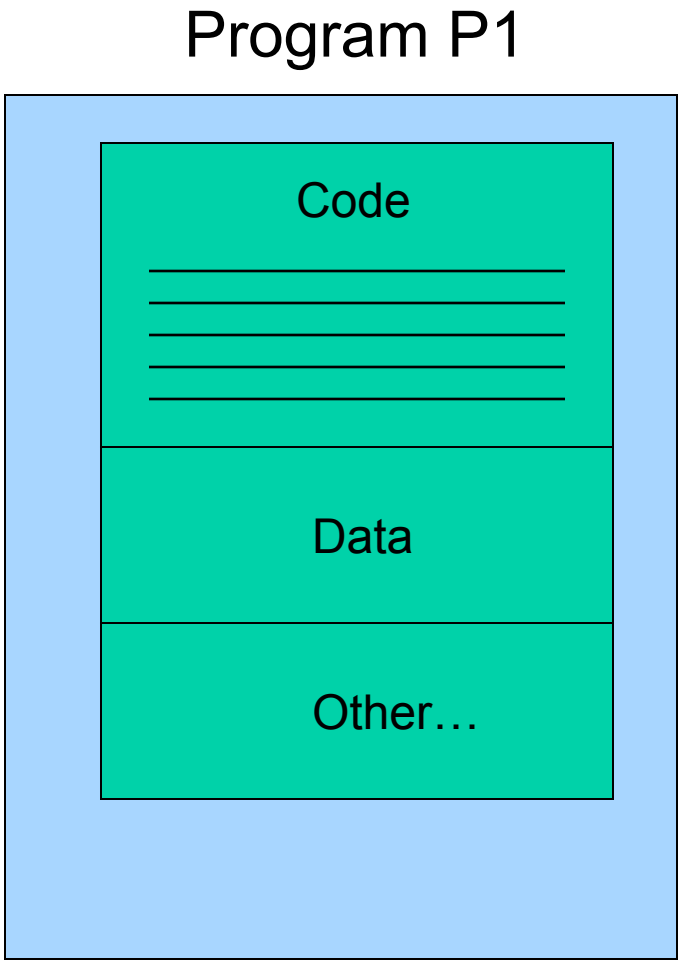


Multiple Applications + OS

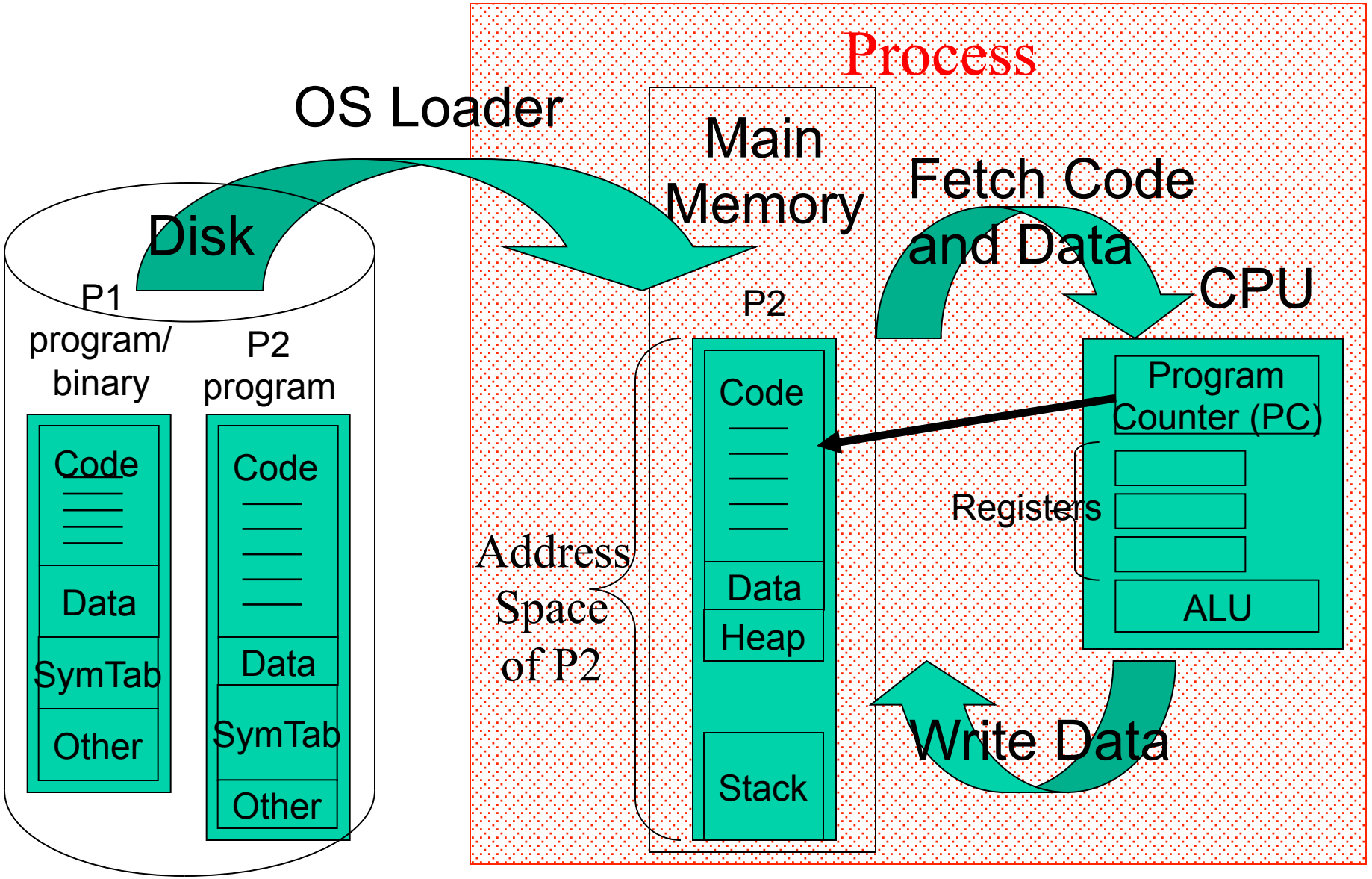


Chapter 3: What is a Process?

- A software *program* consist of a sequence of code instructions and data stored on disk
  - A program is a *passive* entity
- A *process* is a program *actively executing* from main memory within its *own address space*

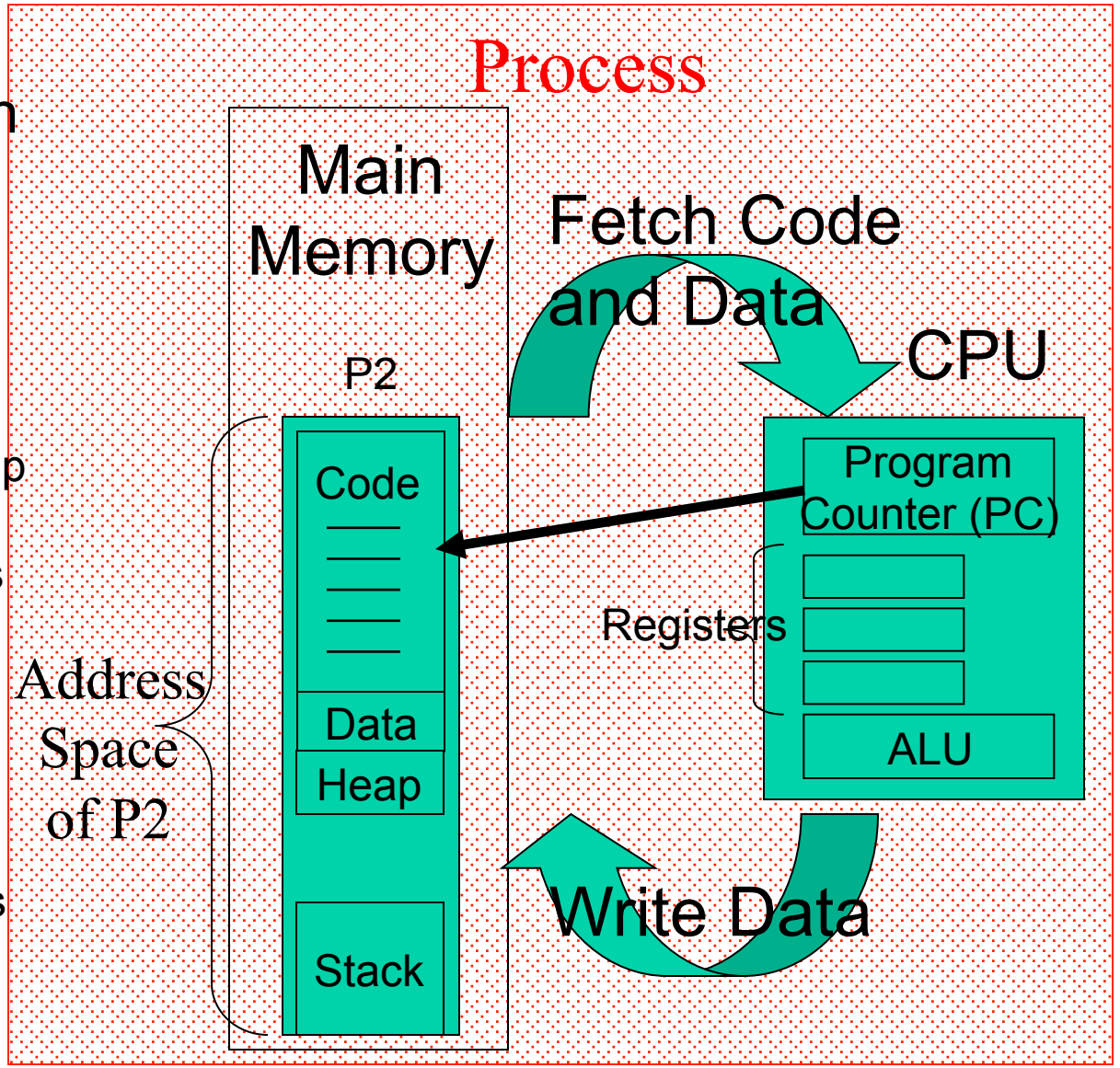


What Is a Process? (2)



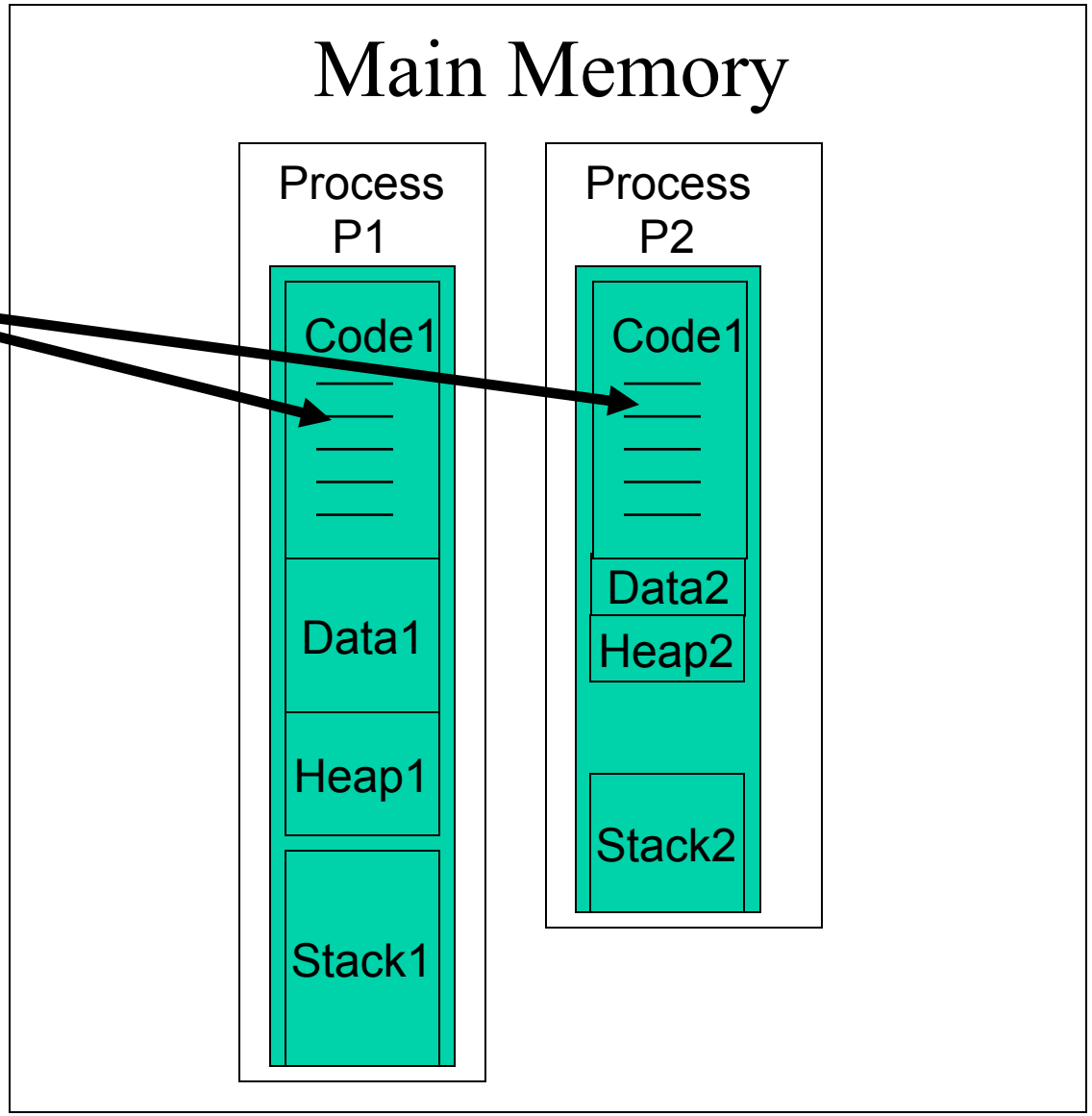
What is a Process? (3)

- A *process* is a program *actively executing* from main memory
  - has a Program Counter (PC) and execution state associated with it
    - CPU registers keep state
    - OS keeps process state in memory
    - it's alive!
  - Owns its own *address space*
    - a limited set of (virtual) addresses that can be accessed by the executing code

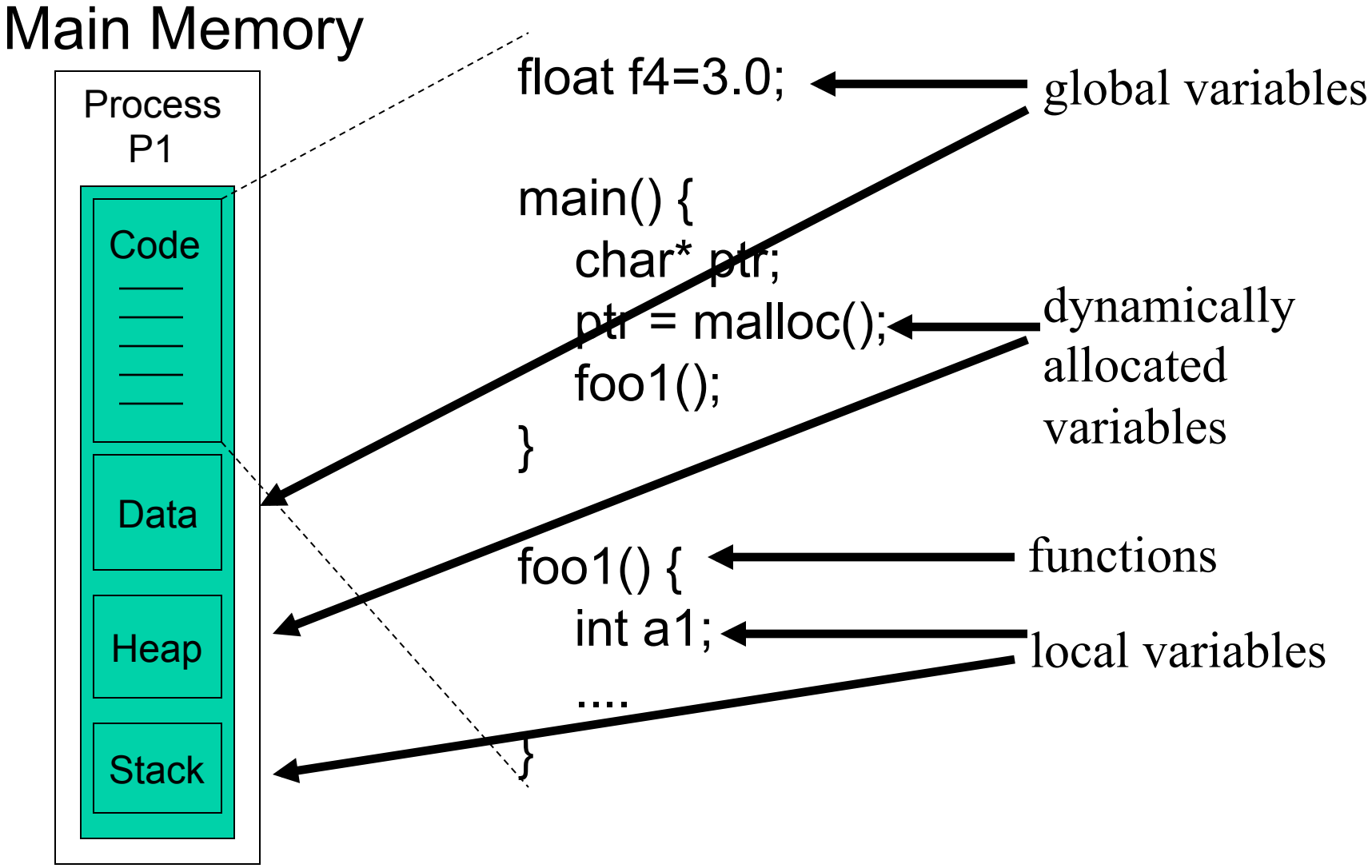


What is a Process? (4)

- 2 processes may execute the same program code, but they are considered *separate execution* sequences
  - e.g. two shell terminals

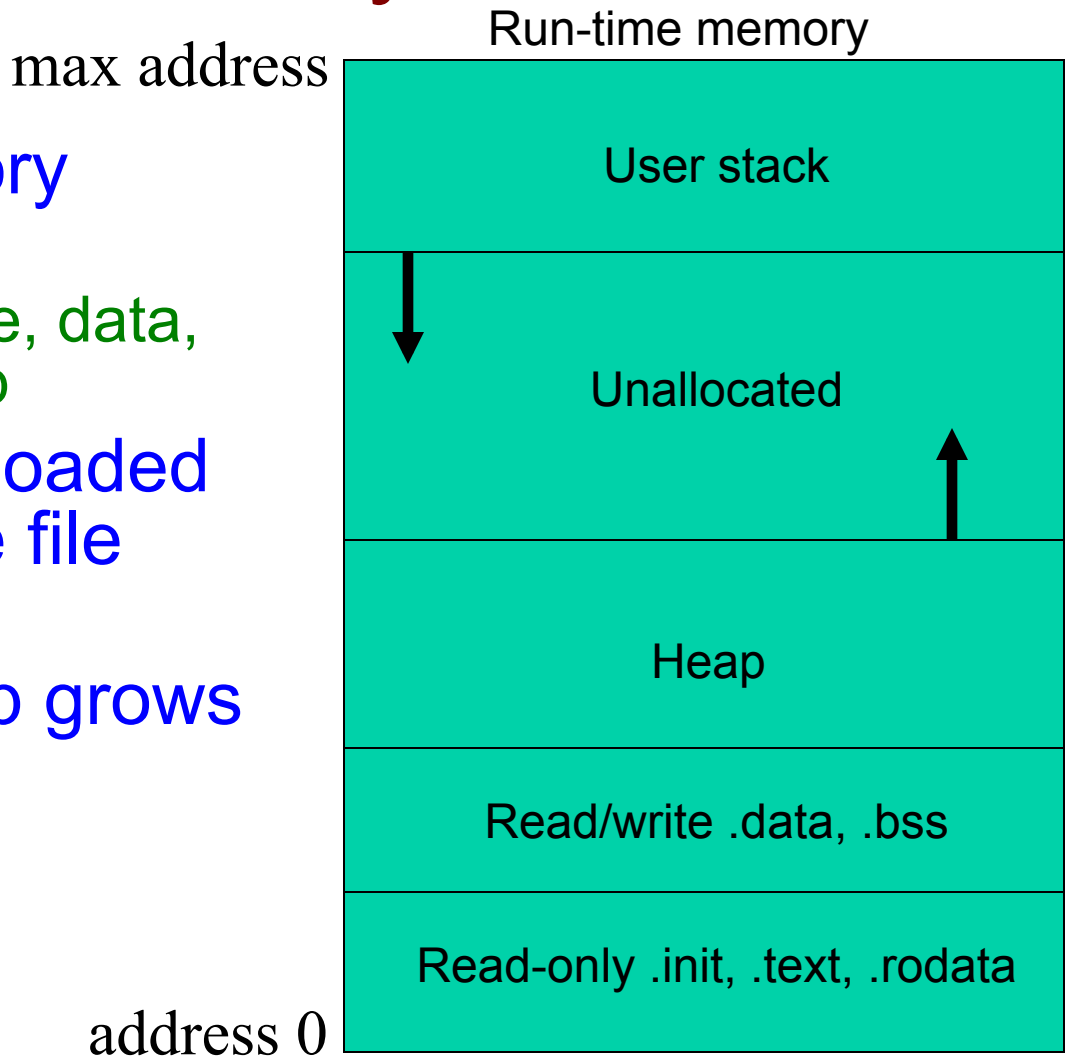


How is a Process Structured in Memory?

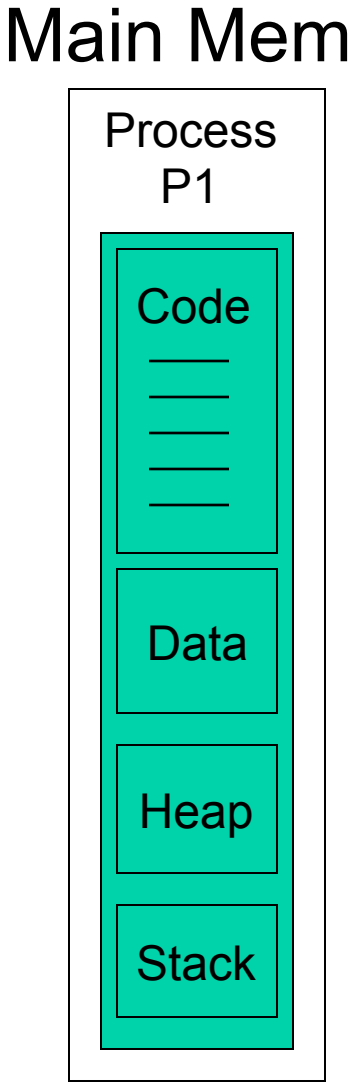


How is a Process Structured in Memory?

