

# Introduction

## Chapter #1

Course Instructor: Nausheen Shoaib

# Marks Distribution

- ▶ Mid1: 15%
- ▶ Mid2: 15%
- ▶ Class activities+ Assignment+Projects: 20%
- ▶ Final: 50%
- ▶ Book: *Operating System Concepts by Abraham Silberschatz 10<sup>th</sup> Edition*

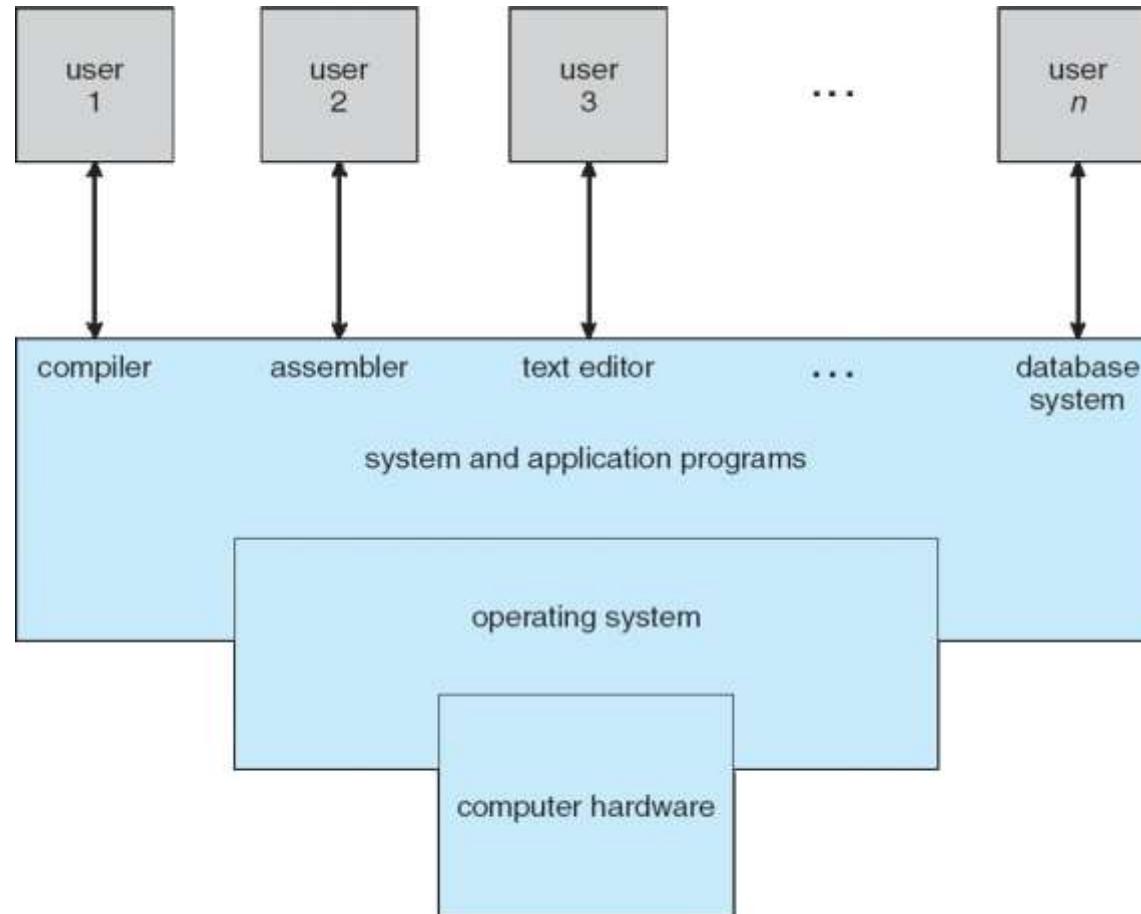
# What is an Operating System?

- ▶ A program that acts as an intermediary between a user of a computer and the computer hardware

# Computer System Structure

- ▶ Computer system can be divided into four components:
  - Hardware – provides basic computing resources
  - Operating system
  - Application programs
  - Users

# Four Components of a Computer System



# Operating System Definition

- ▶ OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- ▶ OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

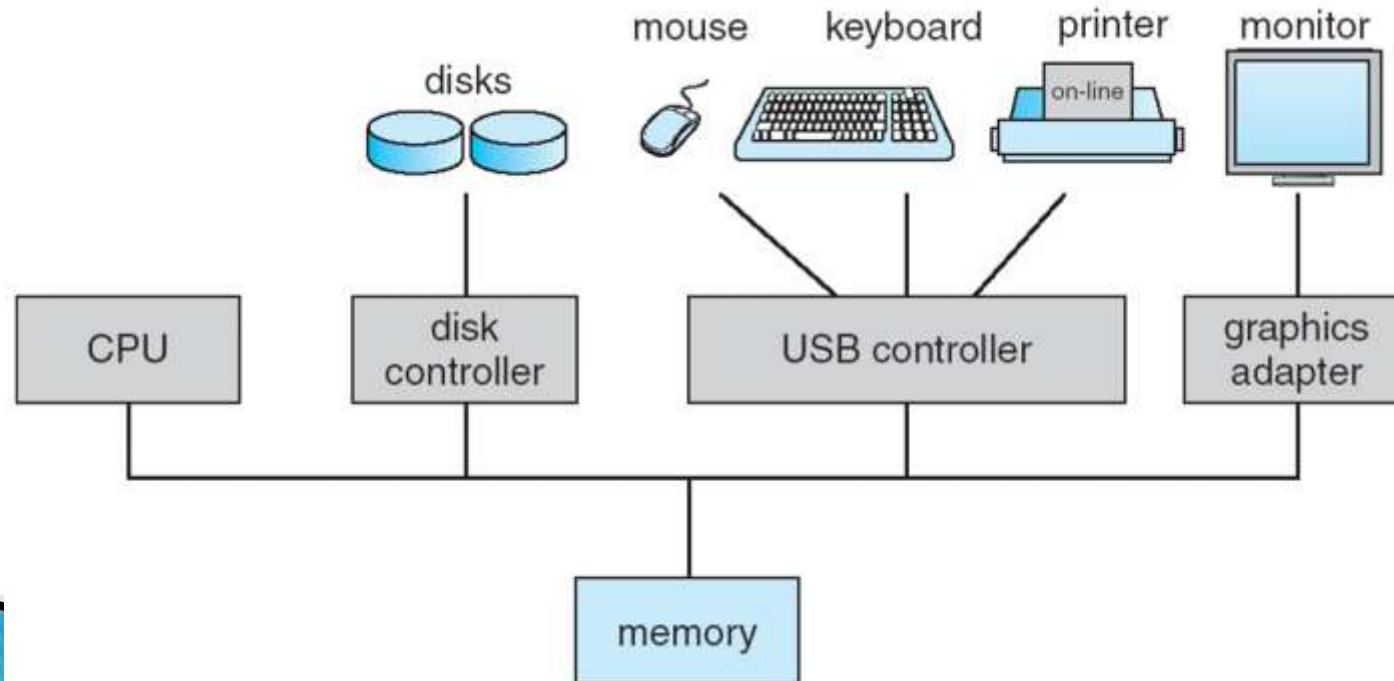
“The one program running at all times on the computer” is the **kernel**.

# Computer Startup

- ▶ **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

# Computer System Organization

- ▶ Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



# Computer-System Operation

- ▶ I/O devices and the CPU can execute concurrently
- ▶ Each device controller is in charge of a particular device type
- ▶ Each device controller has a local buffer
- ▶ CPU moves data from/to main memory to/from local buffers
- ▶ I/O is from the device to local buffer of controller
- ▶ Device controller informs CPU that it has finished its operation by causing an **interrupt**

# Common Functions of Interrupts

- n Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- n Interrupt architecture must save the address of the interrupted instruction
- n A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- n An operating system is **interrupt driven**

# Interrupt Handling

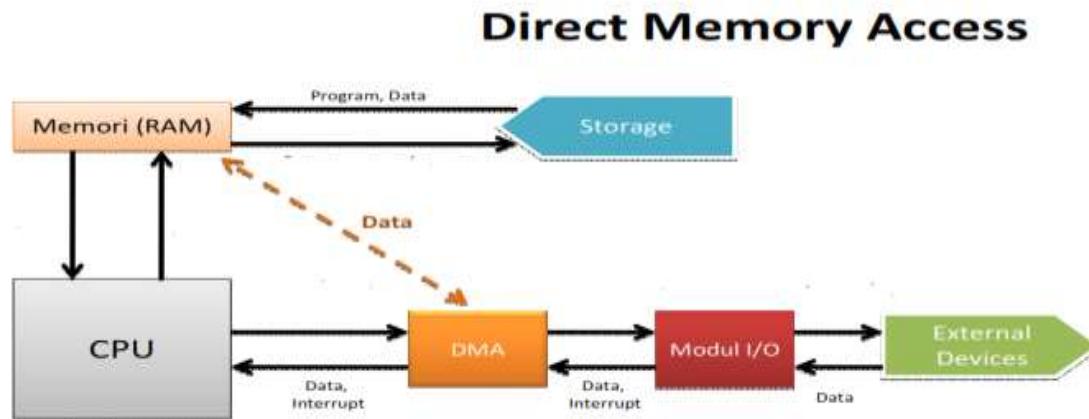
- ▶ The operating system preserves the state of the CPU by storing registers and the program counter
- ▶ Determines which type of interrupt has occurred:
  - **polling**
  - **vectored** interrupt system
- ▶ Separate segments of code determine what action should be taken for each type of interrupt

# I/O Structure

- ▶ After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- ▶ After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the OS to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  -

# Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds



- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

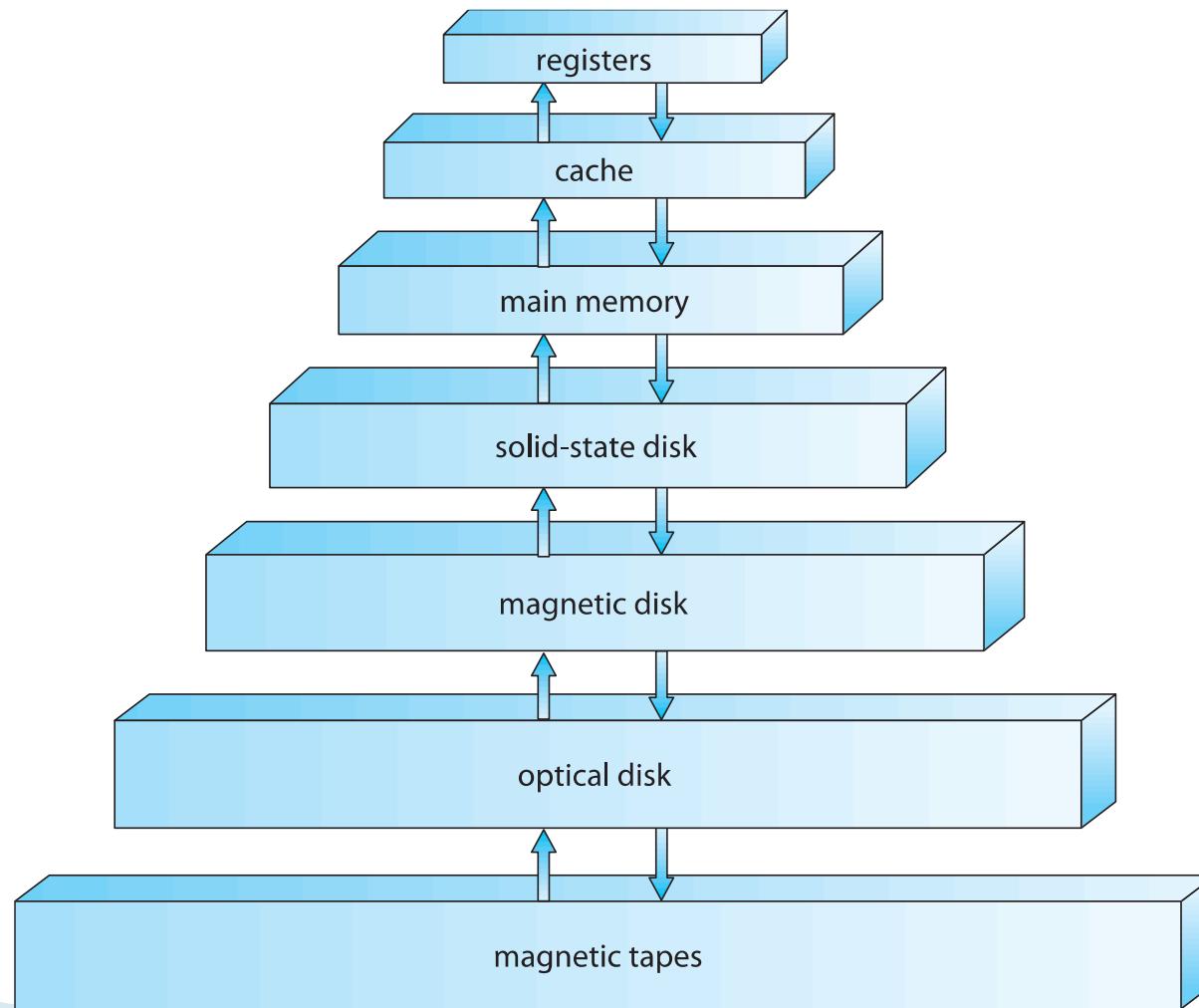
# Storage Structure

- ▶ Main memory – only large storage media that the CPU can access directly
- ▶ Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- ▶ Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer

# Storage Hierarchy

- ▶ Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- ▶ **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- ▶ **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

# Storage-Device Hierarchy



# Caching

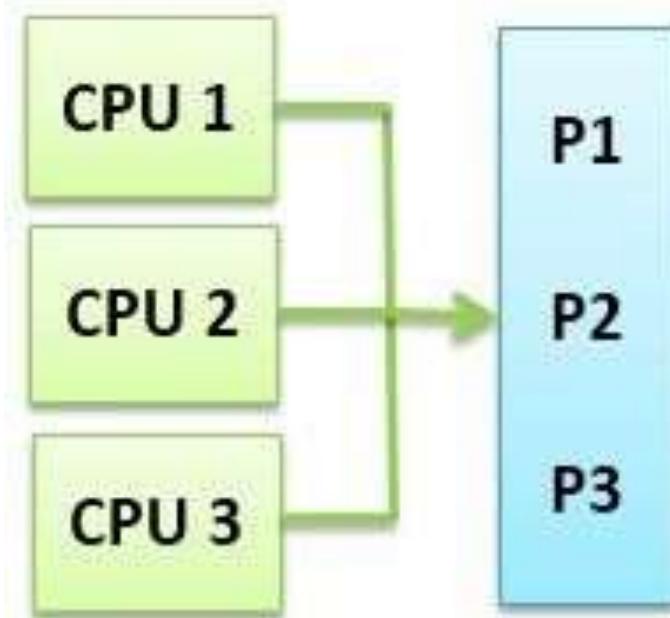
- ▶ Important principle, performed at many levels in a computer (in hardware, operating system, software)
- ▶ Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there

# Computer-System Architecture

- ▶ **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. Increased throughput
    2. Economy of scale
    3. Increased reliability – graceful degradation or **fault tolerance**
  - Two types:
    1. Asymmetric Multiprocessing
    2. Symmetric Multiprocessing

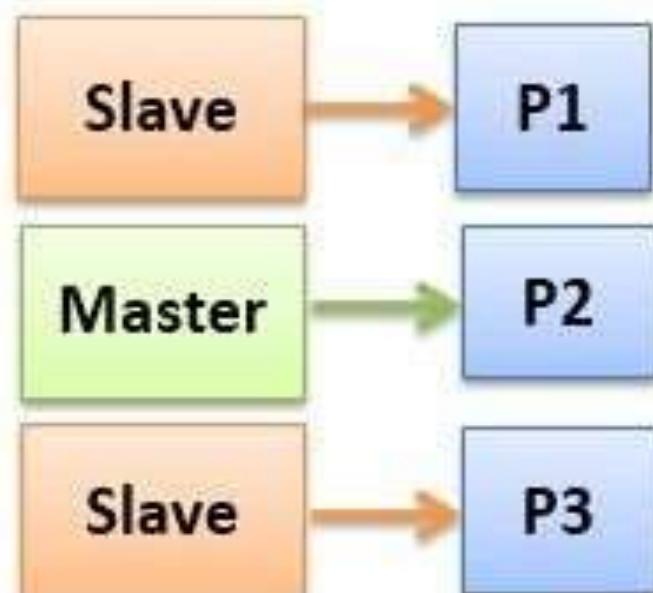
# Symmetric vs. Asymmetric Multiprocessing Architecture [1 / 2]

## Symmetric Multiprocessing



(Shared Memory)

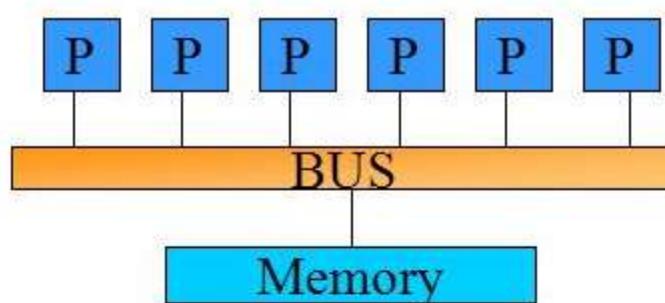
## Asymmetric Multiprocessing



(No Shared Memory)

# A Dual-Core Design

- ▶ **UMA** and **NUMA** architecture variations
- ▶ Multi-chip and **multicore**

