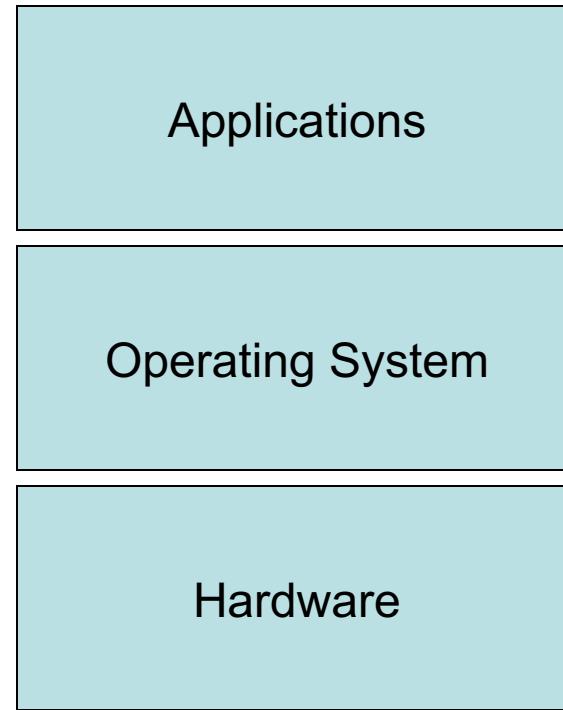


What is an Operating System?

- Name some OSs:
Windows, Linux, Mac
OS X, Google Android,
...
• What is common
across these OSs?
• An operating system is
a layer of software
between applications
and hardware



What is an Operating System?

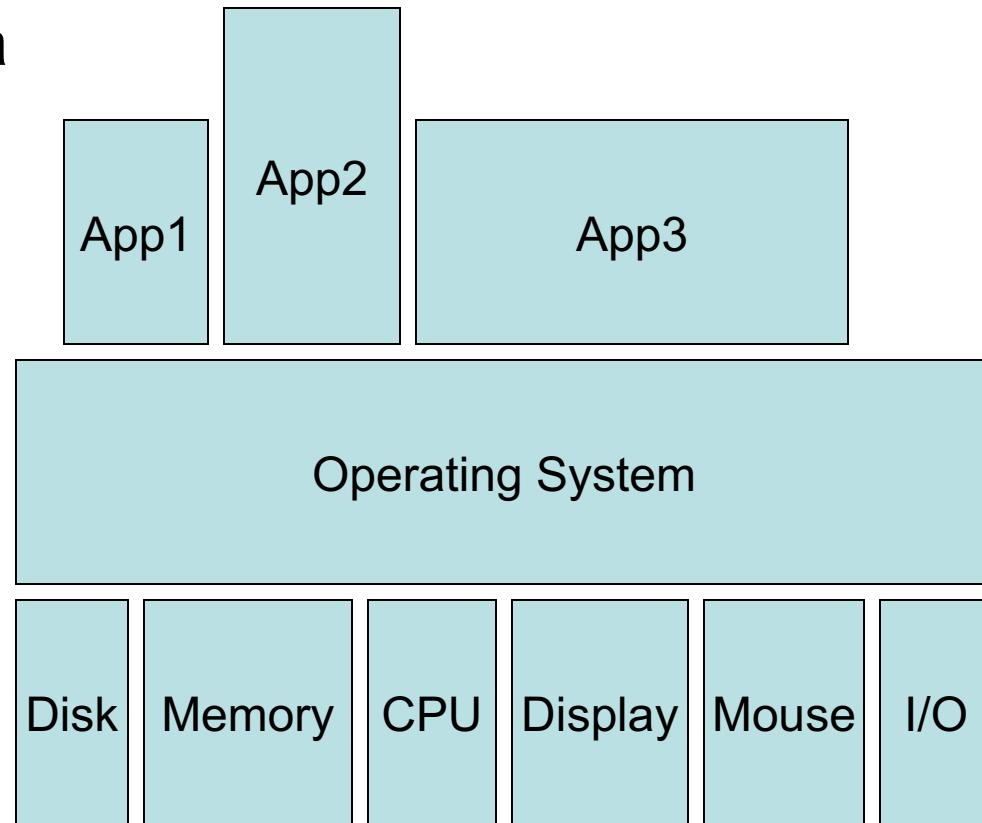
- An operating system is a layer of software between *many* applications and *diverse* hardware that

- Provides a hardware abstraction* so an application doesn't have to know details about the hardware.

- otherwise an application saving a file to disk would have to know how the disk operates

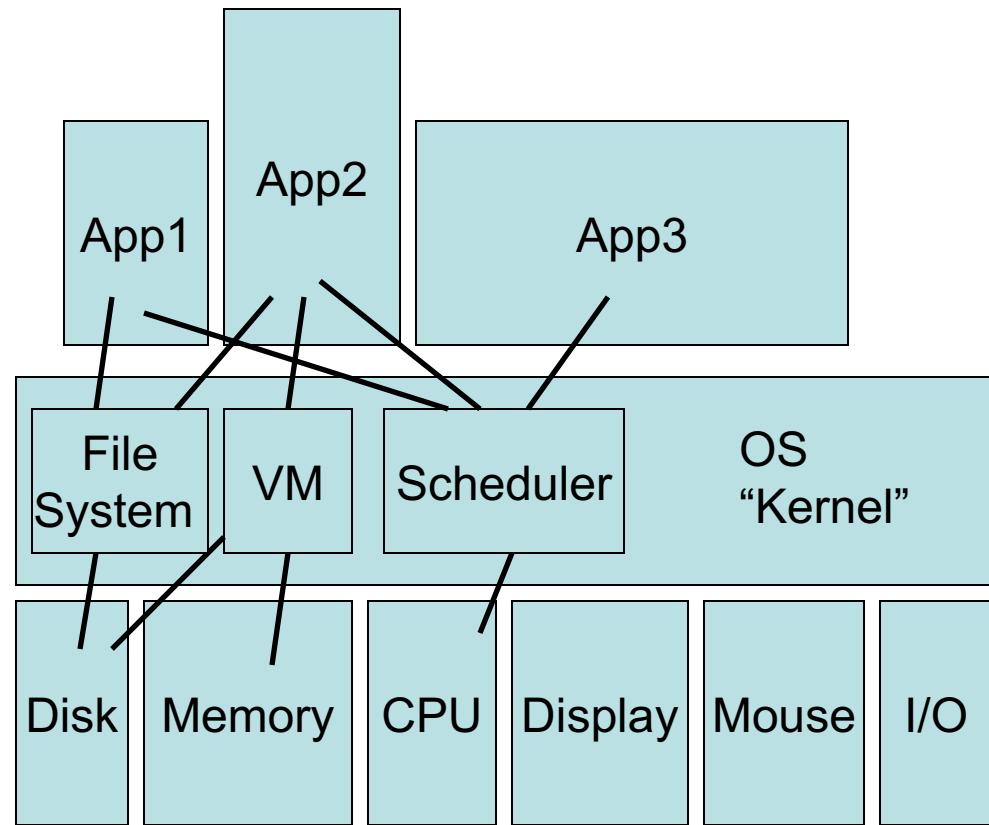
- Arbitrates access to resources among multiple applications:*

- Sharing* of resources
 - Isolation* protects app's from each other



What is an Operating System?

- A PC operating system consists of multiple components
 - scheduler
 - virtual memory system
 - file system
 - device management
 - other...



What is an Operating System?

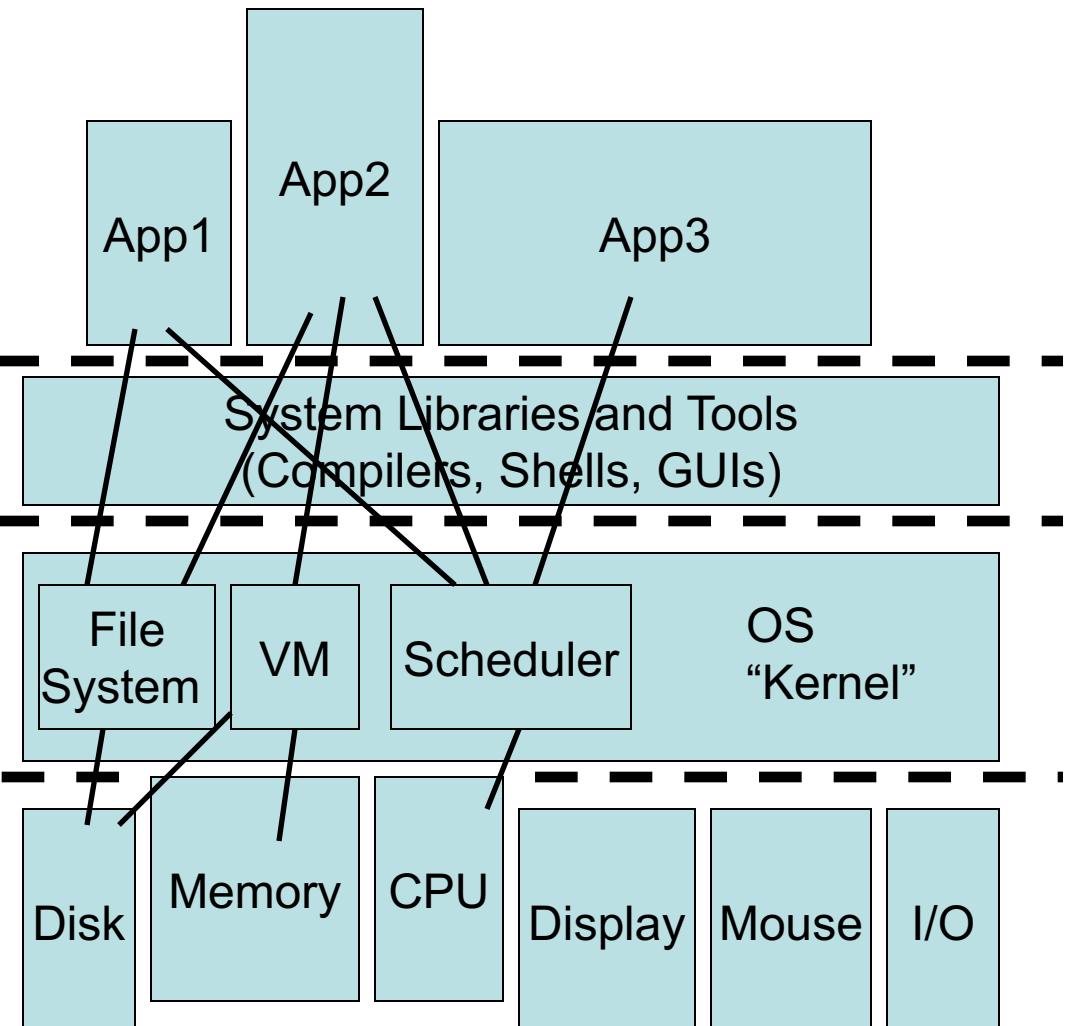
Posix, Win32,
Java, C library API

System call API

– 160 in Linux

Device driver “API”

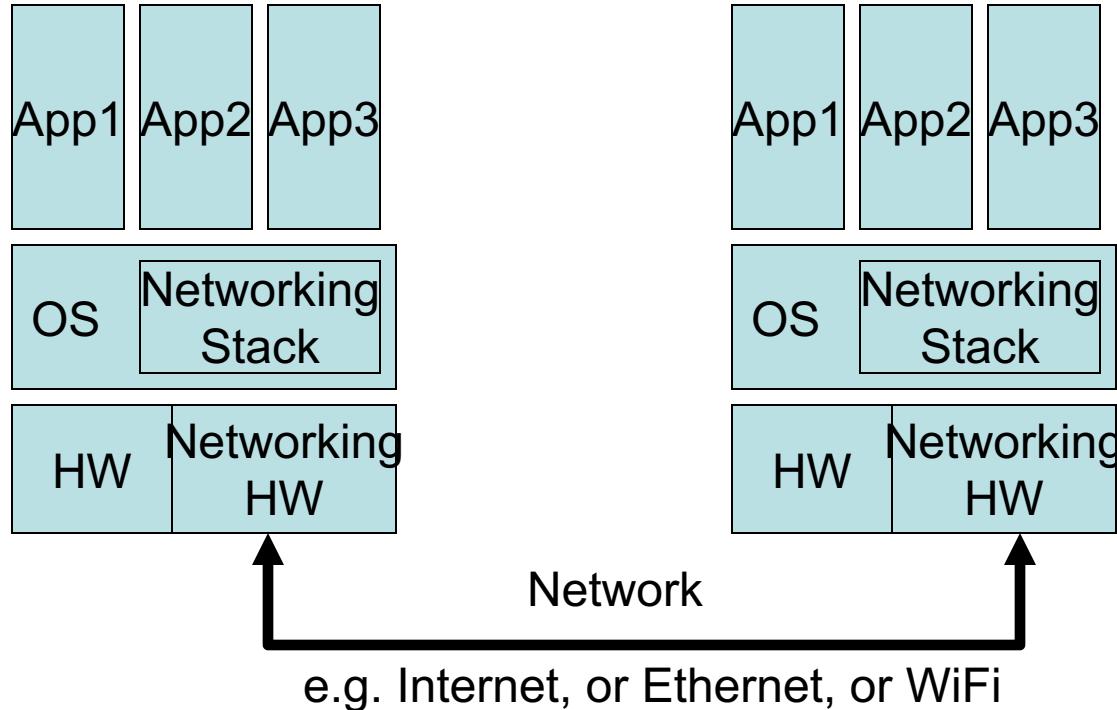
Note: different OS kernels can
support the same system call API



Outline of the OS course

1. Hardware supporting design, user/supervisor mode, system calls, trap table, device I/O, interrupts, DMA, mem-mapped I/O
2. Processes, threads, scheduling, synchronization, deadlock
3. Memory management, paging, virtual memory
4. File system design, allocation, networked file systems
5. Security: authorization, access control

What is an Operating System?



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

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

Outline of the OS course

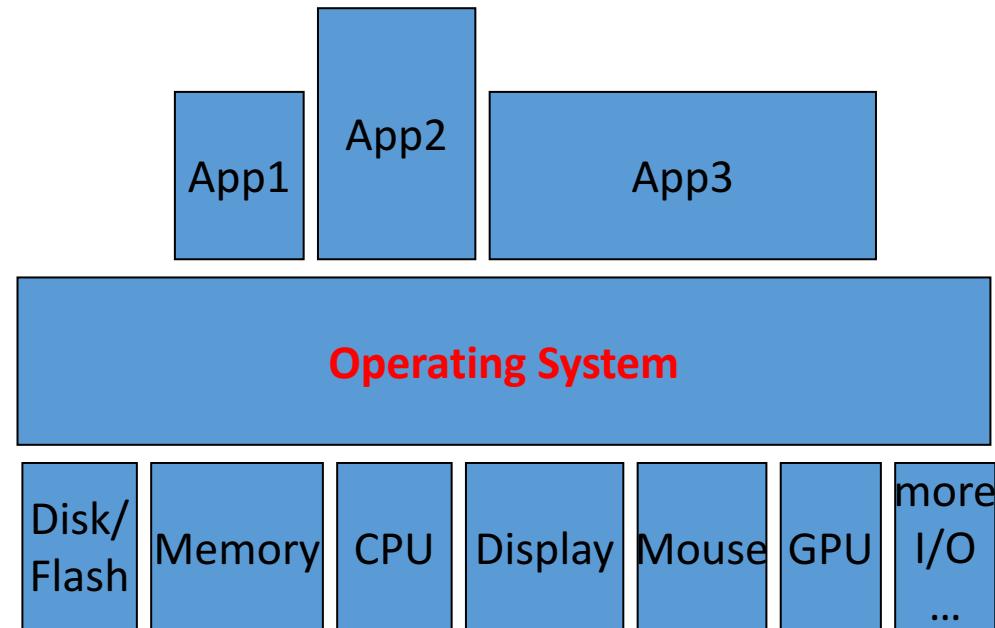
1. Hardware support, virtual machines, user/supervisor mode, system calls, trap table, device I/O, interrupts, DMA, memory-mapped I/O
2. Processes, threads, scheduling, synchronization, deadlock
3. Memory management, paging, virtual memory
4. File system design, allocation
5. Advanced Topics:
 - Networked file systems
 - Security: authorization, access control

What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

1. Provides a ***hardware abstraction*** so an application doesn't have to know the details about the hardware.

E.g. An application saving a file to disk doesn't have to know how the disk operates

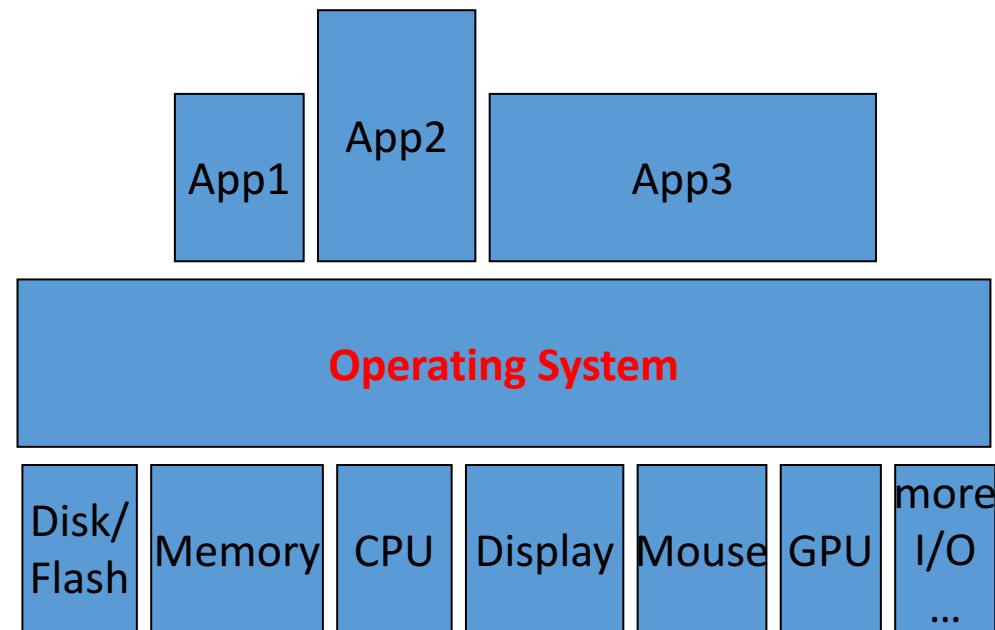


What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

2. **Arbitrates access** to resources among multiple applications:
+ Sharing of resources.

E.g. Sharing a Printer among applications

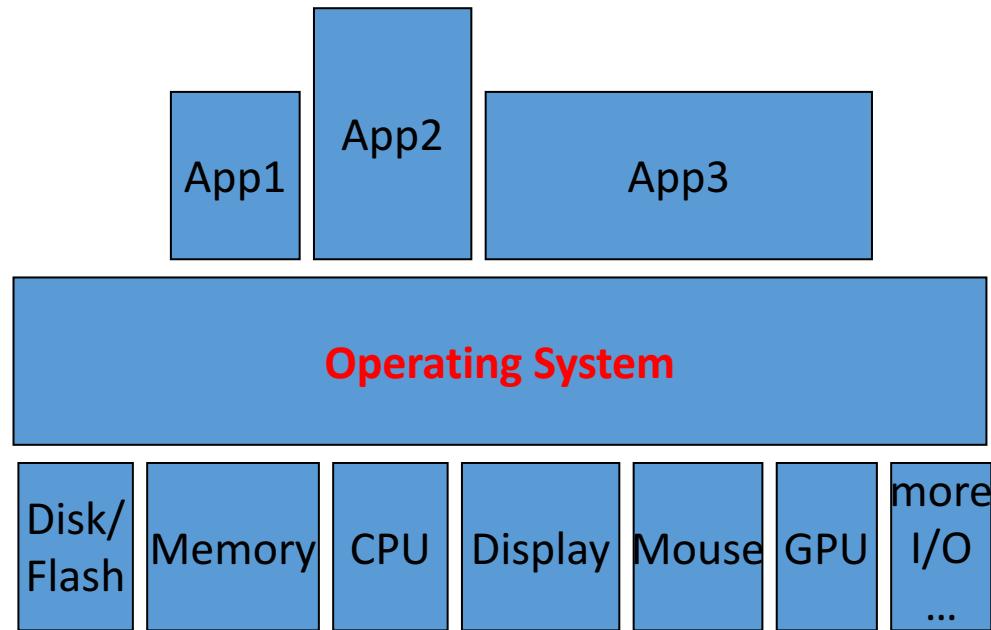


What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

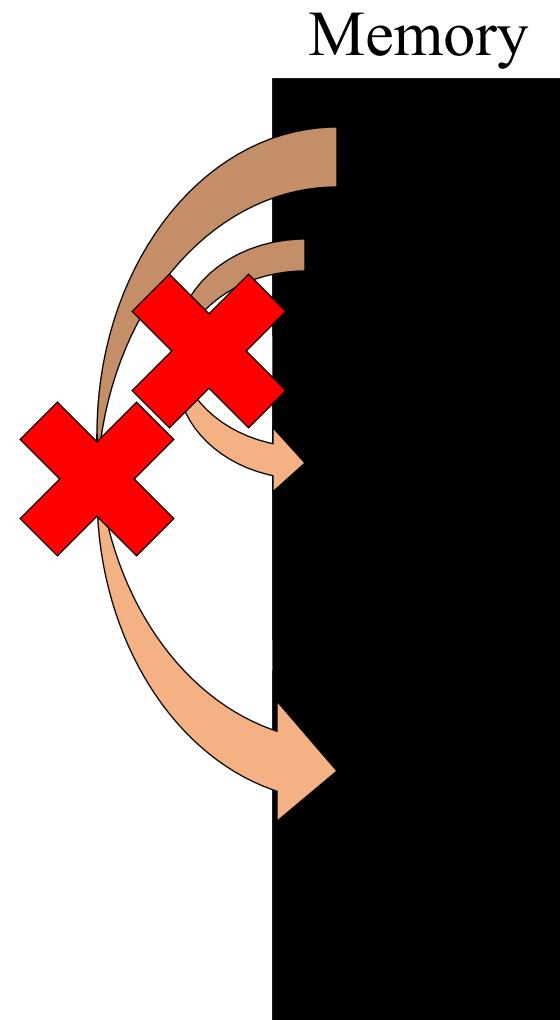
3 . Provide protections:

- *Isolation* protects **app's from each other**
- *Isolation* also to protect the **OS from applications**
- *Isolation to limit resource consumption by any one app*



Protection in Operating Systems

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



Privileged Instruction Examples

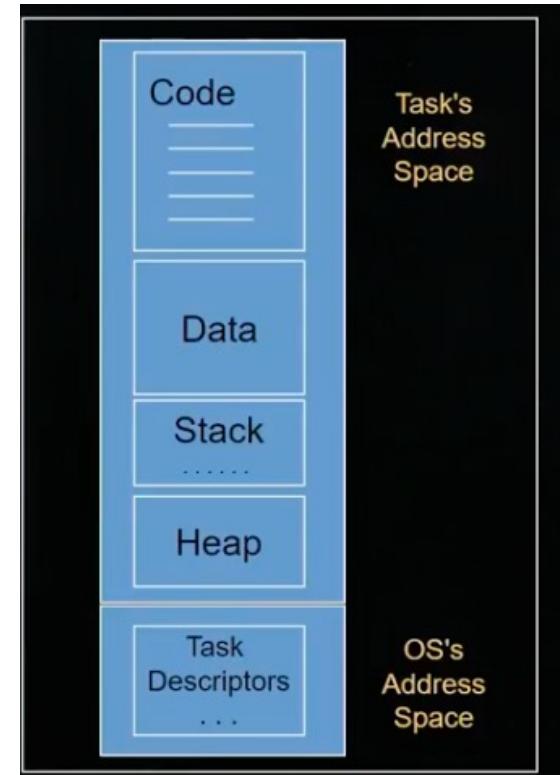
- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB (Translation Lookaside Buffer) entries
- Load and read system registers
- Change processor modes from K to U
- Change the voltage and frequency of processors
- Halt/reset processor
- Perform I/O operations

What is an unit of work for an OS?

- Application
- Task
- Job
- Process

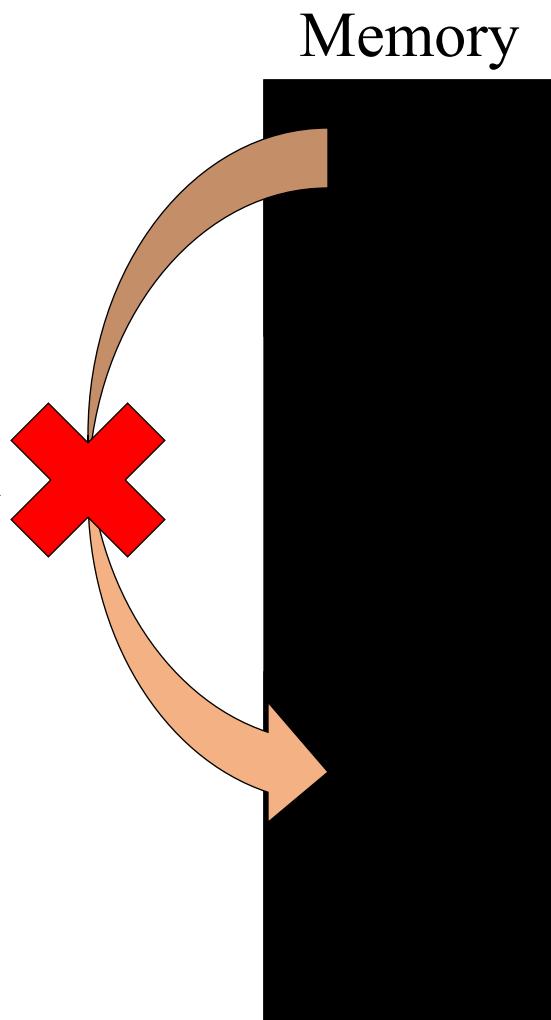
What does a TASK consist of?

- Code – placed into memory
- Data – stored in memory
- OS data for task – task descriptors



How can we access the OS functionality?

- **Problem:** If a task is protected from getting into the OS code and data, OS functionality are restricted from these tasks
- How does CPU know if a certain instruction should be allowed?
- How does OS grant a task access to certain OS data structures but not the other?
- How to switch from running the task's code to running OS's code
- Need to use a hardware assistant called **mode bit**

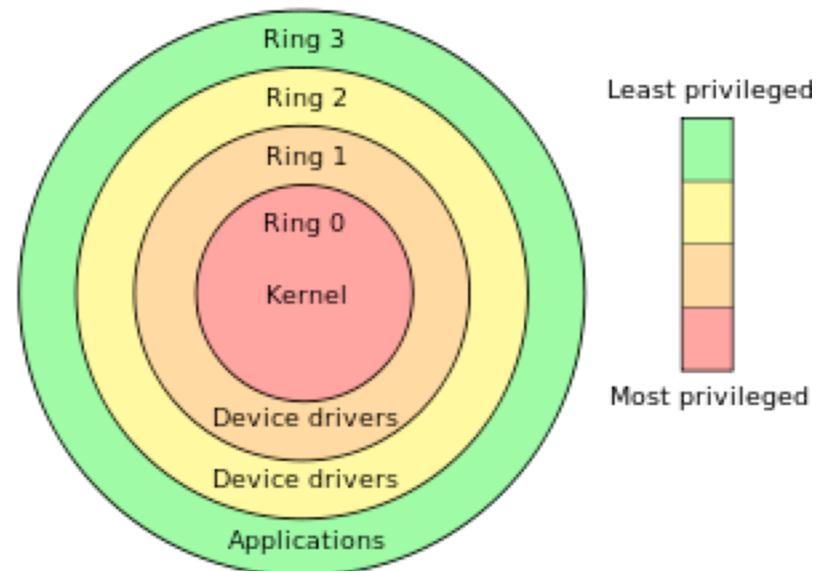


Kernel Mode vs User Mode

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

Multiple Rings/Modes of Privilege

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



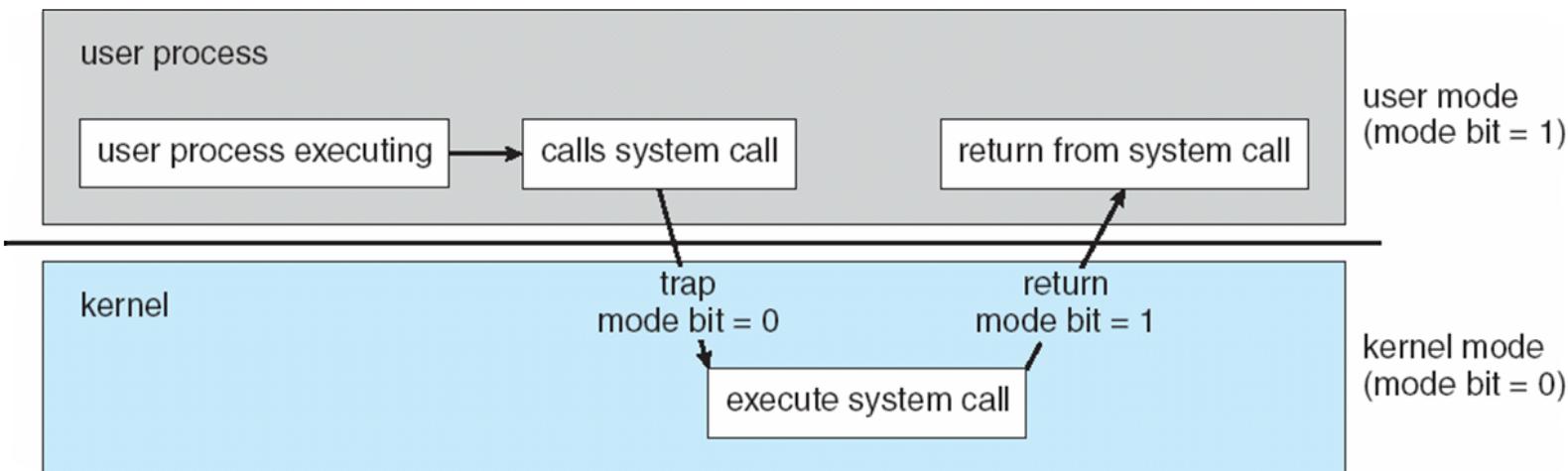
- **Virtual machines (one possible configuration)**
 - VM's hypervisor runs in ring 0, guest OS runs in ring 1 or 2, Apps run in ring 3



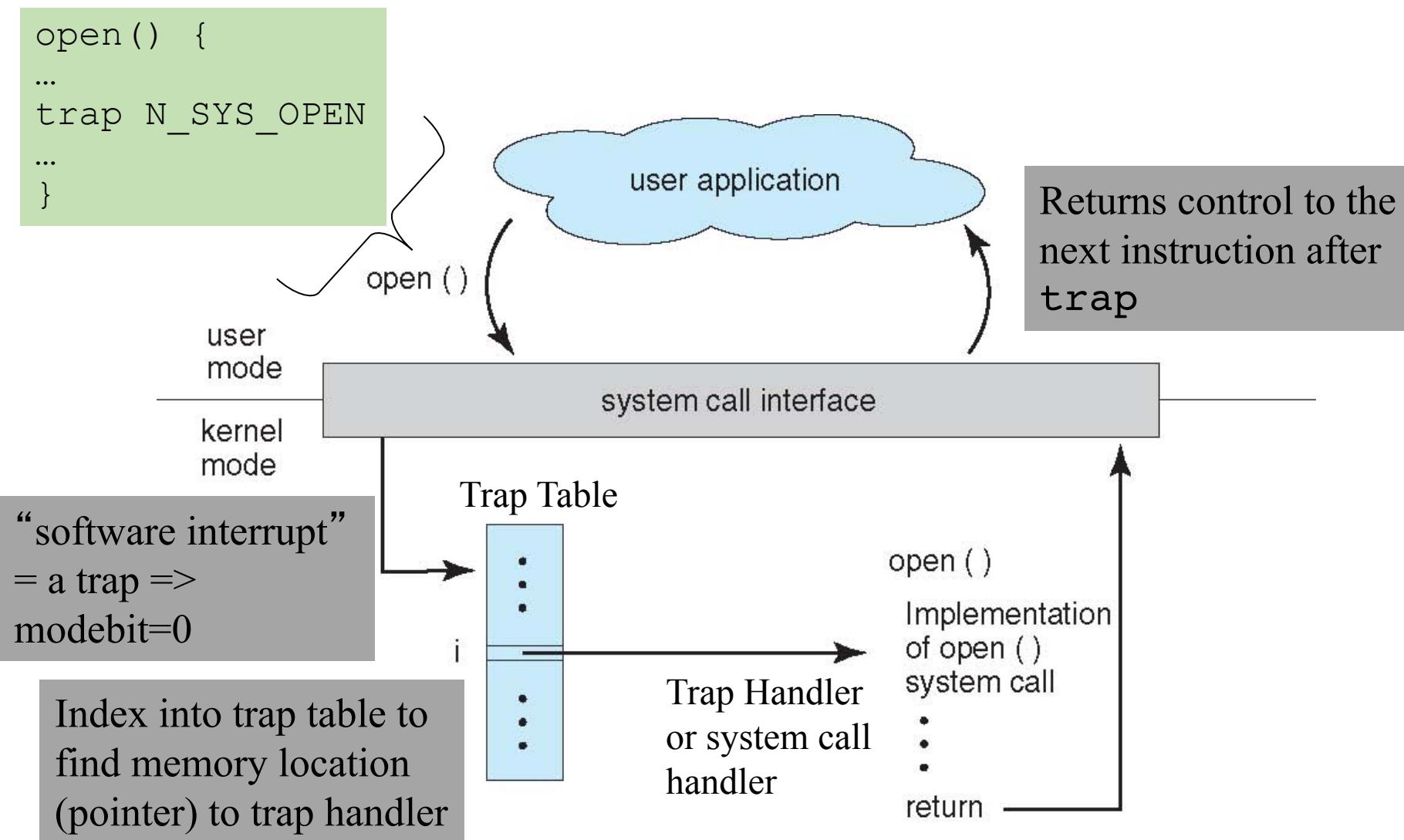
System Call

System Calls: How Apps and the OS Communicate

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



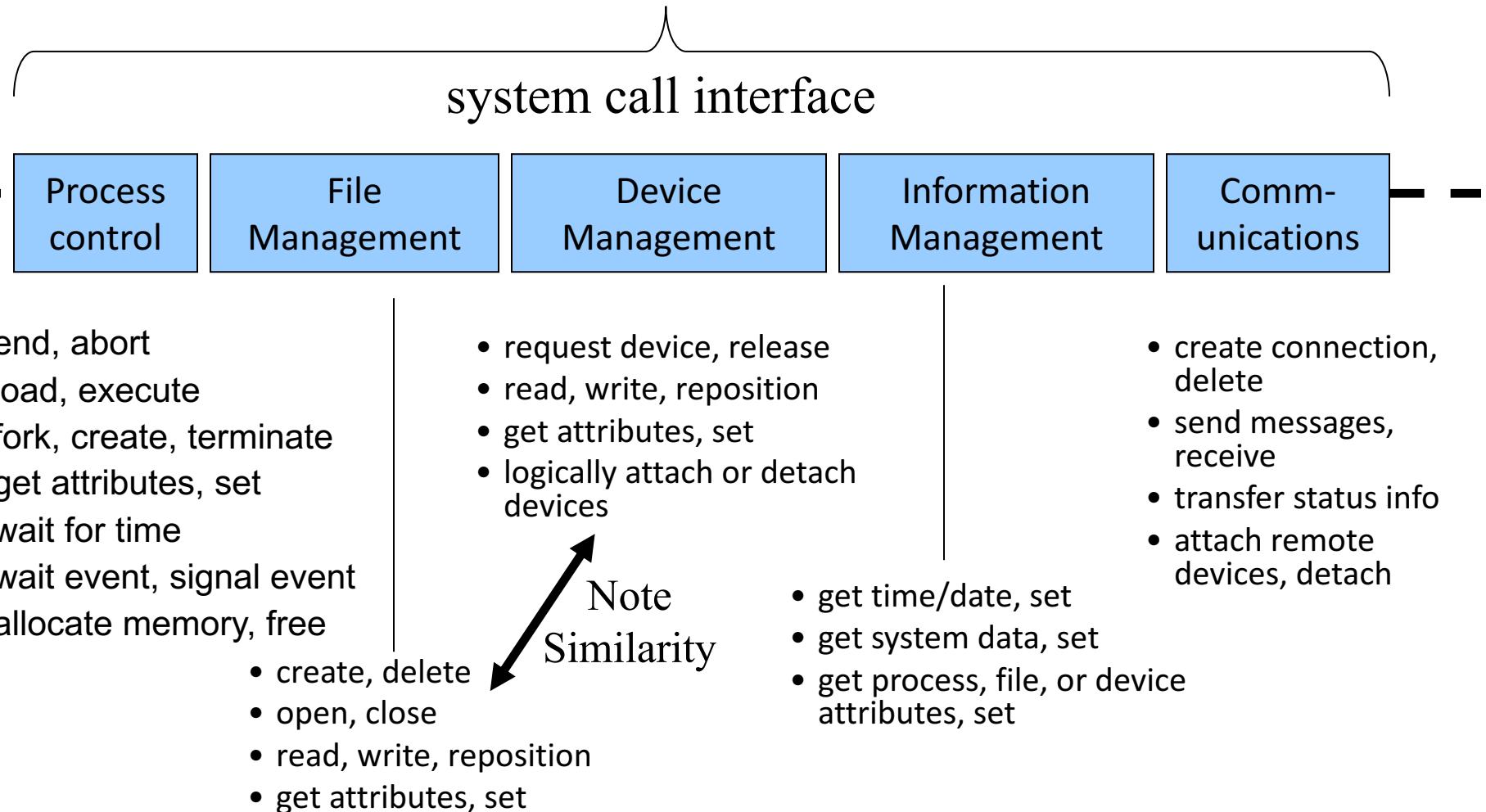
API – System Call – OS Relationship



Trap Table

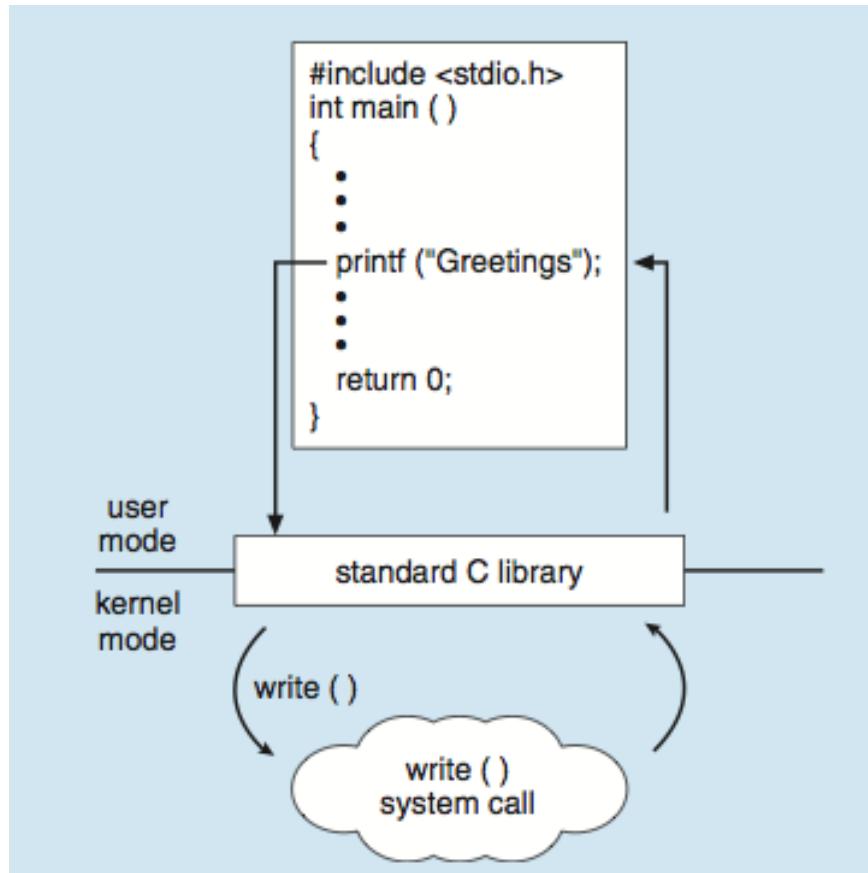
- Trap handling: The process of indexing into the trap table to jump to the trap handler routine is also called *dispatching*
- The trap table is also called a *jump table* or a *branch table*
- “A trap is a *software interrupt*”
- Trap handler (or system call handler) performs the specific processing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

Classes of System Calls Invoked by trap



Standard C Library Example

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



e.g. INT or SYSCALL

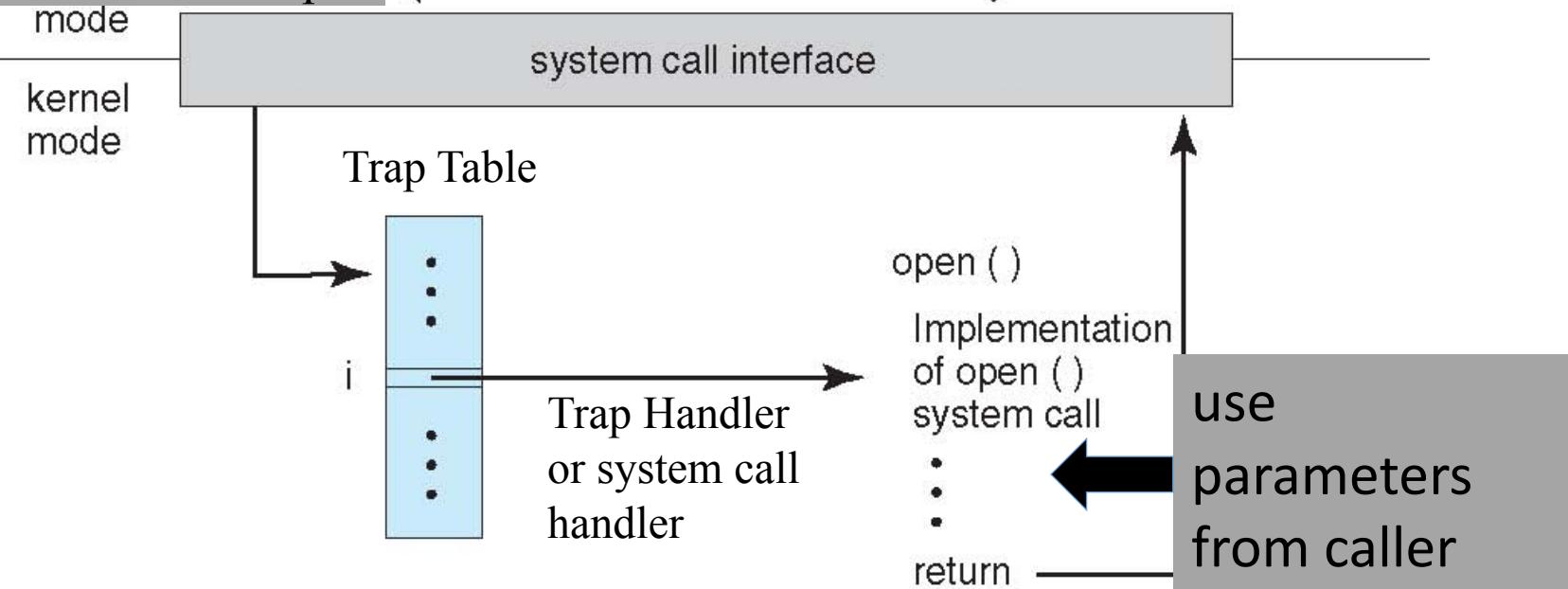
Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
int open(const char * pathname, int flags);		
Manipulation	ReadConsole() WriteConsole()	read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

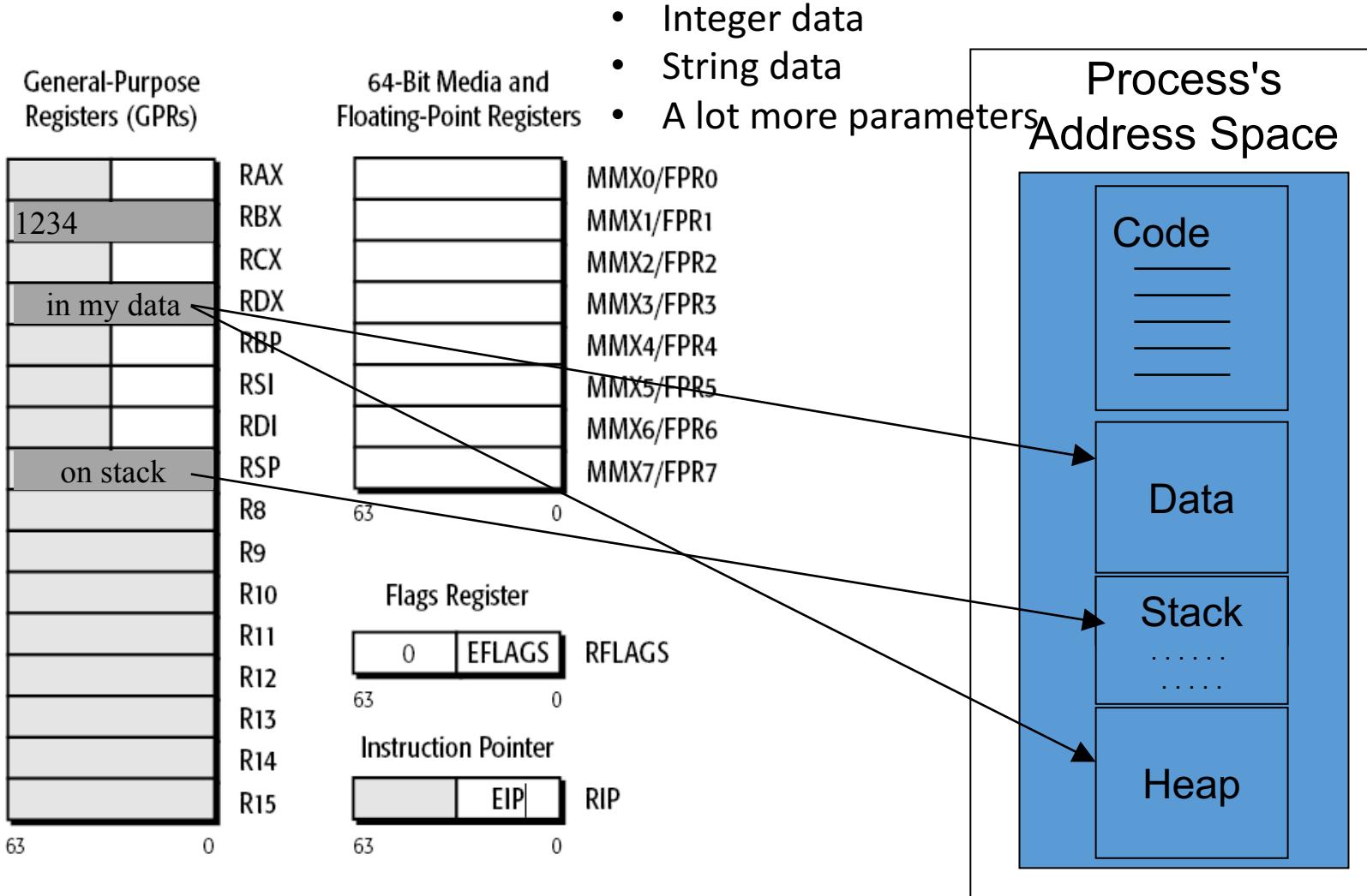
Passing Parameters to System Call

```
open () {  
...  
setup parameters...  
trap N_SYS_OPEN  
...  
}  
}
```

trap == “software interrupt”



Passing Parameters to System Calls



System Call Parameter Passing

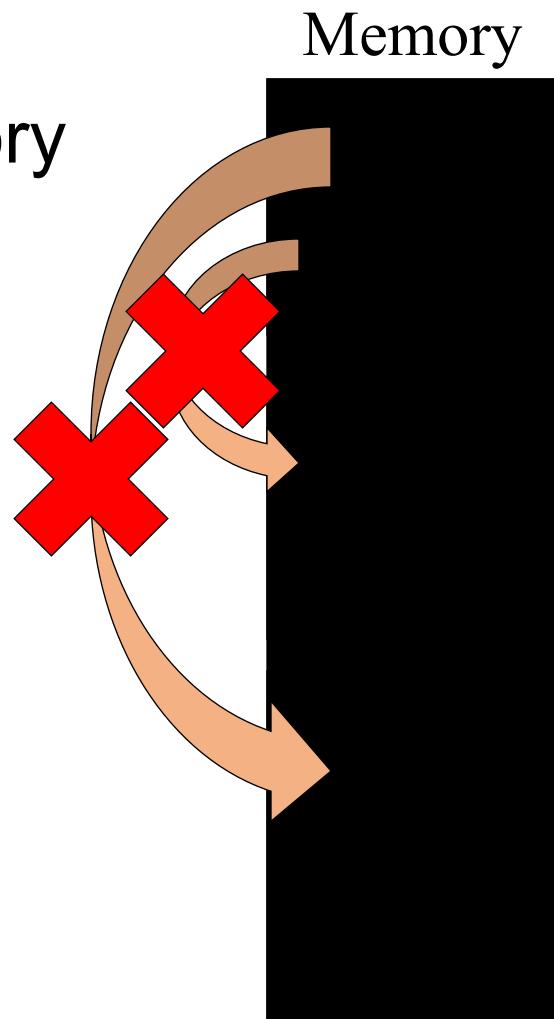
- Three general methods used to pass parameters to the OS
 1. Register: Simplest, pass the parameters in *registers*
 - In some cases, may be more parameters than registers
 2. *Pointer*: Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 3. *Stack*: 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

Protection of Applications

1. The OS can't just access any memory

E.g. App1 tell the OS to access App2's data

2. Need to explicitly ask itself for permission
3. When in the kernel, extra caution is needed to access to data



API for User Space memory access from Kernel Space

Function	Description
access_ok	Checks the validity of the user space memory pointer
get_user	Gets a simple variable from user space
put_user	Puts a simple variable to user space
clear_user	Clears, or zeros, a block in user space
copy_to_user	Copies a block of data from the kernel to user space
copy_from_user	Copies a block of data from user space to the kernel
strnlen_user	Gets the size of a string buffer in user space
strncpy_from_user	Copies a string from user space into the kernel



Loading an OS

How do we get a computer started?

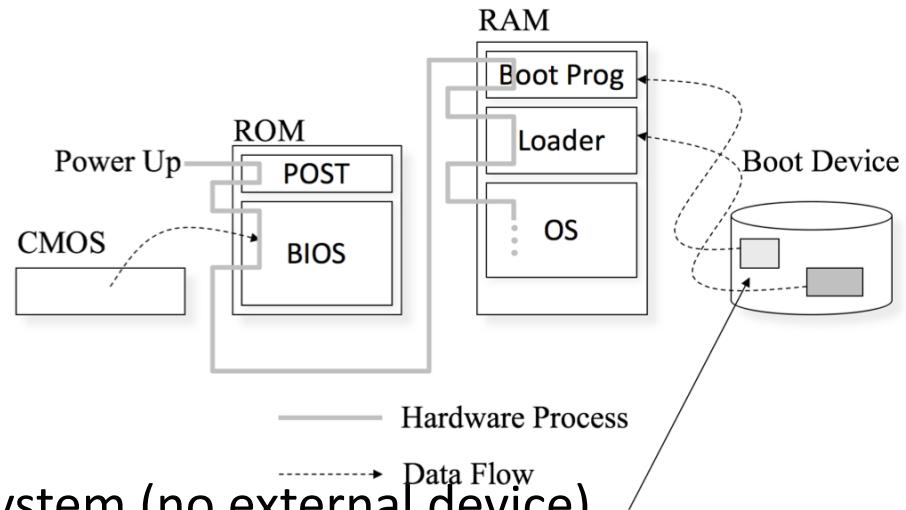
- We have hardware and we give it power, what happens?
 - CPU only knows how to perform its basic operations, load, move, add, ...
 - Computer only does what it is told
 - How do we get it to run the operating system?
 - How does it know where or how to load the first program?



System Boot

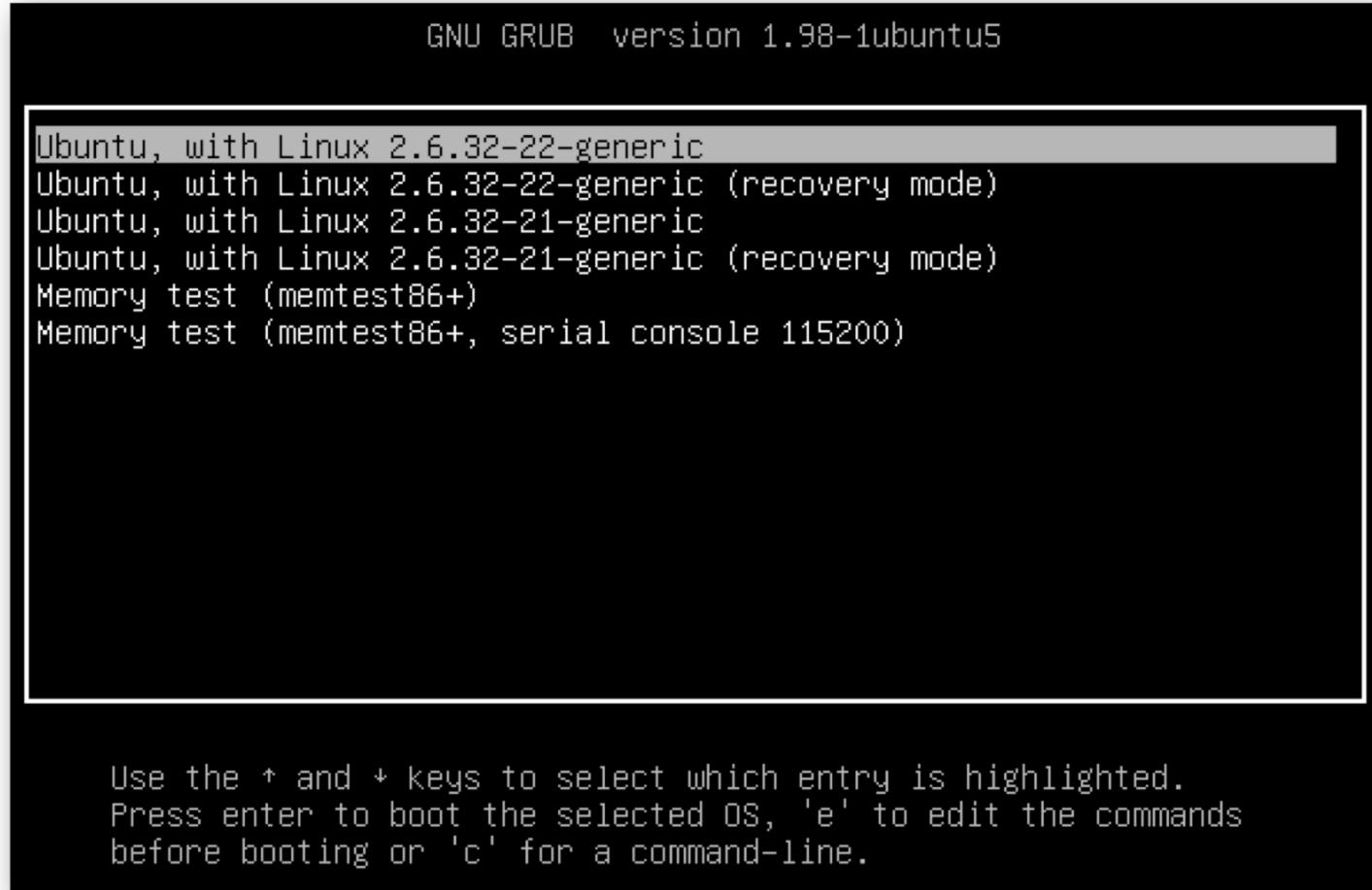
- *Booting* or "*Bootstrapping*" a system is the procedure for loading and starting the operating system
- *Bootstrap* program or *Bootstrap* loader locates the OS kernel, loads it into memory, and starts its execution
- Booting is usually a two step process
 - Load and execute a simple bootstrap loader
 - which fetches a more complex kernel from disk and switches execution to the kernel

System Boot



- **Initial boot loader** must be within the system (no external device)
 - ROM or EPROM
- Steps to Booting
 - run diagnostics to make sure the hardware is working (CPU, Memory)
 - find a device to boot from
 - (disk, flash, dvd, network, ...)
 - load the primitive boot loader and transfer control
 - Primitive boot loader then loads the secondary stage bootloader
 - Examples of this secondary 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

GRUB Boot Loader



What is a Virtual Machine?

- An simulated computer running within a real computer
- The virtual computer runs an operating system that can be different than the host operating
- All the requests to access real hardware are routed to the appropriate host hardware, then virtual operating system or applications don't know they are virtual
- Similar to a person embedded in the Matrix (virtual people)

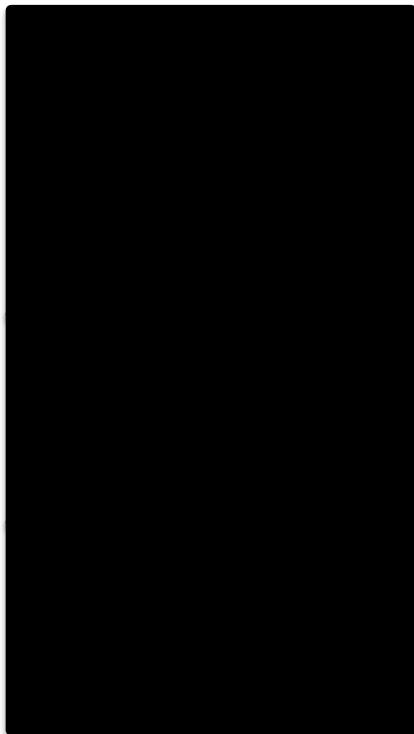
Virtual X concept.

- A process already is given the illusion that it has its
 - Own memory, via virtual memory
 - Own CPU, via time slicing
- Virtual machine extends this idea to give a process the illusion that it also has its own hardware
 - Moreover, extend the concept from a process to an entire OS being given the illusion that it has its own memory, CPU, and I/O devices

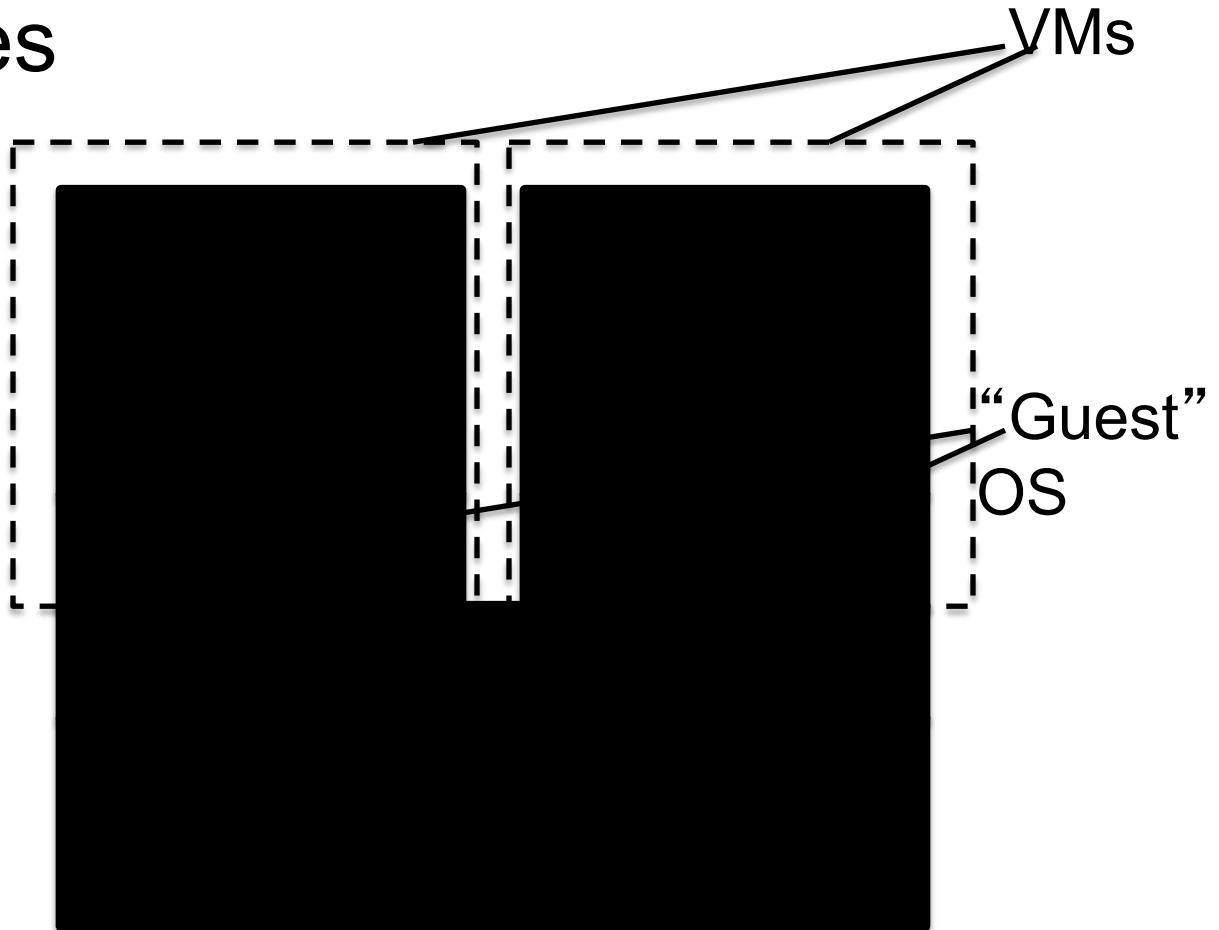
Virtual Machines

- Benefits include:
 - Can run multiple OS's simultaneously on the same host
 - Fault isolation if an OS fails – doesn't crash another VM. This is also useful for debugging a new OS.
 - Easier to deploy applications – can deploy an app within a VM instance that is customized for the app, rather than directly deploying the app itself and worrying about compatibility with the target OS – useful for cloud server deployments

Virtual Machines



Traditional OS



A Type 1 *Hypervisor* provides a virtualization layer for guest OSs and resides just above the hardware.

Virtual Machines

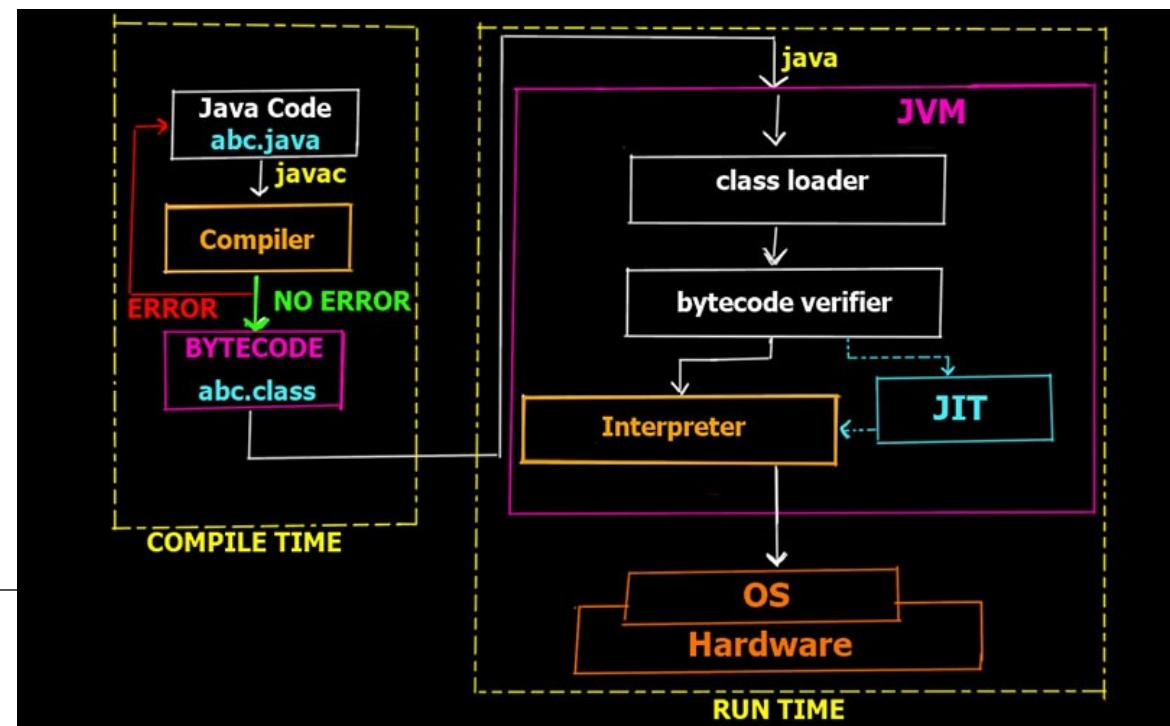
- How it basically works:
 - Goal: want to create a virtual machine that executes at close to native speeds on a CPU, so emulation and interpreting instruction by instruction are not good options – too much software overhead
 - Solution: have the guest OS execute normally, directly on the CPU, except that it is not in kernel mode.
Therefore, any special privileged instructions invoked by the guest OS will be trapped to the hypervisor, which is in kernel mode.
 - The hypervisor then emulates only these privileged instructions and when done passes control back to the guest OS, also known as a “VM entry”
 - This way, most ordinary (non-privileged) instructions operate at full speed, and only privileged instructions incur the overhead of a trap, also known as a “VM exit”, to the hypervisor/VMM.
 - This approach to VMs is called *trap-and-emulate*

Virtual Machines

- Cloud Computing
 - Very easy to provision and deploy VM instances on the cloud
 - E.g. Amazon's Elastic Compute Cloud (EC2) uses Xen virtualization
 - There are different types of VMs or instances that can be deployed:
 - Standard, High-Memory, High-CPU
 - Users can create and reboot their own VMs
 - To store data persistently, need to supplement EC2 with an additional cloud service, e.g. Amazon's Simple Storage Service (S3)

Java Virtual Machines

- Process VMs, e.g. Java VMs
 - Differ from System VMs in that the goal is NOT to try to run multiple OSs on the same host, but to **provide portable code execution** of a single application across different hosts
- Java applications are compiled into Java byte code that can be run on any Java VM
 - Java VM acts as an *interpreter* of byte code, translating each byte code instruction into a local action on the host OS



Java Virtual Machines

- Just in time compilation can be used to speed up execution of Java code
 - Java byte code is compiled at run time into native machine code that is executed directly on the hardware, rather than being interpreted instruction by instruction
- Note Java VMs virtualize an abstract machine, not actual hardware, unlike system VMs
 - i.e. the target machine that Java byte code is being compiled for is a software specification

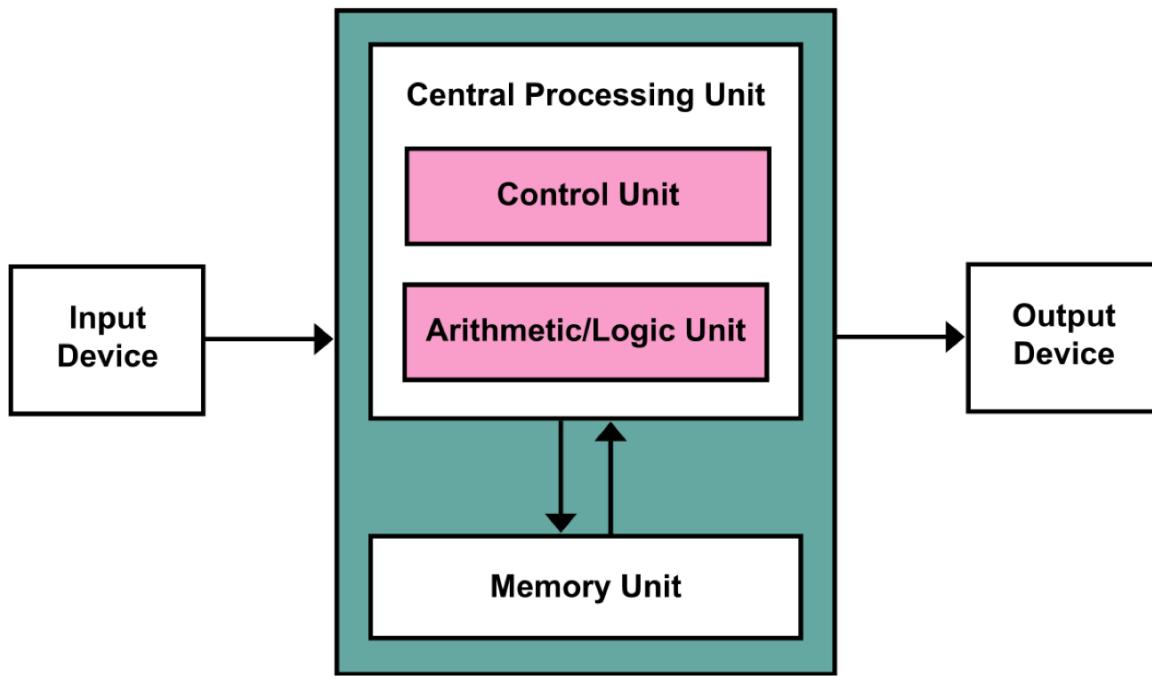
Questions?