- Run-time memory image
  - Essentially code, data, stack, and heap
- Code and data loaded from executable file
- Stack grows downward, heap grows upward



A Process Executes in its Own Address Space

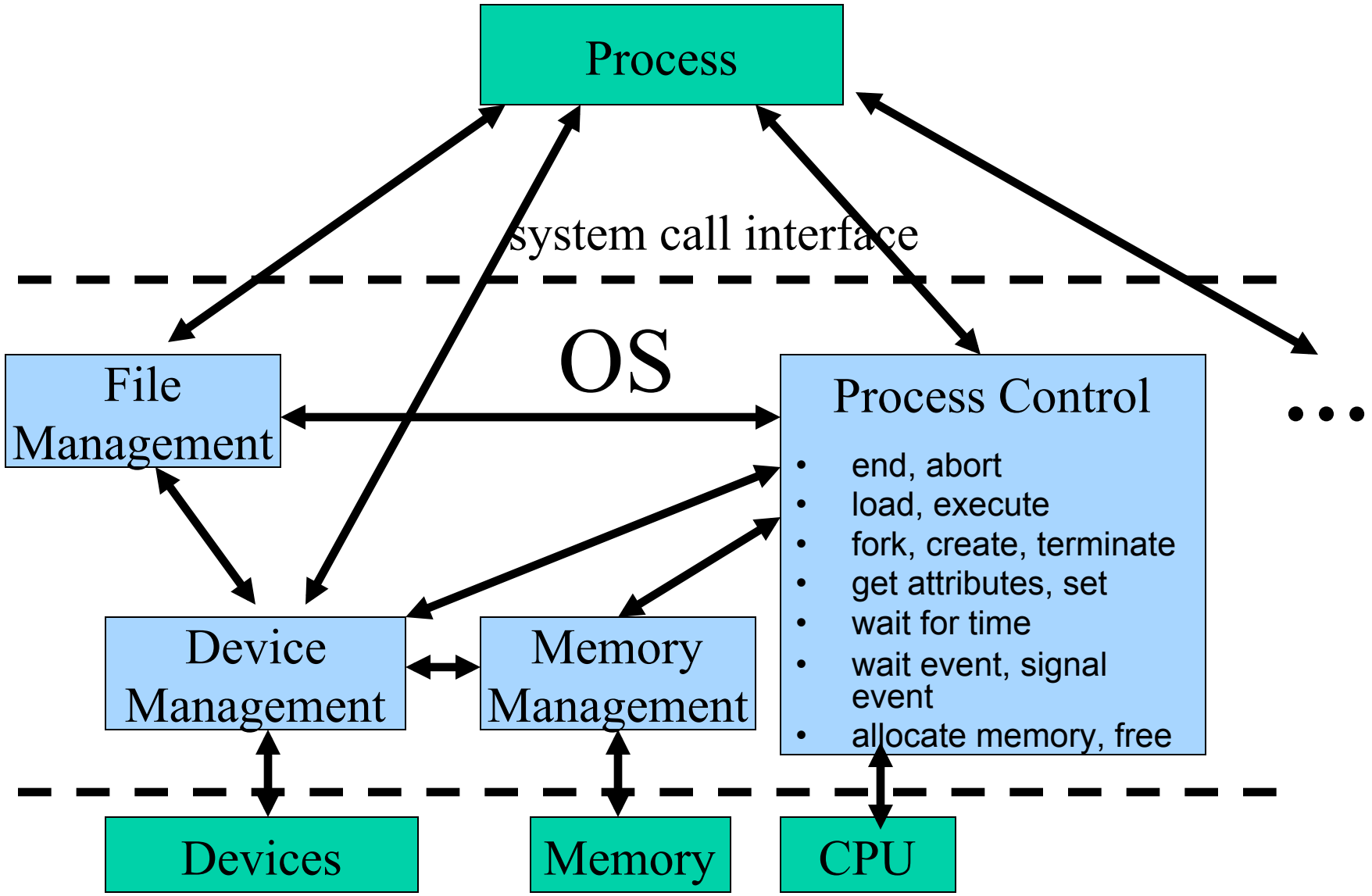


- OS tries to provide the illusion or *abstraction* to the process that it executes
  - in its own subset of RAM, i.e. its own address space
  - on its own subset (time slice) of the CPU

Applications and Processes

- Application = Σ_i Processes$_i$
  - e.g. a server could be split into multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)
  - The Application's various processes talk to each other using Inter-Process Communication (IPC). We'll see various forms of IPC later.

Process Management



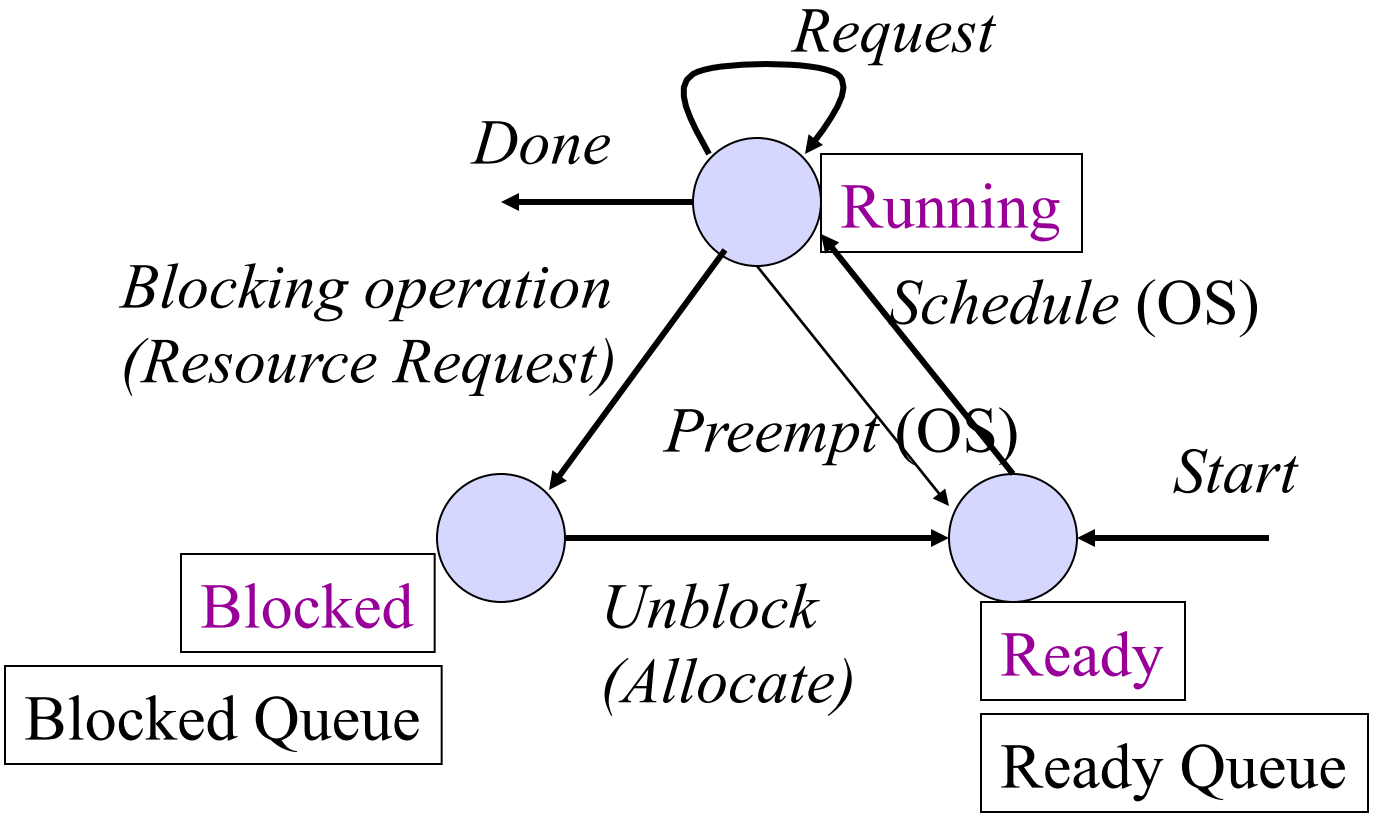
Process Manager

- Creation/deletion of processes (and threads)
- Synchronization of processes (and threads)
- Managing process state
  - Processor state like PC, stack ptr, etc.
  - Resources like open files, etc.
  - Memory limits to enforce an address space
- Scheduling processes
- Monitoring processes
  - Deadlock, protection

Process State

- Memory image: Code, data, heap, stack
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID
- Program counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

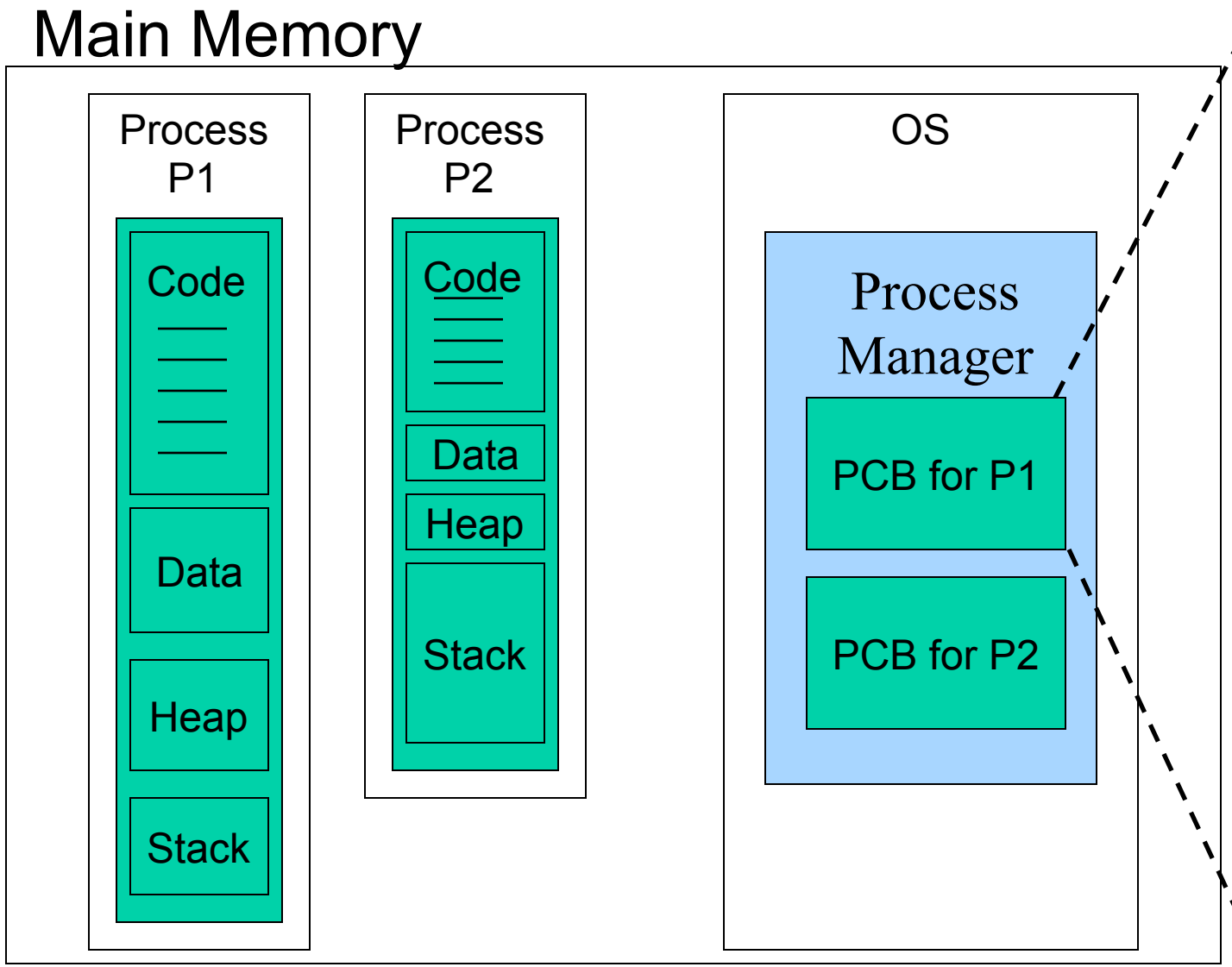
Process State Diagram



Process Control Block

- Each process is represented in OS by a process control block (PCB).
- PCB: Complete information of a process
- OS maintains a PCB table containing one entry for every process in the system.
- PCB table is typically of fixed size. This size determines the maximum number of processes an OS can have
  - The actual maximum may be less due to other resource constraints, e.g. memory.

Process Control Block (PCB)



- Process state, e.g. ready, running, or waiting
- accounting info, e.g. process ID
- Program Counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

A PCB is also called a Process Descriptor

Context Switch

- Running state → Ready state
- Running state → Blocked state
- Switching the CPU from currently running process to another process
  - Save the state of the currently running process in its PCB
  - Load the saved state of new process scheduled to run from its PCB
  - Context switch time is pure overhead: 1 – 1000 microseconds.
  - An important goal in OS design is to minimize context switch time.

Creating Processes

- In Windows, there is a CreateProcess() call
  - Pass an argument to *CreateProcess()* indicating which program to start running
  - Invokes a system call to OS that then invokes process manager to:
    - allocate space in memory for the process
    - Set up PCB state for process, assigns PID, etc.
    - Copy code of program name from hard disk to main memory, sets PC to entry point in *main()*
    - Schedule the process for execution
  - As we will see, this combines UNIX's *fork()* and exec() system calls and achieves the same effect

Creating Processes in UNIX

- Use *fork()* command to create/spawn new processes from within a process
  - When a process (called parent process) calls *fork ()*, a new process (called child process) is created
  - Child process is an exact copy of the parent process
    - All addresses are appropriately mapped – We'll see this later under memory management
  - The child starts executing at the same point as the parent, namely just after returning from the *fork()* call

fork ()

- The fork() call returns an *int* value
  - In the parent process, returned value is child's PID
  - In the child, returned value is 0
  - Since both parent and child execute the same code starting from the same place, i.e. just after the fork(), then to differentiate the child's behavior from the parent's, you could add code:

```
PID = fork();
if (PID==0) { /* child */
   codeforthechild();
   exit(0);
}
/* parent's code here */
```

Loading Processes

- The exec() system call loads program code into the calling process's memory (same address space!), clears the stack, and begins executing the new code at its main entry point
  - The calling code is erased!
  - Use *fork()* and *exec()* (actually *execve()*) to create a new process executing a new program in a new address space

```
PID = fork();
if (PID==0) { /* child */
   exec("/bin/ls");
   exit(0);
}
/* the parent's code here */
```

More on Processes

- Copying the entire code of a parent into a child can be expensive on a *fork()*, so
  - at start, child can share parent's code pages. Only create a copy on a write.
- The *wait()* system call is used by a parent process to be informed of when a child has completed, i.e. called *exit()*
  - Once the parent has called *wait()*, the child's PCB and address space can be freed

More about this in recitation

Process Hierarchy

- OS creates a single process at the start up.
- An existing process can spawn one or more new processes during execution
  - Parent-child relationship
  - A parent process may have some control over its child process(es): suspend/activate execution; wait for termination; etc.
- A tree-structured hierarchy of processes
- Process hierarchy in Unix

<Figure>

Communicating Between Processes

- Inter-Process Communication (IPC): Want to communicate between two processes because (why?)
  - An application may split its tasks into two or more processes for reasons of convenience and/or performance
    - e.g. Web server and cgi-bin (Common Gateway Interface) – for certain URLs, Web server creates a new cgi-bin process to service the request, e.g. dynamic content generation
      - Creating a new process is expensive
  - Sharing information
  - Improved fault isolation

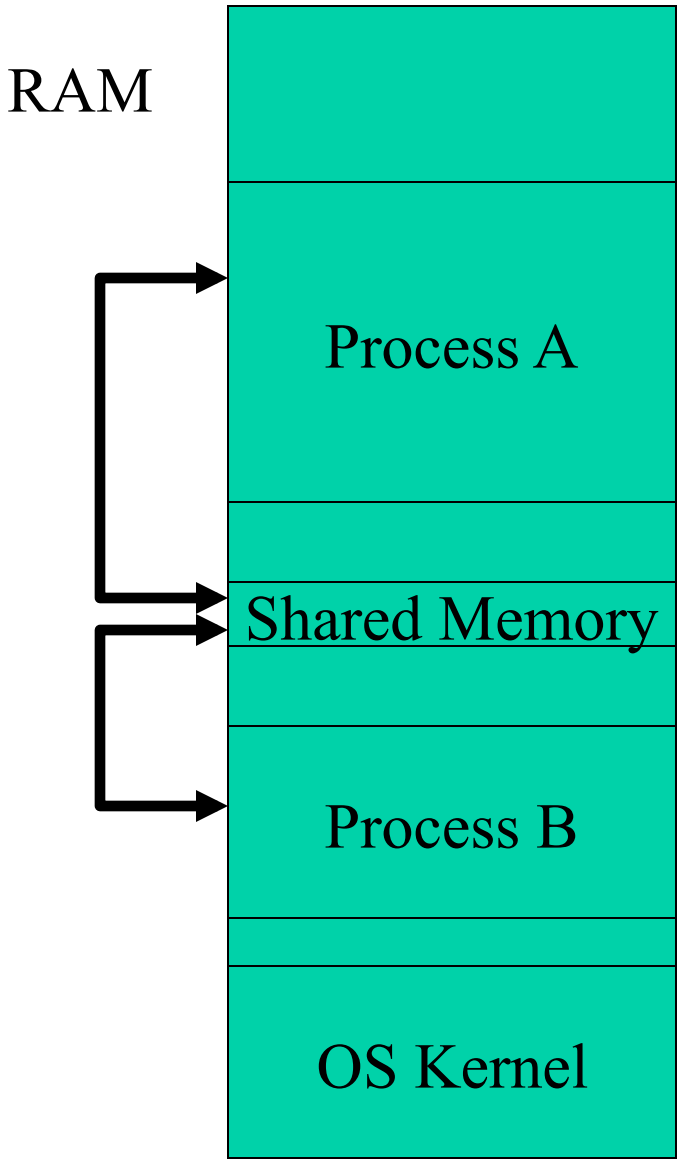
Communicating Between Processes

- Two types of IPC
  1. *shared memory* - OS provides mechanisms that allow creation of a shared memory buffer between processes
     - shmid = *shmget* (key name, size, flags) is part of the POSIX API that creates a shared memory segment, using a name (key ID)
       - – All processes sharing the memory need to agree on the key name in advance.
       - – Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory. If it already exists, then just return the handle.

Communicating Between Processes (2)

- Two types of IPC
  1. *shared memory* (cont.)
     - shm_ptr = *shmat* (shmid, NULL, 0) to attach a shared memory segment to a process's address space
       - This association is also called binding
       - Reads and writes now just use shm_ptr
     - *shmctl*() to modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

IPC Shared Memory

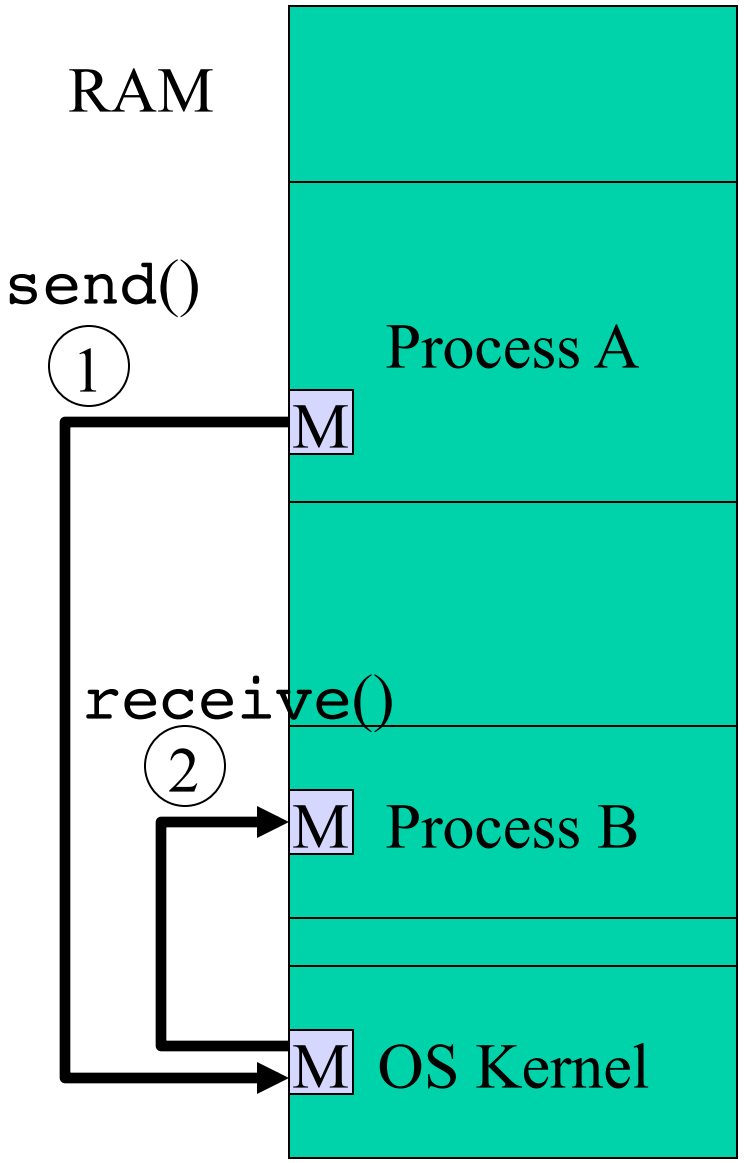


- allows fast and high volume reading/writing of buffer in memory
- applies to processes on the same machine
- Problem: shared access to the same memory introduces complexity
  - need to synchronize access
  - Producer-Consumer example
    - if two producers write at the same time to shared memory, then they can overwrite each other's data
    - if a producer writes while a consumer is reading, then the consumer may read inconsistent data

IPC Message Passing

- Two types of IPC
  2. *message passing* - OS provides constructs that allow communication via buffers
     - Basic primitives are `send`() and `receive`()
     - typically implemented via system calls, and is hence slower than shared memory
     - Sending process has a buffer to send/receive messages, as does the receiving process and OS
     - In direct message-passing, processes send directly to each other's buffers
     - In indirect message-passing, sending process sends a message to OS by calling send(). OS acts as a buffer or mailbox for relaying the message to the receiving process, which calls receive() to retrieve message