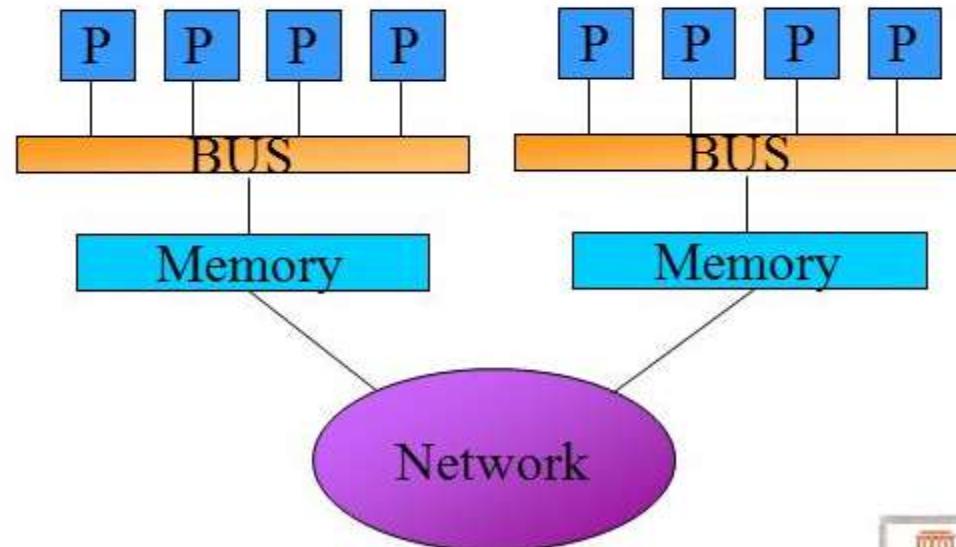


Uniform memory access (UMA):

Each processor has uniform access to memory. Also known as **symmetric multiprocessors**, or SMPs (Sun E10000)

---

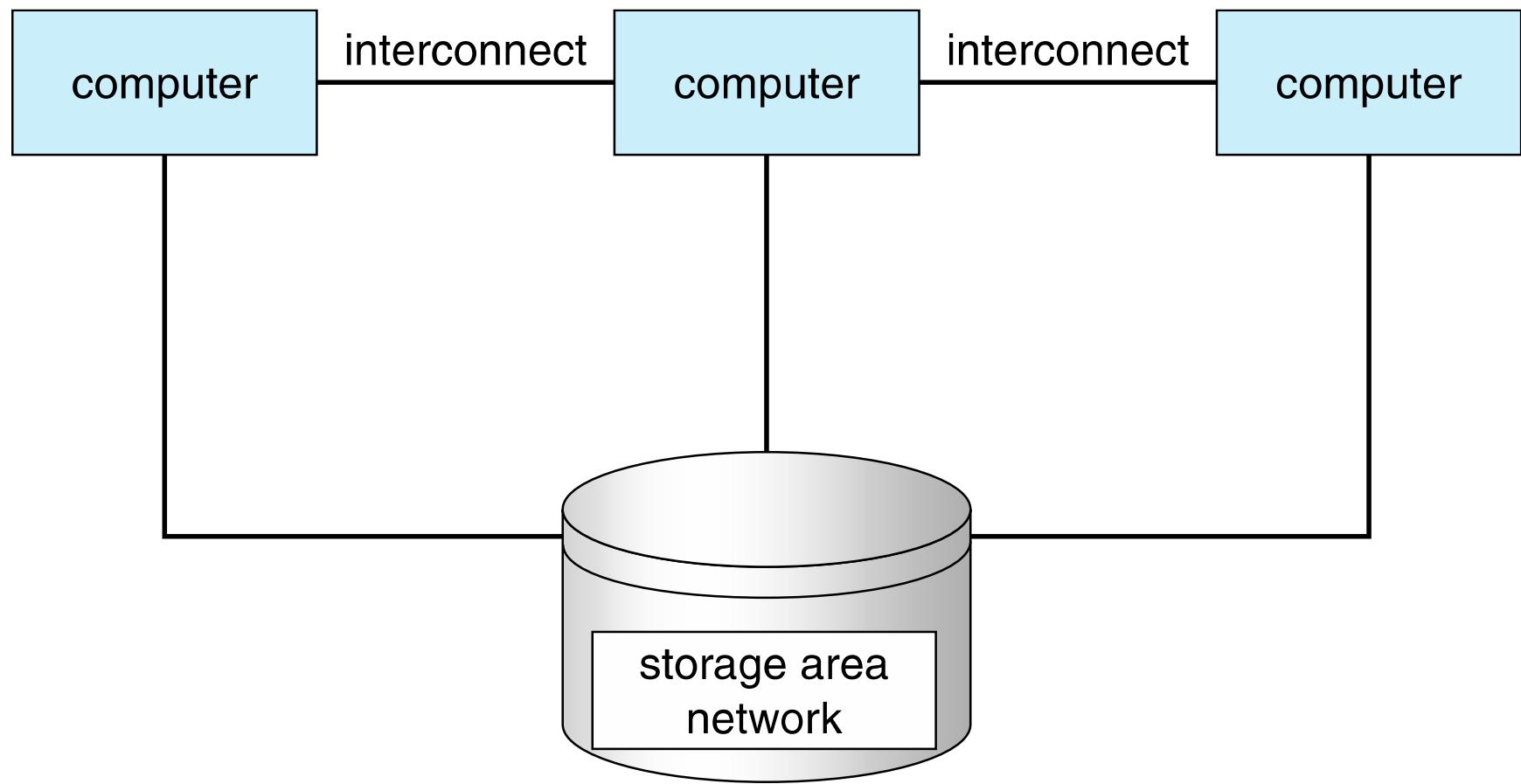
Non-uniform memory access (NUMA): Time for memory access depends on location of data. Local access is faster than non-local access. Easier to scale than SMPs (SGI Origin)



# Clustered Systems

- ▶ Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network (SAN)**
  - Provides a **high-availability** service which survives failures
    - **Asymmetric clustering** has one machine in hot-standby mode
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other

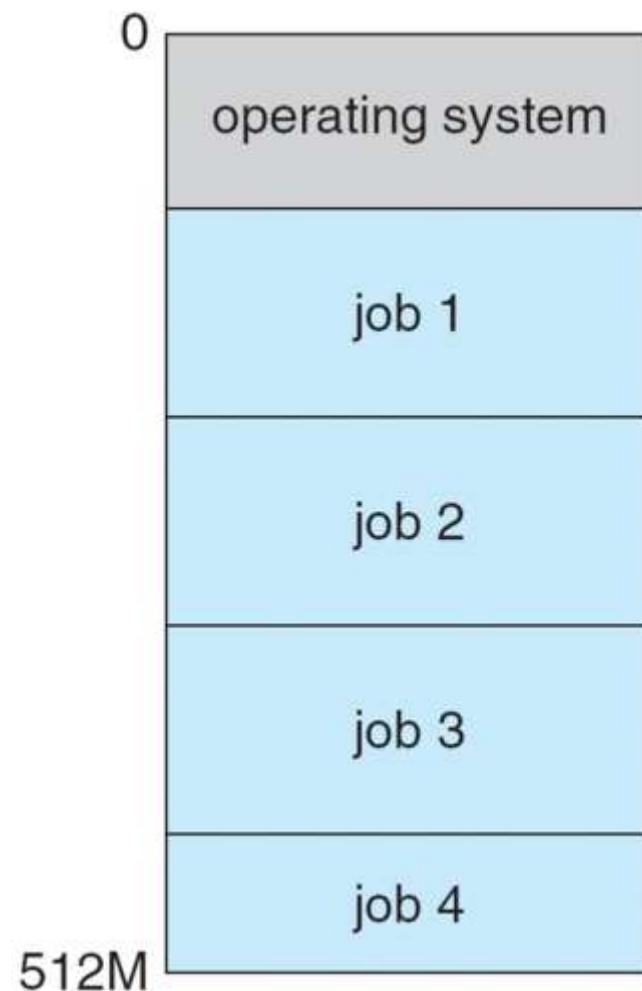
# Clustered Systems



# Operating System Structure

- ▶ **Multiprogramming** needed for efficiency
  - ❖ Single user cannot keep CPU and I/O devices busy at all times
  - ❖ Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - ❖ A subset of total jobs in system is kept in memory
  - ❖ One job selected and run via **job scheduling**
  - ❖ When it has to wait (for I/O for example), OS switches to another job
- ▶ **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - ❖ **Response time** should be < 1 second
  - ❖ Each user has at least one program executing in memory ⇒ **process**
  - ❖ If several jobs ready to run at the same time ⇒ **CPU scheduling**
  - ❖ If processes don't fit in memory, **swapping** moves them in and out to run
  - ❖ **Virtual memory** allows execution of processes not completely in memory

# Memory Layout for Multiprogrammed System

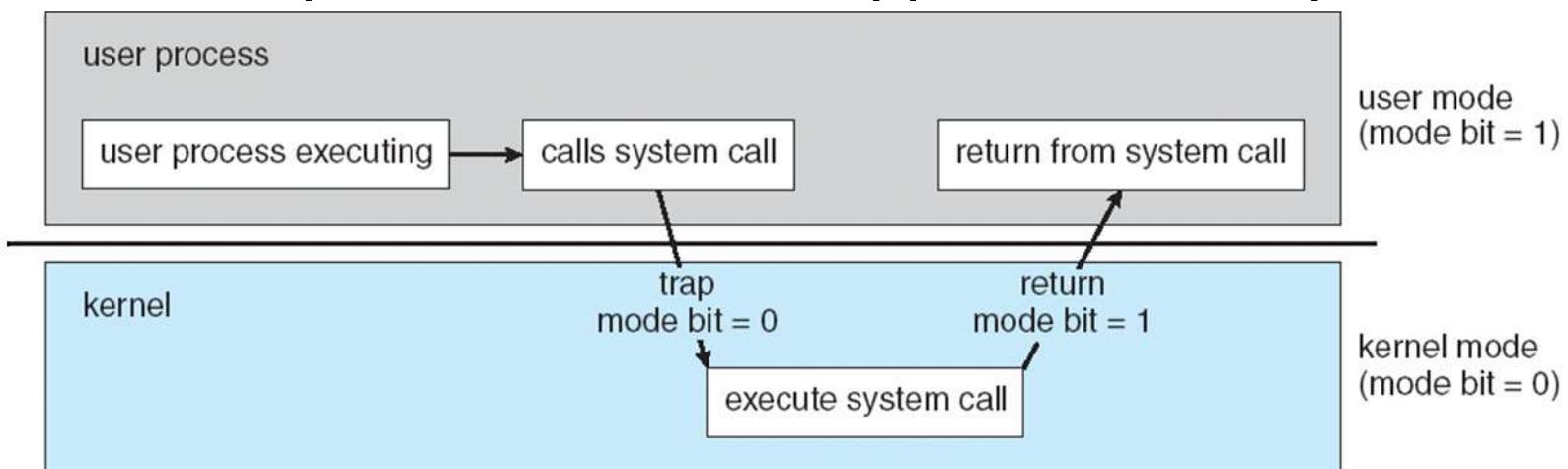


# Operating-System Operations

- ▶ Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- ▶ Increasingly CPUs support multi-mode operations
  - i.e. virtual machine manager (VMM) mode for guest VMs

# Transition from User to Kernel Mode

- ▶ Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain



# Process Management

- ▶ A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- ▶ Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- ▶ **Program counter (PC):** Contains the address of an instruction to be fetched
- ▶ Single-threaded process has one **program counter** specifying location of next instruction to execute
- ▶ Multi-threaded process has one program counter per thread

# Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- ▶ Creating and deleting both user and system processes
- ▶ Suspending and resuming processes
- ▶ process synchronization
- ▶ process communication
- ▶ deadlock handling

# Memory Management

- ▶ Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

# Storage Management

## ▶ File-System management

- Files usually organized into directories
- Access control on most systems to determine who can access what
- OS activities include
  - Creating and deleting files and directories
  - Primitives to manipulate files and dirs
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media

# Mass-Storage Management

- ▶ OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- ▶ Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications
  - Varies between WORM (write-once, read-many-times) and RW (read-write)

# I/O Subsystem

- ▶ I/O subsystem responsible for
  - Memory management of I/O
  - caching (storing parts of data in faster storage for performance)
  - spooling (the overlapping of output of one job with input of other jobs)

# Protection and Security

- ▶ **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- ▶ **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service

# Computing Environments – Distributed

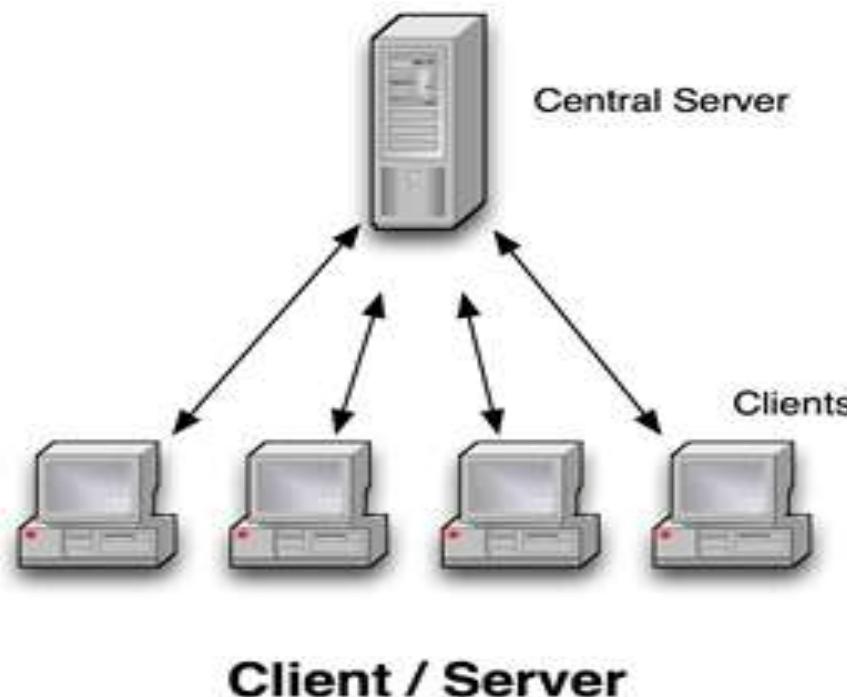
## ▶ Distributed

- Collection of separate, possibly heterogeneous, systems networked together
  - **Network** is a communications path,
    - **Local Area Network (LAN)**
    - **Wide Area Network (WAN)**
  - **Network Operating System** provides features between systems across network

# Computing Environments – Client–Server

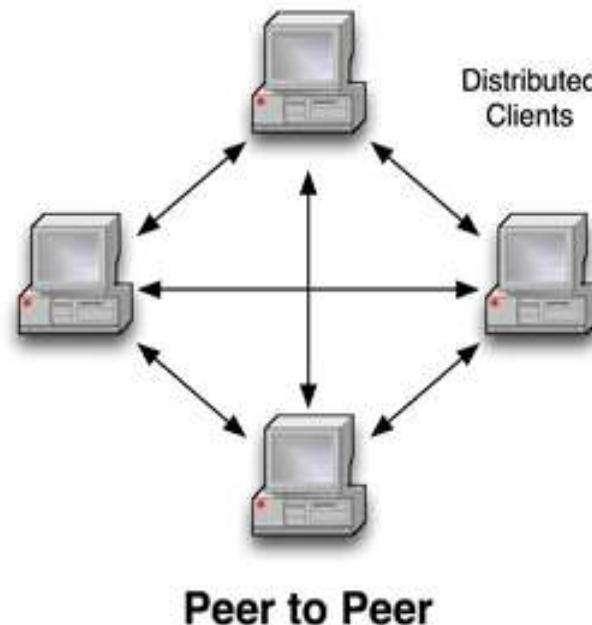
## ■ Client-Server Computing

- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**

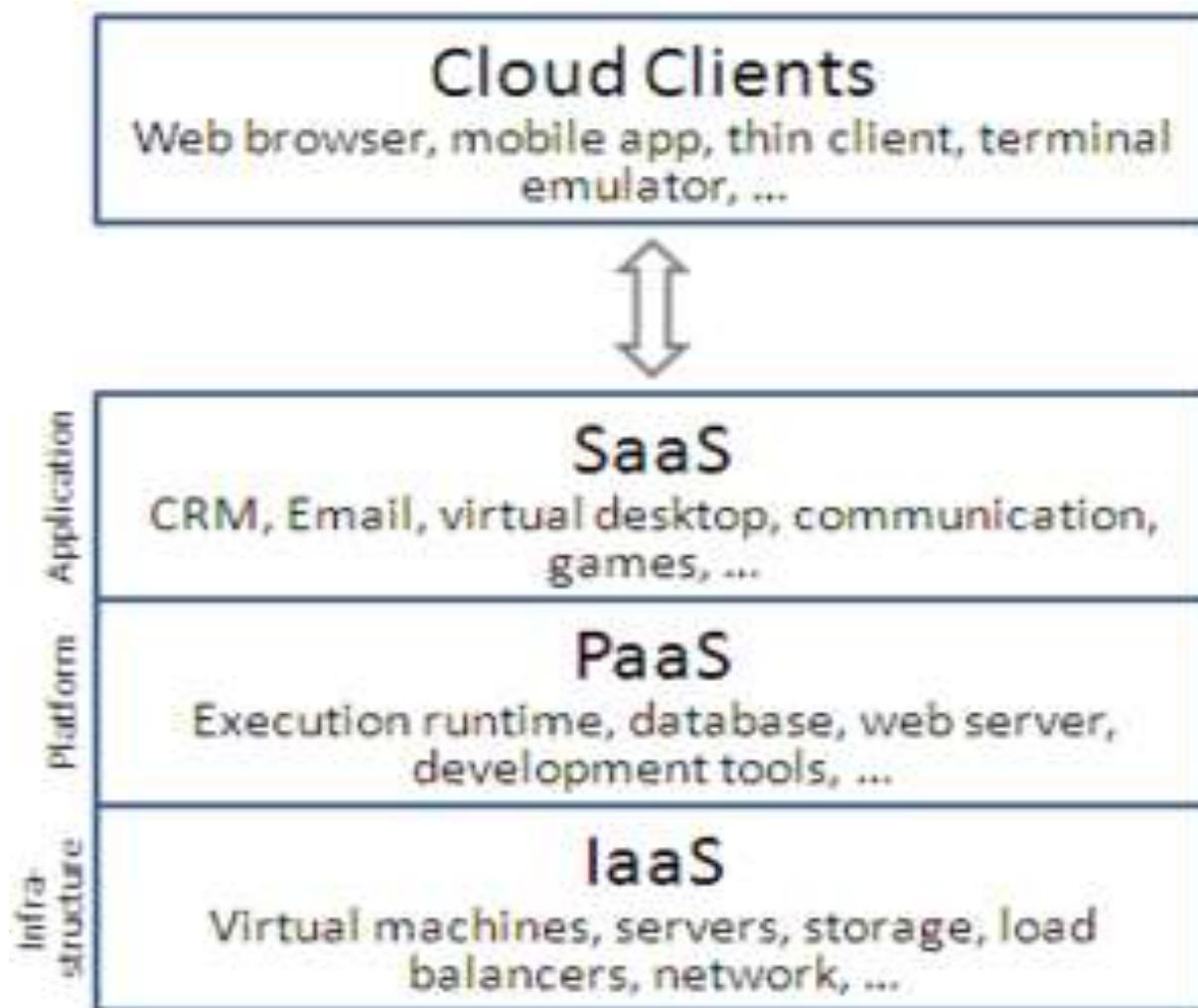


# Computing Environments – Peer-to-Peer

- ▶ Another model of distributed system
- ▶ P2P does not distinguish clients and servers