Lecture 3

Device Management

Von Neumann Computer Architecture

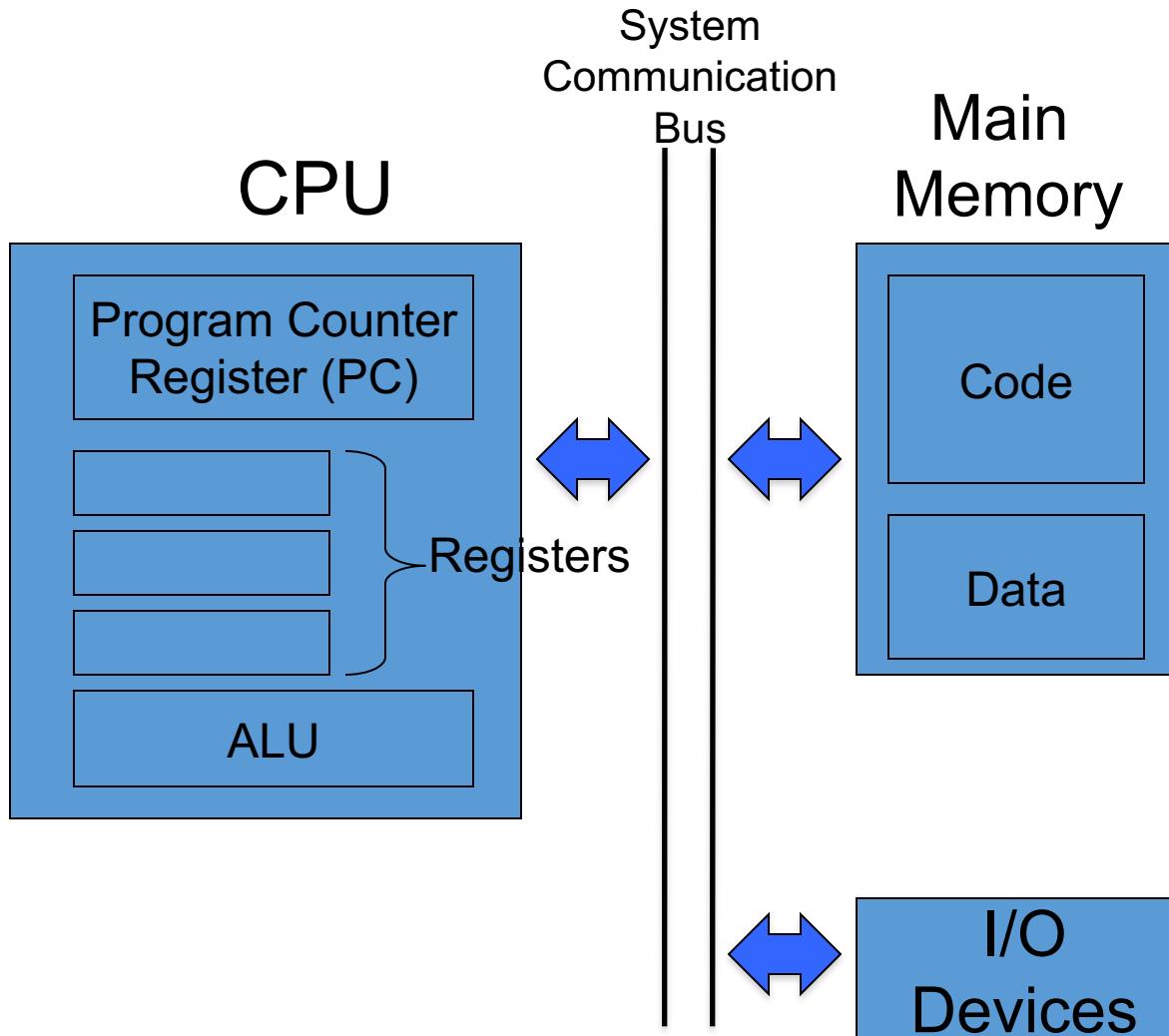


In 1945, von Neumann described a “stored-program” digital computer in which memory stored both instructions **and** data

This simplified loading of new programs and executing them without having to rewire the entire computer each time a new program needed to be loaded



Von Neumann Computer Architecture

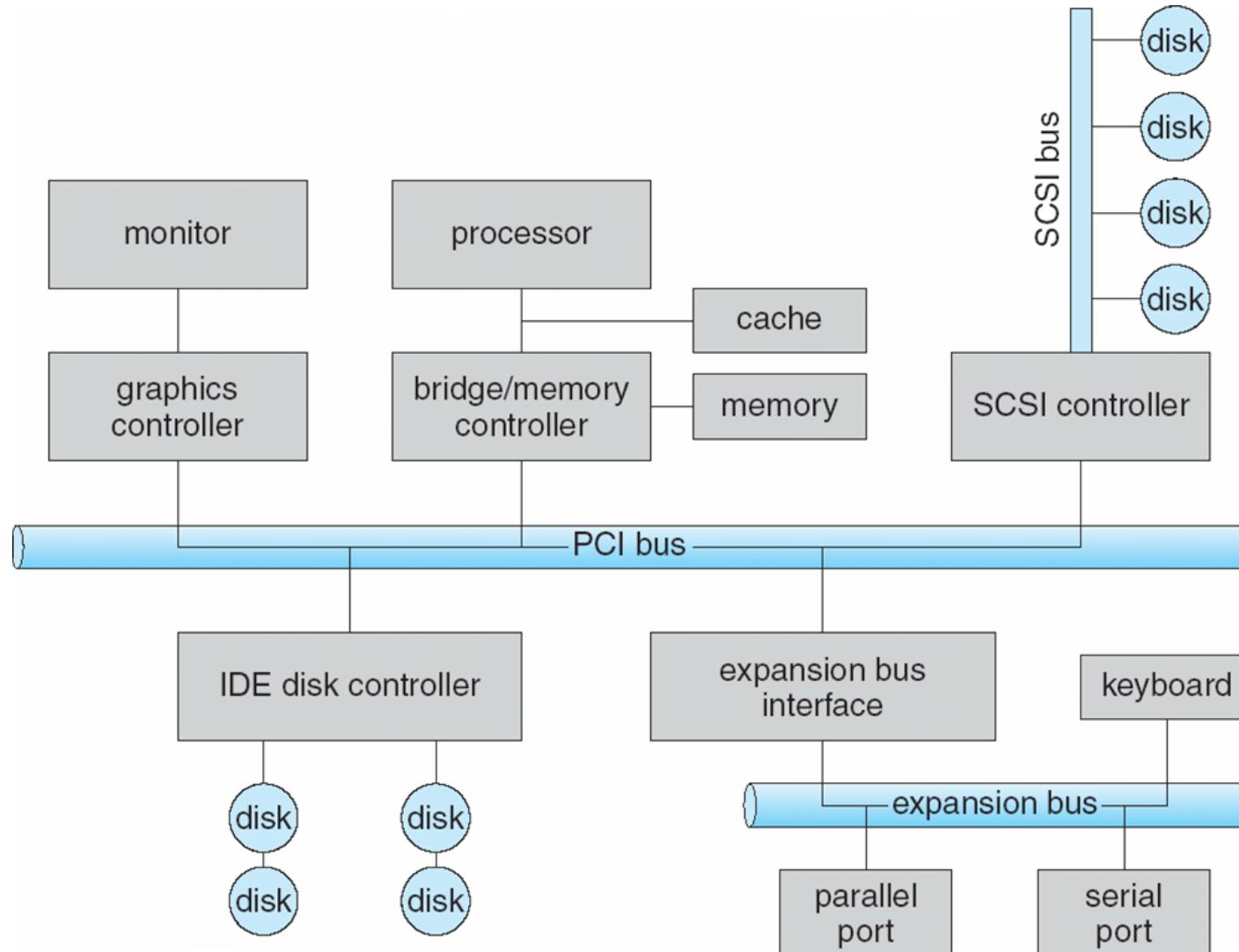


Want to support more devices: card reader, magnetic tape reader, printer, display, disk storage, etc.

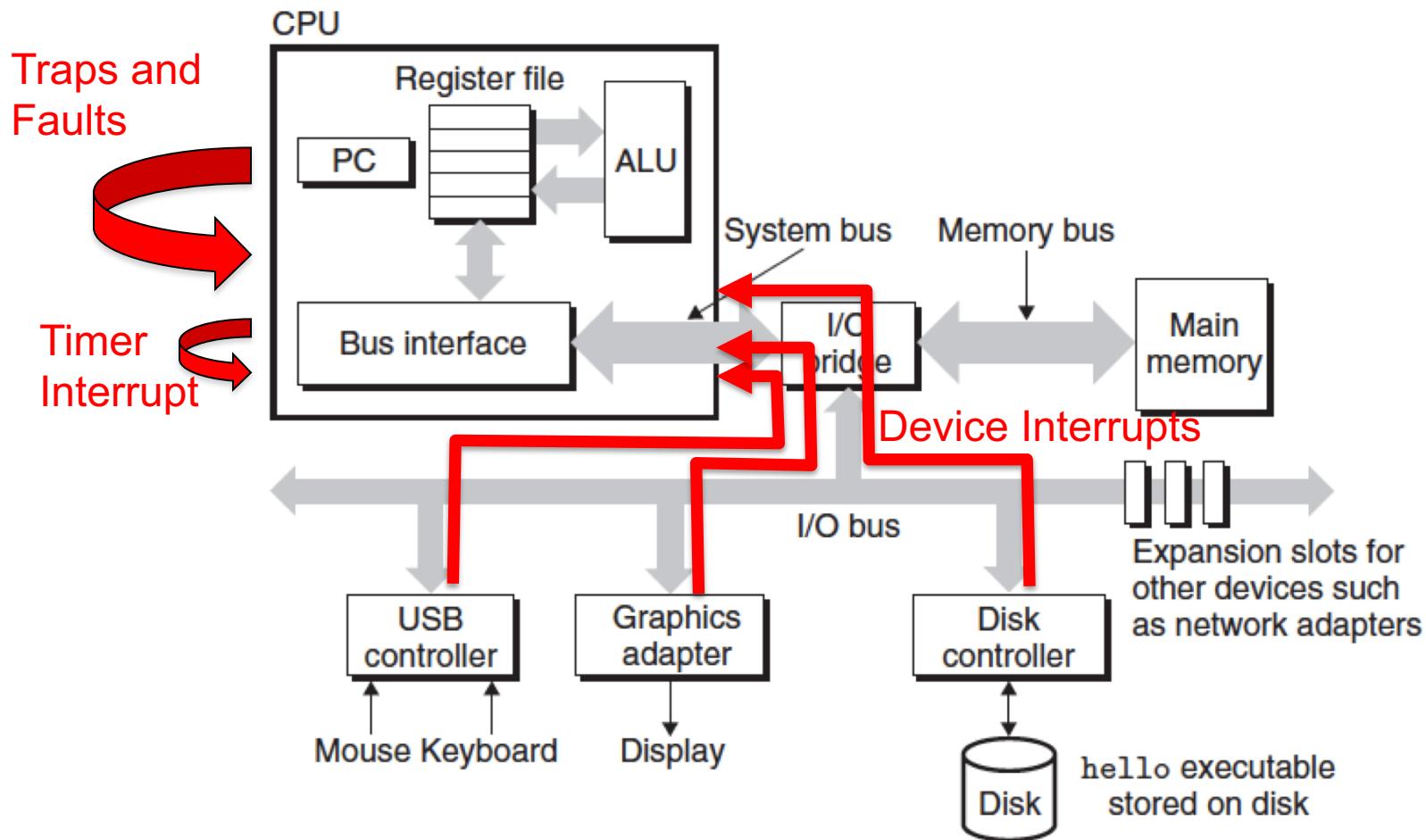
System bus evolved to handle multiple I/O devices.

Includes control, address and data buses

A Typical PC Bus Structure



Modern Computer Architecture: Devices and the I/O Bus



Classes of Exceptions

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, segmentation fault	might return to current instruction, synchronous
(Hardware) Interrupt	signal from I/O device	Disk read finished, packet arrived on network interface card (NIC)	always returns to next instruction, asynchronous
Abort	nonrecoverable error	Hardware bus failure	never returns, synchronous



Examples of x86 Exceptions

Exception Table

Exception Number	Description	Exception Class	Pointer to Handler
0	Divide error	fault	---
13	General protection fault	fault	---
14	Page fault	fault	---
18	machine check	abort	---
32-127	OS-defined	Interrupt or trap	---
128	System call	Trap	---
129-255	OS-defined	Interrupt or trap	---

0-31
reserved
for
hardware

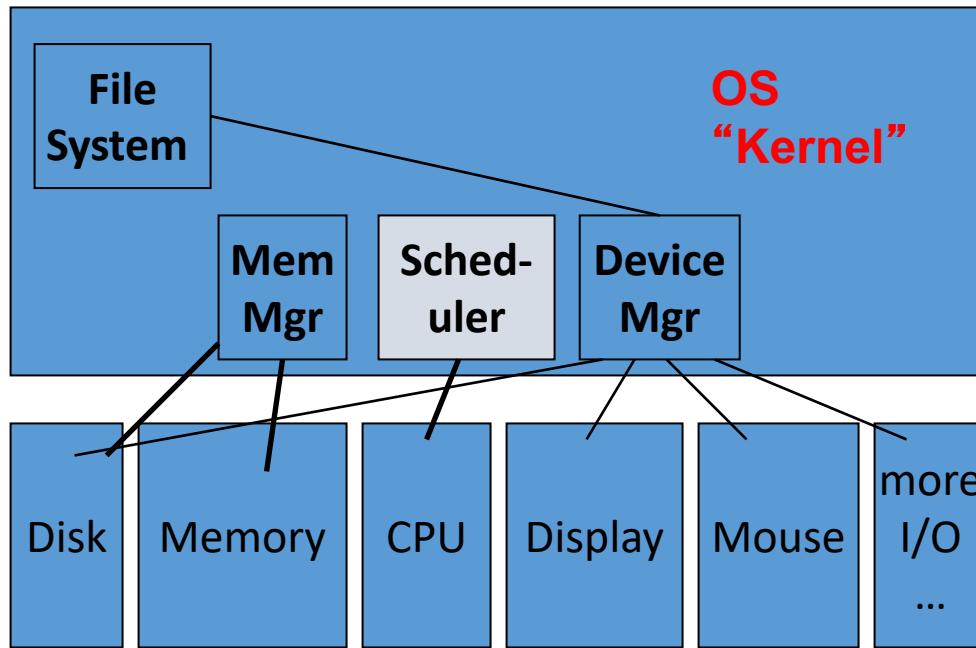
OS
assigns

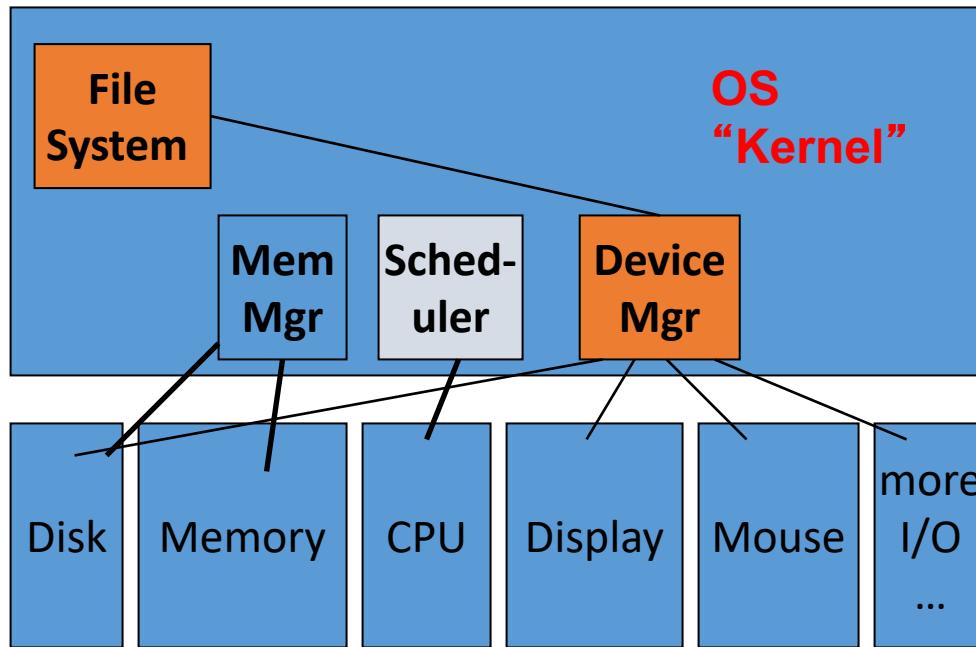
offsets
form
*interrupt
vector*



Examples of x86 Exceptions

- x86 Pentium: Table of 256 different exception types
 - some assigned by CPU designers (divide by zero, memory access violations, page faults)
 - some assigned by OS, e.g. interrupts or traps
- Pentium CPU contains exception table base register that points to this table, so it can be located anywhere in memory



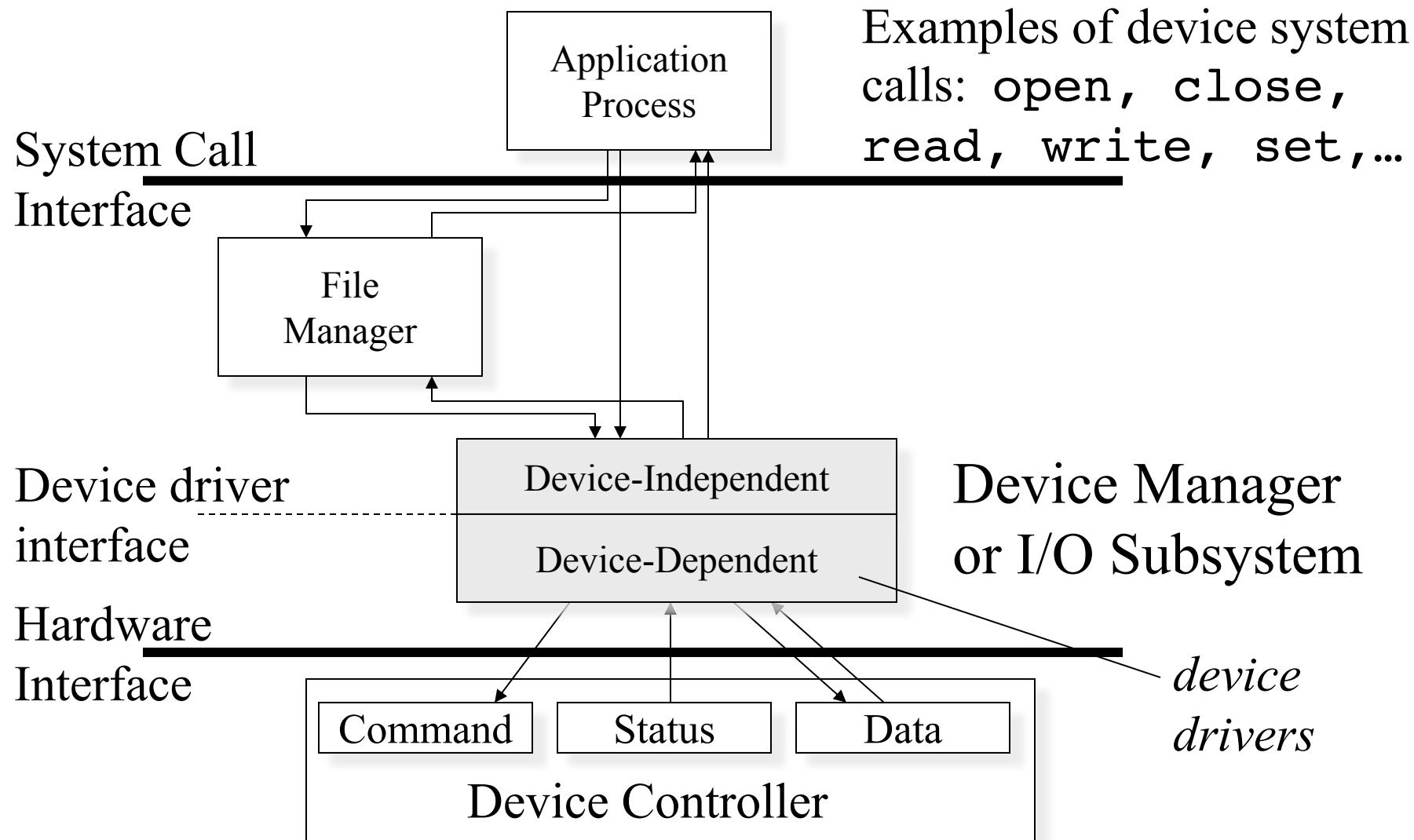


Device Manager

- Controls 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 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
 - Specify how we talk to the device
 - Sequential vs direct/random access
 - Old tape vs. Disk
 - Blocking I/O 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
 - Synchronous versus asynchronous
 - asynchronous returns immediately, but at some later time, the full number of bytes requested is transferred



ioctl and fcntl (input/output control)

- Want a richer interface for managing I/O devices than just open, close, read, write, ...
- ioctl allows a user-space application to configure parameters and/or actions of an I/O device
 - e.g set the speed of a device, or eject a disk
- Usage: *int ioctl(int fd, int cmd, ...);*
 - Invokes a system call to execute **device-specific cmd** on I/O device *fd*
 - Used for I/O operations and other operations which cannot be expressed by regular system calls
 - Requests are directed to the correct device driver



ioctl and fcntl (input/output control)

- Avoids having to create new system calls for each new device and/or unforeseen device function
 - Helps make the OS/kernel extensible
- UNIX, Linux, MacOS X all support ioctl, and Windows has its own version
- In UNIX, each device is modeled as a file
 - *fcntl* for file control is related to ioctl and is used for configuring file parameters, hence in many cases I/O communication
 - e.g. use fcntl to set a network socket to non-blocking
 - part of POSIX API, so portable across platform



Device Characteristics

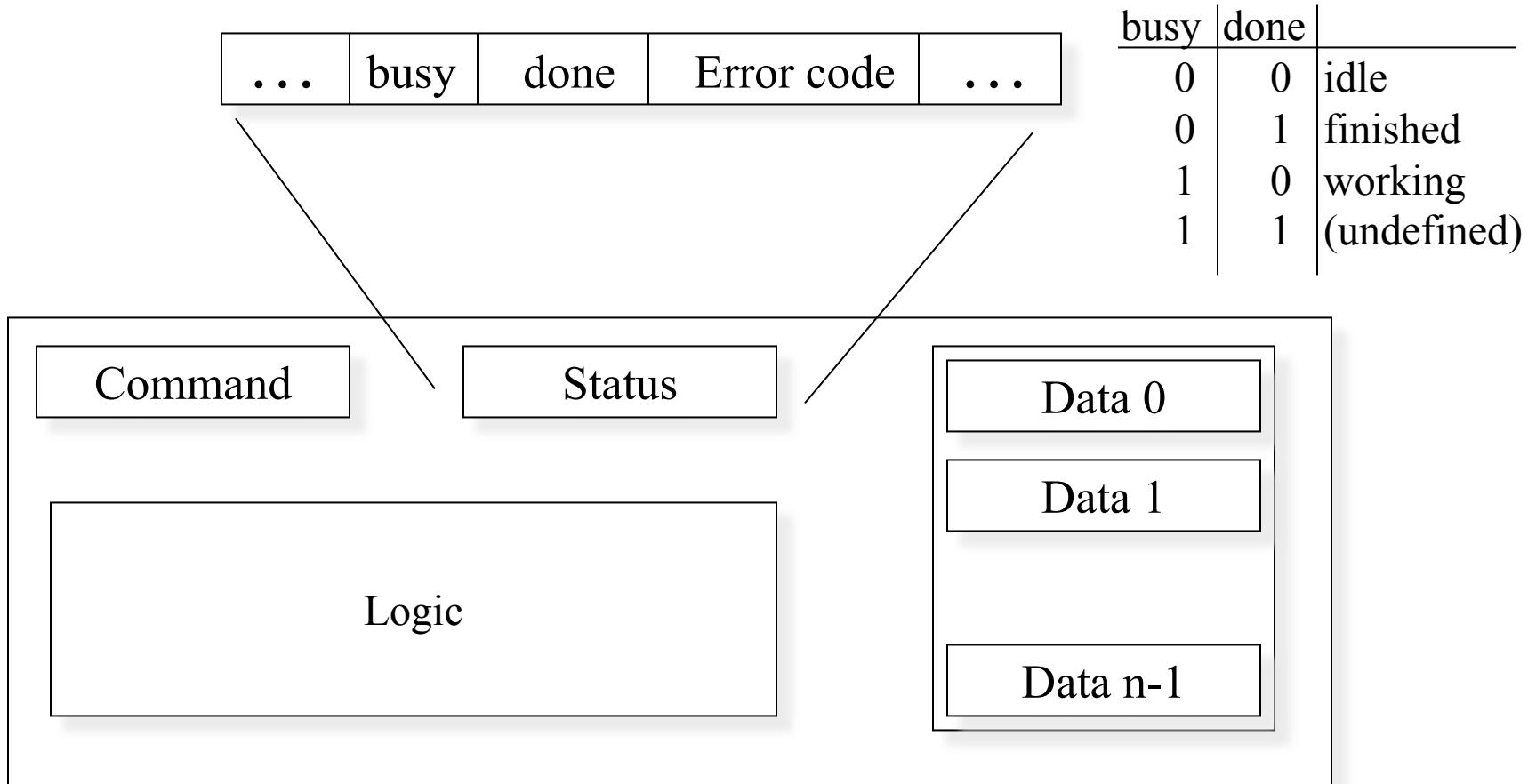
- I/O devices consist of two high-level components
 - Mechanical components
 - Electronic components:
The device is operated by Device controllers
- OS deals with device controllers
 - Through device driver

Device Drivers

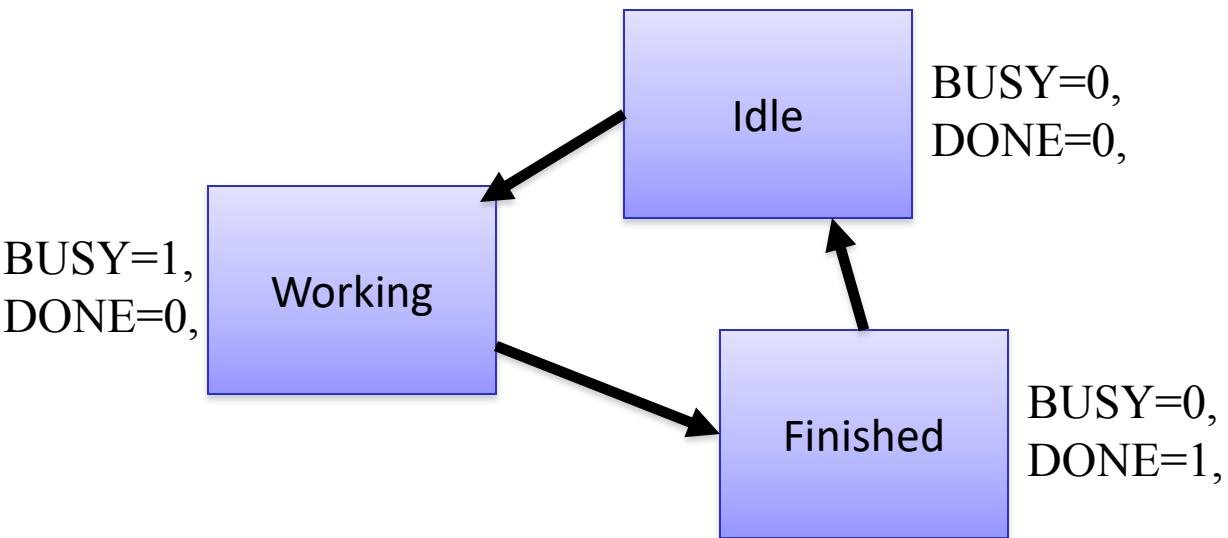
- Support the device system call interface functions open, read, write, etc. for that device
- Interact directly with the device controllers
 - Know the details of what commands the device can handle, how to set/get bits in device controller registers, etc.
 - Are part of the device-dependent component 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 Interface



Device Controller States

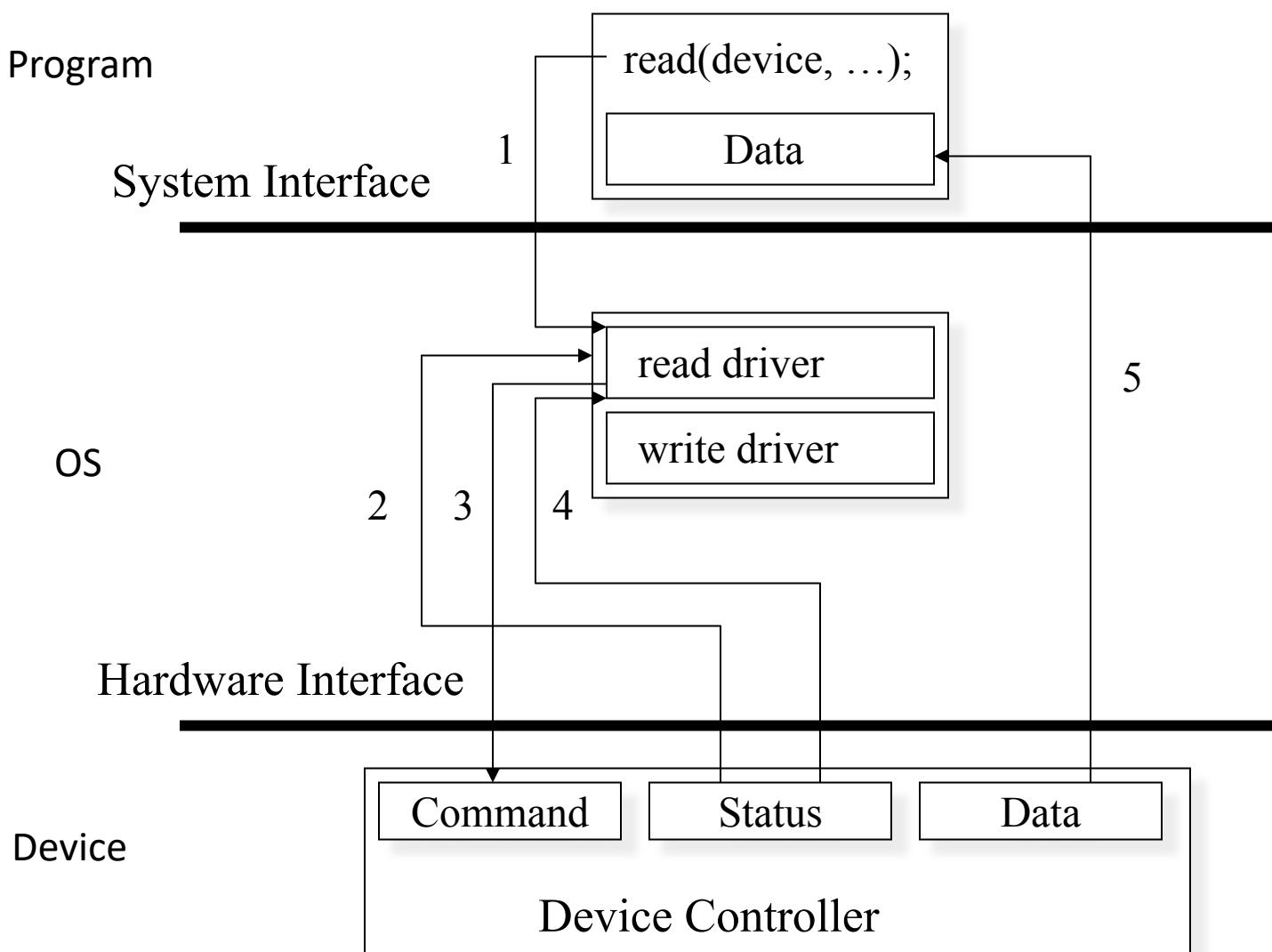


- Need three states to distinguish the following:
 - Idle: no app is accessing the device
 - Working: one app only is accessing the device
 - Finished: the results are ready for that one app

- Therefore, need 2 bits for 3 states:
 - A BUSY flag and a DONE flag
 - $\text{BUSY}=0, \text{DONE}=0 \Rightarrow \text{Idle}$
 - $\text{BUSY}=1, \text{DONE}=0 \Rightarrow \text{Working}$
 - $\text{BUSY}=0, \text{DONE}=1 \Rightarrow \text{Finished}$
 - $\text{BUSY}=1, \text{DONE}=1 \Rightarrow \text{Undefined}$



Example: Polling I/O Read Operation



University of Colorado
Boulder

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;		
	1	0
while (deviceN.busy) <waiting>;		
	0	1
/* finished, so read some status bits... */		
deviceN.done = FALSE;		
	0	0

Polling I/O – Problem

- 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
 - Busy waiting: 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



Lecture 4

Device Strategies

Polling I/O – Problem

- 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
 - Busy waiting: 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

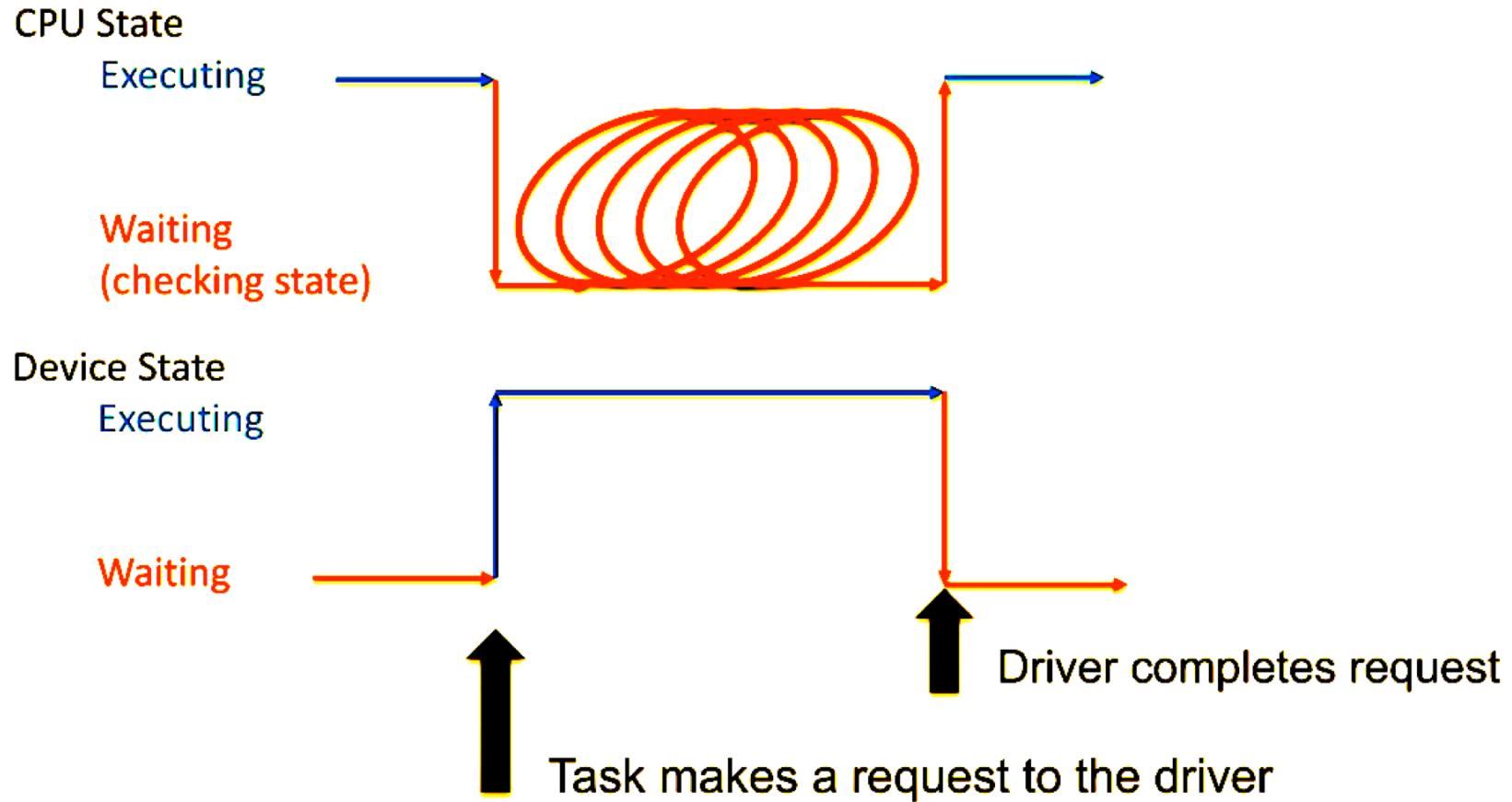


Device Manager I/O Strategies

- 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



Direct I/O with Polling



Direct I/O with Interrupt

CPU State

Executing

Waiting
(checking state)

Device State

Executing

Waiting

CPU checks the state of the device and transfers data

Task makes a request to the driver,
CPU returns to executing instructions

Driver completes request,
Driver interrupts the CPU



University of Colorado
Boulder

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



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

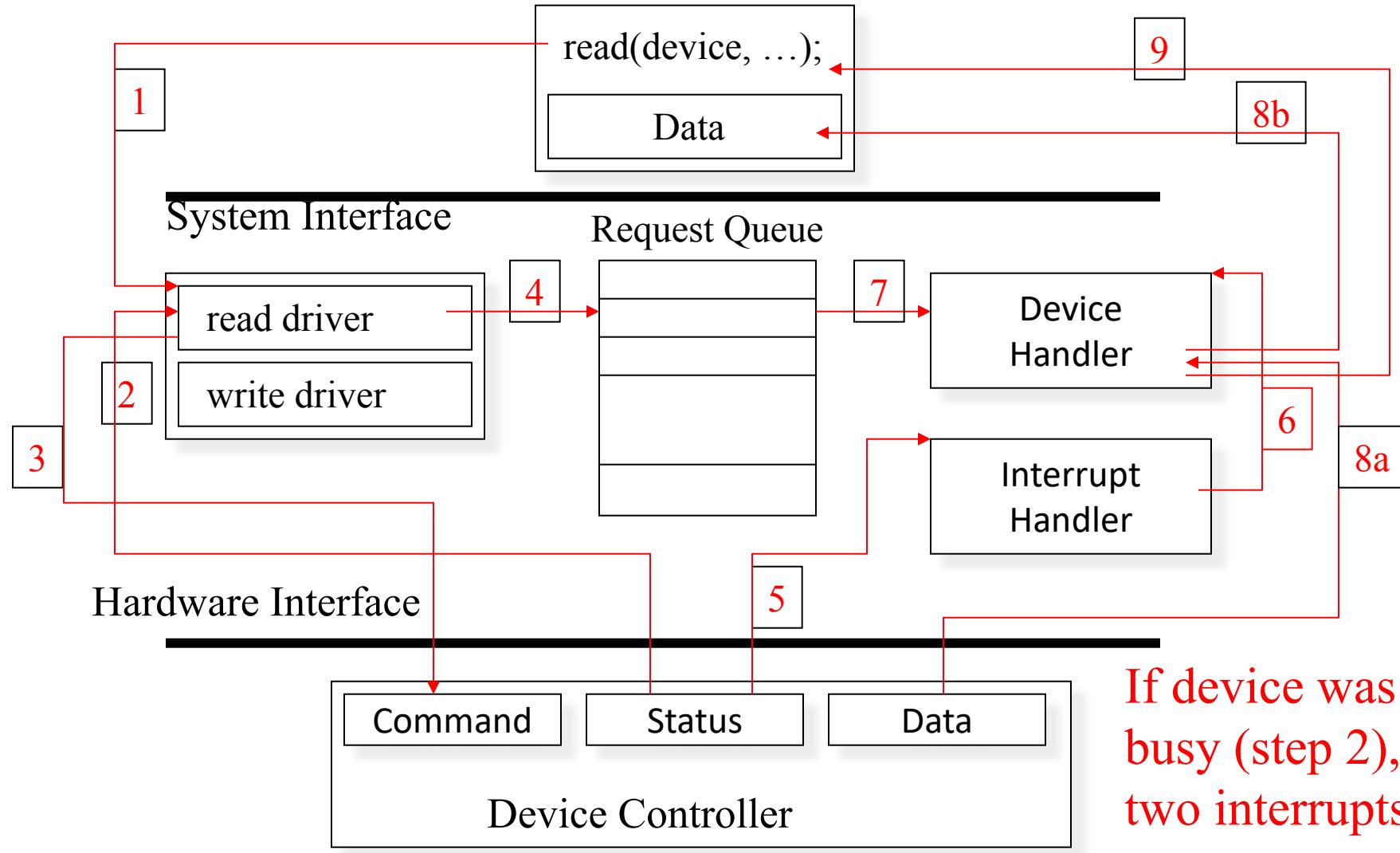


Interrupt Handler

- Prep: Disable interrupts
- 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



When is Polling BETTER than Interrupt handling?

- Setting up the interrupts takes overhead
 - Handling the interrupts takes overhead
 - Handling the scheduling of the processes takes overhead
-
- If it is always a short wait for the IO then Polling is better
 - If the wait is predictable then Polling is better

Problem with Interrupt driven I/O

- Data transfer from disk can become a bottleneck if there is a lot of I/O copying data back and forth between memory and devices
 - Example: read a 1 MB file from disk into memory

The disk is only capable of delivering 1 KB blocks
So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU
This slows down execution of normal programs and the OS
 - Worst case: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card



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
 - direct I/O with *interrupts*
 - More efficient than busy waiting, but still has overhead for every transfer
 - DMA with *interrupts*



Direct Memory Access (DMA)

- Idea: 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
- Modern systems offload some of this work to a special-purpose processor, **Direct-Memory-Access (DMA) controller**
- The DMA controller operates the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU



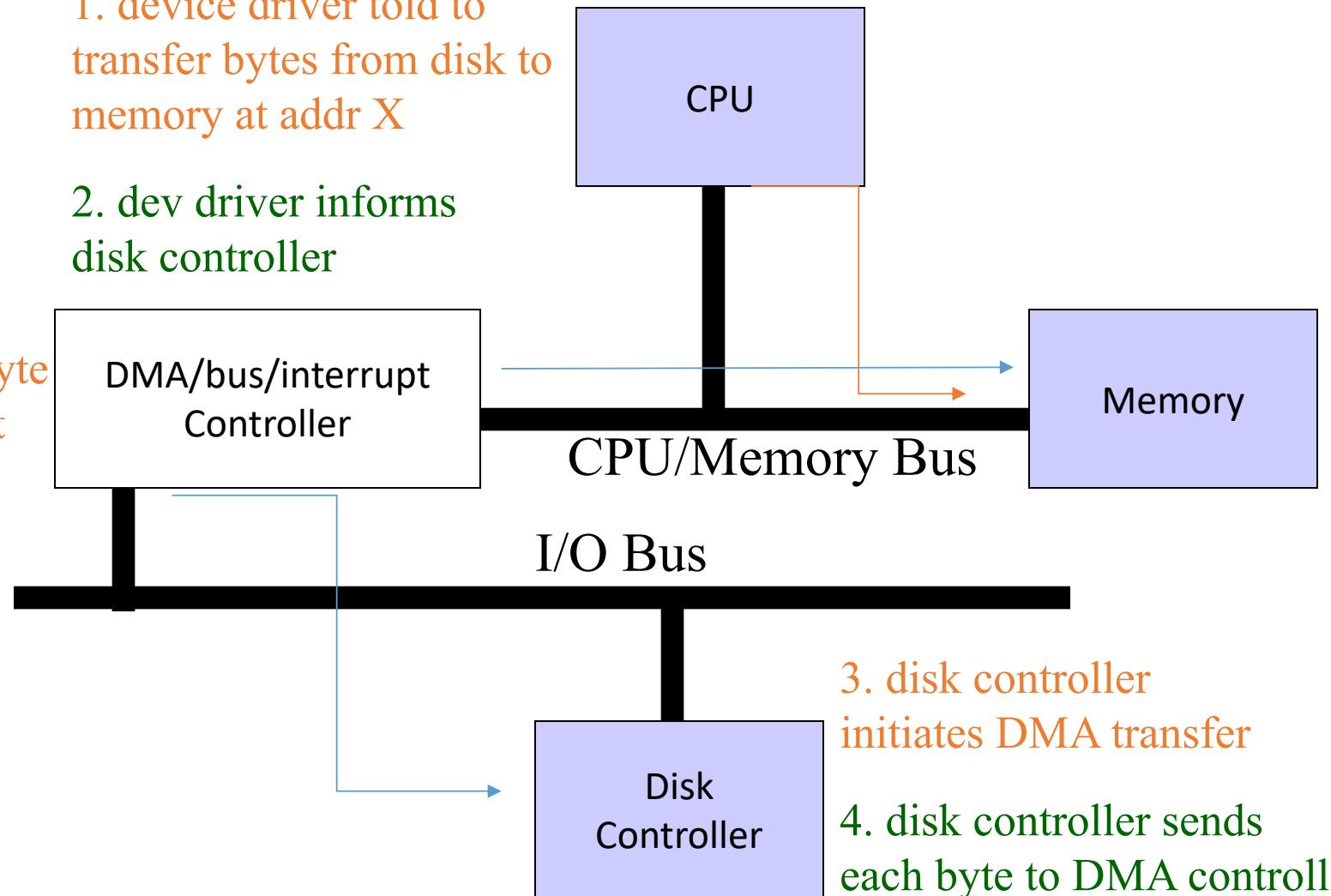
DMA with Interrupts Example

1. device driver told to transfer bytes from disk to memory at addr X

2. dev driver informs disk controller

5. DMA sends each byte to memory at addr X

3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller

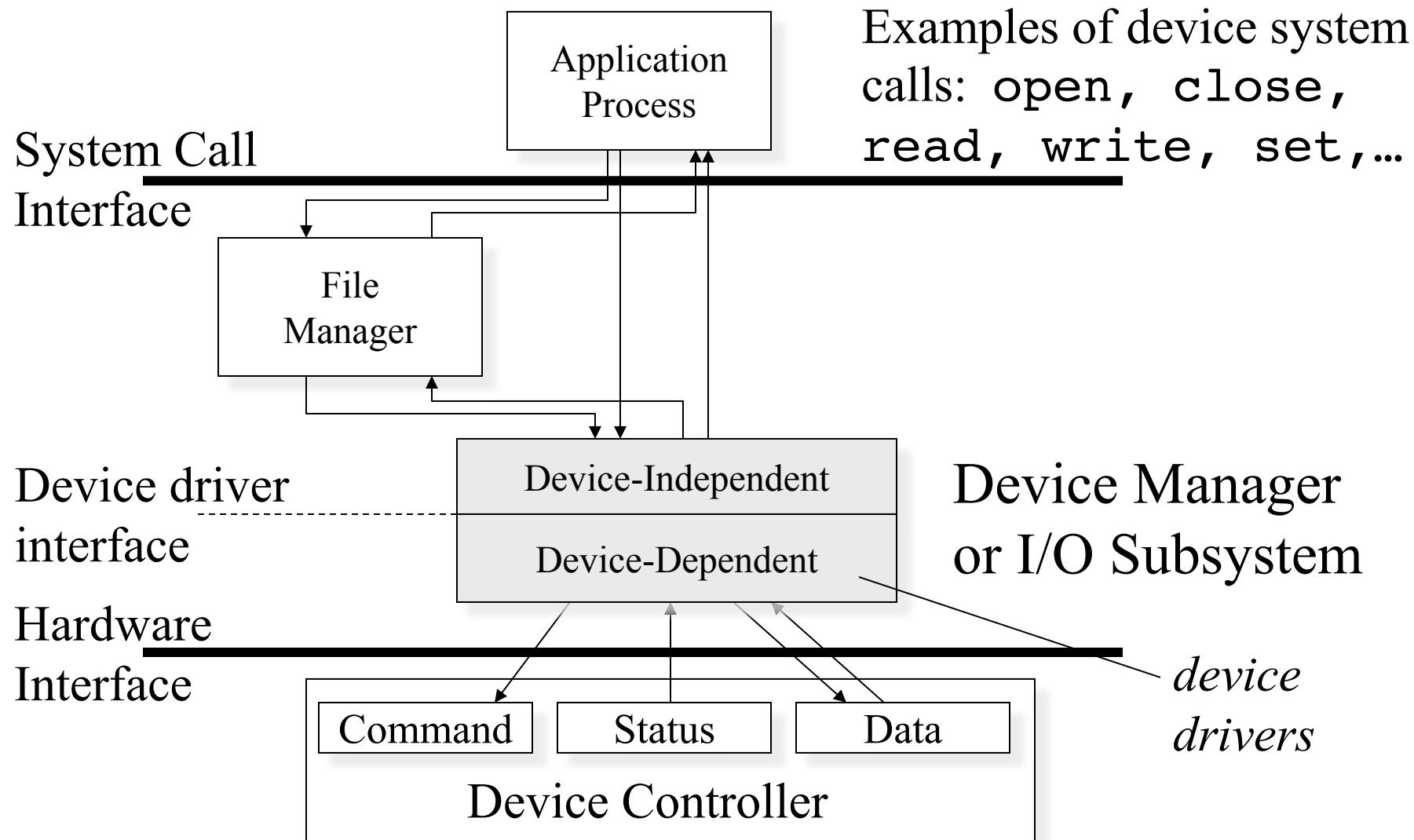


Direct Memory Access (DMA)

- 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



Device Management Organization



Port-Mapped I/O

- Port or port-mapped (non-memory 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.

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)



Port-Mapped I/O

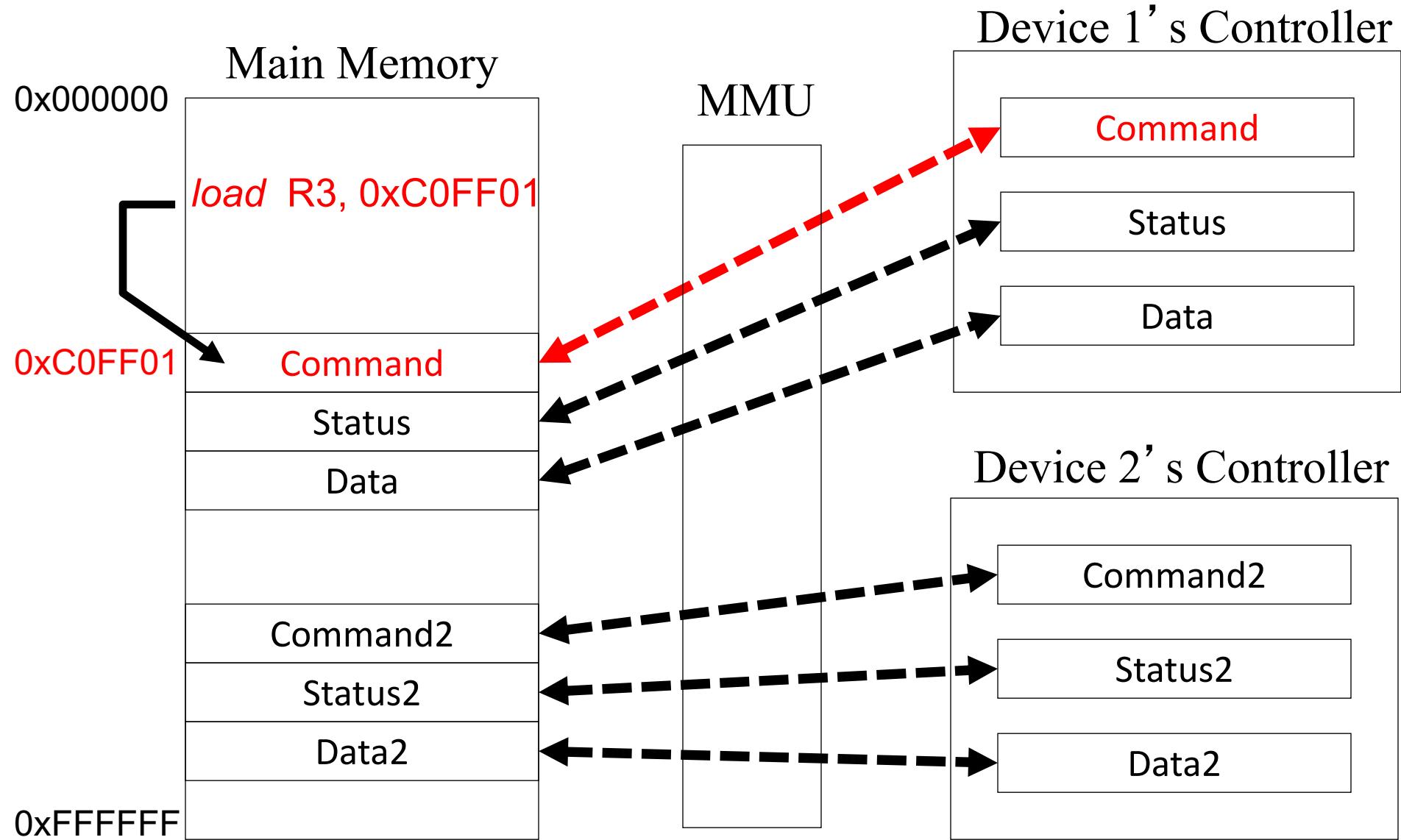
- Port-mapped I/O is quite limited
 - IN and OUT can only store and load
 - don't have full range of memory operations for normal CPU instructions
 - Example: to increment the value in say a device's data register, have to copy register value into memory, add one, and copy it back to device register.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

Memory-Mapped I/O

- Memory-mapped I/O: device registers and device memory are mapped to the system address space (system's memory)
- 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 an I/O device's register
- 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 to reflect on the device



Memory-Mapped I/O



Memory-Mapped I/O

- 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 10-100 MB of memory, which in modern address spaces of GBs is quite modest
 - so memory-mapped I/O is a small penalty

What is difference between Port and Memory Mapped IO?

- **Port mapped I/O** uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.
- **Memory mapped I/O** is **mapped** into the same address space as program **memory** and/or user **memory**, and is accessed in the normal way.

Recap ...

- What are the three device controller states?
 - Idle, Working, Busy
- What are the three I/O strategies
 - Direct I/O with polling
 - CPU first waits for device to become idle
 - CPU issue I/O command
 - CPU waits for device to complete
 - Direct I/O with interrupts
 - No busy waiting
 - DMA with interrupts
 - large data transfer without using CPU
- What are the differences between Port and Memory-Mapped IO?
 - Only OS can access port registers at specific memory locations
 - Memory for device registers is mapped into user or kernel space and accessed in the same manner as any other memory





Lecture 5

Loadable Kernel Modules

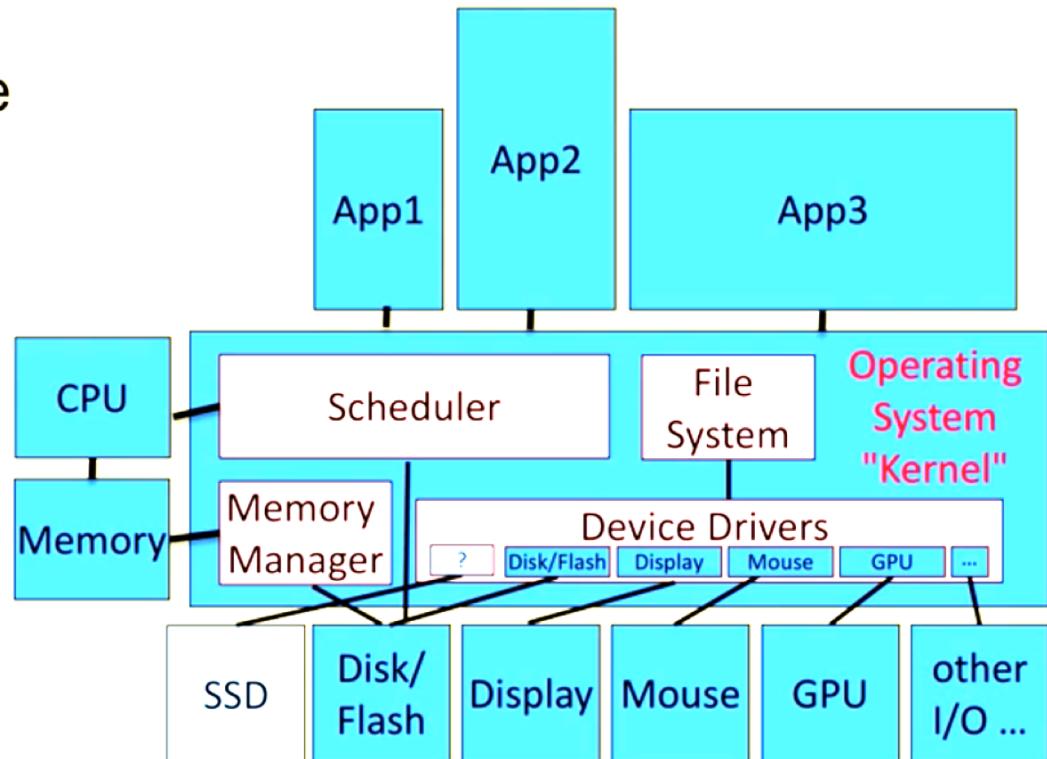
Device have both device-independent and device-dependent code

There is special device driver code associated with each different device connected to the system

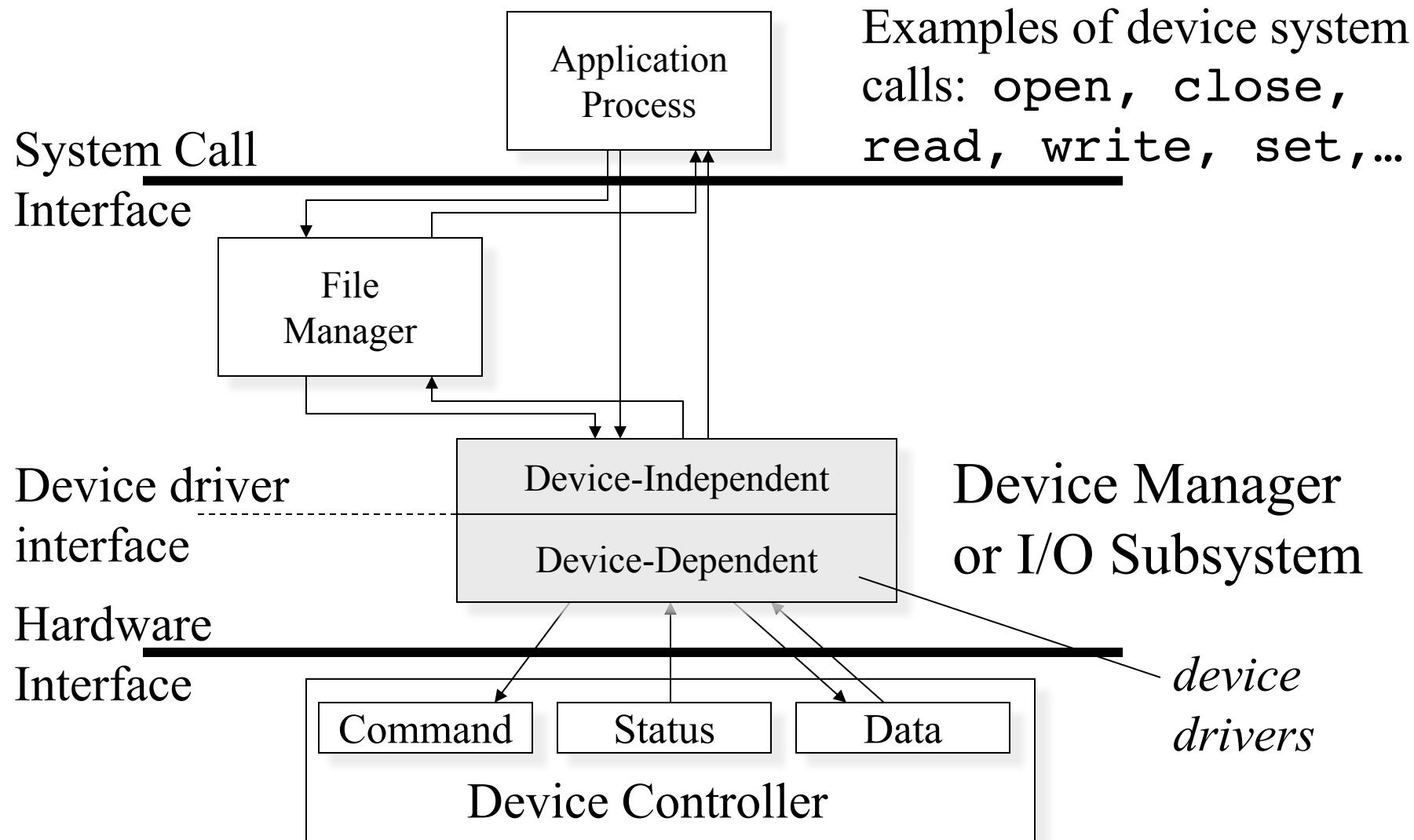
1. Device-Independent API

2. Device-Dependent driver code

3. Device Controller



Device Management Organization

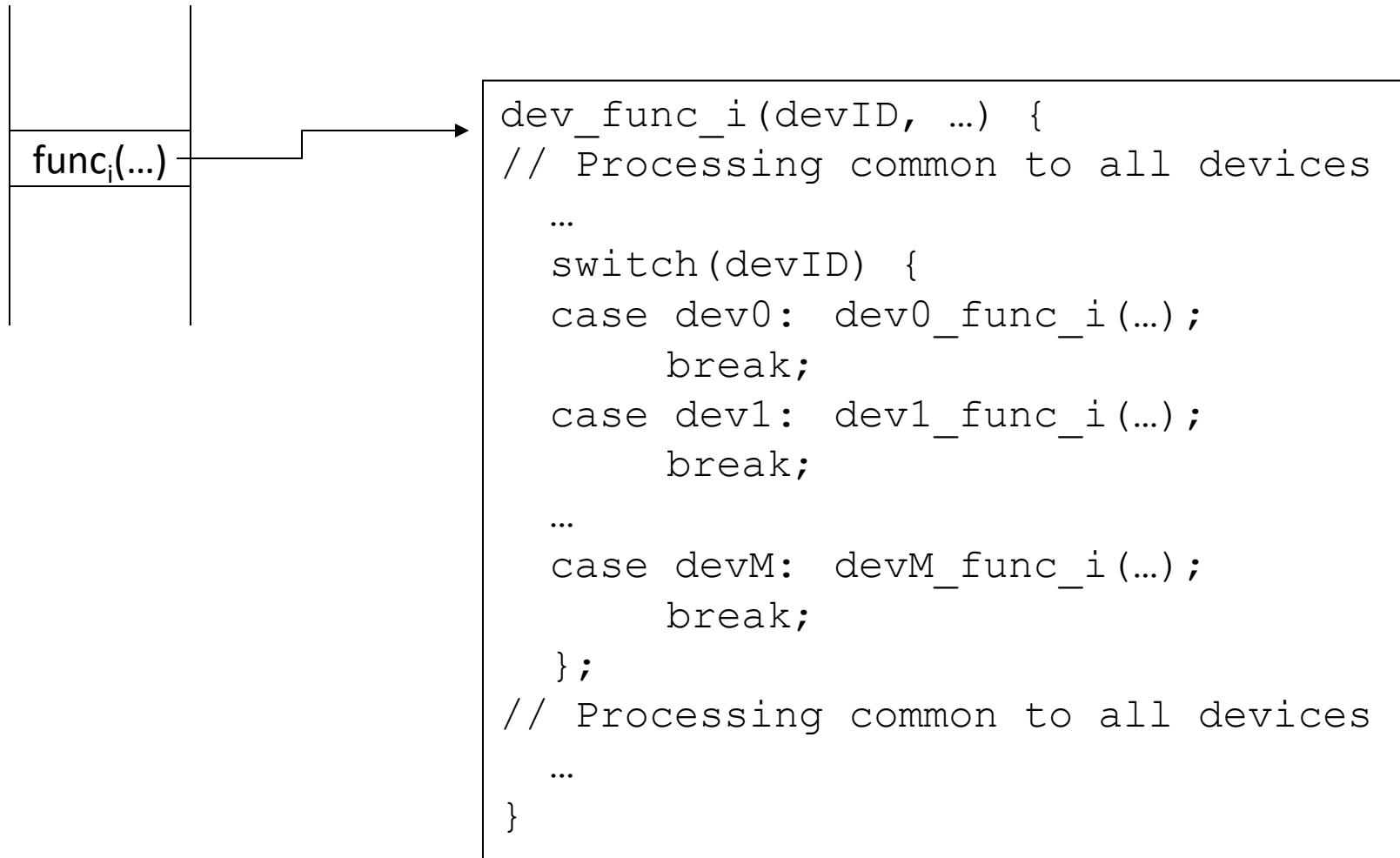


Device Independent Part

- 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

Trap Table

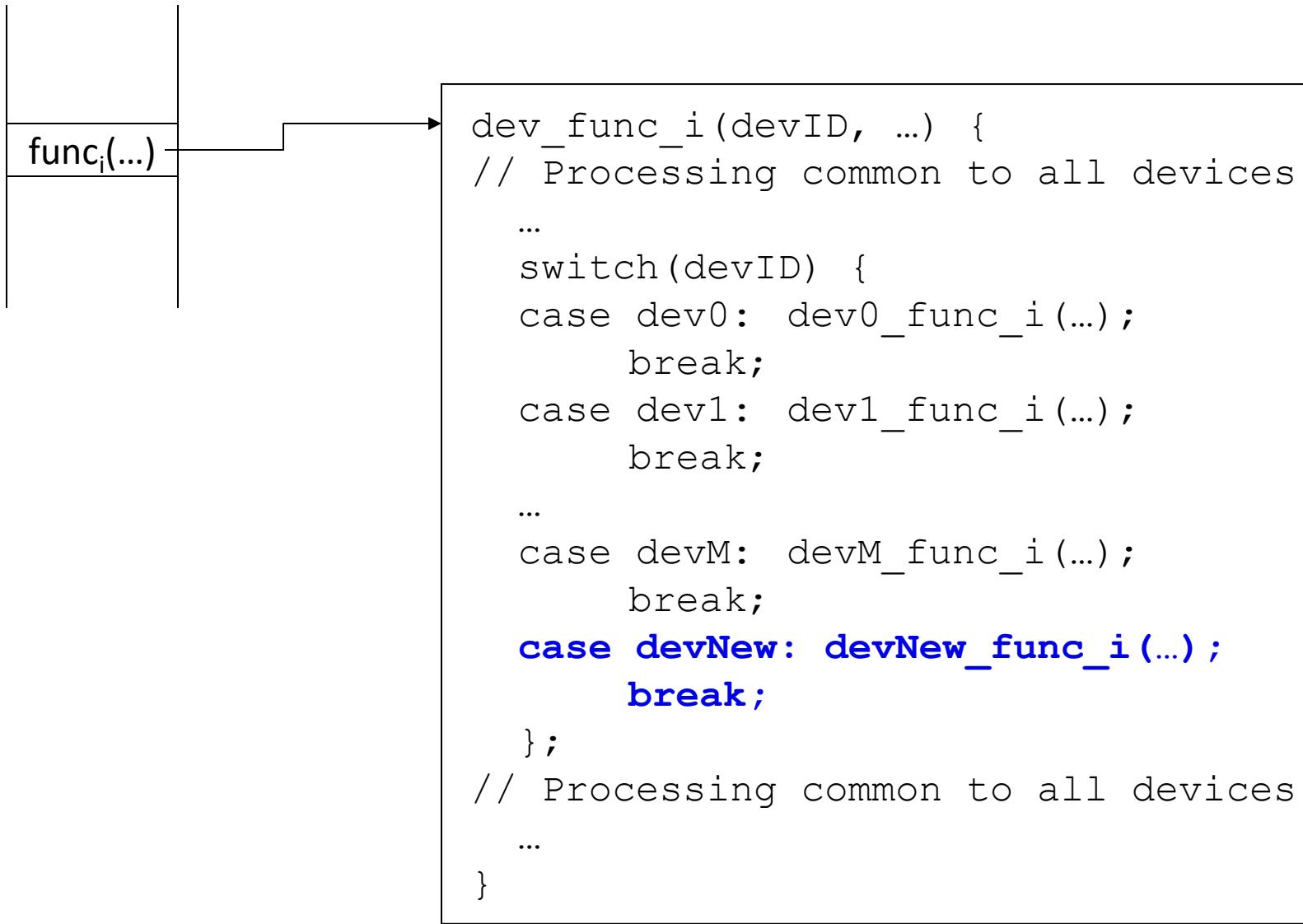


Adding a New Device

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



Trap Table



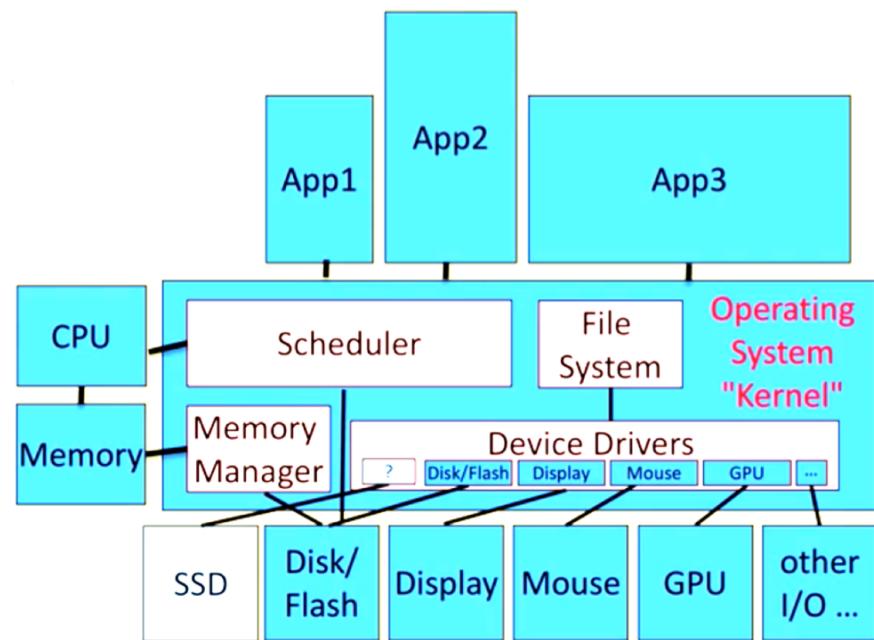
Adding a New Device

- After updating all `dev_func_*`(...) in the kernel, compile the kernel

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

Device have both device-independent and device-dependent code

- Need a way to **Dynamically add** new code into the OS kernel when new device needs to be supported
- Load *Device-Dependent* driver code into kernel
- Only load the device drivers as needed
- No kernel recompilation for changes in the device driver



Loadable Kernel Modules

- LKM is an object file that contains code to extend a running kernel
- Windows (kernel-mode driver), Linux (LKM), OS X (Kernel extension: kext), VmWorks
- LKMs can be loaded and unloaded from kernel on demand at runtime

LKMs

- Offer an easy way to extend the functionality of the kernel without having to rebuild or recompile the kernel again
- Simple and efficient way to create programs that reside in the kernel and run in privileged mode
- Most of the drivers are written as LKMs
 - What's out there in the kernel? See `/lib/modules` for all the LKMs
 - `lsmod`: lists all kernel modules that are already loaded



How to write a kernel module?