IPC Message Passing



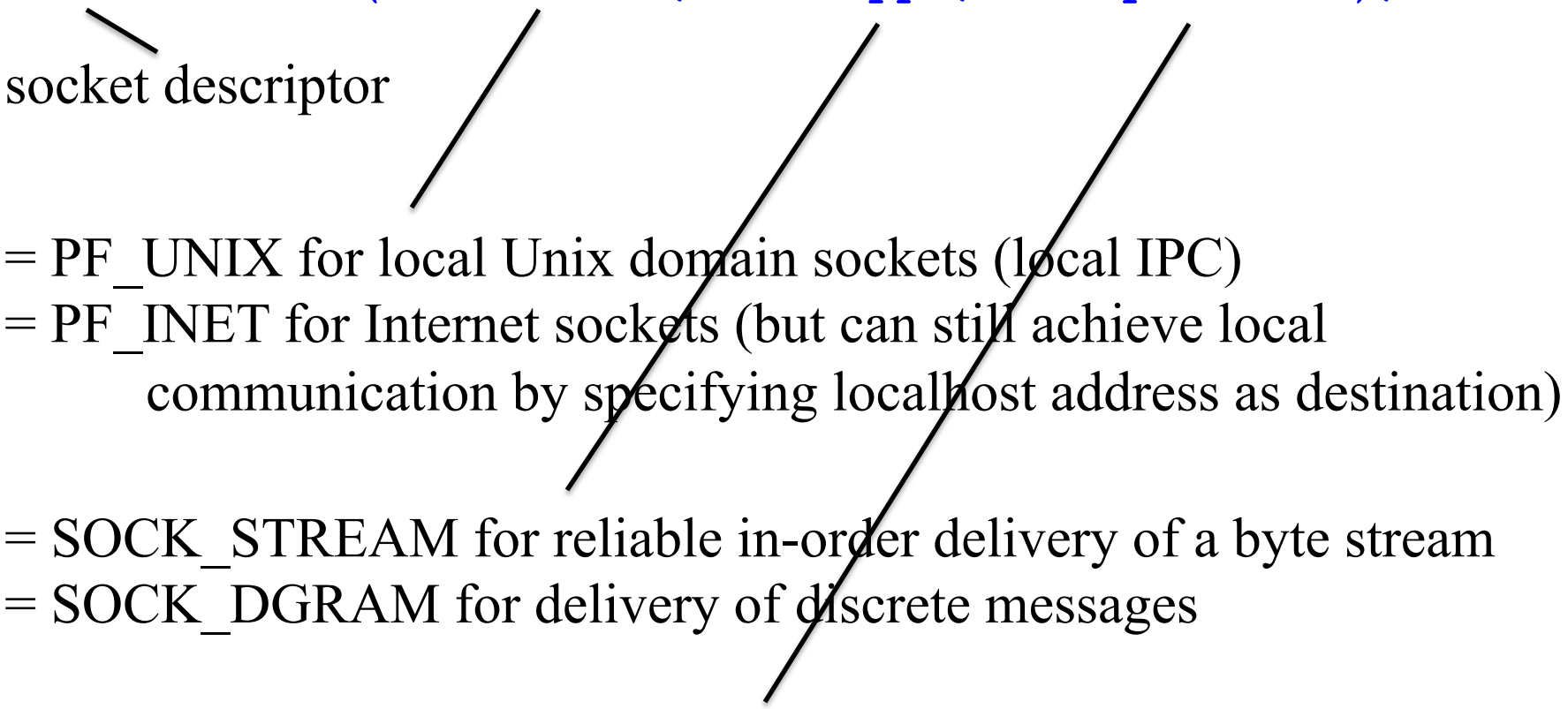
Example of indirect message-passing

- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- used to pass small messages
- Advantage: doesn't require synchronization
- Disadvantage: Slow - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

Using Sockets for UNIX IPC

Sockets are an example of message-passing IPC. Created in UNIX using socket() call:

```
sd = socket(int domain, int type, int protocol);
```



socket descriptor

= PF_UNIX for local Unix domain sockets (local IPC)
= PF_INET for Internet sockets (but can still achieve local communication by specifying localhost address as destination)

= SOCK_STREAM for reliable in-order delivery of a byte stream
= SOCK_DGRAM for delivery of discrete messages

= 0 usually to select default protocol associated with a type

Using Sockets for UNIX IPC (2)

- Each communicating process will first create its own socket, usually SOCK_STREAM.
- Now let's consider only the case of UNIX domain sockets (PF_UNIX domain):
  - Used only for local communication only among a computer's processes
  - Emulates reading/writing from/to a file
  - Each process `bind()`'s its socket to a filename:

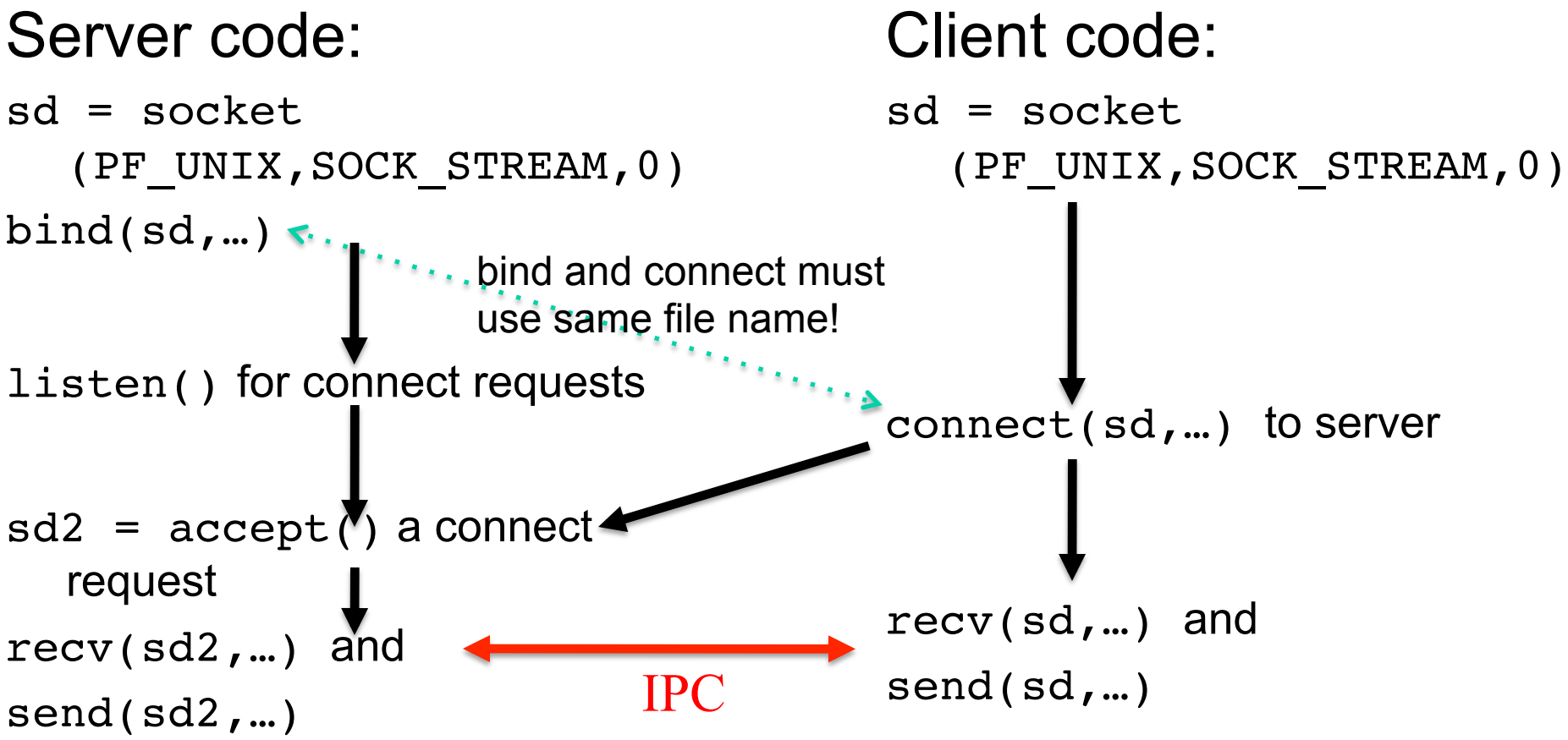
```
bind(sd, (struct sockaddr *)&local, length);
```

socket descriptor

data structure containing unique unused file name, e.g. "/users/shiv/myipcsocketfile"

Using Sockets for UNIX IPC (3)

- UNIX domain sockets (cont.):
  - Usually, one process acts as the server, and the other processes connect to it as clients



IPC via Internet domain sockets

- Set up and connect Internet sockets in a way very similar to Unix domain sockets, but…
  - Configure the socket with domain PF_INET instead and set destination to localhost (say 127.0.0.1) instead of the usual remote Internet IP address
  - And need to choose a well-known port # that is shared between processes, i.e. P1 and P2 know this port # in advance – this is similar to the well-known file name
  - Both processes then send() and receive() messages via this port and socket
  - Advantage of writing applications to use this type of IPC via Internet domain sockets is that it is arguably more portable than UNIX-domain sockets

CSCI 3753
Operating Systems

Interprocess Communication

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 02/06/13

Communicating Between Processes

- Inter-Process Communication (IPC): Want to communicate between two processes because (why?)
  - An application may split its tasks into two or more processes for reasons of convenience and/or performance
    - e.g. Web server and cgi-bin (Common Gateway Interface) – for certain URLs, Web server creates a new cgi-bin process to service the request, e.g. dynamic content generation
      - Creating a new process is expensive
  - Sharing information
  - Improved fault isolation

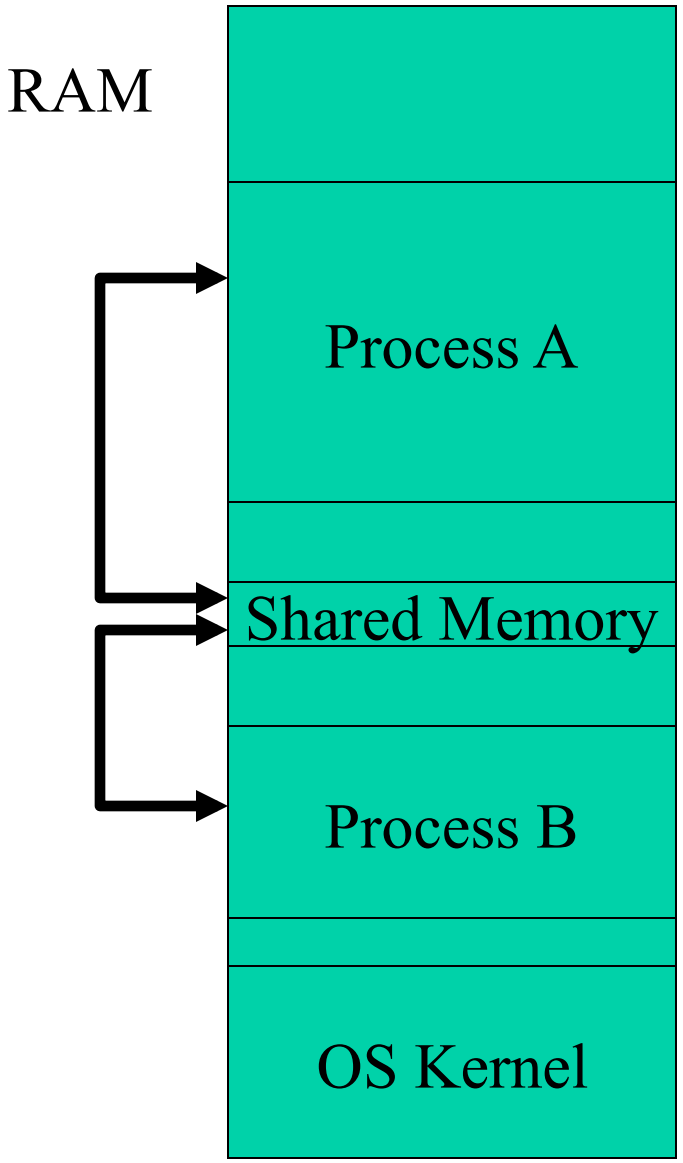
Communicating Between Processes

- Two types of IPC
  1. *shared memory* - OS provides mechanisms that allow creation of a shared memory buffer between processes
     - shmid = *shmget* (key name, size, flags) is part of the POSIX API that creates a shared memory segment, using a name (key ID)
       - – All processes sharing the memory need to agree on the key name in advance.
       - – Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory. If it already exists, then just return the handle.

Communicating Between Processes (2)

- Two types of IPC
  1. *shared memory* (cont.)
     - shm_ptr = *shmat* (shmid, NULL, 0) to attach a shared memory segment to a process's address space
       - This association is also called binding
       - Reads and writes now just use shm_ptr
     - *shmctl*() to modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

IPC Shared Memory

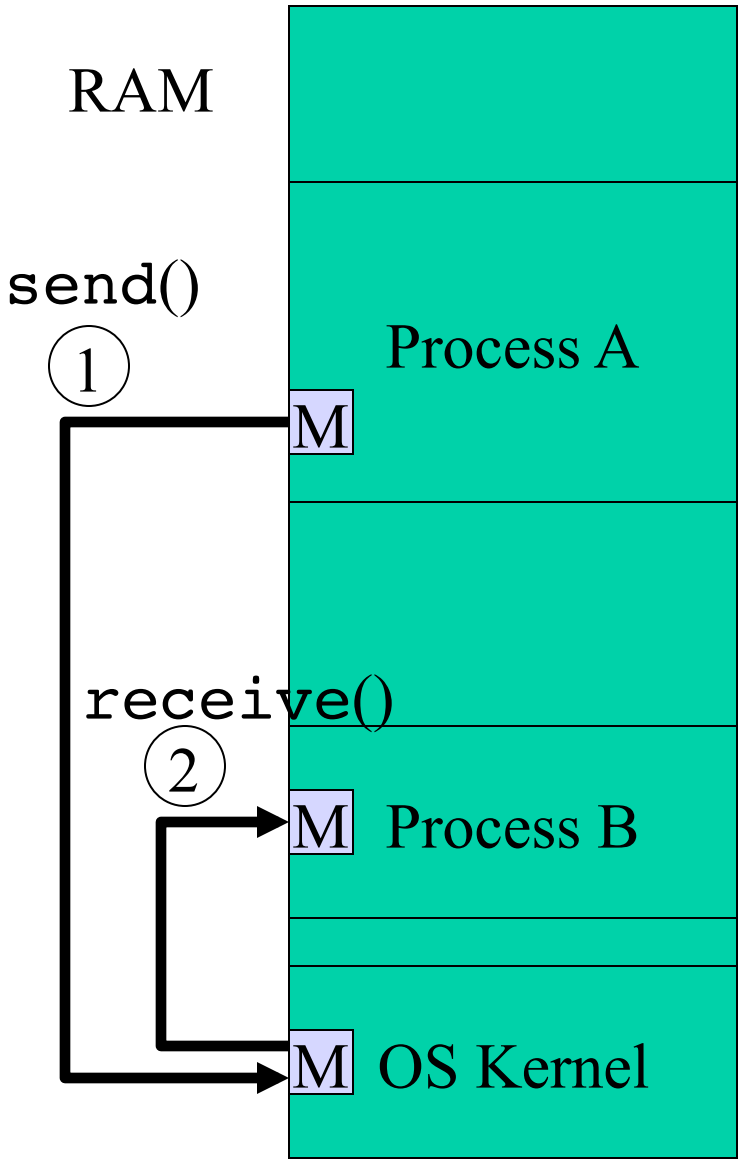


- allows fast and high volume reading/writing of buffer in memory
- applies to processes on the same machine
- Problem: shared access to the same memory introduces complexity
  - need to synchronize access
  - Producer-Consumer example
    - if two producers write at the same time to shared memory, then they can overwrite each other's data
    - if a producer writes while a consumer is reading, then the consumer may read inconsistent data

IPC Message Passing

- Two types of IPC
  2. *message passing* - OS provides constructs that allow communication via buffers
     - Basic primitives are `send`() and `receive`()
     - typically implemented via system calls, and is hence slower than shared memory
     - Sending process has a buffer to send/receive messages, as does the receiving process and OS
     - In direct message-passing, processes send directly to each other's buffers
     - In indirect message-passing, sending process sends a message to OS by calling send(). OS acts as a buffer or mailbox for relaying the message to the receiving process, which calls receive() to retrieve message

IPC Message Passing



Example of indirect message-passing

- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- used to pass small messages
- Advantage: doesn't require synchronization
- Disadvantage: Slow - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

IPC via Pipes

- Process 1 writes into one end of the pipe, & process 2 reads from other end of the pipe, e.g. "ls | more"
  - Form of IPC similar to message-passing but data is viewed as a stream of bytes rather than discrete messages
  - was one of UNIX's original forms of IPC
  - essentially FIFO buffers accessed like file I/O API, so standard read() and write() for files can be used
  - Asynchronous/non-blocking send() and blocking/ synchronous receive()
- This is a one-way pipe
- This also called an *anonymous* pipe in Windows
  - Parent process uses `pipe()` system call to create pipe

IPC via Pipes (2)

Unix-style pseudocode example:

```
int pid;
int piped[2];

pipe(piped);
pid = fork();
If (pid==0) { /* child */
   /* childs blocks on read */
   read(piped[0], readdata, length)
} elseif (pid>0) {  /* parent */
   /* parent writes data to child */
   write(piped[1], writedata, length);
}
```

piped[0] is file descriptor to read end of the pipe

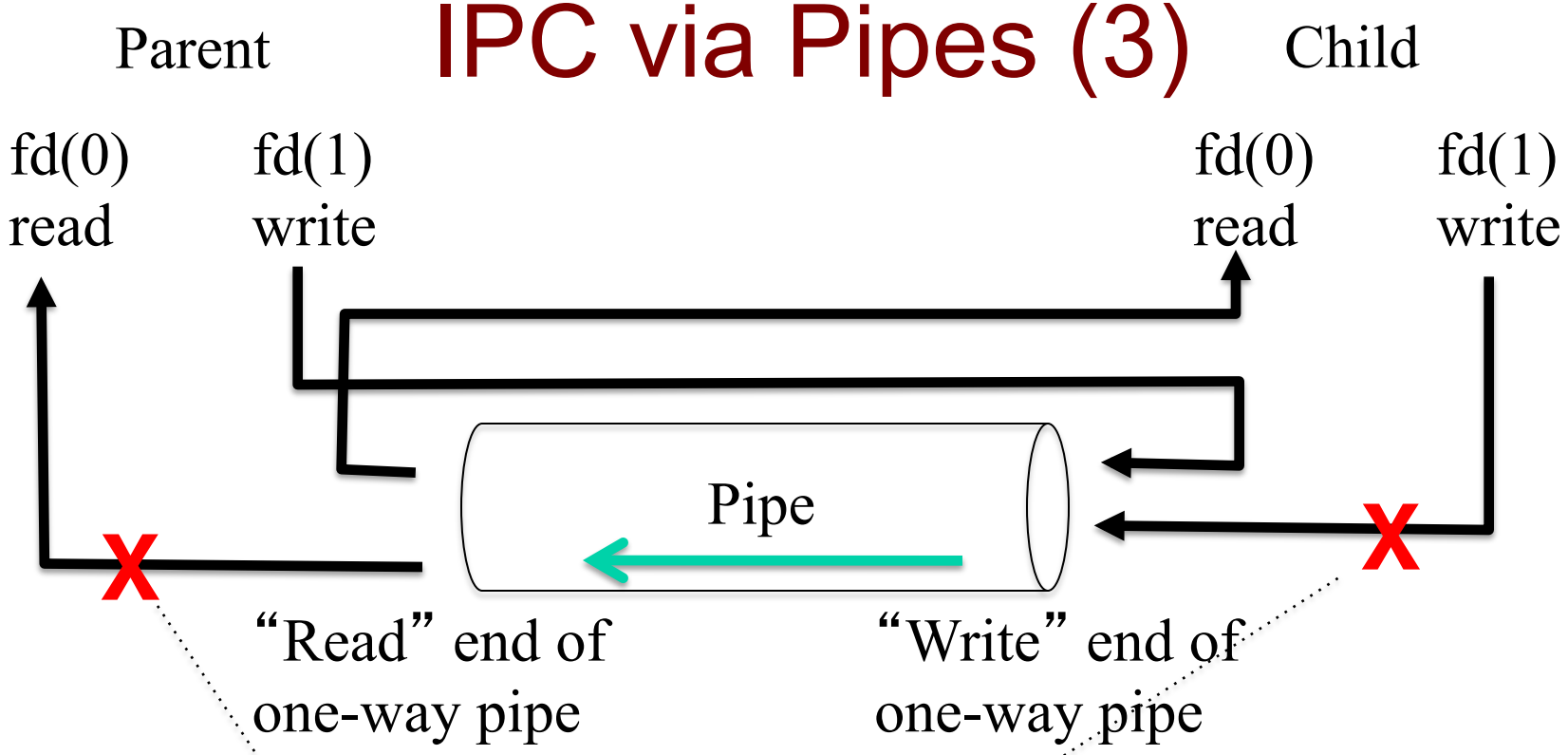
piped[1] is file descriptor to write end of pipe

System call to create pipe

Once there are *length* bytes, read returns, so as the parent sender streams bytes, the child reader can process them

Send message to child

IPC via Pipes (3)



- Chapter 3 textbook example is more detailed, e.g. closes the unused write fd for child and closes the unused read fd for parent