# Computing Environments - Cloud Computing



# Computing Environments – Real-Time Embedded Systems

- ▶ Real-time embedded systems most prevalent form of computers
  - Vary considerable, special purpose, limited purpose OS, **real-time OS**
  - Use expanding
- ▶ Many other special computing environments as well
  - Some have OSes, some perform tasks without an OS
- ▶ Real-time OS has well-defined fixed time constraints
  - Processing **must** be done within constraint
  - Correct operation only if constraints met

# Operating System Structures

## Chapter #2

Course Instructor: Nausheen Shoaib

# Operating System Services

- ▶ One set of operating-system services provides functions that are helpful to the user:
  - **User interface** – Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Program execution** – The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** – A running program may require I/O, which may involve a file or an I/O device

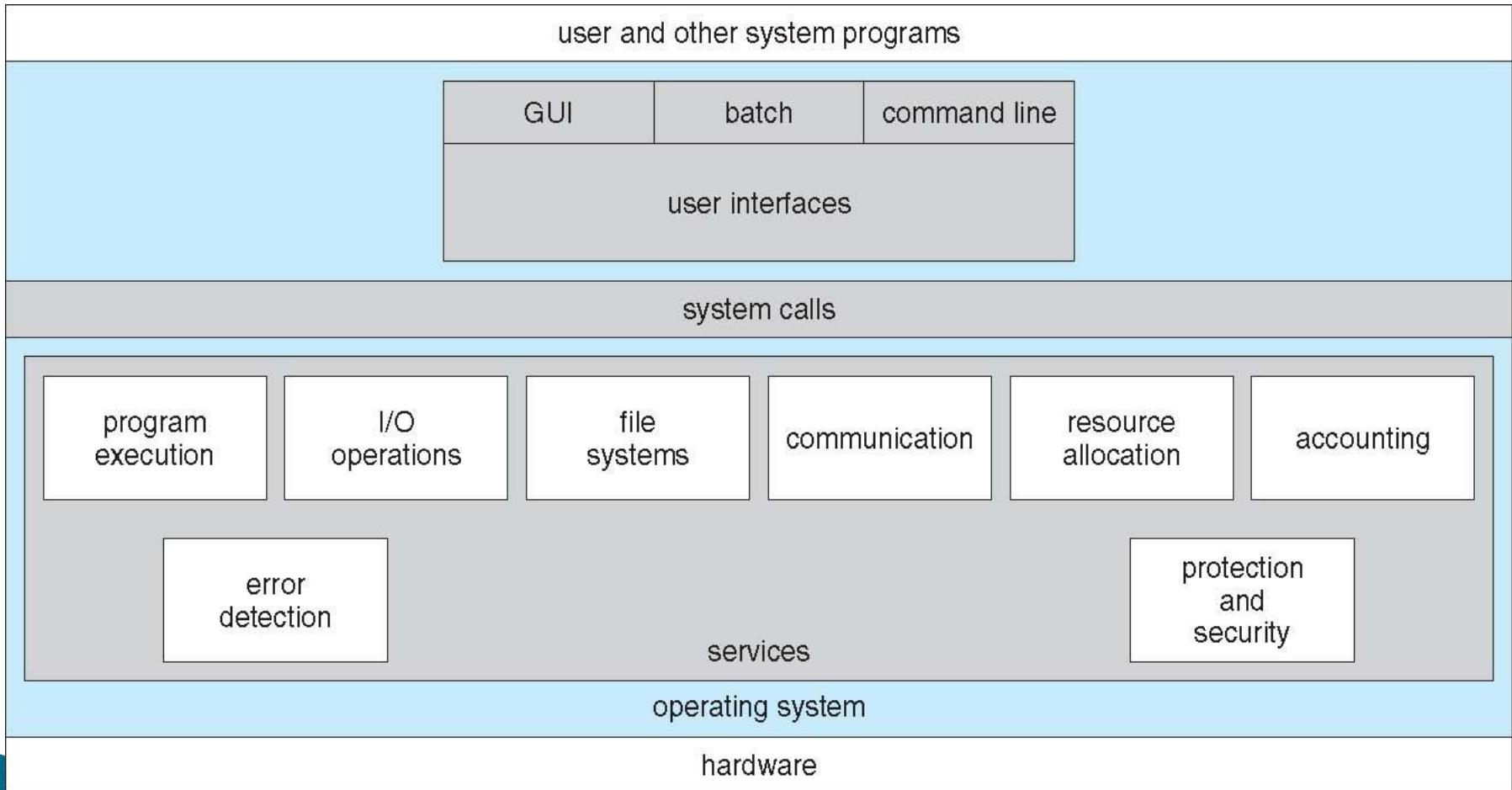
# Operating System Services (Cont.)

- **File-system manipulation** – The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action

# Operating System Services (Cont.)

- **Resource allocation** – When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources – CPU cycles, main memory, file storage, I/O devices.
- **Accounting** – To keep track of which users use how much and what kinds of computer resources
- **Protection and security** – The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

# A View of Operating System Services



# User Operating System Interface – CLI

CLI or **command interpreter** allows direct command entry

- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs

# User Operating System Interface – GUI

- ▶ User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
- ▶ Many systems now include both CLI and GUI interfaces
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

# Touchscreen Interfaces

## ■ Touchscreen devices require new interfaces

- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry
- Voice commands.

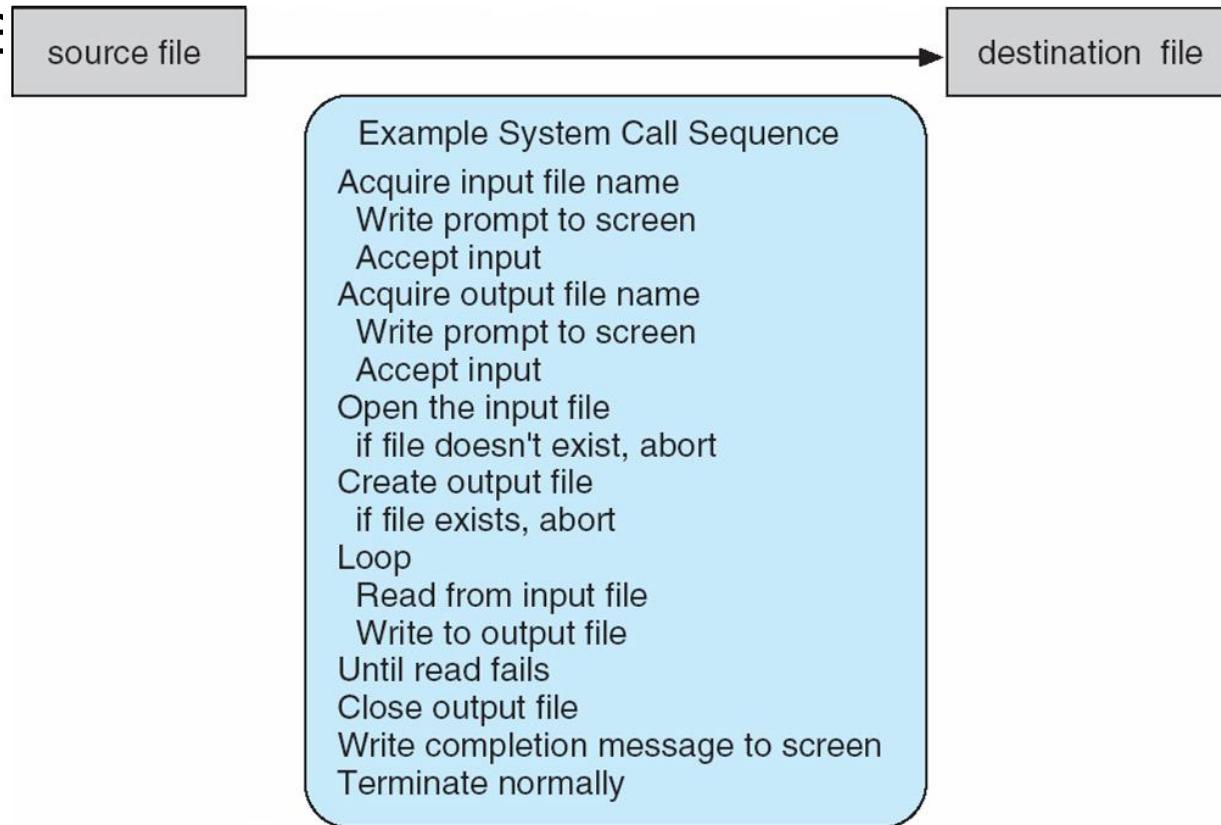


# System Calls

- ▶ Programming interface to the services provided by the OS
- ▶ Typically written in a high-level language (C or C++)
- ▶ Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- ▶ Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

# Example of System Calls

- ▶ System call sequence to copy the contents of one



# Example of Standard API

## EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

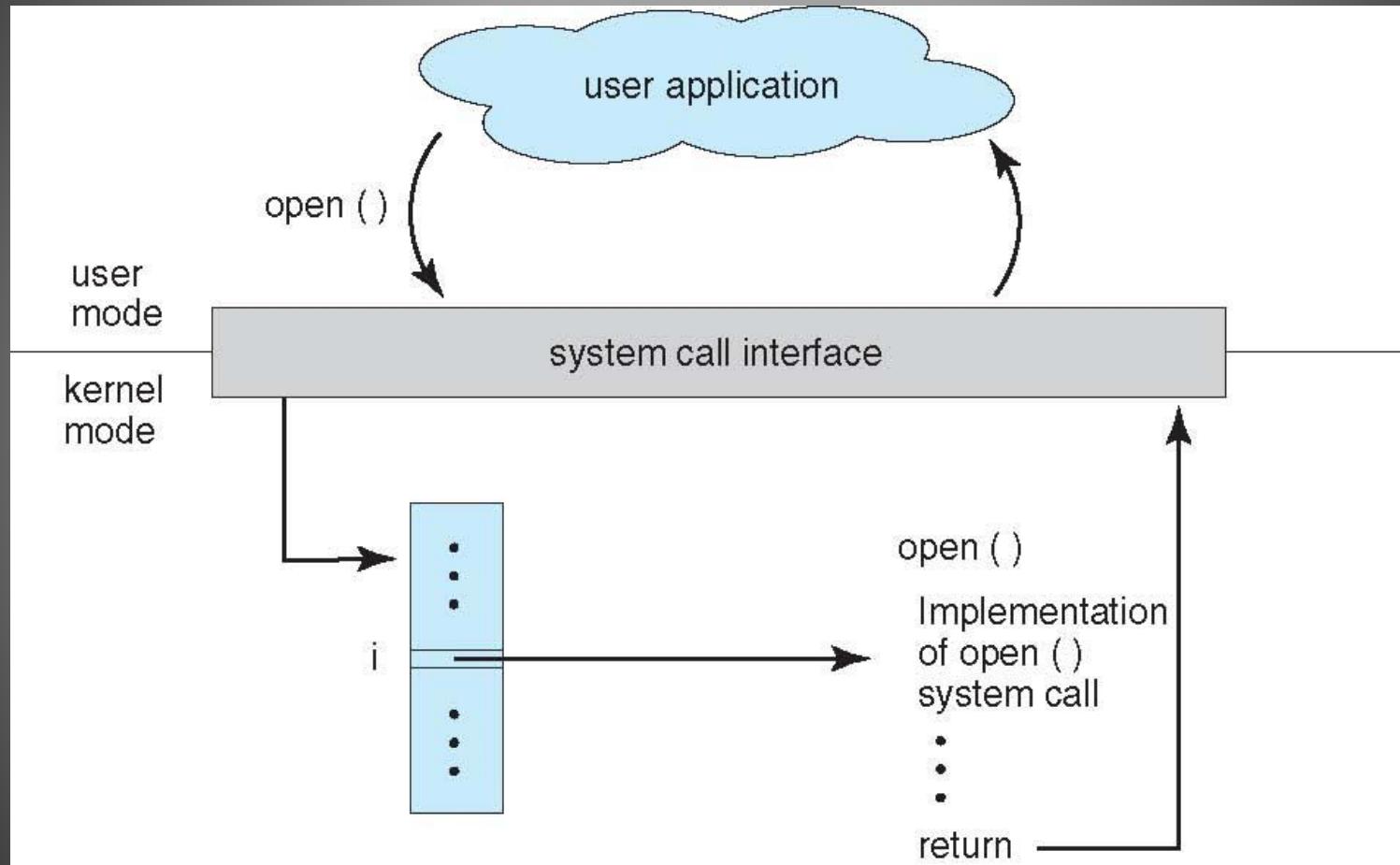
return value      function name      parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

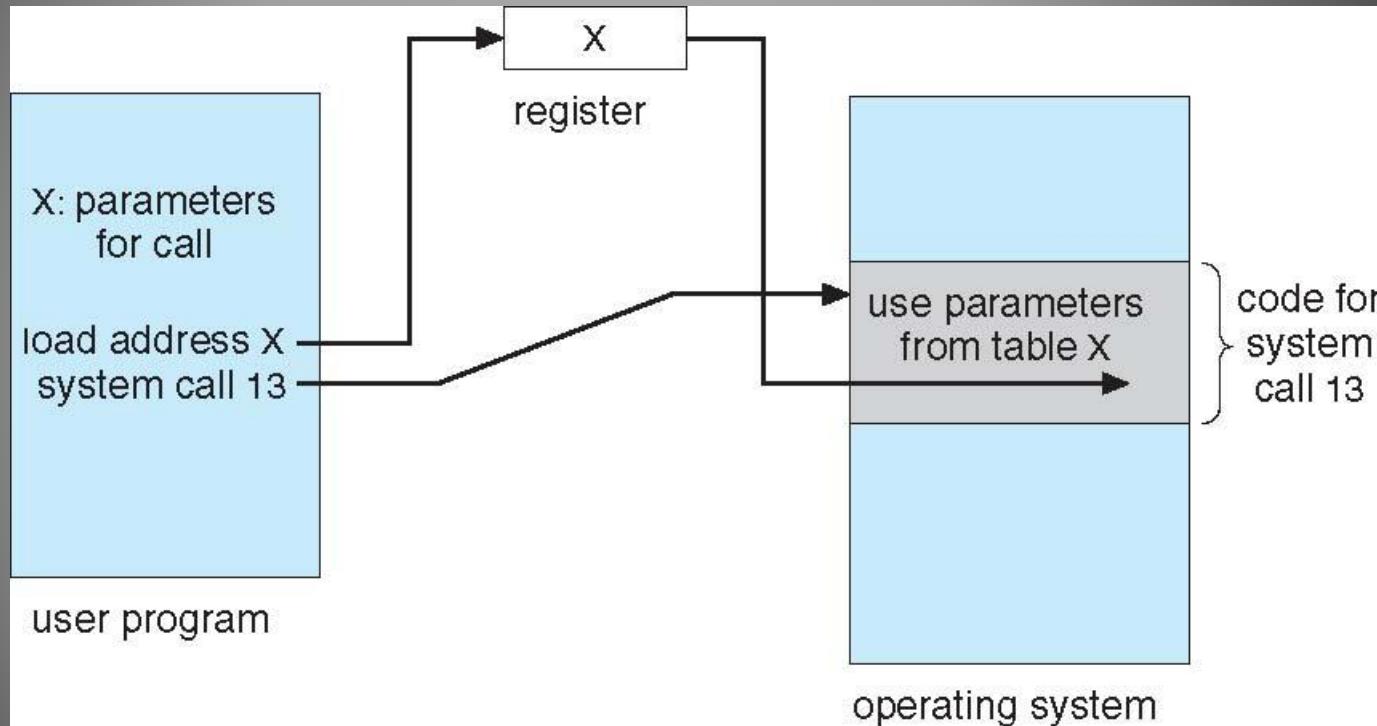
# API - System Call - OS Relationship



# System Call Parameter Passing

- ▶ Three general methods used to pass parameters to the OS
  - ❖ Simplest: pass the parameters in registers
  - ❖ Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - ❖ Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

# System Call Parameter Passing via Table



# Types of System Calls

System calls can be roughly grouped into five major categories:

1. Process Control
2. File management
3. Device Management
4. Information Maintenance
5. Communication

# Types of System Calls

## 1. Process Control

- ▶ load
- ▶ execute
- ▶ create process (for example, fork on Unix-like systems or NtCreateProcess in the Windows NT Native API)
- ▶ terminate process
- ▶ get/set process attributes
- ▶ wait for time, wait event, signal event
- ▶ allocate free memory