- Kernel Modules are written in the C programming language.
- You must have a Linux kernel source tree to build your module.
- You must be running the **same kernel version** you built your module with to run it.
- Linux kernel object: **.ko** extension

Kernel Module: Basics

- A kernel module file has several typical components:
 - MODULE_AUTHOR("your name")
 - MODULE_LICENSE("GPL")
 - The license must be an open source license (GPL, BSD, etc.) or you will “taint” your kernel.
 - Tainted kernel loses many abilities to run other open source modules and capabilities.

Kernel Module: Key Operations

- `int init_module(void)`
 - Called when the kernel loads your module.
 - Initialize all your stuff here.
 - Return 0 if all went well, negative if something blew up.
- Typically, `init_module()` either
 - registers a handler for something with the kernel,
 - or replaces one of the kernel functions with its own code (usually code to do something and then call the original function)



Kernel Module: Key Operations

- `void cleanup_module(void)`
 - Called when the kernel unloads your module.
 - Free all your resources here.

Hello World Example

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_AUTHOR("Awesome Developer");
MODULE_LICENSE("GPL");

int init_module(void)
{
    printk(KERN_ALERT "Hello world: I am a developer in CS3753
                      speaking from the Kernel");
    return 0;
}
```

Hello World Example

```
void cleanup_module(void)
{
    printk(KERN_ALERT "Goodbye from a developer in CS3753,
                    I am exiting the Kernel");
}
```

Building Your Kernel Module

- Accompany your module with a 1-line GNU Makefile:
 - obj-m += hello.o
 - Assumes file name is “hello.c”
- Run the magic make command:
 - make -C <kernel-src> M=`pwd` modules
 - Produces: hello.ko
- Assumes current directory is the module source.



`obj-$(CONFIG_FOO) += foo.o`

- Good definitions are the main part (heart) of the kbuild Makefile.
- The most simple kbuild makefile contains one line:
`obj-$(CONFIG_FOO) += foo.o`
- This tell **kbuild** that there is one object in that directory named `foo.o`. `foo.o` will be built from `foo.c` or `foo.S`.
- `$(CONFIG_FOO)` evaluates to either `y` (for built-in) or `m` (for module). If `CONFIG_FOO` is neither `y` nor `m`, then the file will not be compiled nor linked.



Loading Your Kernel Module: *insmod*

- Use *insmod* to manually load your kernel module
`sudo insmod helloworld.ko`
- *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



Where is our Hello World message

- Dmesg
- `/var/log/system.log`

Unloading Your Kernel Module

- Use *rmmod* command

```
rmmod hello.ko
```

- Should print the Goodbye message

Kernel Module Dependencies: *modprobe*

- *insmod/rmmod* can be cumbersome...
 - You must manually enforce inter-module dependencies.
- *modprobe* automatically manages dependent modules
 - Copy hello.ko into /lib/modules/<version>
 - Run depmod
 - modprobe hello / modprobe -r hello
- Dependent modules are automatically loaded/unloaded.



- *depmod* creates a Makefile-like dependency file, based on the symbols it finds in the set of modules mentioned on the command line or from the directories specified in the configuration file
- This dependency file is later used by *modprobe* to automatically load the correct module or stack of modules

modinfo command

- .ko files contain an additional .modinfo section where additional information about the module is kept
 - Filename, license, dependencies, ...
- modinfo command retrieves that information

```
modinfo hello.ko
```

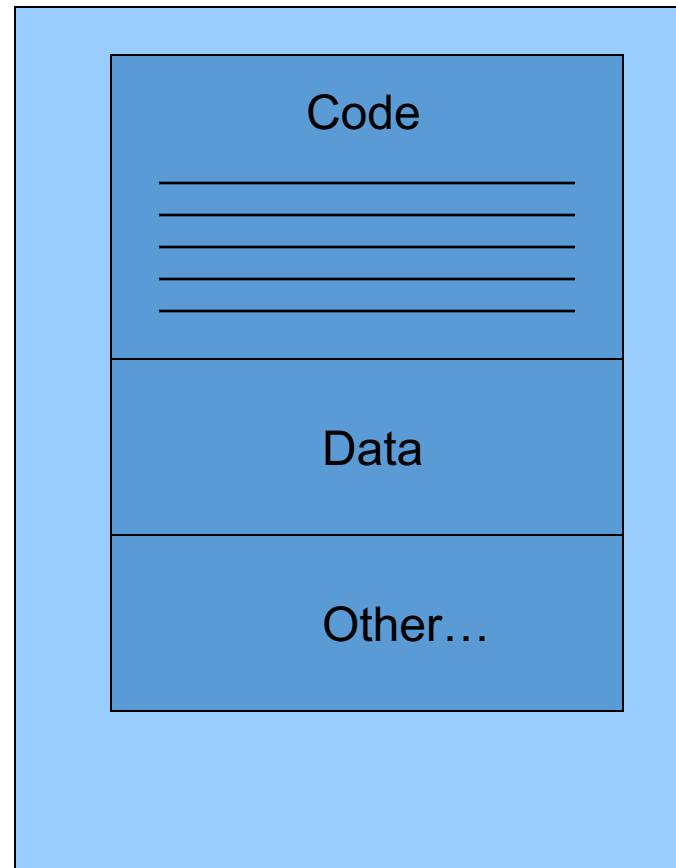


Processes

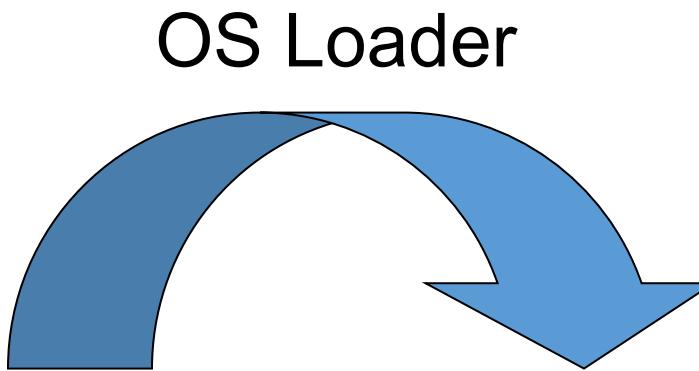
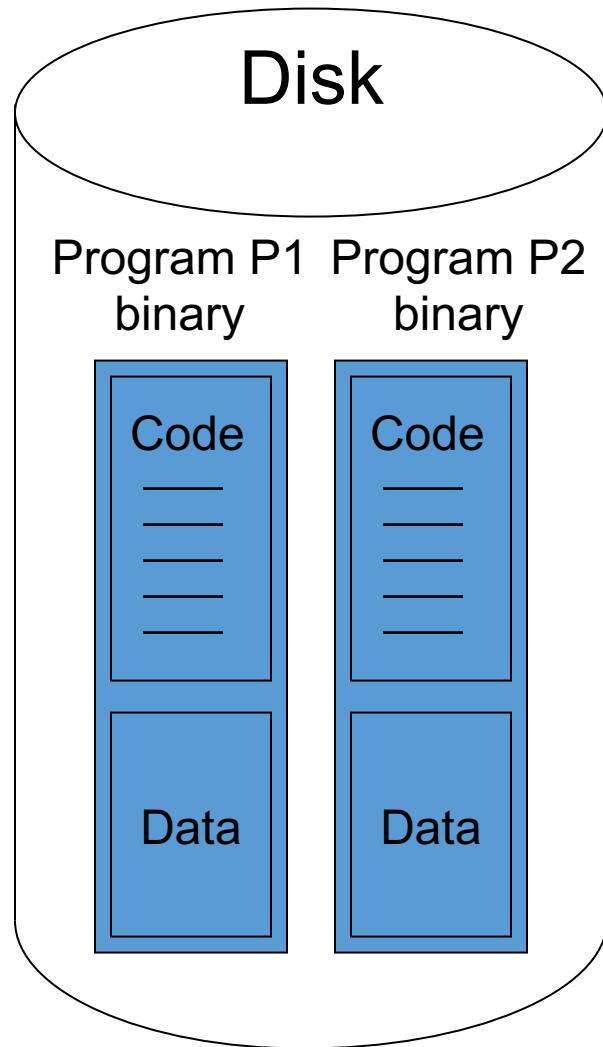
What is a Process?

- 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***

Program P1



Loading a Program into Memory

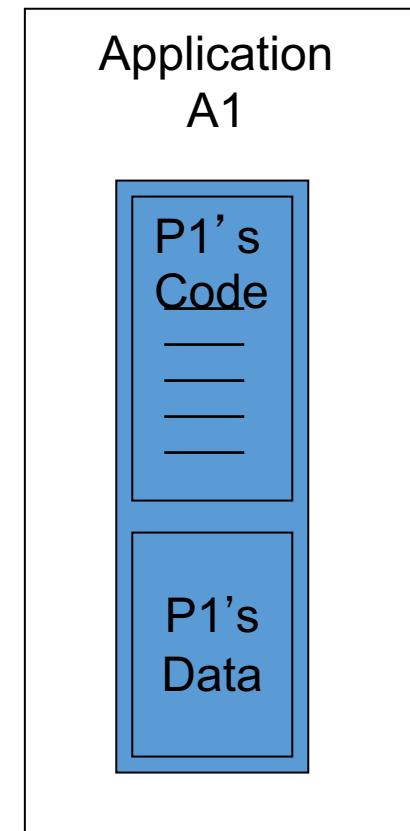


- Invoked by typing a program name in shell or double-clicking on its icon

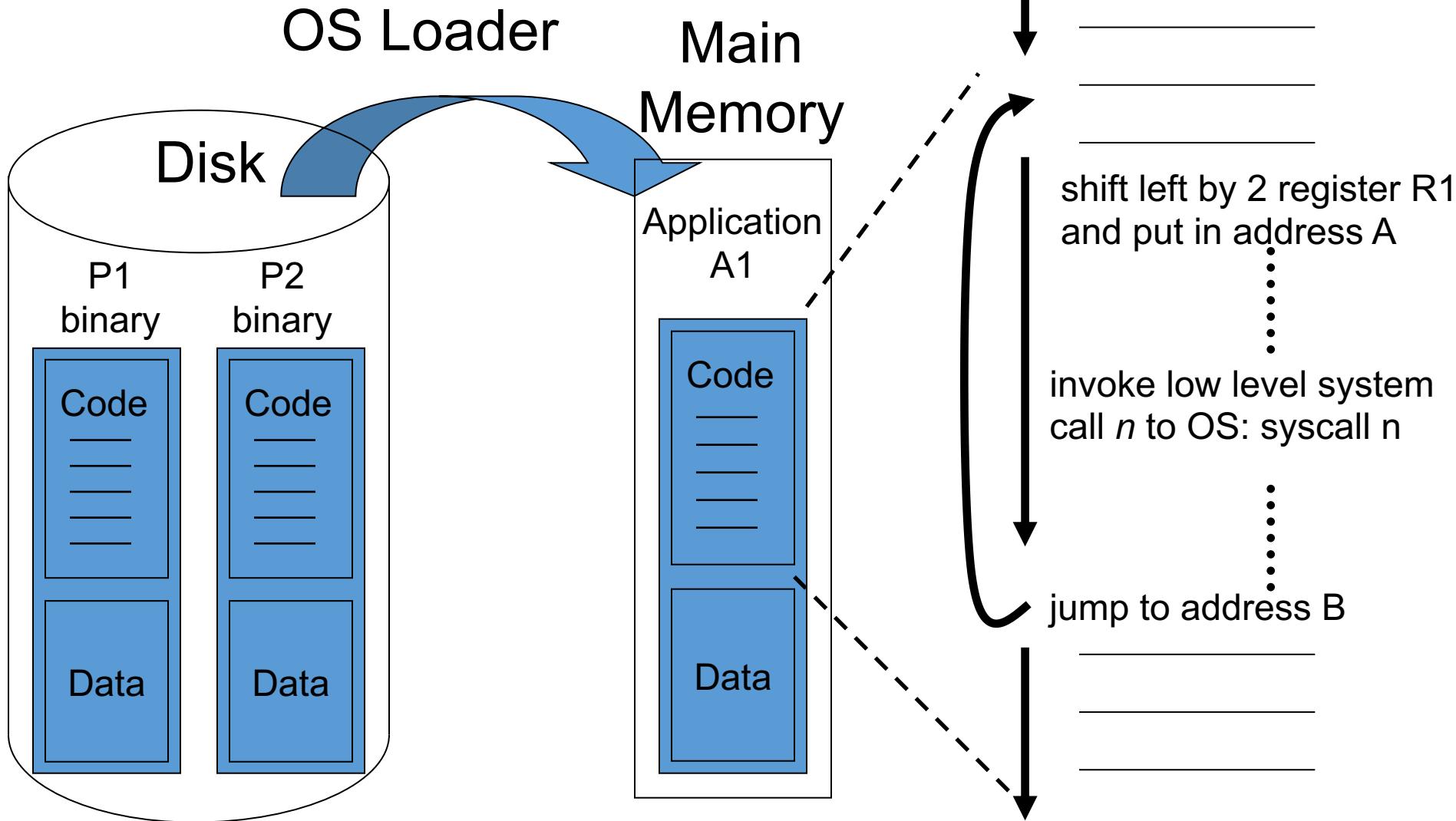
- Copies P1 from disk to RAM

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

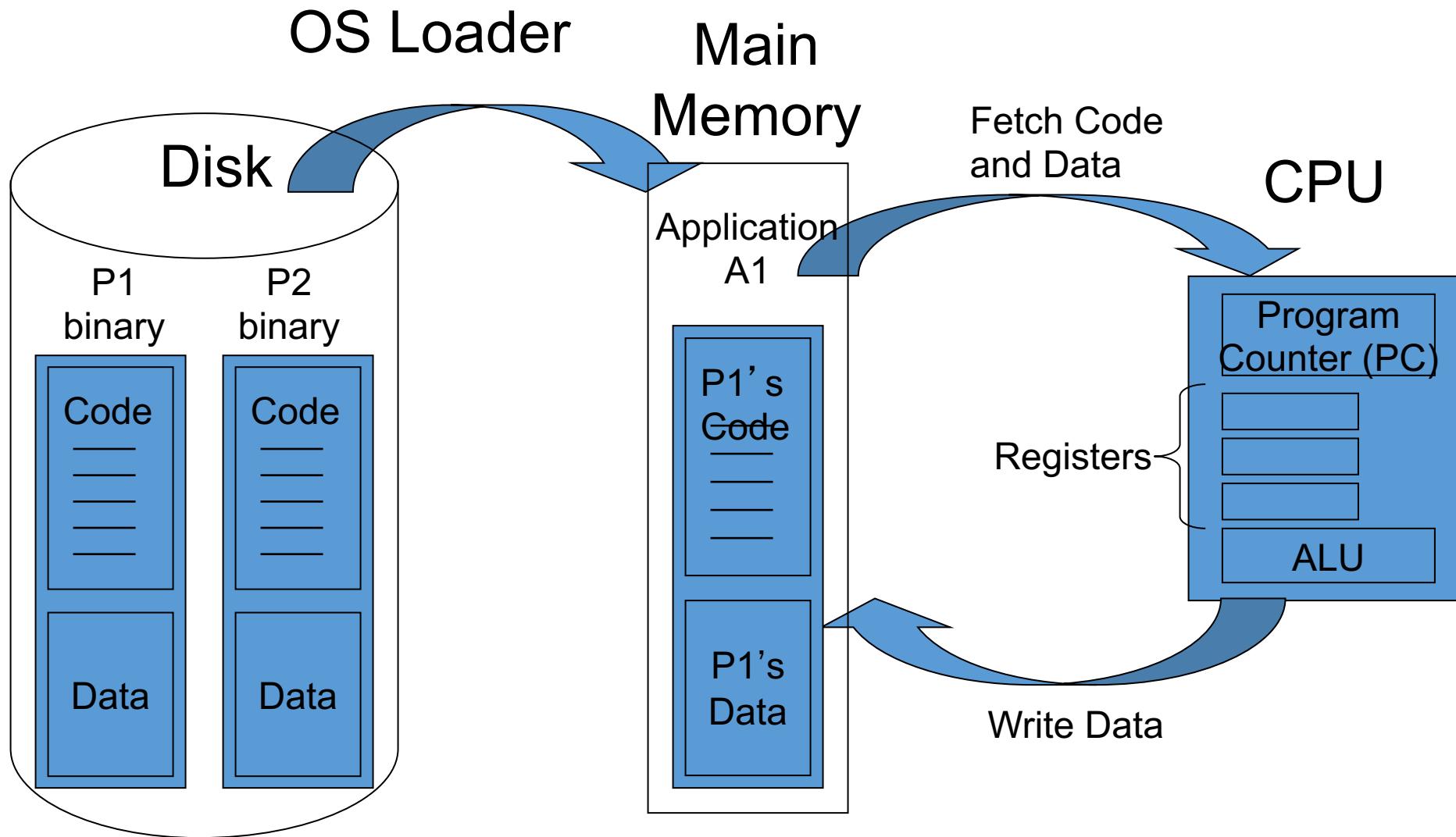
Main
Memory



Loading and Executing a Program



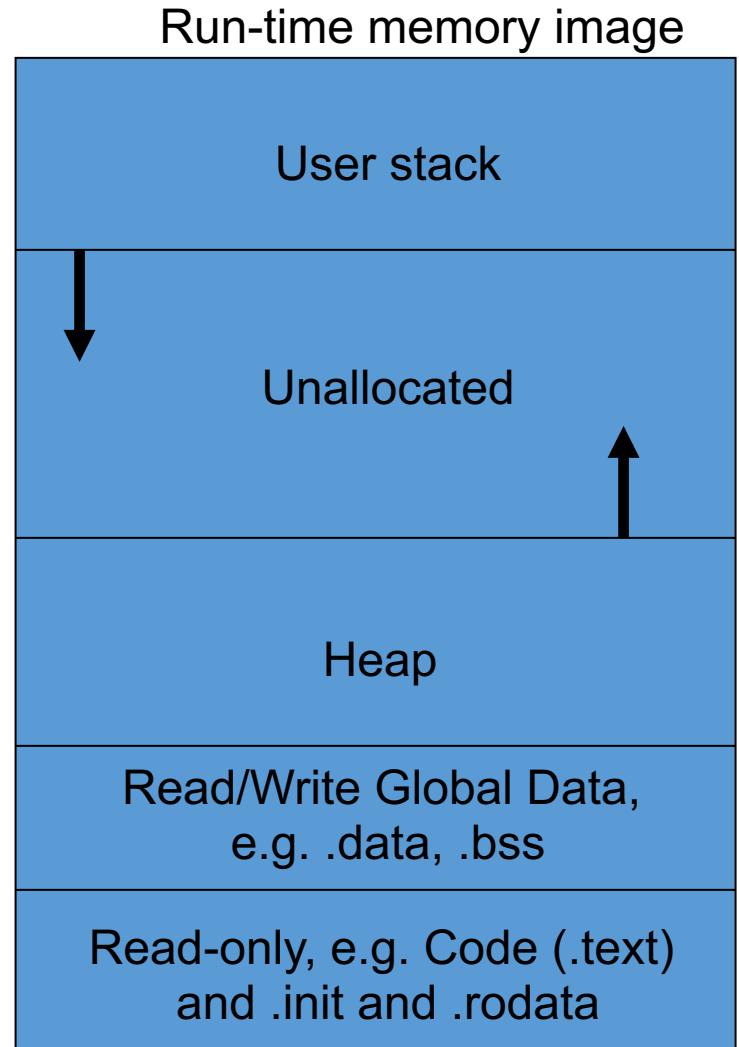
Loading and Executing a Program



Loading Executable Object Files

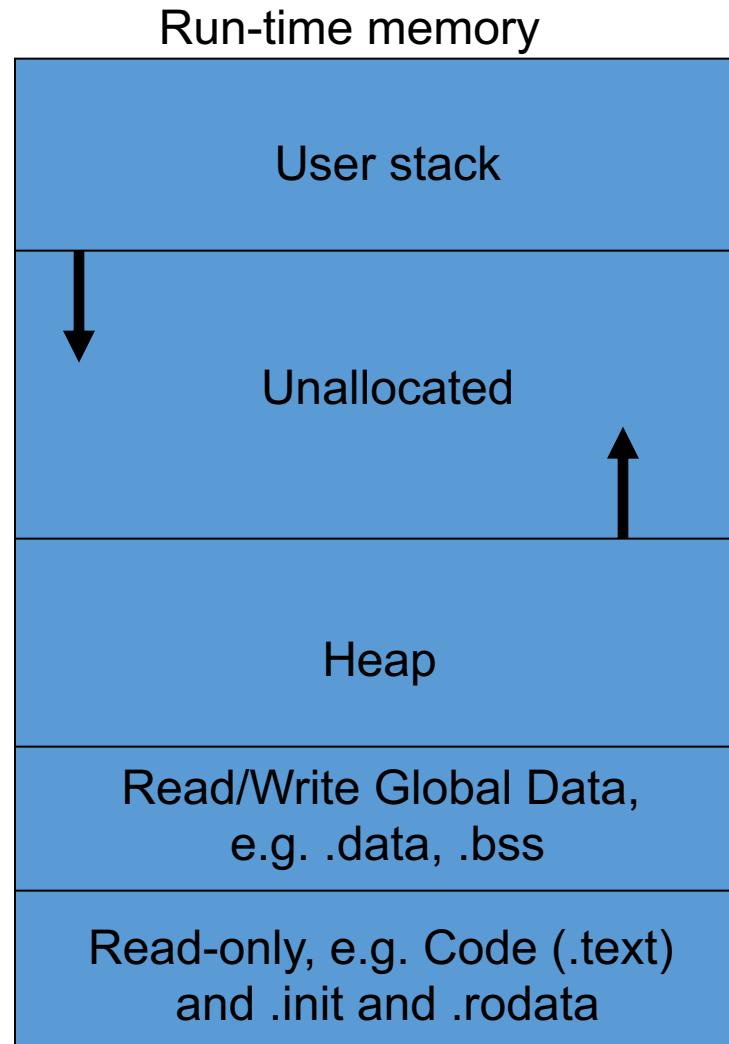
- When a program is loaded into RAM, it becomes an actively executing application
- The OS allocates a stack and heap to the app in addition to code and global data.
 - A call stack is for local variables
 - A heap is for dynamic variables, e.g. malloc()
 - Usually, stack grows downward from high memory,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

But this is architecture-specific



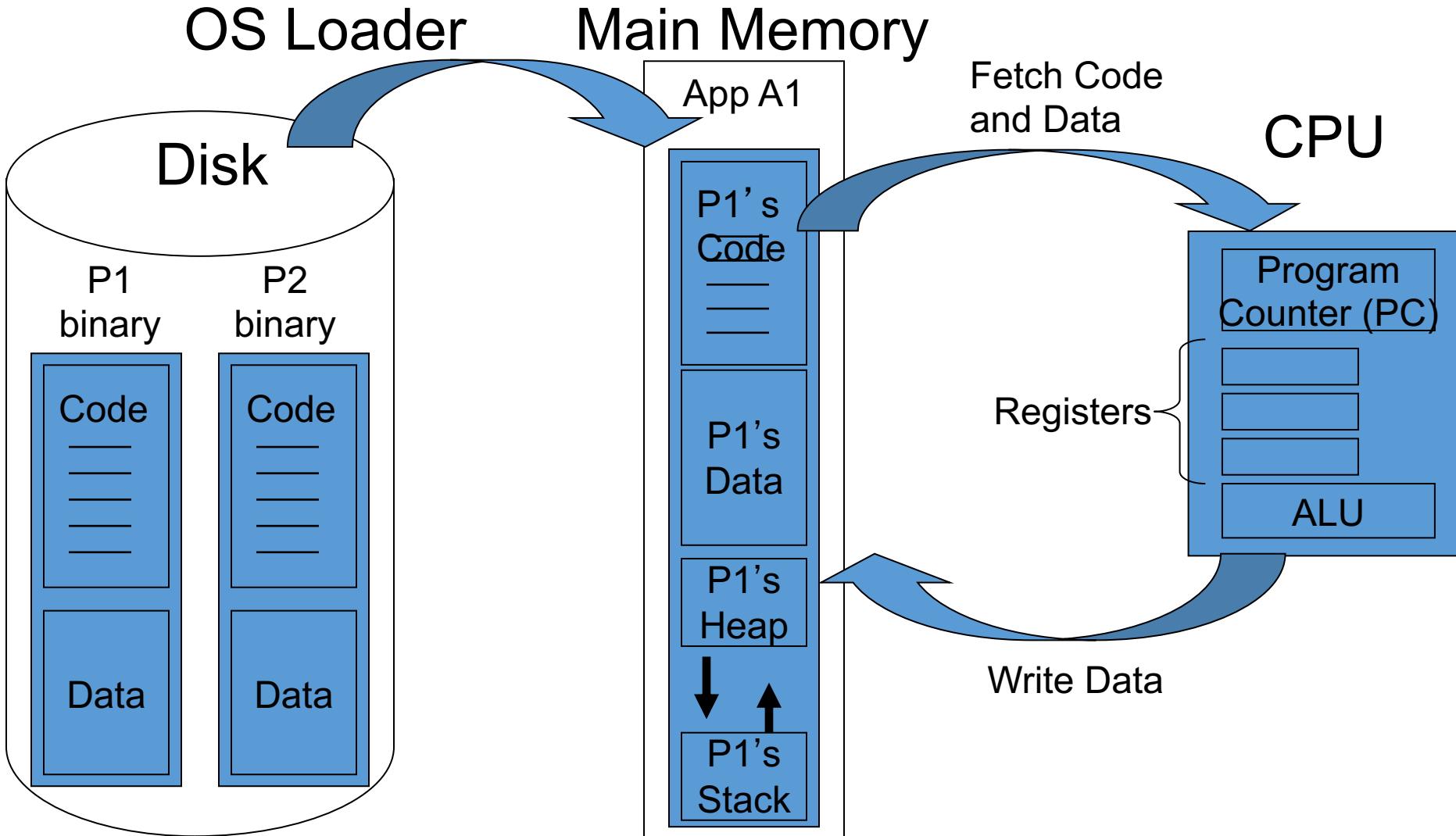
Running Executable Object Files

- Stack contains local variables
 - As main() calls function f_1 , we allocate f_1 's local variables on the stack
 - If f_1 calls f_2 , we allocate f_2 's variables on the stack below f_1 's, thereby growing the stack, etc...
 - When f_2 is done, we deallocate f_2 's local variables, popping them off the stack, and return to f_1
- 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



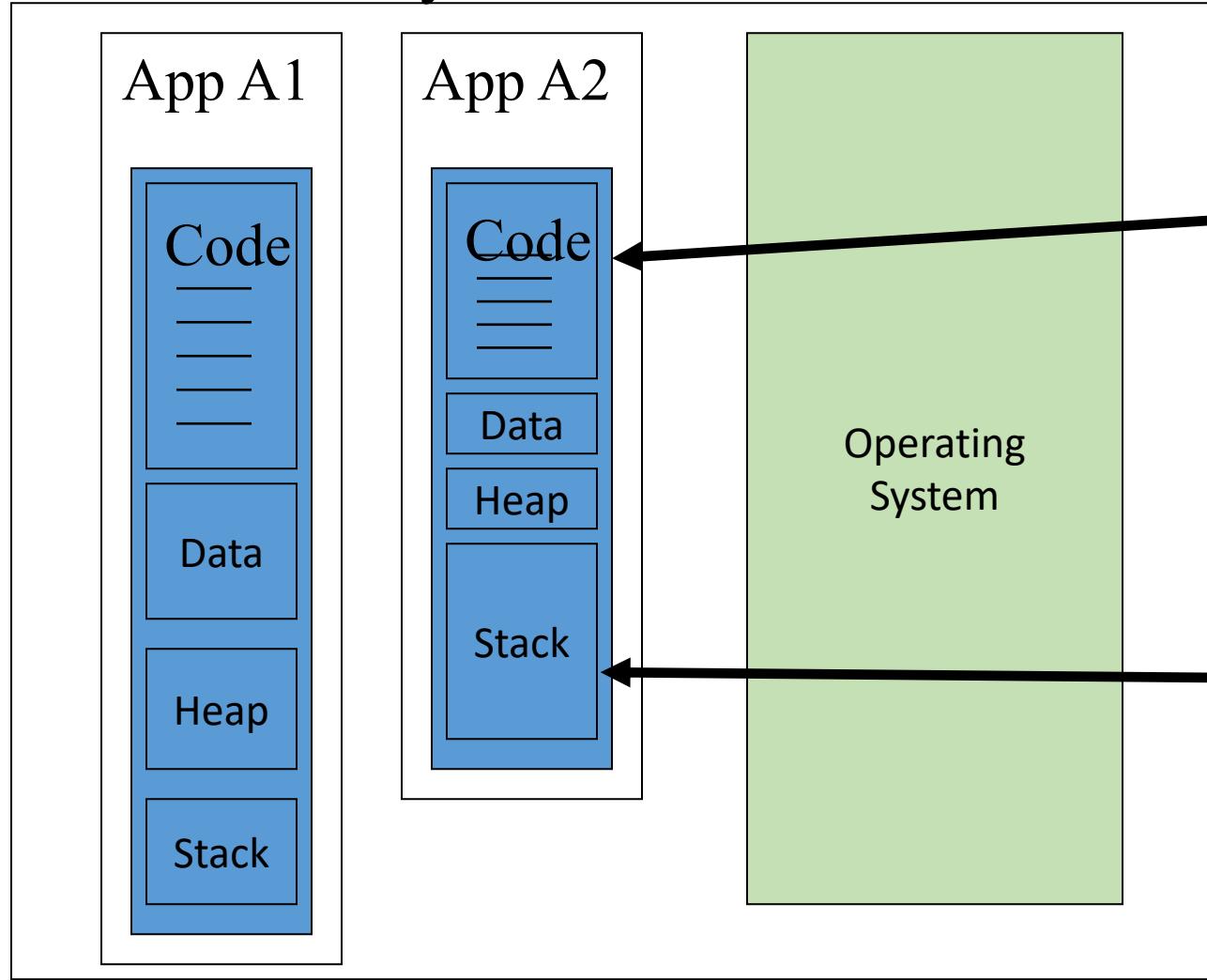
Loading and Executing a Program

– a more complete picture –

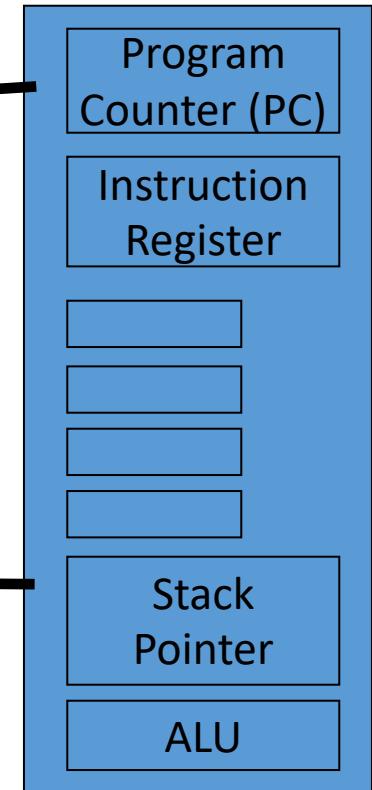


Multiple Applications + OS

Main Memory

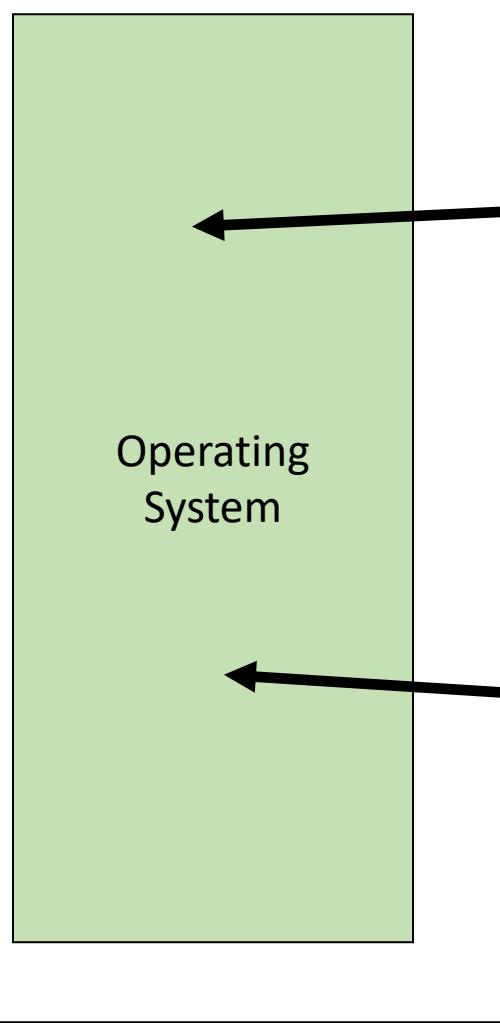
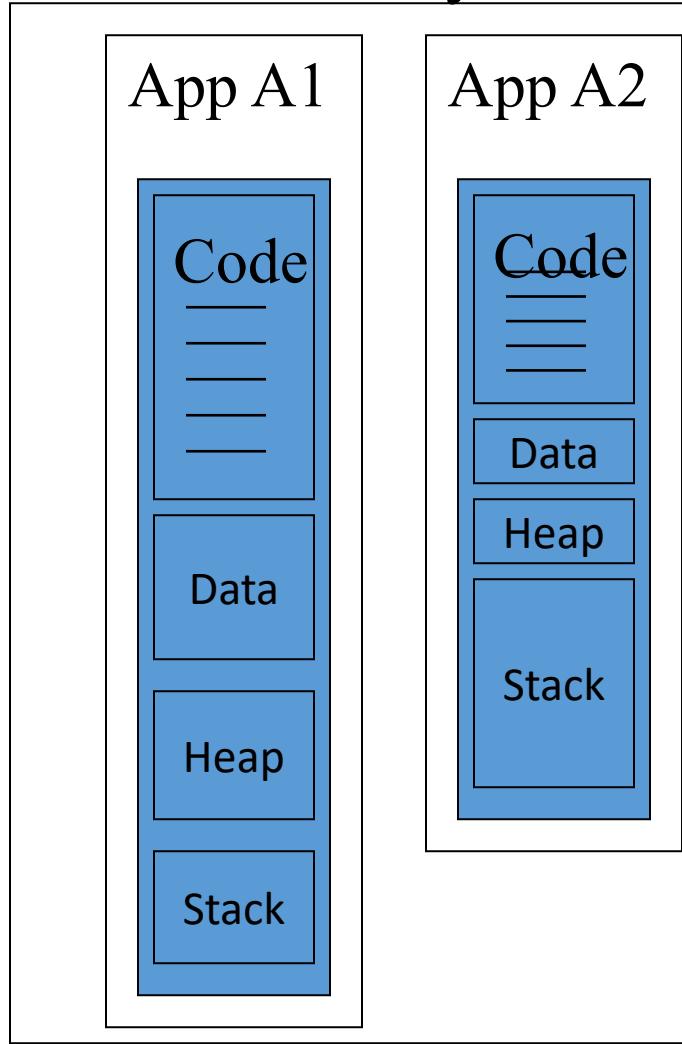


CPU Execution

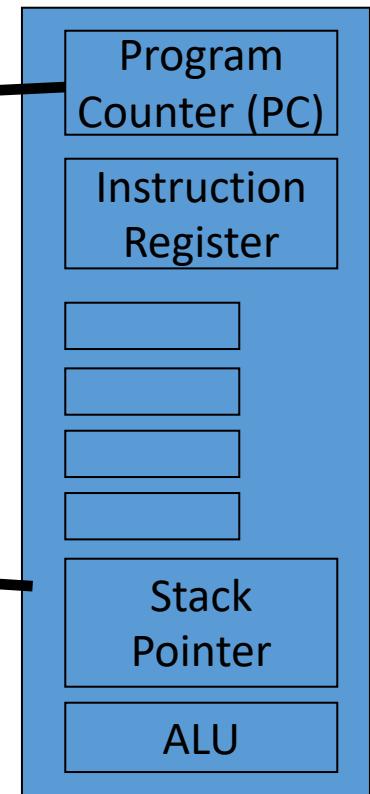


Multiple Applications + OS

Main Memory

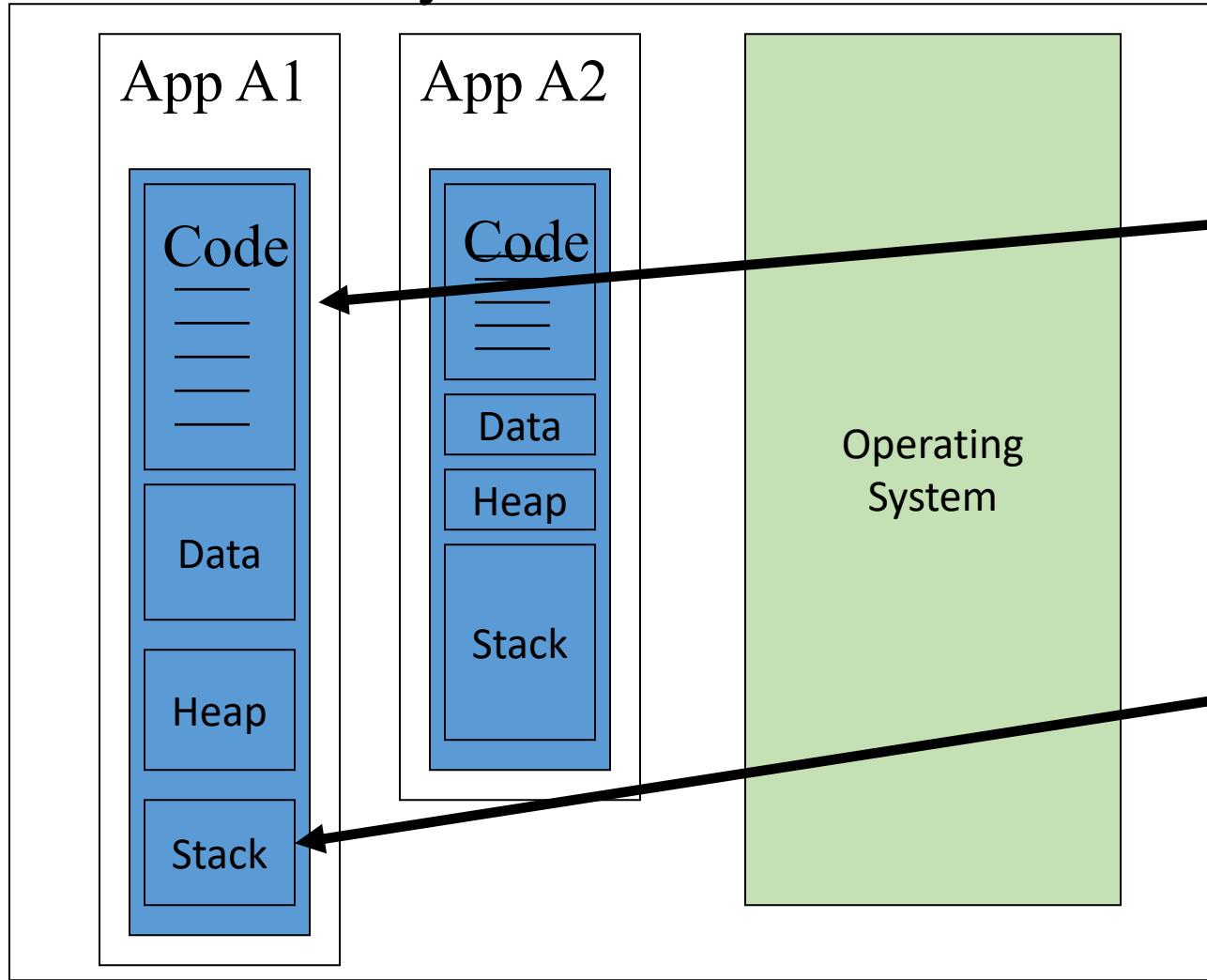


CPU Execution

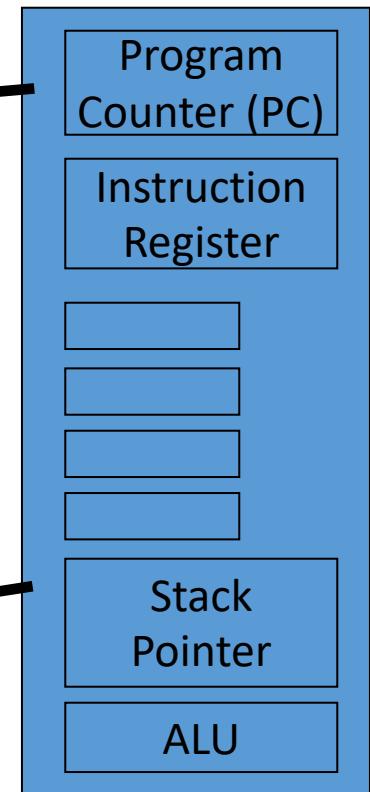


Multiple Applications + OS

Main Memory



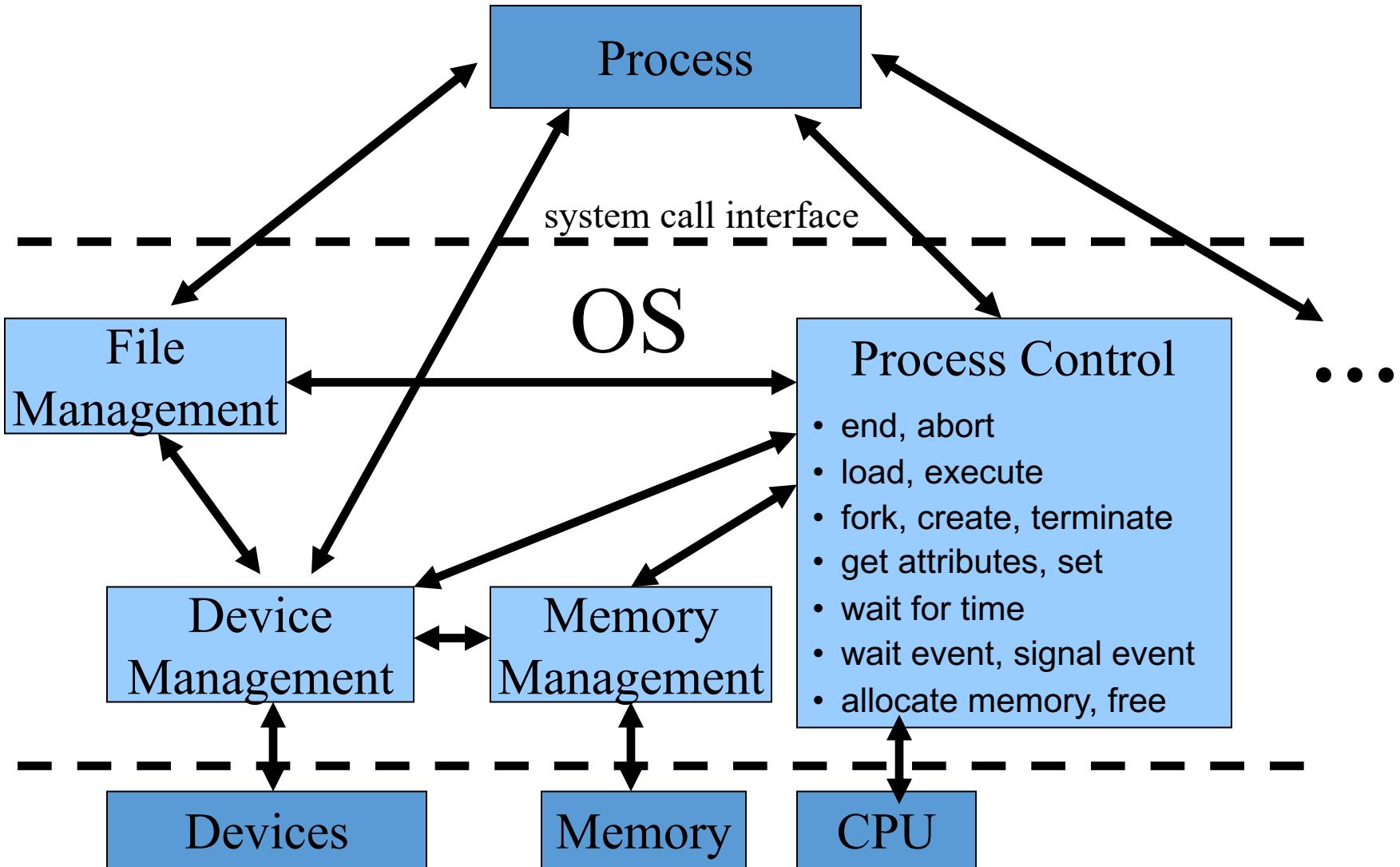
CPU Execution



Applications and Processes

- An application may consist of multiple processes, each executing in its own address space
 - 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 functionalities

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

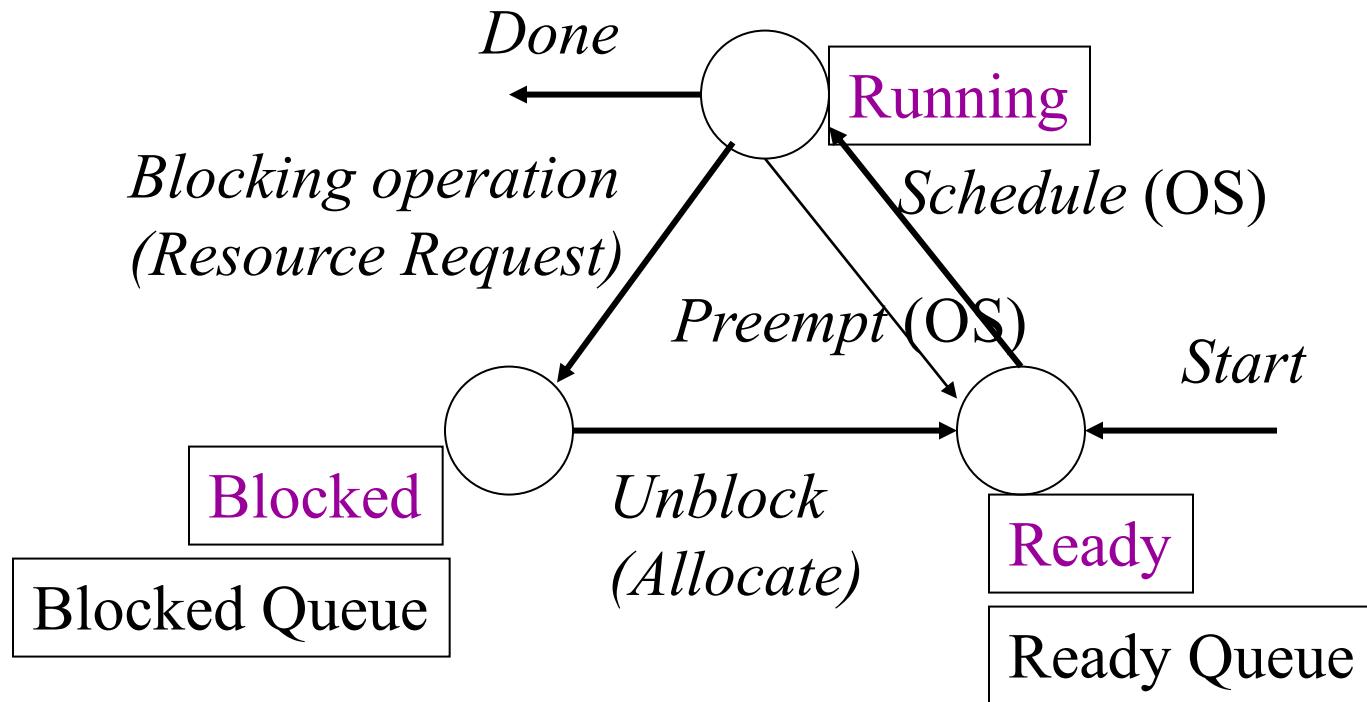


what defines the State of a Process

- Memory image: Code, data, heap, stack
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID, privilege
- Program counter (PC) value
- CPU registers' values
- 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



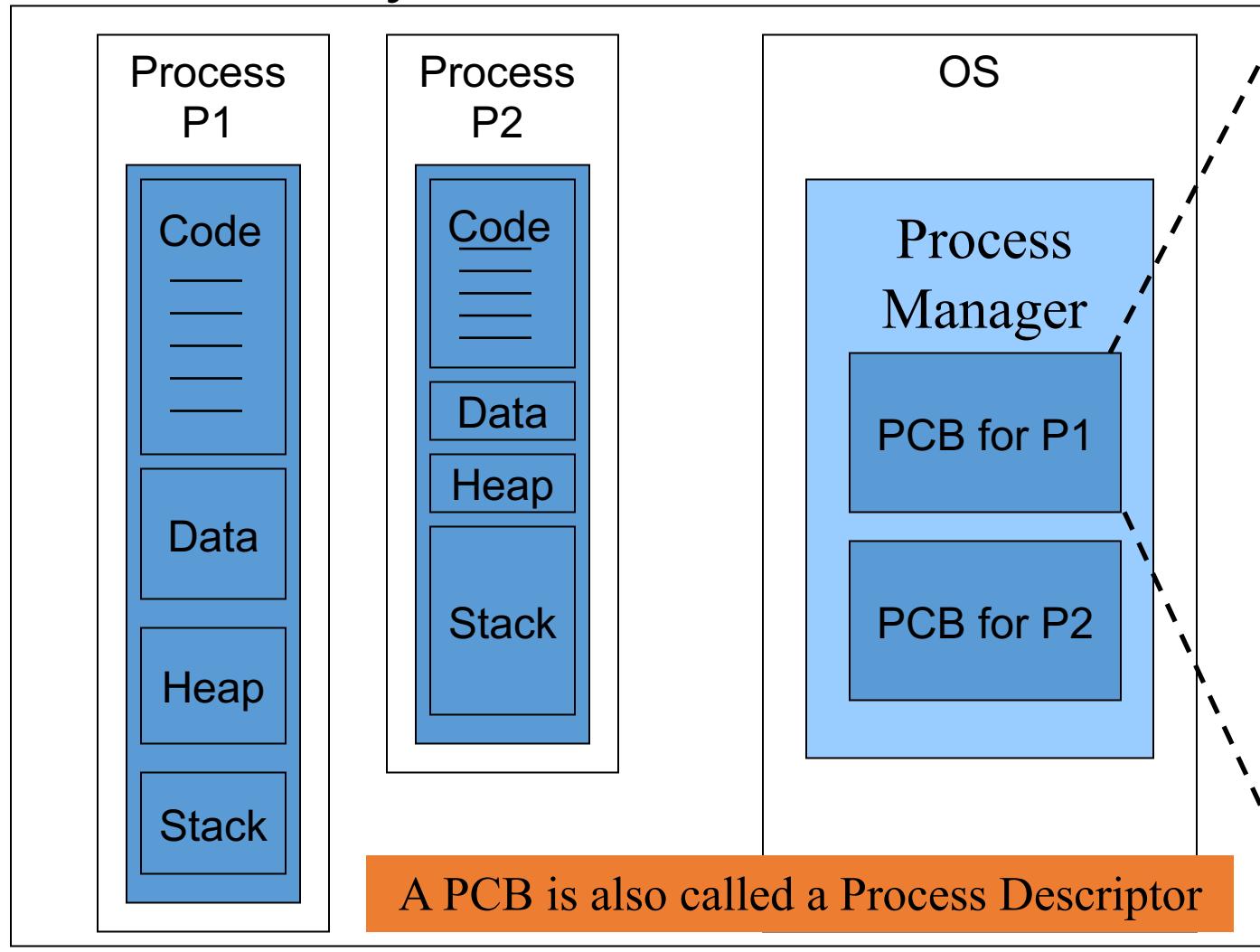
Process Control Block

- 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)

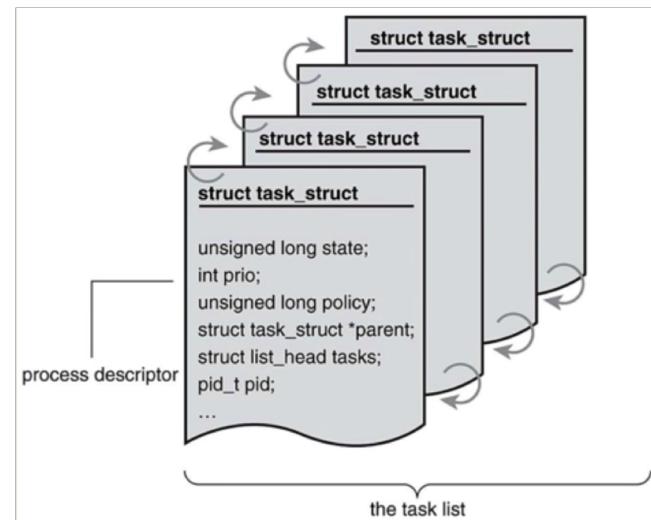
Main Memory



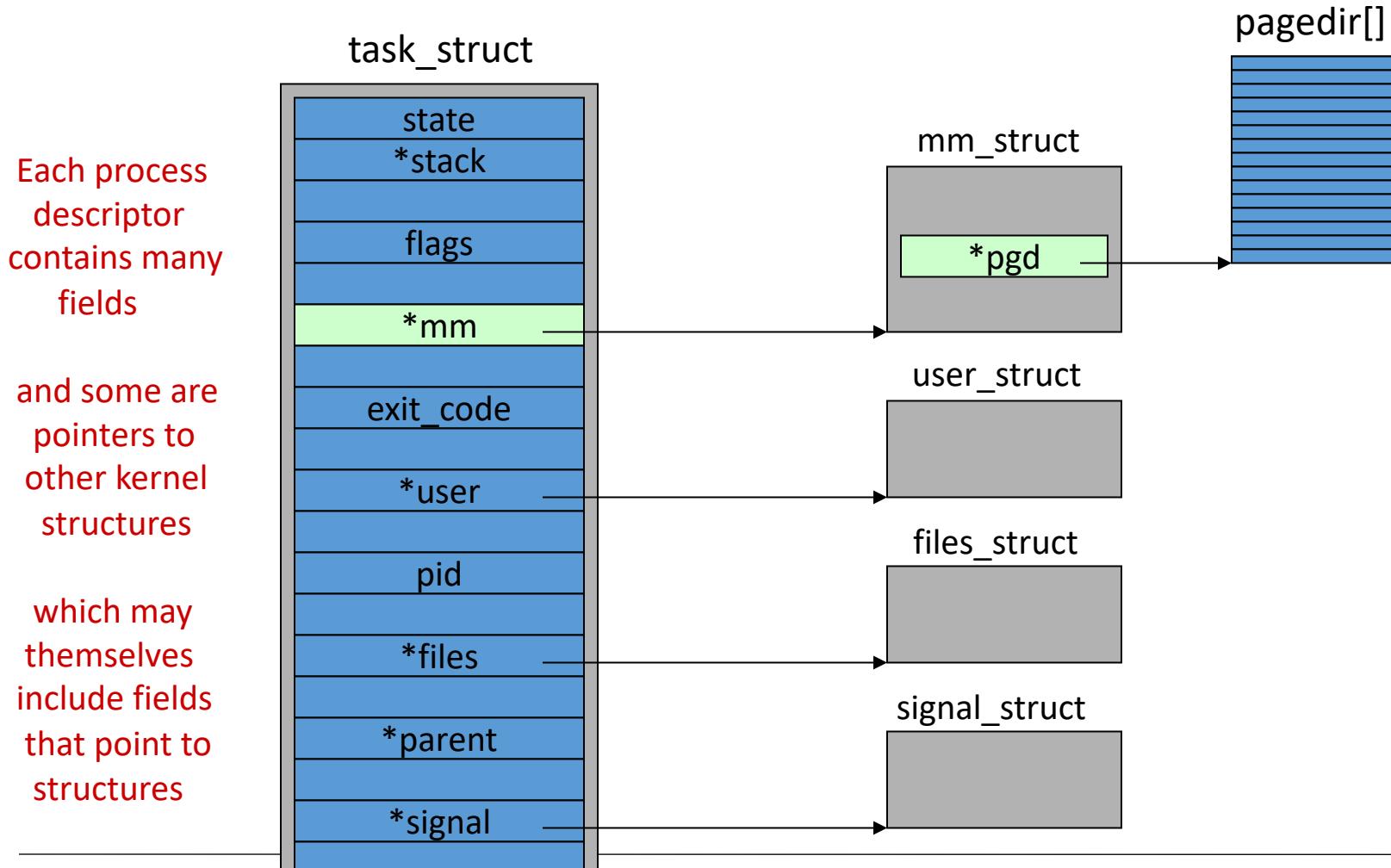
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID
- Program Counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

Process Descriptor

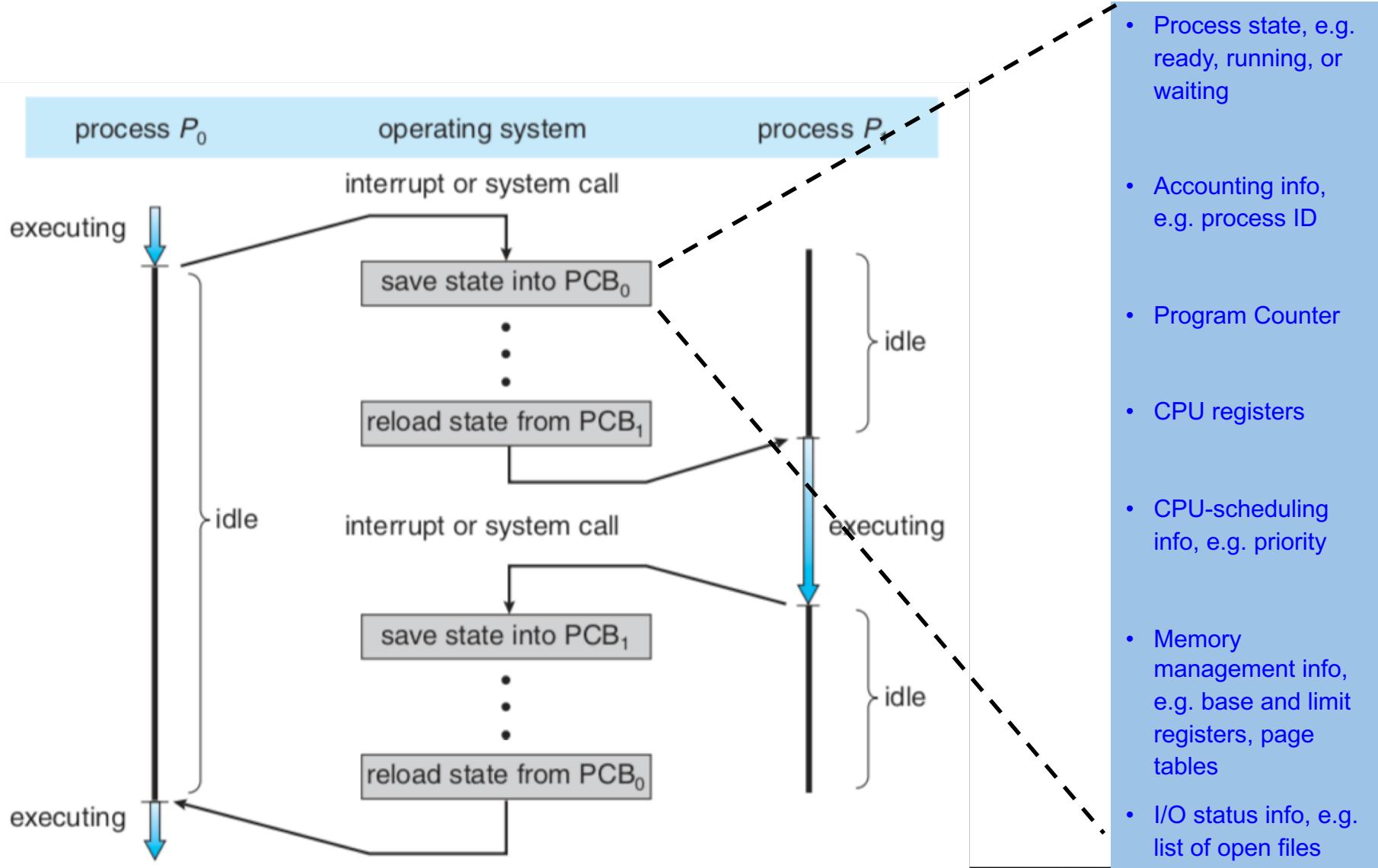
- Process – dynamic, program in motion
 - Kernel data structures to maintain "state"
 - Descriptor, task_struct
 - Complex struct with pointers to others
- Type of info in task_struct
 - state,
 - id,
 - priorities,
 - locks,
 - files,
 - signals,
 - memory maps,
 - queues, list pointers, ...



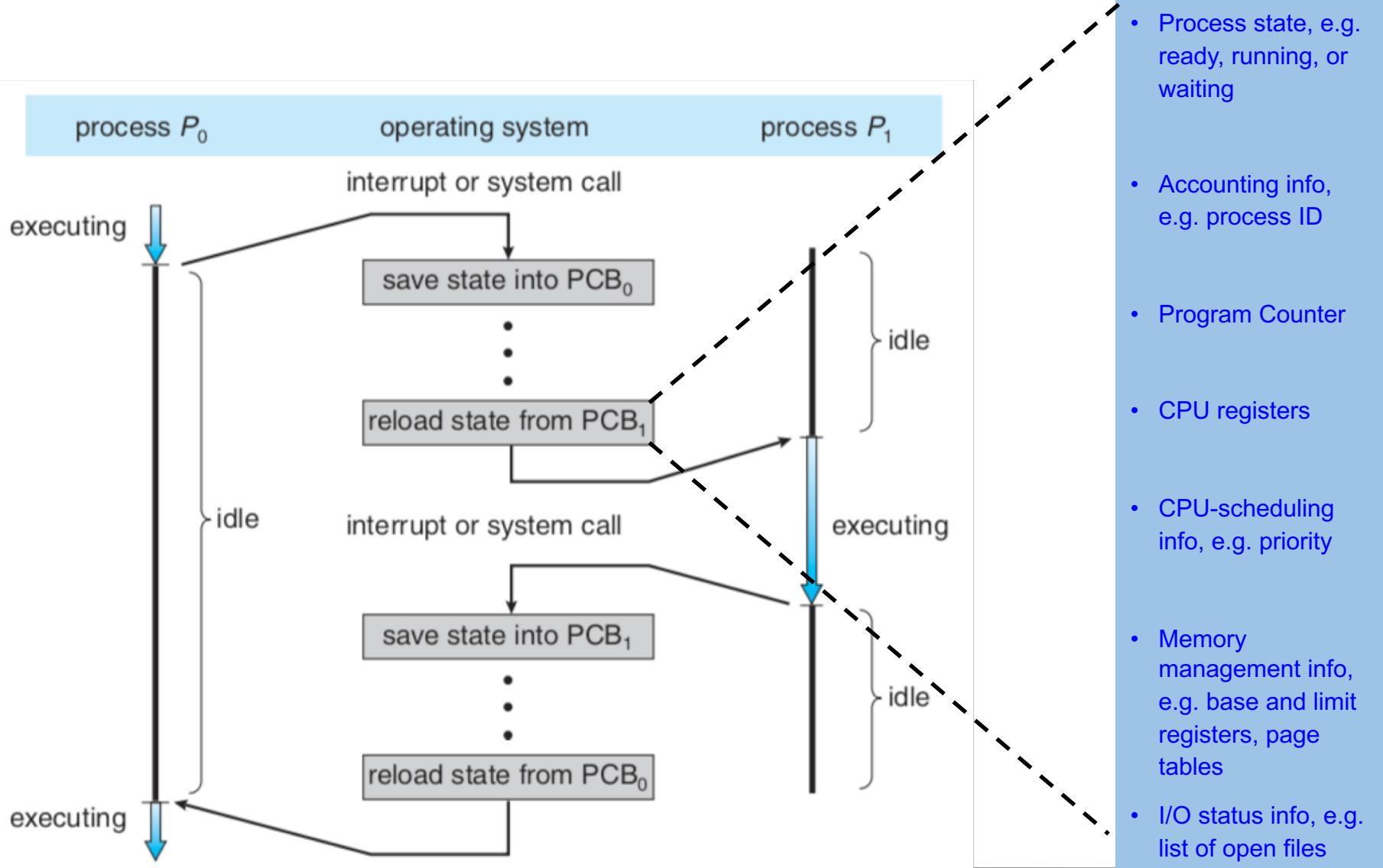
The Linux process descriptor



Context switch from one processes to another



Context switch from one processes to another

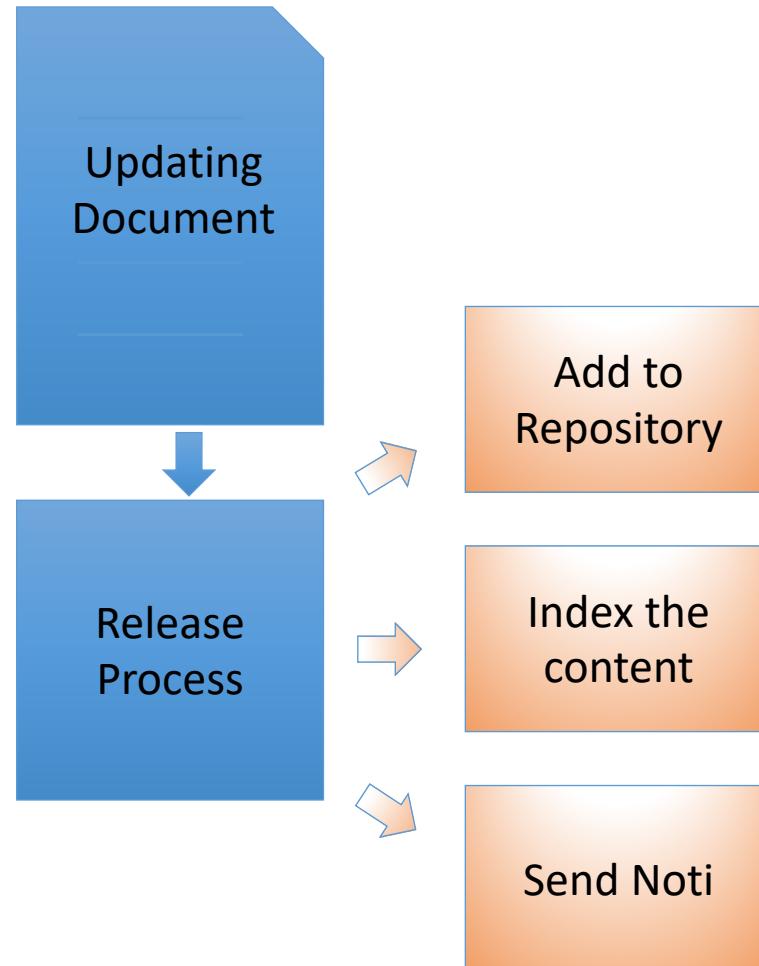


Process Manager

- Creation/deletion of processes (and threads)
- Synchronization of processes (and threads)
- Managing process state
 - Process 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 pipeline Example

- Example: When updated document is being released (Dropbox)
 - Place into the repository
 - Add to context retrieval system
 - Send notification all copies



Creating Processes in Windows

- CreateProcess() call, in Windows
 - 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
 - Combines *fork()* and *exec()* system calls in UNIX/Linux and achieves the same effect



Creating Processes in UNIX/Linux

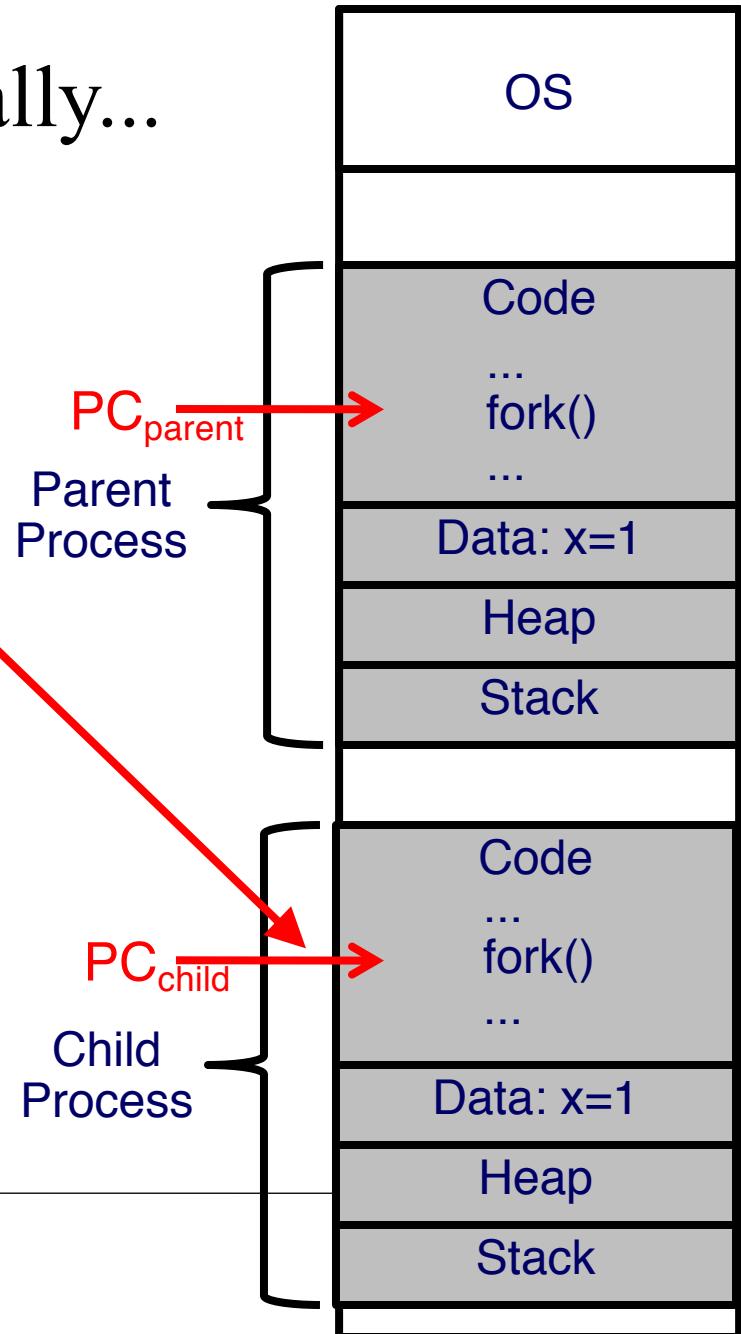
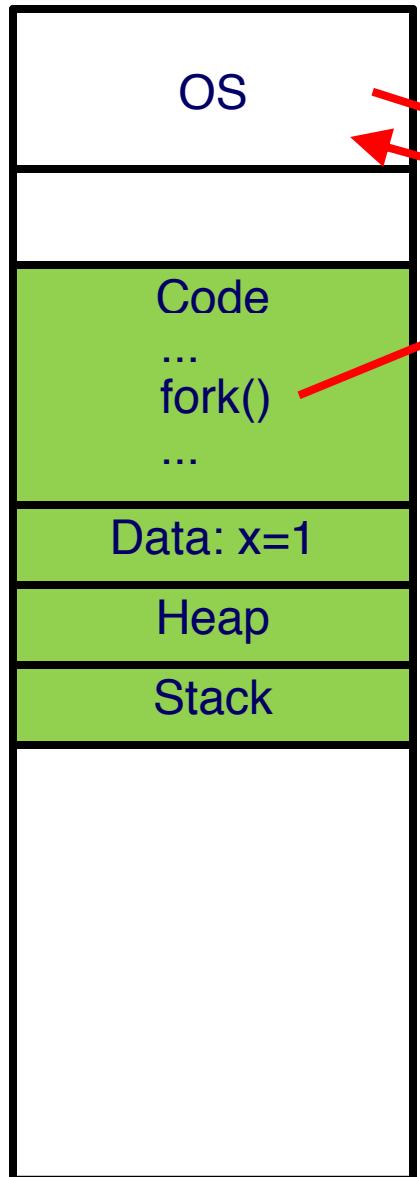
- Use *fork()* command to create/spawn new processes from within a process
 - When a process (called parent process) calls *fork()*, a new process (called child process) is created
 - Child process is an exact copy of the parent process
 - All code and data are exactly the same
 - All addresses are appropriately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



Memory (before fork)

Fork() conceptually...

Memory (after fork)



- Fork() duplicates address space of parent in the child
- Both execute **concurrently**
- How does a process know if it is the parent or the child?



fork ()

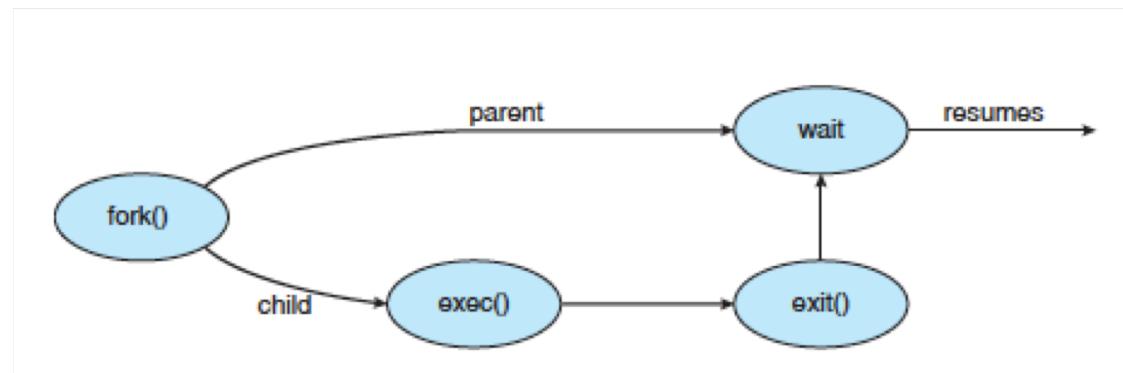
- The `fork()` call returns an *int* value
 - In the parent process, returned value is child's PID
 - In the child, returned value is 0
 - Since both parent and child execute the same code starting from the same place, i.e. just after the `fork()`, then to differentiate the child's behavior from the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



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

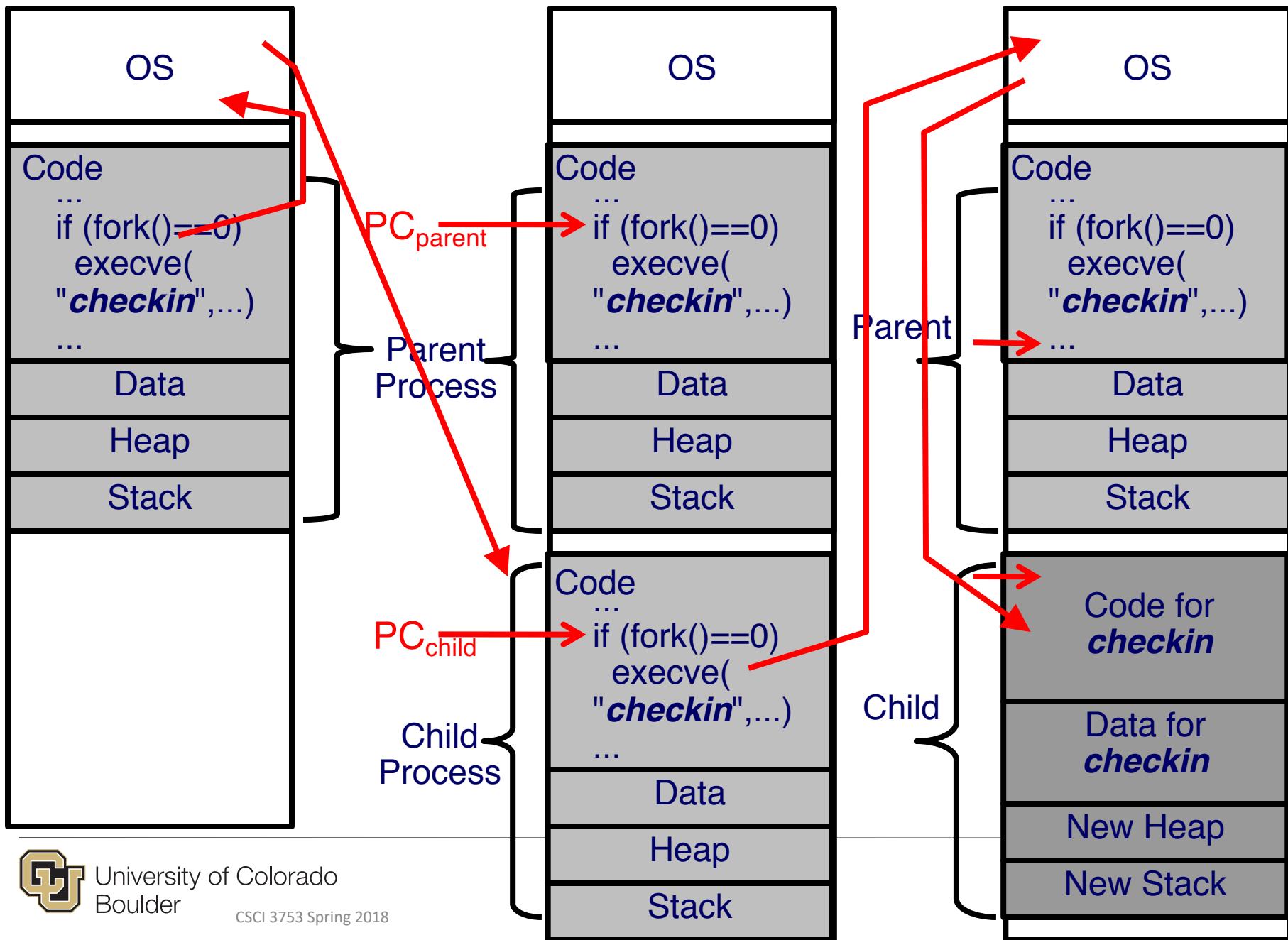
```
PID = fork();
if (PID==0) { /* child */
    exec("checkin");
    exit(0);
}
/* the parent's code here */
```



Memory (before fork)

Memory (after fork)

Memory (after execve)



University of Colorado
Boulder

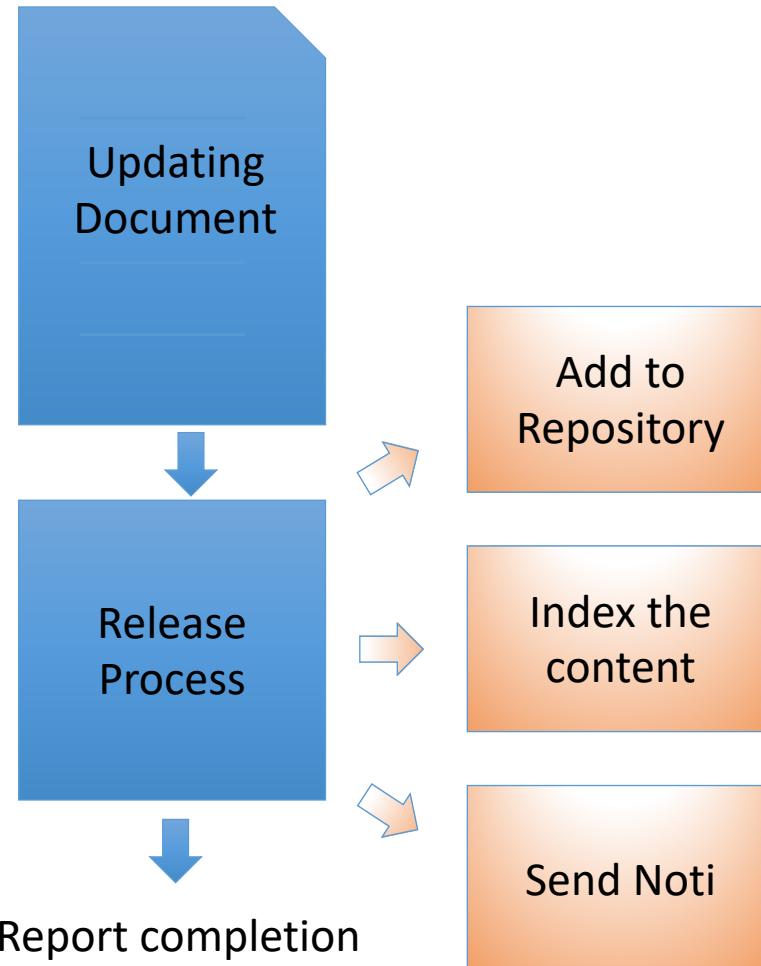
CSCI 3753 Spring 2018

Process pipeline Example

- Example: When updated document is being released (Dropbox)

- Place into the repository
- Add to context retrieval system
- Send notification all copies

Wait for each process to complete before starting the next process



Waiting on Processes

- 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
- There is also *waitpid()* to wait on a particular child process to finish