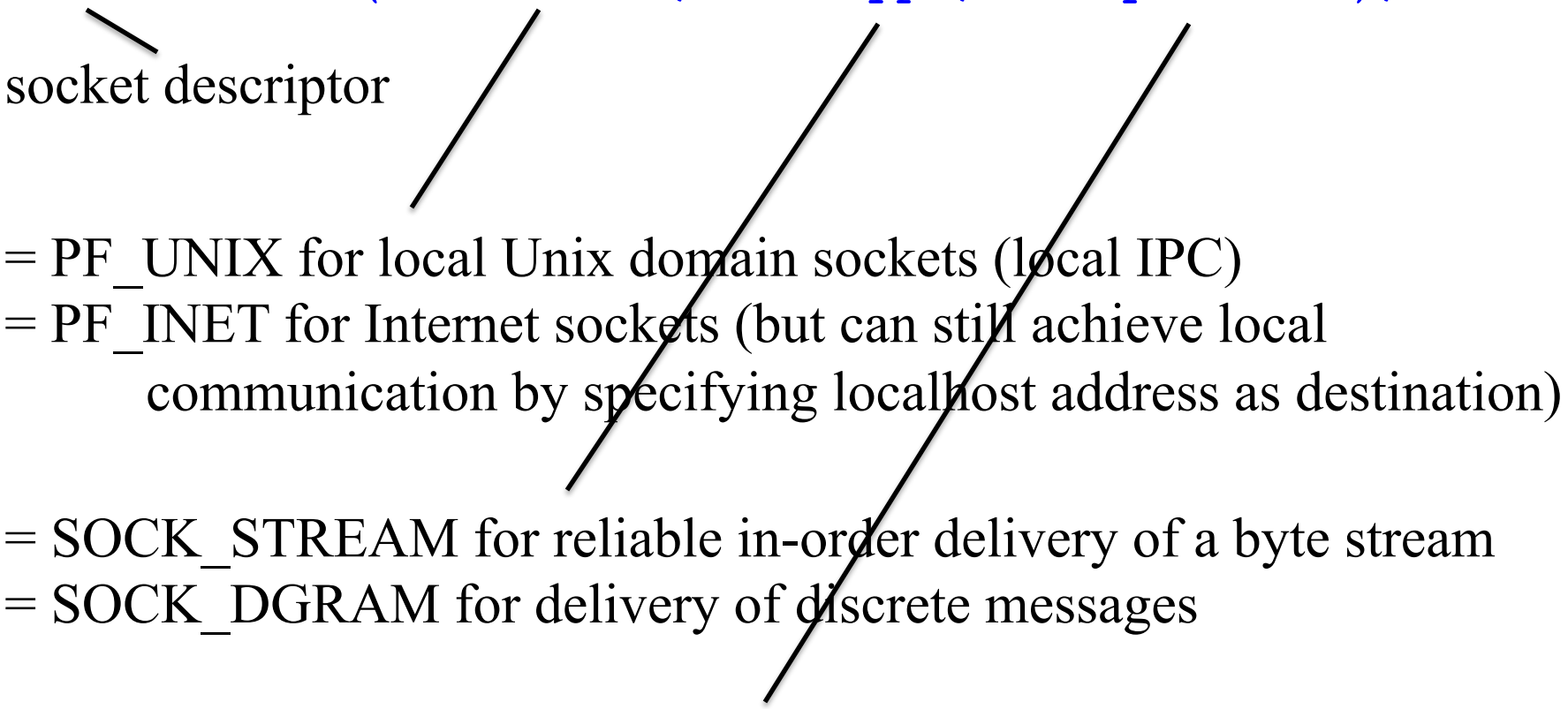
Named Pipes

- Traditional one-way or anonymous pipes only exist transiently between the two processes connected by the pipe
  - As soon as these processes complete, the pipe disappears
- Named pipes persist across processes
  - Operate as FIFO buffers or files, e.g. created using mkfifo(unique_pipe_name) on Unix
  - Different processes can attach to the named pipe to send and receive data
  - Need to explicitly remove the named pipe
  - See textbook for more info on named pipes

Using Sockets for UNIX IPC

Sockets are an example of message-passing IPC. Created in UNIX using socket() call:

```
sd = socket(int domain, int type, int protocol);
```



socket descriptor

= PF_UNIX for local Unix domain sockets (local IPC)
= PF_INET for Internet sockets (but can still achieve local communication by specifying localhost address as destination)

= SOCK_STREAM for reliable in-order delivery of a byte stream
= SOCK_DGRAM for delivery of discrete messages

= 0 usually to select default protocol associated with a type

Using Sockets for UNIX IPC (2)

- Each communicating process will first create its own socket, usually SOCK_STREAM.
- Now let's consider only the case of UNIX domain sockets (PF_UNIX domain):
  - Used only for local communication only among a computer's processes
  - Emulates reading/writing from/to a file
  - Each process `bind()`'s its socket to a filename:

```
bind(sd, (struct sockaddr *)&local, length);
```

socket descriptor

data structure containing unique unused file name, e.g. "/users/shiv/myipcsocketfile"

Using Sockets for UNIX IPC (3)

- UNIX domain sockets (cont.):
  - Usually, one process acts as the server, and the other processes connect to it as clients

Server code:

```
sd = socket
   (PF_UNIX,SOCK_STREAM,0)
bind(sd,…)
```

bind and connect must use same file name!

`listen()` for connect requests

`sd2 = accept()` a connect request

`recv(sd2,…)` and

`send(sd2,…)`

Client code:

```
sd = socket
   (PF_UNIX,SOCK_STREAM,0)
```

`connect(sd,…)` to server

`recv(sd,…)` and

`send(sd,…)`

IPC

IPC via Internet domain sockets

- Set up and connect Internet sockets in a way very similar to Unix domain sockets, but…
  - Configure the socket with domain PF_INET instead and set destination to localhost (say 127.0.0.1) instead of the usual remote Internet IP address
  - And need to choose a well-known port # that is shared between processes, i.e. P1 and P2 know this port # in advance – this is similar to the well-known file name
  - Both processes then send() and receive() messages via this port and socket
  - Advantage of writing applications to use this type of IPC via Internet domain sockets is that it is arguably more portable than UNIX-domain sockets

Signals

- Are also a form of inter-process communication, but of limited control information, not data
- Used to inform processes of unexpected external events such as a time out or forced termination of a process
- Allows one process to interrupt another process and send it a coded signal using OS signaling mechanisms
  - A signal is an indication that notifies a process that some event has occurred
- Textbook only briefly explains this concept. We'll cover it in more detail.
  - It is a controlled form of IPC
  - More precisely, some signals are useful in process-to-process communication (IPC)
  - Other signals are primarily for OS-to-process communication

Signals

- Without signals, low-level hardware exceptions are processed by the kernel's exception handlers only, and are not normally visible to user processes
  - e.g. a user process would block on a system call, say read(), to be notified that an I/O event (completion) has occurred
  - Signals expose occurrences of such low-level exceptions to user processes
- 30 types of signals on Linux/UNIX systems
- Windows does not explicitly support signals, but does have *asynchronous procedure calls* (APCs) that emulate signals

Linux/UNIX Signals

Number	Name/Type	Event
2	SIGINT	Interrupt from keyboard (Ctrl-C)
8	SIGFPE	Floating point exception (arith. error)
9	SIGKILL	Kill a process
10, 12	SIGUSR1, SIGUSR2	User-defined signals
11	SIGSEGV	invalid memory ref (seg fault)
14	SIGALRM	Timer signal from alarm function
29	SIGIO	I/O now possible on descriptor

Sending Signals

- Kernel sends a signal to a destination process by updating some state in the context of the destination process, then waking the process up to handle the signal
- A process can send a signal to itself using a library call like alarm() – we'll see this example later. This still goes through the OS.
- A process X can send a signal to process Y by calling `kill(process_id, signal_num)`
  - for example, `kill(Y,SIGUSR1)` sends a SIGUSR1 signal to process Y, which will know how to interpret this signal
  - This call still goes through the OS, not directly from process to process.

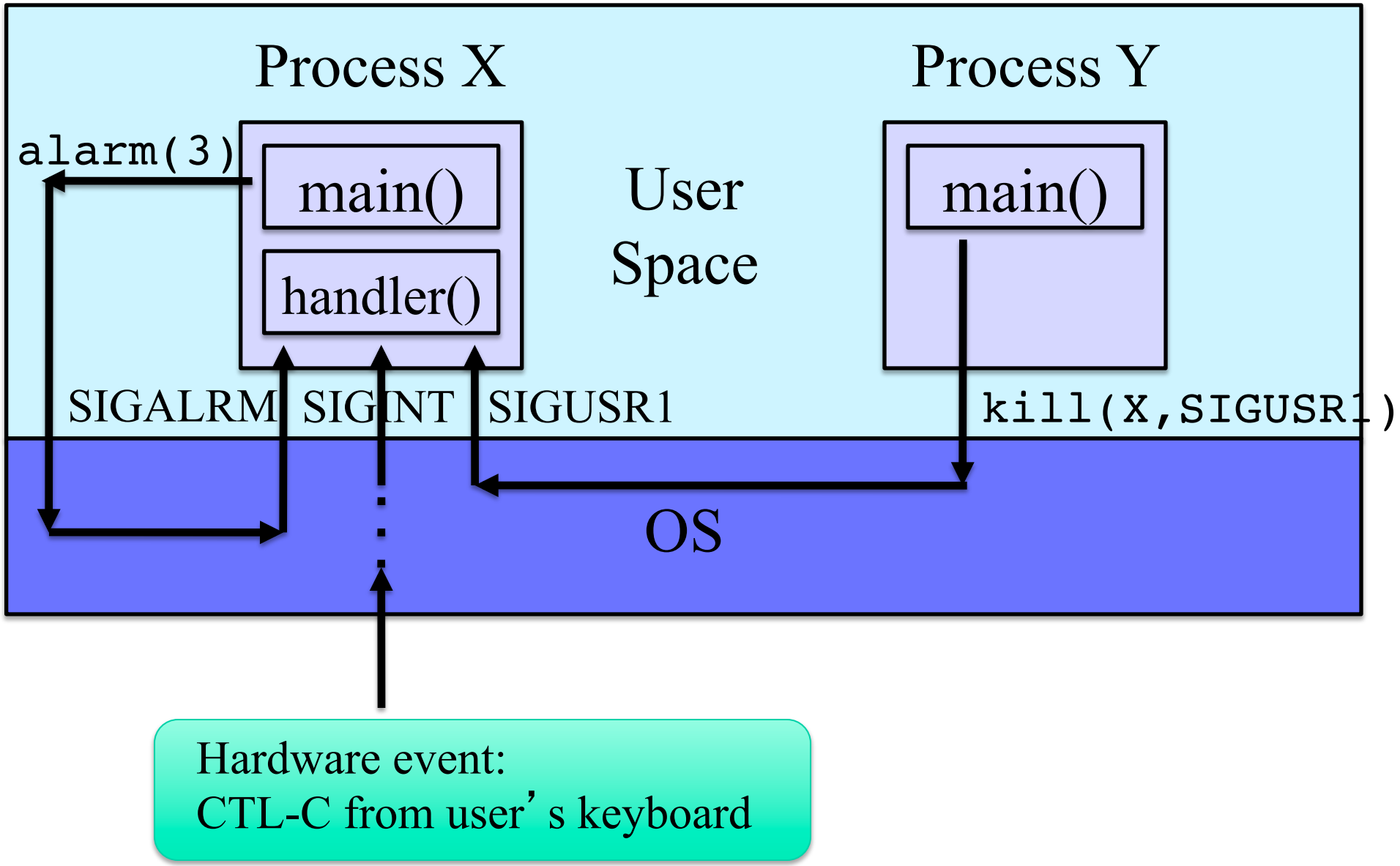
Receiving Signals

- When a kernel is returning from some exception handler, it checks to see if there are any pending signals for a process before passing control to the process
- The user may register a signal `handler()` via the `signal()` function. If no handler is registered, then default action is typically termination
- `signal(signum, handler)` function is used to change the action associated with a signal
  - if handler is SIG_IGN, then signals of type signum are ignored
  - if handler is SIG_DFL, then revert to default action, usually termination
  - otherwise if there is a user-defined function `handler`, then call it to handle the signal.

Catching Signals

- Invocation of the signal handler is called *catching the signal.* We use this term interchangeably with *handling the signal*
- The user-specified signal handler executes in the context of the affected process
  - The kernel calls the user-specified handler, and passes control to user space. When the handler is done (call returns), control returns to the kernel.
  - Note: the next time the process executes, it will resume back at the instruction where the process was originally interrupted/signaled

Signals


Process X
Process Y
alarm(3)
main()
handler()
User
Space
main()
SIGALRM
SIGINT
SIGUSR1
kill(X,SIGUSR1)
OS
Hardware event:
CTL-C from user's keyboard

Signaling Example

- A process can send SIGALRM signals to itself by calling the alarm function
  - alarm(T seconds) arranges for kernel to send a SIGALRM signal to calling process in T seconds
  - see code example next slide
    - #include<signal.h>
    - uses `signal` function to install a signal handler function that is called asynchronously, interrupting the infinite while loop in main, whenever the process receives a SIGALRM signal
    - When handler returns, control passes back to main, which picks up where it was interrupted by the arrival of the signal, namely in its infinite loop

Signaling Example (2)

```
#include <signal.h>
int beeps=0;

void handler(int sig) {
    if (beeps<5) {
        alarm(3);
        beeps++;
    } else {
        printf("DONE\n");
        exit(0);
    }
}

int main() {
    signal(SIGALRM, handler);
    alarm(3);

    while(1) { ; }
    exit(0);
}
```

Signal handler, passed signal #

cause next SIGALRM to be sent to this process in 3 seconds. Assume that SIGALRM is the only signal handled. Otherwise, would need a *case* statement in handler for other signals.

register signal handler

cause first SIGALRM to be sent to this process in 3 seconds

infinite loop that gets interrupted by signal handling

Signals

- More generally, when a process catches a signal of type signum=k, the handler installed for signal k is invoked with a single integer argument set to k
- This argument allows the same handler function to catch different types of signals.

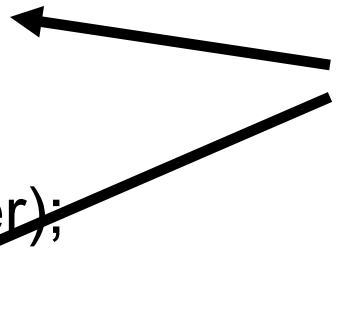
Signals

- Signals are an *asynchronous* signaling mechanism in UNIX
  - A process or thread never knows when a signal will occur. Its execution can be interrupted at any time. A process must be written to handle this asynchrony. Otherwise, could get race conditions.

Example:

```
int global=10;
handler(int signum) {global++;}

main() {
signal(SIGUSR1,handler);
while(1) {global--;}
}
```



- both the main program and the signal handler are manipulating the global variable – could have a *race condition* if main is interrupted in mid-execution of global-- by signal handling of SIGUSR1

Signals

- In addition, if there are multiple signals, can have a race conditions
  - i.e. if a signal handler is processing signal S1 but is interrupted by another signal S2, then could have a race condition inside the handler.
    - In the previous example, we'd have global++ happening in two handlers in rapid succession, could lead to unpredictable results.
  - The solution is to block other signals while handling the current signal.

CSCI 3753
Operating Systems

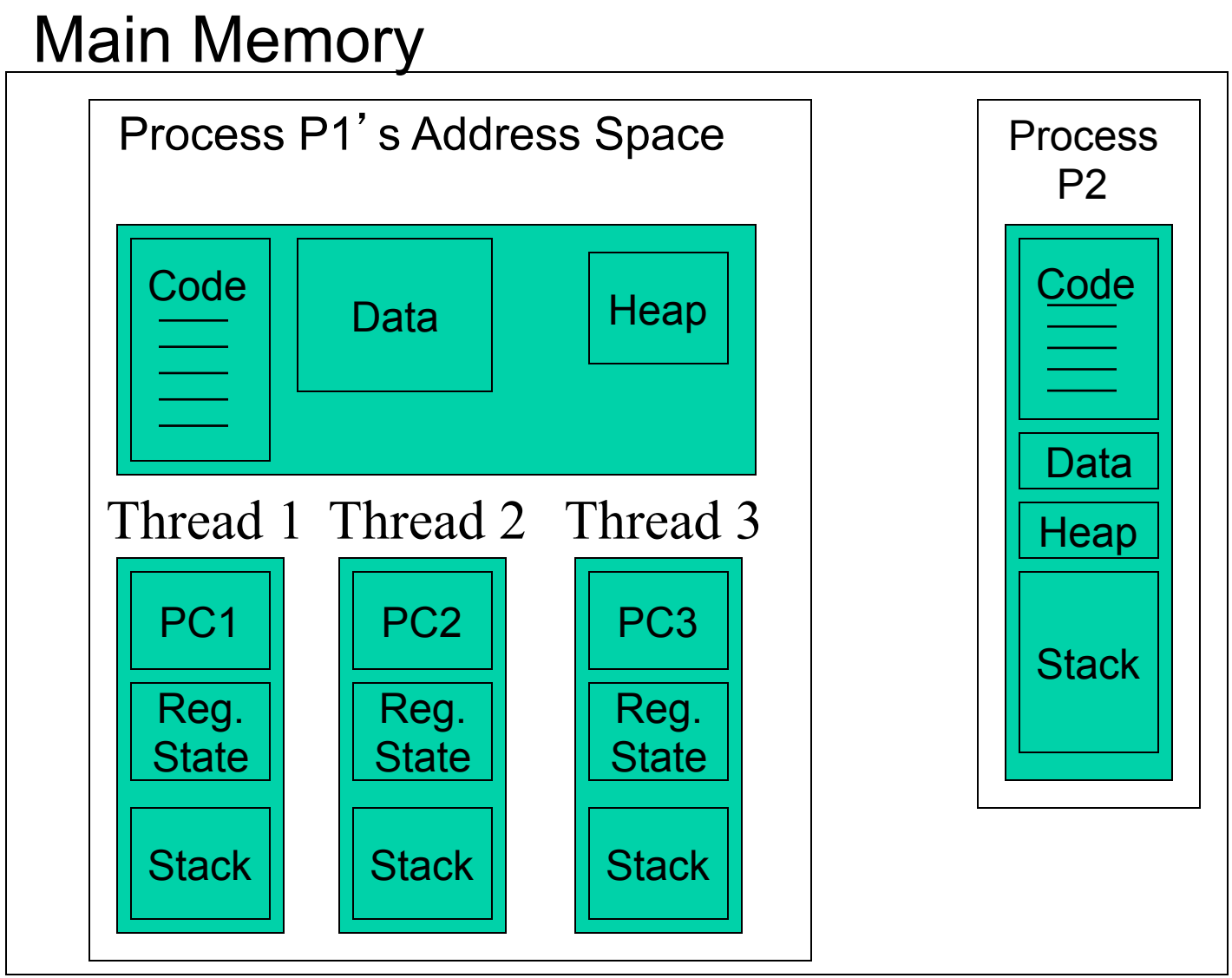
Threads

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 01/31/13

Threads

- A thread is a logical flow or unit of execution that runs within the context of a process
  - has its own program counter (PC), register state, and stack
  - shares the memory address space with other threads in the same process
    - share the same code and data and resources (e.g. open files)

Threads



- Process P1 is *multithreaded*
- Process P2 is single threaded
- The OS is *multiprogrammed*
- If there is preemptive timeslicing, the system is *multitasked*

Applications, Processes, and Threads

- Application = Σ_i $Processes_i$
  - e.g. a server could be split into multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)
- $Process_j$ = Σ $Threads_j$
  - e.g. a Web server process could spawn a thread to handle each new http request for a Web page
- An application could thus consist of many processes and threads

Threads

- Why would you want multithreaded processes?
  - reduced context switch overhead
    - In Solaris, context switching between processes is 5x slower than switching between threads
    - Don't have to save/restore context, including base and limit registers and other MMU registers, also TLB cache doesn't have to be flushed
  - shared resources => less memory consumption => more threads can be supported, especially for a scalable system, e.g. Web server must handle thousands of connections
  - inter-thread communication is easier and faster than inter-process communication – threads share the same memory space, so just read/write from/to the same memory location!
  - thread also called a *lightweight* process

User-Space & Kernel Threads

- User-space threads are created without any kernel support
- User space threads are usually cooperatively multitasked, i.e. user threads within a process voluntarily give up the CPU to each other
- OS is unaware of user-space threads – only sees user-space processes
- User-space thread library
  - provides interface to create, delete threads in the same process
  - If one user space thread blocks, the entire process blocks in a many-to-one scenario (see text)
- User space thread also called a *fiber*

User-Space & Kernel Threads (2)

- *Kernel threads* are supported by the OS
- kernel sees threads and schedules at the granularity of threads
- Most modern OSs like Linux, Mac OS X, Win XP support kernel threads
- Mapping of user-level threads to kernel threads is usually one-to-one, e.g. Linux and Windows, but could be many-to-one, or many-to-many
- Win32 thread library is a kernel-level thread library

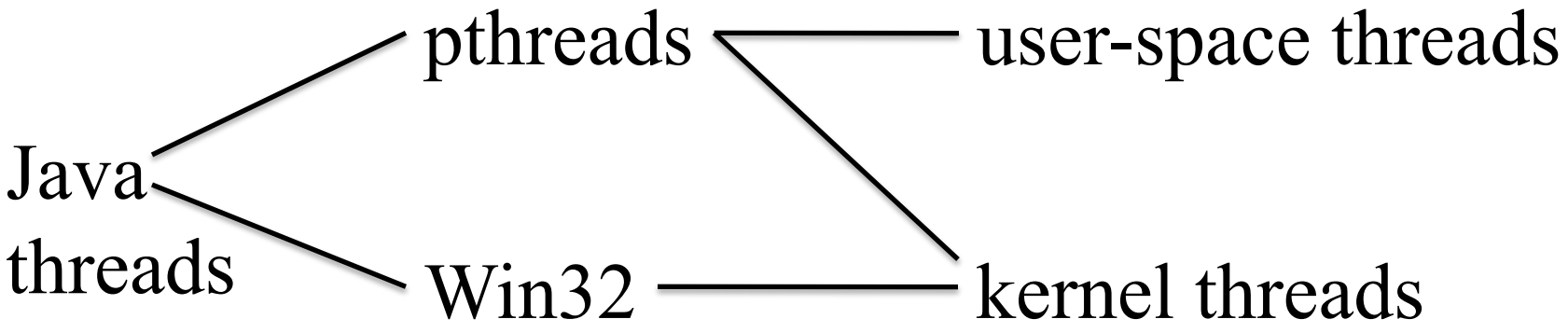
Pthreads

- *pthreads* is a POSIX threading API
- implementations of pthreads API differ underneath the API; could be user space threads; there is also pthreads support for kernel threads as well

- Details in recitation

User-Space & Kernel Threads (3)

- Java thread library is running in Java VM on top of host OS, so on Windows it's implemented on top of Win32 threading, while on Linux/Unix it's implemented on top of pthreads
- Possible scenarios:



CSCI 3753
Operating Systems

Interprocess Synchronization
(Test-and-Set, Semaphores)

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/11/13

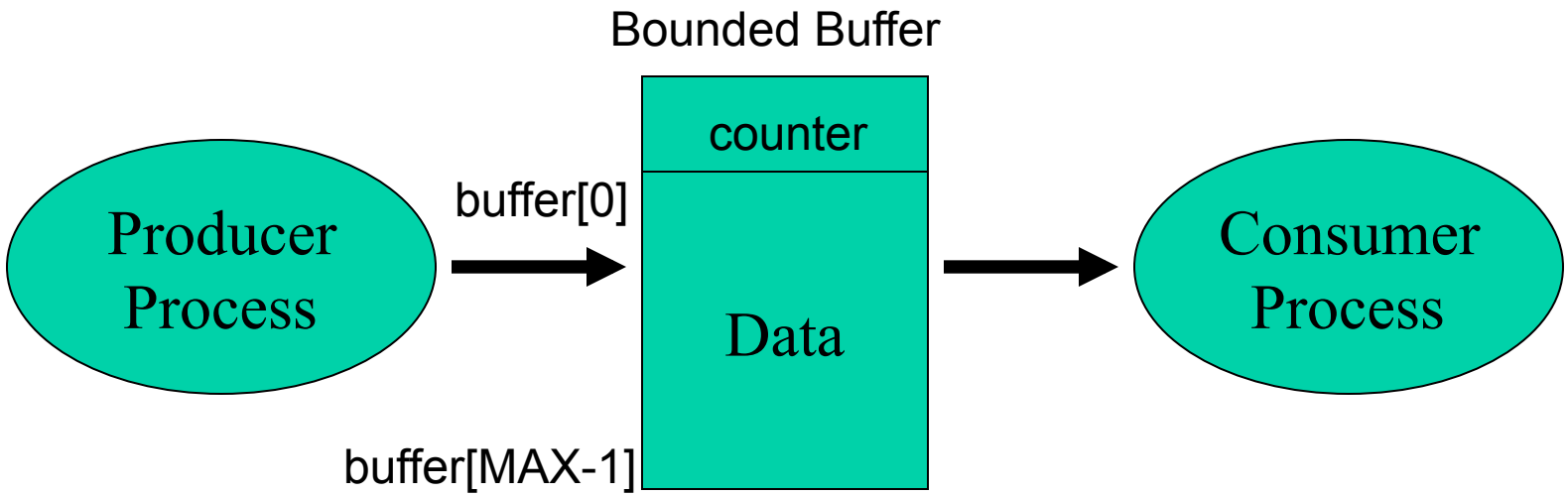
Concurrency

- Multiple processes/threads executing at the same time accessing a shared resource
  - Reading a file
- Value of concurrency – speed & economics
- But few widely-accepted concurrent programming languages (Java is an exception)
- OS tools to support concurrency tend to be "low level"

Producer consumer problem

- Also known as *bounded buffer problem*.
- Two processes (producer and consumer) share a fixed size buffer.
- Producer puts new information in the buffer.
- Consumer takes out information from the buffer.