# **Types of System Calls**

## **2.File management**

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

## **3.Device Management**

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

# **Types of System Calls**

## **4.Information Maintenance**

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

## **5.Communication**

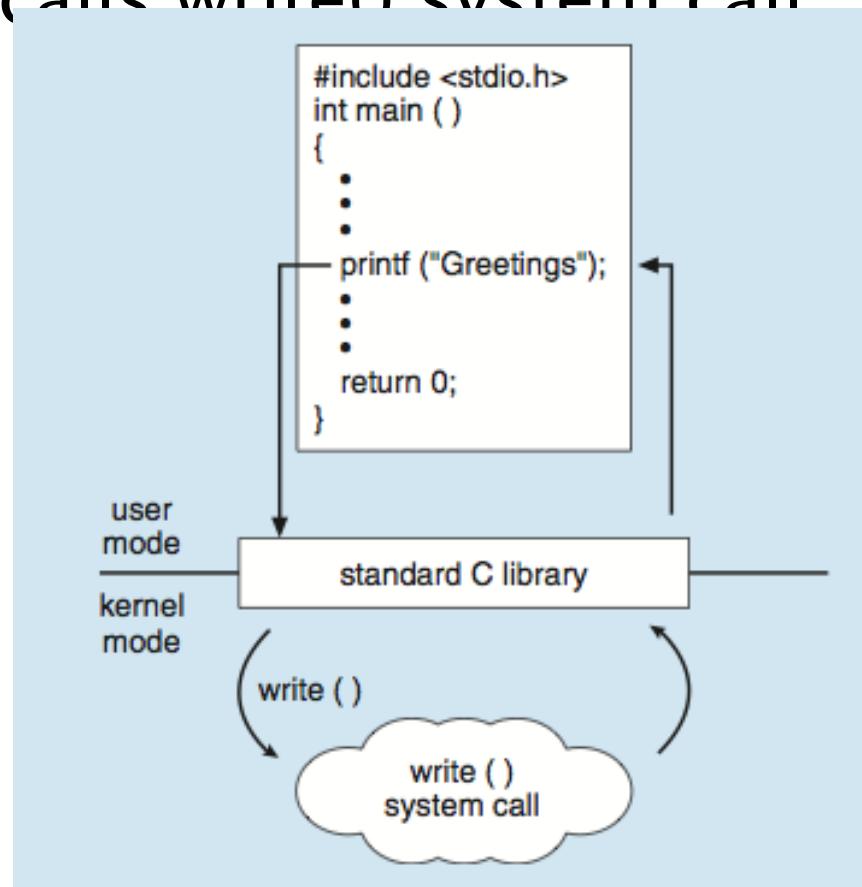
- ▶ create, delete communication connection
- ▶ send, receive messages
- ▶ transfer status information
- ▶ attach or detach remote device

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

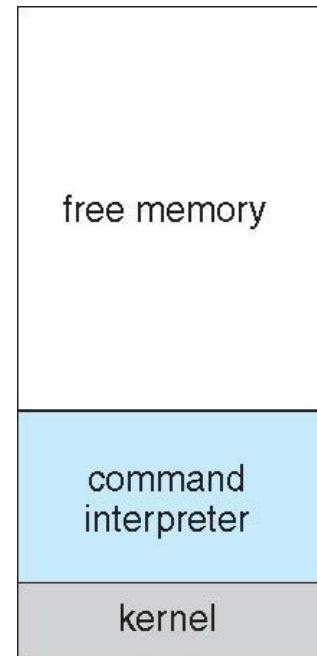
# Standard C Library Example

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



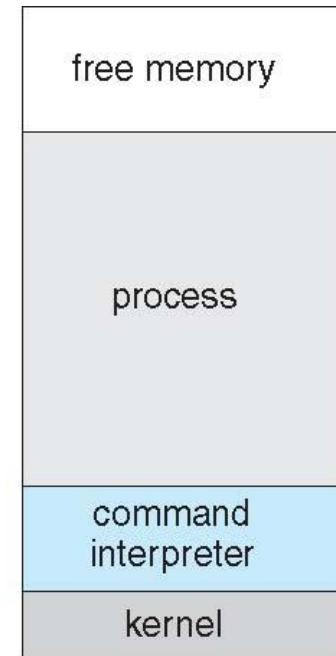
# Example: MS-DOS

- ▶ Single-tasking
- ▶ Shell invoked when system booted
- ▶ Simple method to run program
  - No process created
- ▶ Single memory space
- ▶ Loads program into memory, overwriting all but the kernel
- ▶ Program exit -> shell reloaded



(a)

At system startup

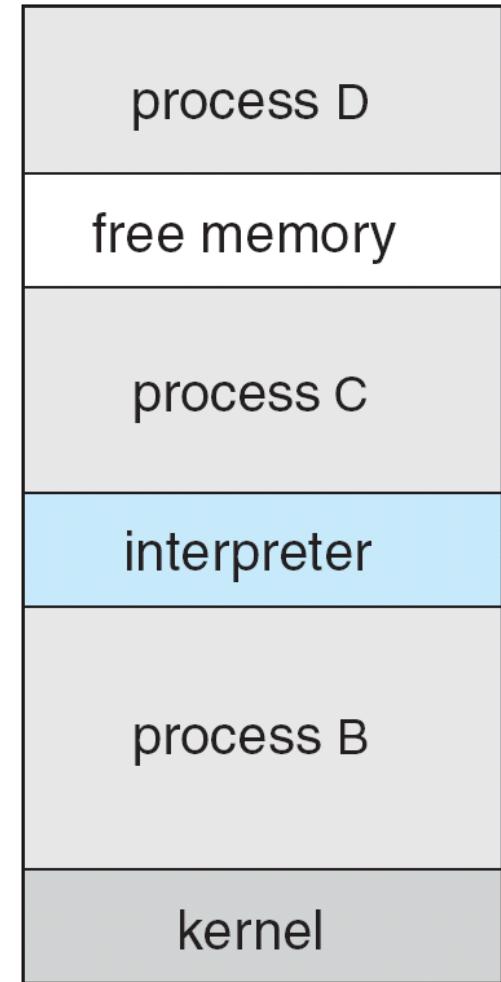


(b)

running a program

# Example: FreeBSD

- ▶ Unix variant
- ▶ Multitasking
- ▶ User login -> invoke user's choice of shell
- ▶ Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- ▶ Process exits with:
  - code = 0 – no error
  - code > 0 – error code



# System Programs

- ▶ System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

# System Programs

- ▶ **File management** – Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- ▶ **Status information**
  - Some ask the system for info – date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information

# System Programs (Cont.)

- ▶ **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- ▶ **Programming-language support –**  
Compilers, assemblers, debuggers and interpreters sometimes provided
- ▶ **Program loading and execution –**  
debugging systems for higher-level and machine language
- ▶ **Communications –** Provide the mechanism for creating virtual connections among processes, users, and computer systems

# System Programs (Cont.)

## ▶ **Background Services**

- Launch at boot time
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services, subsystems, daemons**

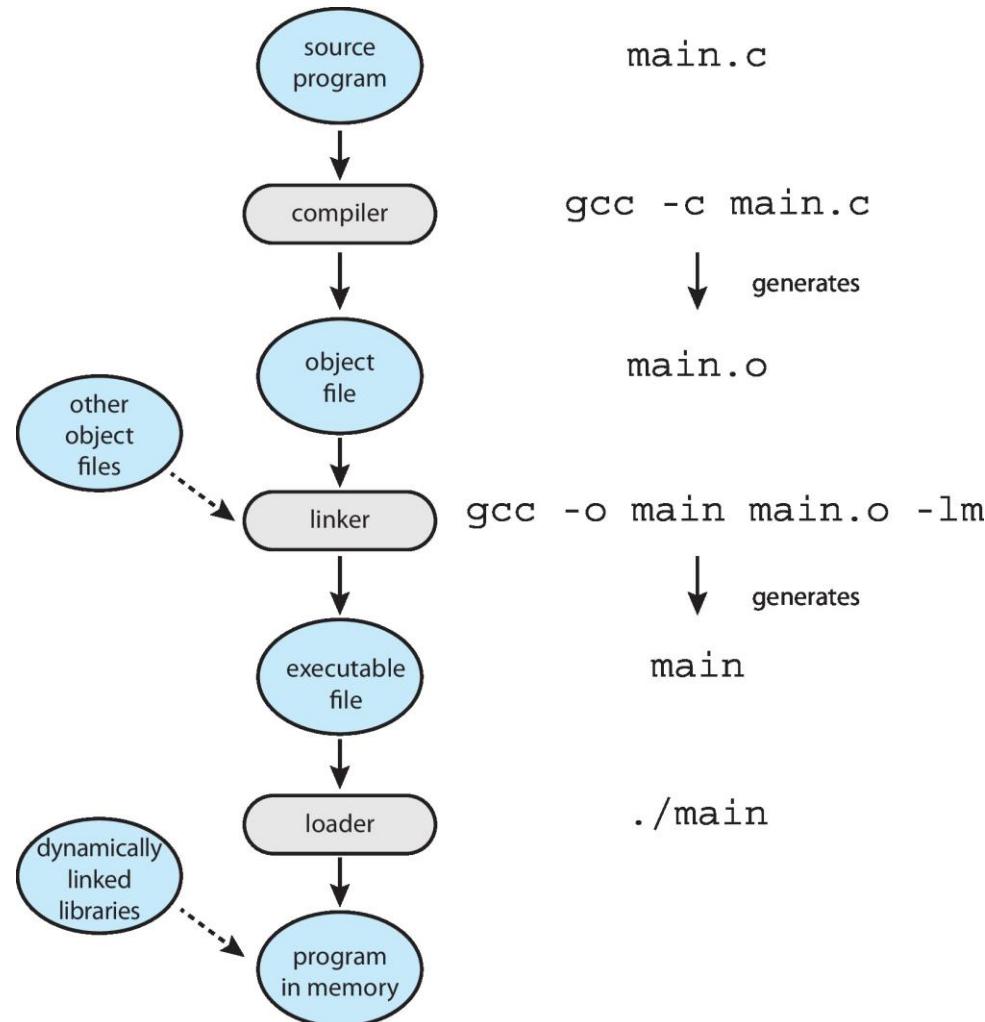
## ▶ **Application programs**

- Run by users

# Linkers and Loaders

- ▶ Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- ▶ **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- ▶ Program resides on secondary storage as binary executable
- ▶ Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- ▶ Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- ▶ Object, executable files have standard formats, so operating system knows how to load and start them

# The Role of the Linker and Loader



# Why Applications are Operating System Specific

- ▶ Apps compiled on one system usually not executable on other operating systems
- ▶ Each operating system provides its own unique system calls
  - Own file formats, etc
- ▶ Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- ▶ **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

# Operating System Design and Implementation

- ▶ Start the design by defining goals and specifications
- ▶ Affected by choice of hardware, type of system
- ▶ **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

# Implementation

- ▶ Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- ▶ Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- ▶ More high-level language easier to **port** to other hardware
  - But slower
- ▶ **Emulation** can allow an OS to run on non-native hardware

# Operating System Design and Implementation (Cont.)

- ▶ Important principle to separate  
**Policy**: *What* will be done?  
**Mechanism**: *How* to do it?
- ▶ Mechanisms determine how to do something, policies decide what will be done
- ▶ The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

# Operating System Structure

There are six different structures:

1. Simple Structure
2. Layered Systems
3. Microkernels
4. Modules
5. Hybrid Machines

# Operating System Structure

## 1. Monolithic Structure:

- ▶ DOS has no modern software engineering techniques,
- ▶ Does not break the system into subsystems, and has no distinction between user and kernel modes.
- ▶ Allow all programs direct access to hardware.

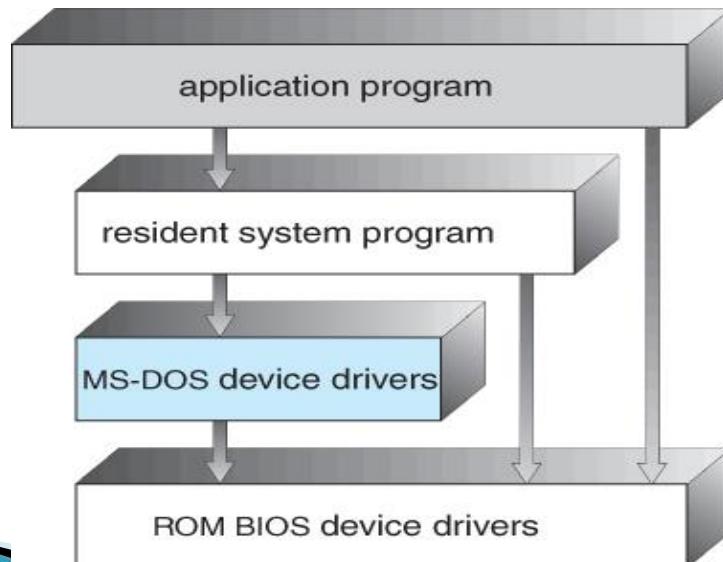


Fig: MS-DOS layer structure

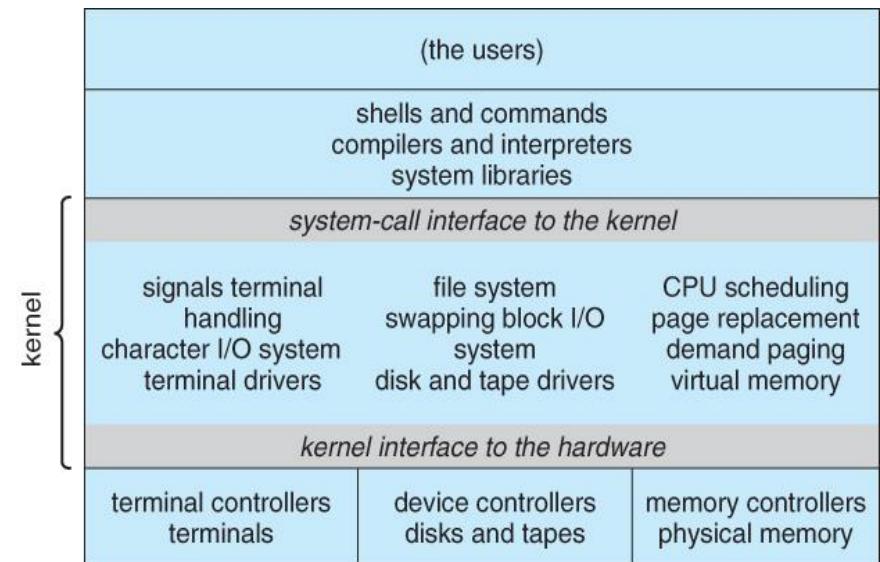
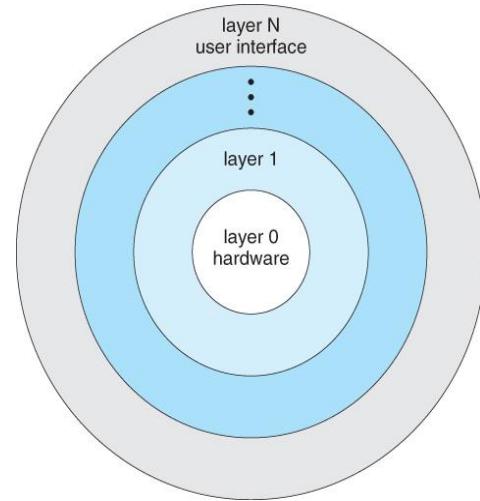


Fig: Traditional UNIX system structure

# Operating System Structure

## 2. Layered Approach:

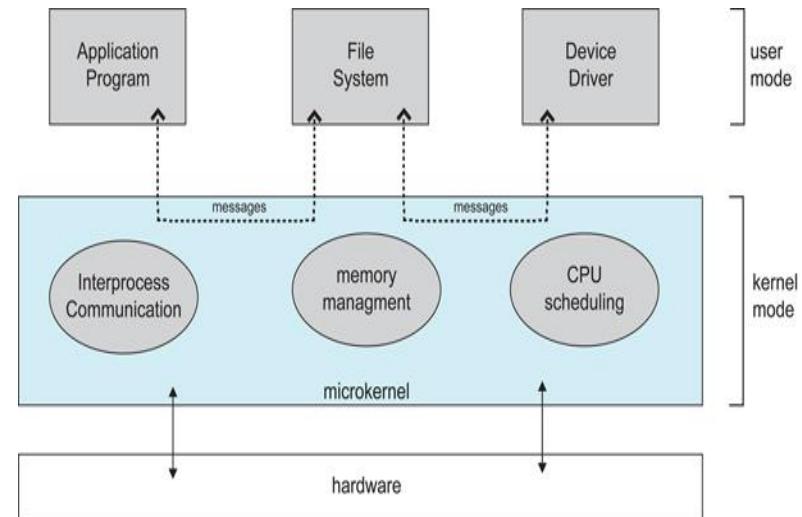
- ▶ Another approach is to break the OS into number of smaller layers and relies on the services provided by next lower layer.
- ▶ allows each layer to be developed and debugged independently, with assumption that all lower layers is already debugged.
- ▶ The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer.
- ▶ Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.