```
PID = fork();
if (PID==0) { /* child */
    exec("checkin");
    exit(0);
}

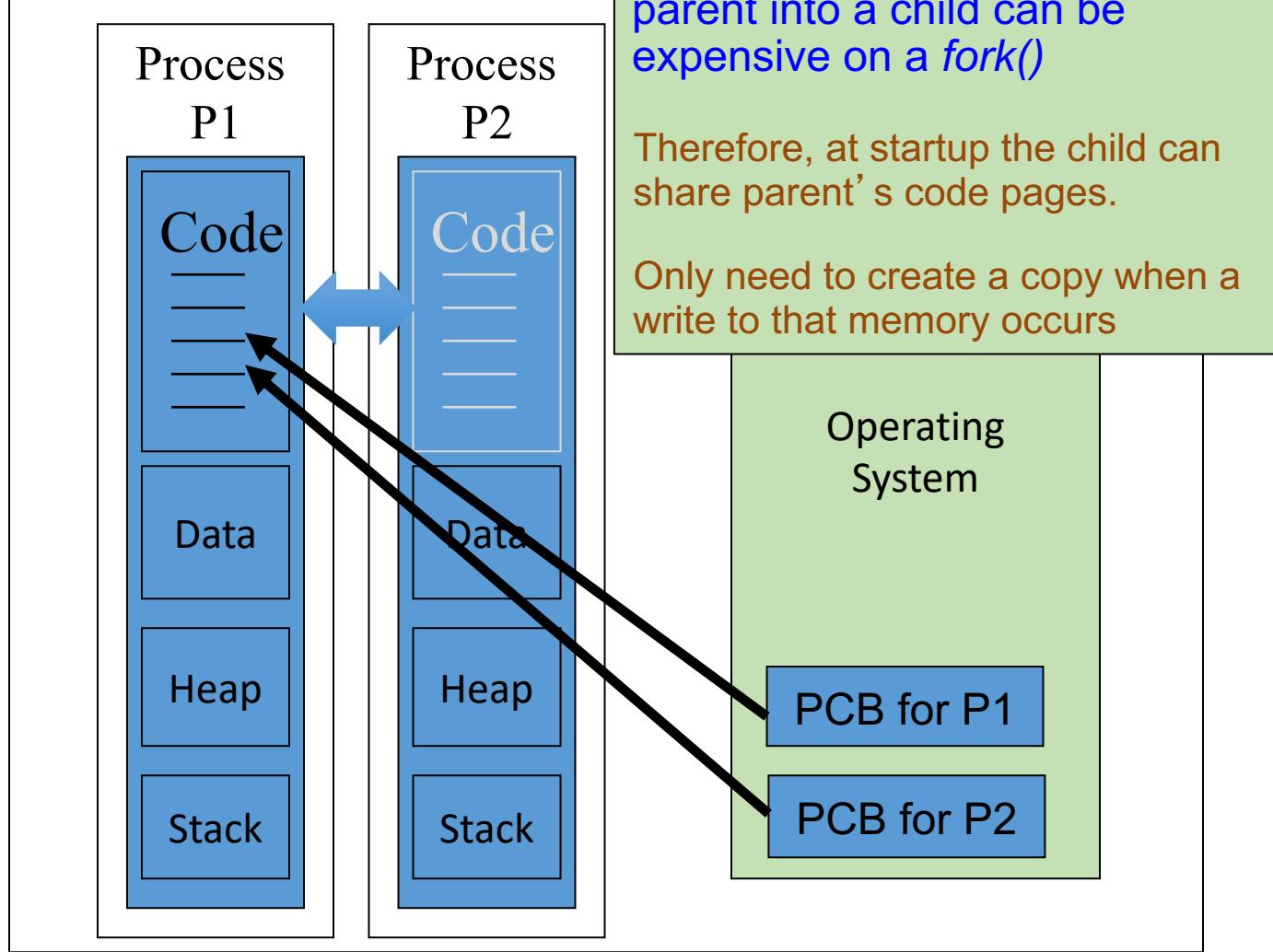
/* the parent's code here */
```

```
/* the parent's code here */
child_pid = wait();
/* child has completed */
. . .
```

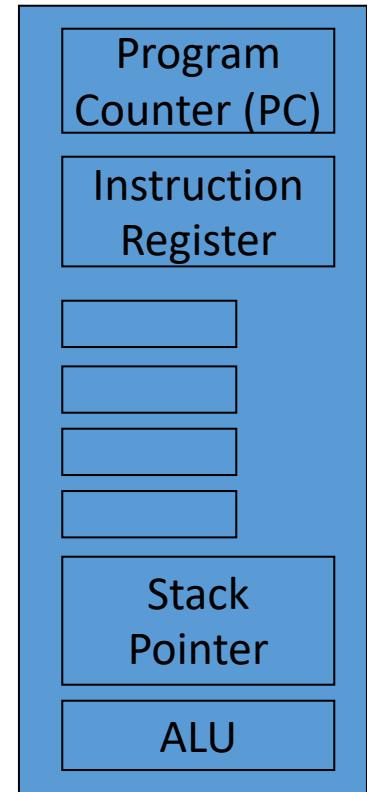


Multiple Processes + OS

Main Memory



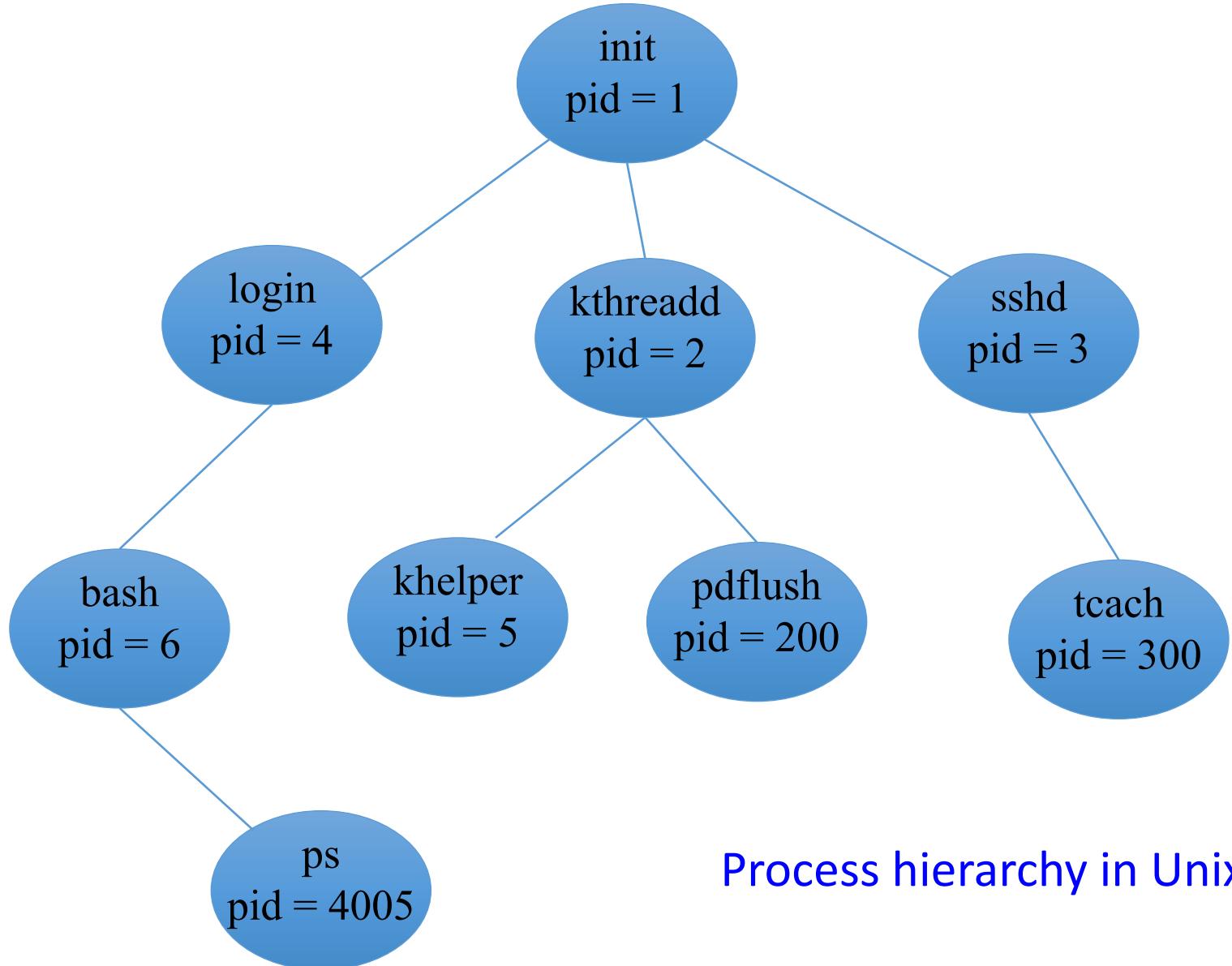
CPU Execution



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

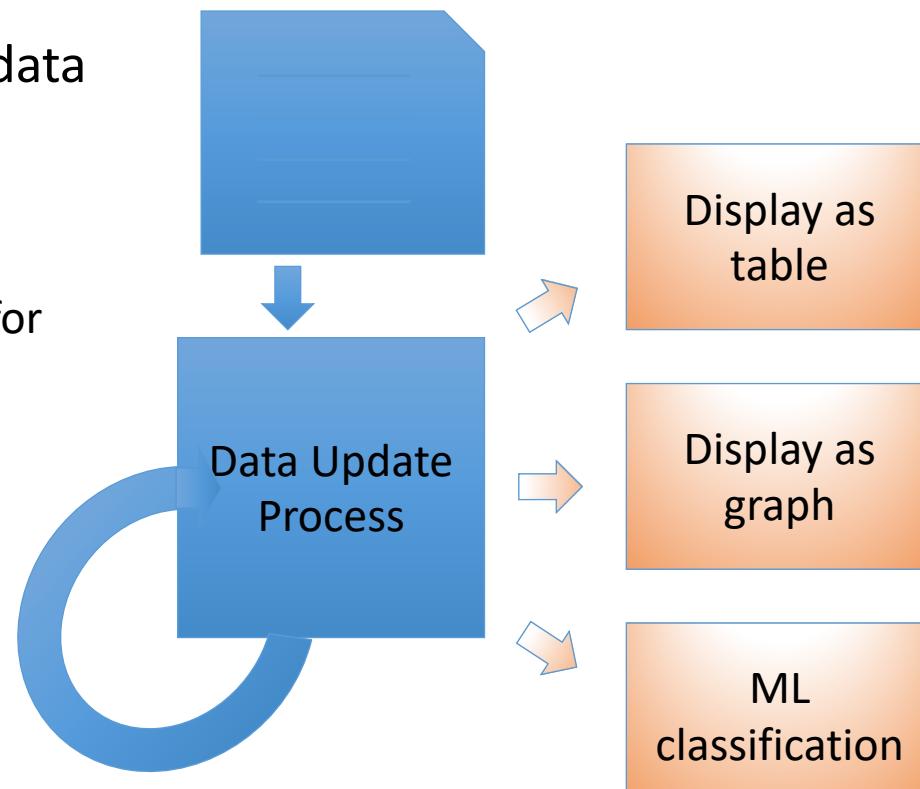




Process hierarchy in Unix

Another Process Example

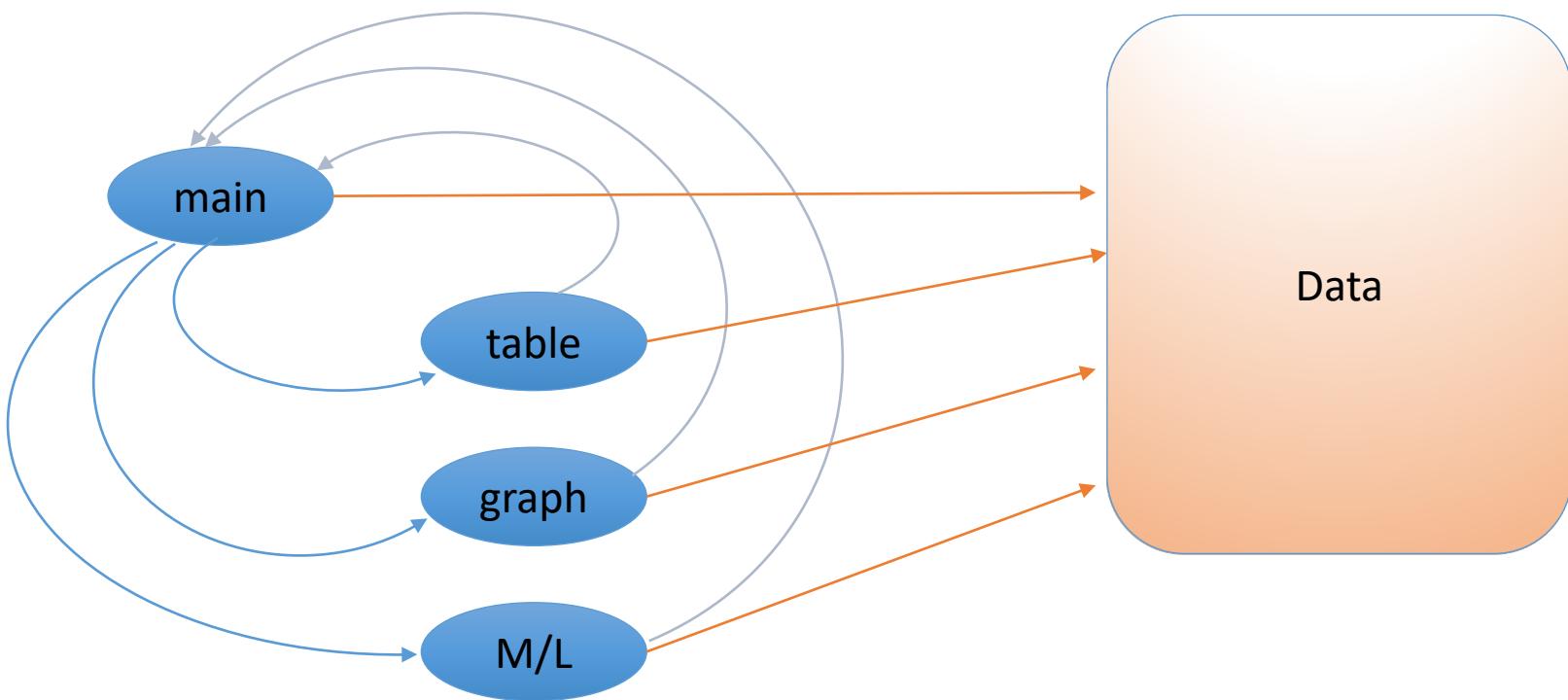
- When a new / updated data is available
 - Display as table
 - Display as graph
 - Perform a ML algorithm for classification



Repeat every time
the data is updated

All processes are independent,
but need to share the same data

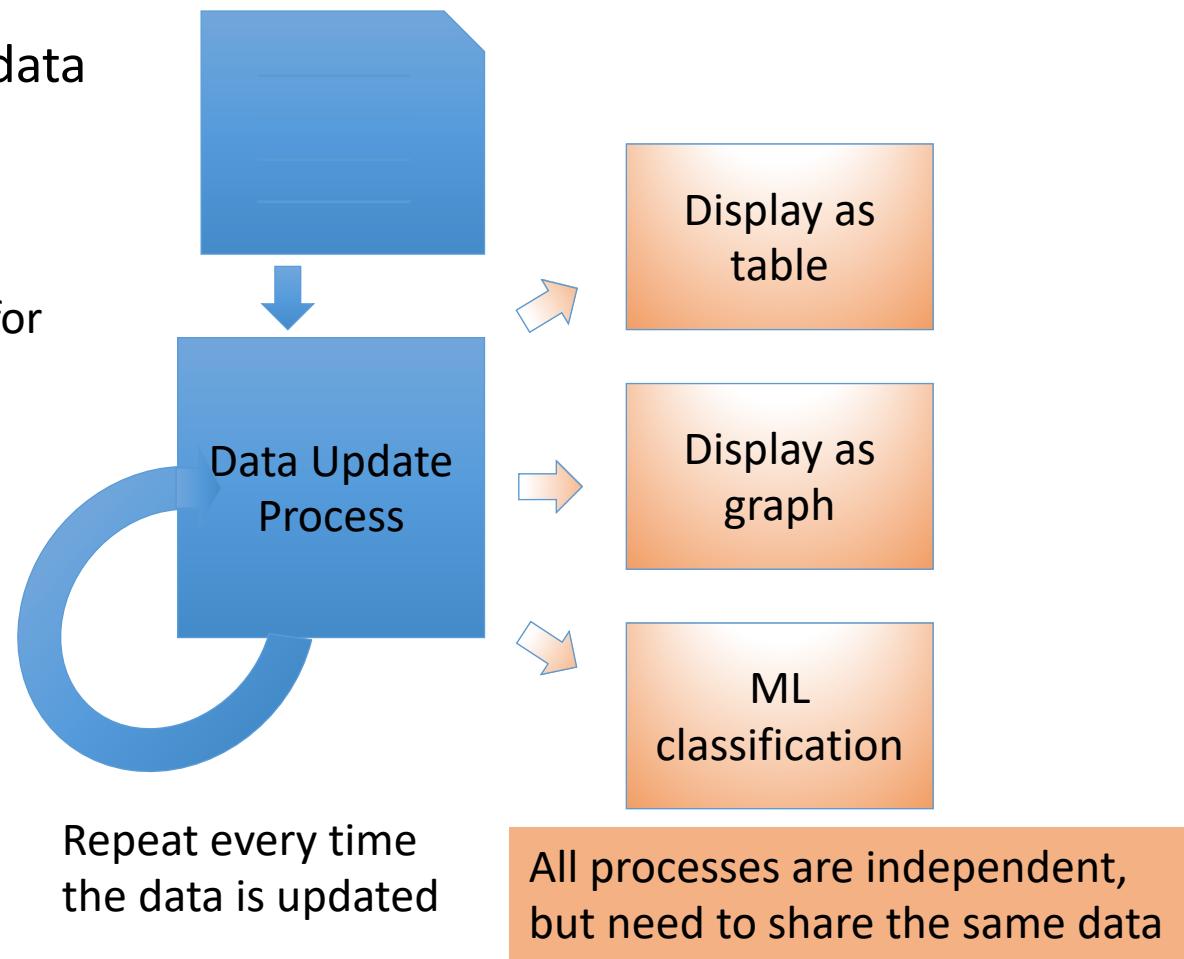
Parallel Pipeline



- All processes need to access the same data

Another Process Example

- When a new / updated data is available
 - Display as table
 - Display as graph
 - Perform a ML algorithm for classification



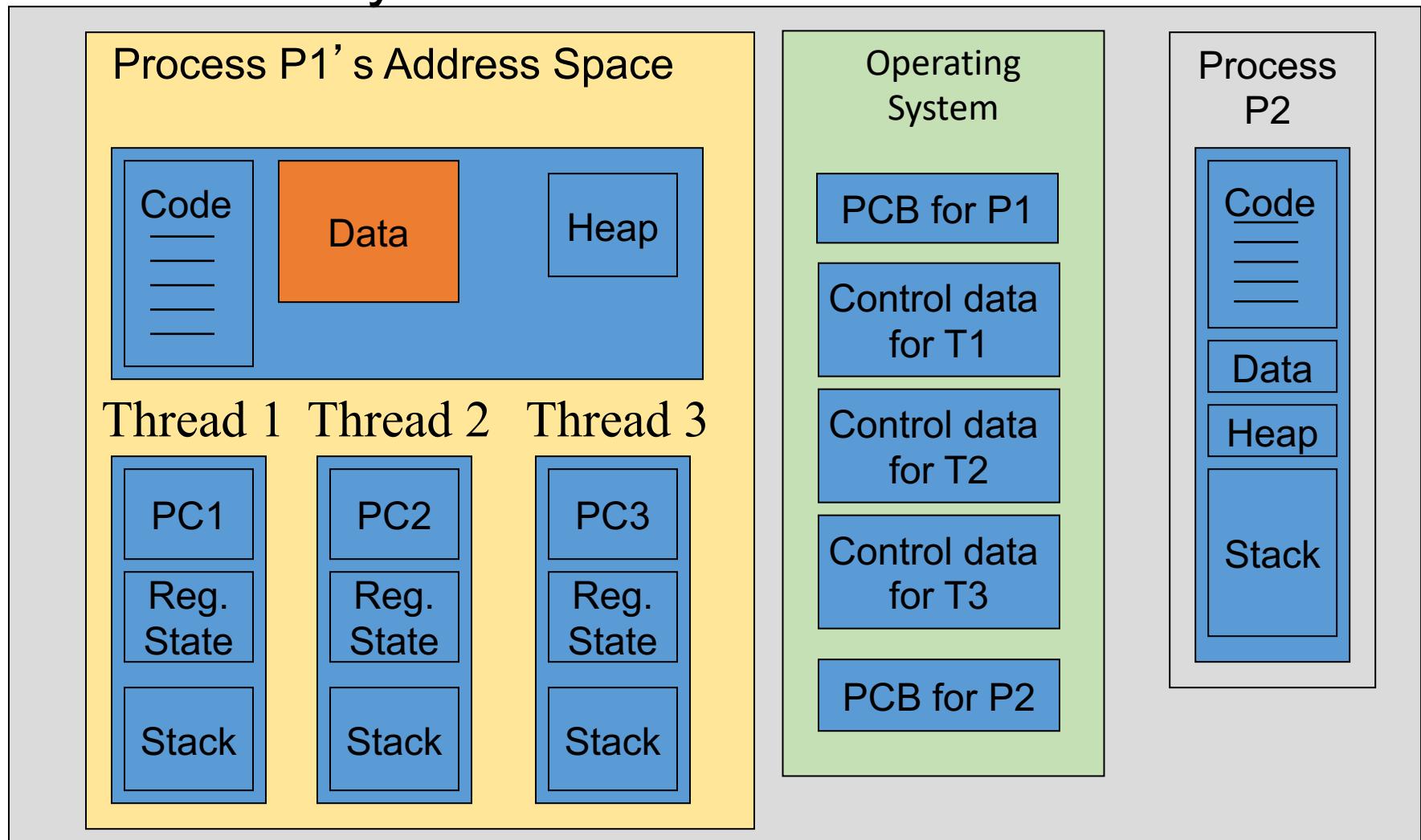
Threads

- 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)
 - A thread is also called a *lightweight process*
 - *Low overhead compared to a separate process*



Multiple Threads

Main Memory



Why do we want to use Threads

- Reduced context switch overhead vs multiple processes
 - E.g. 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
 - Don't duplicate code, data or heap or have multiple PCBs as for multiple processes
 - Supports more threads – more scalable, e.g. Web server must handle thousands of connections



More reasons for

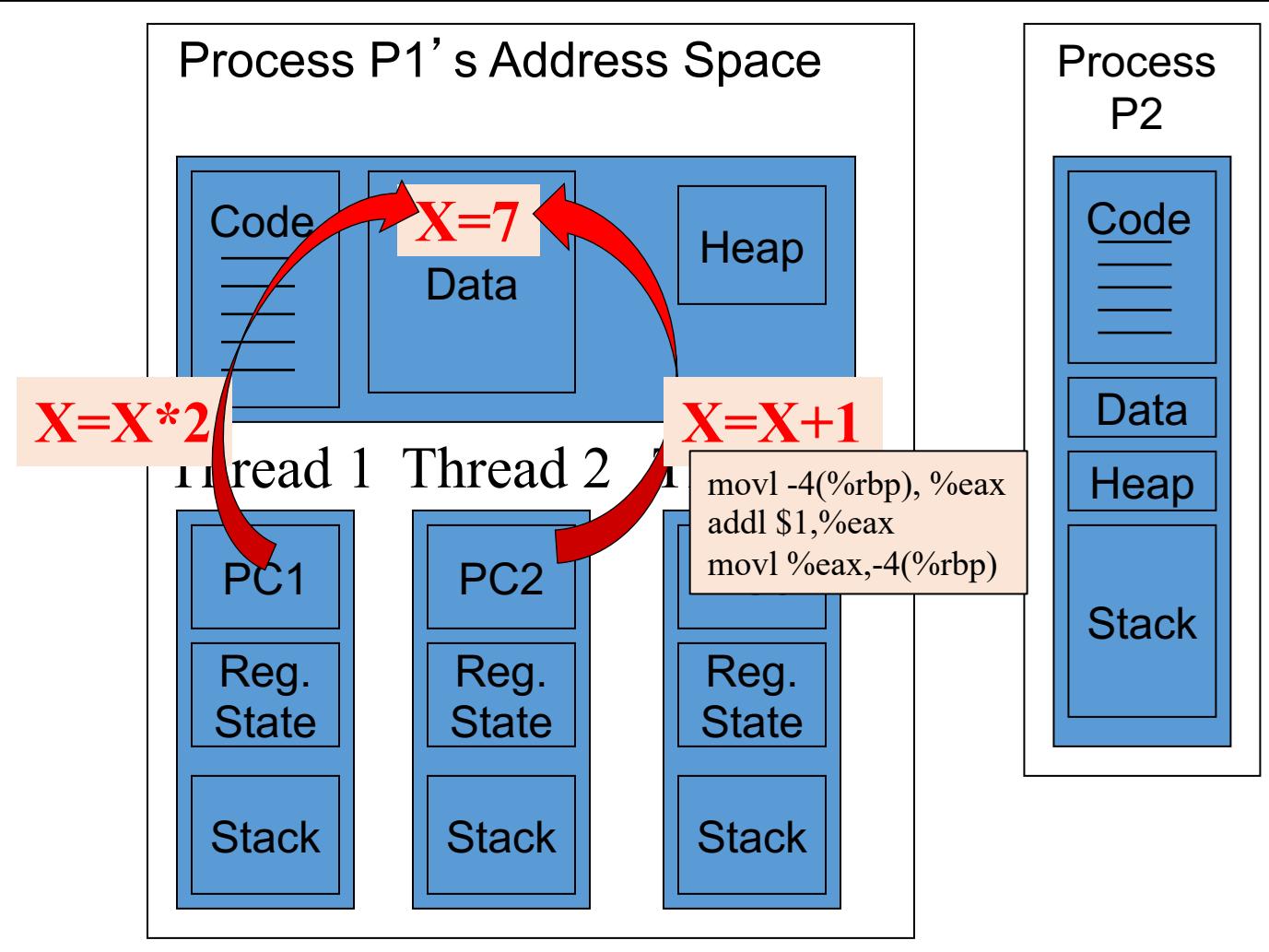
Why do we want to use Threads

- 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 !!!
 - IPC via message passing uses system calls to send/receive a message, which is slow
 - IPC using shared memory may be comparable to inter-thread communication



Thread Safety

Main Memory



Suppose:

- Thread1 wants to multiply X by 2
- Thread2 wants to increment X
- *Could have a race condition (see Chapter 5)*



Thread-Safe Code

- A piece of code is **thread-safe** if it functions correctly during simultaneous or *concurrent* execution by multiple threads.
 - In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.
- If two threads share and execute the same code, then unprotected use of shared
 - global variables is not thread safe
 - static variables is not thread safe
 - heap variables is not thread safe

We will learn how to write thread-safe code in Chapter 5

Processes vs. Threads

- Why are processes still used when threads bring so many advantages?
 1. Some tasks are sequential and not easily parallelizable, and hence are single-threaded by nature
 2. No fault isolation between threads
 - If a thread crashes, it can bring down other threads
 - If a process crashes, other processes continue to execute, because each process operates within its own address space, and so one crashing has limited effect on another
 - **Caveat:** a crashed process may fail to release synchronization locks, open files, etc., thus affecting other processes . But, the OS can use PCB's information to help cleanly recover from a crash and free up resources.



Processes vs. Threads (2)

- Why are processes still used when threads bring so many advantages? (cont.)
3. Writing thread-safe/reentrant code is difficult. Processes can avoid this by having separate address spaces and separate copies of the data and heap

Threads vs. Processes

- Advantages of multithreading
 - Sharing between threads is easy
 - Faster creation
- Disadvantages of multithreading
 - Ensure threads-safety
 - Bug in one thread can bleed to other threads, since they share the same address space
 - Threads must compete for memory
- Considerations
 - Dealing with signals in threads is tricky
 - All threads must run the same program
 - Sharing of files, users, etc



Applications, Processes, and Threads

- An application can consist of multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)
- Each process can consist of multiple threads
- An application could thus consist of many processes and threads



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS
- Three main thread libraries in use today:
 - POSIX Pthreads
 - Win32
 - Java



Thread Libraries

- Three main thread libraries in use today:

- **POSIX pthreads**

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library

- **Win32**

- Kernel-level library on Windows system

- **Java**

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS



The pthreads API

- ***Thread management:*** The first class of functions work directly on threads - creating, terminating, joining, etc.
- ***Semaphores:*** provide for create, destroy, wait, and post on semaphores.
- ***Mutexes:*** provide for creating, destroying, locking and unlocking mutexes.
- ***Condition variables:*** include functions to create, destroy, wait and signal based upon specified variable values.

Thread Creation

pthread_create (tid, attr, start_routine, arg)

- It returns the new thread ID via the *tid* argument.
- The *attr* parameter is used to set thread attributes, NULL for the default values.
- The *start_routine* is the C routine that the thread will execute once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.



Thread Termination and Join

`pthread_exit (value) ;`

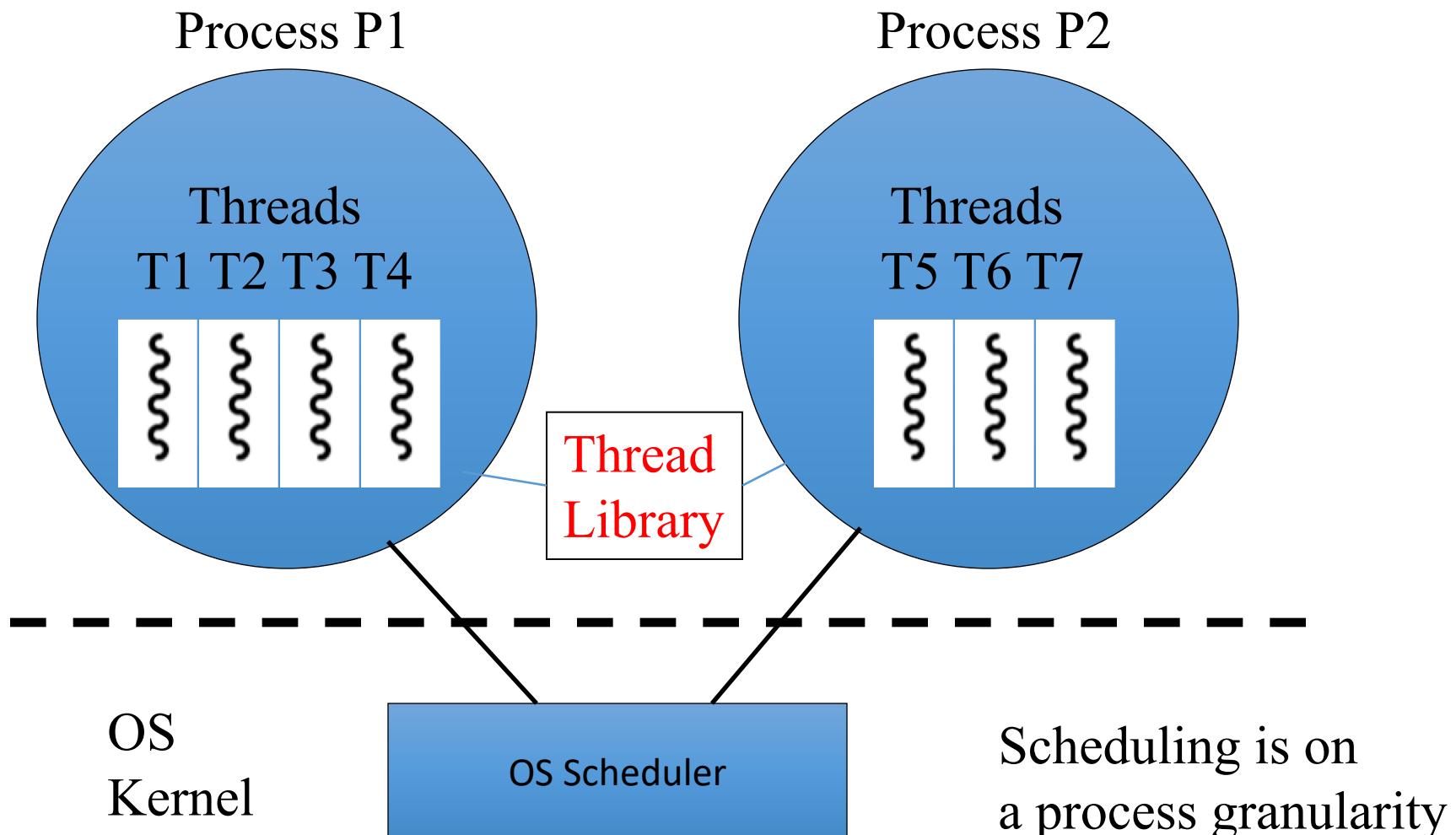
- This Function is used by a thread to terminate.
- The return value is passed as a pointer.

`pthread_join (tid, value_ptr);`

- The `pthread_join()` subroutine blocks the calling thread until the specified *threadid* thread terminates.
- Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.



User-Space Threads



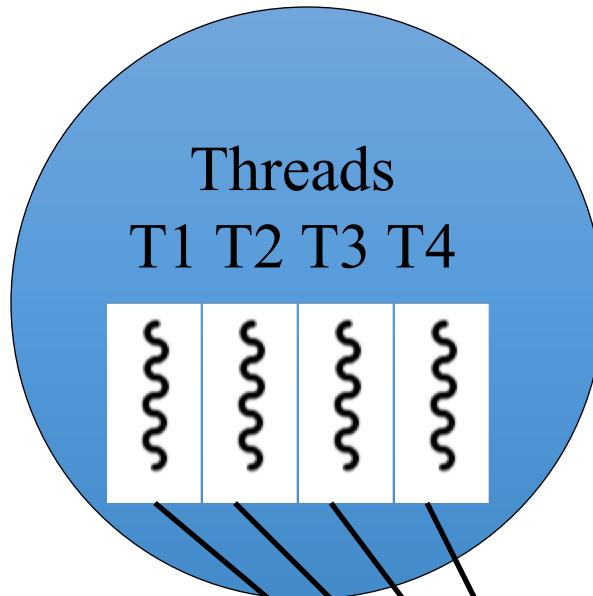
User-Space Threads

- *User space threads* are usually cooperatively multitasked, i.e. user threads within a process voluntarily give up the CPU to each other
 - threads will synchronize with each other via the user space threading package or library
 - Thread library: provides interface to create, delete threads in the same process
- OS is unaware of user-space threads – only sees user-space processes
 - If one user space thread blocks, the entire process blocks in a many-to-one scenario (see text)
- *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
- User space thread also called a *fiber*

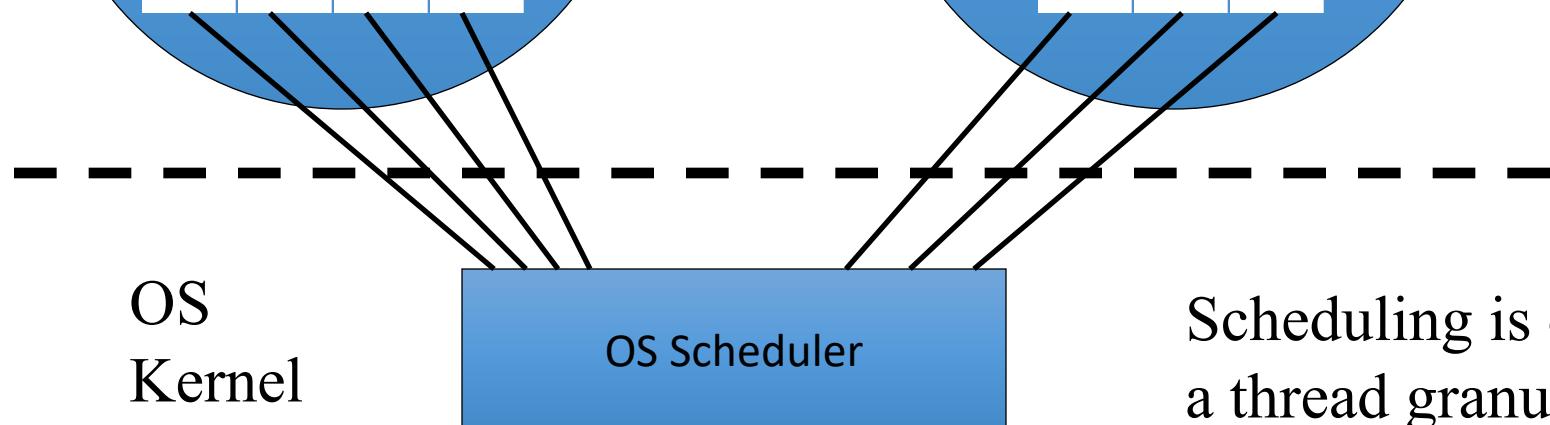
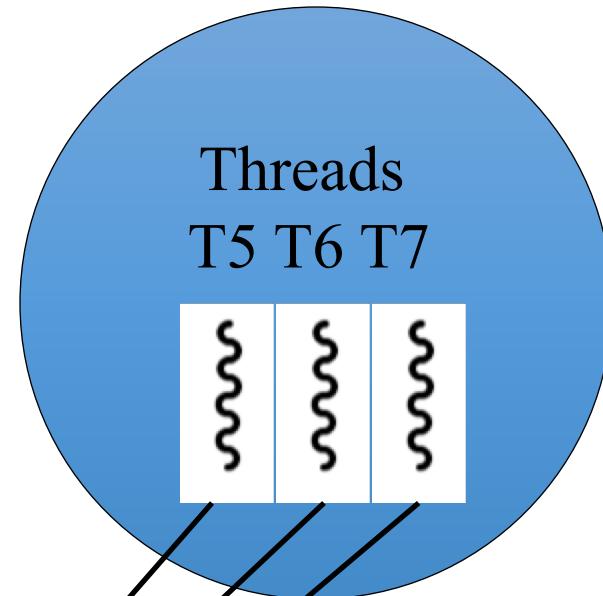


Kernel Threads

Process P1



Process P2

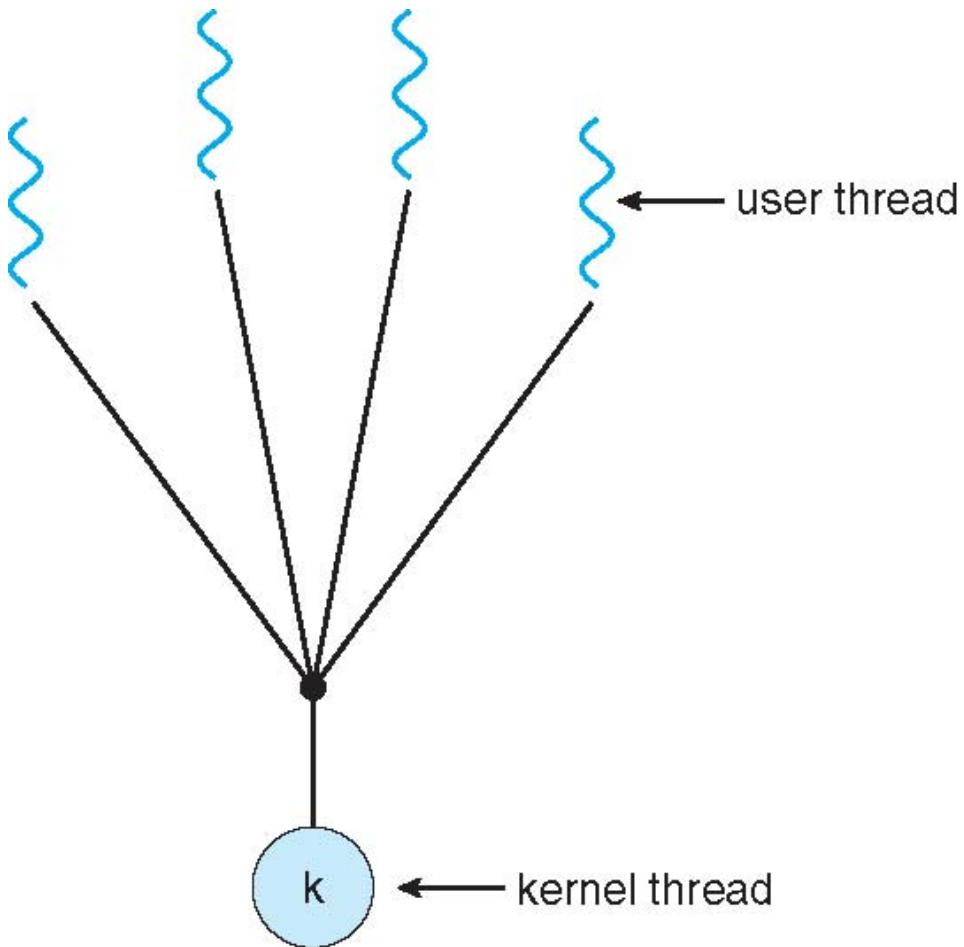


Kernel Threads

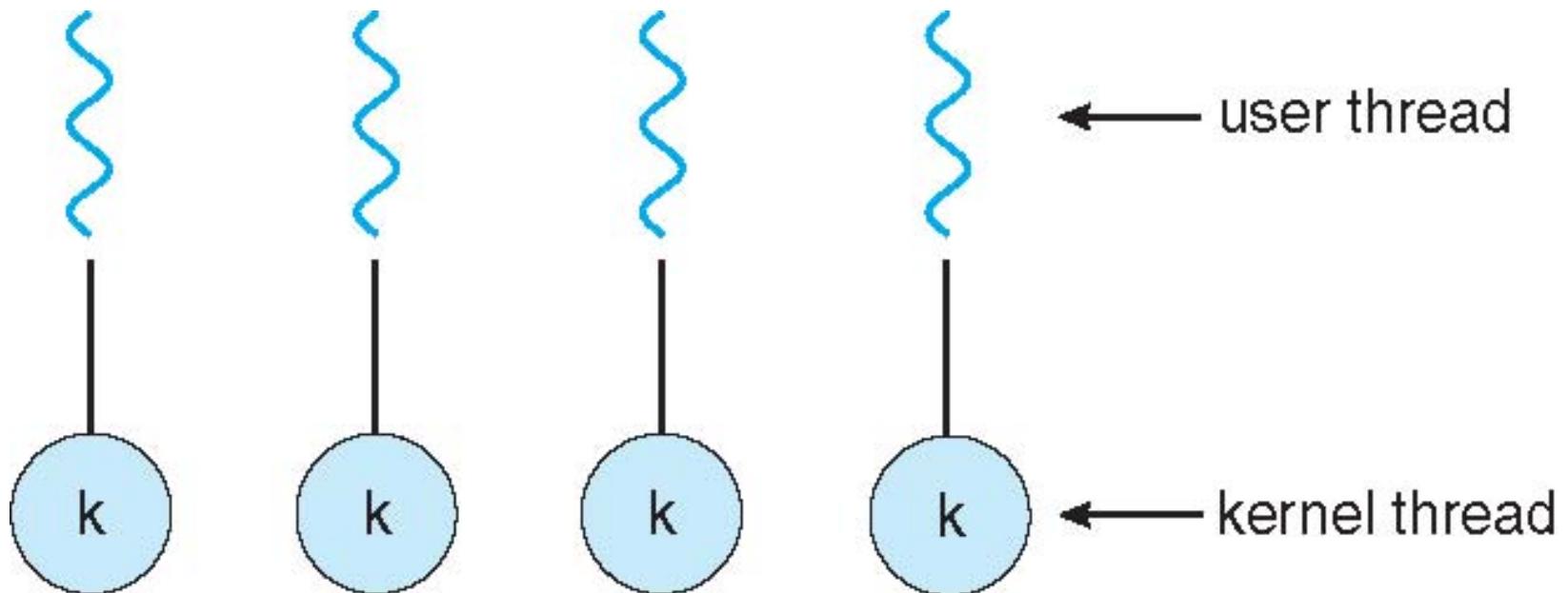
- *Kernel threads* are supported by the OS
 - kernel sees threads and schedules at the granularity of threads
 - Most modern OSs like Linux, Mac OS X, Win XP support kernel threads
 - Mapping of user-level threads to kernel threads is usually one-to-one, e.g. Linux and Windows, but could be many-to-one,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



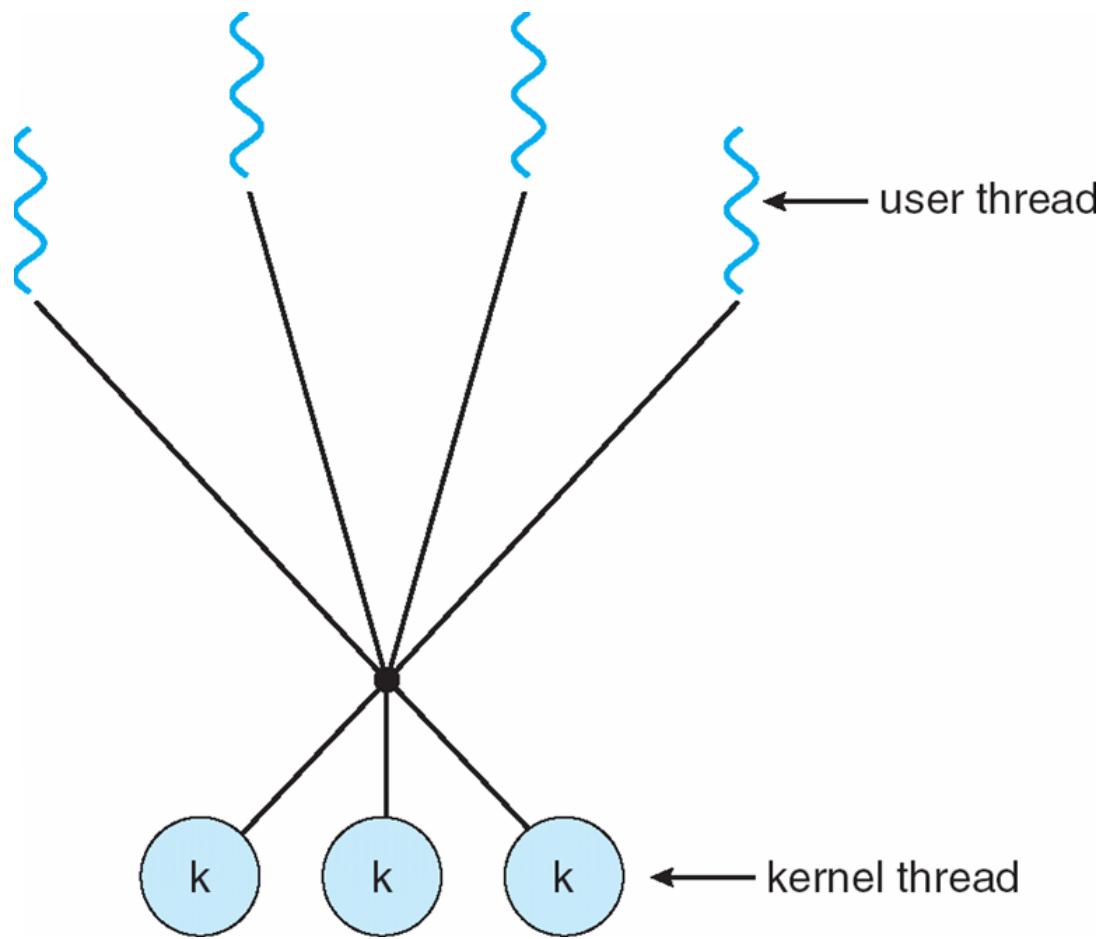
Many-to-One Model



One-to-one Model



Many-to-Many Model





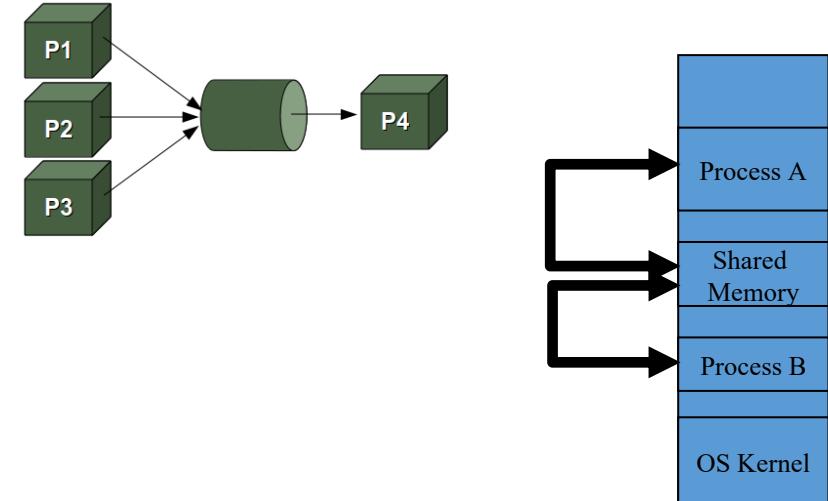
Inter-Process Communication

Inter-Process Communications (communications *between* processes)

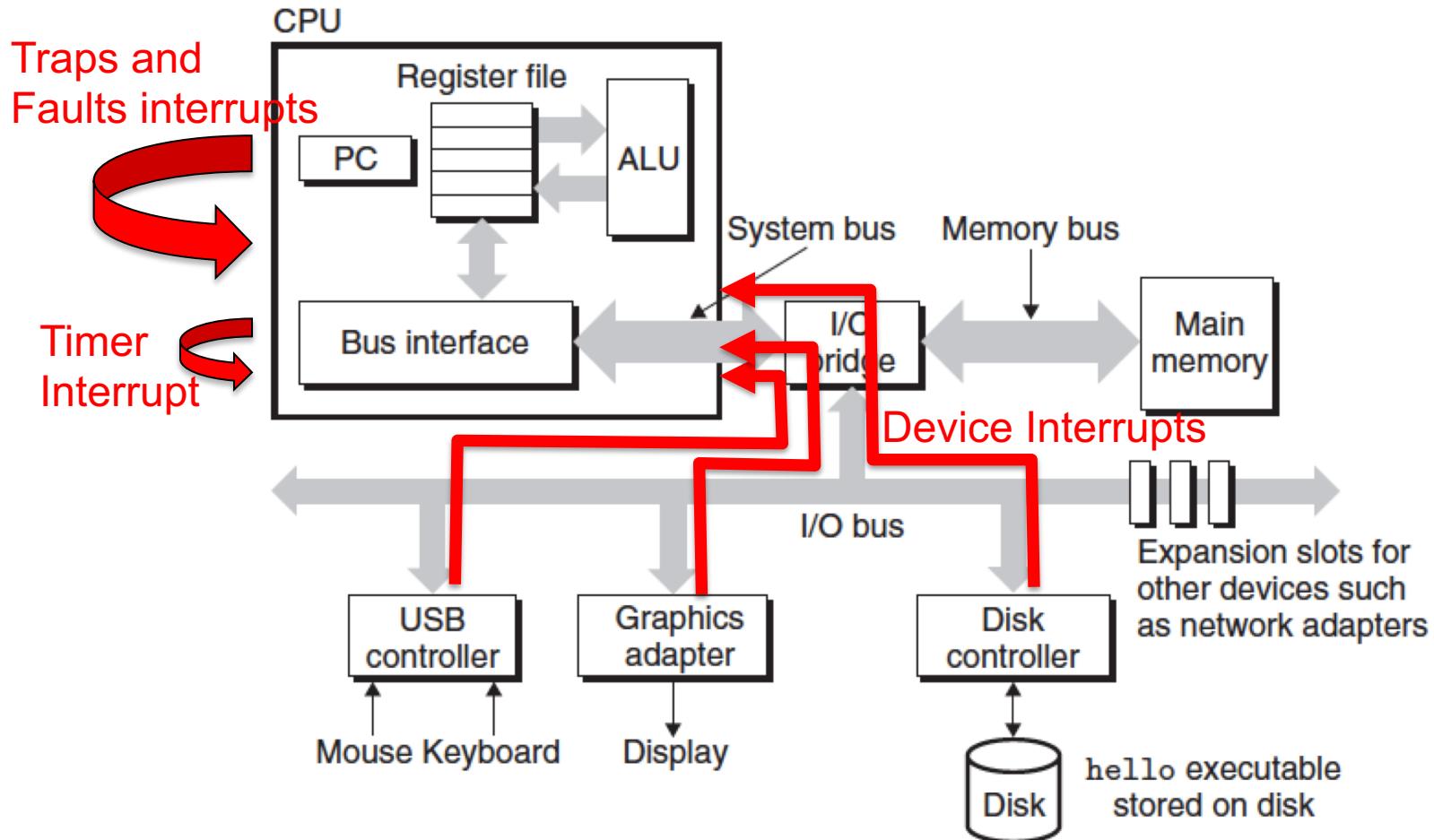
- Signals / Interrupts
 - Notifying that an event has occurred



- Message Passing
 - Pipes
 - Sockets
- Shared Memory
 - Race conditions
 - Synchronization
- Remote Procedure Calls



Linux Signals and Interrupts



Linux Signals and Interrupts

If a process **catches** a signal, it means that it includes code that will take appropriate action when the signal is received.

If the signal is not caught by the process, the kernel will take **default action** for the signal.

We will revisit these when we discuss process synchronization

Signal Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)

Signal Name	Number	Description
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)
SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

There is not much information that can be passed through an interrupt, only that an event has occurred

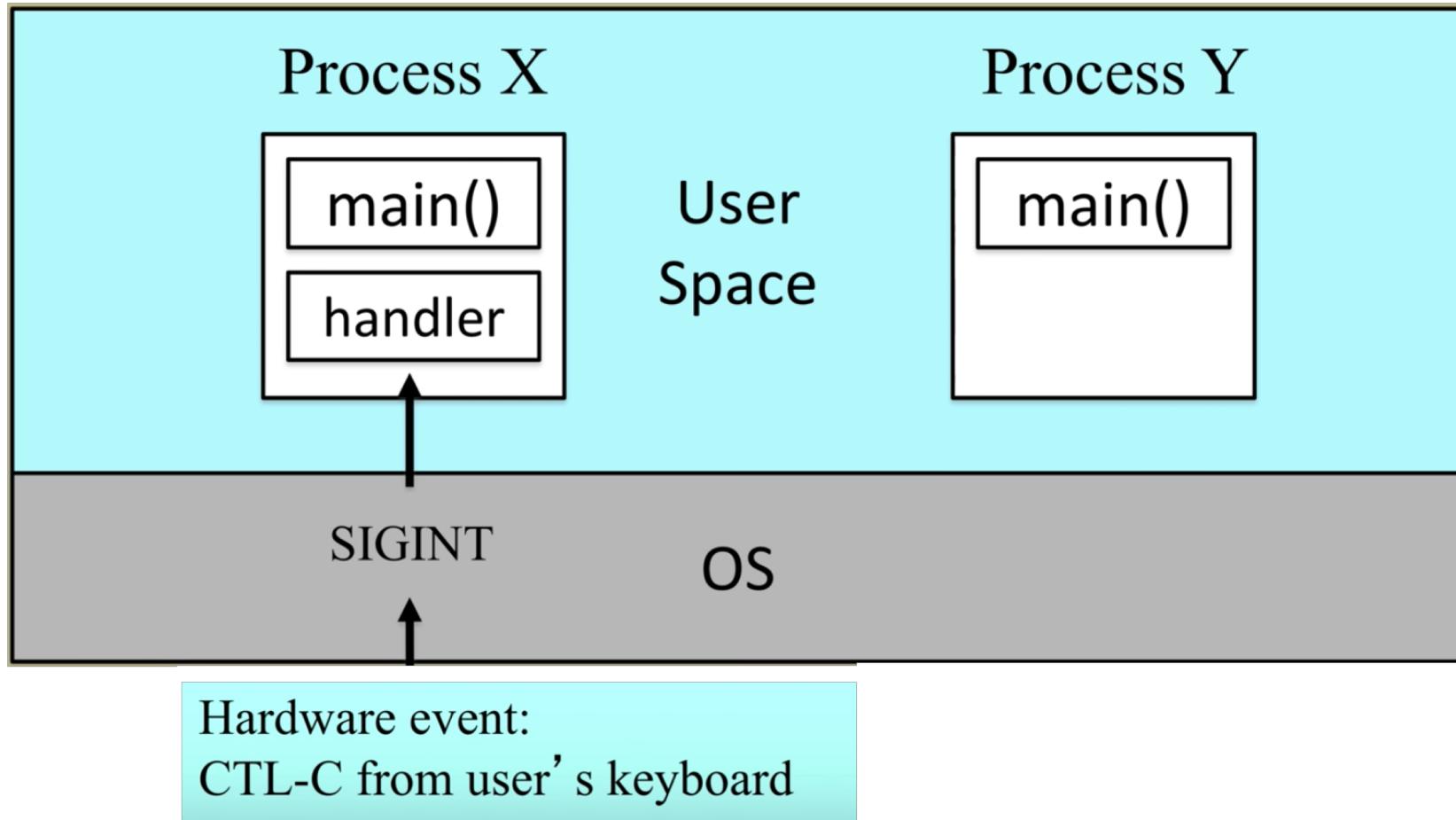


Commonly used Signals

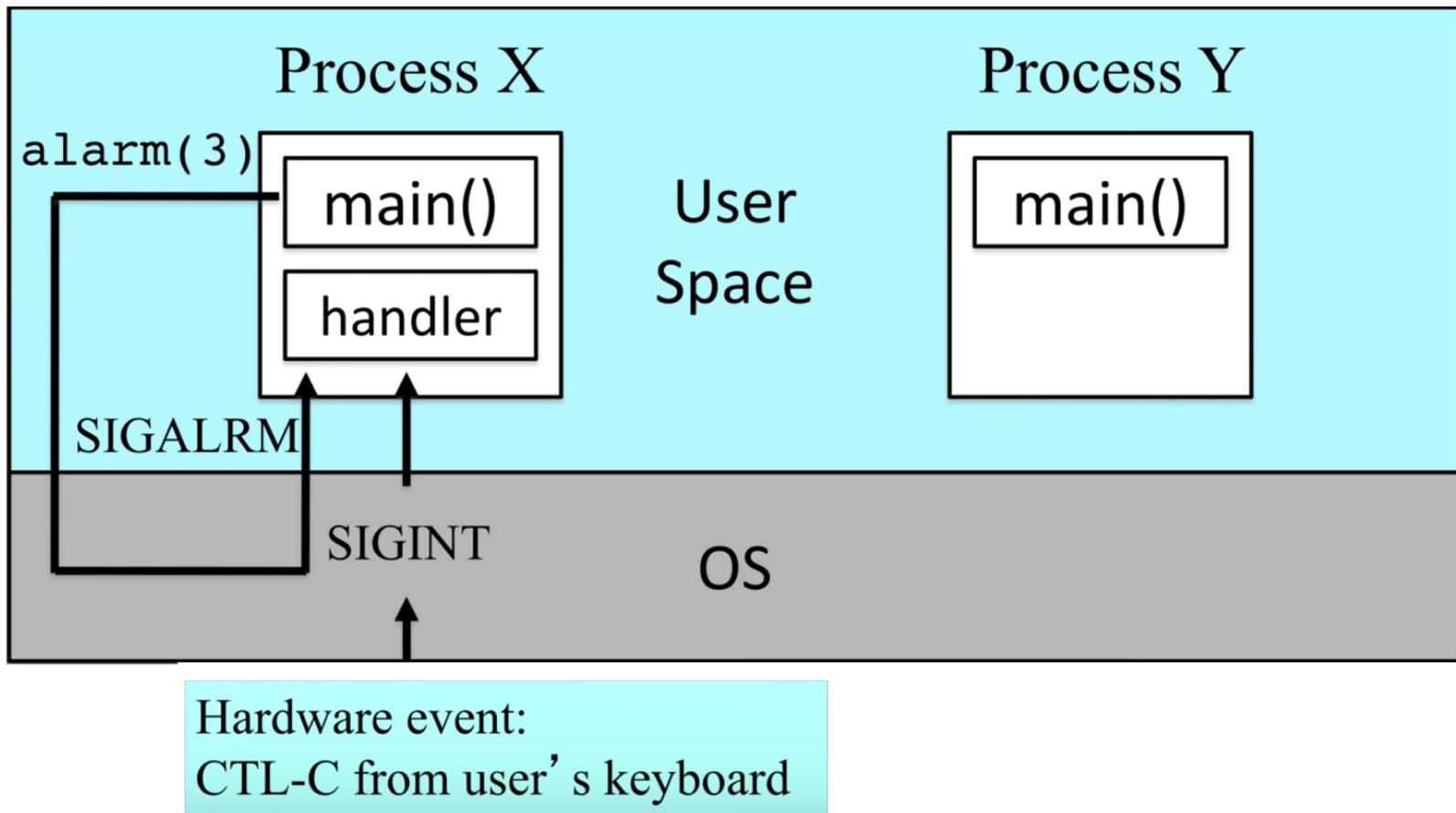
Number	Name/Type	Event
2	SIGINT	Interrupt from keyboard (Ctrl-C)
8	SIGFPE	Floating point exception (arith. error)
9	SIGKILL	Kill a process
10, 12	SIGUSR1, SIGUSR2	User-defined signals
11	SIGSEGV	invalid memory ref (seg fault)
14	SIGALRM	Timer signal from alarm function
29	SIGIO	I/O now possible on descriptor



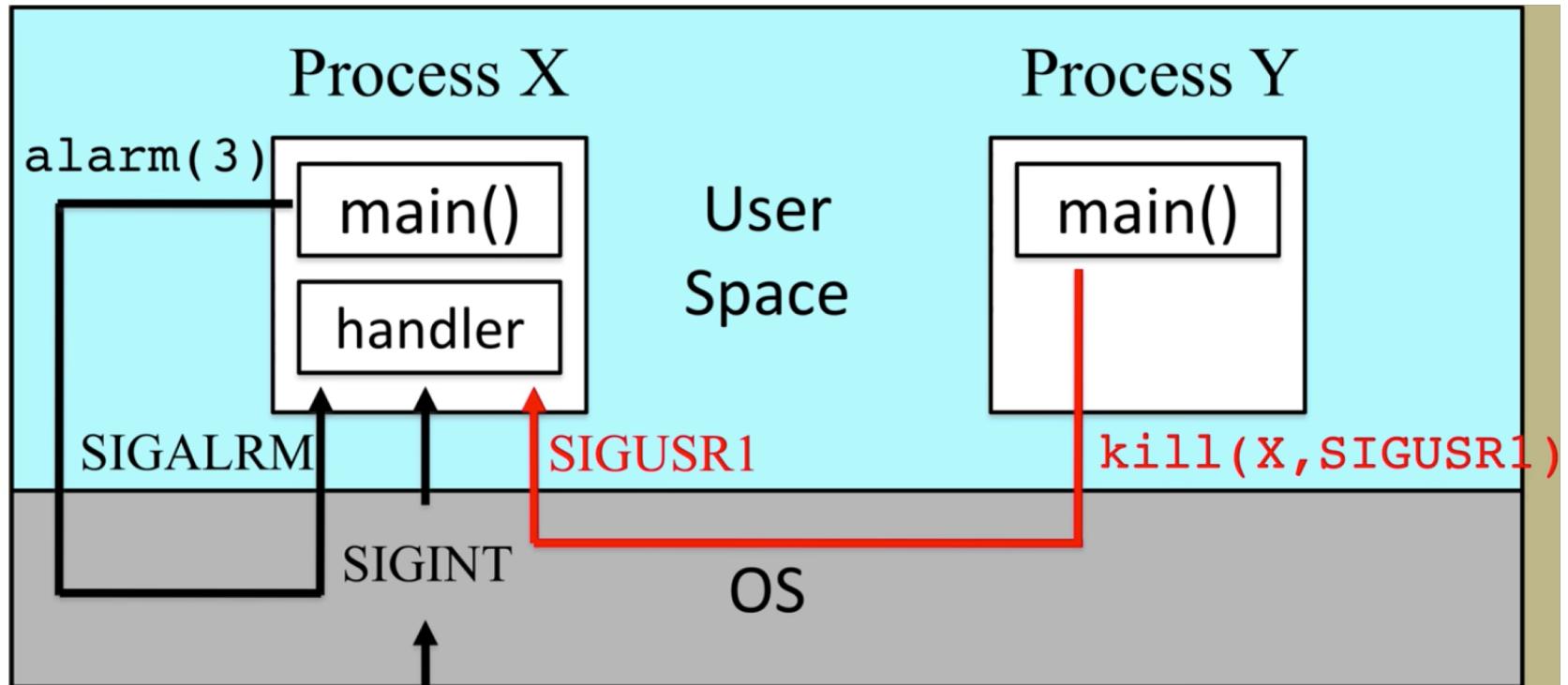
Signals allow limited IPC



Signals allow limited IPC



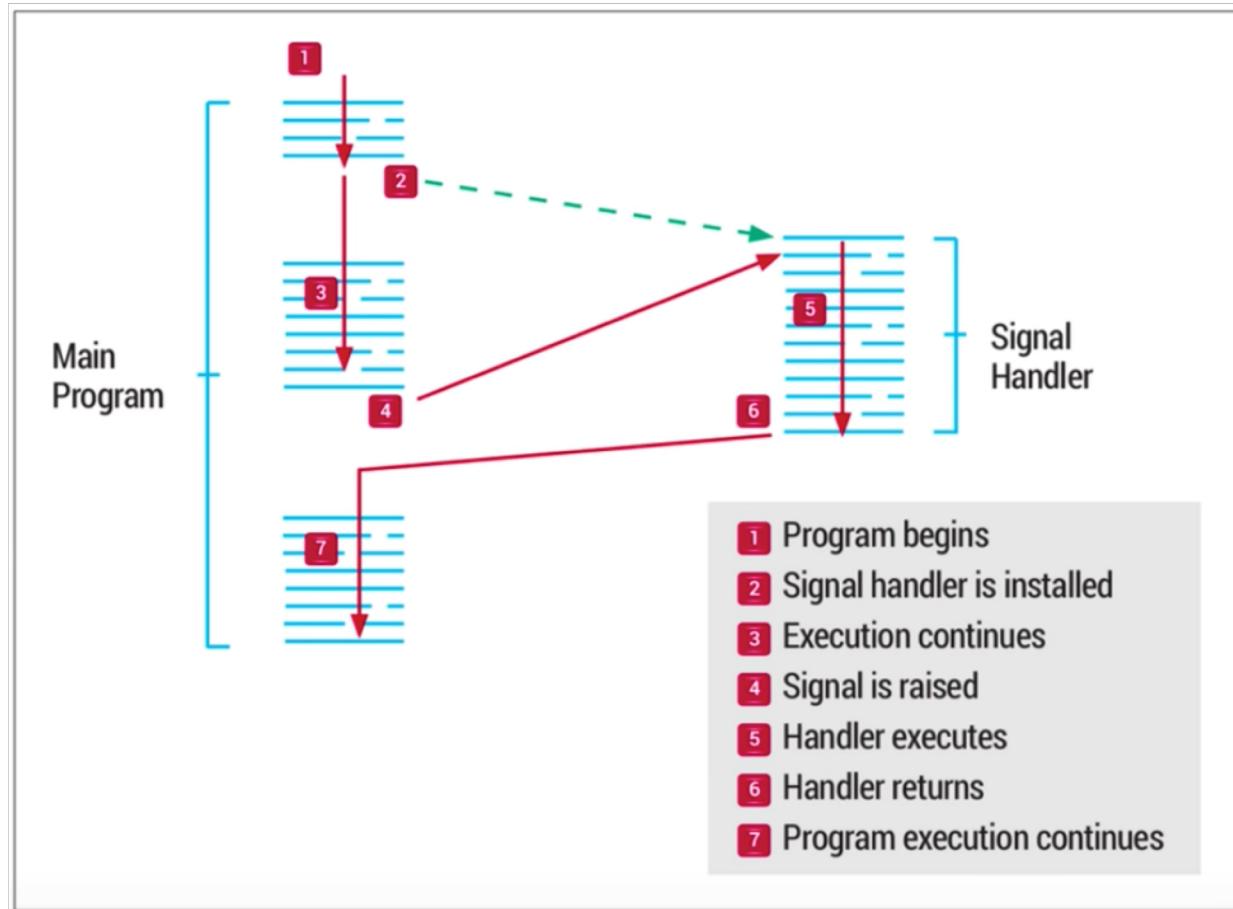
Signals allow limited IPC



Hardware event:
CTL-C from user's keyboard



Signal Handler



The handler resides in the same code base as the program.

It is usually a routine to be called when the signal is seen.



Signals

- Kernel-to-Process:
 - Kernel sets the numerical code in a process variable, then wakes the process up to handle the signal
- Process-to-Process
 - Call `kill(process_id, signal_num)`
 - e.g., `kill(Y, SIGUSR1)` sends a SIGUSR1 signal to process Y, which will know how to interpret this signal
 - Call still goes through kernel, not directly from process to process.



Signals

- Kernel-to-Process:
 - Kernel sets the numerical code in a process variable, then wakes the process up to handle the signal
- Process-to-Process
 - Call `kill(process_id, signal_num)`
 - e.g., `kill(Y, SIGUSR1)` sends a SIGUSR1 signal to process Y, which will know how to interpret this signal
 - Call still goes through kernel, not directly from process to process.
 - A process can send a signal to itself using a library call like `alarm()`



Signals and Race Conditions

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