Synchronization




```
while(1) {
    while(counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
}
```

```
while(1) {
    while(counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
}
```

Producer writes new data into buffer and increments counter

counter updates can conflict!

Consumer reads new data from buffer and decrements counter

Synchronization

- Suppose we have the following sequence of interleaving, where the brackets [value] denote the local value of counter in either the producer or consumer's process. Let counter=5 initially.

// counter++		// counter--;	
(1) reg1 = counter;	[5]	(2) reg2 = counter;	[5]
(3) reg1 = reg1 + 1;	[6]	(4) reg2 = reg2 - 1;	[4]
(5) counter = reg1;	[6]	(6) counter = reg2;	[4]

- At the end, desired value of counter should be 5 with one producer and one consumer, but counter = 4! Plus if steps (5) and (6) were reversed, then counter=6 !!! - undesirable and unpredictable *race condition*
- Basic Problem: unprotected access to a shared variable (counter)

Synchronization

counter++; can compile into several machine language instructions, e.g.

```
reg1 = counter;
reg1 = reg1 + 1;
counter = reg1;
```

counter--; can compile into several machine language instructions, e.g.

```
reg2 = counter;
reg2 = reg2 - 1;
counter = reg2;
```

If these low-level instructions are *interleaved*, e.g. the Producer process is context-switched out, and the Consumer process is context-switched in, and vice versa, then the results of counter's value can be unpredictable

Race Condition

- Situations when two are more processes (or threads) are accessing a shared resource, and the final result depends on which process runs precisely when are called race conditions.
- Race conditions can occur if two are more processes are accessing a shared resource.
- The part of the program where a shared resource is accessed is called *critical section*.
- We need a mechanism to prohibit multiple processes from accessing a shared resource at the same time.

Mutual Exclusion

- No more than one process can execute in a critical section at any time
  - Two or more processes may not execute in a critical section (access to the same shared resource) at the same time.
- How can we implement mutual exclusion?

```
entry section

    critical section (manipulate common var's)

exit section
    remainder section code
```

```
//Producer
    entry section

        counter++;

    exit section

        remainder section code
```

```
//Consumer
    entry section

        counter--;

    exit section

        remainder section code
```

Critical Section

- Critical section access should satisfy multiple properties
  - **mutual exclusion**
    - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - **progress**
    - if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next
    - this selection cannot be postponed indefinitely (OS must make a decision eventually, hence "progress")
  - **bounded waiting**
    - there exists a bound, or limit, on the number of times other processes can enter their critical sections after a process X has made a request to enter its critical section and before that request is granted (no starvation)
- For most of the following slides, we will primarily be concerned with how to achieve mutual exclusion

Disabling interrupts

- Ensure that when a process is executing in its critical section, it cannot be preempted.
- Disable all interrupts before entering a CS.
- Enable all interrupts upon exiting the CS.

```
shared int counter;
```

Code for p_1

```
disableInterrupts();
counter++;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
counter--;
enableInterrupts();
```

- Problems:
  1. If a user forgets to enable interrupts???
  2. Two or more CPUs???

- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; this blocks all processes
- Blocks overlapping I/O

A Flawed Lock Implementation

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(lock){ no_op;}
  lock = TRUE;
/* Execute critical
       section */
  counter++;
/* Release lock */
  lock = FALSE;
```

Acquire(lock)

Code for p_2

```
/* Acquire the lock */
while(lock){ no_op;}
lock = TRUE;
/* Execute critical
       section */
counter--;
/* Release lock */
lock = FALSE;
```

Both processes may enter their critical section if there is a context switch just before the <lock = TRUE> statement

Need a way to test and set the value of a variable atomically

Atomic Test-and-Set

- Need to be able to look at a variable and set it up to some value without being interrupted

  y = read (x); x = value;

- Modern computing systems provide such an instruction called *test-and-set (TS);*

```
boolean TS(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

- The entire instruction (sequence) is atomic, enforced by hardware

Mutual exclusion using TS

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(TS(&lock)) ;

/* Execute critical section */
  counter++;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
   while(TS(&lock)) ;

/* Execute critical section */
        counter--;
/* Release lock */
   lock = FALSE;
```

- The boolean TestandSet() instruction is essentially a swap of values
  - The x86 CPU instruction set contains atomic instructions such as XCHG that are essentially swap statements
  - Can use atomic XCHG to implement spinlocks
- Mutual exclusion is achieved - no race conditions
  - If one process X tries to obtain the lock while another process Y already has it, X will wait in the loop
  - If a process is testing and/or setting the lock, no other process can interrupt it
- The system is exclusively occupied for only a short time - the time to test and set the lock, and not for entire critical section
  - typically only about 10 instructions
- Don't have to disable and reenable interrupts - time-consuming
- Do you see any problems? → busy waiting

wait( ) and **signal( )** primitives

- Also called *sleep()* and *wakeupl()* primitives
- *wait()*: causes a process to block.
- *wakeup(pid)*: causes the process whose id is *pid* to move to ready state.
  - No effect if process *pid* is not blocked.

```
while(1) {
    if (counter==MAX) wait();
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
    if (counter == 1) signal (p2);
}
```

```
while(1) {
    if (counter==0) wait();
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
    if (counter == MAX – 1) signal (p1);
}
```

- Consumer reads counter and counter = 0.
- Scheduler schedules the producer.
- Producer puts an item in the buffer and signals the consumer
  - Since consumer has not yet invoked wait(), the signal() invocation by the producer has no effect.
- Consumer is scheduled, and it blocks.
- Eventually, producer fills up the buffer and blocks.
- How can we solve this problem?
  - Need a mechanism to count the number of wait() and signal() invocations.

Semaphores

- More general solution to mutual exclusion proposed by Dijkstra
- Semaphore S is an abstract data type that, apart from initialization, is accessed only through 2 standard atomic operations
  - P(), also called wait(), short for Dutch word *proberen* "to test"
    - somewhat equivalent to a test-and-set, but also involves *decrementing* the value of S
  - V(), also called signal(), short for Dutch word *verhogen* "to increment"
    - *increments* the value of S
  - OS provides ways to create and manipulate semaphores atomically

Semaphores

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
P(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
V(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Both P and V operations are atomic

Mutual Exclusion with Semaphores

```
semaphore S = 1;  // initial value of semaphore is 1
int counter;      // assume counter is set correctly somewhere in
                     code
```

Process P1:

```
P(S);
   // execute critical section
   counter++;
V(S);
```

Process P2:

```
P(S);
   // execute critical section
   counter--;
V(S);
```

- Both processes atomically P() and V() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable counter

Problems with semaphores

- Potential for deadlock

```
Semaphore Q= 1;    // binary semaphore as a mutex lock
Semaphore S = 1;   // binary semaphore as a mutex lock
variable R1, R2;
```

Process P1:

```
P(S);          (1)
P(Q);          (3)

modify R1 and R2;

V(S);
V(Q);
```

Process P2:

```
P(Q);          (2)
P(S);          (4)

modify R1 and R2;

V(Q);
V(S);
```

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The V() statements will never get executed, so there is no way to wake up the two processes
  - There is no rule wrt the order in which P( ) and V( ) operations may be invoked
  - In general, with N processes sharing N semaphores, the potential for deadlock grows

Other problematic scenarios

- A programmer mistakenly follows a P() with a second P() instead of a V()
- A programmer forgets and omits the P(mutex) or V(mutex)
- A programmer reverses the order of P() and V()

CSCI 3753
Operating Systems

Classic Synchronization Problems (Producer-Consumer, Readers-Writers, Dining Philosophers)

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 02/12/13

Classic Synchronization Problems

- Bounded Buffer Producer-Consumer Problem
- Readers-Writers Problem
  - First Readers Problem
- Dining Philosophers Problem
- These are not just abstract problems
  - They are representative of several classes of synchronization problems commonly encountered in the real world when trying to synchronize access to shared resources among multiple processes or threads

Producer consumer problem

- We have already seen this problem with one producer and one consumer
- General problem: multiple producers and multiple consumers
- Producers puts new information in the buffer.
- Consumers takes out information from the buffer.

Semaphore empty = 0, full = MAX, m = 1;

```
producer()
    produce_info(item);
    P(full);
      P(m);
        enter_info(item);
      V(m);
    V(empty);
```

```
consumer ()
    P(empty);
       P(m);
          remove_info(item);
       V(m);
    V(full);
    consume_info(item);
```

Semaphores empty and full are used for maintaining counter values and signaling between producer and consumer processes.

Semaphore mutex1 is used for mutual exclusion among producer processes and mutex2 is used for mutual exclusion among consumer processes.

Pthreads Synchronization

- Mutex locks
  - Used to protect critical sections
- Some implementations provide semaphores through POSIX SEM extension
  - Not part of Pthreads standard

```
#include <pthread.h>
pthread_mutex_t m; //declare a mutex object
Pthread_mutex_init (&m, NULL); // initialize mutex object
```

```
//thread 1
pthread_mutex_lock (&m);
   //critical section code for th1
pthread_mutex_unlock (&m);
```

```
//thread 2
pthread_mutex_lock (&m);
   //critical section code for th2
pthread_mutex_unlock (&m);
```

From pthreads handout

odd function

```
…
  pthread_mutex_lock(&m);
  for (i = 0; i < 10000; i++)
    printf("odd\n");
  pthread_mutex_unlock(&m);
…
```

even function

```
…
  pthread_mutex_lock(&m);
  for (i = 0; i < 10000; i++)
    printf("even\n");
  pthread_mutex_unlock(&m);
…
```

main function

```
…
  pthread_mutex_lock(&m);
  for (i = 0; i < 10000; i++)
    printf("main\n");
  pthread_mutex_unlock(&m);
…
```

All three functions are writing to the standard output

What can happen if we do not use mutexes?

- *try it*

Binary Semaphores

Similar to semaphores with one key difference

- value can be only 0 or 1

```
typedef struct {
    int value;
    struct process *list;
} bin_semaphore;
```

```
P(bin_semaphore *s) {
    if (s→value == 0) {
        add this process to s→list;
        sleep ( );
    }
    else s→value = 0;
}
```

```
V(bin_semaphore *s) {
    if (s→list is not empty) {
        remove a process P from s→list;
        wakeup (P);
    }
    else s→value = 1;
}
```

Both P and V operations are atomic

Pthread mutex and binary semaphores

- Like binary semaphores, pthread mutexes can have only one of two states: lock or unlock
- But, there is a key difference
  - Mutex ownership: Only the thread that locks a mutex can unlock that mutex, while any thread can call the V operation on a binary semaphore irrespective of which thread called the P operation on that binary semaphore
  - So, mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads

POSIX semaphores

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
//pshared: 0 (among threads); 1 (among processes)

int sem_wait(sem_t **sem*); //P( ) operation

int sem_post(sem_t *sem); //V( ) operation

sem_getvalue( ), sem_close( ),

Kernel Synchronization

- At any time, many kernel mode processes may be active
  - Share kernel data structures
  - Notice that even though user processes have their own address spaces, race conditions can still arise when they execute in kernel mode, e.g. executing a system call
- Preemptive and non-preemptive kernels
  - Preemptive kernel: allows a process to be preempted while running in kernel mode
    - Race conditions can occur
  - Non-preemptive kernel: does not allow a process to be preempted while running in kernel mode
    - Race conditions cannot occur

Windows Synchronization

- Kernel level
  - Singe processor system: temporarily mask interrupts for all interrupt handlers that may also access a shared resource
  - Multiprocessor system: use spin lock (busy waiting)
- User level
  - Dispatcher objects: mutex locks, semaphores, …

Linux Synchronization

- Kernel level
  - Prior to version 2.6, non-preemptive kernel, but later versions are fully preemptive
  - Atomic integers: all math operations on atomic integers are performed without interruptions

```
atomic_t counter;
atomic_set(&counter, 5);
atomic_add(10, &counter);
…
```

  - Mutex locks, spin locks and semaphores, enabling/ disabling interrupts on single processor systems
- User level
  - Futex, semop( ): system call

Readers writers problem

- A database is accessed by two types of processes: reader processes and writer processes.
- Readers only read information from the database.
- Writers modify the database.

- Constraints
  - Writers must have exclusive access to the database.
  - Multiple readers can access the database concurrently.

Readers-Writers Problem: First Attempt

Semaphore mutex = 1;

```
Reader( )
{
   P(mutex);
      read database
   V(mutex);
}
```

```
Writer( )
{
   P(mutex);
      write database
   V(mutex);
}
```

Exclusive access to the writer processes is provided
BUT: No concurrency among reader processes

Readers-Writers Problem: Second Attempt

```
int rc = 0; /* Number of readers in the database */
Semaphore db = 1; /* controls access to database for writers */
Semaphore mutex = 1; /* controls access to variable rc */

Reader ()
    P(mutex);
    if (rc == 0) P(db);
    rc++;
    V(mutex);

        read database

    P(mutex);
    rc--;
    if (rc == 0) V(db);
    V(mutex);
```

```
Writer( )
{
    P(db);
        write database
    V(db);
}
```

Readers-Writers Problem: Second Attempt

- Semaphore mutex is used for mutual exclusion to update rc
- Semaphore db is used for exclusive database access for writer processes
- Multiple reader processes can access the database concurrently if there is no writer process.

Problem: What happens to a writer if readers keep coming to read the database?

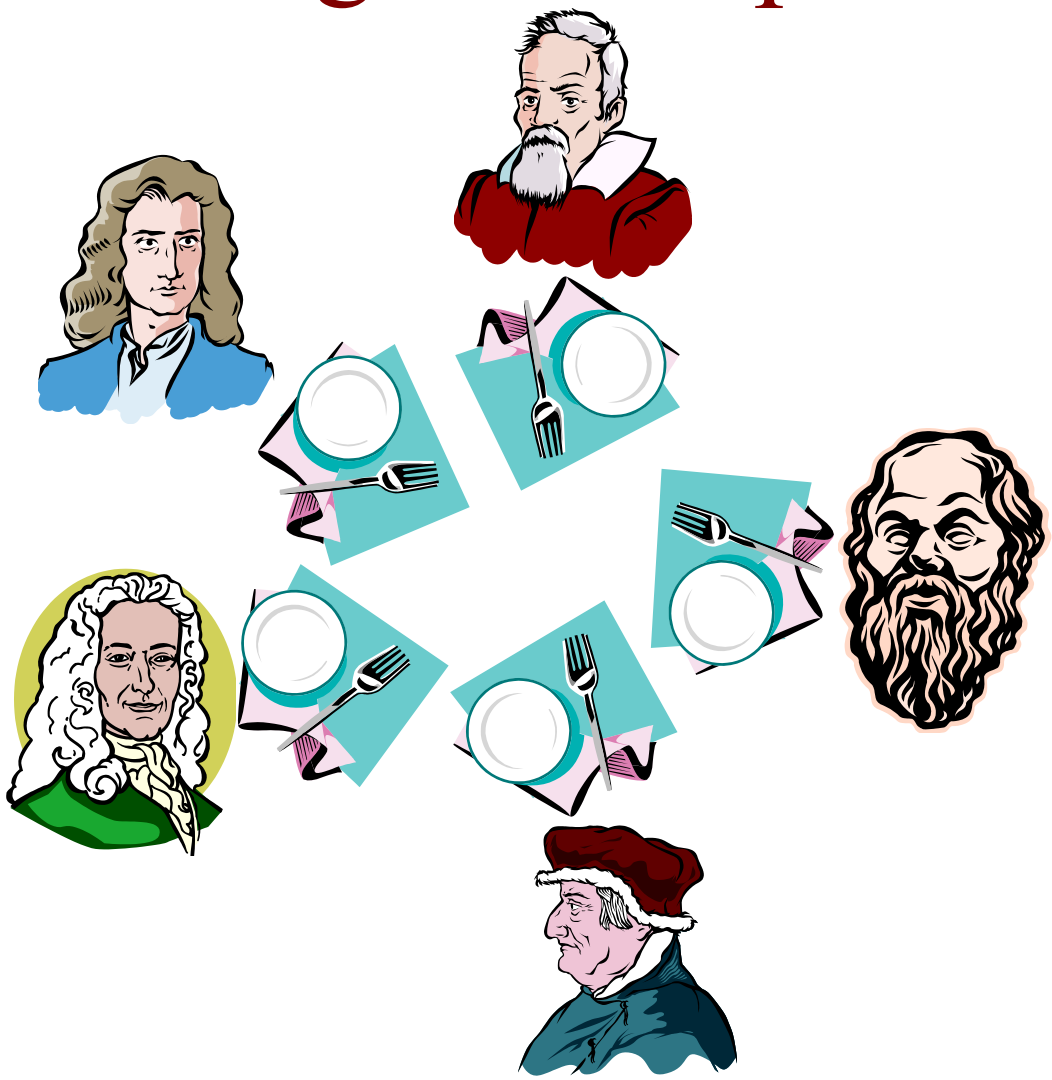
- Writer starvation
- Readers have priority over writers.

- Write a solution that gives preference to writer processes
- Write a solution that is fair to both readers and writers

Dining Philosophers Problem

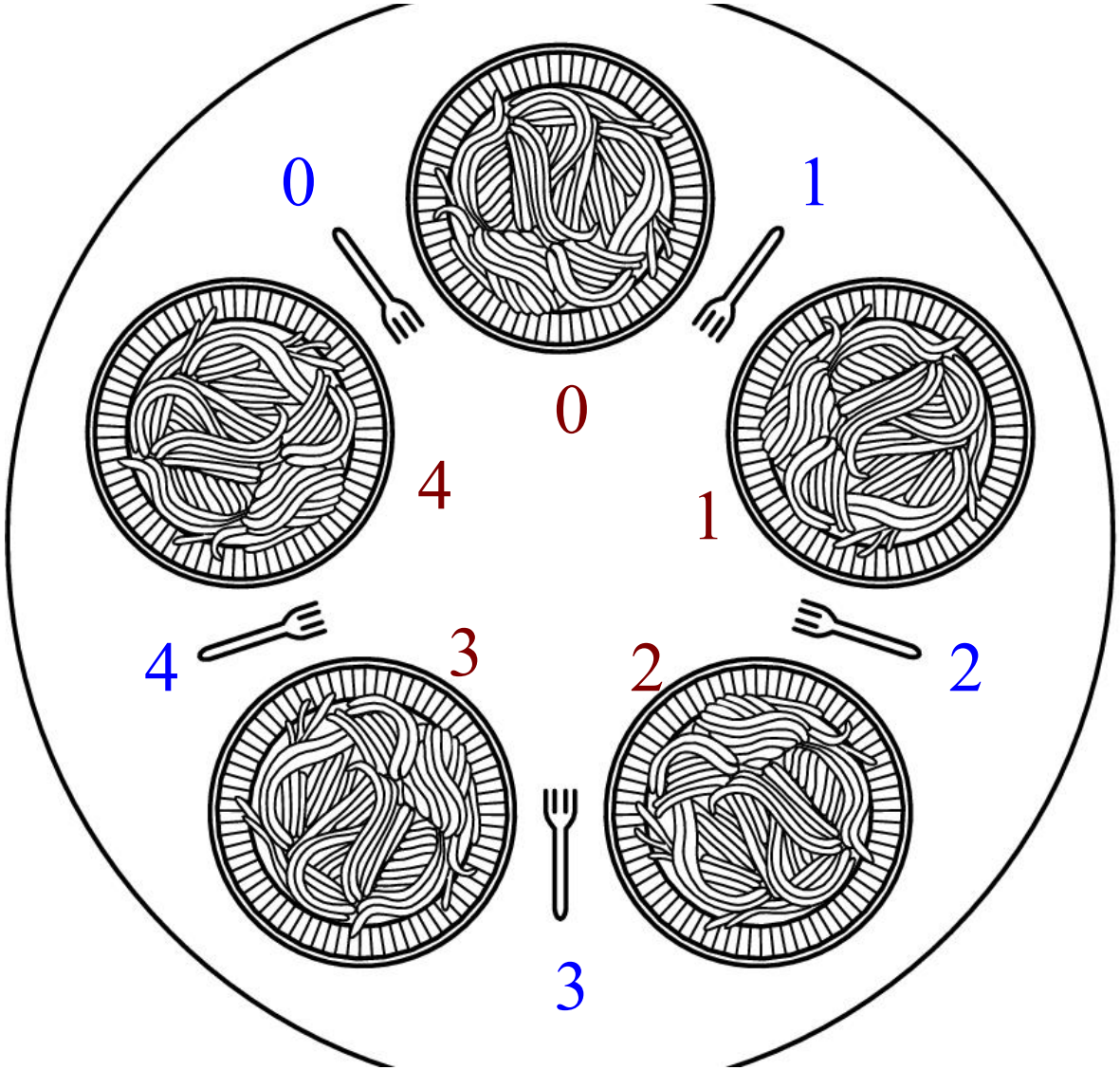
- The most famous synchronization problem.
- Represents a situation that can occur in large community of processes that share a large pool of resources.
- Five philosophers sit around a round dining table. A plate of spaghetti is placed in front of each philosopher, and a fork is placed between any two adjacent plates.
- A philosopher needs two forks to eat spaghetti.
- All philosophers alternate between two activities: thinking and eating.

Dining Philosophers Problem



Write a synchronization program that allows all five philosophers to run their lives

- Deadlock free
- Starvation free


0
1
0
4
1
4
3
2
2
3

Dining Philosophers: First Attempt