# Operating System Structure

## 3. Microkernels:

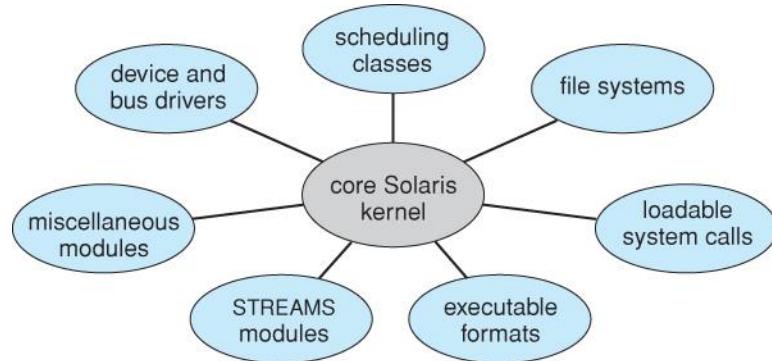
- ▶ micro kernels is to remove all non-essential services from the kernel, and implement them as system applications
- ▶ provide basic process and memory management, and message passing between other services
- ▶ Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- ▶ System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- ▶ Mach, Windows NT are examples of kernel



# Operating System Structure

## 4. Modules

- ▶ Modern OS development is object-oriented, with a relatively small core kernel and a set of *modules* which can be linked in dynamically.
- ▶ Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces.
- ▶ The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

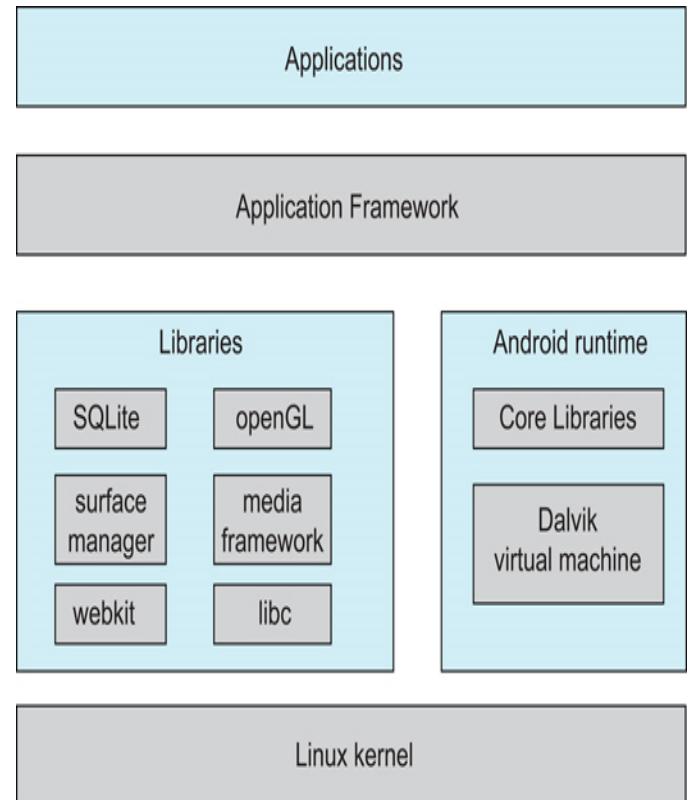


# Operating System Structure

**5. Hybrid Systems:** Most OS today do not strictly adhere to one architecture, but are hybrids of several.

**Android:**

- ▶ open-source OS
- ▶ includes versions of Linux and a JVM
- ▶ apps are developed using Java



# Operating-System Debugging

- ▶ Debugging is finding and fixing errors, or bugs
- ▶ OS generate log files containing error information
- ▶ Failure of an application can generate core dump file capturing memory of the process
- ▶ Operating system failure can generate crash dump file containing kernel memory

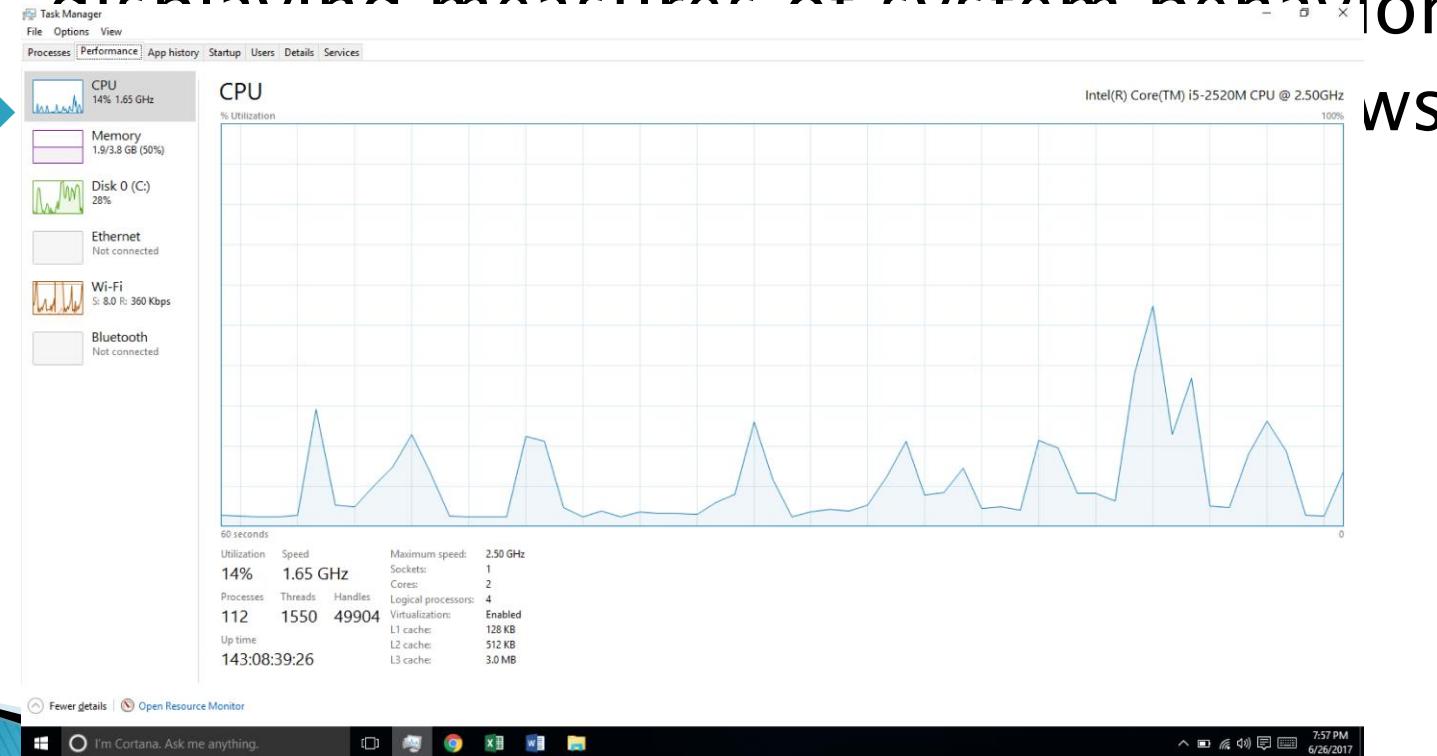
Kernighan's Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

# Operating System Generation

- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned

# Performance Tuning

- ▶ Improve performance by removing bottlenecks
- ▶ OS must provide means of computing and displaying measures of system behavior



# Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

# BCC

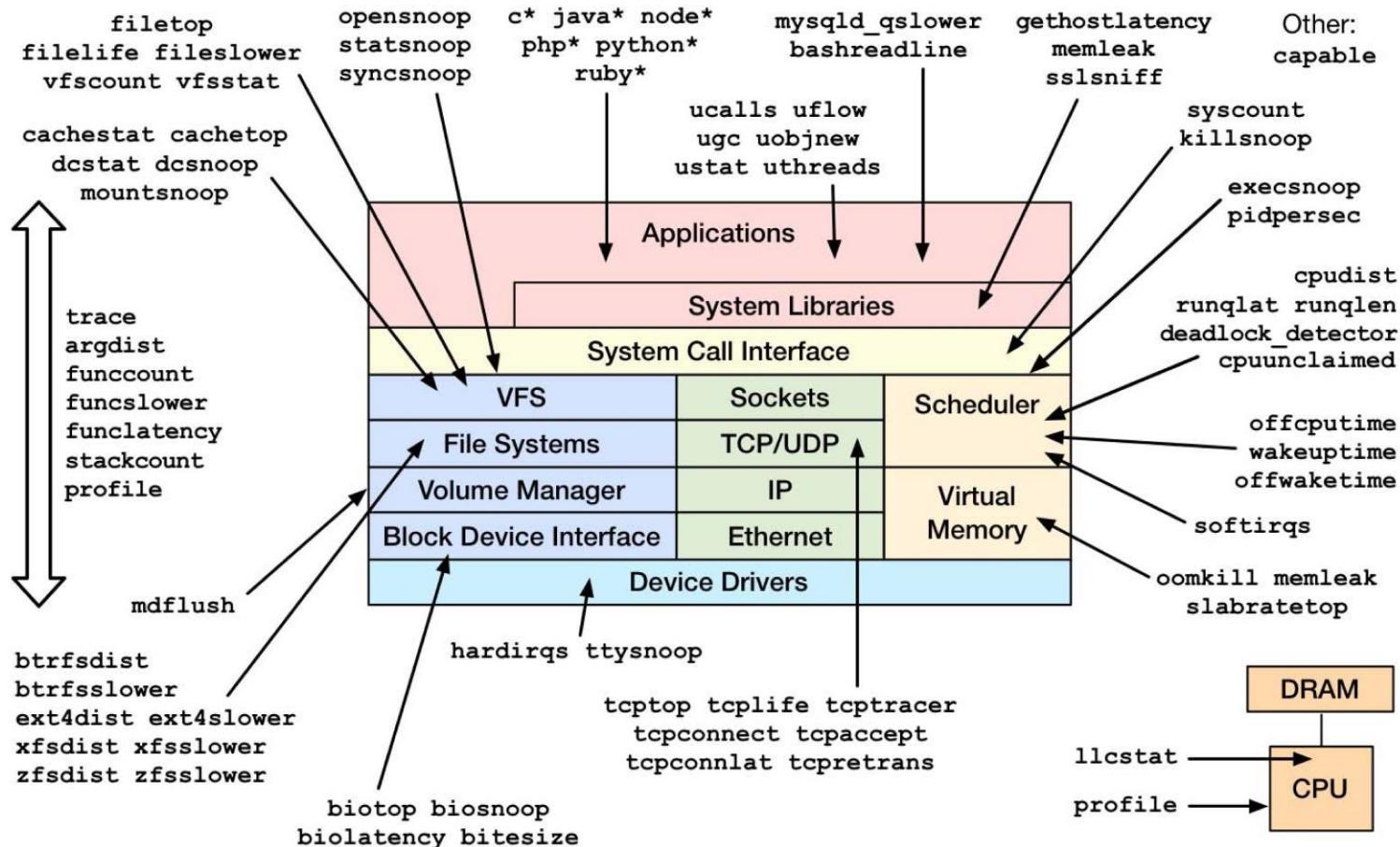
- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and can instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, `disksnop.py` traces disk I/O activity

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

- Many other tools (next slide)

# Linux bcc/BPF Tracing Tools

## Linux bcc/BPF Tracing Tools



# Processes

**Course Instructor: Nausheen Shoaib**

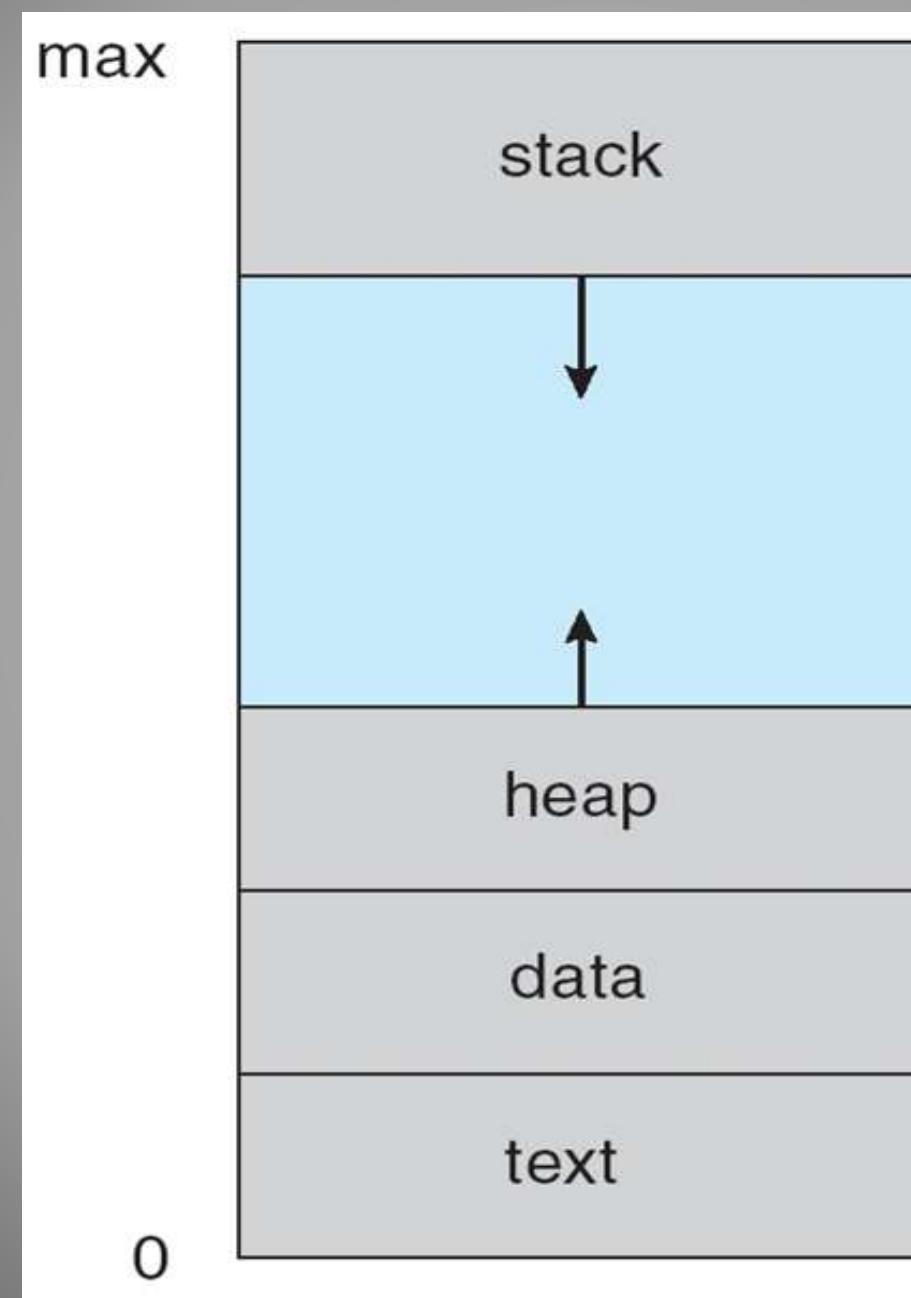
# Process Concept [1 / 2]

- ▶ An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- ▶ Process – a program in execution; process execution must progress in sequential fashion
- ▶ A process includes:
  - program counter
  - stack
  - data section

# The Process [2 / 2]

- ▶ Multiple parts
  - ❖ The program code, also called **text section**
  - ❖ Current activity including **program counter**, processor registers
  - ❖ **Stack** containing temporary data
    - ❖ Function parameters, return addresses, local variables
  - ❖ **Data section** containing global variables
  - ❖ **Heap** containing memory dynamically allocated during run time
- ▶ Program is passive entity, process is active
  - ❖ Program becomes process when executable file loaded into memory
- ▶ Execution of program started via GUI mouse clicks, cmd line etc

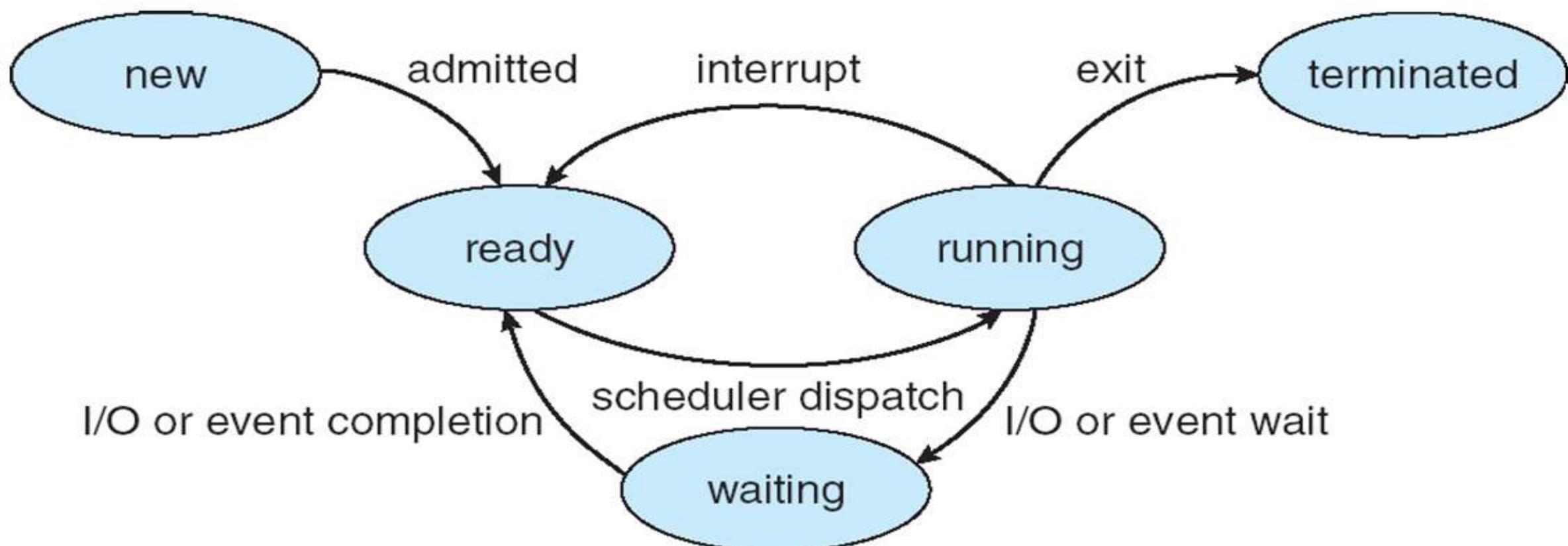
# Process in Memory



# Process State

- ▶ As a process executes, it changes *state*
  - ❖ **new**: The process is being created
  - ❖ **running**: Instructions are being executed
  - ❖ **waiting**: The process is waiting for some event to occur
  - ❖ **ready**: The process is waiting to be assigned to a processor
  - ❖ **terminated**: The process has finished execution