```
int global=10;
handler(int signum) {
    global++;
}
main() {
    signal(SIGUSR1,handler);
    while(1) {global--;}
}
```



Blocking versus Non-Blocking I/O

- Blocking system call
 - process put on wait queue until I/O read or write completes
 - I/O command succeeds completely or fails
- Non-blocking system call
 - a write or read returns immediately with partial number of bytes transferred (possibly zero),
 - e.g. keyboard, mouse, network sockets
 - makes the application more complex
 - not all the data may have been read or written in single call
 - have to add additional code to handle this, like a loop



Synchronous versus Asynchronous

- Synchronous will make the request I/O and not continue until the command is completed
 - often synchronous and blocking are used interchangeably
- Asynchronous returns immediately (like non-blocking)
 - often asynchronous and non-blocking are used interchangeably
 - but in asynchronous write I/O, at some later time, the full number of bytes requested is transferred
 - subtle difference with non-blocking definition
 - can be implemented using signals and handlers



Signals and Race Conditions

- Multiple signals can also have a race condition
 - signal handler is processing signal S1 but is interrupted by another signal S2
 - The solution is to *block* other signals while handling the current signal.
 - Use `sigprocmask()` to selectively block other signals
 - A blocked signal is pending
 - There can be at most one pending signal per signal type, so signals are not queued



Signal Example

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

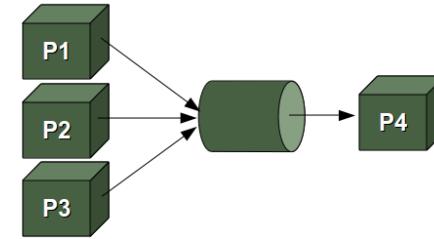
void handler(int sig) {
    if (beeps<5) {
        alarm(3);
        beeps++;
    } else {
        printf("DONE\n");
        exit(0);
    }
}

int main() {
    signal(SIGALRM, handler); ← Register signal handler
    alarm(3); ← Cause first SIGALRM to be sent
    while(1) { ; } ← Infinite loop that get interrupted by signal
    exit(0); ← handling
}
```

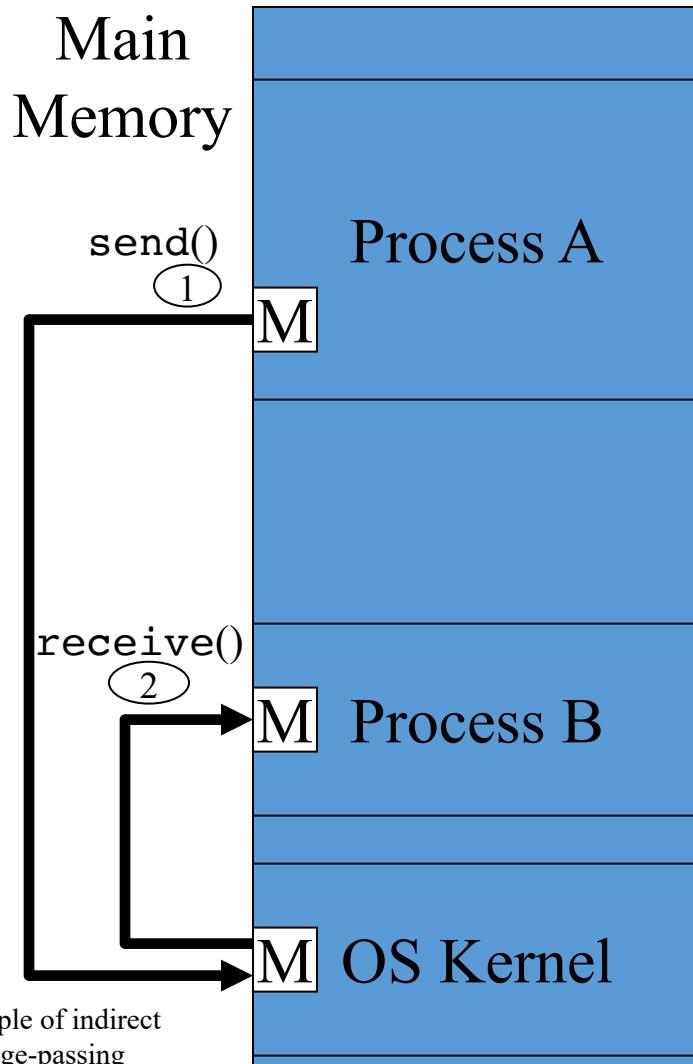


Inter-Process Communications (communications *between* processes)

- Signals
- Message Passing
 - Pipes
 - Sockets



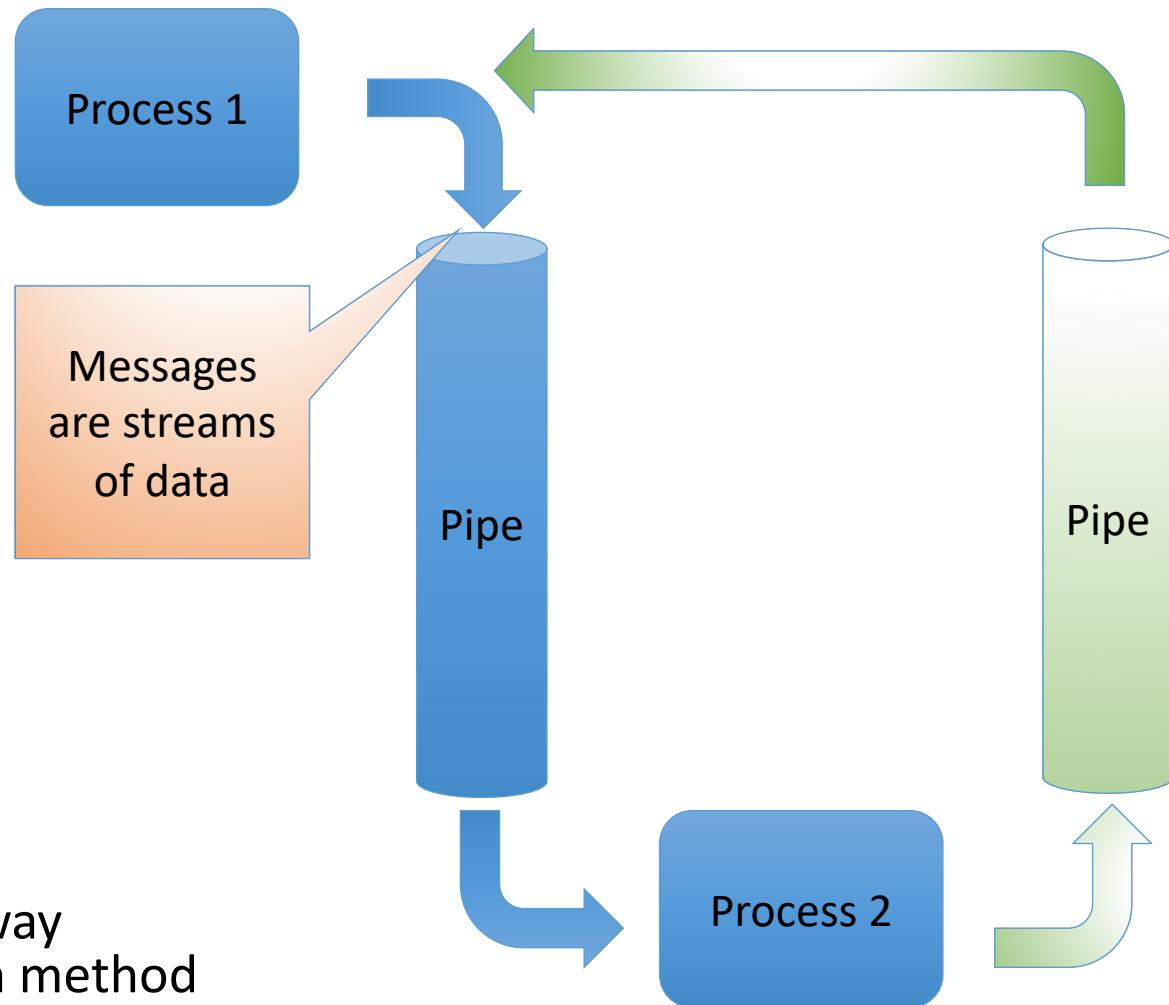
IPC Message Passing



- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- Used to pass small messages
- Advantage: OS handles 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
 - remote procedure calls (RPC)



IPC via Pipes

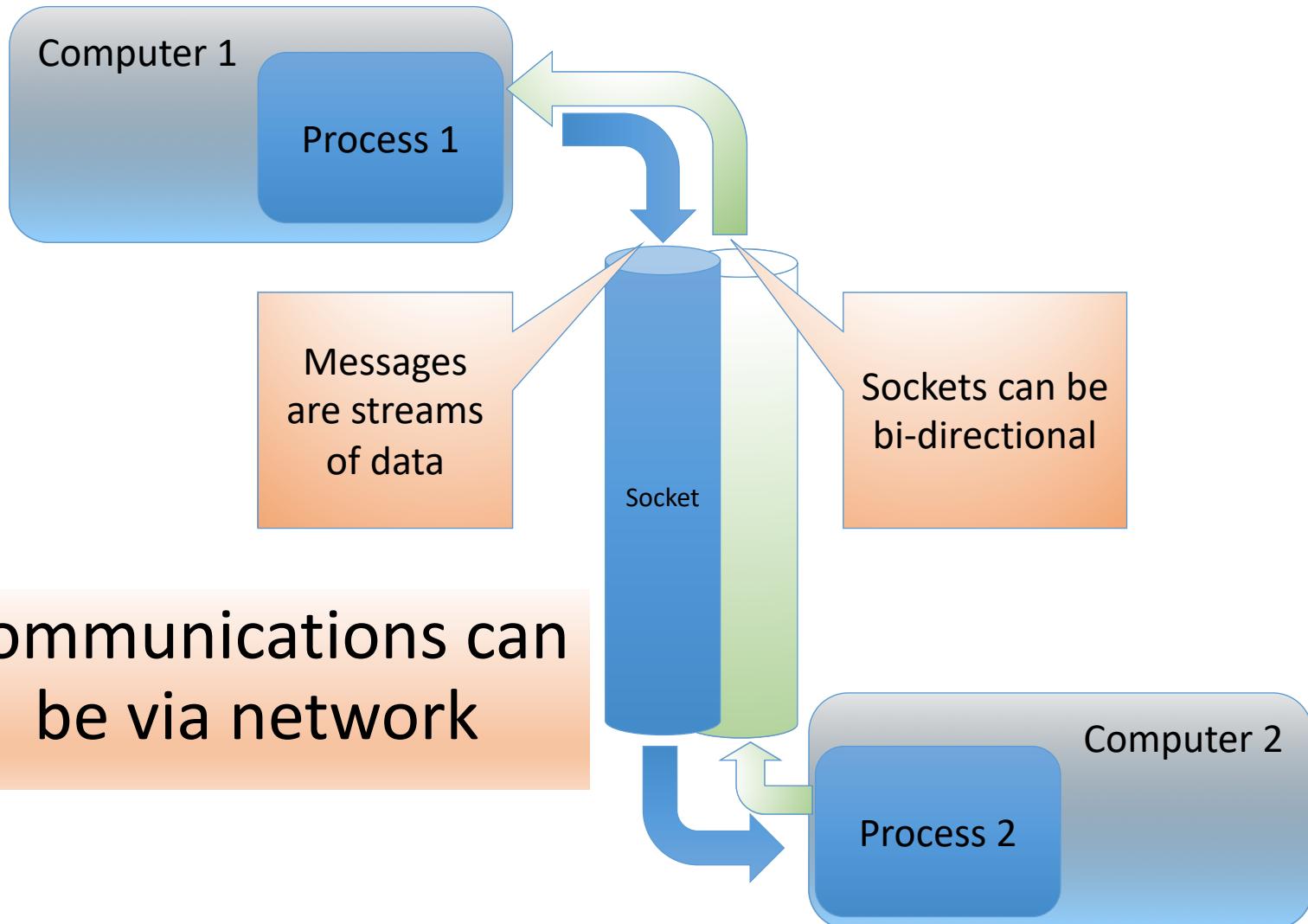


Named Pipes

- 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



IPC via Sockets



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 sockets
PF_INET for remote sockets

0 to select default protocol
associated with a type

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



Using Sockets for UNIX IPC (2)

- 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/dave/my_ipc_socket_file"



Using Sockets for UNIX IPC (3)

- 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,...)
```

```
listen() // for connect requests
```

```
sd2 = accept() // a connect request
```

```
recv(sd2,...)/send(sd2,...)
```

Client code:

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

```
connect(sd,...) to server
```

```
recv(sd,...)/send(sd,...)
```

bind and connect must
use same file name!

IPC



University of Colorado
Boulder

Inter-Process Communications (communications *between* processes)

- Signals
- Message Passing
 - Pipes
 - Sockets
- Remote Procedure Calls

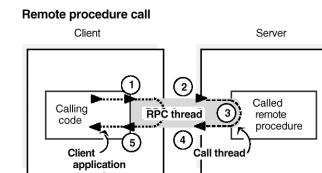
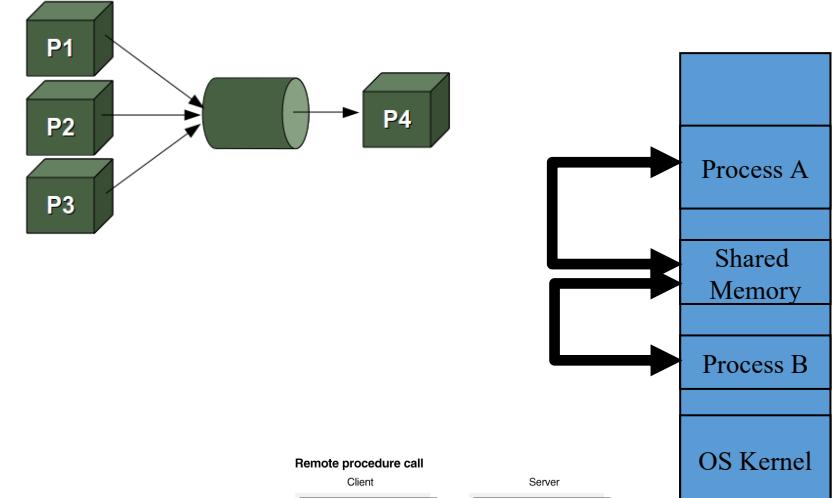
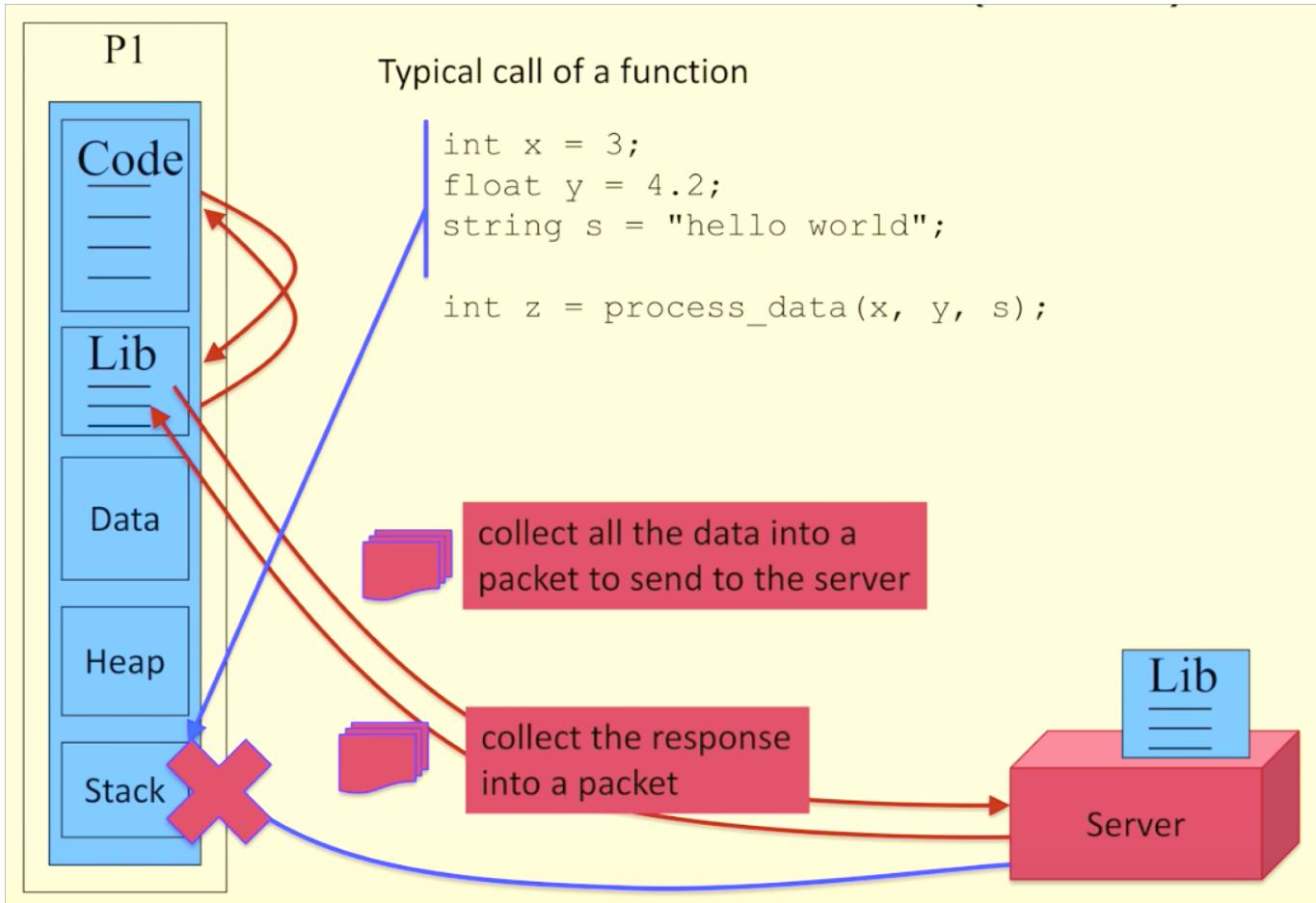


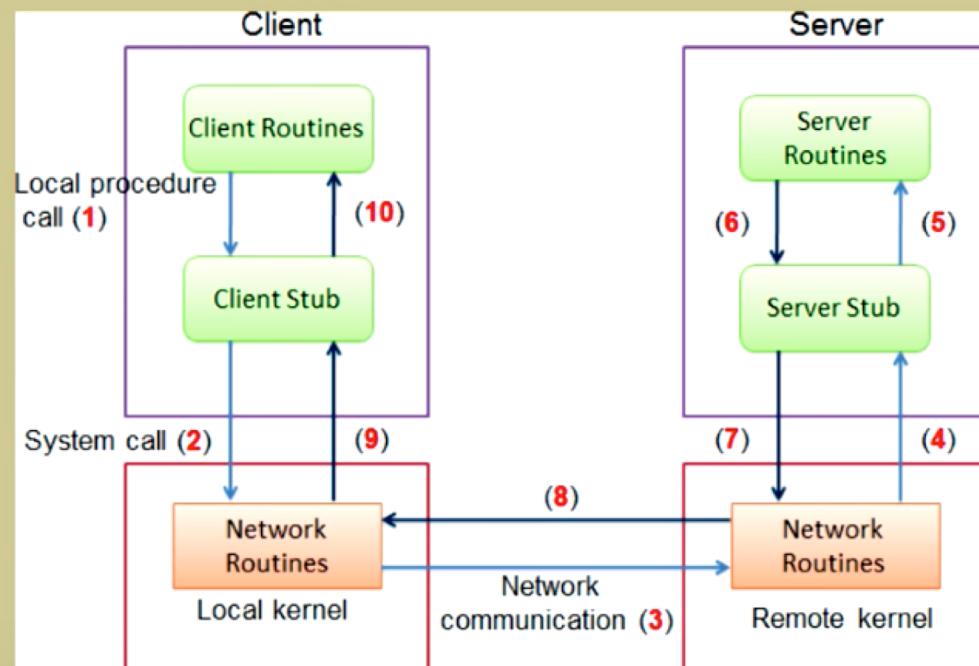
Figure 6-1 Execution Phases of an RPC Thread

Remote Procedure Call (RPC)



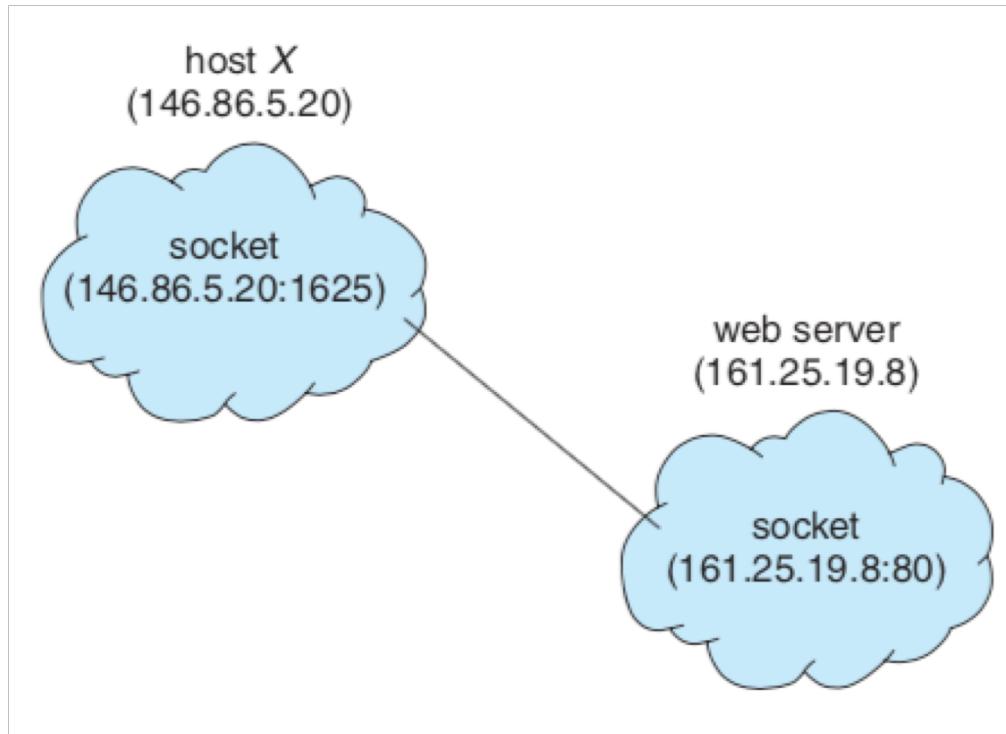
Remote Procedure Call (RPC)

1. Client makes a call to a function and passes parameters
2. The client has linked a stub for the function. This stub will marshal (packetize) the data and send it to a remote server
3. The network transfers the information to a server
4. A service listening to the network receives a request
5. The information is unmarshalled (unpacked) and the server's function is called
6. The results are returned to be marshalled into a packet
7. The network transfers the packet is sent back to the client
8. TCP/IP used to transmit packet



Example: Client-Server system

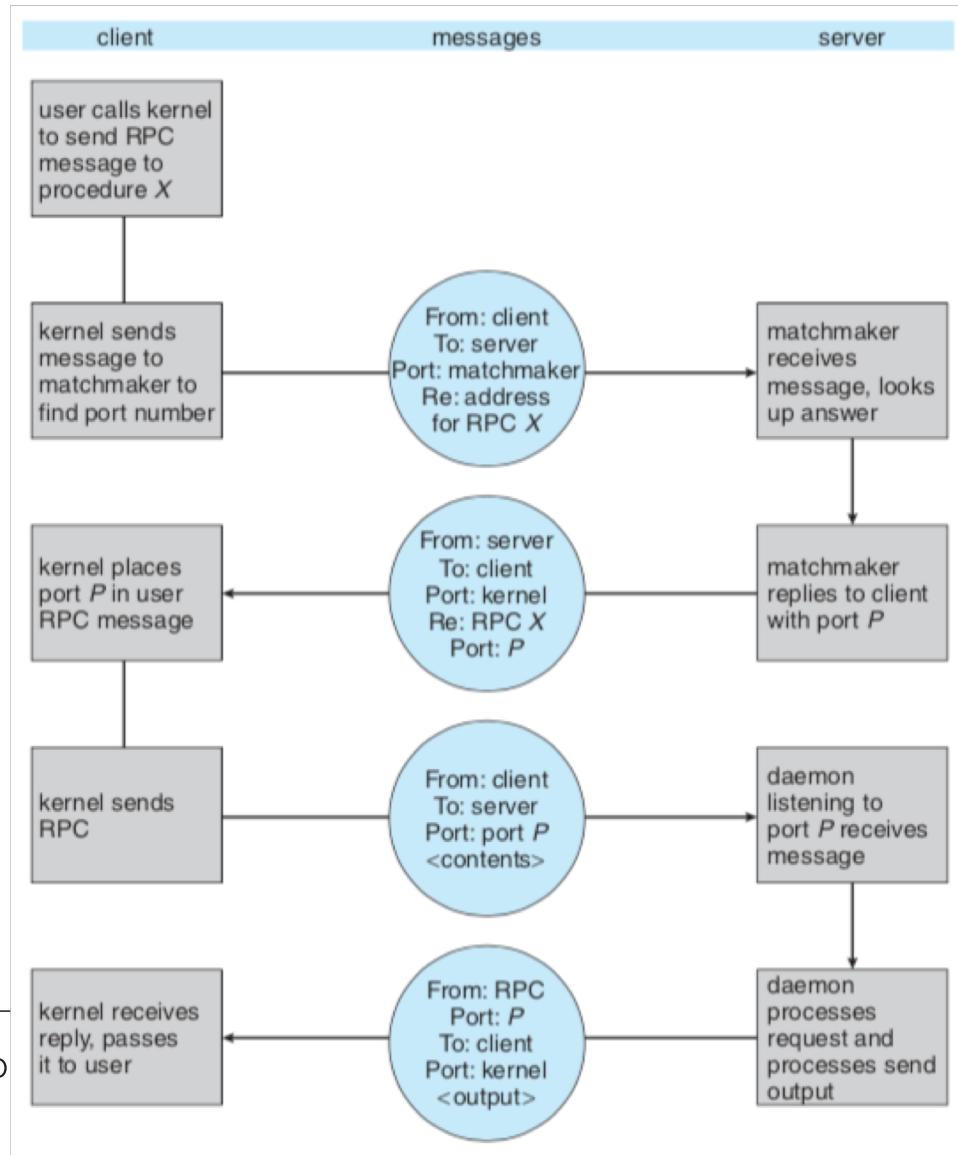
- Web server



Remote Procedure Call (RPC)

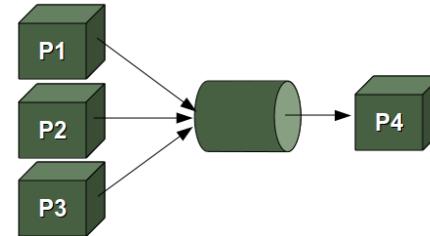
- Can fail more often
- Can be executed more than once
 - Exactly once vs. at most once
 - Timestamped message record is needed for “at most once”
 - An ACK protocol is needed for “exactly once”
- Remote port selection
 - Static port => Pre-selected
 - Dynamic port => matchmaker is needed

Remote Procedure Call (RPC)



Inter-Process Communications (communications *between* processes)

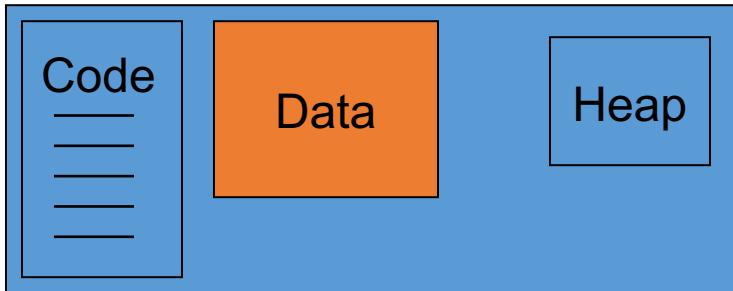
- Signals
- Message Passing
 - Pipes
 - Sockets
- Remote Procedure Calls
- Shared Memory



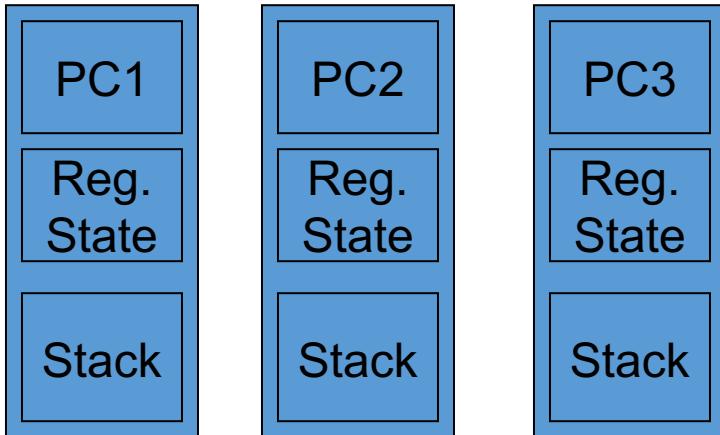
Multiple Threads

Main Memory

Process P1's Address Space



Thread 1 Thread 2 Thread 3



- Processes have separate Code, Data, Stack, & Heap
- Threads SHARE Code, Data, & Heap
- Threads have separate Stack
- Can processes share more?

Process P2



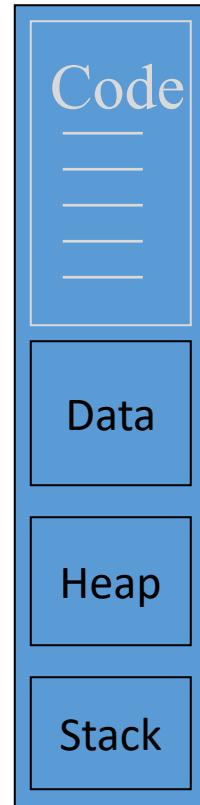
Duplication of a Process

Process
P1



- When process 2 is a fork() of process 1, what actually needs to be copied?
- Code is identical – no need to copy, just share
- Data is the same – just share??
- Okay to share until data changes
- "Copy on write" – will share until changed and then create its own copy

Process
P2



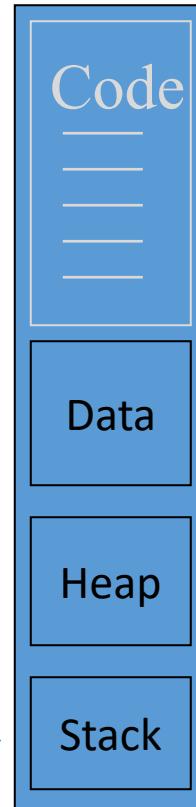
Sharing Data with Multiple Processes

Process
P1



- What if we want to share some information?
- Pipes
- Sockets
- Send() and receive()
- All messages passing needs to copy data from process to kernel and copy data from kernel to other processes
- Can we make it faster?
- Share some memory

Process
P2

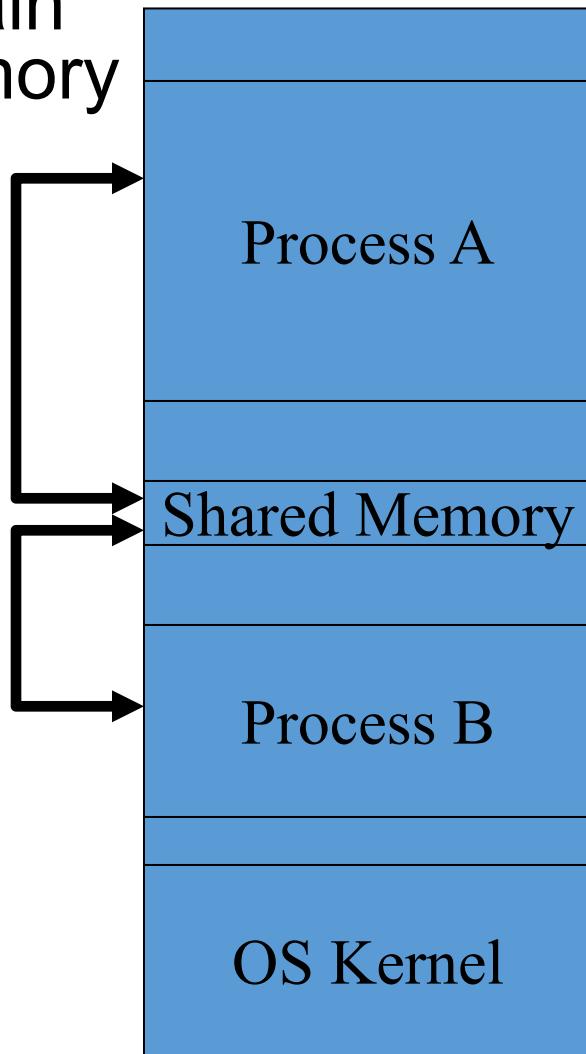


Share Memory



IPC Shared Memory

Main
Memory



- OS provides mechanisms for creation of a shared memory buffer between processes (both processes have address mapped to their process)
- Applies to processes on the same machine
- Problem: shared access introduces complexity
 - need to synchronize access



IPC Shared Memory (Linux)

- `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.

IPC Shared Memory (Linux)

- Attach a shared memory segment to a process address space
 - `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

Details about Linux support for shared memory IPC will be covered in recitation

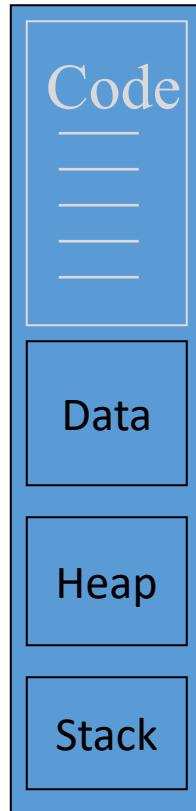
Multiple Threads

Process
P1



- What problems can occur when sharing data?
- Race Conditions
- Requires Reentrant Code
- Requires Thread Process Safe Code
- The use of shared memory or global data requires that the code be synchronized to limit the possible interruptions while using the shared data

Process
P2



Share Memory



Summary

- Definition of process and its states
- How processes are managed by the OS (PCB)
- Mechanisms for processes to be created, executed, and terminated
- Cooperating processes needs IPC
- IPC includes
 - Signaling/interups
 - Memory sharing
 - Message passing
 - Remote procedure call
- Smaller operating unit of OS is threads
 - Lighter weight unit with faster context switching
 - More overhead in managing them by the application programmer

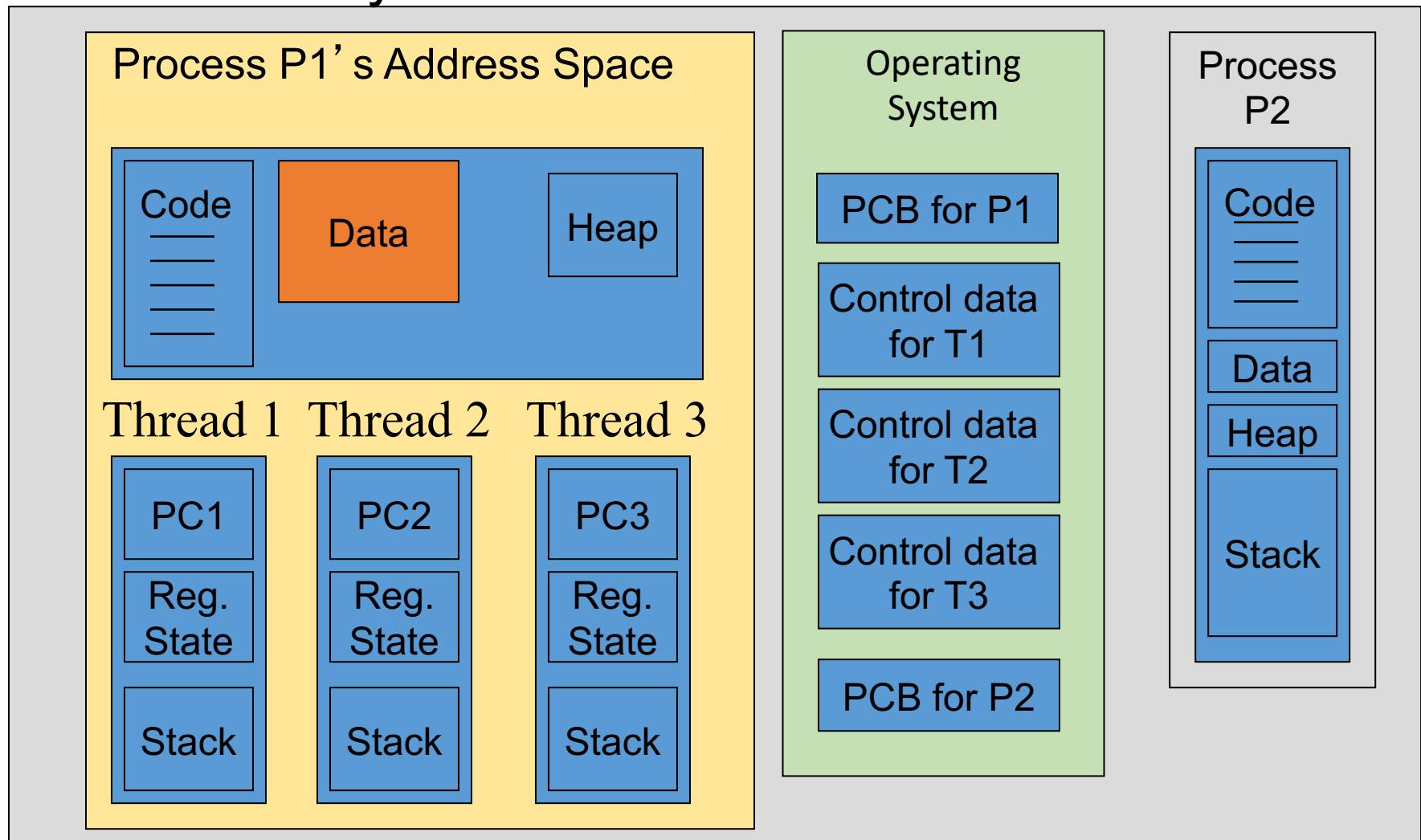




Threads

Multiple Threads

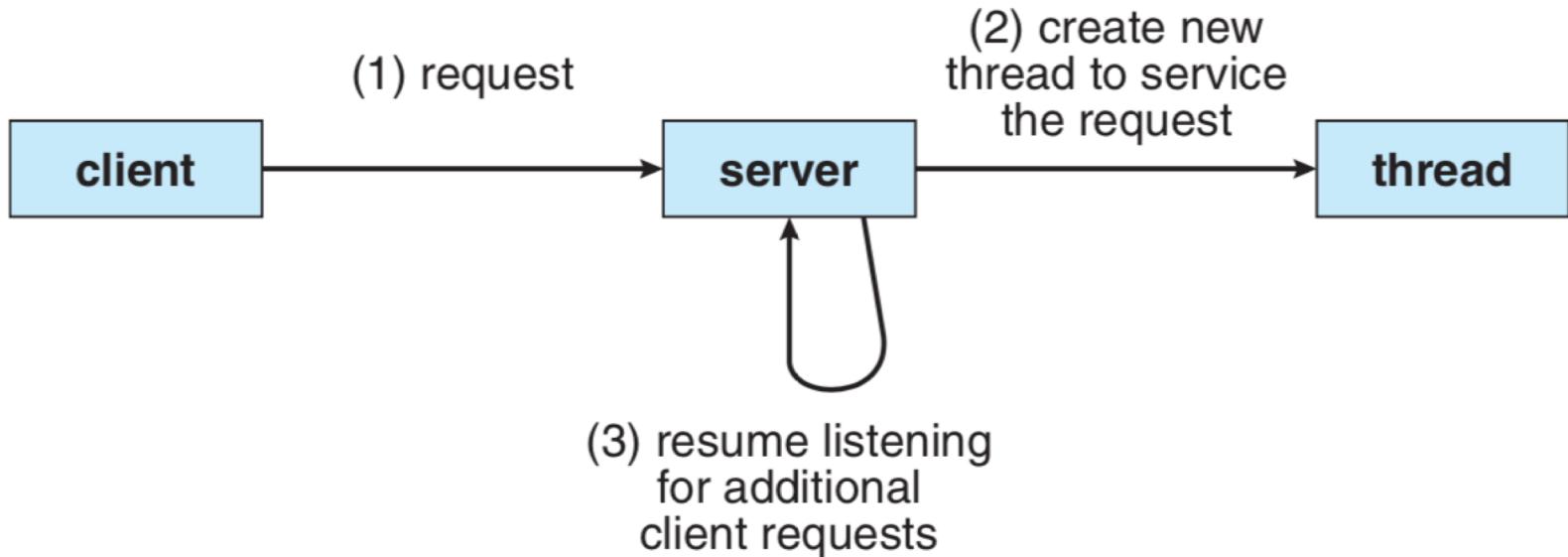
Main Memory



Benefits of multithreaded architecture

- Responsiveness.
 - Useful for interactive applications
- Resource sharing
 - Memory and resources are shared within process
- Context-switching overhead
 - Low creating and managing overhead
- Scalability
 - Threads can run in **parallel** with multicore system

Example.



The benefits often come with cost

- Who pay?
Programmer/system designer
- Identifying tasks
 - Dividing tasks into concurrent subtasks
- Balance
 - Balance the load on different cores
- Splitting data
 - Divide data to be access by different cores
- Identifying data dependency
 - Identifying synchronization mechanisms
- Testing and debugging
 - Different path of executions need to be considered



Thread Safety

Thread Safety

- Thread Safe

If the code behaves correctly during **simultaneous** or **concurrent** execution by multiple threads

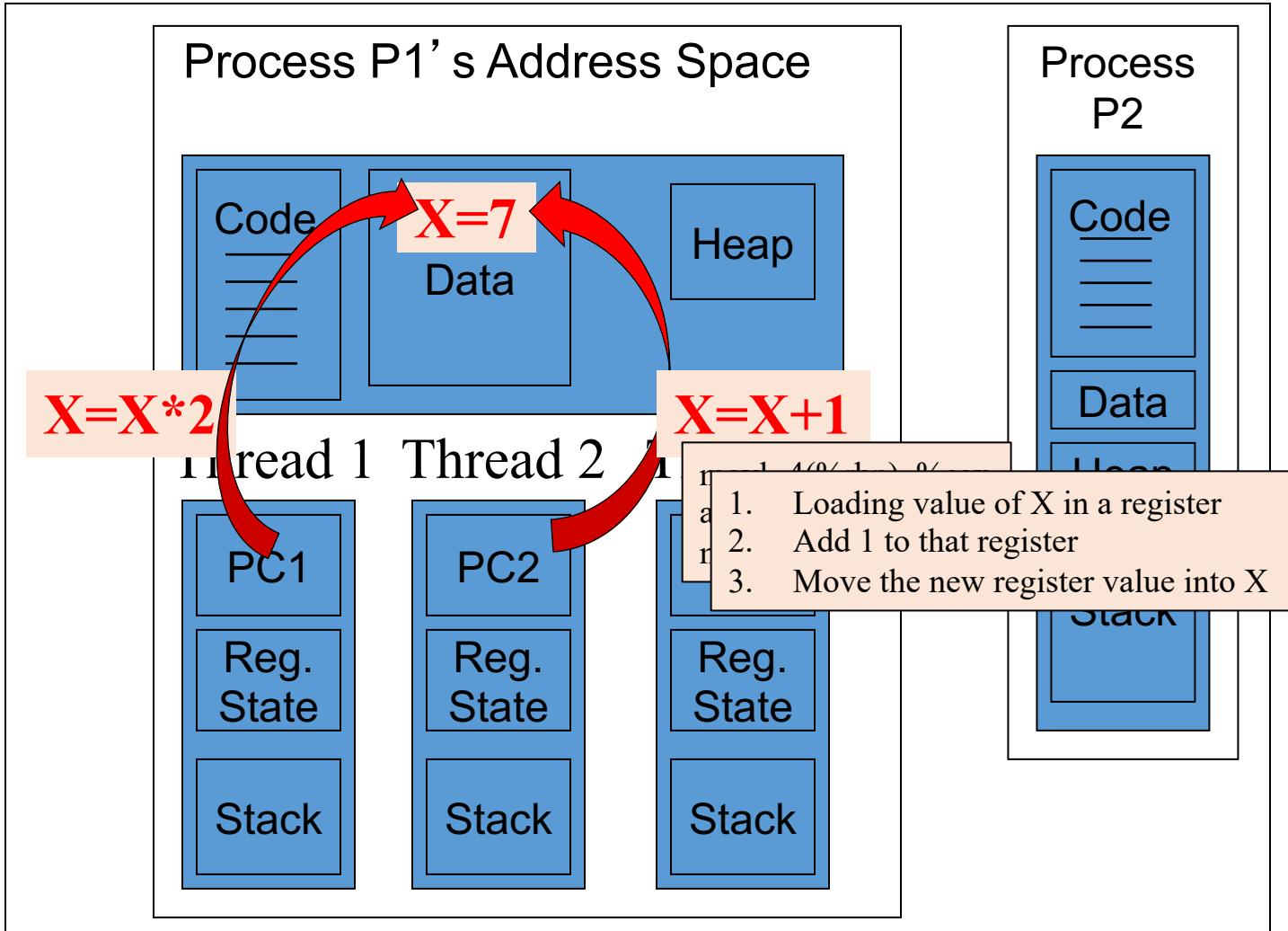
- Reentrant

- If the code behaves correctly when a **single** thread is interrupted in the middle of executing the code reenters the same code



Thread Safety

Main Memory



Suppose:

- Thread1 wants to multiply X by 2

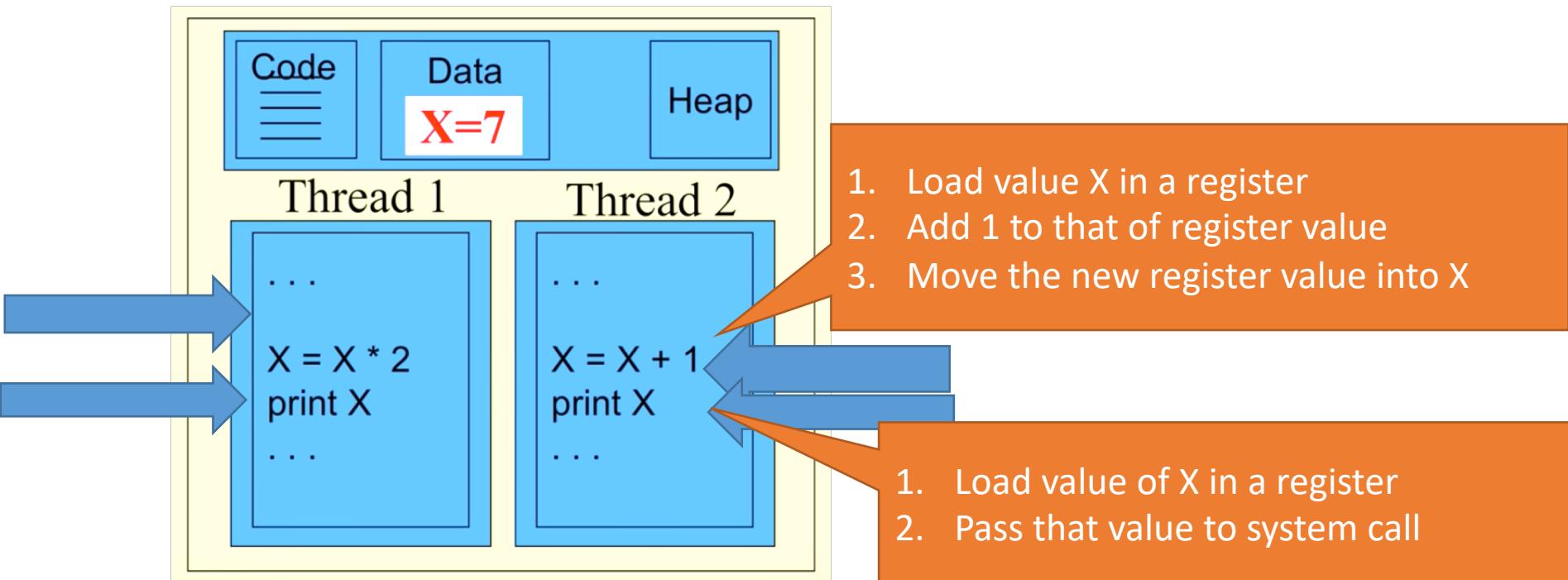
Thread2 wants to increment X

- Could have a ***race condition***



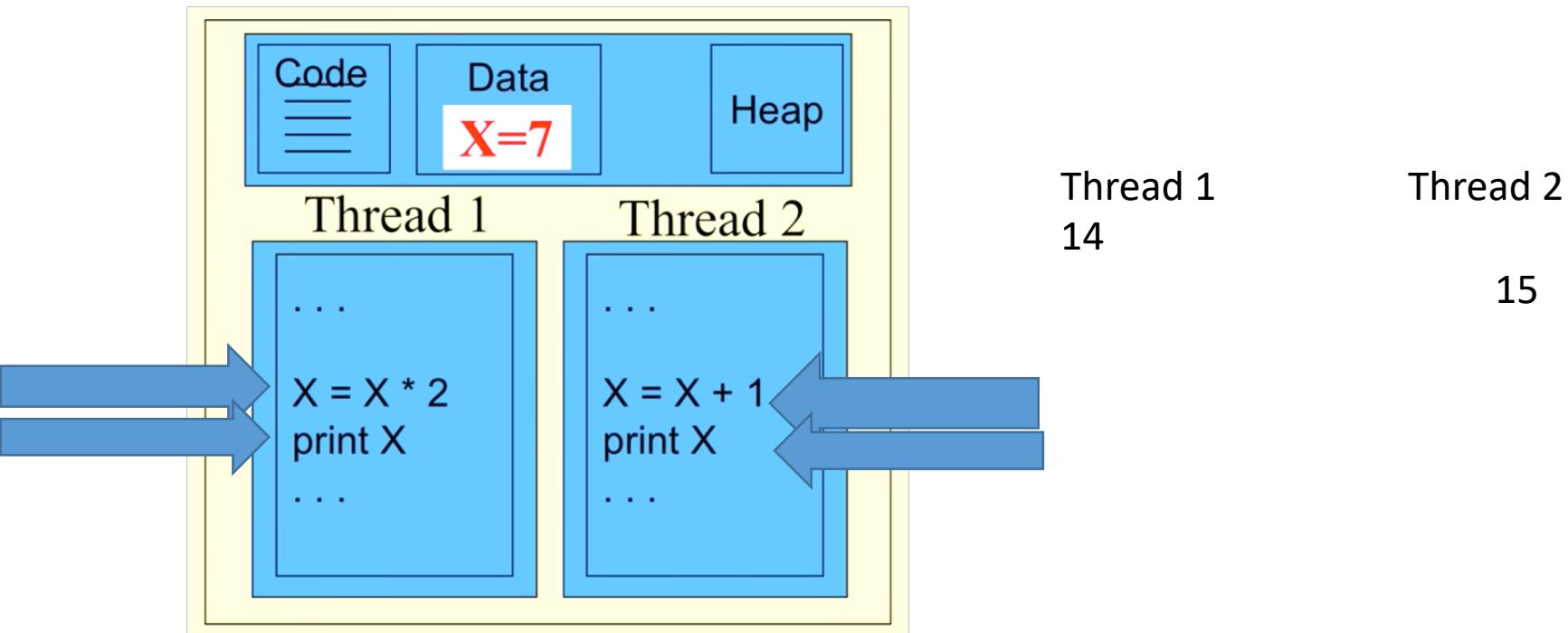
NOT Thread Safe

- Given the following code, what are the possible values that are printed



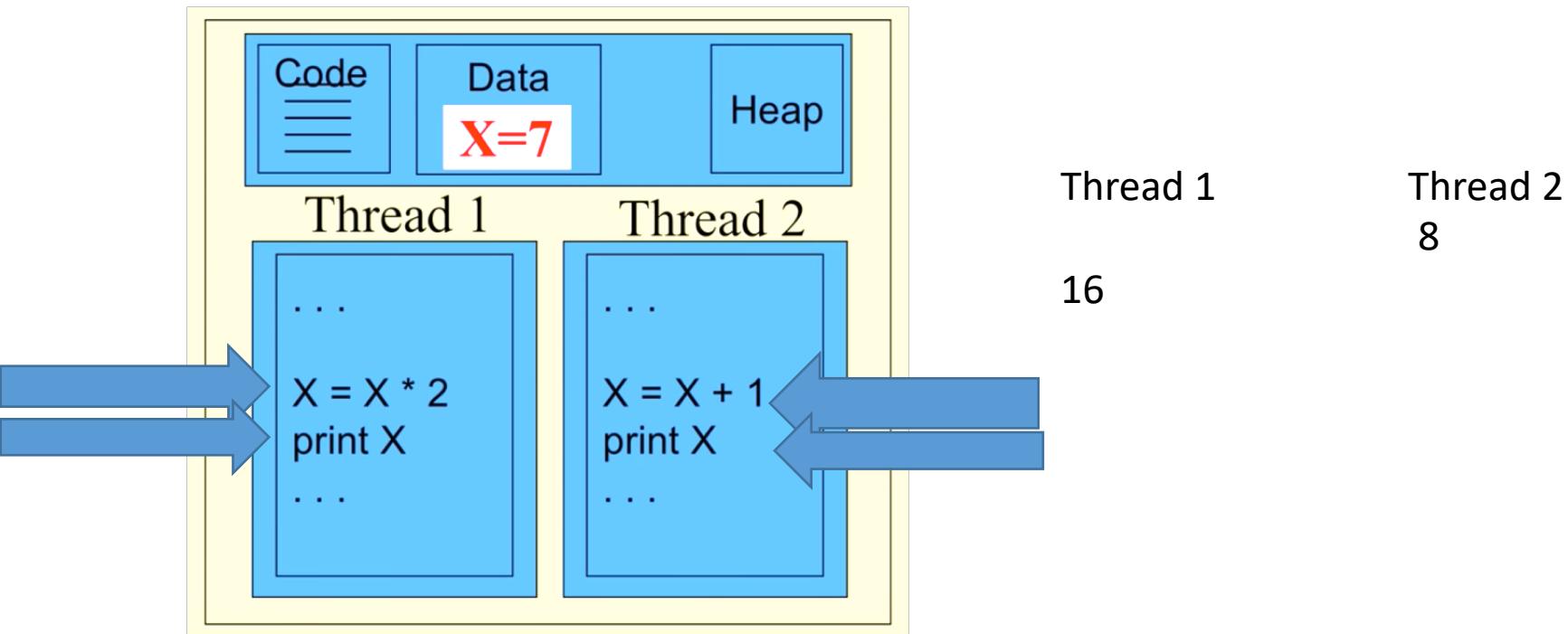
NOT Thread Safe

- Given the following code, what are the possible values that are printed



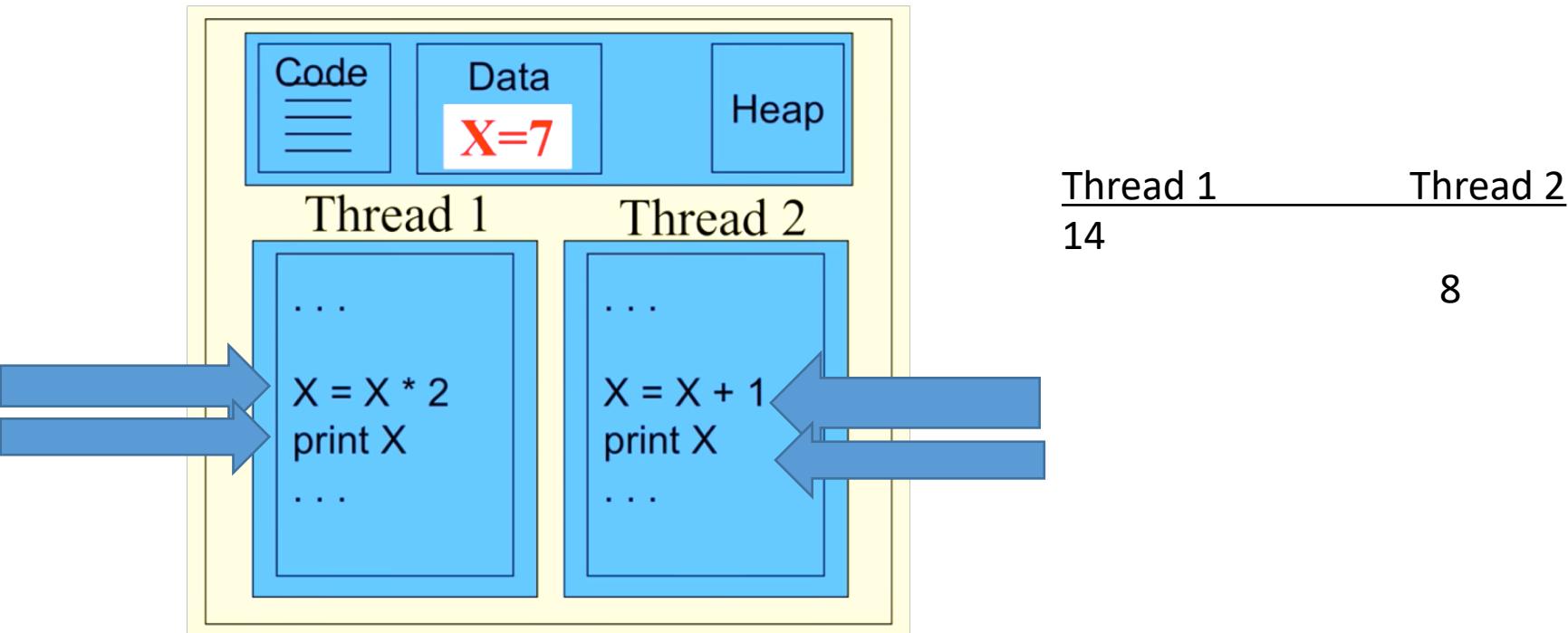
NOT Thread Safe

- Given the following code, what are the possible values that are printed



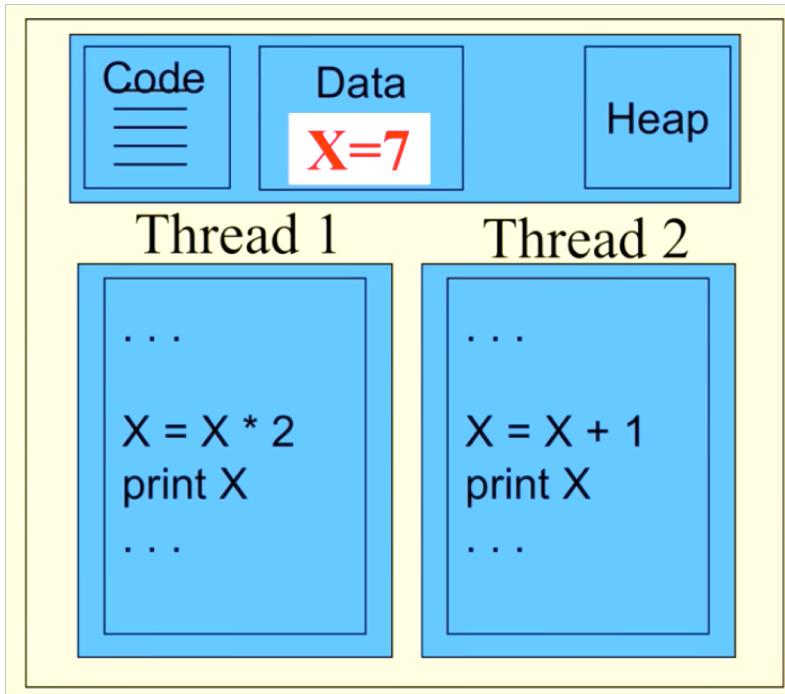
NOT Thread Safe

- Given the following code, what are the possible values that are printed



NOT Thread Safe

- Given the following code, what are the possible values that are printed

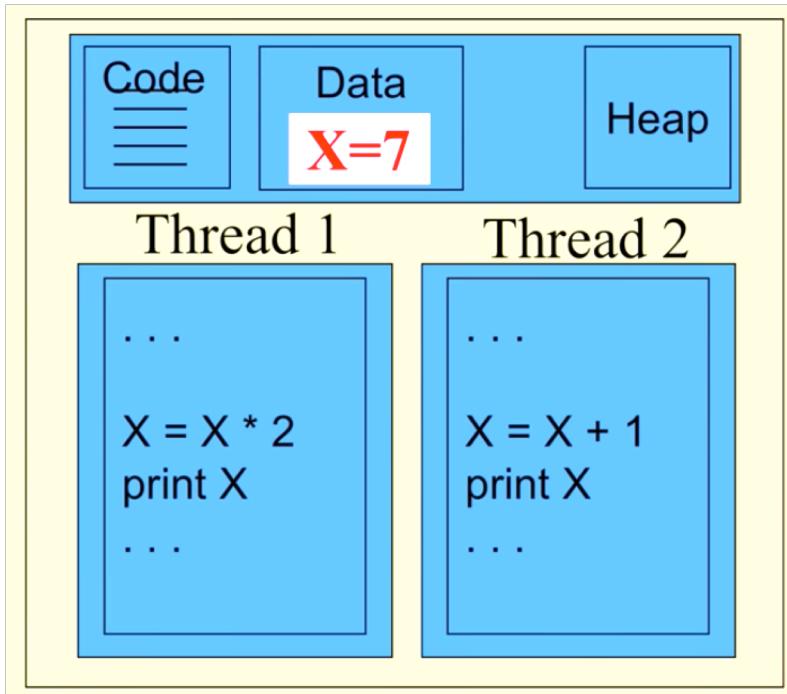


<u>Thread 1</u>	<u>Thread 2</u>
Load Multiply Save Print 14	Load Add Save Print 8



NOT Thread Safe

- Given the following code, what are the possible values that are printed



Thread 1	Thread 2
14	15
16	8
14	8
8	8
14	14
....	



Thread-Safe Code

- Code is **thread-safe**
 - it behaves correctly during simultaneous or *concurrent* execution by multiple threads.
 - multiple threads accessing the same shared data can cause different values to be used in different threads (**race condition**)
 - can be thread safe if each piece of **shared data is only be accessed by only one thread at any given time**
- If two threads share and execute the same code, then it is not safe to use the following unprotected **shared** data
 - use of global variables is not thread safe
 - use of static variables is not thread safe
 - use of heap variables is not thread safe



Reentrant Code

- Reentrancy:

- Code is reentrant if a **single thread** (really, a sequence of execution) can be interrupted in **the middle of executing the code** and then reenter the same code **later in safe manner** (before the first entry has been completed).