```
philosopher(int i) {
  while(TRUE) {
    // Think
    P(fork[i]);
      P(fork[(i+1) mod 5]);
        // EAT
      V(fork[(i+1) mod 5]);
    V(fork[i]);
  }
}
semaphore fork[5] = (1,1,1,1,1);
```

Problem

- On a fateful day, all philosophers decide to eat at the same time.
- All philosophers pick up their right fork.
- All philosophers now wait forever for their left fork to become available, and die of starvation
  - Deadlock.

Dining Philosophers: Second Attempt

```
philosopher(int i) {
  while(TRUE) {
    // Think
    P(mutex);
      P(fork[i]);
      P(fork[(i+1) mod 5]);
    V(mutex);
    // Eat
    V(fork[(i+1) mod 5]);
    V(fork[i]);
  }
}
semaphore fork[5] = (1,1,1,1,1);
semaphore mutex = 1;
```

- This solution doesn't suffer from deadlock
- Problem
  - May not allow two non-adjacent philosophers to eat at the same time

Dining Philosophers: Third Attempt

After picking up a fork, if a philosopher finds that the other fork is not available, she keeps down the fork, waits for some time, and tries again.

- Does this solution work?
- Starvation
- A deadlock-free solution is not necessarily starvation-free

Dining Philosophers: Some possible solutions

- Allow at most 4 philosophers at the same table when there are 5 resources
- Odd philosophers pick first left then right, while even philosophers pick first right then left
- Allow a philosopher to pick up forks *only if both are free.* This requires protection of critical sections to test if both forks are free before grabbing them.
  - we'll see this solution next using monitors
  - Also, there is a construct called an AND semaphore

Higher Level Synchronization Primitives

- Semaphores can result in deadlock due to programming errors
  - forgot to add a P() or V(), or mis-ordered them, or duplicated them
- Relatively simple problems, such as the dining philosophers problem, can be very difficult to solve using low level constructs like semaphores
- Higher level synchronization primitives
  - AND synchronization
  - Events
  - Critical Conditional Regions
  - Monitors: *We will study this*
  - many others…

Monitors

- Abstract data type (similar to C++ classes)
  - Monitors are found in high-level programming languages like Java and C#
- A monitor is a collection of procedures, variables, and data structures
- Processes can access these variables only by calling procedures in the monitor
- Each function in the monitor can only access variables declared locally within the monitor and its parameters
- At most one process may be active at any time in a monitor

- monitor *monitor_name* {
  // shared local variables

  function f1(...) {
  ...
  }
  ...
  function fN(...) {
  ...
  }
  init_code(...) {
  ...
  }
}

Monitors and Condition Variables

- While the above definition of a monitor achieves mutual exclusion (hiding P() and V() from user), it loses the ability that semaphores had to enforce order
  - i.e. P() and V() are used to provide mutual exclusion, but the unique ability for one process to signal another blocked process using V() is lost
- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, augment monitors with *condition variables*.

Monitors and Condition Variables

- Declare a condition variable with pseudo-code

  condition x, y;
- A condition variable *x* in a monitor allows three main operations on itself
  - x.wait()
    - blocks the calling process
    - can have multiple processes suspended on a condition variable, typically released in FIFO order, but textbook describes another variation specifying a priority p, i.e. call x.wait(p)
  - x.signal()
    - resumes exactly 1 suspended process. If none, then *no effect*.
  - x.queue()
    - Returns true if there is at least one process blocked on x

- Note that x.signal() is unlike the semaphore's signaling operation V(), which preserves state in terms of the value of the semaphore.
    - Example: if a process Y calls x.signal() on a condition variable x before process Z calls x.wait(), then Z will wait. The condition variable doesn't remember Y's signal.
    - Comparison: if a process Y calls V(mutex) on a binary semaphore mutex (initialized to 0) before process Z calls P(mutex), then Z will not wait, because the semaphore remembers Y's V() because its value = 1, not 0.

Monitors and Condition Variables

- Within a monitor, if a process P1 calls x.signal(), then normally that would wake another process P2 blocked on x.wait(). But we must avoid having two processes at the same time in the monitor, so need "wake-up" semantics on a x.signal():
  - Hoare semantics, also called signal-and-wait
    - The signaling process P1 either waits for the woken up process P2 to leave the monitor before resuming, or waits on another CV
  - Mesa semantics, also called signal-and-continue
    - The signaled process P2 waits until the signaling process P1 leaves the monitor or waits on another condition

Dining Philosophers: Monitor-based Solution

- Key insight: pick up 2 forks only if both are free
  - Avoids deadlock
  - A philosopher moves to his/her eating state only if both neighbors are not in their eating states
    - Need to define a state for each philosopher
  - If one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - States of each philosopher: thinking, hungry, eating
    - Need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() forks

Dining Philosophers: Monitor-based Solution

```
monitor DiningPhilosophers
{
   enum {Thinking, Hungry, Eating} state[5];
   condition self[5];

   void test(int i) {
      //Called by philosopher i or neighbors of i
      //Check if both neighbors of i are not eating
      //If so, set state[i] to Eating, and signal philosopher i
   }
   void pickup(int i) {
      //Called by philosopher i
      //Set state[i] to Hungry and call test(i)
      //If at least one neighbor is eating, block on self[i];
   }
```

... cond. to the next slide

... cond. from the previous slide

```
    void putdown(int i) {
        //Called by philosopher i after eating
        //change state[i] to Thinking and signal neighbors in
        //case they are waiting to eat
    }

    init( ) {
        for (int i = 0; i < 5; i++)
            state[i] = Thinking;
    }
}
```

```
philosopher (int i)
{
    DiningPhilosophers.pickup(i);
        // pick up forks and eat
    DiningPhilosophers.putdown(i);
}
```

```
void test(int i) {
    if ((state[(i+1)%5] != Eating) &&
       (state[(i-1)%5] != Eating) &&
       (state[i] == Hungry))
    {
       state[i] = eating;
       self[i].signal();
    }
  }
```

```
void pickup(int i) {
   state[i] = hungry;
   test(i);
   if(state[i]!=Eating)
      self[i].wait;
}
```