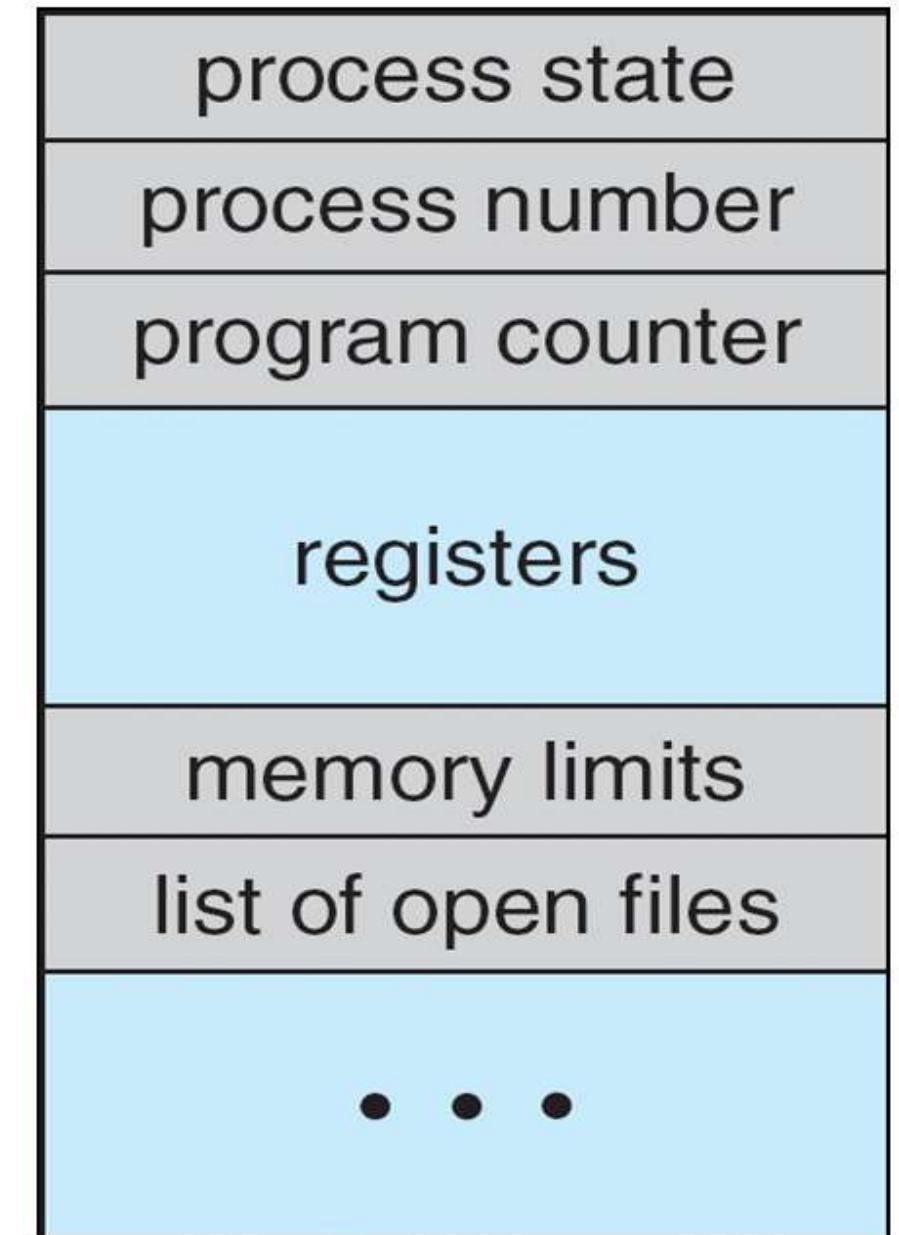
# Diagram of Process State



# Process Control Block (PCB)

Information associated with each process  
(also called **task control block**)

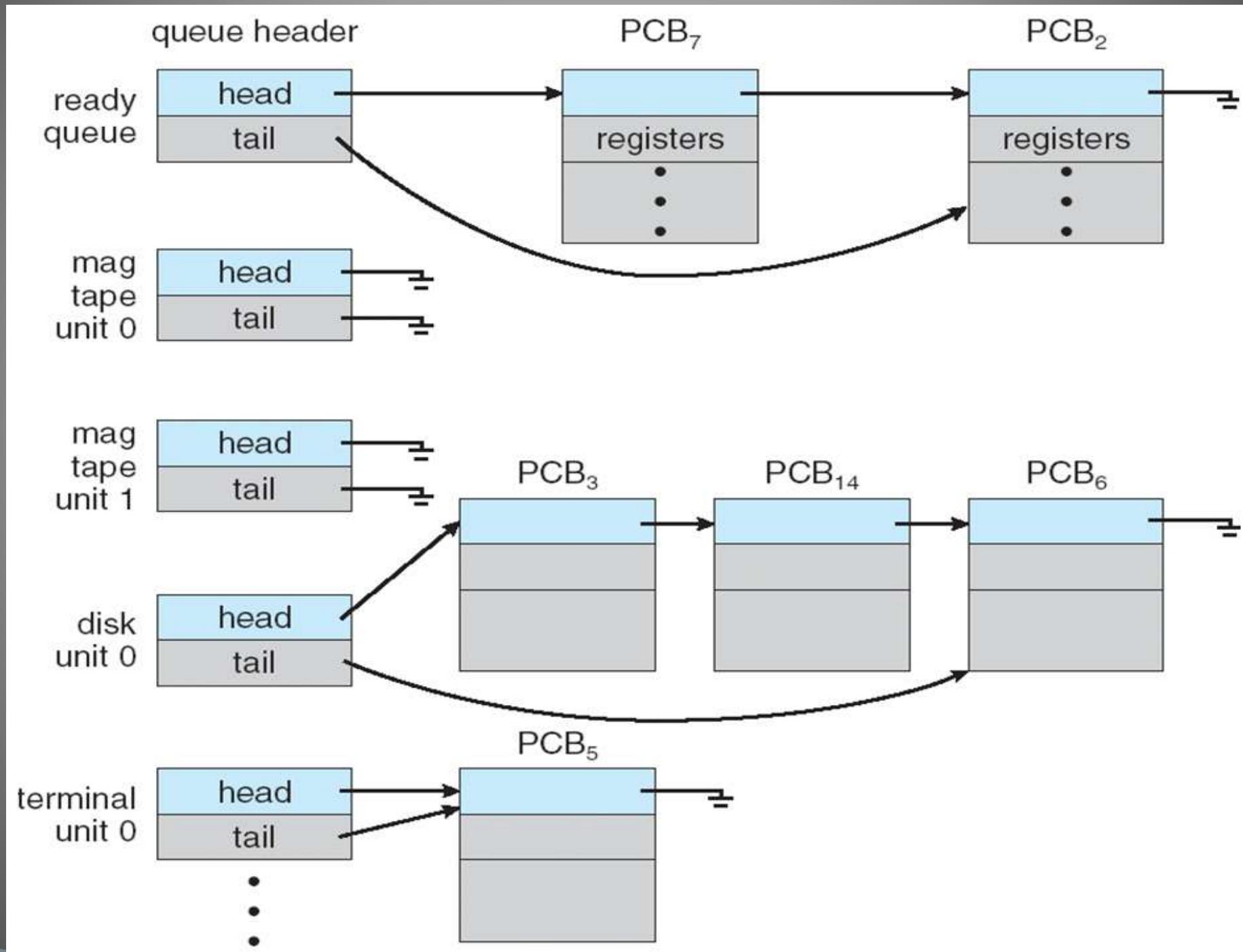
- ▶ Process state – running, waiting, etc
- ▶ Program counter – location of instruction to next execute
- ▶ CPU registers – contents of all process-centric registers
- ▶ CPU scheduling information– priorities, scheduling queue pointers
- ▶ Memory-management information – memory allocated to the process
- ▶ Accounting information – CPU used, clock time elapsed since start, time limits
- ▶ I/O status information – I/O devices allocated to process, list of open files



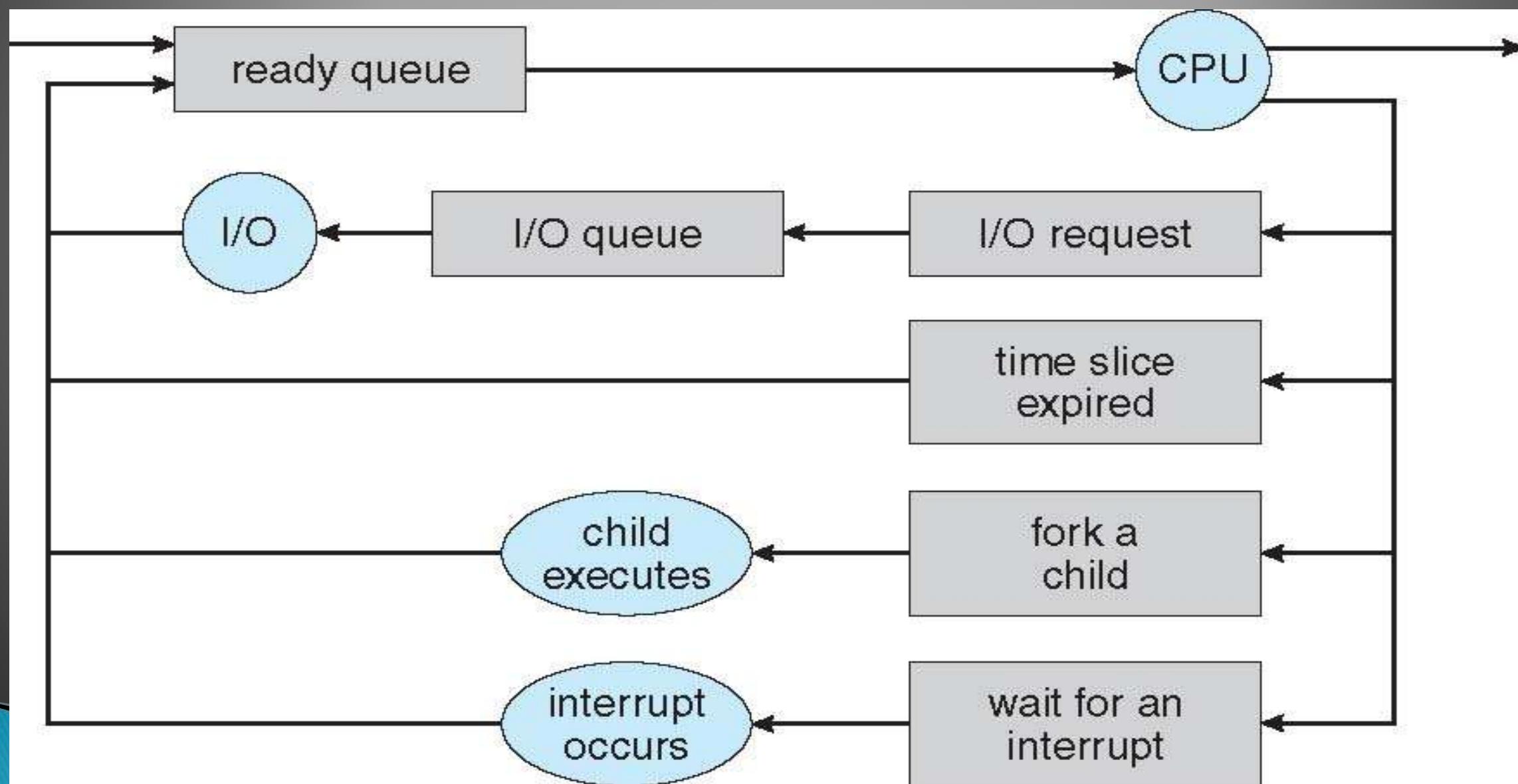
# Process Scheduling

- ▶ Maximize CPU use, quickly switch processes onto CPU for time sharing
- ▶ **Process scheduler** selects among available processes for next execution on CPU
- ▶ Maintains **scheduling queues** of processes:
  - ❖ **Job queue** – set of all processes in the system
  - ❖ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - ❖ **Device queues** – set of processes waiting for an I/O device

# Ready Queue And Various I/O Device Queues



# Representation of Process Scheduling



# Schedulers

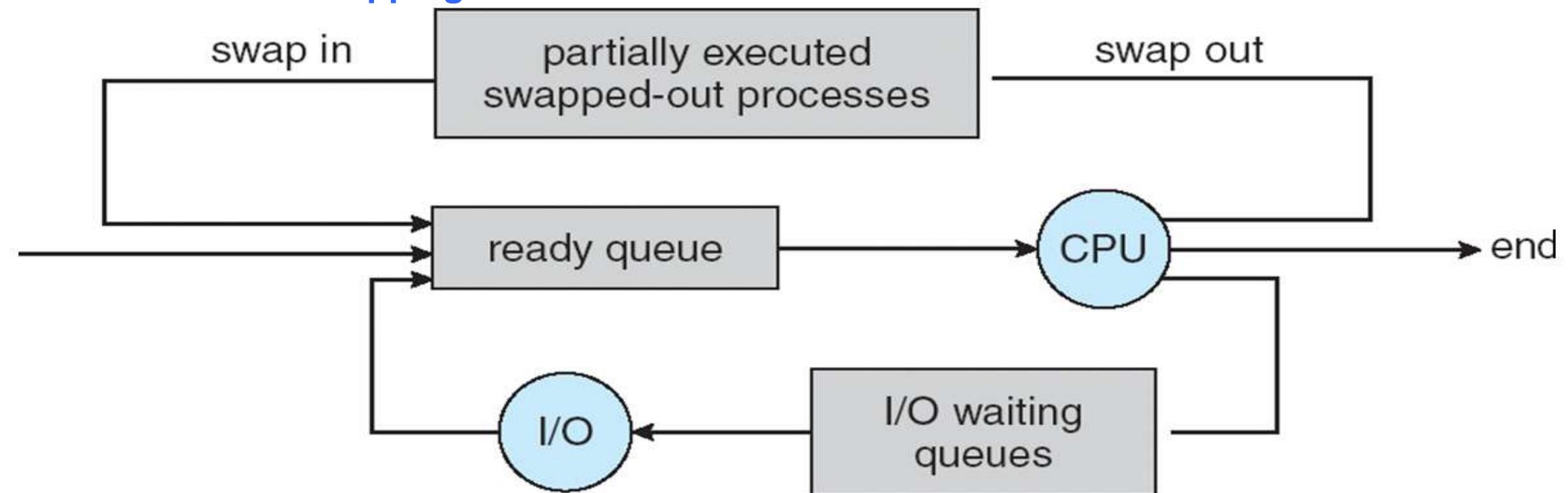
- ▶ **Long-term scheduler** (or job scheduler) - selects which processes should be brought into the ready queue
- ▶ **Short-term scheduler** (or CPU scheduler) - selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

# Schedulers (Cont.)

- ▶ Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- ▶ Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- ▶ The long-term scheduler controls the *degree of multiprogramming*
- ▶ Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

# Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



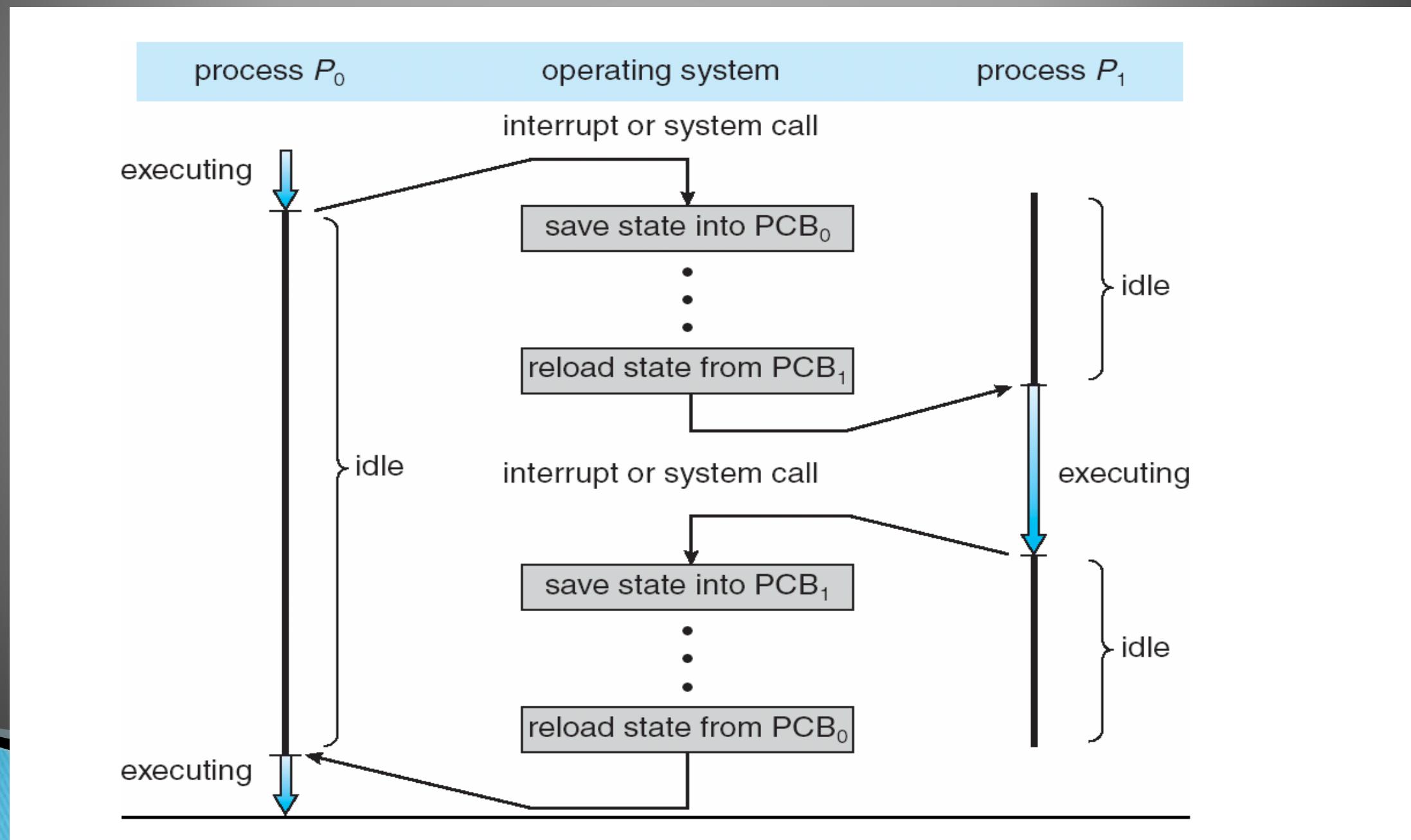
# Multitasking in Mobile Systems

- ▶ Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- ▶ Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process – controlled via user interface
  - Multiple **background** processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

# Context Switch

- ▶ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- ▶ **Context** of a process represented in the PCB
- ▶ Context-switch time is overhead; the system does no useful work while switching