```
void swap(int* x, int* y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

Reentrant Code

```
void swap(int* x, int* y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

```
1  int tmp;
2
3  void swap(int* x, int* y)
4  {
5      /* Save global variable. */
6      int s;
7      s = tmp;
8
9      tmp = *x;
10     *x = *y;
11     *y = tmp;
12
13     /* Restore global variable. */
14     tmp = s;
15 }
16
17 void isr()
18 {
19     int x = 1, y = 2;
20     swap(&x, &y);
21 }
```

- Reentrancy:
 - Code is reentrant if a **single thread** (really, a sequence of execution) **can be interrupted in the middle of executing the code and then reenter the same code later in safe manner** (before the first entry has been completed).

Reentrant Code

- Reentrancy was developed for interrupt service routines (ISRs)
 - In the middle of an OS processing an interrupt, its ISR can be interrupted to process a second interrupt
 - The same OS ISR code may reentered a 2nd time before the 1st interrupt has been fully processed.
 - If the ISR code is not well-written, i.e., reentrant, then the system could hang or crash.

Reentrant Code Example

Neither Reentrant Nor Thread-safe

```
int tmp;

void swap(int* x, int* y)
{
    tmp = *x;
    ...
    *x = *y;
    ...
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

Reentrant and Thread-safe

```
void swap(int* x, int* y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```



Thread-safe but not Reentrant

- Must limit access to one thread at a time

```
function f() {  
  
    lock(); // gives this thread exclusive use  
            // keeps all other threads waiting  
    change global variable G;  
    →  
    unlock (); // Give use to another thread  
}
```

This code is NOT reentrant because if `f()` is interrupted just before the `unlock()`, and `f()` is called a 2nd time, the system will hang, because the 2nd call will try to lock, then be unable to lock, because the 1st call had not yet unlocked the system.



Thread Safe and Reentrant Code

- Code can be



How to write Safe Code

- What is safe?
 - The output would be something that you actually mean to get
- **How to write reentrance code:**
 - Do not access mutable global or function-static variables.
 - Do not self-modify code.
 - Do not invoke another function that is itself non-reentrant.
- Tips
 - Make sure all functions have NO state
 - Make sure your objects are “recursive-safe”
 - Make sure your objects are correctly encapsulated
 - Make sure your thread-safe code is recursive-safe
 - Make sure users know your object is not thread-safe.

Synchronization

Concurrency

- Multiple processes/threads executing at the same time accessing a shared resource
 - Reading the same file/resource
 - Accessing shared memory
- Benefits of Concurrency
 - Speed
 - Economics



Concurrency

- **Concurrency** is the interleaving of processes in time to give the appearance of simultaneous execution.
 - Differs from parallelism, which offers genuine simultaneous execution.
- **Parallelism** introduces the issue that different processors may run at different speeds
 - This problem is mirrored in concurrency as different processes progress at different rates

Condition for Concurrency

- Must give the same results as serial execution
- Using shared data requires synchronization

Example

Shared data/variables

data



count



avg



Serial Execution

```
// Waiting for data
while (true)
{
    v = get_value()
    add_new_value (v)
}

// Process data
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum + v) / count;
}
```



Concurrent Execution

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum + v)/ count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum + v)/ count;
}
```

Are we still going to have data consistency?

Correct values at all times in all conditions?



Concurrency

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum +v)/ count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum +v)/ count;
}
```



Concurrency

More tricky issue:

C statements can compile into several machine language instructions

e.g. `count++`

```
mov R2, count  
inc R2  
mov count, R2
```

If these low-level instructions are *interleaved*, e.g. one process is preempted, and the other process is scheduled to run, then the results of the **count** value can be unpredictable

Concurrency

- Must give the same results as serial execution
- Using shared data requires synchronization

How can various mechanisms be used to ensure the **orderly execution of cooperating processes** that share address space so that data consistency is maintained?

Synchronization techniques:

- Mutex
- Semaphore
- Condition Variable
- Monitor



Race Condition

- Occurs in situations where:
 - Two or more processes (or threads) are accessing a shared resource
 - and the final result **depends on the order of instructions** are executed
- *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*

Critical Section

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Critical Section

Where is the critical section in the *add_new_value ()* code?



University of Colorado
Boulder

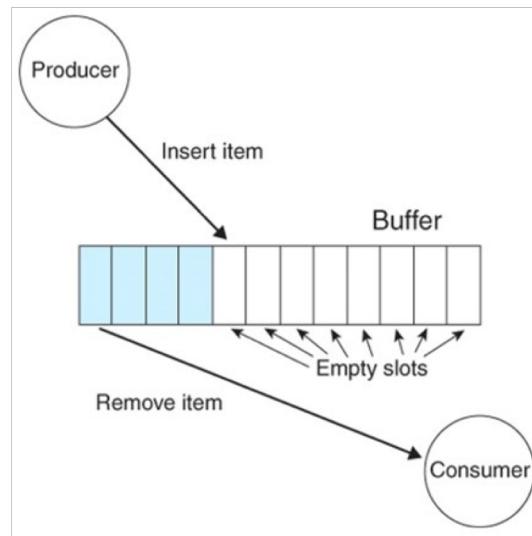
Race Condition: Solution

- Solution must satisfy the following conditions:
 - **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 wish to enter their critical sections can participate in the decision on which will enter its critical section next
 - this selection **cannot** be postponed indefinitely (OS must run a process, 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**)



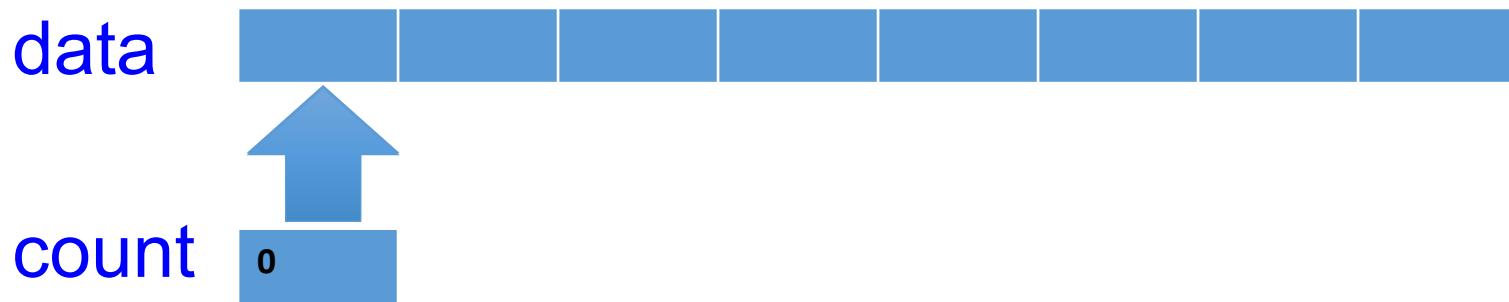
Producer-Consumer Problem (Bounded Buffer Problem)

- Two processes (producer and consumer) share a fixed size buffer
- Producer puts new information in the buffer
- Consumer takes out information from the buffer

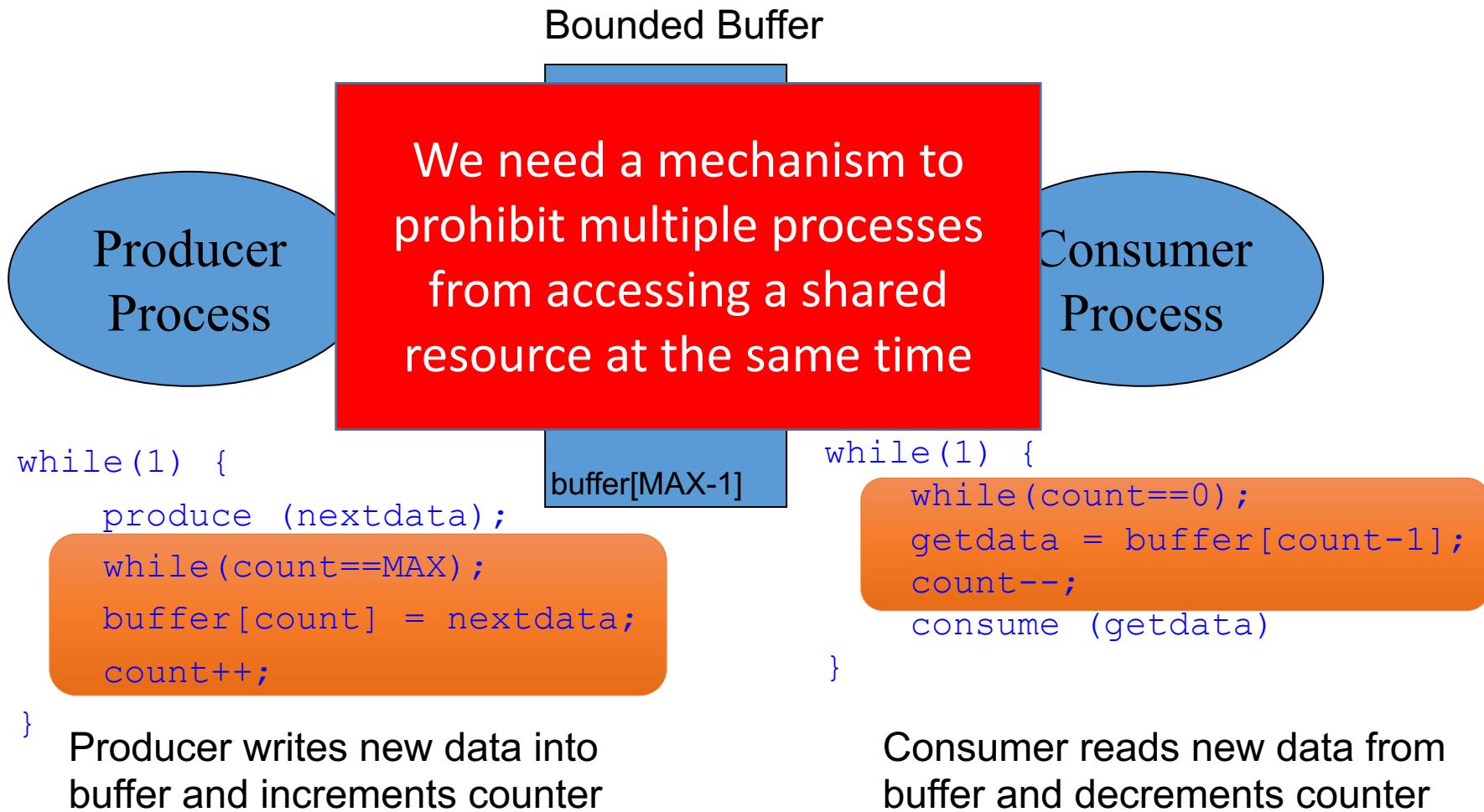


Producer-Consumer Problem

Shared data



Producer-Consumer Problem



Mutual Exclusion

- Mechanism to make sure no more than one process can execute in a critical section at any time
- How can we implement mutual exclusion?

Regular code

Entry critical section

Critical section

Access shared resource

Exit critical section

Regular code



Critical Section

```
// Producer  
while(1) {  
    produce (nextdata);  
    while (count==MAX);  
    Entry critical section  
    buffer[count] = nextdata;  
    count++;  
    Exit critical section  
}  
  
// Consumer  
while(1) {  
    while (count==0);  
    Entry critical section  
    getdata = buffer[count-1];  
    count--;  
    Exit critical section  
    consume (getdata)  
}
```



Solution 1: 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;
```

```
    producer code  
disableInterrupts();  
counter++;  
enableInterrupts();  
. . . remaining producer code
```

```
    consumer code  
disableInterrupts();  
counter--;  
enableInterrupts();  
. . . remaining consumer code
```



Solution 1: Disabling Interrupts

Problems:

- If a user forgets to enable interrupts???
- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; disabling interrupts blocks all processes
- Two or more CPUs???



Solution 2: Software Only Solution

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

Code for producer

```
/* Acquire the lock */  
while(lock) { no_op; } (1)  
lock = TRUE; (3)
```

```
/* Execute critical  
section */  
counter++;
```

```
/* Release lock */  
lock = FALSE;
```

Code for consumer

```
/* Acquire the lock */  
while(lock) { no_op; } (2)  
lock = TRUE; (4)
```

```
/* Execute critical  
section */  
counter--;
```

```
/* Release lock */  
lock = FALSE;
```

A flawed lock implementation:

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



Solution 2: Software Only Solution

- Implementing mutual exclusion in software is not hard but it is not always work
- Need help from hardware
- Modern processors provide such support
 - Atomic test and set instruction
 - Atomic compare and swap instruction

Solution 2: Software Only Solution

- Peterson's solution:
Restricted to only 2 processes.

```
int turn;
boolean flag[2];

do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = false;

    remainder section

} while (true);
```

- Turn indicates who will be run next
- Flag indicates who is ready to run next

- Need to prove:
 - (1) Mutual exclusion is preserved
 - (2) Progress is made
 - (3) Bounded-waiting is met

Atomic Test-and-Set

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

$y = \text{read}(x); x = \text{value};$

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

```
boolean TS(boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv; // returns original value of the target  
}
```

- The entire sequence is a single instruction (atomic), implemented in hardware



Solution 3: Mutual exclusion using TS

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

Code for p₁

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

```
/* Execute critical section */  
counter++;
```

```
/* Release lock */  
lock = FALSE;
```

Code for p₂

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

```
/* Execute critical section */  
counter--;
```

```
/* Release lock */  
lock = FALSE;
```



Atomic Test-and-Set

- The boolean TS () instruction is essentially a swap of values
- 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
- Don't have to disable and reenable interrupts
- Do you see any problems?
 - busy waiting.
`while(TS(&lock)) ;`



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
Acquire(lock);  
    data = buffer[count-1];  
    count--;  
Release(lock);  
    consume(data);  
}
```



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
Acquire(lock);  
    data = buffer[count-1];  
    count--;  
Release(lock);  
    consume(data);  
}
```

CPU is spinning & waiting



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```

CPU is spinning & waiting



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```

CPU is spinning & waiting



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```

How can we eliminate the busy waiting?



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```



Mutual exclusion using TS

```
shared boolean lock = FALSE;  
shared int count;  
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Code for p₂

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```

- Look first at the busy wait when there is no room for data or no data
 - Need a way to pause a process/thread until the space is available or data is available.



Mutual Exclusion

- How can we eliminate the **busy waiting**?
- Need a way to pause a process/thread until the lock is available

`sleep()` and `wakeup()` primitives

- *sleep()*: causes a running 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

```
// producer – place data into buffer
while(1) {
    if (counter==MAX) sleep();
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
    if (counter == 1) wakeup (p2);
}
```

```
// consumer – take data out of buffer
while(1) {
    if (counter==0) sleep();
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
    if (counter == MAX - 1) wakeup (p1);
}
```



sleep() and wakeup() primitives

- Problem with counter++ and counter-- still exist
 - Can be solved using TS but it has busy waiting
- Possible problem with order of execution:
 - Consumer reads counter and counter == 0
 - Scheduler schedules the producer
 - Producer puts an item in the buffer and signals the consumer to wake up
 - Since consumer has not yet invoked sleep(), the wakeup() 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 sleep() and wakeup() invocations



Semaphores

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

Semaphores

```
typedef struct {  
    int value;  
    PID *list[ ];  
} semaphore;
```

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

Both wait() and signal()
operations are atomic

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



Mutual Exclusion using Semaphores

Binary Semaphore

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

Process P1:

```
wait(S);  
// execute critical section  
counter++;  
signal(S);
```

Process P2:

```
wait(S);  
// execute critical section  
counter--;  
signal(S);
```

- Both processes atomically wait() and signal() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable **counter**
- ***This solve mutual exclusion but will also eliminate busy wait***



Problems with semaphores

shared R1, R2;

```
semaphore Q = 1; // binary semaphore as a mutex lock for R1
```

```
semaphore S = 1; // binary semaphore as a mutex lock for R2
```

Process P1:

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

modify R1 and R2;

```
signal(S);  
signal(Q);
```

Process P2:

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

modify R1 and R2;

```
signal(Q);  
signal(S);
```

Potential for deadlock

Deadlock

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

Other problematic scenarios

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

Another problem with synchronization

shared R1, R2;

semaphore S=1; // binary semaphore as a mutex lock for R1 & R2

Process P1:

wait(S)

modify R1 and R2;

Signal (S);

Process P2:

wait(S)

modify R1 and R2;

signal(S);

Process P3:

wait(S);

modify R1 and R2;

signal(S);

Potential for starvation



Starvation

- The possibility that a process would never get to run
- For example, in a multi-tasking system the resources could switch between two individual processes
- Depending on how the processes are scheduled, a third process may never get to run
- The third task is being starved of accessing the resource



Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p₂

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```



Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p₁

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p₂

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```

- Busy waiting removed from the mutual exclusion when waiting on lock
- Does it solve all busy waiting issues?



pthread Synchronization

- Mutex locks
 - Used to protect critical sections
- Some implementations provide semaphores through POSIX SEM extension
 - Not part of pthread 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);
```



pthread mutex

- pthread mutexes can have only one of two states: **lock or unlock**
- Important restriction
 - Mutex ownership:
Only the thread that locks a mutex can unlock that mutex
 - Mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads or processes

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); //same as wait( )
```

```
int sem_post(sem_t *sem); //same as signal( )
```

```
sem_getvalue( ); // check the current value of the semaphore
sem_close( ); // done with the semaphore then close
```

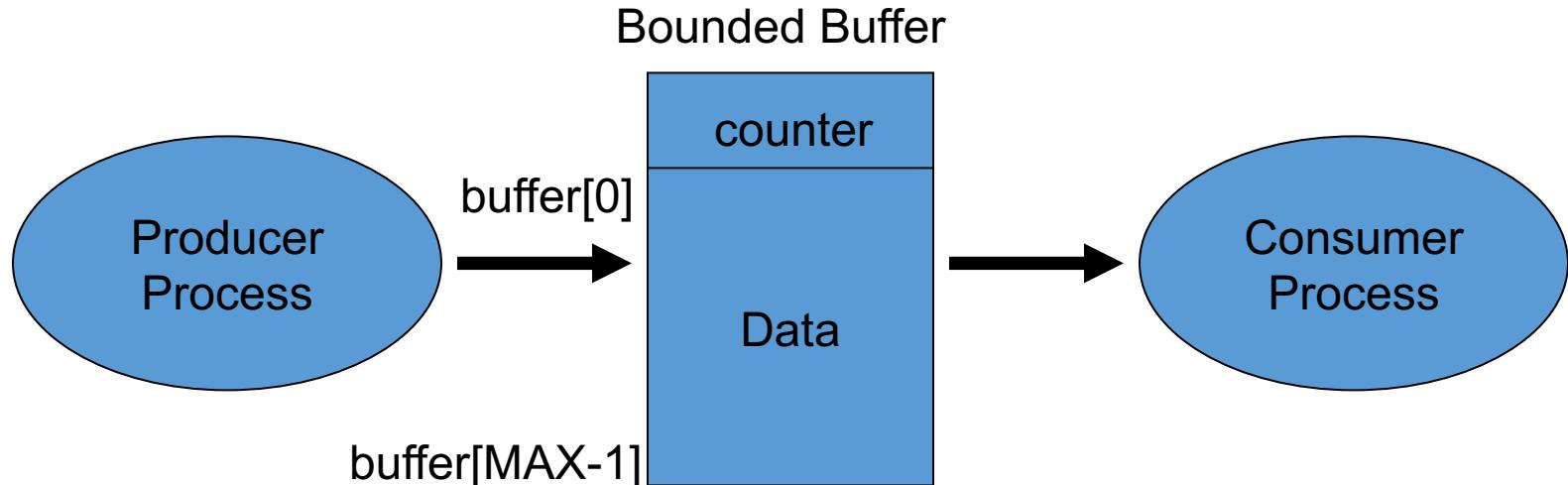
Producer-Consumer Problem

also known as

Bounded Buffer Problem

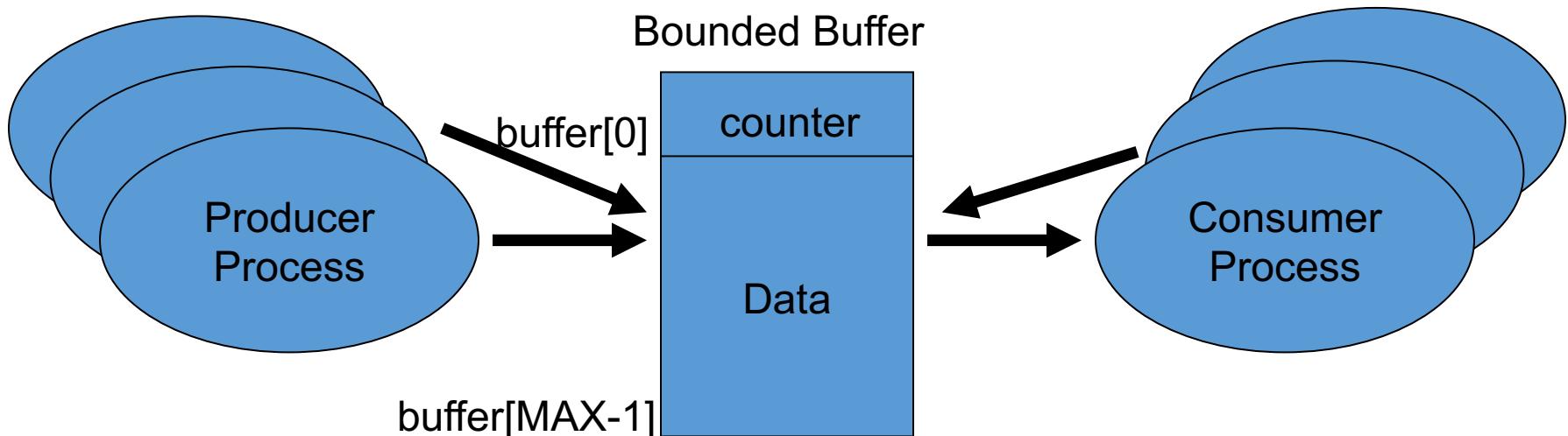
- We have already seen this problem with one producer and one consumer
- General problem: **multiple** producers and multiple consumers
- **Producers** puts new information in the buffer
- **Consumers** takes out information from the buffer

Prior Bounded-Buffer P/C Approach



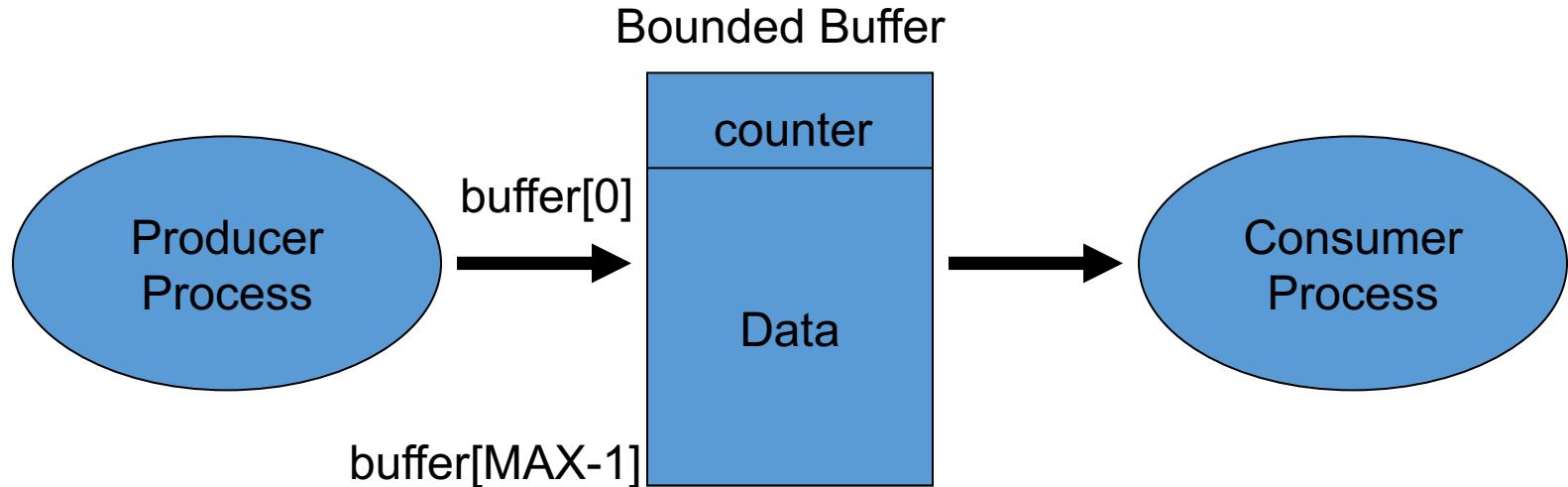
- Producer places data into a buffer at the next available position
- Consumer takes information from the earliest item

Prior Bounded-Buffer P/C Approach



- Producer places data into a buffer at the next available position
- Consumer takes information from the earliest item

Prior Bounded-Buffer P/C Approach

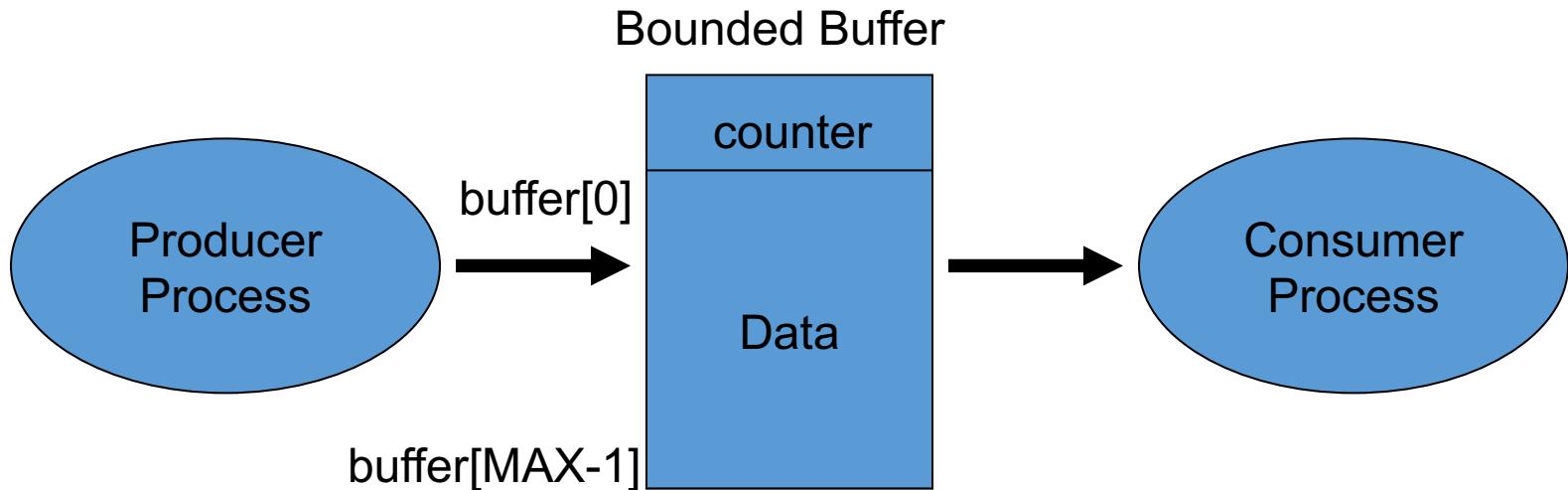


```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```



Prior Bounded-Buffer P/C Approach



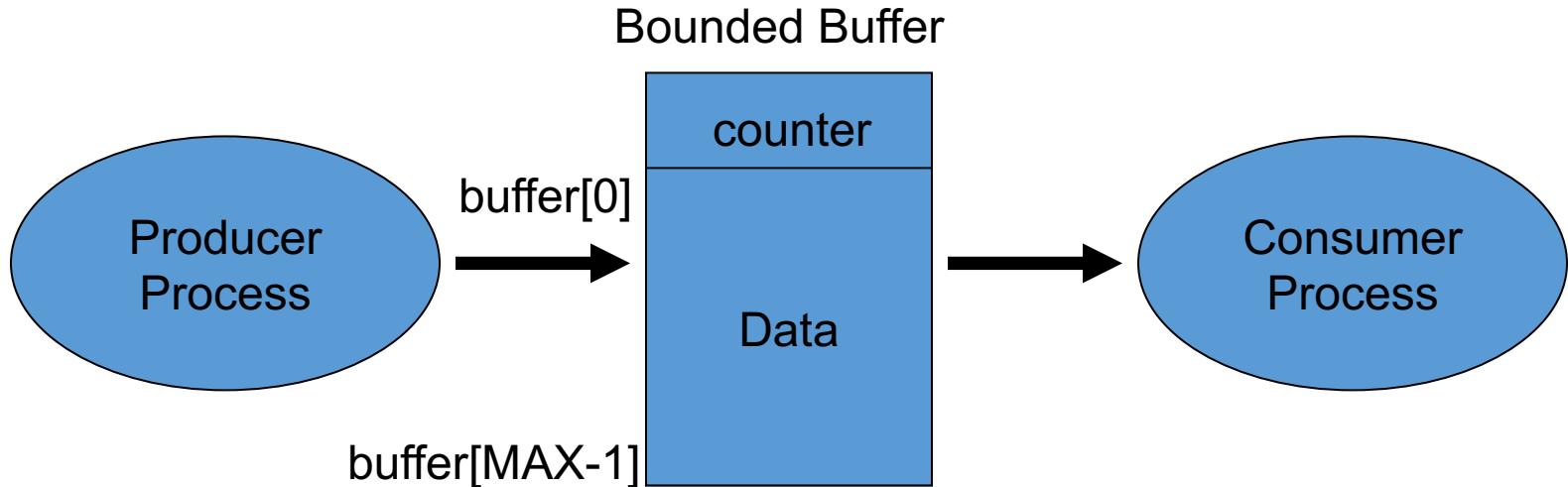
```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Busy-wait!

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```



Prior Bounded-Buffer P/C Approach



```
while(1) {  
    produce (nextdata)  
    while(count==MAX);  
    Acquire(lock);  
    buffer[count] = nextdata;  
    count++;  
    Release(lock);  
}
```

Busy-wait!

```
while (1) {  
    while(count==0);  
    Acquire(lock);  
    data = buffer[count-1];  
    count--;  
    Release(lock);  
    consume(data);  
}
```

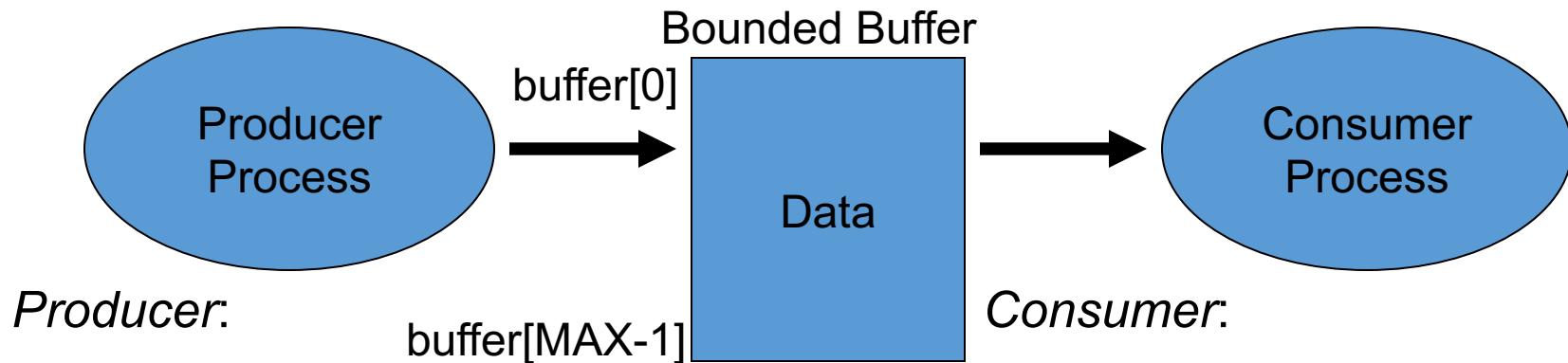


Bounded-Buffer Goals

- In the prior approach, both the producer and consumer are ***busy-waiting*** using locks
- Instead, want both to sleep as necessary
 - Goal #1: Producer should block when buffer is full
 - Goal #2: Consumer should block when the buffer is empty
 - Goal #3: mutual exclusion when buffer is partially full
 - Producer and consumer should access the buffer in a synchronized mutually exclusive way



Bounded Buffer Solution

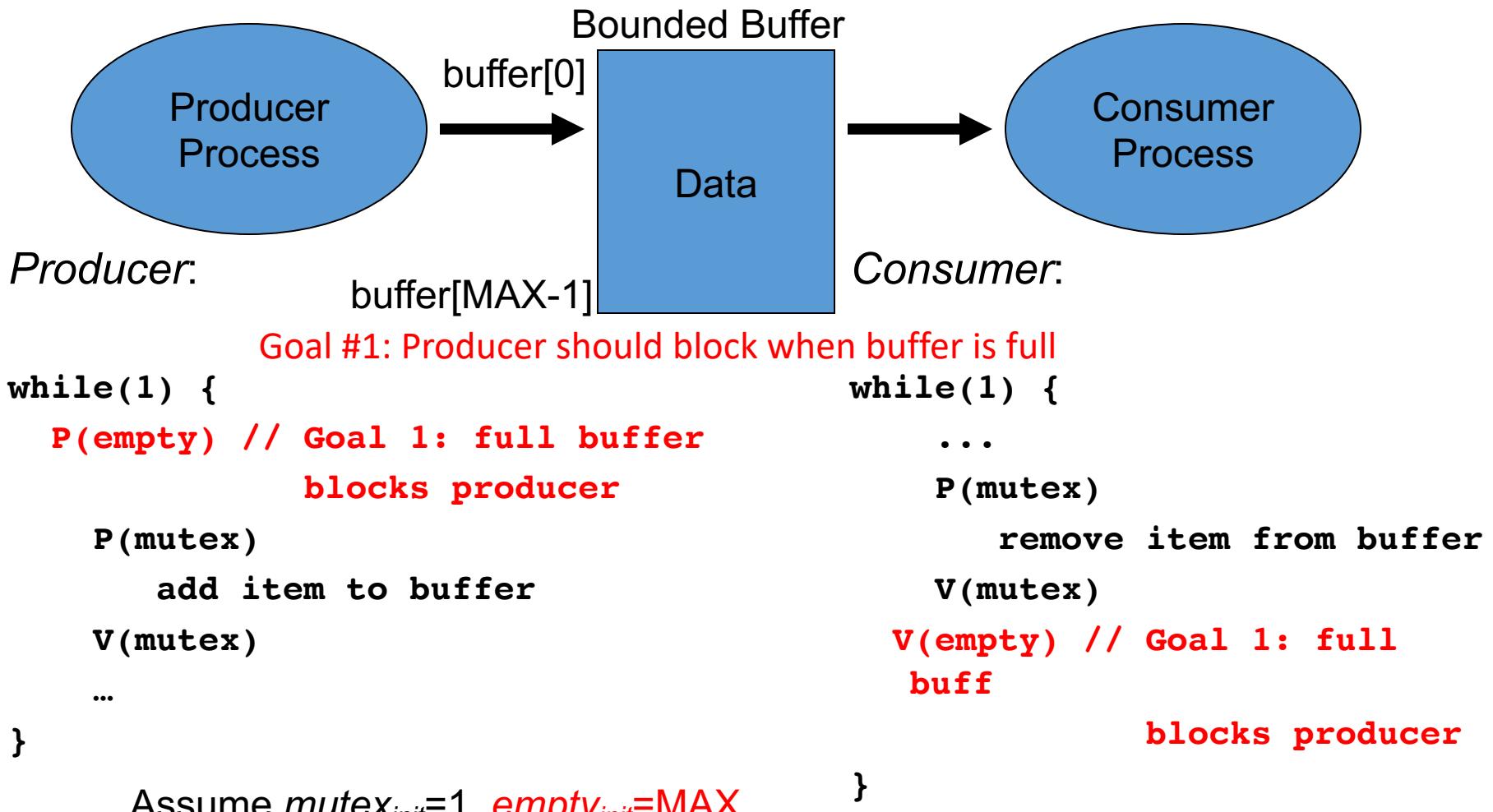


```
while(1) {  
    ...  
    P(mutex) // Goal 3: mutual  
            exclusion  
    add item to buffer  
    V(mutex)  
    ...  
}
```

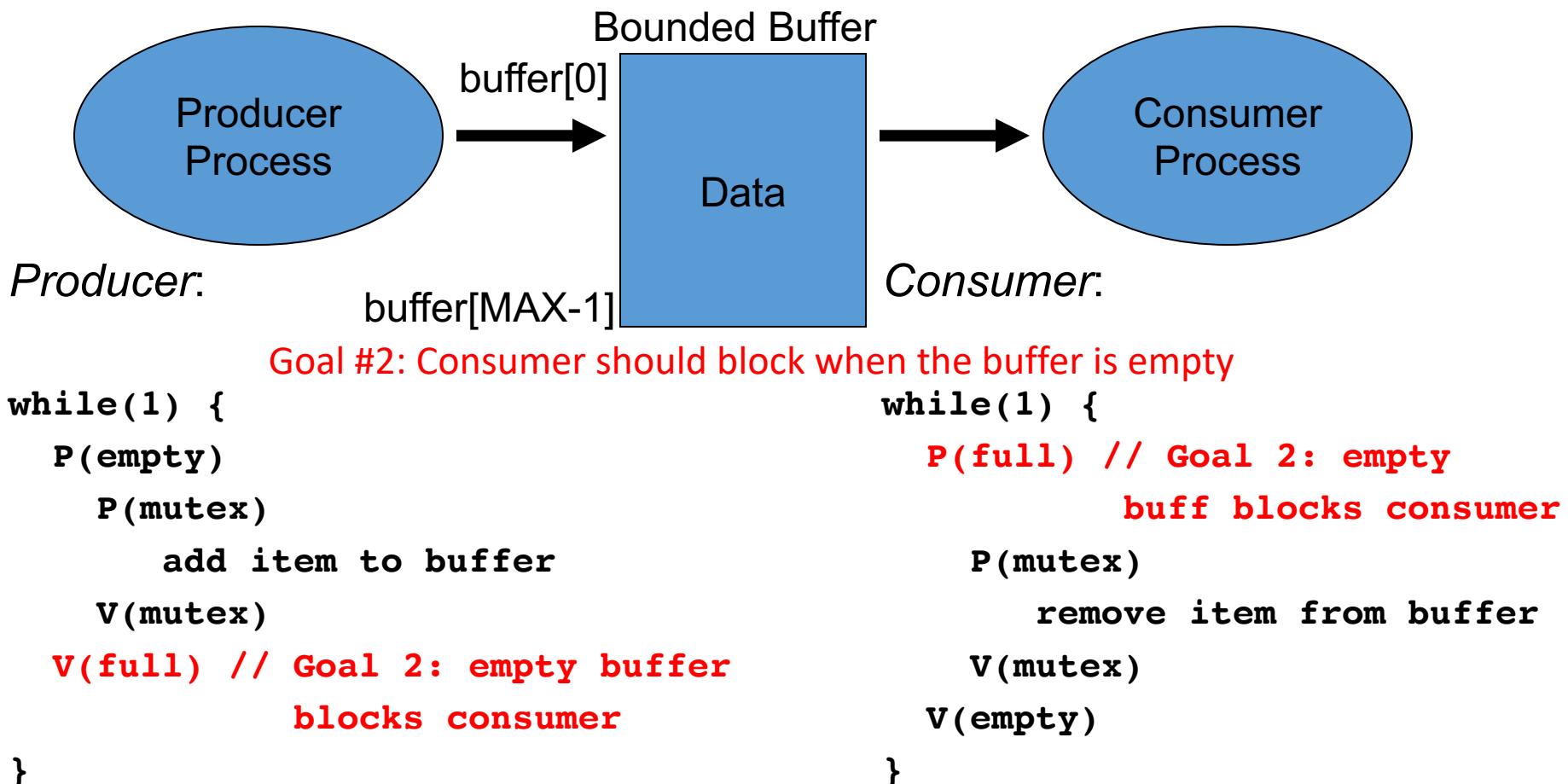
```
while(1) {  
    ...  
    P(mutex) // Goal 3: mutual  
            exclusion  
    remove item from buffer  
    V(mutex)  
    ...  
}
```

Assume $mutex_{init} = 1$

Bounded Buffer Solution (2)

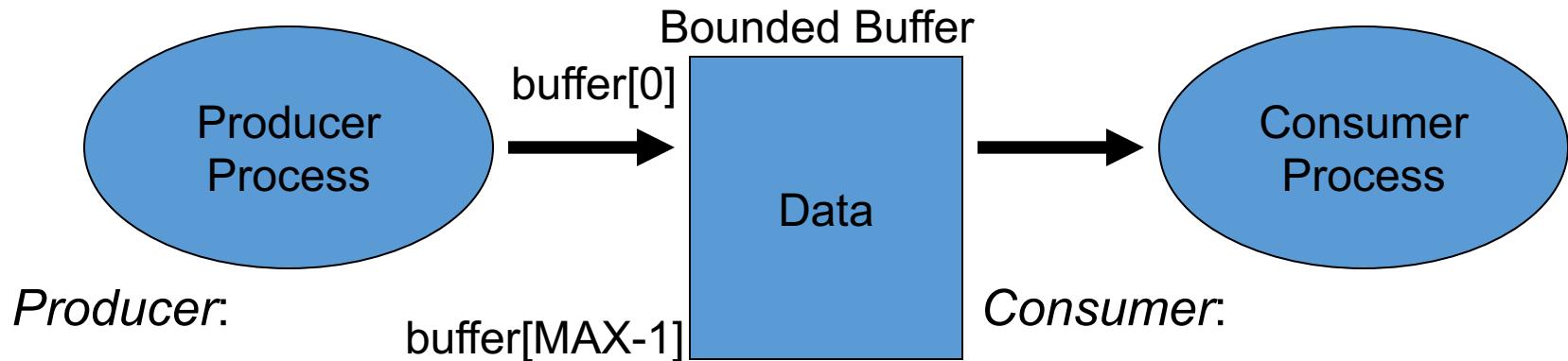


Bounded Buffer Solution (3)



Assume $mutex_{init}=1$, $empty_{init}=MAX$, $full_{init}=0$

Bounded Buffer Solution (4)



```
while(1) {  
    P(empty)  
    P(mutex)  
    add item  
    V(mutex)  
    V(full)  
}
```

Achieves:

- 1) mutual exclusion
- 2) blocked producer if buffer full
- 3) blocked consumer if buffer empty
- 4) no deadlock
- 5) is efficient (no busy wait)

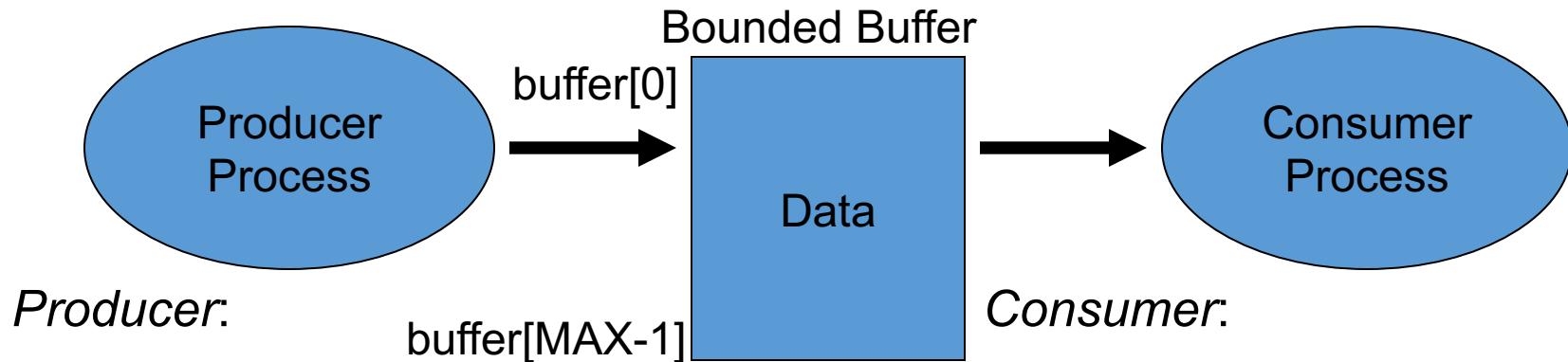
```
while(1) {  
    P(mutex)  
    remove item from buffer  
    V(mutex)  
}
```

Assume $mutex_{init}=1$, $empty_{init}=MAX$, $full_{init}=0$



University of Colorado
Boulder

Bounded Buffer Solution (5)



```
while(1) {  
    P(space_avail)  
    P(mutex)  
    add item to buffer  
    V(mutex)  
    V(items_avail)  
}
```

Assume $mutex_{init}=1$,
 $space_avail_{init}=MAX$,
 $items_avail_{init}=0$

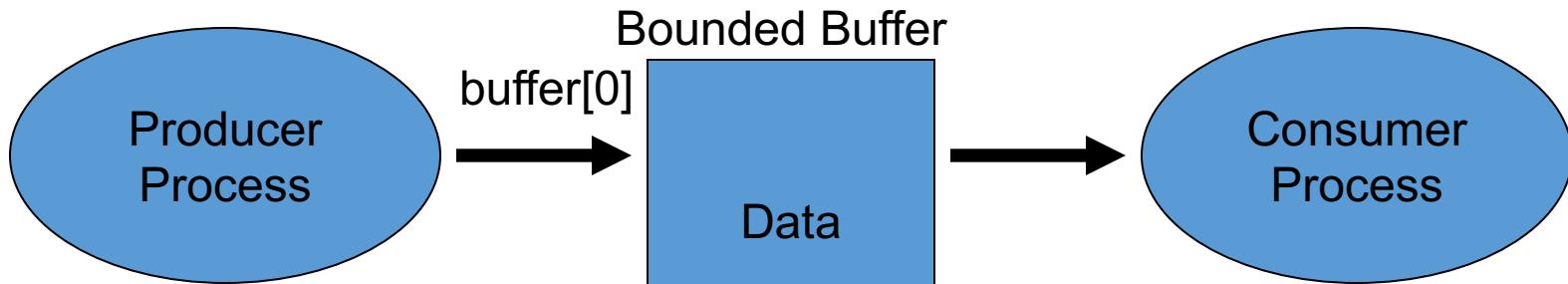
```
while(1) {  
    P(items_avail)  
    P(mutex)  
    remove item from buffer  
    V(mutex)  
    V(space_avail)  
}
```



Bounded-Buffer Design

- **Goal #1: Producer should block when buffer is full**
 - Use a counting semaphore called *empty* that is initialized to $empty_{init} = MAX$
 - Each time the producer adds an object to the buffer, this decrements the # of empty slots, until it hits 0 and the producer blocks
- **Goal #2: Consumer should block when the buffer is empty**
 - Define a counting semaphore *items_avail* that is initialized to $items_avail_{init} = 0$
 - *items_avail* tracks the # of full slots and is incremented by the producer
 - Each time the consumer removes a full slot, this decrements *items_avail*, until it hits 0, then the consumer blocks
- **Goal #3: Mutual exclusion when buffer is partially full**
 - Use a mutex semaphore to protect access to buffer manipulation, $mutex_{int} = 1$

Bounded Buffer Solution (6)



Producer:

```
while(1) {  
    P(space_avail)  
    P(mutex)  
    add item to buffer  
    V(mutex)  
    V(items_avail)  
}
```

Assume $mutex_{init}=1$,
 $space_avail_{init}=MAX$,
 $items_avail_{init}=0$

Will this solution work for
multiple Producers and
multiple customers?

```
    . . .  
    . . .  
    . . .  
    remove item from buffer  
    V(mutex)  
    V(space_avail)  
}
```





Monitors and Condition Variables

Deadlock when using Semaphores

- It can easily occur
 - Two tasks, each desires a resource locked by the other process
 - Circular dependency
- Programming errors
 - Switching order of P() and V()
 - Calling wait multiple times
 - Forgetting a signal
- Semaphores provide mutual exclusion, but can introduce deadlock



Monitors

- Semaphores can result in deadlock due to programming errors
 - forgot to add a P() or V(), or misordered them, or duplicated them
- To reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*
 - essentially automates insertion of P() and V() for you
 - As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#
 - underneath, the OS may implement monitors using semaphores and mutex locks



Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {  
    // shared local variables  
  
    function f1(...) {  
        ...  
    }  
    ...  
    function fN(...) {  
        ...  
    }  
    init_code(...)  
    ...  
}
```

- A monitor ensures that only 1 process/thread at a time can be active within a monitor
 - simplifies programming, no need to explicitly synchronize
- Implicitly, the monitor defines a mutex lock

```
semaphore mutex = 1;
```
- Implicitly, the monitor also defines mutual exclusion around each function
 - Each function's critical code is effectively:

```
function fj(...) {  
    P(mutex)  
    // critical code  
    V(mutex)  
}
```



Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
    // shared local variables
    int count;
    data_type data[MAX_COUNT];
    function f1(...) {
        ...
    }
    ...
    function fN(...) {
        ...
    }
    init_code(...) {
        ...
    }
}
```

- The monitor's local variables can only be accessed by local monitor functions
- Each function in the monitor can only access variables declared locally within the monitor and its parameters



Monitors and Condition Variables

- Previous definition of a monitor achieves
 - mutual exclusion
 - hiding of `wait()` and `signal()` from user
 - **loses the ability that semaphores had to enforce order**
 - `wait()` and `signal()` are used to provide mutual exclusion
 - but have lost the unique ability for one process to signal another blocked process using `signal()`
 - there is no way to have a process sleep waiting on the signal
- 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*.

Condition Variables

- Augment the mutual exclusion of a monitor with an ordering mechanism
 - Recall: Semaphore P() and V() provide both mutual exclusion and ordering
 - Monitors alone only provide mutual exclusion
- A condition variable provides ordering through Signaling
 - Used when one task wishes to wait until a condition is true before proceeding
 - Such as a queue being full enough or data being ready
 - A 2nd task will signal the waiting task, thereby waking up the waiting task to proceed



Monitors and Condition Variables

condition y;

A condition variable y in a monitor allows three operations

- y.wait()
 - blocks the calling process
 - can have multiple processes suspended on a condition variable, typically released in FIFO order
 - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
- y.signal()
 - resumes exactly 1 suspended process.
 - If no process is waiting, then function has *no effect*.
- y.queue()
 - Returns true if there is at least one process blocked on y

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```



Condition Variables Example

Block Task 1 until a condition holds true, e.g. queue is empty

```
condition wait_until_empty;
```

Task 1:

```
wait_until_empty.wait();
...// proceed after
queue empty
```

Task 2:

```
...
// queue is empty so signal
wait_until_empty.signal();
```

Problem: If Task 2 signals before Task 1 waits, then Task 1 waits forever because CV has no state!

Condition Variables vs Semaphores

- Both have wait() and signal(), but semaphore's signal()/V() preserves states in its integer value

Semaphore wait_until_empty=0;

Task 1:

```
wait(wait_until_empty);  
...// proceed after  
queue empty
```

Task 2:

```
...  
// queue is empty so signal  
signal(wait_until_empty);
```

If Task 2 signals before Task 1 waits, then Task 1 does not wait forever because semaphore has state and remembers earlier signal()



University of Colorado
Boulder

Complex Conditions

- Suppose you want task T1 to wait until a complex set of conditions set by T2 becomes TRUE
 - Use a condition variable

```
condition = x;  
int count = 0;  
Float f=0.0;
```

Task 1

```
----  
...  
while (f!=7.0 && count<=0){  
    x.wait();  
}  
... // proceed
```

Task 2

```
----  
...  
f=7.0;  
count++;  
x.signal();
```

Problem: could be a race condition in testing and setting shared variables



University of Colorado
Boulder

Complex Conditions (2)

```
lock mutex;  
Condition x;  
int count=0;  
Float f=0.0;
```

T1

...

```
Acquire(mutex);  
while(f!=7.0 && count<=0) {  
    Release(mutex);  
    x.wait();  
    Acquire(mutex);  
}  
Release(mutex);  
...// proceed
```

- Surround the test of conditions with a mutex to atomically test the set of conditions

T2

...

```
Acquire(mutex);  
f=7.0;  
count++;  
Release(mutex);  
x.signal();
```



Complex Conditions (3)

- Surround the test of conditions with a mutex to atomically test the set of conditions

```
lock mutex;
Condition x;
int count=0;
Float f=0.0;

T1
-----
...
Acquire(mutex);
While(f!=7.0 && count<=0) {
    pthread_cond_wait(&x,&mutex);
}
Release(mutex);
...// proceed

T2
-----
...
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```



Semaphores vs. Condition Variables

Semaphores

```
typedef struct {  
    int value;  
    PID *list[ ];  
} semaphore;
```

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

Both wait() and signal()
operations are atomic

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



Monitors and Condition Variables

condition y;

A condition variable y in a monitor allows three operations

- y.wait()
 - blocks the calling process
 - can have multiple processes suspended on a condition variable, typically released in FIFO order
 - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
- y.signal()
 - resumes exactly 1 suspended process.
 - If no process is waiting, then function has *no effect*.
- y.queue()
 - Returns true if there is at least one process blocked on y

Semaphores vs. Condition Variables

- **S.signal vs. C.signal**

- C.signal has no effect if no thread is waiting on the condition
 - Condition Variables are not variables (ha!) They have no value
 - S.signal has the same effect whether or not a thread is waiting
 - Semaphore retains a “memory”

- **S.wait vs. C.wait**

- S.signal check the condition and block only if necessary
 - Programmer does not need to check the condition after returning from S.wait
 - Wait condition is defined internally but limited to a counter.
 - C.wait is explicit: It does not check the condition, ever
 - Condition is defined externally and protected by integrated mutex



Other Mechanisms

- EventBarrier
 - Combine semaphores and condition variables
 - Has binary memory and no associated mutex
 - Broadcast to notify all waiting threads
 - Wait until an event is handled

EventBarrier::Wait()

If the EventBarrier is not in the signaled state, wait for it.