```
void putdown(int i) {
      state[i] = thinking;
      test((i+1)%5);
      test((i-1)%5);
   }
```

Starvation

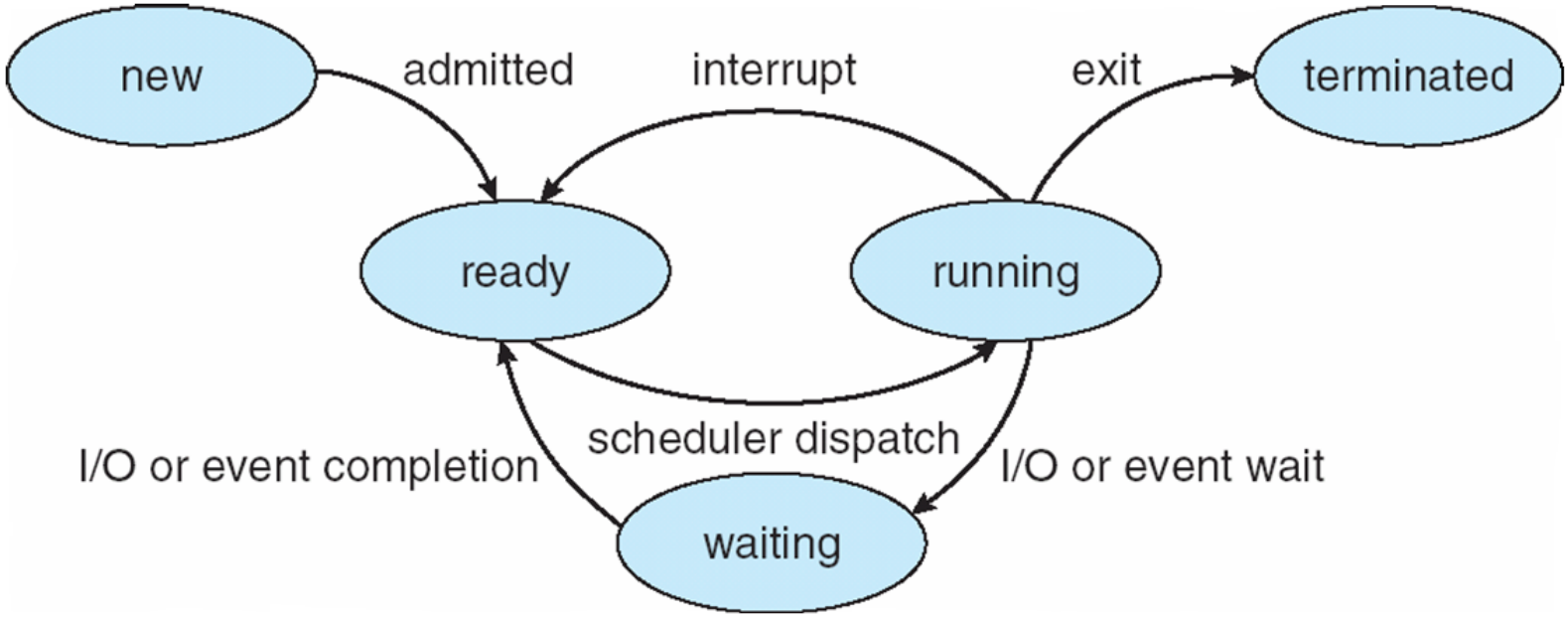
- Note that starvation is still possible in the DP monitor solution
  - Suppose P1 arrives first, and start eating, then P2 arrives and sets its state to hungry and blocks
  - Next P3 arrives and starts eating
  - P1 ends, but P2 can't start eating because P3 is eating
  - Now P1 starts eating again before P3 finishes eating
  - P3 ends, but P2 still can't eat
  - P1 and P3 can alternate this way and P2 will never get to eat →starvation

CSCI 3753
Operating Systems

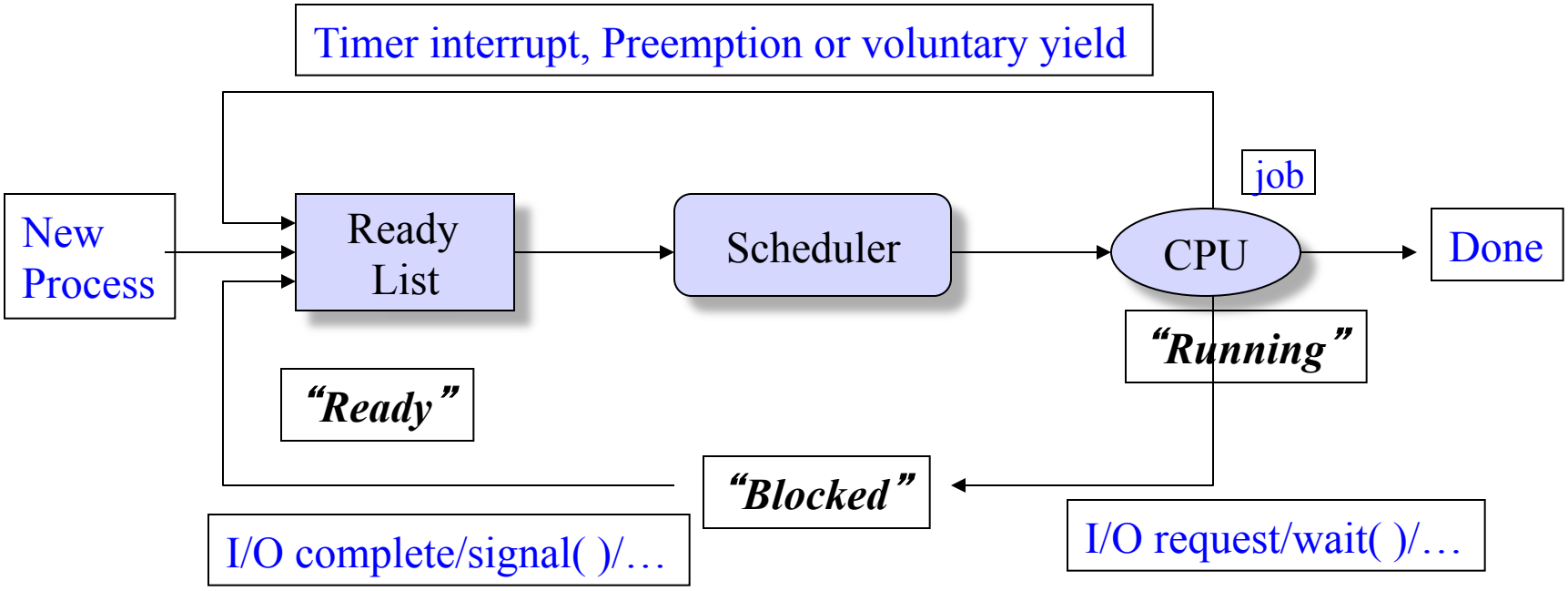
CPU Scheduling

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/15/13

Diagram of Process State



Also called “blocked” state



Process scheduling problem: When more than one processes are in Ready list, how does OS decide which process to run next?

- Carried out by short term scheduler or CPU scheduler

Scheduling Definitions

- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
- *wait time* $W(P_i)$ = sum of the times process i spends in the ready state
- *turnaround time* $T(P_i)$ = the time from 1st entry of process i into the system to its final exit from the system (exits last run state)
- *response time* $R(P_i)$ = the time from 1st entry of process i into the ready queue to its 1st scheduling on the CPU (1st run state)
  - Some processes can generate early results, so if they get some CPU time quickly, they can start producing output sooner. A quick response time from the scheduler benefits such processes.

- *CPU utilization*: Percentage of time the CPU is busy
- *Throughput*: # of processes completed per time unit

Scheduling Goals

- Maximize CPU utilization: 40% to 90%
- Maximize throughput
- Minimize average or peak turnaround time
- Minimize average or peak waiting time
- Minimize average or peak response time
- Maximize fairness
- Meet deadlines or delay guarantees
- Ensure priorities are adhered to

Some of these goals are contradictory.
Any scheduling algorithm that favors
one class of jobs hurts another class of jobs.

Preemptive vs Non-preemptive Scheduling

- Non-preemptive scheduling
  - A running process keeps the CPU until terminating or switching to waiting state
  - A long-running CPU-bound process can prevent other processes from getting the CPU
- Preemptive scheduling
  - A running process may be forced to give up CPU to move to the Ready state
  - Relies on timer interrupts
  - Can result in race conditions among processes

Scheduling Analysis

- We analyze various scheduling policies to see how efficiently they perform with respect to metrics like:
  - Wait time, turnaround time, response time, etc.
- Some algorithms will be optimal in certain metrics
- To simplify analysis assume:
  - No blocking I/O. Focus only on scheduling processes/tasks that have provided their execution times
  - Processes execute until completion, unless otherwise noted, e.g round robin.

FCFS Scheduling

- First Come First Serve: order of arrival dictates order of scheduling
  - Non-preemptive, processes execute until completion
- If processes arrived in order P1, P2, P3 before time 0, then *Gantt chart* of CPU service time is:

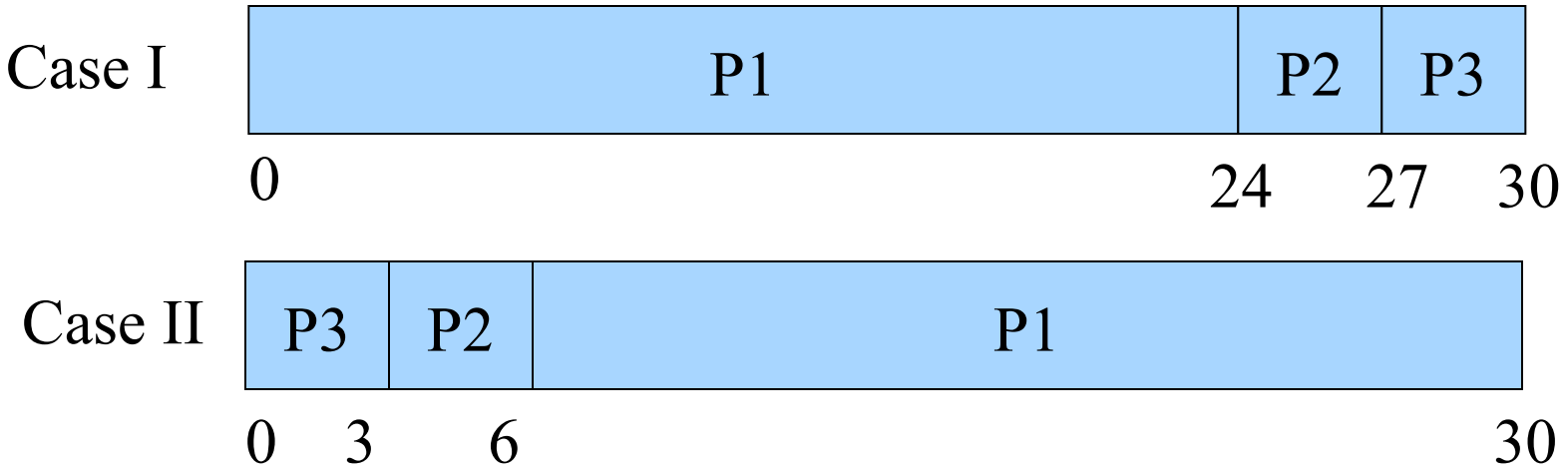
Process	CPU Service Time
P1	24
P2	3
P3	3

FCFS Scheduling (2)

- If processes arrive in reverse order P3, P2, P1 around time 0, then Gantt chart of CPU service time is:

Process	CPU Service Time
P1	24
P2	3
P3	3

FCFS Scheduling (3)



- Case I: average wait time is (0+24+27)/3 = 17 seconds
- Case II: average wait time is (0+3+6)/3 = 3 seconds
- FCFS wait times are generally not minimal - vary a lot if order of arrival changed, which is especially true if the process service times vary a lot (are spread out)
- Case I: average turnaround time is (24+27+30)/3 = 27 seconds
- Case II: average turnaround time is (3+6+30)/3 = 13 seconds
- A lot of variation in turnaround time too.

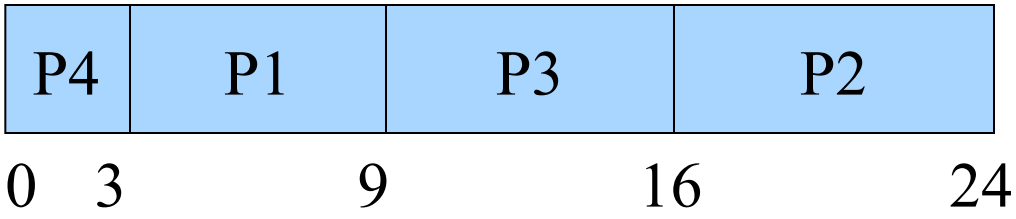
Shortest Job First (SJF) Scheduling

- Choose the process/thread with the lowest execution time
  - gives priority to shortest or briefest processes
  - minimizes the average wait time
    - Intuition: moving a long process before a short one increases the wait time of short processes a lot.
    - Conversely, moving long process to the end decreases wait time seen by short processes
- SJF minimizes the average wait time out of all possible scheduling policies.
- Problem: must know run times in advance unlike FCFS
  - Predict using exponential averages … (see textbook)

Shortest Job First Scheduling

- In this example, P1 through P4 are in ready queue at time 0:
  - *can prove SJF minimizes wait time* - out of 24 possibilities of ordering P1 through P4, the SJF ordering has the lowest average wait time

Process	CPU Execution Time
P1	6
P2	8
P3	7
P4	3



average wait time
= (0+3+9+16)/4
= 7 seconds

Shortest Job First Scheduling

- Can be preemptive
  - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job's remaining execution time, then it can preempt the current job
    - Shortest remaining time first
  - For simplicity, we assumed in the preceding analysis that jobs ran to completion and no new jobs arrived until the current set had finished.
  - Compare to FCFS: a new process can't preempt earlier processes, because its order is later than the earlier processes

Deadline Scheduling

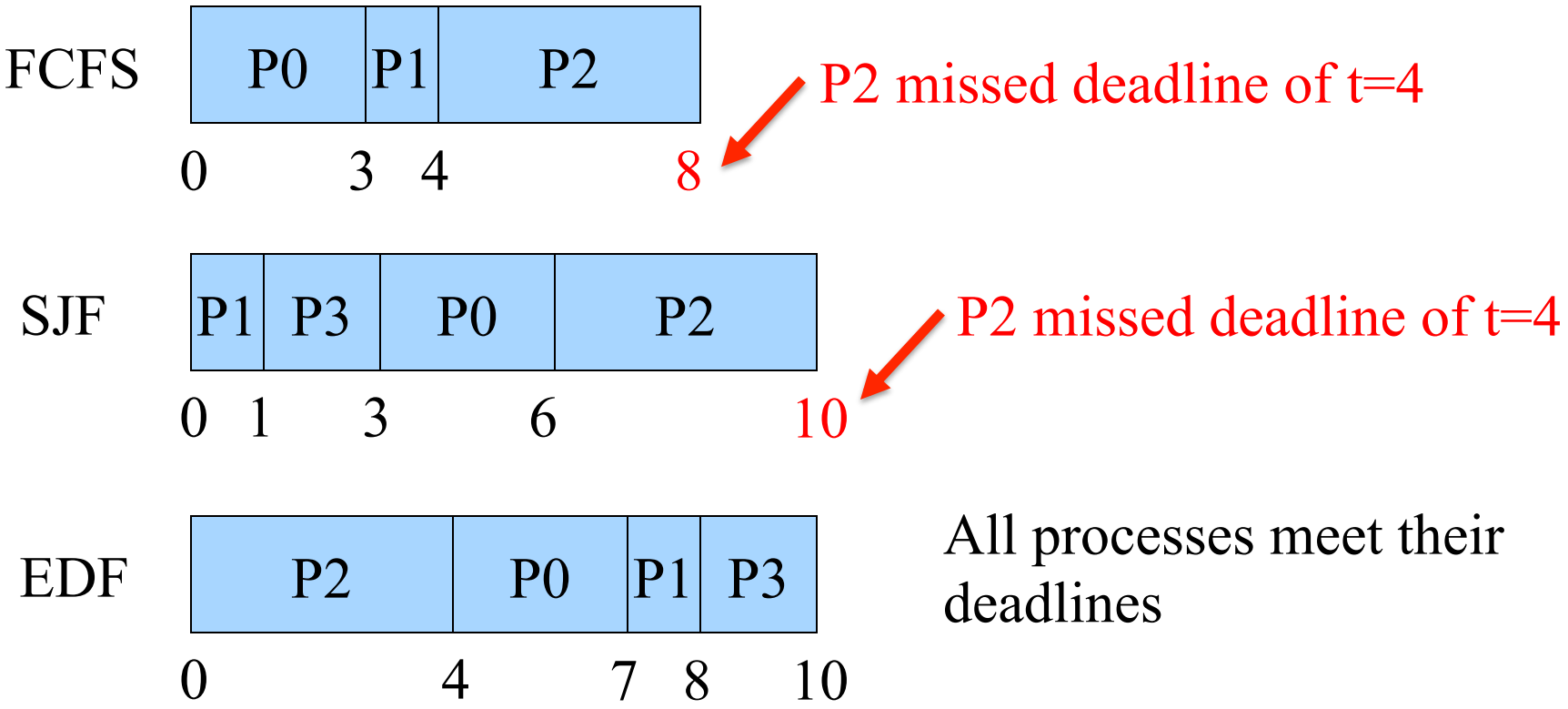
Hard real time systems require that certain processes *must* complete execution within a certain time, or the system crashes

- e.g. robots need a real time OS (RTOS) whose processes (actuating an arm/ leg) must be scheduled by a certain deadline

Process	CPU Execution Time	Deadline from now
P0	3	7
P1	1	9
P2	4	4
P3	2	10

Deadline Scheduling

- *Earliest deadline first* (EDF) selects the process with the nearest/soonest deadline
  - This is the process that most urgently needs to be completed



Deadline Scheduling

- Even EDF may not be able to meet all deadlines:
  - In previous example, if P3's deadline was t=9, then EDF cannot meet P3's deadline
- Admission control policy
  - Check on entry to system whether a process's deadline can be met, given the current set of processes already in the ready queue and their deadlines
    - If all deadlines can be met with the new process, then admit it
    - Else, deny admission to this process if its deadline can't be met. Note FCFS or SJF had no notion of refusing admission

Deadline Scheduling

- Admission control used when scheduling policies try to provide different Qualities of Service (QOS)
  - It's common in network-based QOS scheduling policies for routers – can't admit a new source of packets if its QOS deadlines or guarantees cannot be met at a router
- EDF can be preemptive
  - A process that arrives with an earlier deadline can preempt one currently executing with a later deadline.

Round Robin Scheduling

- The CPU scheduler rotates among the processes in the ready queue, giving each a time slice
  - e.g. if there are 3 processes P1, P2, & P3, then the scheduler will keep rotating among the three: P1, P2, P3, P1, P2, P3, P1, …
  - treats the ready queue as a circular queue
  - useful for time sharing multitasking systems and therefore is a popular scheduling algorithm

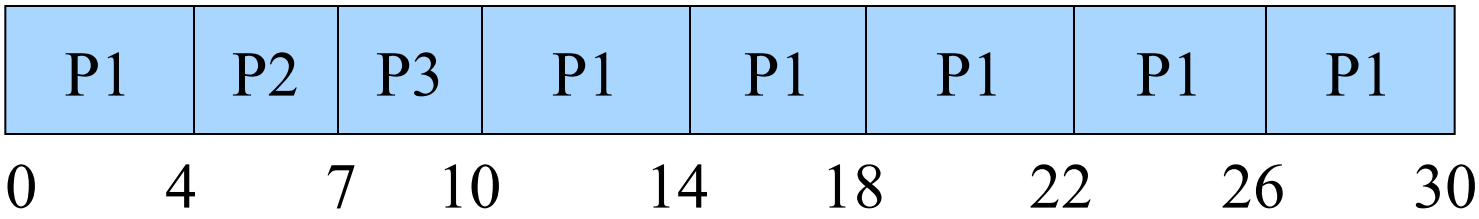
Round Robin Scheduling

- Simple and fair, though wait times can be long
  - Fair: If there are n processes, each process gets 1/n of CPU
  - Simple: Don't need to know service times a priori
- RR is an example of preemptive scheduling – a process may be forced to relinquish CPU before it's done
  - This was not the case for FCFS, and only occurred when new processes arrived for SJF and EDF. Now, we allow timer-based interrupts.
- A process can finish before its time slice is up. The scheduler just selects the next process in the queue

Round Robin Scheduling

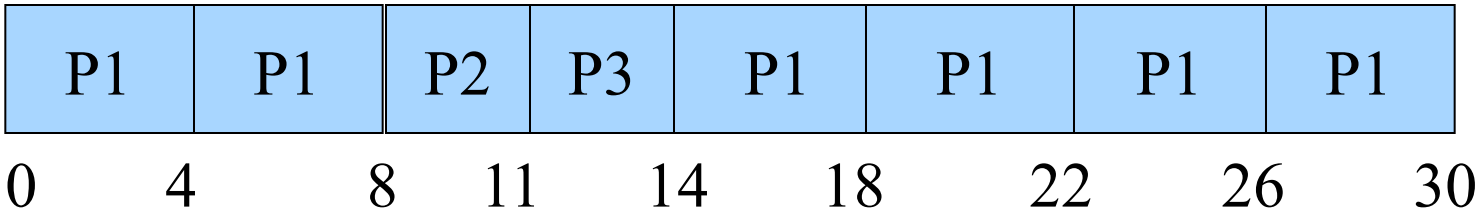
- Example: suppose we use a time slice of 4 ms, and ignoring context switch overhead
- Now P1 is time sliced out, and P2 and P3 are allowed to run sooner than FCFS
- average response time is fast at 3.66 ms
  - Compare to FCFS w/ long 1st process

Process	CPU Execution Time (ms)
P1	24
P2	3
P3	3



Round Robin Scheduling

- Weighted Round Robin - each process is given some number of time slices, not just one per round
  - In previous example, could give P1 2 time slices, and P2 and P3 only 1 each round



Round Robin Scheduling

- Weighted Round Robin is a way to provide preferences or priorities even with preemptive time slicing
  - Example: If 3 processes all want a great deal of compute time, & OS gives P1 2 time slots per round, P2 1 time slot/round, and P3 4 time slots/round, then in steady state, P1 gets 2/7 of CPU bandwidth, P2 gets 1/7 of CPU, and P3 gets 4/7 of CPU
  - In general, if process P_i gets N_i slots per round, then the fraction α_i of the CPU bandwidth that process i gets in steady state in WRR is:

$$\alpha_\iota = \frac{N_i}{\sum_i N_i}$$

CSCI 3753
Operating Systems

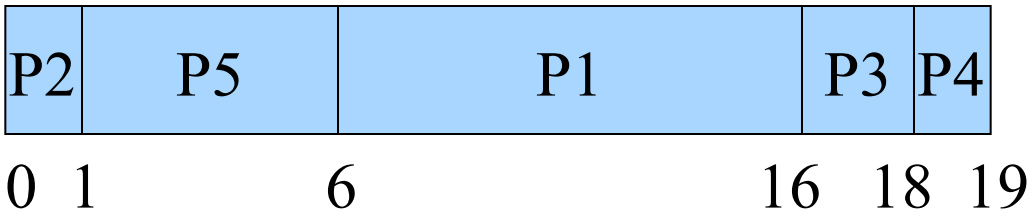
CPU Scheduling (Advanced)

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/21/13

Priority Scheduling

- Assign each task a priority, and schedule higher priority tasks first
- Priority can be based on
  - any measurable characteristics of the process, or
  - some external criteria
- Can be preemptive

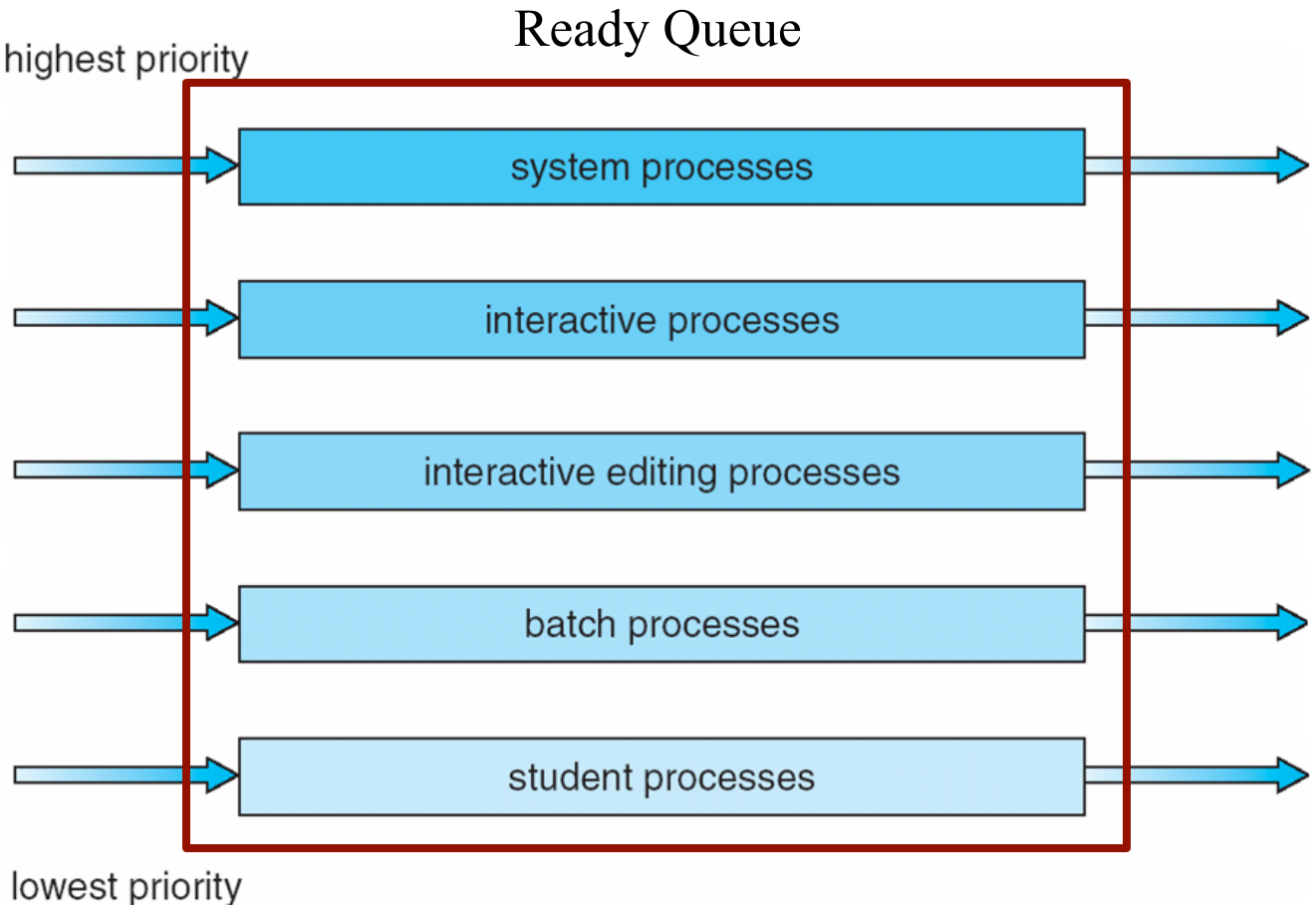
Process	CPU Execution Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Multi-level Queue Scheduling

- Use priorities to partition the ready queue into several separate queues
  - Different processes have different needs, e.g. foreground and background
  - If there are multiple processes with the same priority queue, need to apply a scheduling policy within that priority level
  - Don’t have to apply the same scheduling policy to every priority level, e.g. foreground gets EDF, background gets RR or FCFS
- Queues can be organized by priority, or each given a percentage of CPU, or a hybrid combination

Multilevel Queue Scheduling



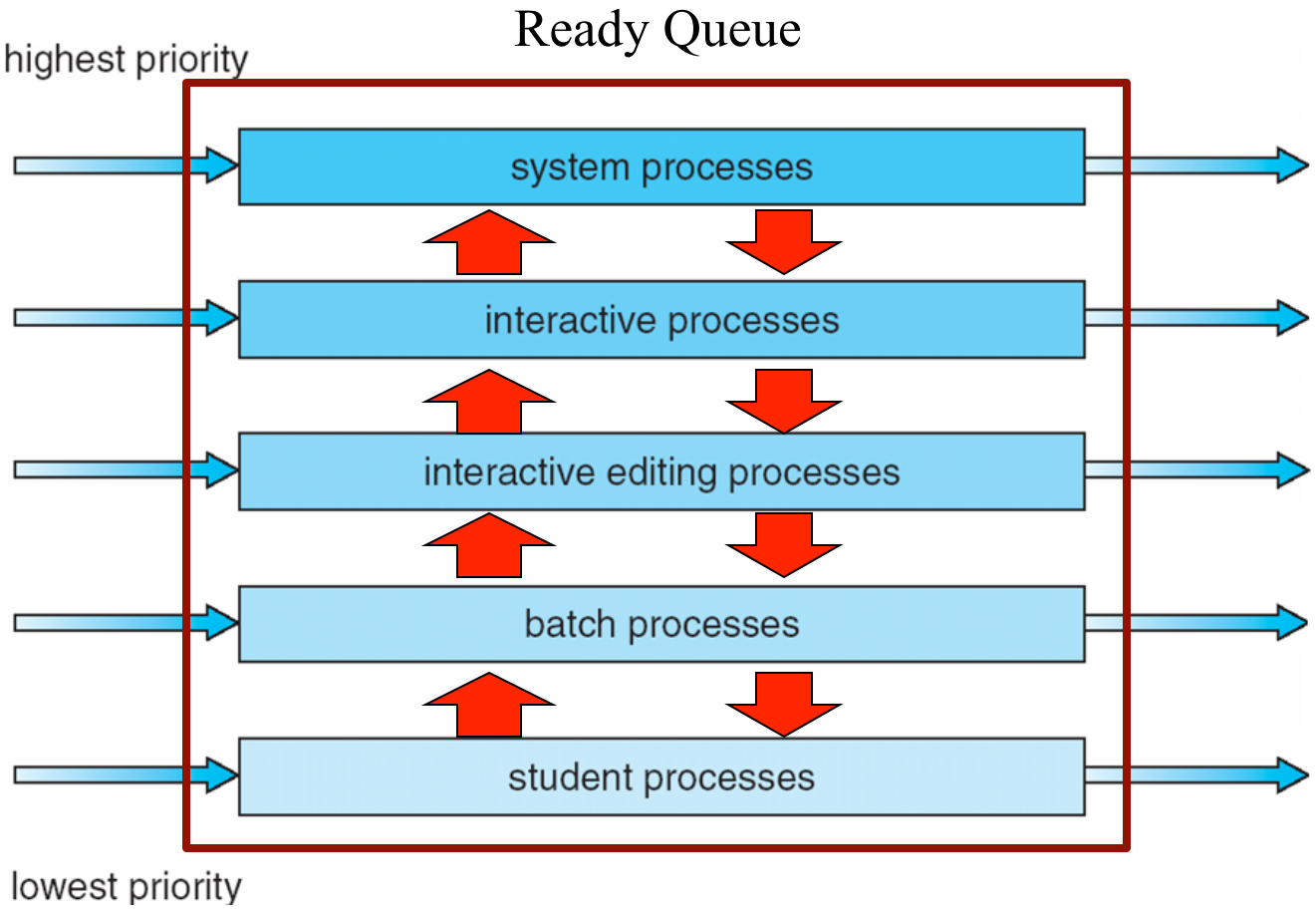
Priority Scheduling

- Preemptive priorities can starve low priority processes
  - A higher priority process always gets served ahead of a lower priority process, which never sees the CPU
- The solution is *multi-level feedback queues* that allow a process to move up/down in priority
  - Avoids starvation

Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue Scheduling



Criteria for Process Movement

1. Age of a process: old processes move to higher priority queues, or conversely, high priority processes are eventually demoted
   - Sample aging policy: if priorities range from 1-128, can decrease (increment) the priority by 1 every T seconds
   - Eventually, the low priority process will get scheduled on the CPU

2. Behavior of a process (CPU bound vs I/O bound)