# CPU Switch From Process to Process (Context Switching)



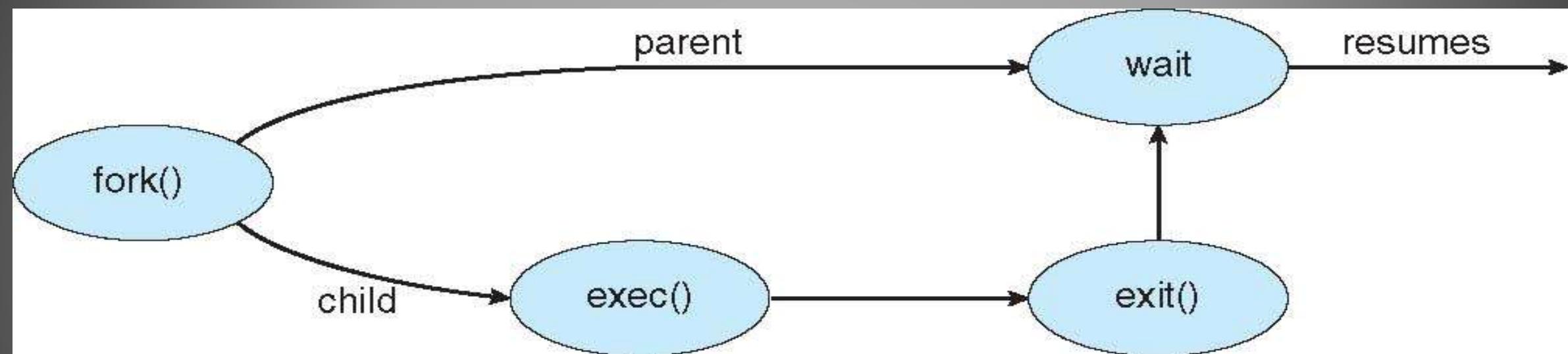
# Process Creation

- ▶ Parent process create **children** processes, which, in turn create other processes, forming a tree of processes
- ▶ Generally, process identified and managed via a **process identifier (pid)**
- ▶ Resource sharing Options
  - Parent and children share all resources
  - Children share subset of parent's resource
  - Parent and child share no resources
- ▶ Execution Options
  - Parent and children execute concurrently
  - Parent waits until children terminate

# Process Creation (Cont.)

- ▶ Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- ▶ UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

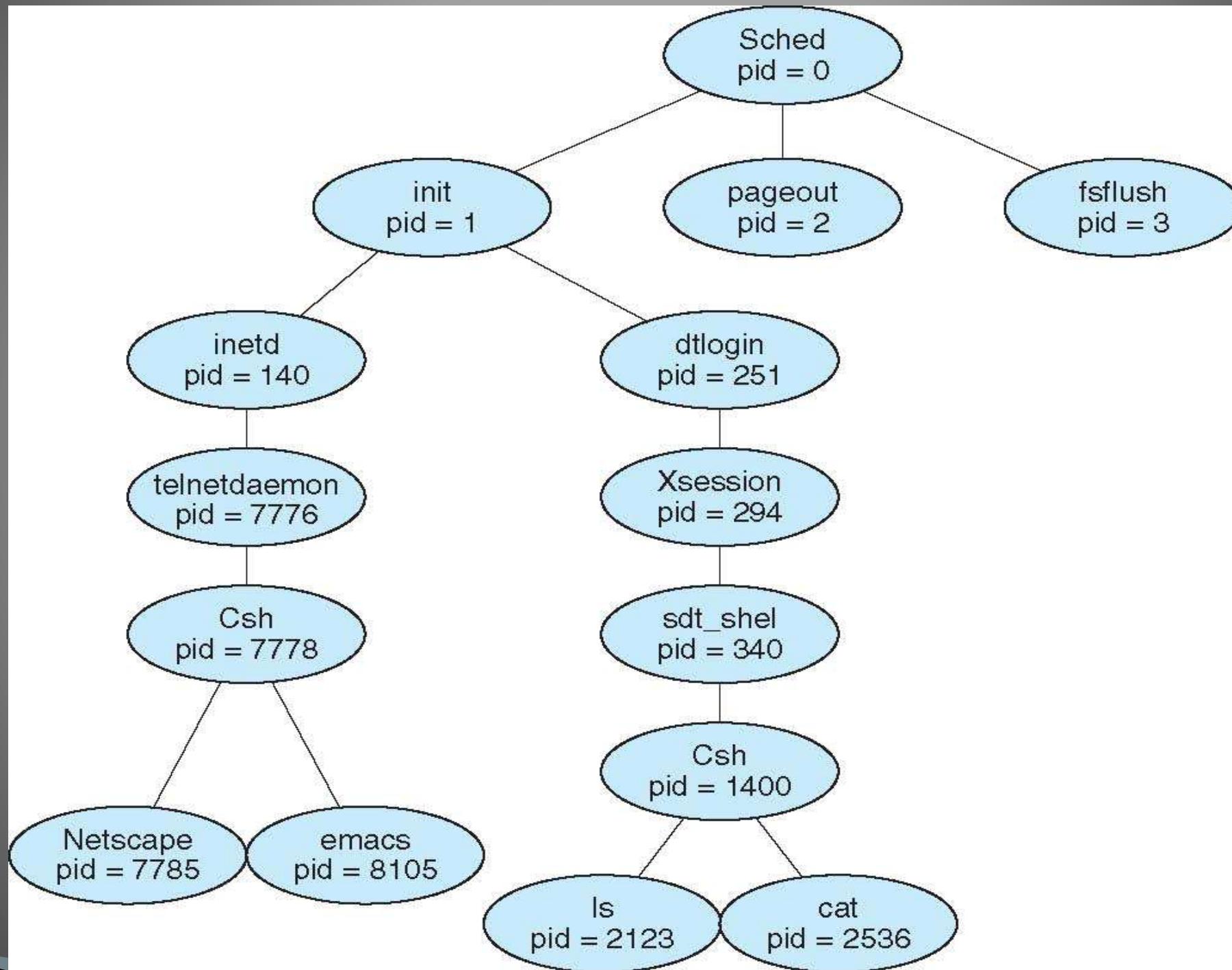
# Process Creation



# C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

# A Tree of Processes on Solaris



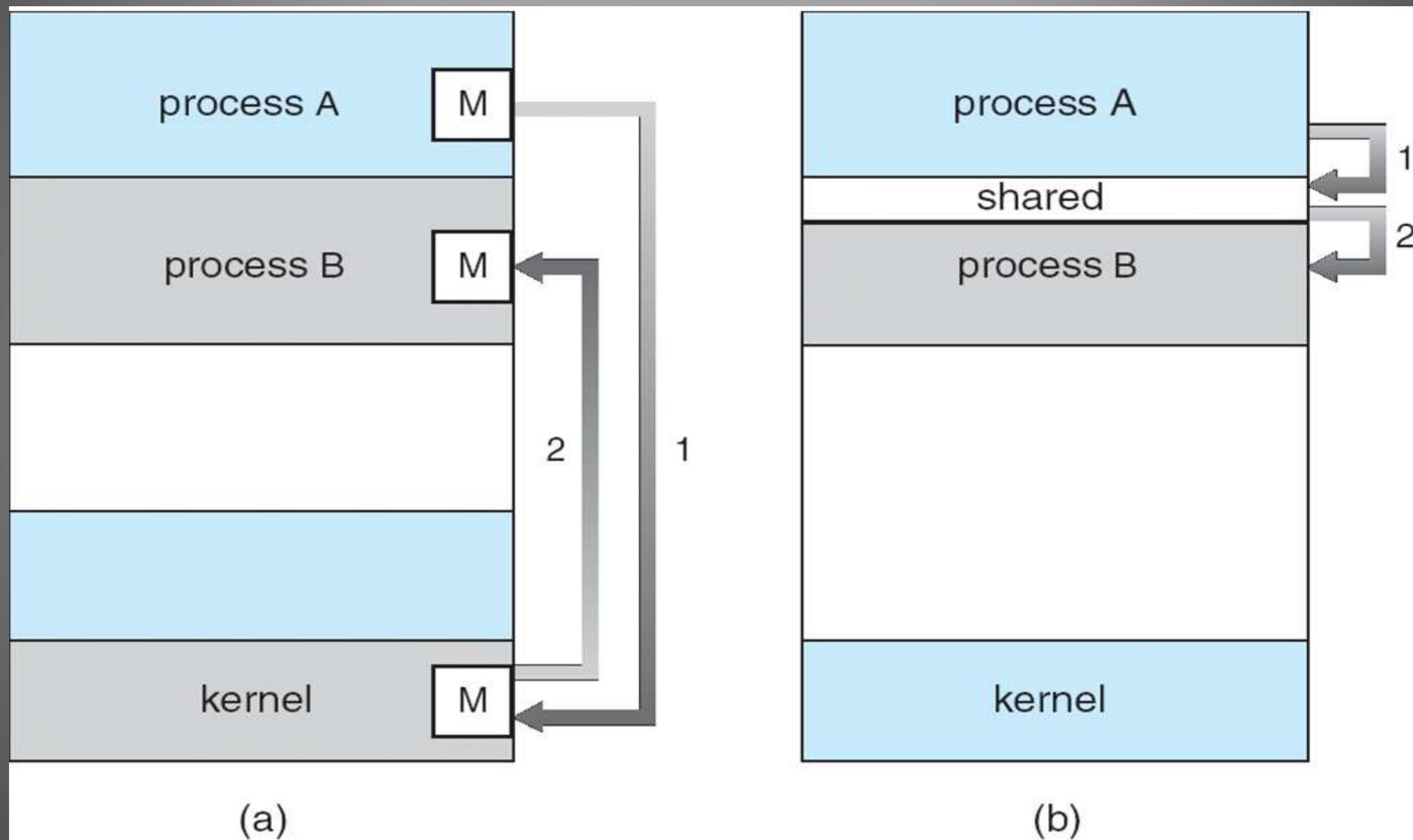
# Process Termination

- ▶ Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- ▶ Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
    - All children terminated – **cascading termination**

# Interprocess Communication

- ▶ Processes within a system may be **independent** or **cooperating**
- ▶ Cooperating process can affect or be affected by other processes, including sharing data
- ▶ Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- ▶ Cooperating processes need **interprocess communication (IPC)**
- ▶ Two models of IPC
  - Shared memory
  - Message passing

# Communications Models



# Cooperating Processes

- ▶ **Independent** process cannot affect or be affected by the execution of another process
- ▶ **Cooperating** process can affect or be affected by the execution of another process
- ▶ Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

# Producer-Consumer Problem

- ▶ Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

# Bounded-Buffer – Shared-Memory Solution

## ▶ Shared data

```
#define BUFFER_SIZE 10
typedef struct {

    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

# Bounded-Buffer – Producer

```
while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count) == out)
        ; /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

# Bounded Buffer – Consumer

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

# Interprocess Communication – Message Passing

- ▶ Mechanism for processes to communicate and to synchronize their actions
- ▶ IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- ▶ If  $P$  and  $Q$  wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- ▶ Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

# Implementation Questions

- ▶ How are links established?
- ▶ Can a link be associated with more than two processes?
- ▶ How many links can there be between every pair of communicating processes?
- ▶ What is the capacity of a link?
- ▶ Is the size of a message that the link can accommodate fixed or variable?
- ▶ Is a link unidirectional or bi-directional?

# Direct Communication

- ▶ Processes must name each other explicitly:
  - **send** ( $P$ , *message*) – send a message to process P
  - **receive**( $Q$ , *message*) – receive a message from process Q
- ▶ Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

# Indirect Communication

- ▶ Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- ▶ Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

# Indirect Communication

- ▶ Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- ▶ Primitives are defined as:  
 **$\text{send}(A, \text{message})$**  – send a message to mailbox A  
 **$\text{receive}(A, \text{message})$**  – receive a message from mailbox A

# Indirect Communication

- ▶ Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive
  - Who gets the message?
- ▶ Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

# Synchronization

- ▶ Message passing may be either blocking or non-blocking
- ▶ **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- ▶ **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

# Buffering

- ▶ Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages waiting in a queue.
  2. Bounded capacity – finite length of  $n$  messages,  $n$  messages waiting in a queue
  3. Unbounded capacity – infinite length ,any number of messages can wait in a queue

# Examples of IPC Systems – POSIX

## ▶ POSIX Shared Memory

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S  
IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared  
memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

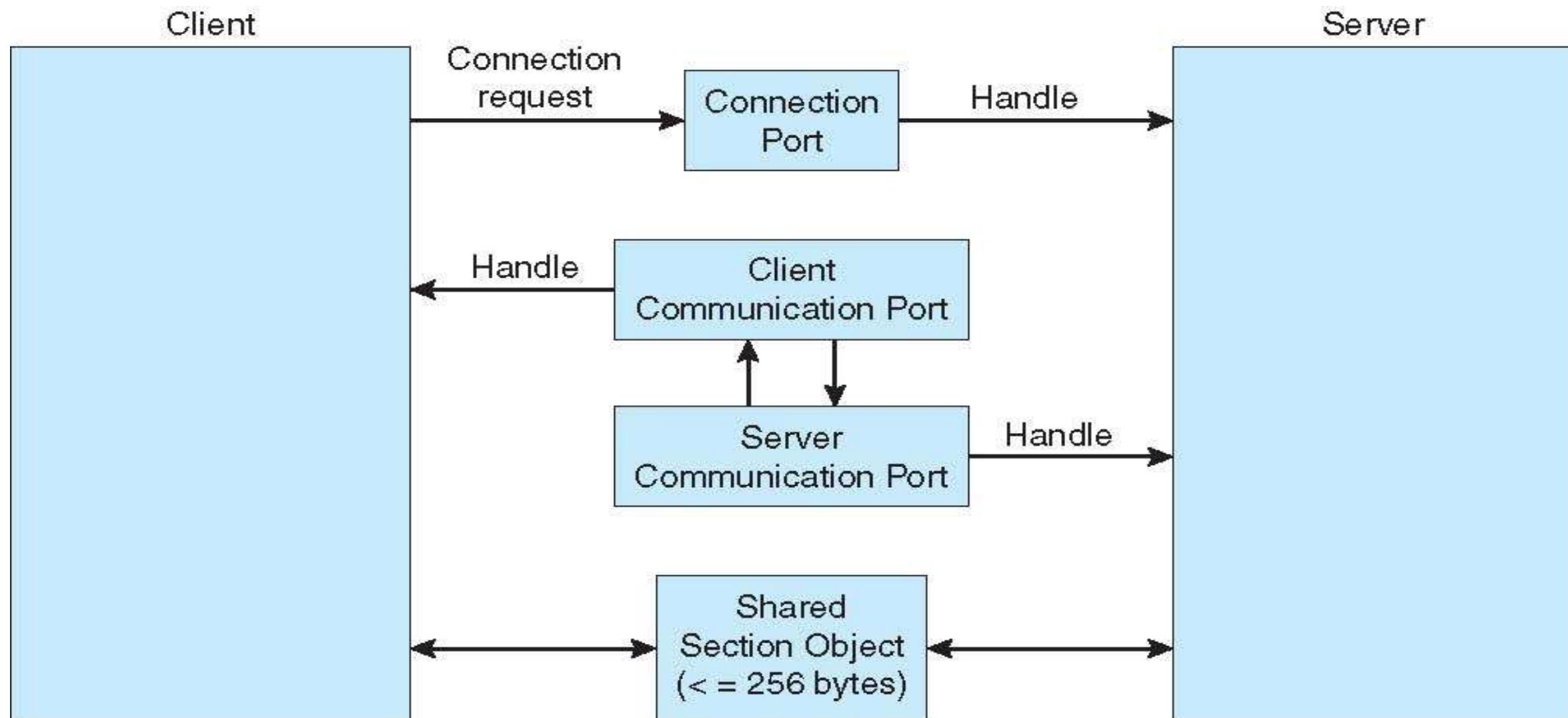
# Examples of IPC Systems – Mach

- ▶ Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation– Kernel and Notify
  - Only three system calls needed for message transfer  
`msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via  
`port_allocate()`

# Examples of IPC Systems – Windows XP

- ▶ Message-passing centric via **local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's connection port object.
    - The client sends a connection request.
    - The server creates two private communication ports and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

# Local Procedure Calls in Windows XP



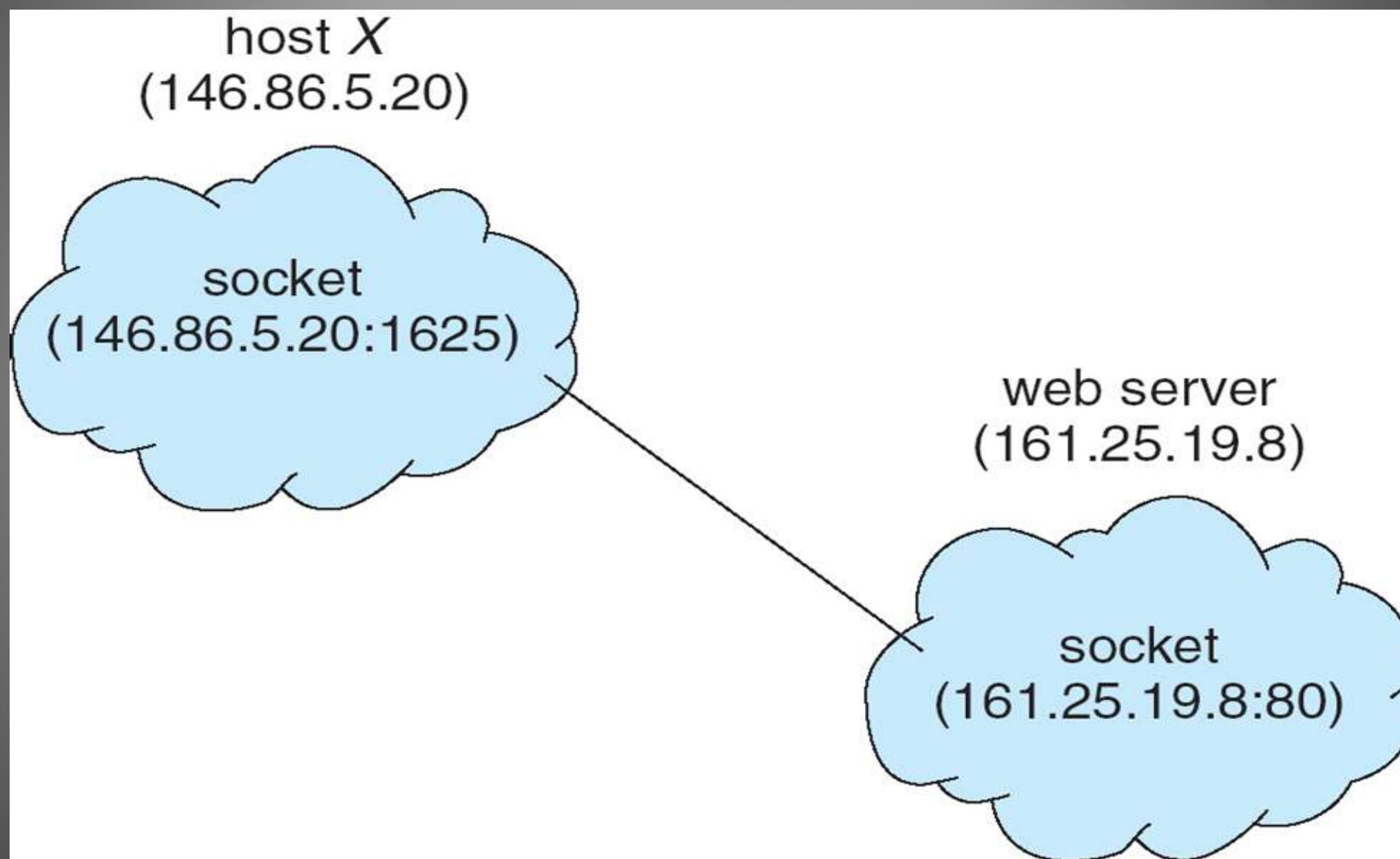
# Communications in Client–Server Systems