EventBarrier:: Signal()

Signal the event, and wait for all waiters/arrivals to respond.

EventBarrier::Complete()

Notify EventBarrier that caller's response to the event is complete.

Block until all threads have responded to the event.



Other Mechanisms

- SharedLock: Reader/Writer Lock
 - Support Acquire and release primitives
 - Guarantee mutual exclusion when writer is present
 - Provides better concurrency for readers when no writer is present

often used in database systems

easy to implement using mutexes
and condition variables

a classic synchronization problem

```
class SharedLock {  
    AcquireRead(); /* shared mode */  
    AcquireWrite(); /* exclusive mode */  
    ReleaseRead();  
    ReleaseWrite();  
}
```



Guidelines for Condition Variables

1. Understand/document the condition(s) associated with each CV.

What are the waiters waiting for?

When can a waiter expect a *signal*?

2. Always check the condition to detect spurious wakeups after returning from a *wait*: “loop before you leap”!

Another thread may beat you to the mutex.

The signaler may be careless.

A single condition variable may have multiple conditions.

3. Don’t forget: *signals on condition variables do not stack!*

A signal will be lost if nobody is waiting: always check the wait condition before calling *wait*.



Guidelines for Choosing Lock Granularity

1. *Keep critical sections short.* Push “noncritical” statements outside of critical sections to reduce contention.
2. *Limit lock overhead.* Keep to a minimum the number of times mutexes are acquired and released.

Note tradeoff between contention and lock overhead.

3. *Use as few mutexes as possible, but no fewer.*

Choose lock scope carefully: if the operations on two different data structures can be separated, it **may** be more efficient to synchronize those structures with separate locks.

Add new locks only as needed to reduce contention. “Correctness first, performance second!”



More Guidelines for Locking

1. Write code whose correctness is obvious.
2. Strive for symmetry.

Show the Acquire/Release pairs.

Factor locking out of interfaces.

Acquire and Release at the same layer in your “layer cake” of abstractions and functions.

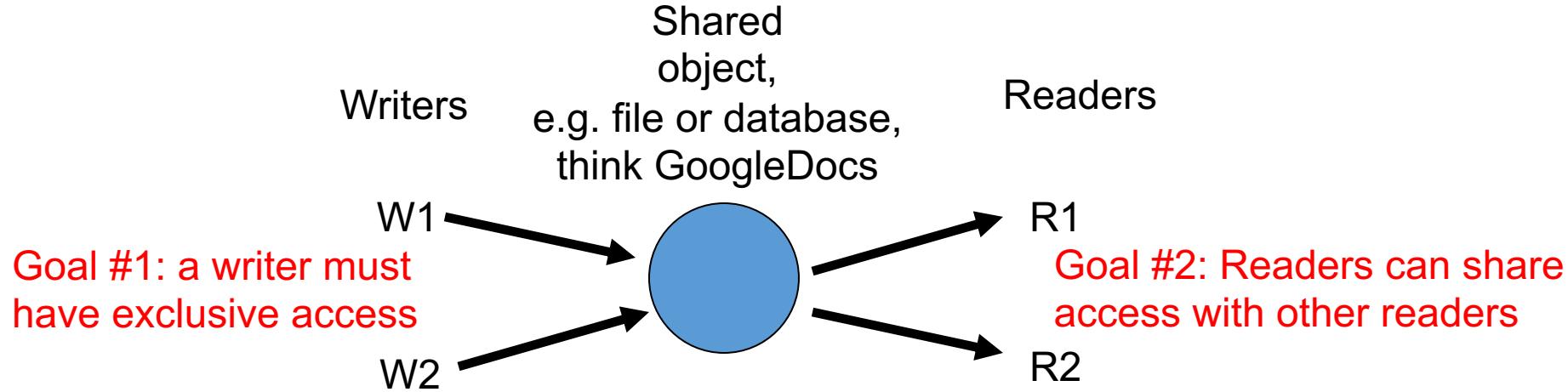
3. Hide locks behind interfaces.
4. Avoid nested locks.
If you must have them, try to impose a strict order.
5. Sleep high; lock low.

Design choice: where in the layer cake should you put your locks?



The Readers/Writers Problem

- N tasks want to write to a shared file
- M other tasks want to read from same shared file
- Must synchronize access



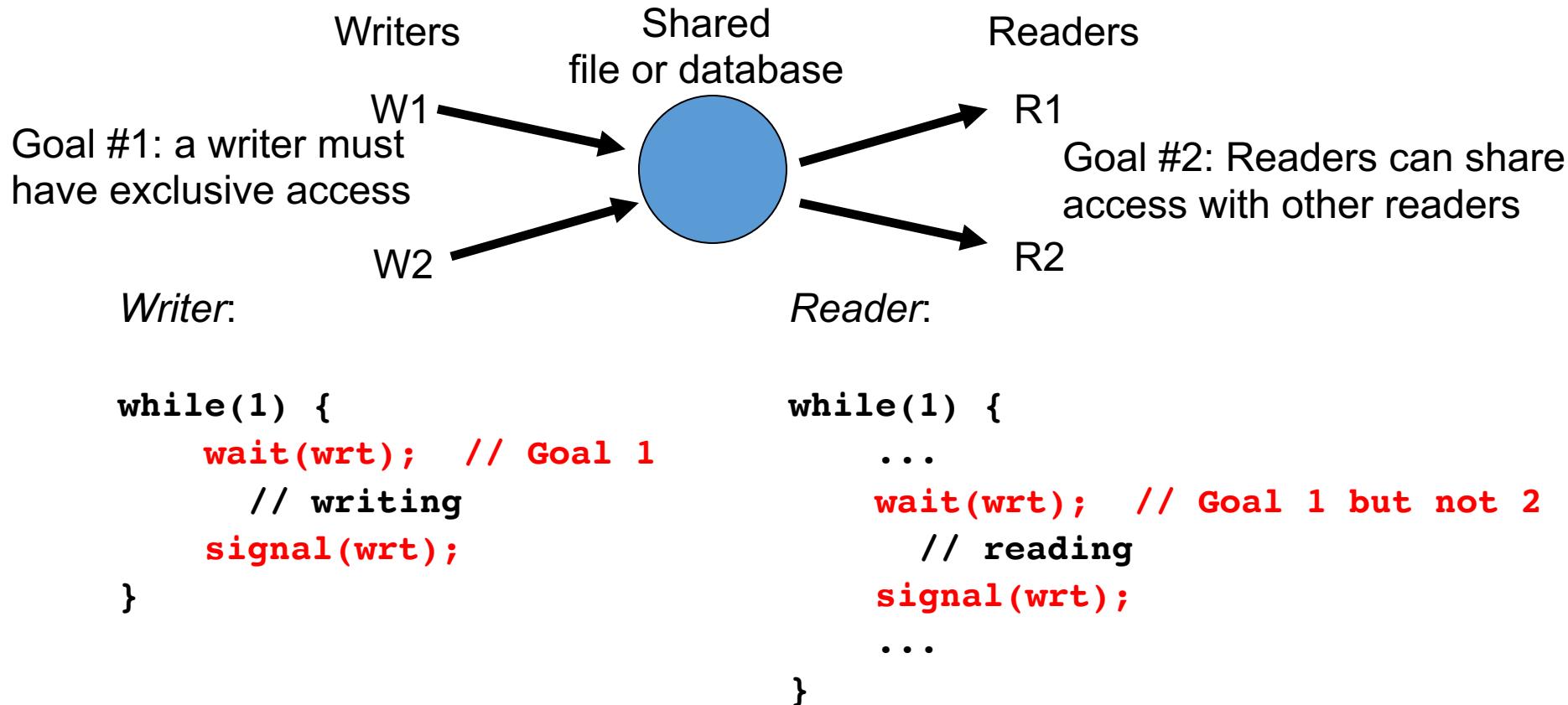
- Goal #2: should support multiple concurrent readers
 - Hence, a single mutex lock won't support that

Readers/Writers Problems

- Additional requirement #1:
 - no reader is kept waiting unless a writer already has seized the shared object
- Additional requirement #2:
 - a pending writer should not be kept waiting indefinitely by readers that arrived after the writer
 - i.e. a pending writer cannot starve

1st Readers/Writers Solution

Assume $wrt_{init}=1$ is a mutex lock/binary semaphore

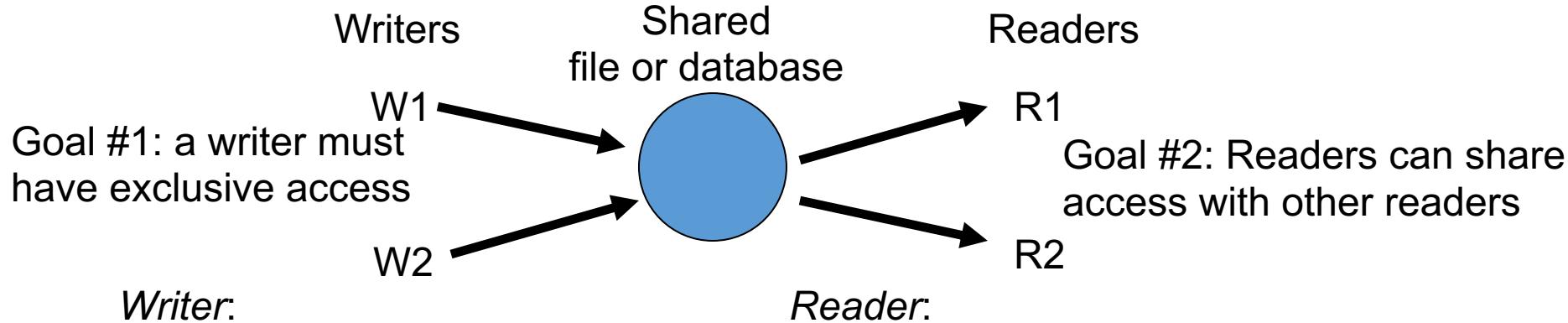


Problem: first reader grabs lock, preventing other readers (& writers)
Solution: only the first reader needs to grab the lock,
and last reader release the lock.



1st Readers/Writers Solution (2)

Assume $wrt_{init}=1$, **readcount initialized = 0**



```
while(1) {  
    wait(wrt);  
    // writing  
    signal(wrt);  
}
```

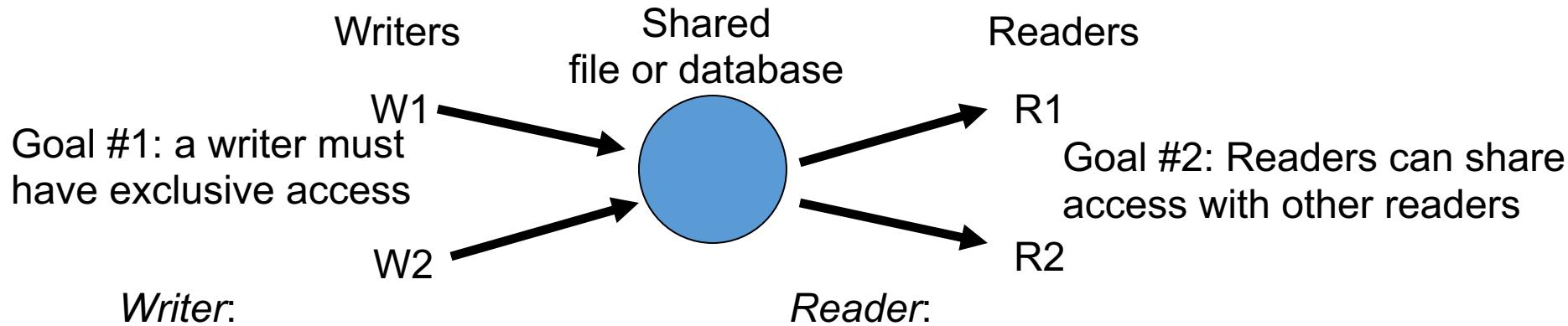
```
while(1) {  
    readcount++;  
    if (readcount==1) wait(wrt);  
    // reading  
    readcount--;  
    if (readcount==0) signal(wrt);  
    ...  
}
```

Problem: both `readcount++` and `readcount--` lead to race conditions
Solution: surround access to `readcount` with a 2nd mutex



1st Readers/Writers Solution (3)

Assume $wrt_{init}=1$, $readcount = 0$, $mutex_{init}=1$



```
while(1) {  
    wait(wrt);  
    // writing  
    signal(wrt);  
}
```

So a writer excludes other writers and readers.

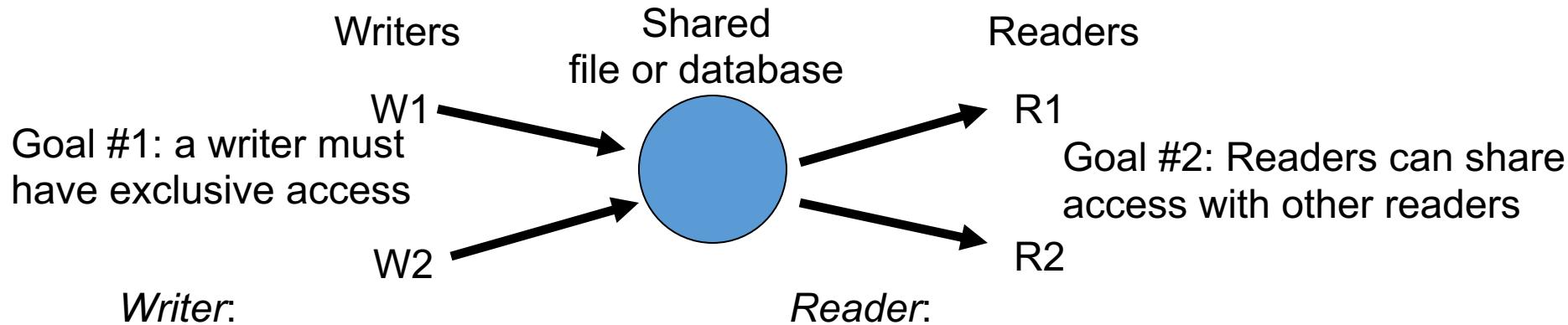
Multiple readers are allowed and exclude writers while at least 1 reader

```
while(1) {  
    wait(mutex)  
    readcount++;  
    if (readcount==1) wait(wrt);  
    signal(mutex)  
    // reading  
    wait(mutex)  
    readcount--;  
    if (readcount==0) signal(wrt);  
    signal(mutex)
```



1st Readers/Writers Solution (4)

Assume $wrt_{init}=1$, $readcount = 0$, $mutex_{init}=1$



```
while(1) {  
    wait(wrt);  
    // writing  
    signal(wrt);  
}
```

Problem: this solution could starve pending writers!

```
while(1) {  
    wait(mutex)  
    readcount++;  
    if (readcount==1) wait(wrt);  
    signal(mutex)  
    // reading  
    wait(mutex)  
    readcount--;  
    if (readcount==0) signal(wrt);  
    signal(mutex)
```



2nd Readers/Writers Solution

A pending writer should not be kept waiting indefinitely by readers that arrived after the writer

- 1st R/W solution gave precedence to readers
 - new readers can keep arriving while any one reader holds the write lock, which can starve writers until the last reader is finished
- Instead, allow a pending writer to block future reads
 - This way, writers don't starve.
 - If there is a writer,
 - New readers should block
 - Existing readers should finish then signal the waiting writer



Original Solution to the 2nd Readers/Writers Problem

```
int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1,
        writePending = 1;

writer() {
    while(TRUE) {

        P(mutex2);
        writeCount++;
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);

        P(writeBlock);
        write(resource);
        V(writeBlock);

        P(mutex2)
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}

reader() {
    while(TRUE) {

        P(writePending);
        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);
        V(writePending);

        read(resource);

        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
    }
}
```

Red = changed from 1st R/W problem solution



University of Colorado
Boulder

2nd Readers/Writers Starvation

- Once 1st writer grabs readBlock,
 - any number of writers can come through while the 1st reader is blocked on readBlock
 - and subsequent readers are blocked on writePending
 - So, behavior is that a writer can block not just new readers, but also some earlier readers
 - Note now that readers can be starved!
- Instead, want a solution that is starvation-free for both readers and writers



Starvation-free Solution to 2nd R/W

Semaphore $wrt_{init}=1$, $mutex_{init}=1$, $readBlock=1$

int readcount = 0

Reader:

Writer:

```
while(1) {  
    wait(readBlock)  
    wait(wrt); // Goal 1  
    // writing  
    signal(wrt);  
    signal(readBlock)  
}
```

```
while(1) {  
    wait(readBlock)  
    wait(mutex)  
    readcount++;  
    if (readcount==1) wait(wrt);  
    signal(mutex)  
    signal(readBlock)  
    // reading
```

This is starvation-free solution
Note how it is a minor variant of the 1st
R/W solution.

```
    wait(mutex)  
    readcount--;  
    if (readcount==0) signal(wrt);  
    signal(mutex)
```

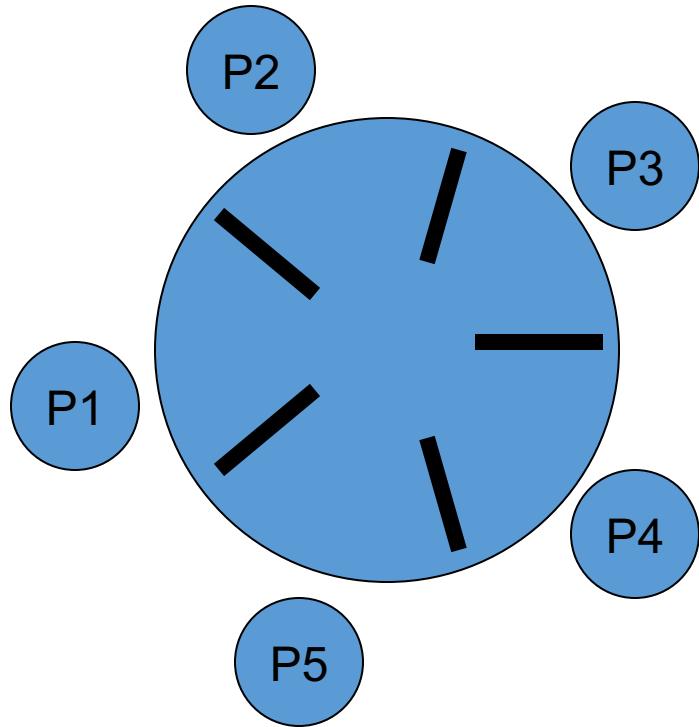
}



University of Colorado
Boulder

Dining Philosophers Problem

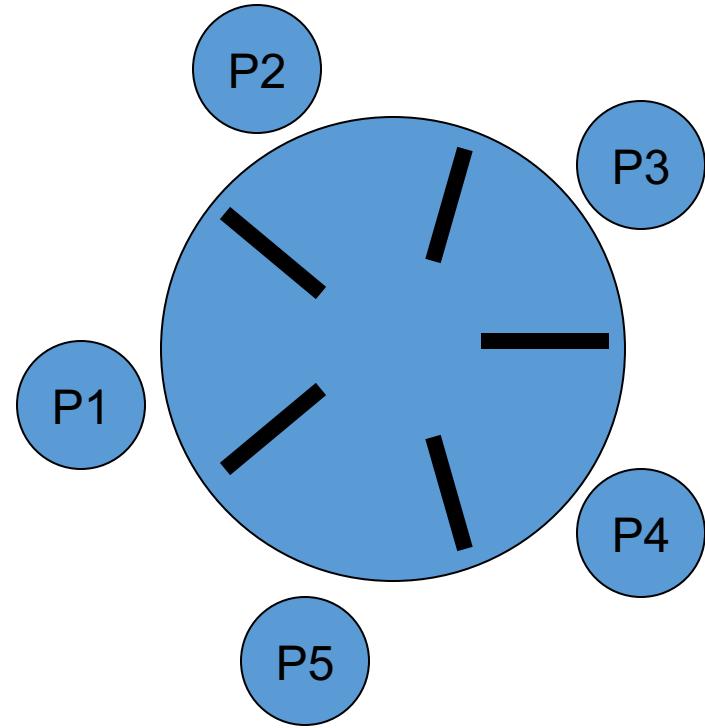
- Simple algorithm for protecting access to chopsticks:
 - Access to each chopstick is protected by a mutual exclusion semaphore
 - Prevent any other phylosophers from pickup the choptick when it is already in use by a philosopher
- **Semaphore chopstick[5];**
 - Each philosohper grabs a chipstick I by `P(chopstick[i])`
 - Each philosopher release a chopstick I by `V(chopstick[i])`



Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {  
    // obtain 2 chopsticks to my  
    // immediate right and left  
    P(chopstick[i]);  
    P(chopstick[(i+1)%N]);  
  
    // eat  
  
    // release both chopsticks  
    V(chopstick[(i+1)%N]);  
    V(chopstick[i]);  
}
```



Problem?

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .



Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX



Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
 - Low resource utilization; starvation possible



Deadlock Prevention (Cont.)

- **No Preemption –**

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released - **All or nothing**
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



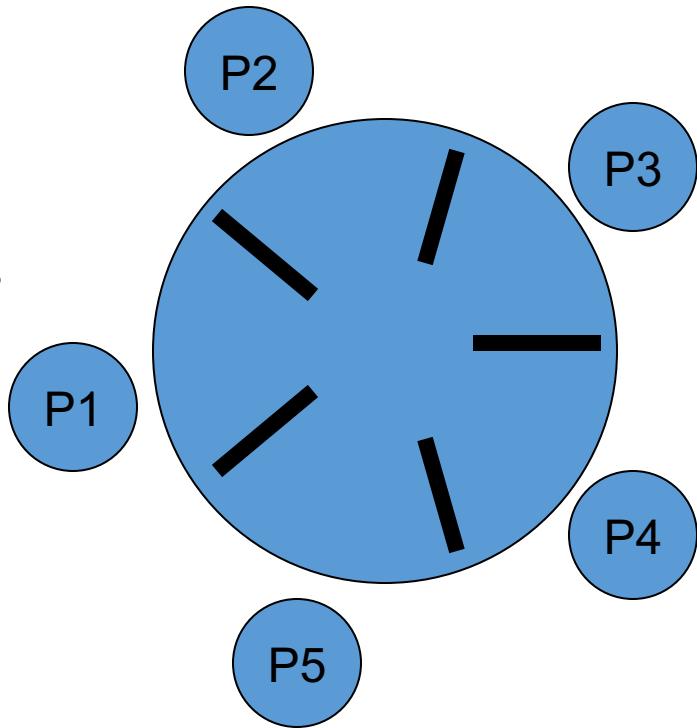
Dining Philosophers Problem

- Deadlock-free 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 chopsticks *only if both are free.*
 - This requires protection of critical sections to test if both chopsticks are free before grabbing them.
 - We'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
 - for now, we'll focus on breaking deadlock



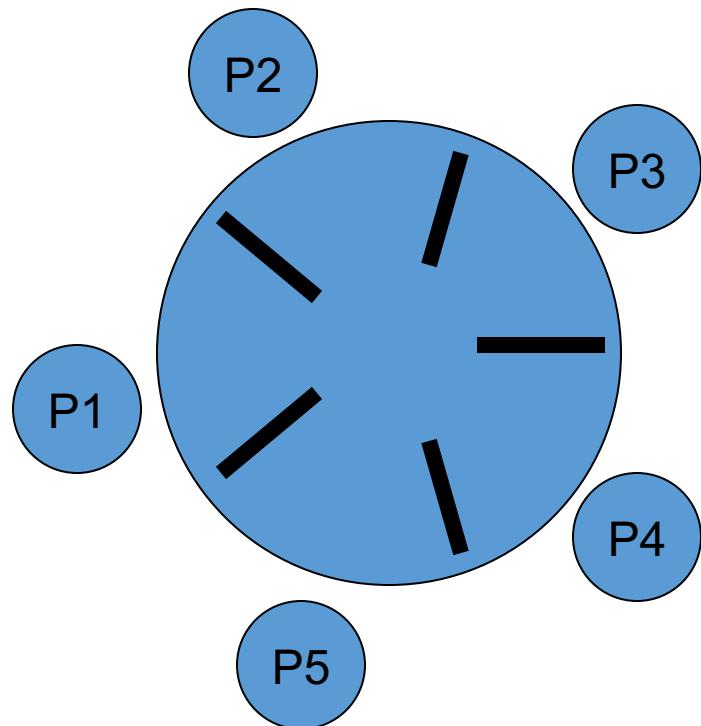
Monitor-based Solution to Dining Philosophers

- Key insight: Pick up 2 chopsticks only if both are free
 - this avoids deadlock
 - **reward insight:** a philosopher moves to his/her eating state only if both neighbors are not in their eating states
 - thus, need to define a state for each philosopher



Monitor-based Solution to Dining Philosophers (2)

- 2nd insight: if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
 - thus, states of each philosopher are:
thinking, hungry, eating
 - thus, need condition variables to signal() waiting hungry philosopher(s)
- Also need to Pickup() and Putdown() chopsticks



Dining Philosophers: Monitor-based Solution

```
philosopher (int i)
{
    while (1) {
        //Think
        DiningPhilosophers.pickup(i);
        // pick up chopsticks and eat
        DiningPhilosophers.putdown(i);
    }
}
```

Dining Philosophers: Monitor-based Solution

monitor DP

{

```
enum {thinking, hungry, eating} state[5];  
condition self[5]; //to block a philosopher when hungry
```

```
void pickup(int i) {
```

```
    //Set state[i] to hungry
```

```
    //If at least one neighbor is eating, (test the neighbors)
```

```
        //      block on self[i]
```

```
    //Otherwise return
```

```
}
```

```
void putdown(int i) {
```

```
    //Change state[i] to thinking and signal neighbors
```

```
        //  in case they are waiting to eat
```

```
}
```



```
void test(int i) {  
    //Check if both neighbors of i are not eating  
    // and i is hungry  
    //If so,  
    //    set state[i] to eating  
    //    and signal philosopher i  
}  
}
```

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

```
}
```



University of Colorado
Boulder

Monitor-based Solution to Dining Philosophers (3)

```
monitor DP {  
    status state[5];  
    condition self[5];  
    Pickup(int i);  
    Putdown(int i);  
    test();  
    init();  
}
```

```
philosopher (int i)  
{  
    while (1) {  
        //Think  
        DiningPhilosophers.pickup(i);  
        // pick up chopsticks and eat  
        DiningPhilosophers.putdown(i);  
    }  
}
```

- Each philosopher i runs pseudo-code:

```
DP.Pickup( $i$ );  
... // eat – grab both  
chopsticks  
DP.Putdown( $i$ );
```



Monitor-based Solution to Dining Philosophers (4)

```
monitor DP {  
    status state[5];  
    condition self[5];
```

```
    atomic {  
        Pickup(int i) {  
            state[i] = hungry;  
            test(i);  
            if(state[i]!=eating)  
                self[i].wait;  
        }  
    }
```

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

- Pickup chopsticks (atomic)
 - indicate that I'm hungry
 - Atomically test if both my left and right neighbors are not eating. If so, then atomically set my state to eating.
 - if unable to eat, wait to be signaled

- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

... monitor code continued next slide ...



University of Colorado
Boulder

Monitor-based Solution to Dining Philosophers (5)

... monitor code continued from previous slide...

...

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

```
init() {  
    for i = 0 to 4  
        state[i] = thinking;  
}
```

```
} // end of monitor
```

University of Colorado
Boulder

- Put down chopsticks (atomic)
 - if left neighbor $L=(i+1)\%5$ is hungry and both of L 's neighbors are not eating, set L 's state to eating and wake it up by signaling L 's CV
- Thus, eating philosophers are the ones who (eventually) turn waiting hungry neighbors into active eating philosophers
 - not all eating philosophers trigger the transformation
 - At least one eating philosophers will be the trigger

Complete Monitor-based Solution to Dining Philosophers

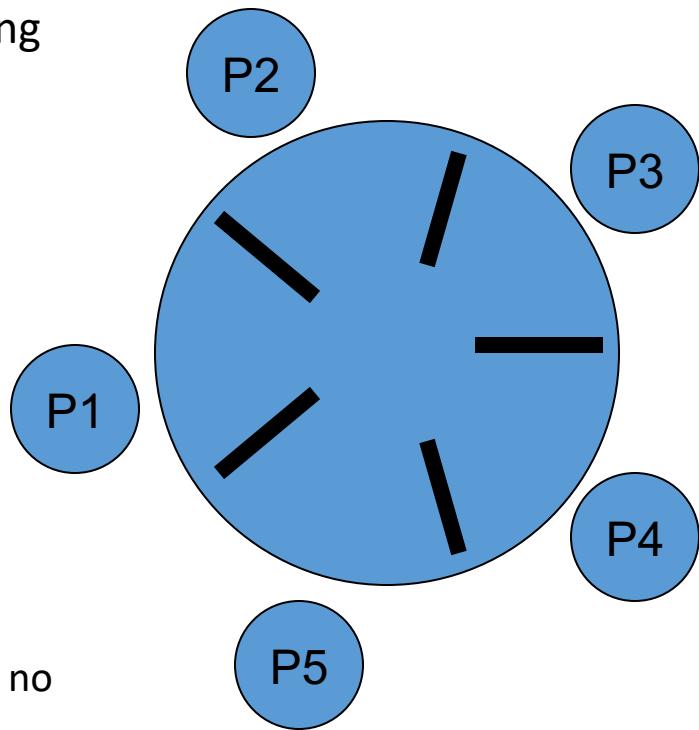
```
monitor DP {  
    status state[5];  
    condition self[5];  
  
    Pickup(int i) {  
        state[i] = hungry;  
        test(i);  
        if(state[i]!=eating)  
            self[i].wait;  
    }  
  
    test(int i) {  
        if (state[(i+1)%5] != eating &&  
            state[(i-1)%5] != eating &&  
            state[i] == hungry)  
        {  
            state[i] = eating;  
            self[i].signal();  
        }  
    }  
  
    Putdown(int i) {  
        state[i] = thinking;  
        test((i+1)%5);  
        test((i-1)%5);  
    }  
  
    init() {  
        for i = 0 to 4  
            state[i] = thinking;  
    }  
} // end of monitor
```

- Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
 - deadlock-free
 - mutually exclusive in that no 2 neighbors can eat simultaneously



DP Monitor Deadlock Analysis

- Try various scenarios to verify for yourself that deadlock does not occur in them
- Start with one philosopher P1
- Now suppose P2 arrives to the left of P1 while P1 is eating
 - What is the perspective from P1?
 - What is the perspective from P2?
- Now suppose P5 arrives to the right of P1 while P1 is eating and P2 is waiting
 - Perspective from P1?
 - Perspective from P5?
 - Perspective from P2?
- Suppose P2 arrives while both P1 and P3 are eating
 - If P1 finishes first, it can't wake up P2
 - But when P3 finishes, its call to test(P2) will wake up P2, so no deadlock
- Suppose there are 6 philosophers and the evens are eating.
How do the odds get to eat?



DP Monitor Solution

- Note that starvation is still possible in the DP monitor solution
 - Suppose P1 and P3 arrive first, and start eating, then P2 arrives and sets its state to hungry and blocks on its CV
 - When P1 ends eating, it will call test(P2), but nothing will happen, i.e. P2 won't be signaled because the signal only occurs inside the if statement of test, and the if condition is not satisfied
 - Next, P1 can eat again, repeatedly, starving P2

Complete Monitor-based Solution to Dining Philosophers

```
monitor DP {  
    status state[5];  
    condition self[5];  
  
    Pickup(int i) {  
        state[i] = hungry;  
        test(i);  
        if(state[i]!=eating)  
            self[i].wait;  
    }  
  
    test(int i) {  
        if (state[(i+1)%5] != eating &&  
            state[(i-1)%5] != eating &&  
            state[i] == hungry) {  
  
            state[i] = eating;  
            self[i].signal();  
        }  
    }  
  
    Putdown(int i) {  
        state[i] = thinking;  
        test((i+1)%5);  
        test((i-1)%5);  
    }  
  