   - Give higher priority to I/O bound processes: allows higher parallelism between CPU and I/O
   - A process typically alternates between bursts of I/O activity and CPU activity
   - Move a process down the hierarchy of queues during CPU burst, allowing interactive and I/O-bound processes to move up
   - Give a time slice to each queue, with smaller time slices higher up
   - If a process uses its time slice completely (CPU burst), it is moved down to the next lowest queue
   - Over time, a process gravitates towards the time slice that typically describes its average local CPU burst

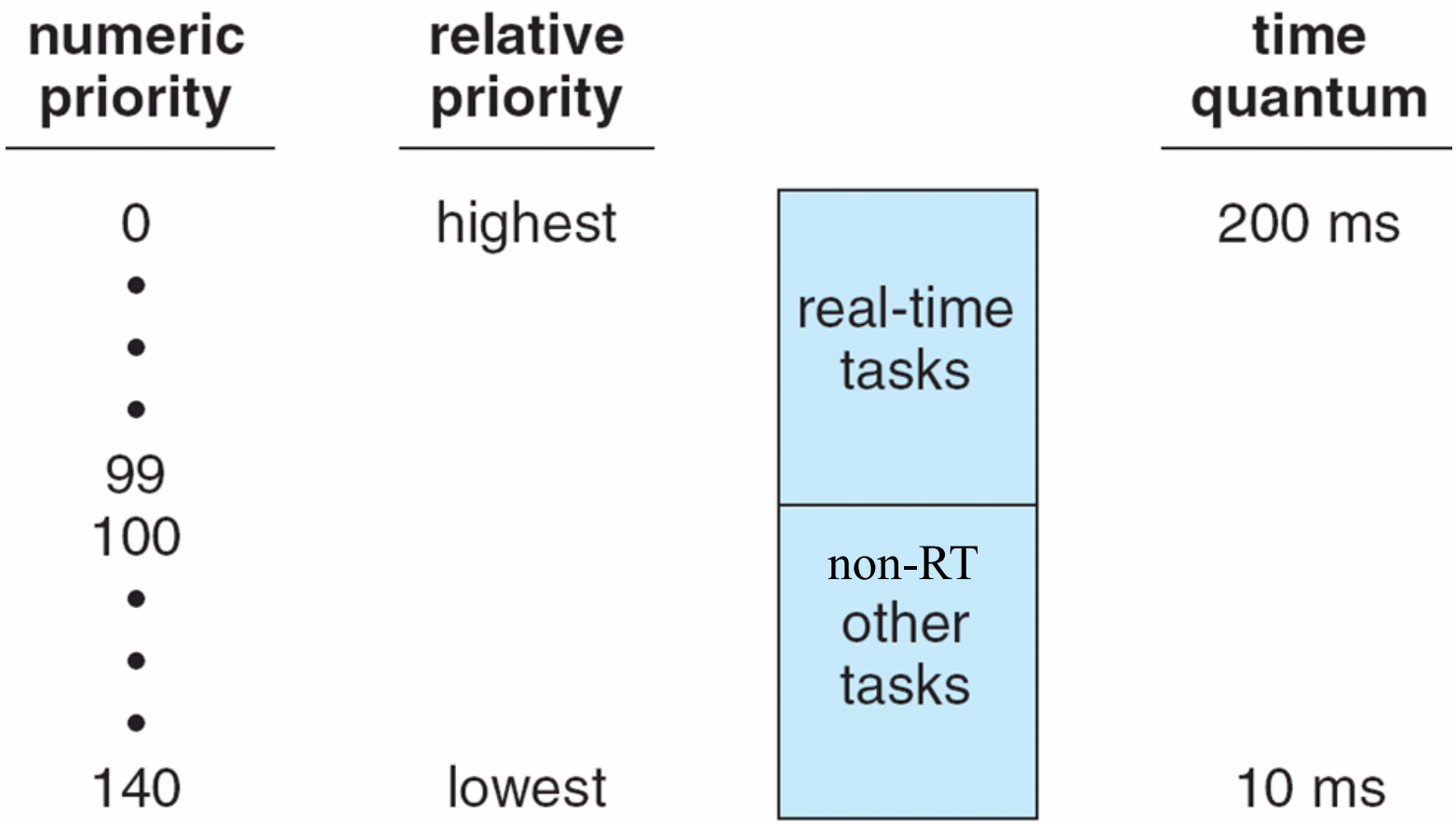
Priority Scheduling

- In Unix/Linux, you can *nice* a process to set its priority, within limits
  - e.g. priorities can range from -20 to +20, with lower values giving higher priority, a process with ‘nice +15’ is “nicer” to other processes by incrementing its value (which lowers its priority)
    - e.g. if you want to run a compute-intensive process compute.exe with low priority, you might type at the command line “nice –n 19 compute.exe”
  - To lower the niceness, hence increase priority, you typically have to be root
  - Different schedulers will interpret/use the nice value in their own ways

Multi-level Feedback Queues

- In Windows XP and Linux, system & real-time processes are grouped in a range of priorities that are higher in priority than the range of priorities given to non-real-time processes
  - XP has 32 priorities. 1-15 are for normal processes, 16-31 are for real-time processes. One queue for each priority.
    - XP scheduler traverses queues from high priority to low priority until it finds a process to run
  - In Linux, priorities 0-99 are for important/real-time processes while 100-139 are for 'nice' user processes. Lower values mean higher priorities.
    - Also, longer time quanta for higher priority tasks (200 ms for highest) and shorter time quanta for lower priority tasks (10 ms for lowest).

Linux Priorities and Time-slice length



Multi-level Feedback Queues

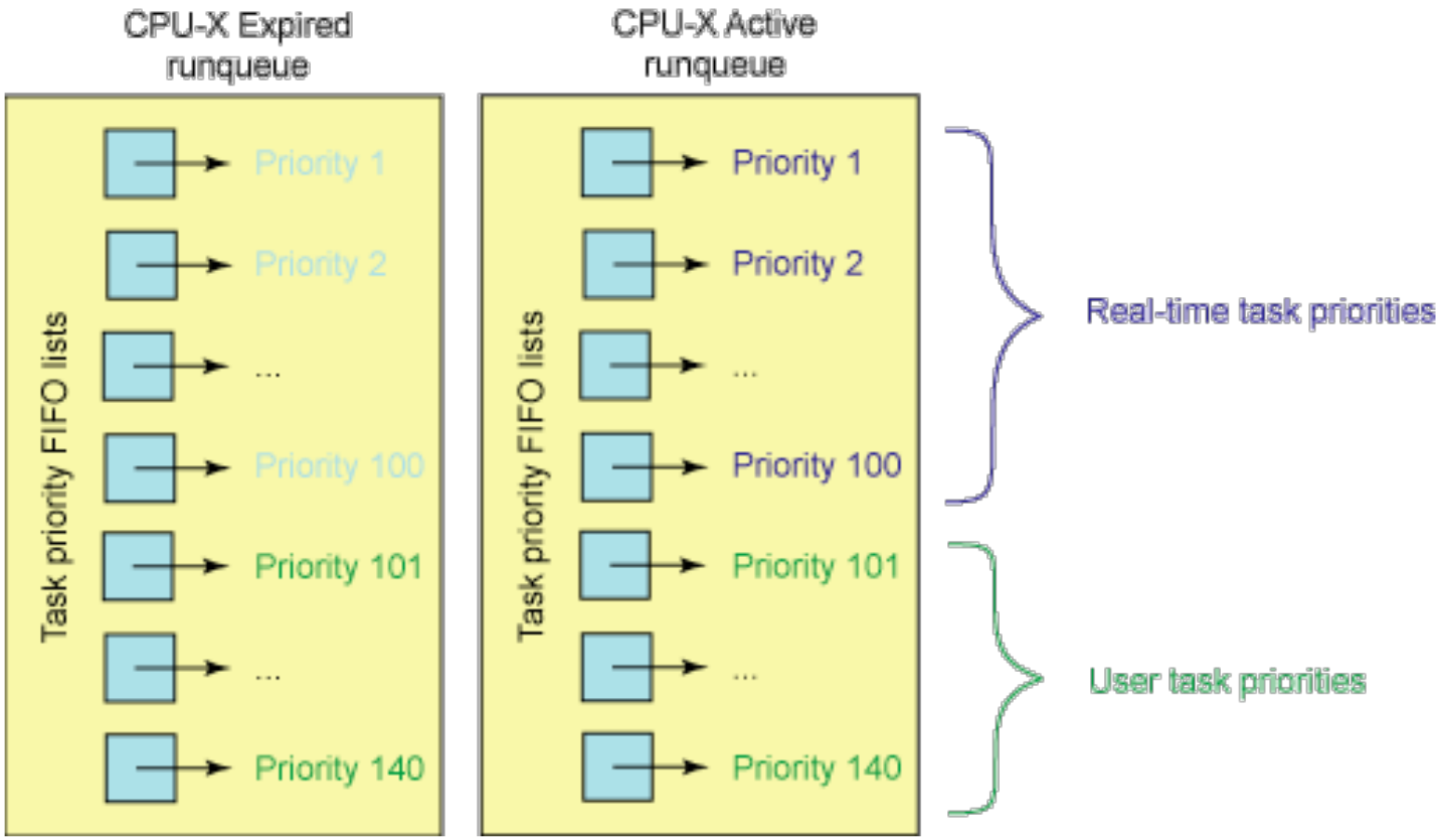
- Most modern OSs use or have used multi-level feedback queues for priority-based preemptive scheduling
  - Windows NT/XP, Mac OS X, FreeBSD/NetBSD and Linux pre-2.6
    - Linux 1.2 used a simple round robin scheduler
    - Linux 2.2 introduced scheduling classes (priorities) for real-time and non-real-time processes and SMP support
    - Linux 2.4 introduced an O(N) scheduler – help interactive processes
  - But Linux 2.6-2.6.23 uses an O(1) scheduler
    - Iterate over fixed # of 140 priorities to find the highest priority task – scales well because larger # processes doesn't affect time to find best next task to schedule
  - And Linux 2.6.23+ uses a "Completely Fair Scheduler"

O(N) Scheduler

- If an interactive process yields its time slice before it's done, then its "goodness" is rewarded with a higher priority next time it executes
- Keep a list of goodness of all tasks. But this was unordered. So had to search over entire list of N tasks to find the "best" next task to schedule – hence O(N) – doesn't scale well
- SMP: A process could be scheduled on a different processor
  - Loses cache affinity
- SMP: Single runqueue lock
  - The act of choosing a task to execute locked out any other processors from manipulating the runqueues
- Preemption not possible

O(1) Scheduler in Linux

- Linux maintains two queues: an active runqueue and an expired runqueue, each indexed by 140 priorities
- Active runqueue contains all tasks with time remaining in their time slices, and expired runqueue contains all expired tasks.
- Once a task has exhausted its time slice, it is moved to the expired runqueue and is not eligible for execution again until all other tasks have exhausted their time slice

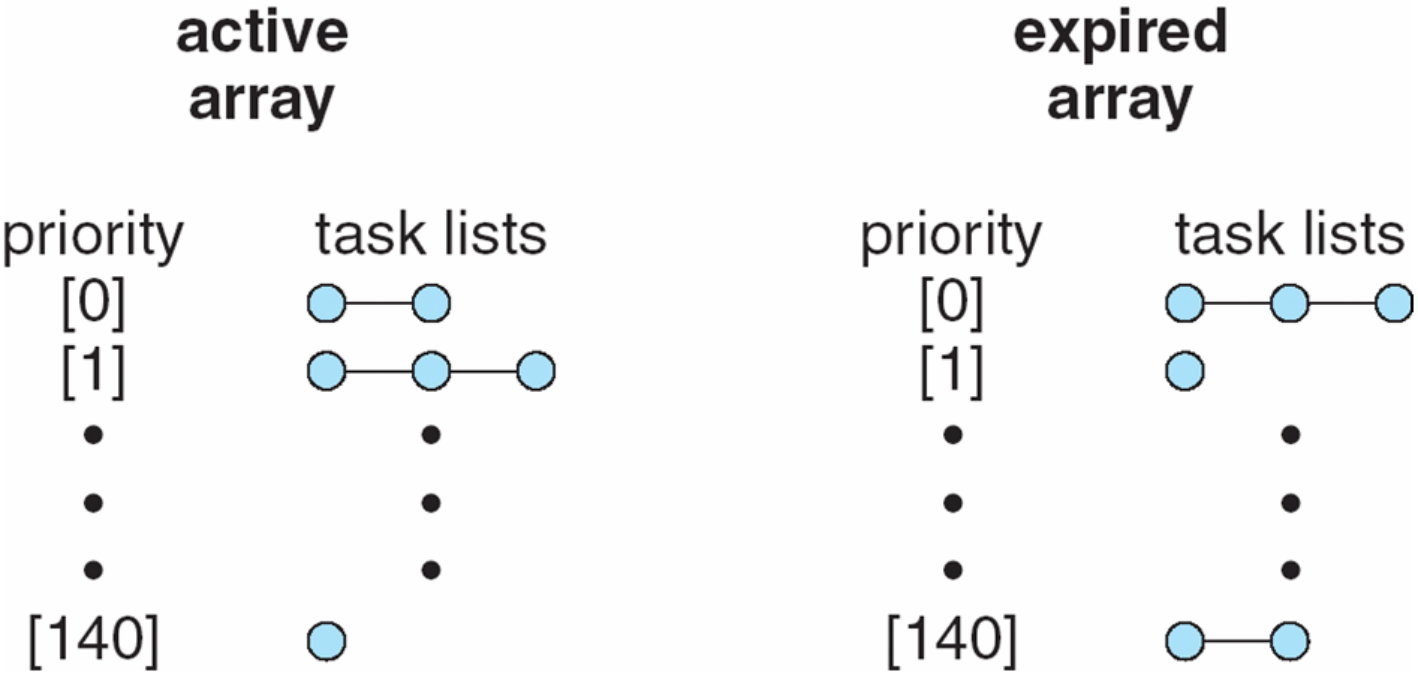


From http://www.ibm.com/developerworks/linux/library/l-scheduler/

O(1) Scheduler

- Scheduler chooses task with highest priority from active runqueue
  - Just search linearly up the active runqueue from priority 1 until you find the first priority whose queue contains at least one unexpired task
  - # of steps to find the highest priority task is in the worst case 140, so it's bounded and depends only on the # priorities, not # of tasks,
  - Hence this is O(1) in complexity

O(1) Scheduler in Linux



O(1) Scheduler in Linux

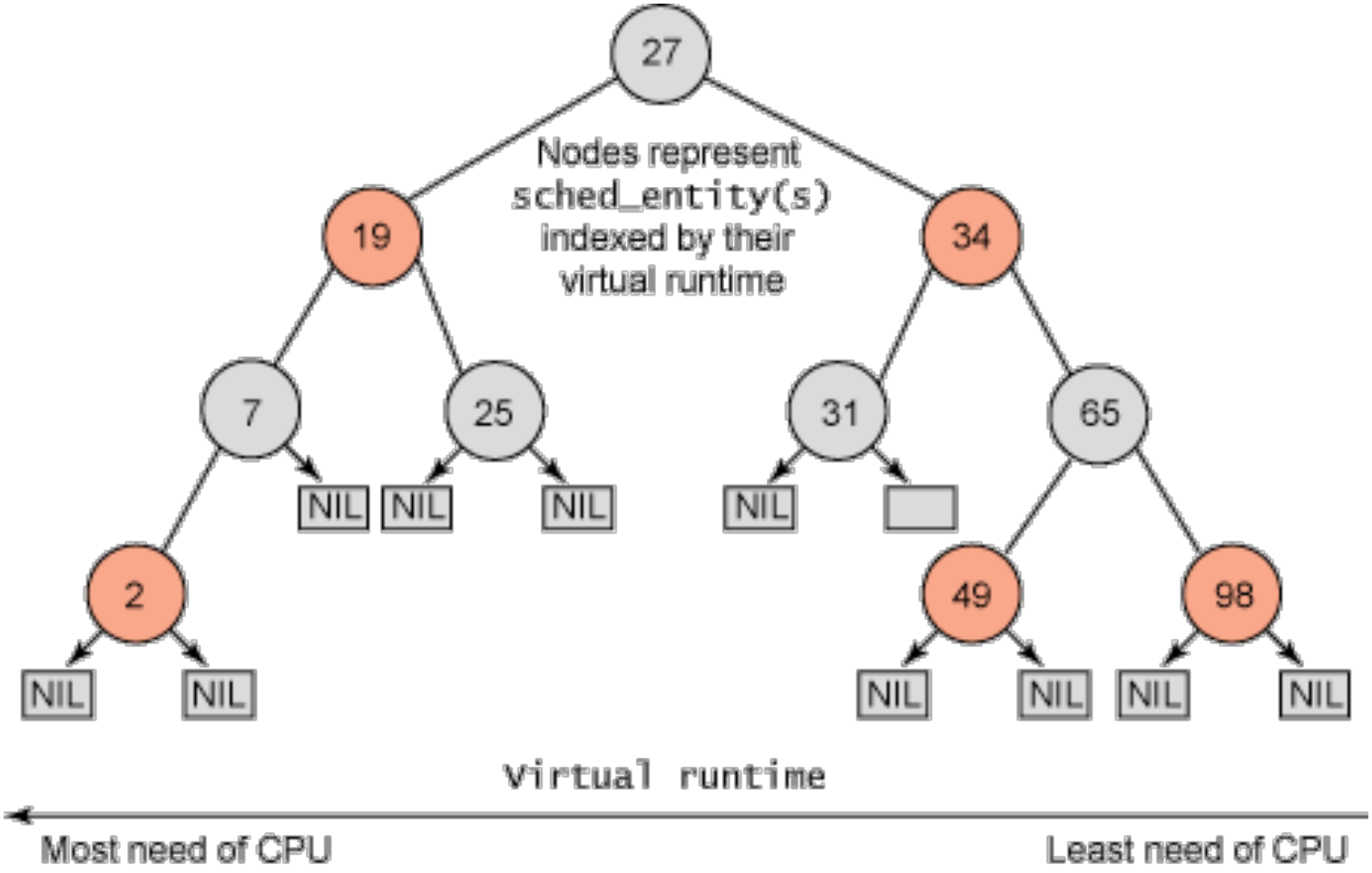
- When all tasks have exhausted their time slices, the two priority arrays are exchanged, and the expired array becomes the active array
- When a task is moved from run to expired, Linux recalculates its priority according to a heuristic
  - New priority = nice value +/- f(interactivity), where f() can change the priority by at most +/-5, and is closer to -5 if a task has been sleeping while waiting for I/O (interactive tasks tend to wait longer times for I/O, and thus their priority is boosted -5), and closer to +5 for compute-bound tasks
  - This dynamic reassignment of priorities affects only the lowest 40 priorities for non-RT/user tasks (corresponds to the nice range of +/- 20)
- The O(1) Scheduler was the default scheduler in Linux

Completely Fair Scheduler (CFS)

- Heuristics of O(1) for dynamic reassignment of priorities were complicated and somewhat arbitrary
- Linux 2.6.23+ has a "completely fair" scheduler
- Virtual run time (vruntime): The amount of time a process has used the CPU so far
  - The smaller the virtual run time, the more need for the processor. This is fair.
  - Decay factor: CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute
    - low priority → high decay factor → the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task
    - elegant solution to avoid maintaining run queues per priority

CFS

- Use (time-ordered) red-black tree instead of queue
  - Self balancing B-Tree: No path in the tree will ever be more than twice as long as any other
  - Insert/delete/search occur in O(log *n*) time
- Lower vruntime processes are on the left side of the tree
- As tasks run more, their virtual run time increases, and they migrate to other positions further to the right in the tree



From http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

Real Time Scheduling in Linux

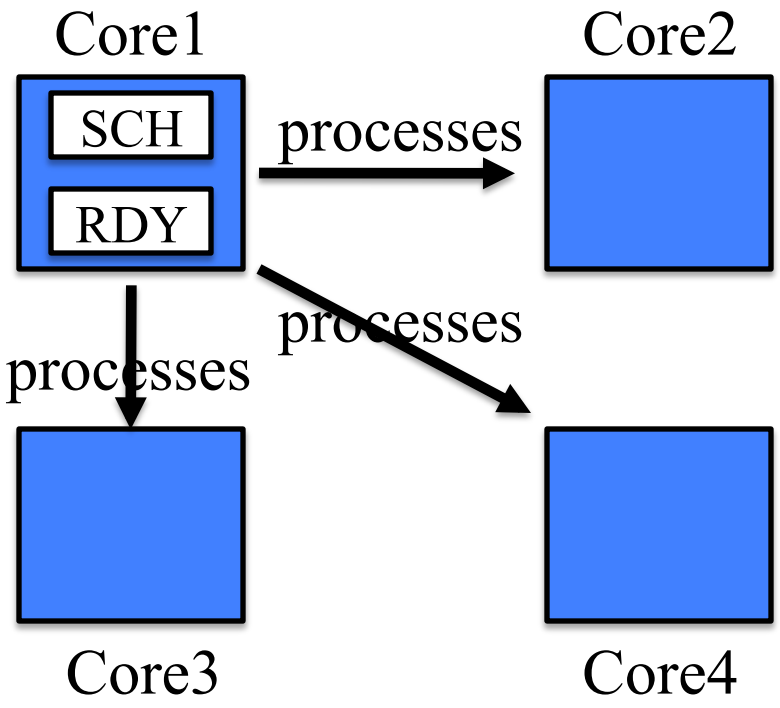
- Linux also includes two real-time scheduling classes:
  - Real time Round Robin
  - Real time FIFO
  - These are soft real time scheduling algorithms, not hard real time scheduling algorithms with absolute deadlines
- Only processes with the priorities 0-99
- "When a Real time FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. All other tasks of lower priority will not be scheduled until it relinquishes the CPU. Two equal-priority Real time FIFO tasks do not preempt each other."

http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler.

Multi-Core Scheduling

- Multicore processors
  - Multiple processor cores on the same physical chip
  - Each core maintains its architectural state and thus appears to OS as a separate physical processor
- Memory stall: waiting time to get data from memory
  - Cache miss, etc.
- Multithreaded processor cores
  - Two or more hardware threads assigned to a single core
  - UltraSPARC T3: 16 cores per chip, 8 hardware threads per core →appears as 128 logical processors to OS

Multi-core Scheduling

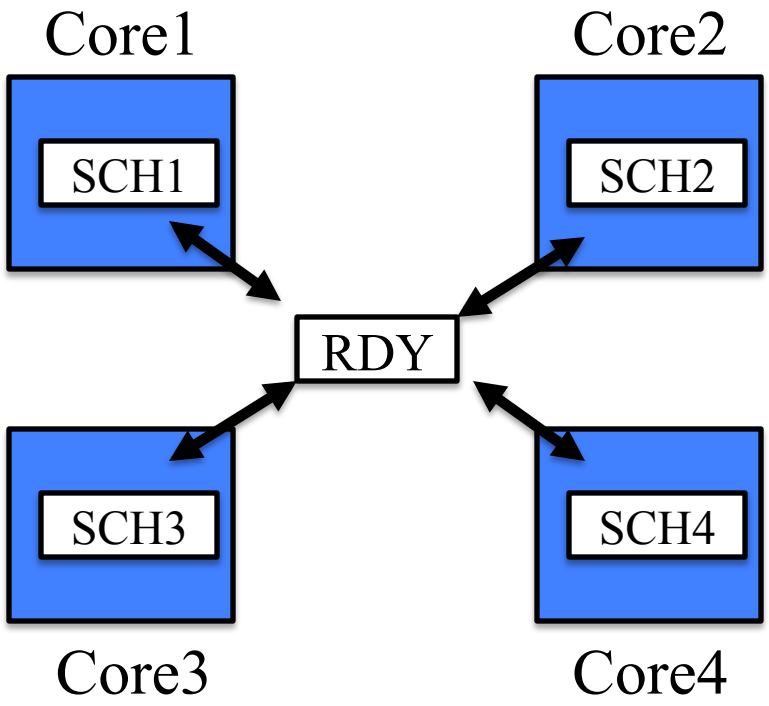


SCH = Scheduler, RDY = Ready Queue

- Variety of multi-core schedulers being tried. We'll just mention some design themes.
- In *asymmetric multiprocessing* (left) – 1 CPU handles all scheduling, decides which processes run on which cores

Multi-core Scheduling

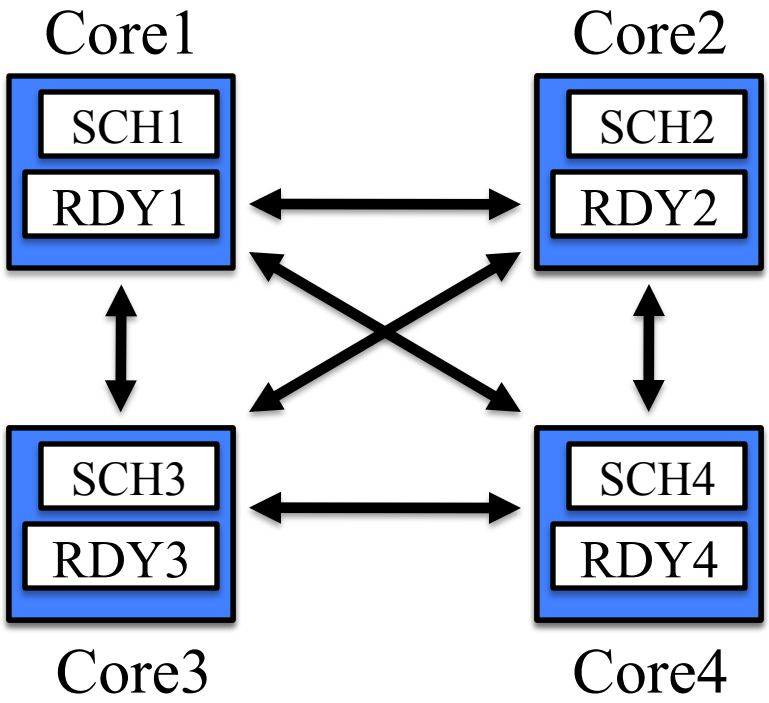
- In symmetric multi-processing (SMP), each process is self-scheduling. All modern OSs support some form of SMP. Two types:
  1. All cores share a single global ready queue.



- Here, each core has its own scheduler. When idle, each scheduler requests another process from the shared ready queue. Puts it back when time slice done.
  - Synchronization needed to write/read shared ready queue

Multi-core Scheduling

2. Another self-scheduling SMP approach is when each core has its own ready queue
   – Most modern OSs support this paradigm



- a typical OS scheduler plus a ready queue designed for a single CPU can run on each core…
- Except that processes now can migrate to other cores/processors
  - There has to be some additional coordination of migration

Multi-core Scheduling

- Caching is important to consider
  - Each CPU has its own cache to improve performance
  - If a process migrates too much between CPUs, then have to rebuild L1 and L2 caches each time a process starts on a new processor
    - L3 caches that span multiple processors can help alleviate this, but there is a performance hit, because L3 is slower than L1 and L2. In any case, L1 and L2 caches still have to be rebuilt.
  - So processes tend to stick to a given processor – processor affinity
    - Hard vs. soft affinity. In hard affinity, a process specifies via a system call that it insists on staying on a given CPU. In soft affinity, there is still a bias to stick to a CPU, but processes can on occasion migrate.

Multi-core Scheduling: Load balancing

- Goal: Keep workload evenly distributed across processors. Otherwise, some CPUs will be under-utilized.
- When there is a single shared ready queue, there is automatic load balancing, because cores just pull in processes from the ready queue whenever they're idle.
- For separate ready Q's,
  - push migration – a dedicated task periodically checks the load on each CPU, and if imbalance, pushes processes from more-loaded to less-loaded CPUs
  - Pull migration – whenever a CPU is idle, it tries to pull a process from a neighboring CPU
  - Linux and FreeBSD use a combination of pull and push

Little's Law

- How big of a ready queue do we need?
  - Employ stochastic queueing theory to answer this question
  - Consider an abstract ready queue where λ = arrival rate of processes in the queue, μ = service rate of processes exiting the queue
  - In steady state, λ must be less than or equal to μ, i.e. $\lambda \leq \mu$
  - Let W = average wait time in the queue per process
    - Note W is not the inverse of the service time μ. Analogy: If we have a series of cars that go into a tunnel, eventually they will emerge on the other side, and the steady state service (exit) rate of the cars will equal the arrival (entry) rate. However, the exit rate doesn't tell us what is the average time spent by each car in the tunnel. The length of the tunnel (and car speed) tells how long each car spent in the tunnel.
  - Let N = average queue length

Little's Law

- Then $N = \lambda * W$ (Little's Law)
  - In W seconds, on average λ*W processes arrive in the ready queue.
  - This conclusion is valid for any scheduling algorithm and arrival distribution.
  - Useful to know how big on average the queue size N should be for allocation by the OS.