    init() {  
        for i = 0 to 4  
            state[i] = thinking;  
    }  
} // end of monitor
```

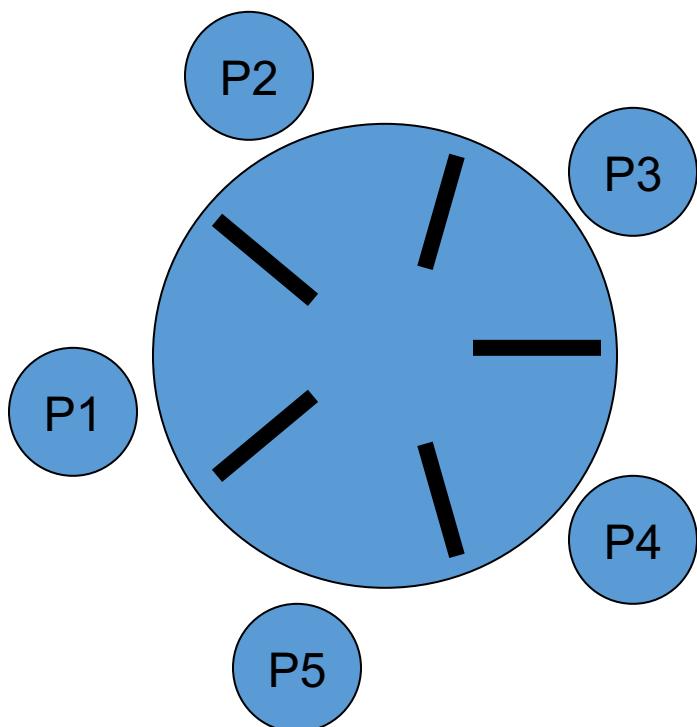


DP Solution Analysis

- Signal() happening before the wait() doesn't matter
 - Signal() in Pickup() has no effect the 1st time
 - Signal() called in Putdown() is the actual wakeup

Deadlock Free Solution

- Monitors for implicit mutual exclusion
- Condition variables for ordering
 - cv.wait()
 - cv.signal() differs from semaphore's signal()
- Monitor-based solution to Dining Philosophers



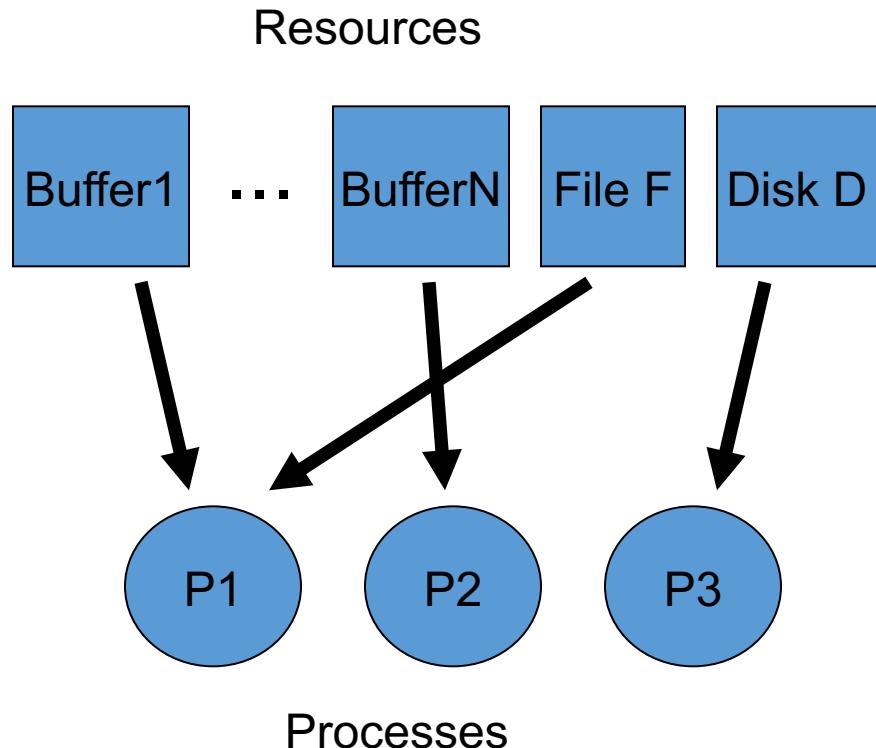
Deadlock

Deadlock: General Solution?

- Want a general solution to deadlock that is not restricted to the solutions for the 3 classic problems of DP, R/W, and BB P/C
- **A set of processes is in a deadlock state when every process in the set is waiting for an event (e.g. release of a resource) that can only be caused by another process in the set**
 - You have a circular dependency
- multithreaded and multi-process applications are good candidates for deadlock
 - thread-thread deadlock within a process
 - process-process deadlock

Modeling Deadlock

- Develop a model so we can see circular dependency
 - to use a resource, a process must
 1. request() a resource -- must *wait* until it's available
 2. use() or hold() a resource
 3. release() a resource
 - thus, we have resources and processes
 - Most of the following discussion will focus on reusable resources



P1 holds Buffer 1 and File F
P2 holds Buffer N
P3 holds Disk D

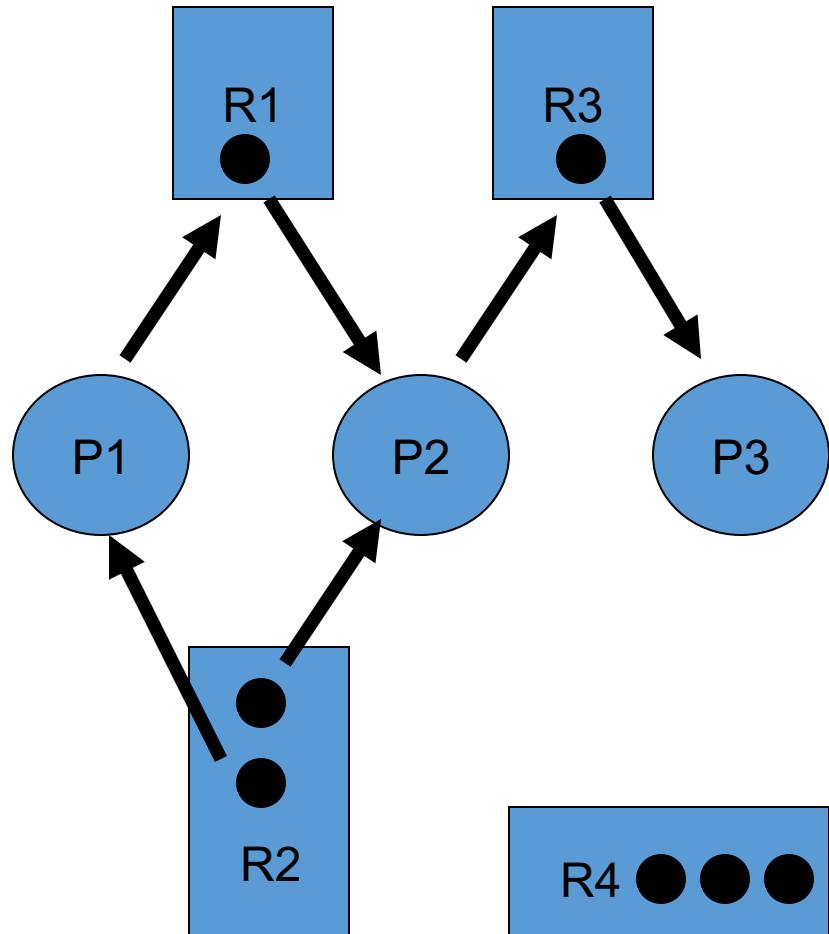
Modeling Deadlock using Directed Graph

- a *resource allocation graph* can be used to model deadlock
 - try to represent deadlock by a *directed graph* $D(V,E)$, consisting of
 - vertices V : namely processes and resources
 - and edges E :
 - a **request()** for a resource R_j by a process P_i is signified by a directed arrow from **process $P_i \rightarrow R_j$**
 - a process P_i will **hold()** a resource R_j via a **directed arrow $R_j \rightarrow P_i$**



Modeling Deadlock

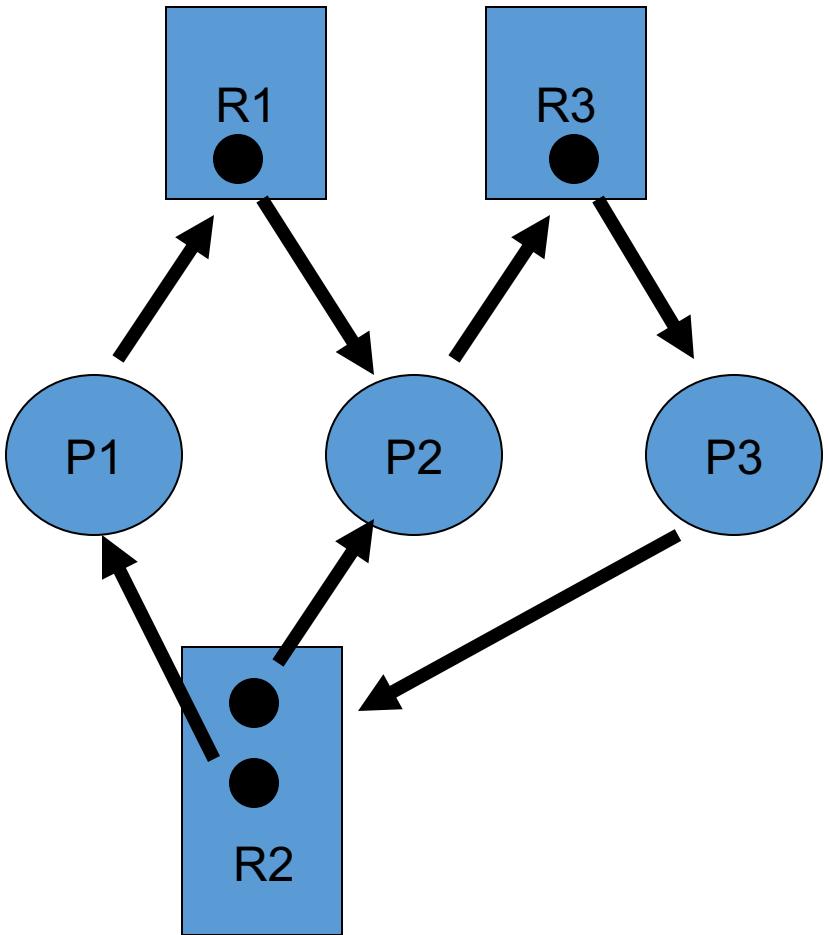
- Example 1:
 - P1 wants resource R1 but that is held by P2
 - P2 wants resource R3 but that is held by P3
 - Also, P1 holds an *instance* of resource R2, and
 - P2 holds an instance of R2
 - There is no deadlock
 - if the graph contains no cycles or loops, then there is no deadlock



Modeling Deadlock

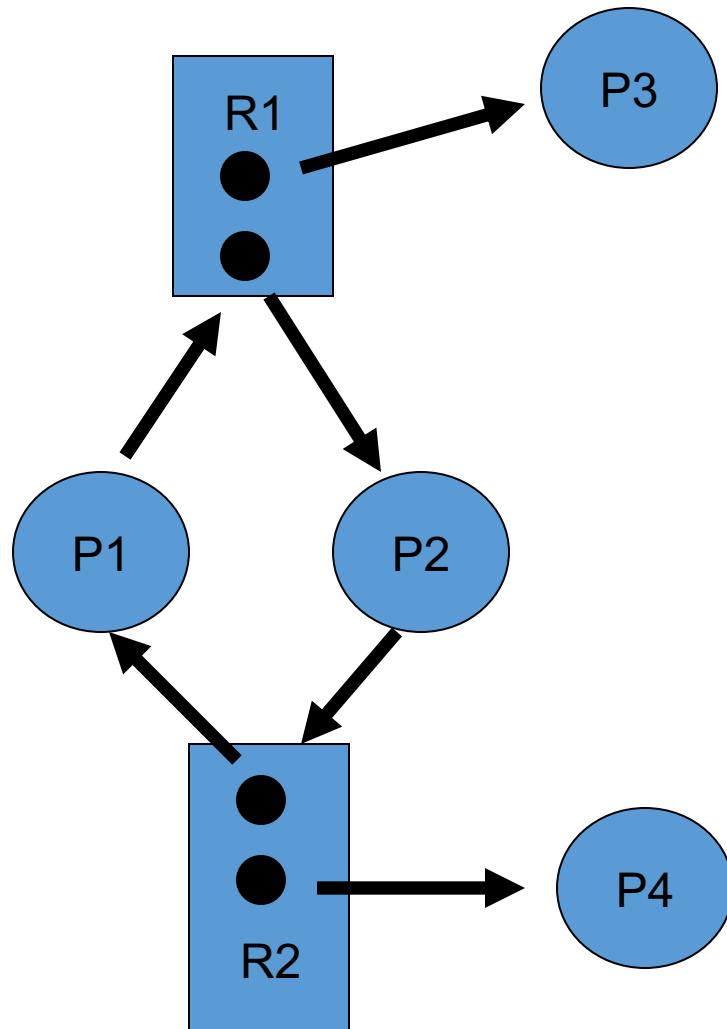
- Example 2:

- same graph as before, except now P3 requests instance of R2
- Deadlock occurs!
 - P3 requests R2, which is held by P2, which requests R3, which is held by P3 - this is a loop
 - $P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
 - If P1 could somehow release an instance of R2, then we could break the deadlock
 - But P1 is part of a second loop:
 - $P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
 - So P1 can't release its instance of R2
- if the graph contains cycles or loops, then there *may be the possibility* of deadlock
 - but does a loop guarantee that there is deadlock?



Modeling Deadlock

- Example 3:
 - there is a loop:
 - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$
 - In this case, there is no deadlock
 - either P_3 can release an instance of R_1 , or P_4 can release an instance of R_2
 - this breaks any possible deadlock cycle
 - if the graph contains cycles or loops, then there *may be the possibility* of deadlock, but this is not a guarantee of deadlock

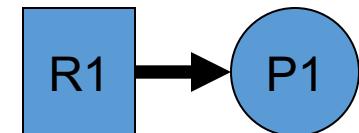


Necessary Conditions for Deadlock

The following 4 conditions must hold simultaneously for deadlock to arise:

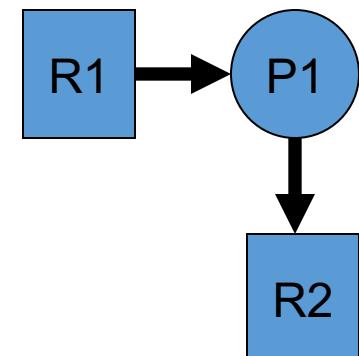
1. Mutual exclusion

- at least 1 resource is held in a non-sharable mode. Other requesting processes must wait until the resource is released



2. Hold and wait

- a process holds a resource while requesting (and waiting for) another one

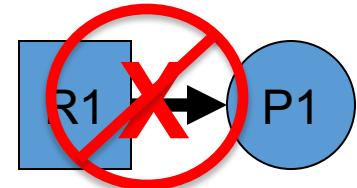


Necessary Conditions for Deadlock

The following 4 conditions must hold simultaneously for deadlock to arise: (continued)

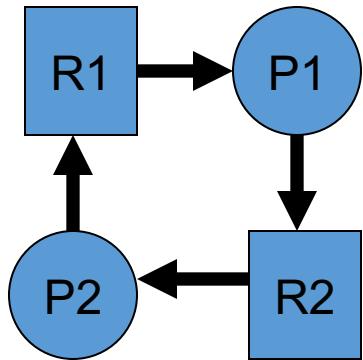
3. No preemption:

- resources cannot be preempted and can only be released voluntarily by the process holding them, after the process is finished. No OS intervention is allowed. A process cannot withdraw its request.



4. Circular wait

- A set of n waiting processes $\{P_0, \dots, P_{n-1}\}$ must exist such that P_i waits for a resource held by $P_{(i+1)\%n}$



Solutions to Handling Deadlocks

1. Prevention by OS

- provide methods to guarantee that at least 1 of the 4 necessary conditions for deadlock does not hold

2. Avoidance by OS

- the OS is given advanced information about process requests for various resources
- this is used to determine whether there is a way for the OS to satisfy the resource requests and avoid deadlock



Solutions to Handling Deadlocks (2)

3. Detection and Recovery by OS

- Analyze existing system resource allocation, and see if there is a **sequence of releases** that satisfies every process' needs.
- If not, then deadlock is detected, so must recover – drastic action needed, such as killing the affected processes!

Solutions to Handling Deadlocks (3)

4. Application-level solutions (OS Ignores and Pretends)

- the most common approach, e.g. UNIX and Windows, based on the assumption that **deadlock is relatively infrequent**
- it's up to the application programmer to implement mechanisms that prevent, avoid, detect and deal with application-level deadlock
- **Map your problem to known deadlock-free solutions: e.g. Bounded Buffer P/C, Readers/Writers problems, Dining Philosophers, ...**



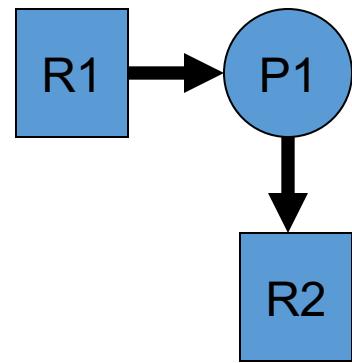
Deadlock Prevention: Mutual Exclusion

- Prevent the *mutual exclusion* condition #1 from coming true
 - Many resources are non-sharable and must be accessed in a mutually exclusive way
 - example: a printer should print a file X to completion before printing a file Y. a printer should not print half of file X, and then print the first half of file Y on the same paper
 - thus, it is unrealistic to prevent mutual exclusion



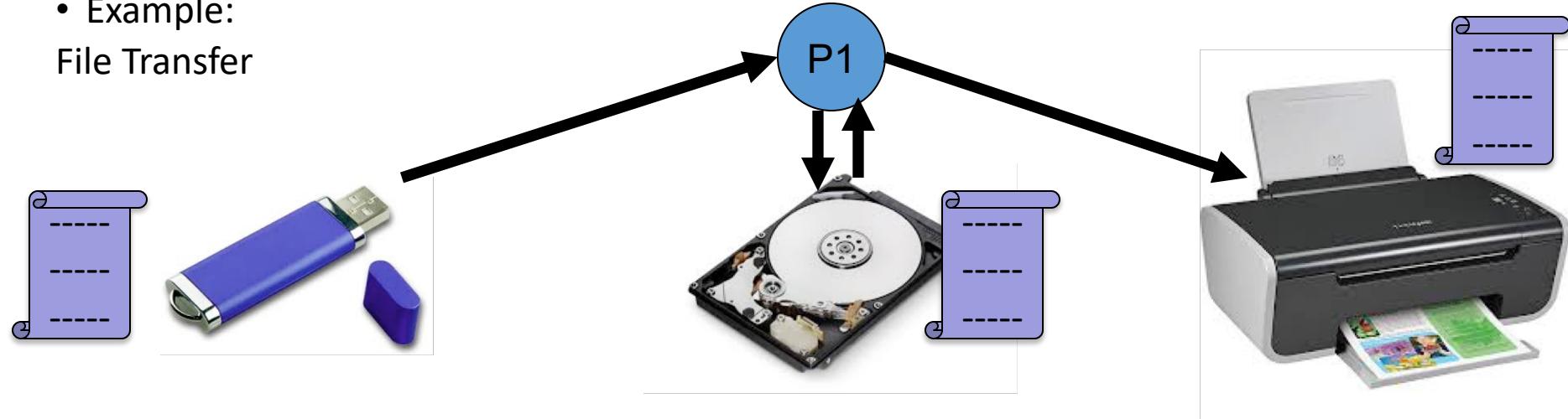
Deadlock Prevention: Hold and Wait

- Prevent the hold and wait condition #2 from coming true
 - prevent a process from holding resources and requesting others
 - *Solution I:* request all resources at process creation
 - *Solution II:* release all held resources before requesting a set of new ones simultaneously
 - *Solution III:* only allow a process to hold one resource at a time



Deadlock Prevention: Hold and Wait

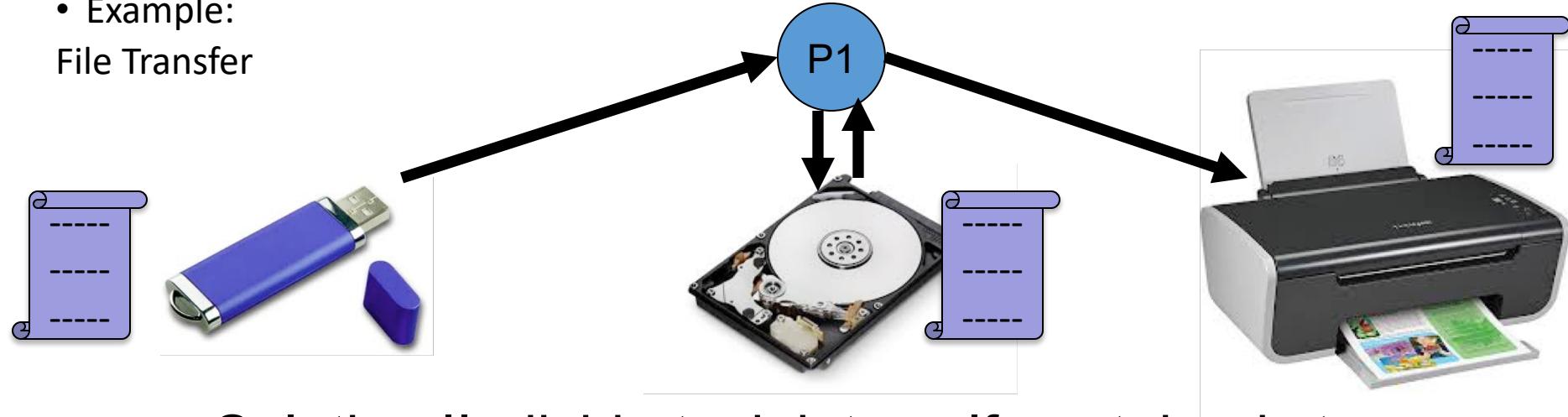
- Example:
File Transfer



- a process reads file from USB drive and writes it to hard drive, retrieves the file, then sends the file to the printer
- Solution I: request the USB drive, hard drive, and printer at process creation

Deadlock Prevention: Hold and Wait

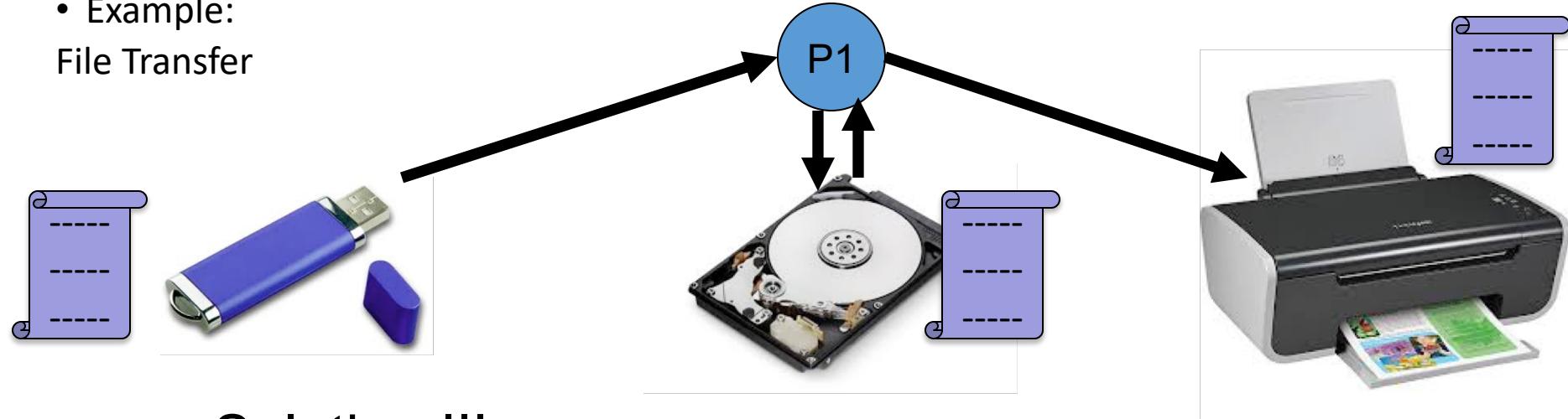
- Example:
File Transfer



- Solution II: divide task into self-contained stages that release all & then request all resources
 - obtain the USB and hard drive together for the file transfer, then release both together
 - next obtain the hard drive and printer together for the printing operation, then release both together

Deadlock Prevention: Hold and Wait

- Example:
File Transfer



– Solution III:

- Request the USB drive then release
- Request the hard drive then release
- Request the hard drive again then release
- Request the printer then release

Deadlock Prevention: Hold and Wait

- Disadvantages of Hold-and-wait solutions
 - Solution I: don't know in advance all resources needed
 - Solutions I & II: poor resource utilization
 - a process that is holding multiple resources for a long time may only need each resource for a short time during execution
 - Solution II: possible starvation
 - a process that needs several popular resources simultaneously may have to wait a very long time



Deadlock Prevention: Hold and Wait

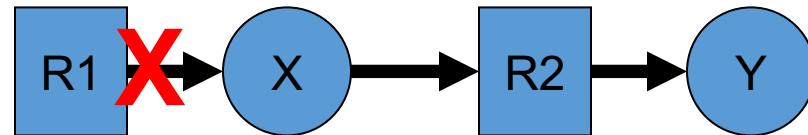
- Disadvantages of Hold-and-wait solutions
 - Solution III: Some processing may require holding more than one resource at a time
 - e.g. writing a file to a printer may require locking both the file and the printer
 - Reading a file from a drive may require locking both the file and the drive

Deadlock Prevention: Hold and Wait

- Example: Dining Philosophers Problem prevented hold-and-wait – How?
 - Enforced a rule that either a philosopher picked up both chopsticks or none at all, i.e. all-or-nothing
 - Hence no holding one chopstick while waiting on the other chopstick

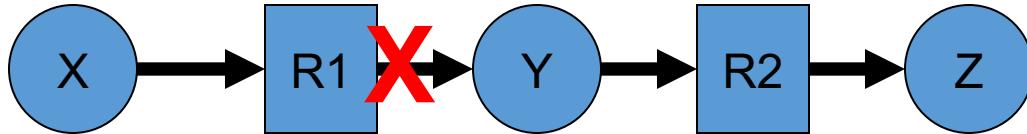
Deadlock Prevention: No Preemption

- Prevent the “No Preemption” condition #3 from coming true
 - allow resources to be preempted
- Policy I:
 - If a Process X requests a held resource, then all resources currently held by X are released.
 - X is restarted only when it can regain all needed resources



Deadlock Prevention: No Preemption

- Policy II:



- If a process X requests a resource held by process Y, then preempt the resource from process Y, but only if Y is waiting on another resource
- Otherwise, X must wait.
- the idea is if Y is holding some resources but is waiting on another resource, then Y has no need to keep holding its resources since Y is suspended

Deadlock Prevention: No Preemption

- Disadvantages:
 - these policies don't apply to all resources, e.g. printers should not be preempted while in the middle of printing, disks should not be preempted while in the middle of writing a block of data
 - can result in unexpected behavior of processes, since an **application developer may not know a priori which policy is being used**



Deadlock Prevention: Circular Wait

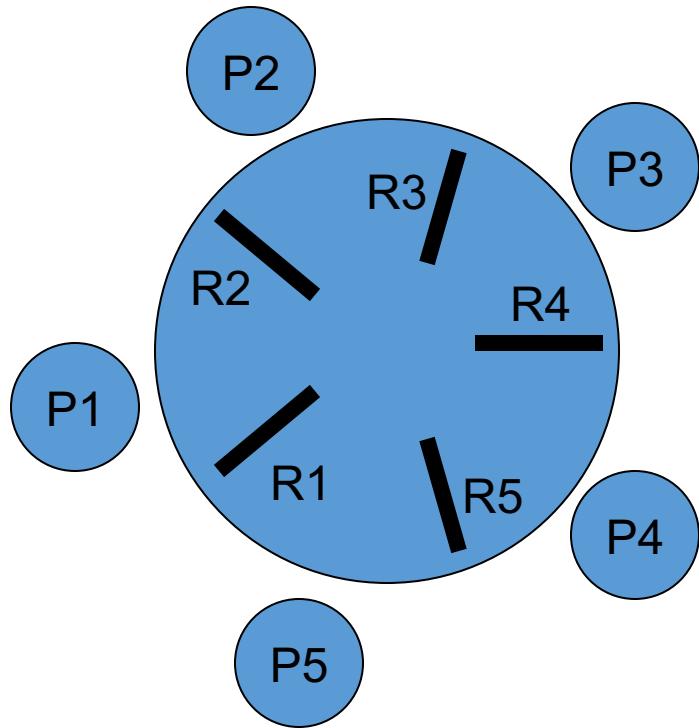
- Prevent the *circular wait* condition #4 from coming true
 - Solution I: a process can only hold 1 resource at a time
 - disadvantage: in some cases, a process needs to hold multiple resources to accomplish a task
 - Solution II: impose a total ordering of all resource types and require each process to request resources in increasing order
 - this prevents a circular wait - see next slide

Deadlock Prevention: Circular Wait

- Solution II example:
 - Order all resources into a list: R_1, R_2, \dots, R_m , where $R_1 < R_2 < \dots < R_m$
 - tape drive = R_1 , disk drive = R_2 , printer = R_{10} , temporary buffer = R_{22}
 - Impose the rule that a process holding R_i can only request R_j if $R_j > R_i$
 - If a process P holds some R_k and requests R_j such that $R_j < R_k$, then the process must release all such R_k , acquire R_j , then reacquire R_k

Deadlock Prevention: Circular Wait

- Applying ordering of resources to break circular waiting in the Dining Philosophers Problem
 - $R1 < R2 < R3 < R4 < R5$
 - Deadlock happened when all processes first requested their right chopsticks, then requested their left chopsticks
 - Here, P1 to P4 can all request their right then left chopsticks
 - *But Process P5 requests its left (R1) then right (R5) chopstick due to ordering*
 - thus, P5 blocks on R1, not R5, which breaks any possibility of a circular deadlock



Deadlock Prevention: Circular Wait

- Disadvantages of ordering resources:
 - can lead to poor performance, due to releasing and then reacquiring resources
 - Difficult to implement in a dynamic resource environment
 - Coming up with a global scheme for numbering resources

Deadlock Avoidance

- Goal: analyze the system state to see if there is a way to avoid deadlock.
- At startup, each process provides OS with information about all of its requests and releases for resources R_i
 - e.g. batch jobs know a priori which resources they'll request, when, and in which order
- OS decides whether deadlock will occur at run time

Deadlock Avoidance

- Disadvantage: need a priori info
- Simple strategy:
 - each process specifies a maximum claim
 - knowing all individual future requests and releases is difficult
 - Having each process estimate its maximum demand for resources is easier and not completely unreasonable
- A *resource allocation state* is defined by
 - # of available resources
 - # of allocated resources to each process
 - maximum demands by each process



Deadlock Avoidance

- A system is in a *safe* state if there exists a *safe sequence* of processes $\langle P_1, \dots, P_n \rangle$ for the current resource allocation state
 - A sequence of processes is safe if for each P_i in the sequence, the resource requests that P_i can still make can be satisfied by:
 - currently available resources + all resources held by all previous processes in the sequence $P_j, j < i$
 - If resources needed by P_i are not available, P_i waits for all P_j to release their resources

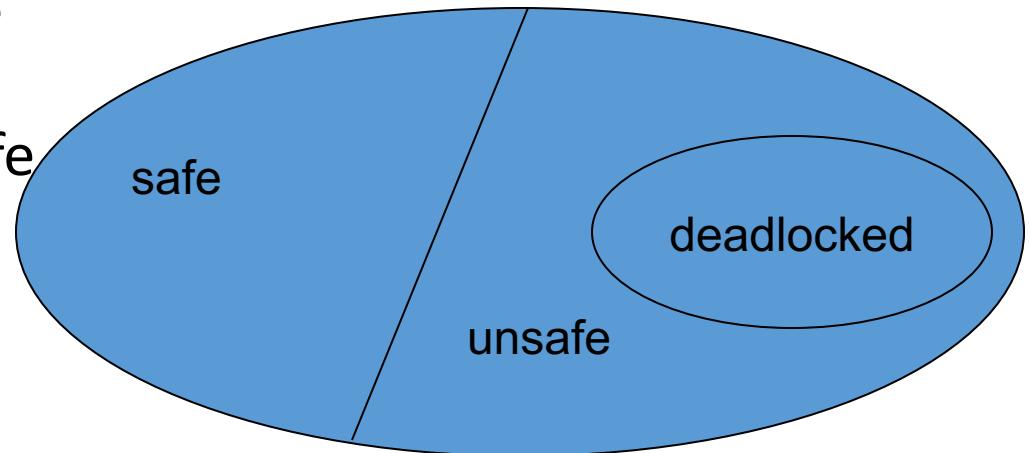


Deadlock Avoidance

- Intuition for a safe state: given that the system is in a certain state, we want to find at least one “way out of trouble”
 - i.e. find a sequence of processes that, even when they demand their maximum resources, won’t deadlock the system
 - this is a worst-case analysis
 - it may be that during the normal execution of processes, none ever demands its maximum in a way that causes deadlock
 - to perform a more optimal (less than worst-case) analysis is more complex, and also requires a record of future accesses

Deadlock Avoidance

- A safe state provides a safe “escape” sequence
- A deadlocked state is unsafe
- An unsafe state is not necessarily deadlocked
- A system may transition from a safe to an unsafe state if a request for resources is granted
 - ideally, check with each request for resources whether the system is still safe



Deadlock Avoidance

- Example 1:
 - 12 instances of a resource
 - At time t0, P0 holds 5, P1 holds 2, P2 holds 2
 - Available = 3 free instances

processes	max needs	allocated
P0	10	5
P1	4	2
P2	9	2

Deadlock Avoidance

- Example 1 (cont):
 - Is the system in a safe state? Can I find a safe sequence?
 - Yes, I claim the sequence $\langle P_1, P_0, P_2 \rangle$ is safe.
 - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
 - Then P_1 releases all of its resources, so 5 free
 - Next, P_0 requests its max (currently has 5, so needs 5 more) and holds 10, so that now 0 free
 - Then P_0 releases all its held resources, so 10 free
 - Next P_2 requests its max of 9, leaving 3 free and then releases them all



Deadlock Avoidance

- Example 1 (cont):
 - Is the system in a safe state? Can I find a safe sequence?
 - Yes the sequence $\langle P_1, P_0, P_2 \rangle$ is safe, and is able in the worst-case to request maximum resources for each process in the sequence, and release all such resources for the next process in the sequence
 - Can this system avoid deadlock? Yes, we can find a safe sequence.



Mars Rover Pathfinder

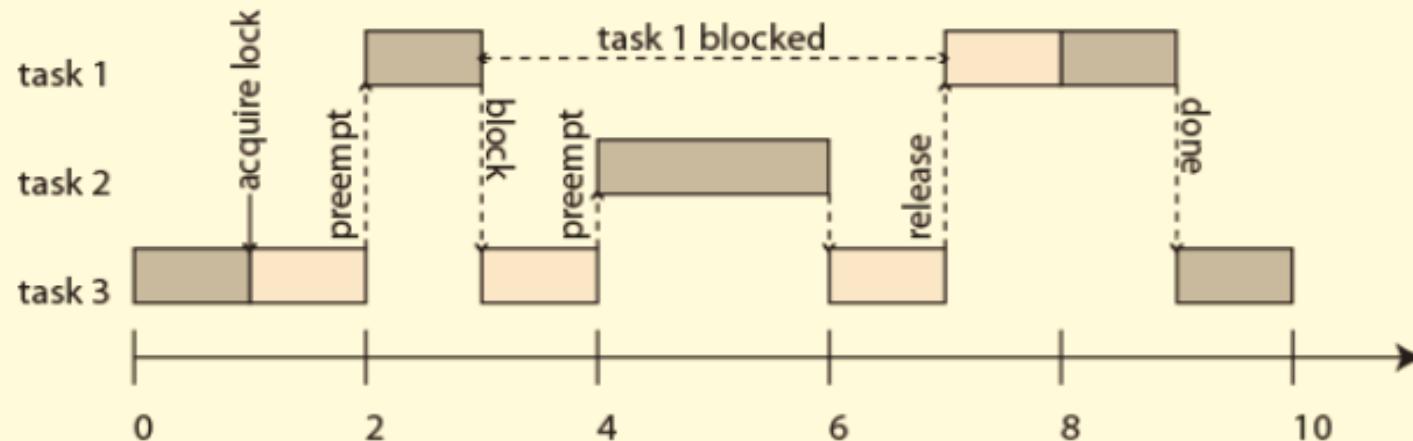
The Mars Rover Pathfinder landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. The problem was diagnosed on the ground as priority inversion, where a low priority meteorological task was holding a lock blocking a high-priority task while medium priority tasks executed.

Source: RISKS-19.49 on the comp.programming.threads newsgroup, December 07, 1997, by Mike Jones (mbj@MICROSOFT.com).



Priority Inversion problem

Priority Inversion: A Hazard with Mutexes



Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

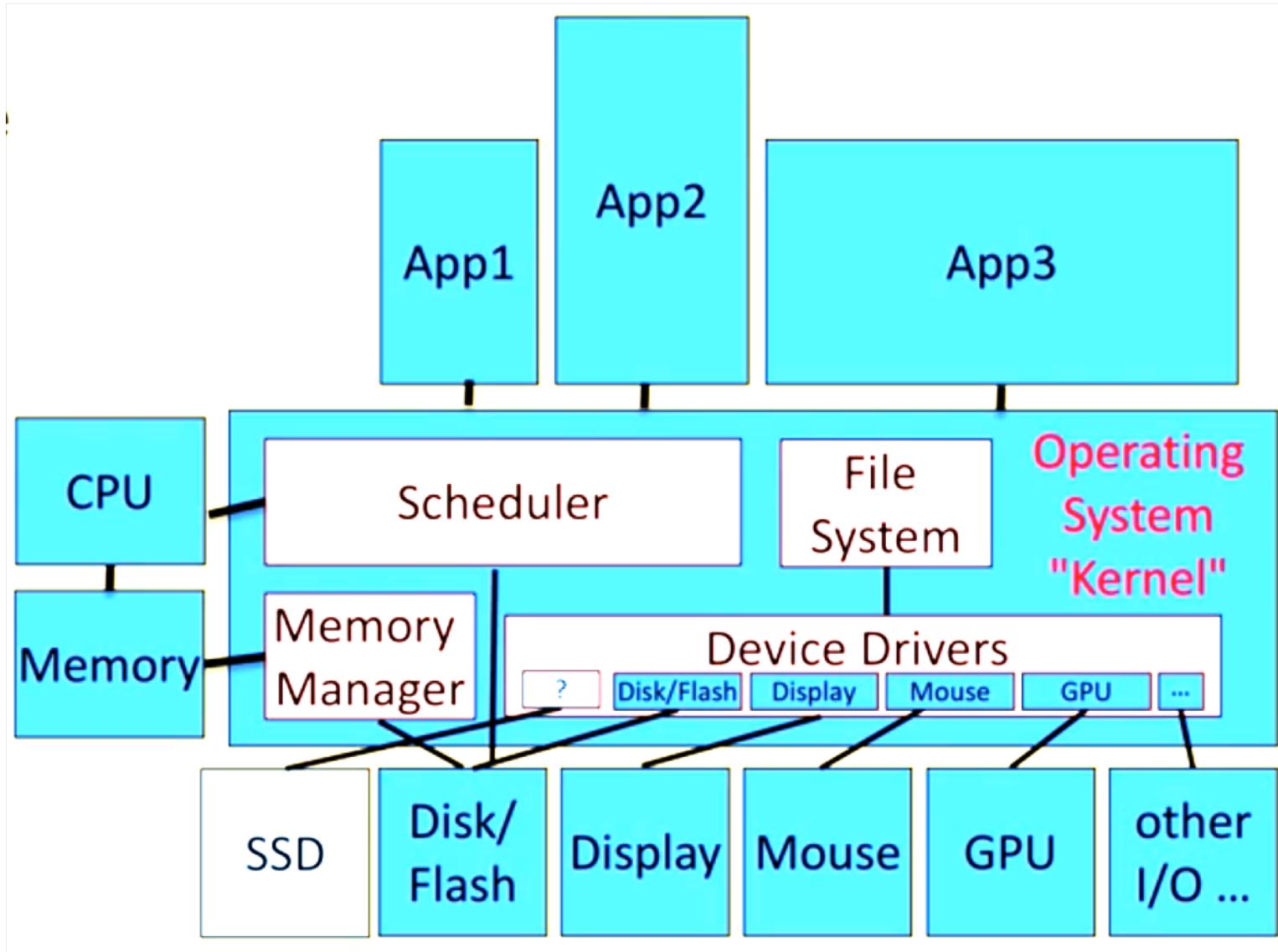


Midterm Review



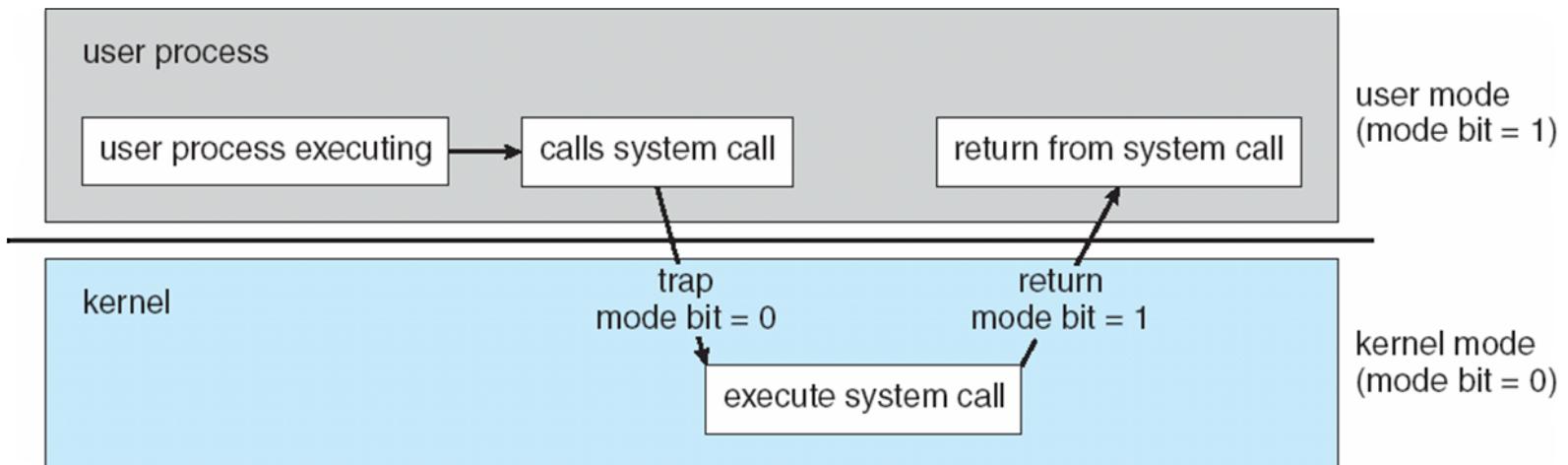
Overview OS

Operating Systems (OS)



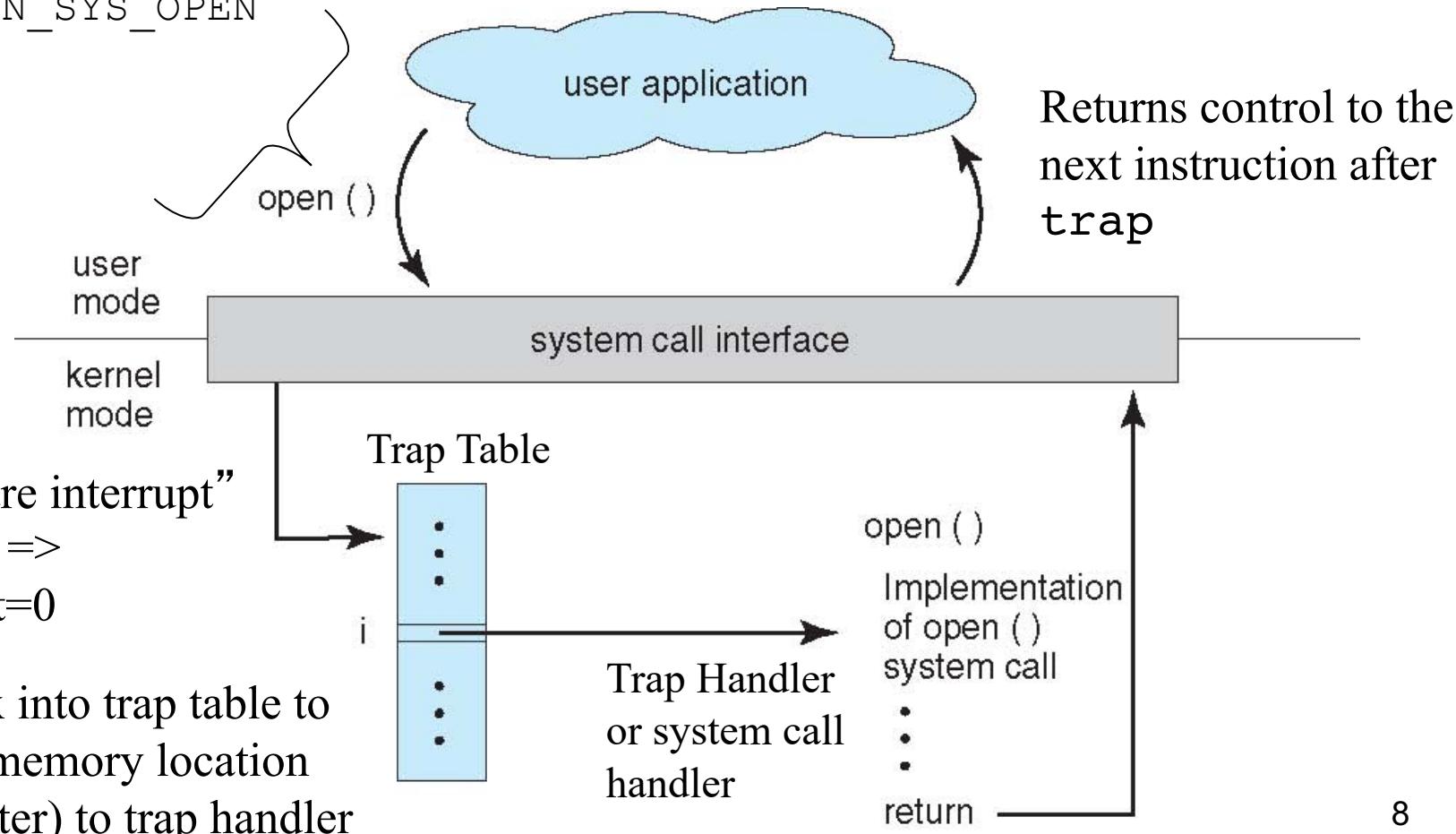
System Calls

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

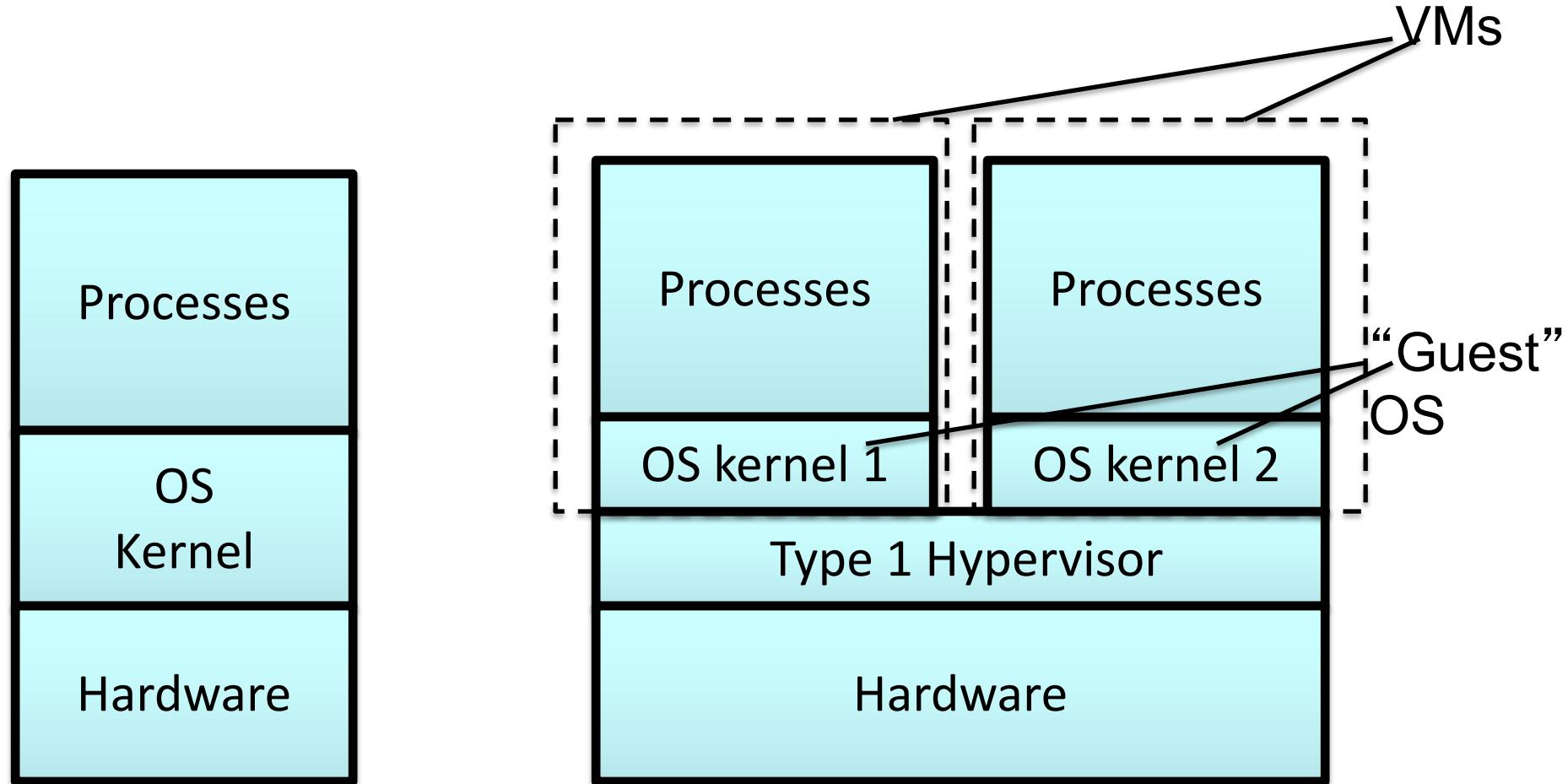


API – System Calls – OS Relationship

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



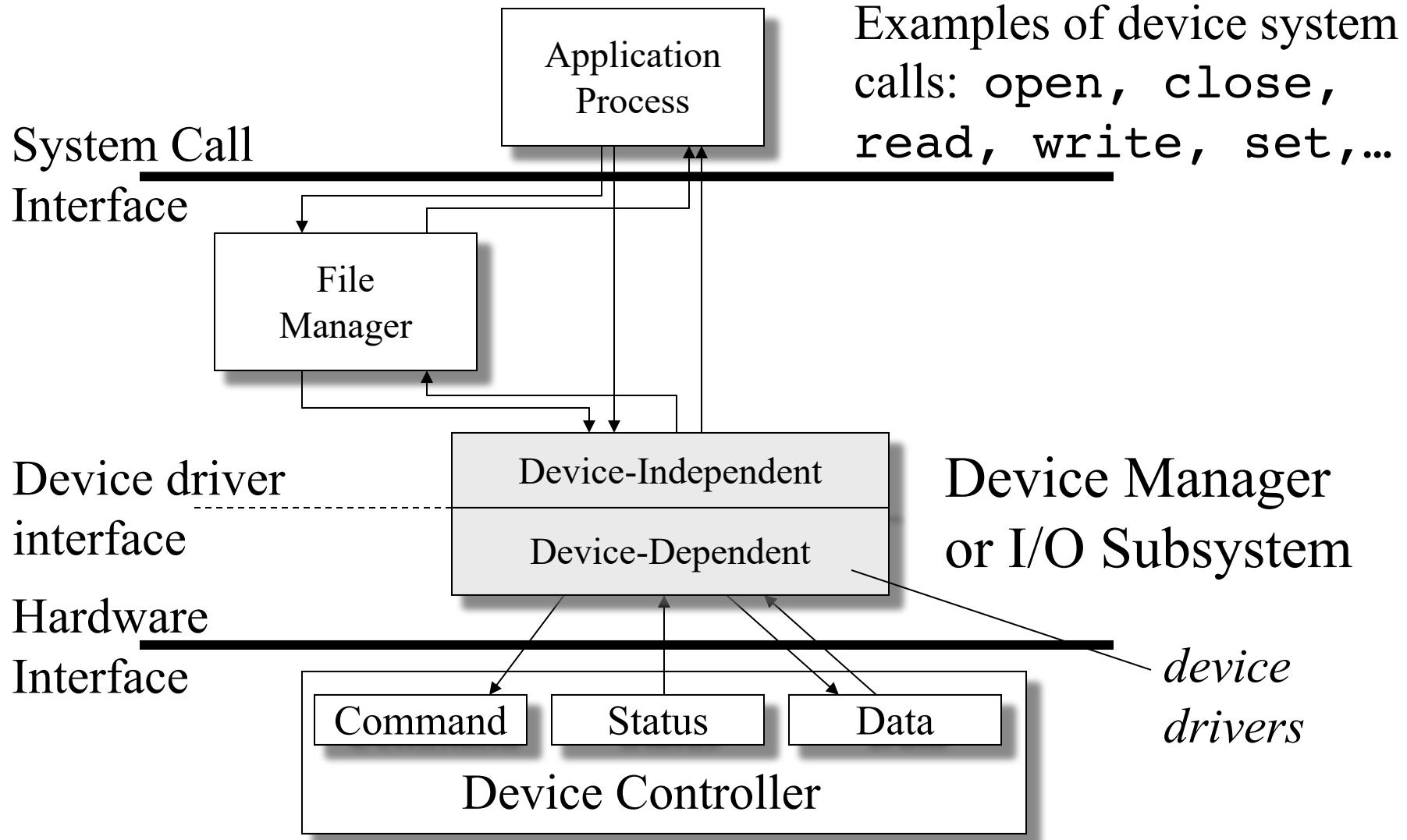
Virtual Machines



Traditional OS

A *Type 1 Hypervisor* provides a virtualization layer for guest OSs and resides just above the hardware.

Device Management Organization



Loadable Kernel Modules (LKM)

- LKM is an object file that contains code to extend a running kernel
- LKMs can be loaded and unloaded from kernel on demand at runtime
- LKMs offer an easy way to extend the functionality of the kernel without having to rebuild or recompile the kernel again
- LKMs are a simple and efficient way to create programs that reside in the kernel and run in privileged mode
- Most of the drivers are written as LKMs

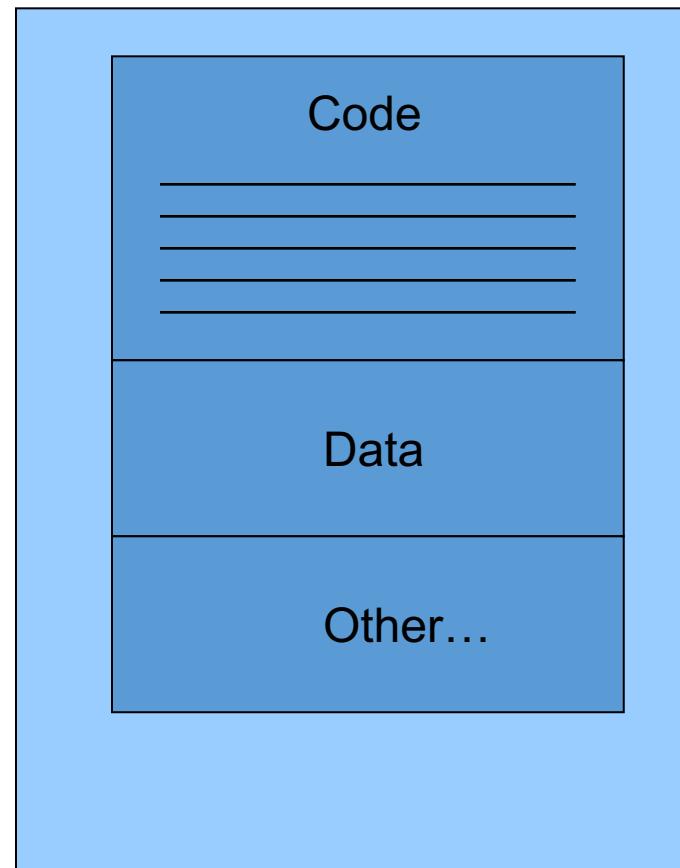


Process

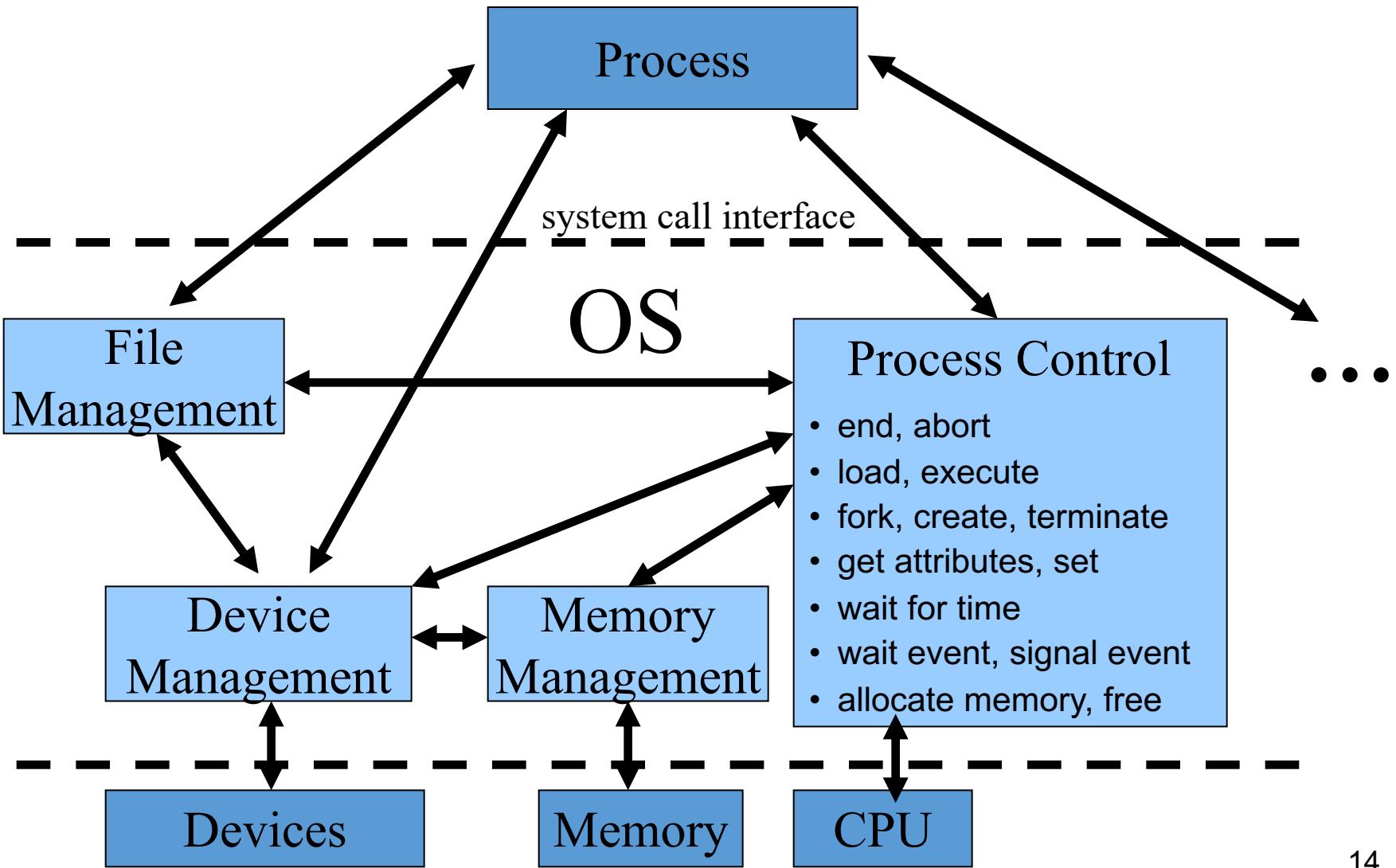
Process

- A software program consists 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

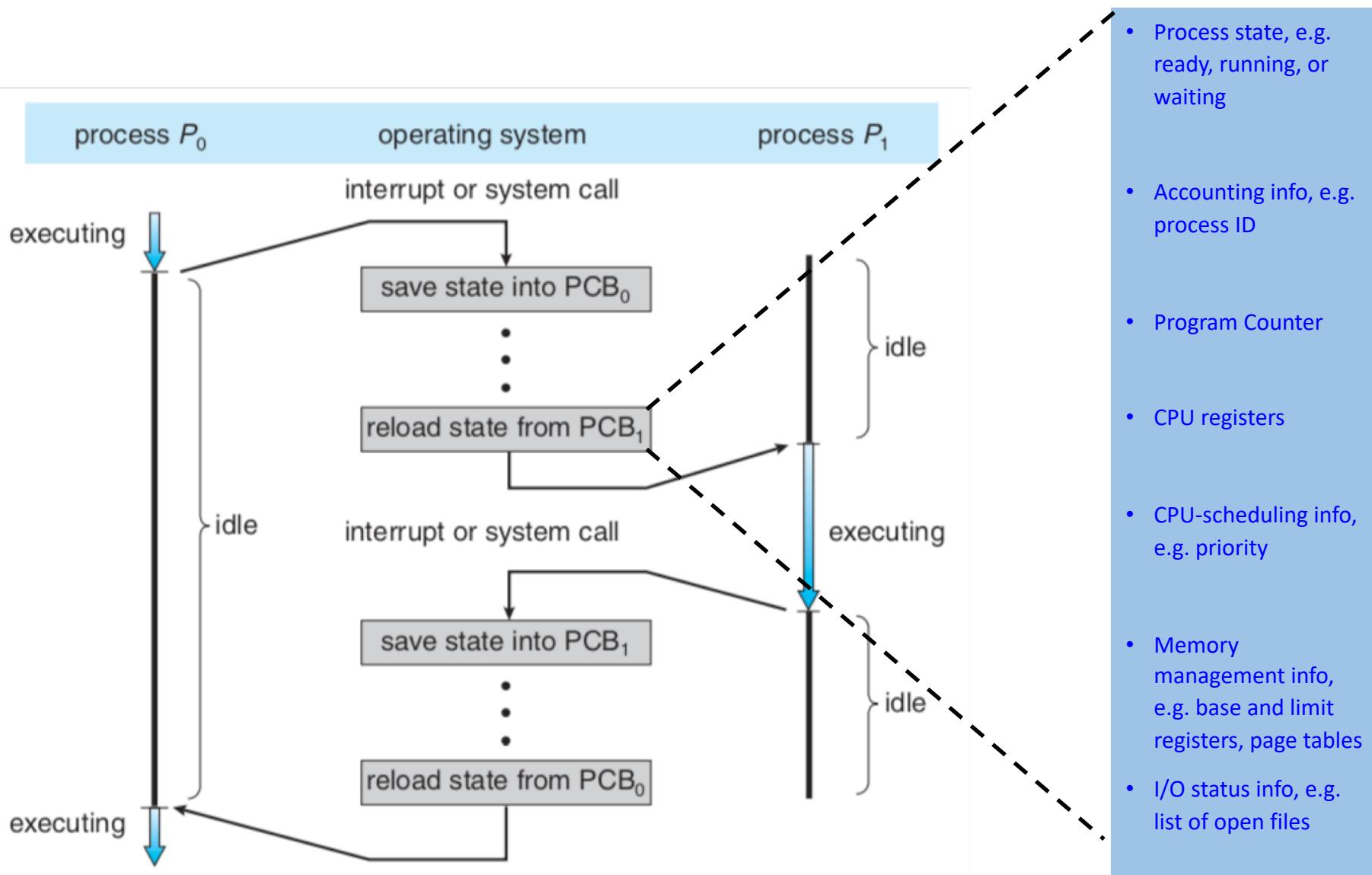
Program P1



Process Management



Context Switching

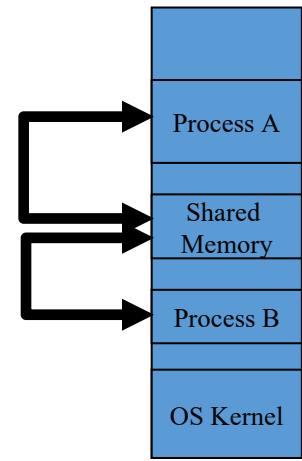
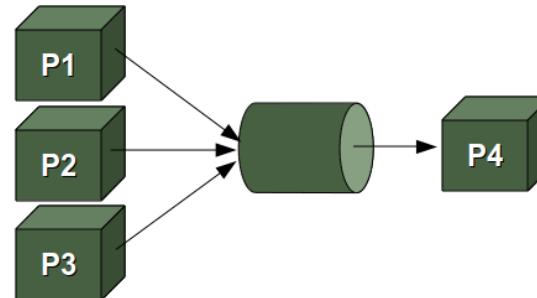




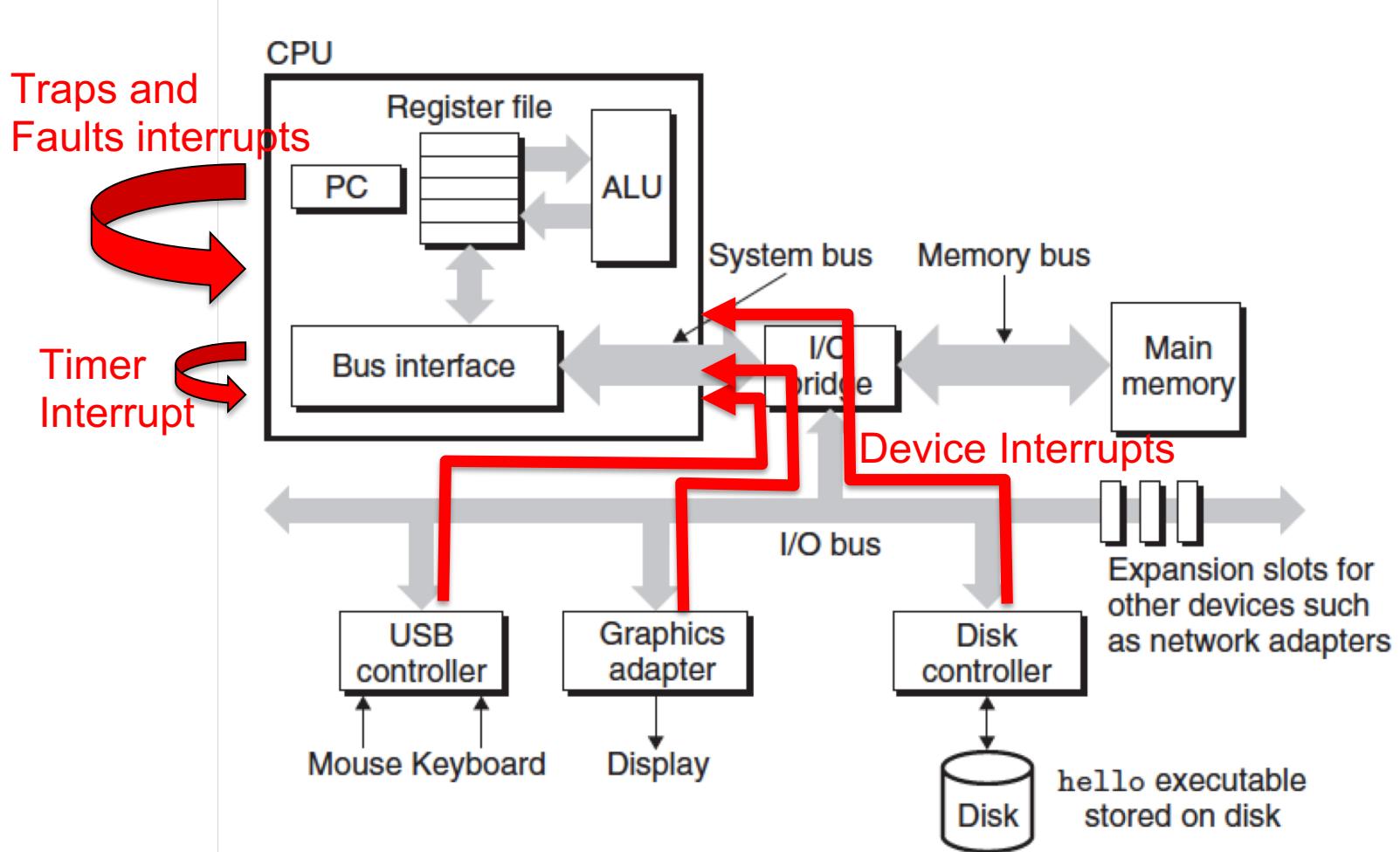
Interprocess Communication (IPC)

Interprocess Communication (IPC)

- Signals / Interrupts
 - Notifying that an event has occurred
- Message Passing
 - Pipes
 - Sockets
- Shared Memory
 - Race conditions
 - Synchronization
- Remote Procedure Calls



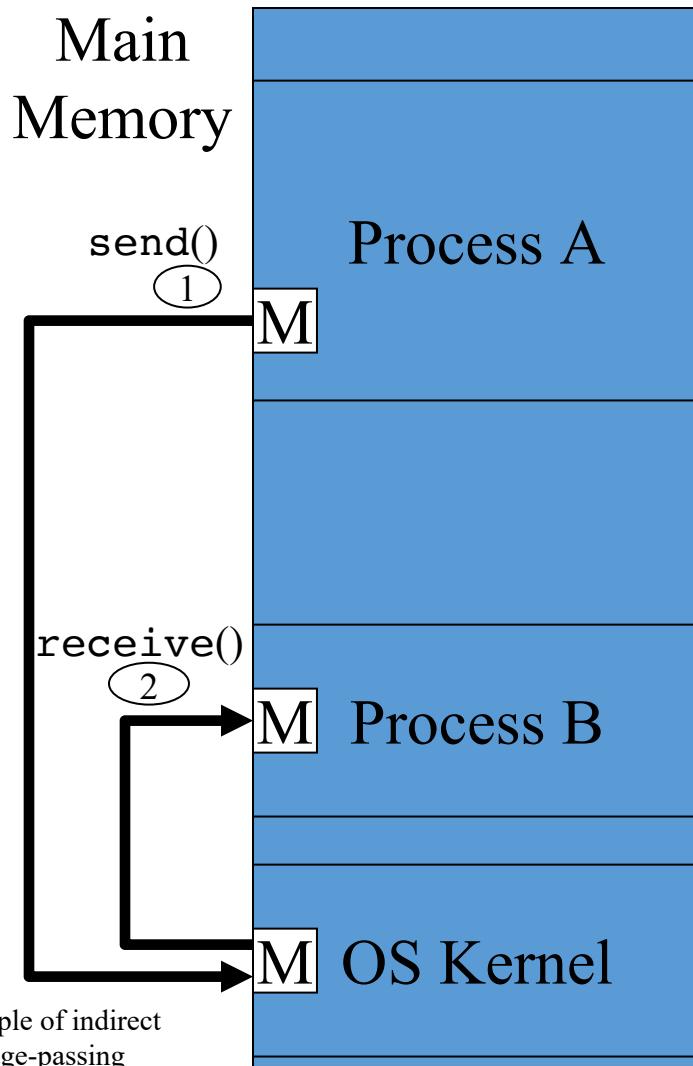
Linux Signals & Interrupts



Signals

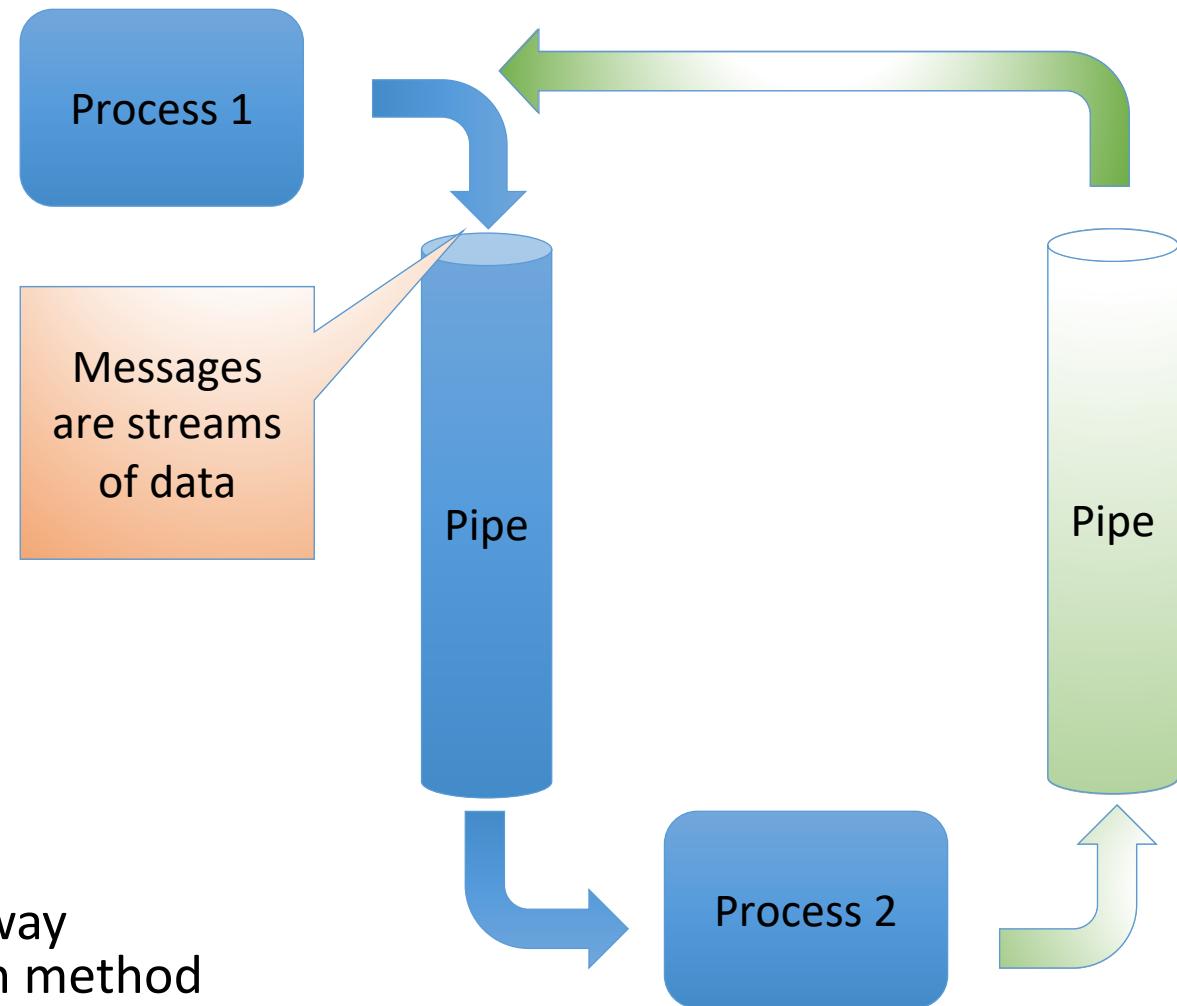
- Kernel-to-Process
- Process-to-Process
- Signals' issues
 - Blocking vs Non-blocking system calls
 - Synchronization vs Asynchronization
 - Race conditions

IPC Message Passing

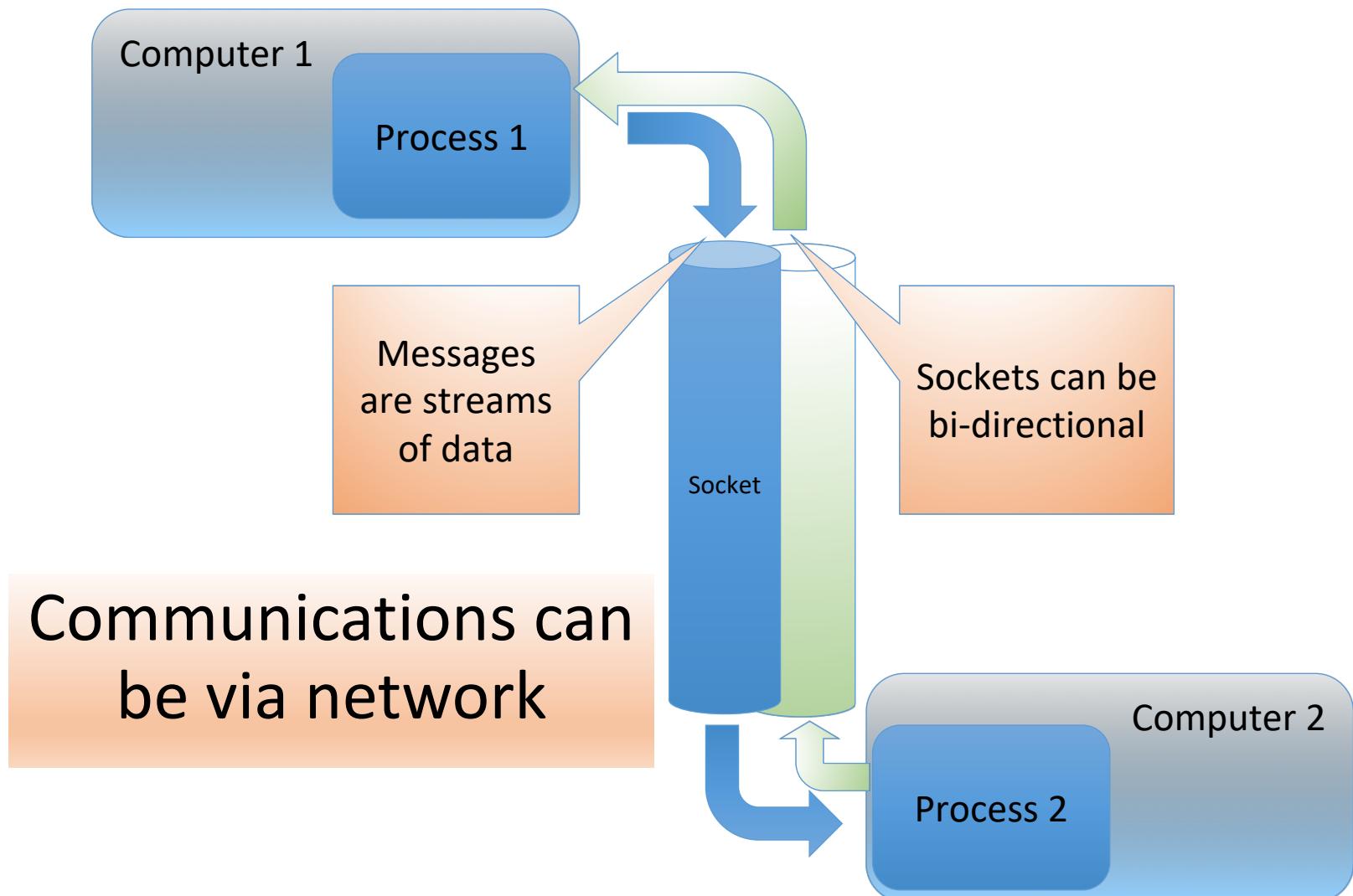


- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- Used to pass small messages
- Advantage: OS handles synchronization
- Disadvantage: Slow
 - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types:
 - Pipes
 - UNIX-domain sockets
 - Internet domain sockets
 - message queues
 - remote procedure calls (RPC)

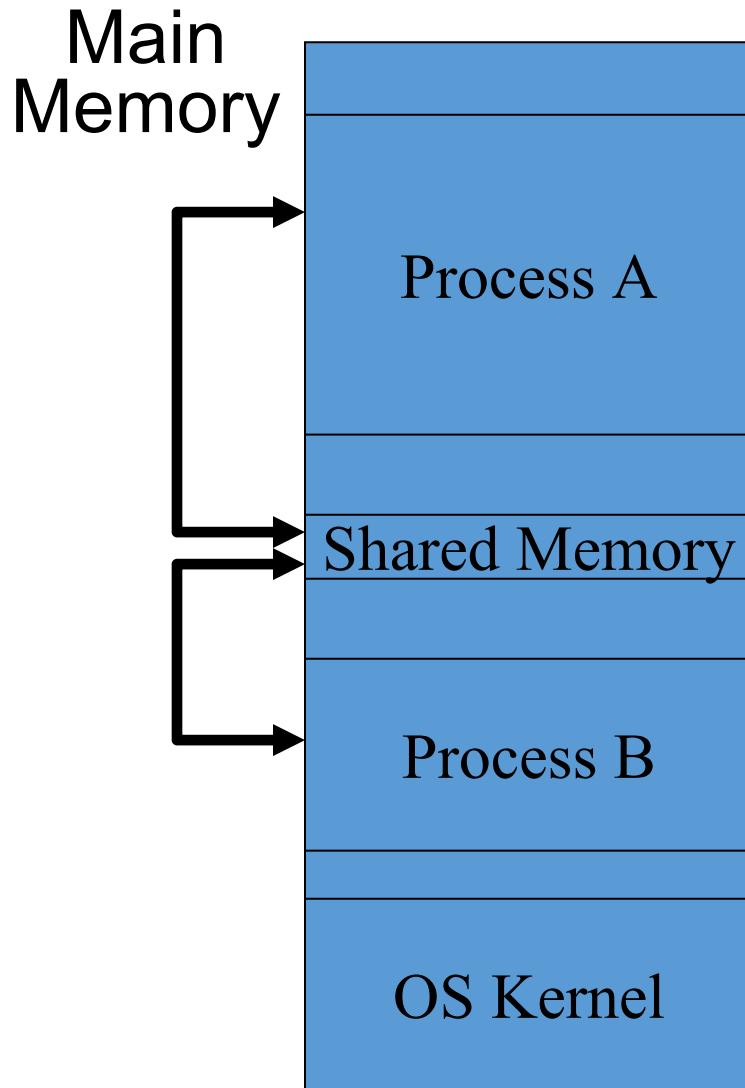
Message Passing via Pipes



Message Passing via Sockets



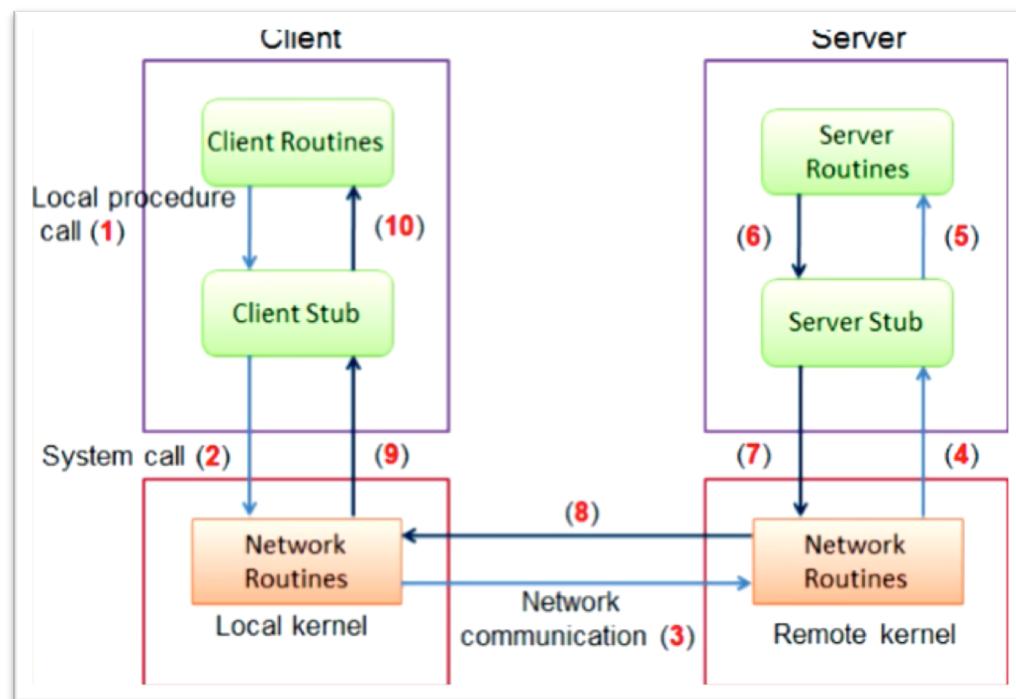
Shared Memory



- OS provides mechanisms for creation of a shared memory buffer between processes (both processes have address mapped to their process)
- Applies to processes on the same machine
- Problem: shared access introduces complexity
 - need to synchronize access

Remote Procedure Calls (RPC)

- Client makes a call to a function and passes parameters
- The client has linked a stub for the function. This stub will marshal (packetize) the data and send it to a remote server
- The network transfers the information to a server
- A service listening to the network receives a request
- The information is unmarshalled (unpacked) and the server's function is called.
- The results are returned to be marshalled into a packet
- The network transfers the packet is sent back to the client
- TCP/IP used to transmit packet

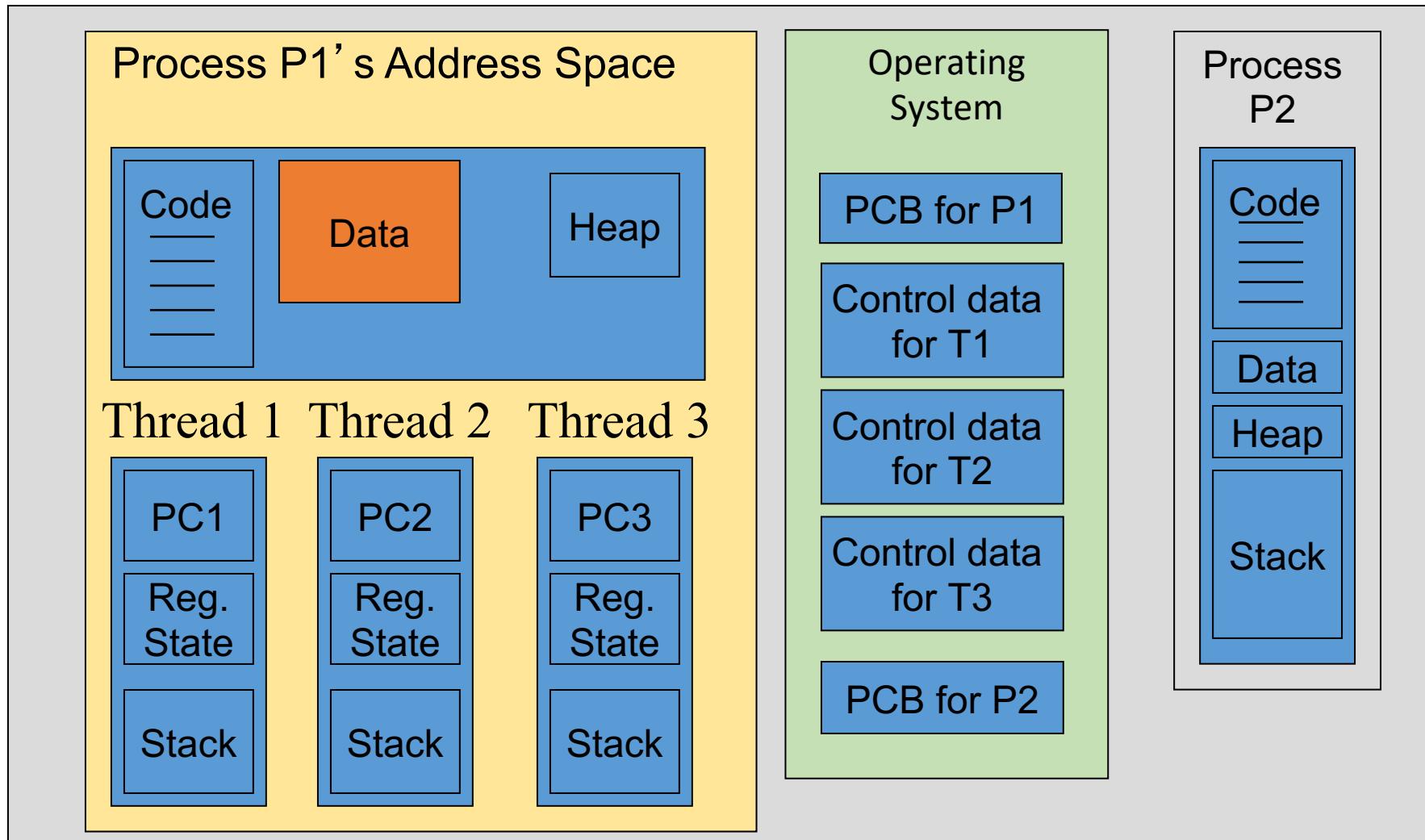




Threads

Threads

Main Memory



Threads

- Process vs Thread
- Kernel vs User-space threads
 - Many-to-One model
 - One-to-One model
 - Many-to-Many model

Benefits vs Thread Safety

- Benefits
 - Responsiveness
 - Resource sharing
 - Low context-switching overhead
 - Scalability
- Thread safety
 - Thread safe
 - If the code behaves correctly during simultaneous or concurrent execution by multiple threads
 - Reentrant
 - If the code behaves correctly when a single thread is interrupted in the middle of executing the code reenters the same code

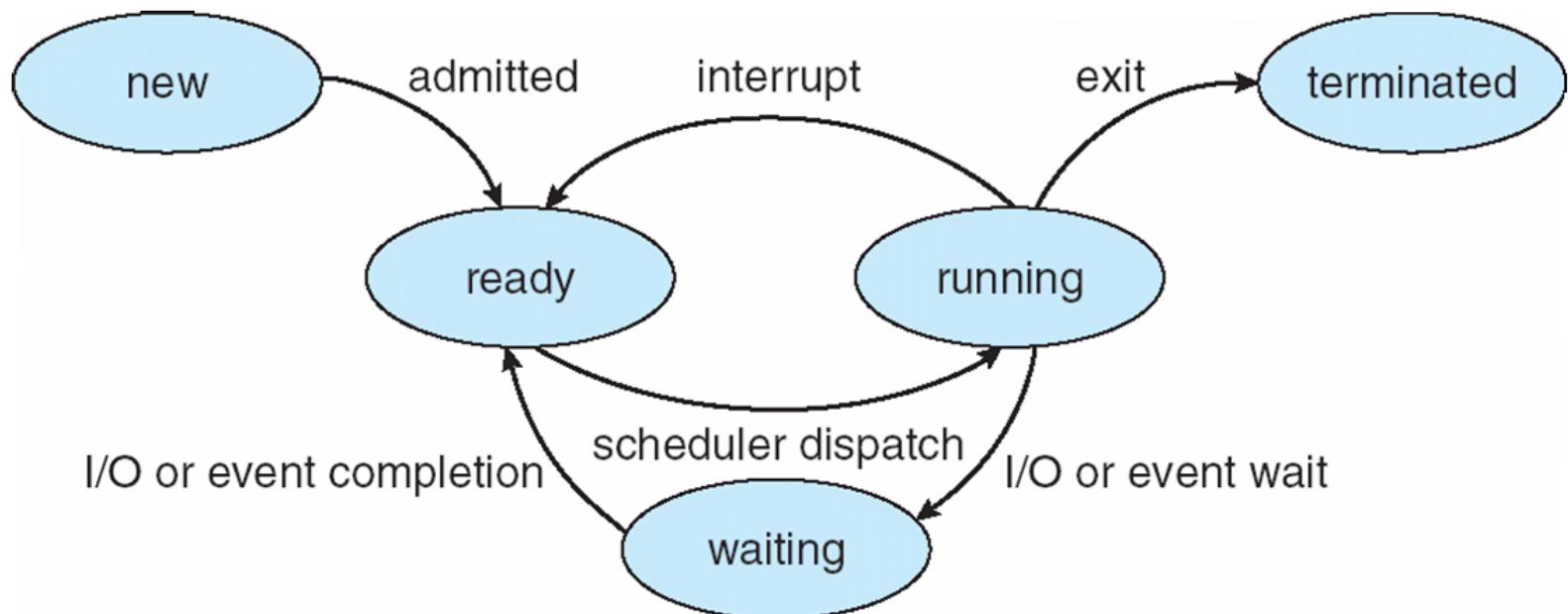
Synchronization

- Problems without synchronization
 - Race condition
 - Critical section
 - Deadlock
 - Starvation
- Synchronization mechanisms
 - Mutual exclusion
 - Atomic Test & Set
 - Semaphore
 - Monitor
 - Conditional variables

Synchronization

- Producer – Consumer (Bounded Buffer) problem
 - Problem description?
 - Solution?
- Reader – Writer problem
 - Problem description?
 - Solution?
- Dining Philosophers problem
 - Problem description?
 - Solution?

Diagram of Process State



Also called “blocked” state



Deadlock

Conditions

The following 4 conditions must hold simultaneously for deadlock to arise:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock

- Detection
- Prevention
- Avoidance

Question 1

Not yet answered

Points out of 1.00

Below are two processes using the semaphores initialized as Q=1 and S=1. Which of the following is true regarding the execution of these processes?

P_0

```
wait(S);  
wait(Q);
```

P_1

```
wait(Q);  
wait(S);
```

·
·
·

·
·
·

```
signal(S);  
signal(Q);  
signal(Q);  
signal(S);
```

Select one or more:

- a. P_1 is subject to starvation
- b. P_0 and P_1 will always complete their execution
- c. P_1 will always run before P_0
- d. P_0 will always run before P_1
- e. P_0 and P_1 can result in deadlock
- f. P_0 is subject to starvation

Question 1

Not yet answered

Points out of 5.00

```
int temp;

void swap(int *y, int *z)
{
    int local;
    local = temp;
    temp = *y;
    *y = *z;
    *z = temp;
    temp = local;
}
```

Select the best answer regarding the given code snippet.

Select one:

- a. The code is re-entrant and thread safe
- b. The code is thread safe but not re-entrant
- c. The code is neither thread safe nor re-entrant
- d. The code is re-entrant but not thread safe

Question 1

Not yet answered

Points out of 5.00

```
int temp;

void swap(int *y, int *z)
{
    int local;
    local = temp;
    temp = *y;
    *y = *z;
    *z = temp;
    temp = local;
}
```

Select the best answer regarding the given code snippet.

Select one:

- a. The code is re-entrant and thread safe
- b. The code is thread safe but not re-entrant
- c. The code is neither thread safe nor re-entrant
- d. The code is re-entrant but not thread safe

Sample Question

Multiple Choice Questions: Choose one option that answers the question best.

1. Advantages of threads over processes include
 - A. lower context switch time
 - B. no possibility of race conditions
 - C. sharing of heap and stack
 - D. smaller code size
 - E. All of the above

Sample Question

Multiple Choice Questions: Choose one option that answers the question best.

2. Which of the following is FALSE about IPC via pipes?
 - A. Basic primitives are `send()` and `receive()`.
 - B. Communication can be blocking or non-blocking.
 - C. Pipes can be anonymous or named.
 - D. Pipes can be used for only one-way communication.
 - E. IPC via pipes is slower than IPC using shared memory.

Sample Question

Multiple Choice Questions: Choose one option that answers the question best.

3. Which of the following is NOT required for a system to be in deadlock?
 - A. mutual exclusion
 - B. no preemption
 - C. acquire and hold
 - D. circular dependency

Sample Question

Short Answer Questions: Consider the following program code for the next question (Question 1).

```
int ret = fork();
if (ret == 0){
    ret = fork();
    printf("Hello \n");
}
printf("World \n");
return 0;
```

1. How many times the following word will be printed?

A. “Hello”: _____ times

B. “World”: _____ times

Sample Question

Short Answer Questions: Consider the following program code for the next question (Question 1).

```
int ret = fork();
if (ret == 0){
    ret = fork();
    printf("Hello \n");
}
printf("World \n");
return 0;
```

1. How many times the following word will be printed?

A. “Hello”: 2 times

B. “World”: 3 times

Sample Question

Short Answer Questions: Consider the following program code for the next question (Question 2).

```
int t;

void swap(int *x, int *y)
{
    int s;
    s = t;
    t = *x;
    *x = *y;
    *y = t;
    t = s;
}
```

2. Is there a possibility of a race condition updating variable t in the code above?

Sample Question

Short Answer Questions: Consider the following program code for the next question (Question 2).

```
int t;

void swap(int *x, int *y)
{
    int s;
    s = t;
    t = *x;
    *x = *y;
    *y = t;
    t = s;
}
```

2. Is there a possibility of a race condition updating variable t in the code above? **YES**

Sample Question

Short Answer Questions:

3. Mark each term with the letter of the correct statement.

- | | |
|---|--|
| A. A piece of code functions correctly during simultaneous or concurrent execution by multiple threads. | C. A function defined in the Linux kernel to copy data from kernel-space. |
| B. A function defined in the Linux kernel to copy data to kernel-space. | D. The procedure for replacing the currently executing process with another. |

Thread-safe

copy_to_user()

Sample Question

Short Answer Questions:

3. Mark each term with the letter of the correct statement.

- | | |
|---|--|
| A. A piece of code functions correctly during simultaneous or concurrent execution by multiple threads. | C. A function defined in the Linux kernel to copy data from kernel-space. |
| B. A function defined in the Linux kernel to copy data to kernel-space. | D. The procedure for replacing the currently executing process with another. |

Thread-safe

A

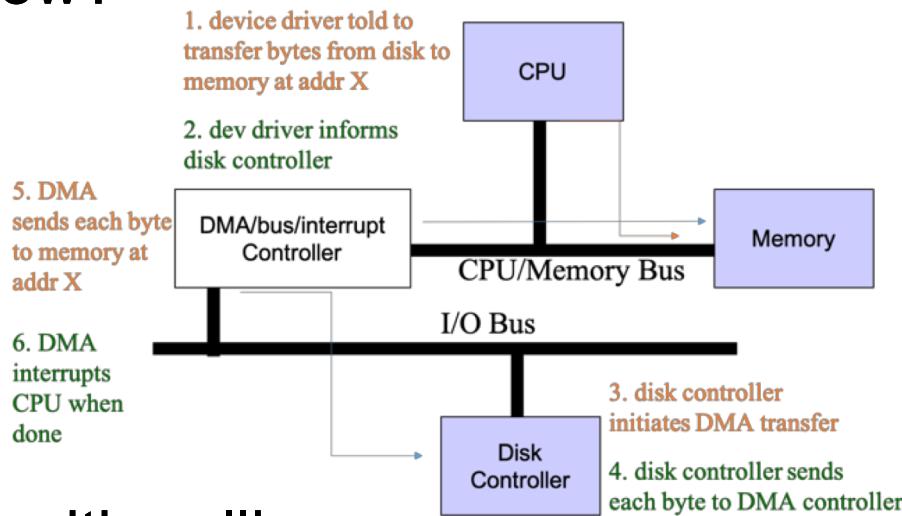
copy_to_user()

C

Sample Question

Short Answer Questions:

4. What is the I/O strategy of the device manager illustrated in the image below?

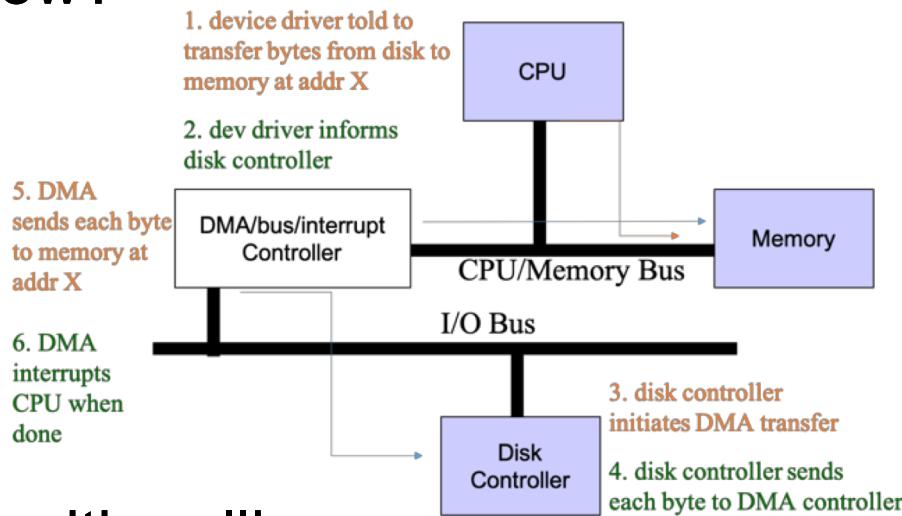


- A. direct I/O with polling
- B. direct I/O with interrupts
- C. DMA with interrupts
- D. hardware interrupts

Sample Question

Short Answer Questions:

4. What is the I/O strategy of the device manager illustrated in the image below?



- A. direct I/O with polling
- B. direct I/O with interrupts
- C. DMA with interrupts
- D. hardware interrupts

Sample Question

1. Consider three processes, P, Q and R with the following code:

- P: ps1; ps2; ps3; ps4;
- Q: qs1; qs2; qs3; qs4;
- R: rs1; rs2; rs3; rs4;

These processes have the following synchronization constraint:

- a) rs1 must be the first statement to execute
- b) process R should be the last process to exit
- c) ps3 must execute after qs2
- d) qs3 must execute after ps4 and rs3
- e) rs4 must execute after either ps4 or qs4 (or both) have executed

Using Condition Variables, provide updated codes for P, Q and R that satisfy these constraints.

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<ps1>	<qs1>	<rs1>
<ps2>	<qs2>	<rs2>
<ps3>	<qs3>	<rs3>
<ps4>	<qs4>	<rs4>

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<u>wait(&s1)</u> <ps1>	<u>wait(&s1)</u> <qs1>	<rs1>
<ps2>	<qs2>	<u>signal(&s1)</u> <u>signal(&s1)</u> <rs2>
<ps3>	<qs3>	<rs3>
<ps4>	<qs4>	<rs4>

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<u>wait(&s1)</u> <ps1>	<u>wait(&s1)</u> <qs1>	<rs1>
<ps2> <u>wait(&s2)</u>	<qs2> <u>signal(&s2)</u>	<u>signal(&s1)</u> <u>signal(&s1)</u> <rs2>
<ps3>	<qs3>	<rs3>
<ps4>	<qs4>	<rs4>

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<u>wait(&s1)</u> <ps1>	<u>wait(&s1)</u> <qs1>	<rs1>
<ps2> <u>wait(&s2)</u>	<qs2> <u>signal(&s2)</u> <u>wait(&s3)</u>	<rs2>
<ps3>	<qs3>	<rs3>
<ps4> <u>signal(&s3)</u>	<qs4>	<rs4>

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<u>wait(&s1)</u> <ps1>	<u>wait(&s1)</u> <qs1>	<rs1>
<ps2> <u>wait(&s2)</u>	<qs2> <u>signal(&s2)</u> <u>wait(&s3)</u>	<rs2>
<ps3>	<qs3>	<rs3>
<ps4> <u>signal(&s3)</u> <u>signal(&s4)</u>	<qs4> <u>signal(&s4)</u>	<rs4>

Sample Question

<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
<u>wait(&s1)</u> <ps1>	<u>wait(&s1)</u> <qs1>	<rs1>
<ps2> <u>wait(&s2)</u>	<qs2> <u>signal(&s2)</u> <u>wait(&s3)</u>	<rs2>
<ps3>	<qs3>	<rs3>
<ps4> <u>signal(&s3)</u> <u>signal(&s4)</u> <u>signal(&s5)</u>	<qs4> <u>signal(&s4)</u> <u>signal(&s5)</u>	<rs4> <u>wait(&s5)</u>