- ▶ Sockets
- ▶ Remote Procedure Calls
- ▶ Pipes

# Sockets

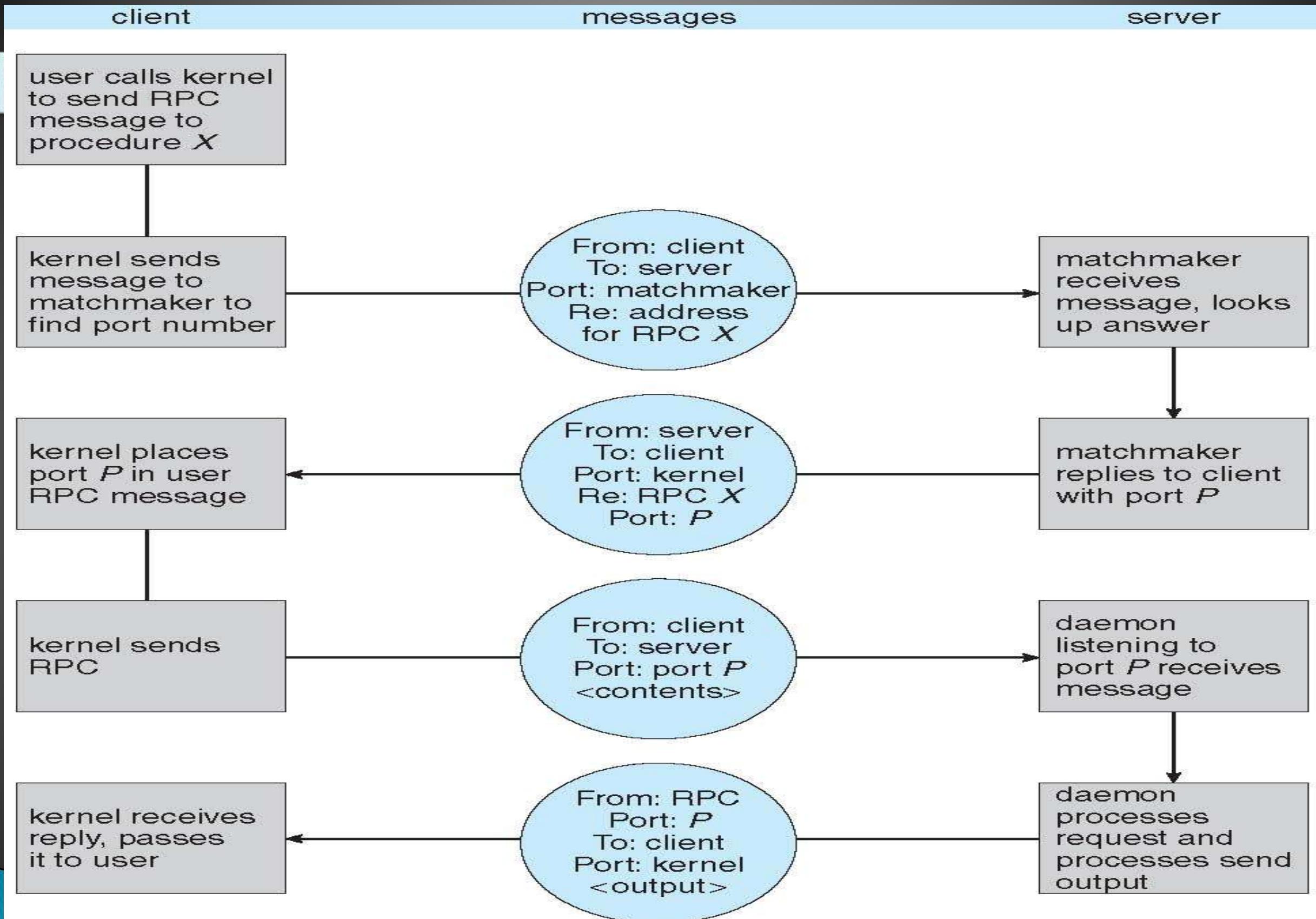
- ▶ A **socket** is defined as an *endpoint for communication*
- ▶ Concatenation of IP address and port
- ▶ The socket **161.25.19.8:1625** refers to port 1625 on host 161.25.19.8
- ▶ Communication consists between a pair of sockets

# Socket Communication



# Remote Procedure Calls

- ▶ Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- ▶ **Stubs** – client-side proxy for the actual procedure on the server
- ▶ The client-side stub locates the server and *marshalls* the parameters
- ▶ The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



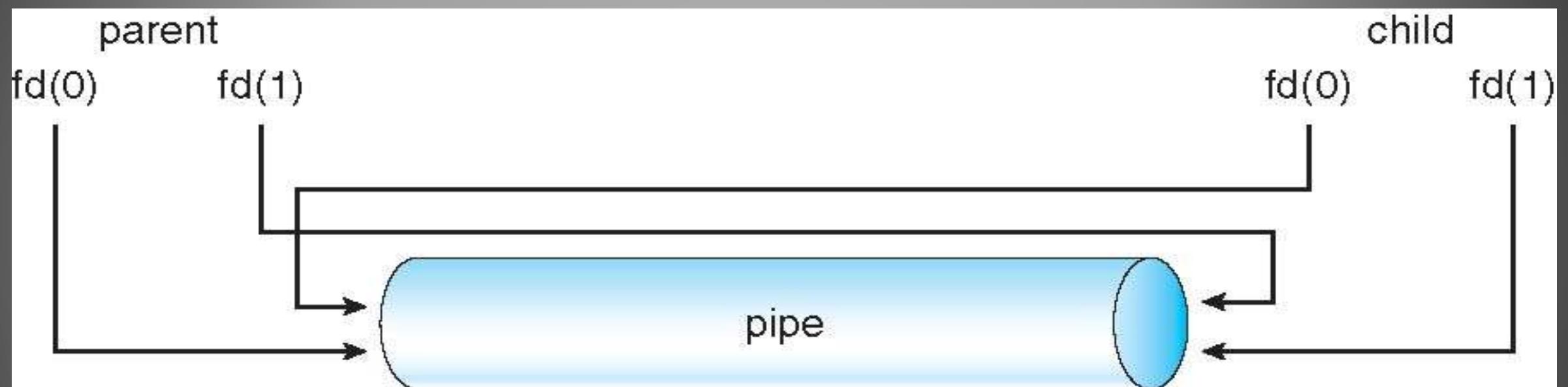
# Pipes

- ▶ Acts as a conduit allowing two processes to communicate
- ▶ Issues
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

# Ordinary Pipes

- ▶ Ordinary Pipes allow communication in standard producer–consumer style
- ▶ Producer writes to one end (the *write-end* of the pipe)
- ▶ Consumer reads from the other end (the *read-end* of the pipe)
- ▶ Ordinary pipes are therefore unidirectional
- ▶ Require parent–child relationship between communicating processes

# Ordinary Pipes



# Named Pipes

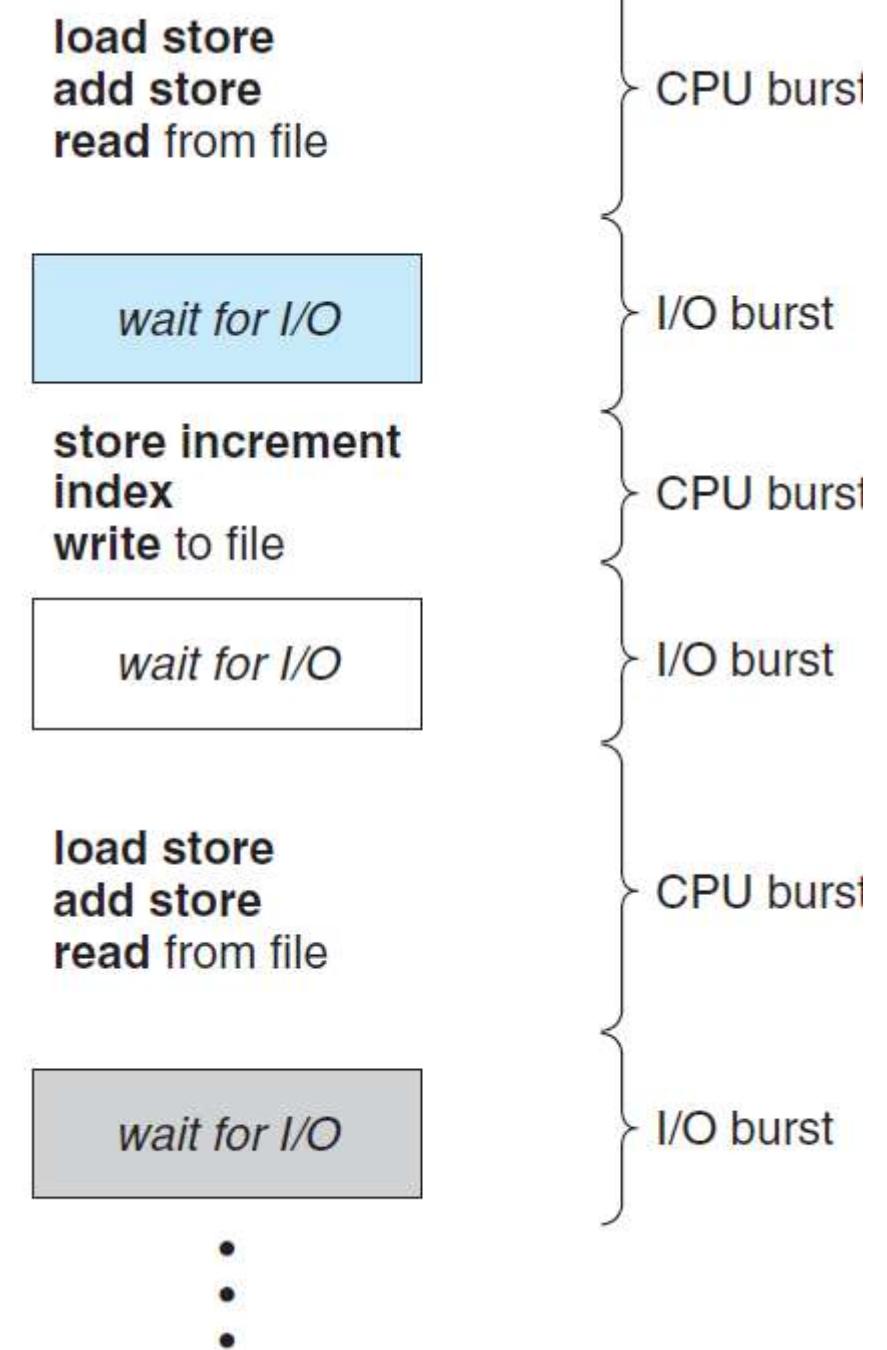
- ▶ Named Pipes are more powerful than ordinary pipes
- ▶ Communication is bidirectional
- ▶ No parent-child relationship is necessary between the communicating processes
- ▶ Several processes can use the named pipe for communication
- ▶ Provided on both UNIX and Windows systems

# CPU Scheduling

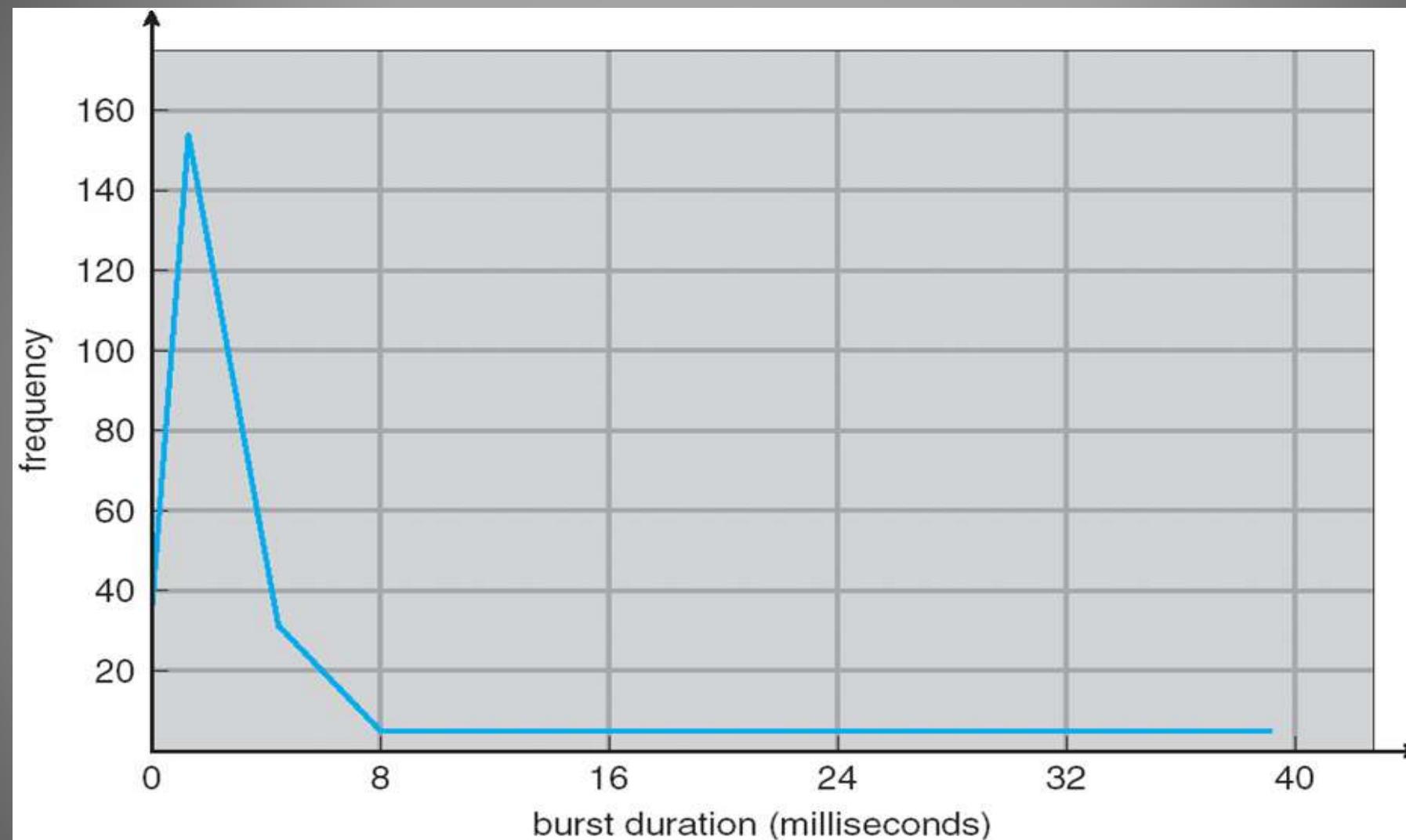
## Course Instructor: Nausheen Shoaib

# Basic Concepts

- ▶ Maximum CPU utilization obtained with multiprogramming
- ▶ CPU-I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- ▶ **CPU burst** followed by **I/O burst**
- ▶ CPU burst distribution is of main concern



# Histogram of CPU-burst Times



# Preemptive Vs. Non Preemptive Scheduling

1. The basic difference between preemptive and non-preemptive scheduling is that in preemptive scheduling the CPU is allocated to the processes for the **limited** time. While in Non-preemptive scheduling, the CPU is allocated to the process till it **terminates** or switches to **waiting state**.
2. The executing process in preemptive scheduling is **interrupted** in the middle of execution whereas, the executing process in non-preemptive scheduling is **not interrupted** in the middle of execution.
3. Preemptive Scheduling has the **overhead** of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. On the other hands, non-preemptive scheduling has **no overhead** of switching the process from running state to ready state.
4. In preemptive scheduling, if a process with high priority frequently arrives in the ready queue then the process with low priority have to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process with larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling is quite **flexible** because the critical processes are allowed to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is **rigid** as even if a critical process enters the ready queue the process running CPU is not disturbed.

# Dispatcher

- ▶ Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- ▶ **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

# Scheduling Criteria

- ▶ **CPU utilization** – keep the CPU as busy as possible
- ▶ **Throughput** – # of processes that complete their execution per time unit
- ▶ **Turnaround time** – amount of time to execute a particular process
- ▶ Turnaround time (TAT)=Completion time – Arrival time
- ▶ **Waiting time** – amount of time a process has been waiting in the ready queue
- ▶ **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

# Scheduling Algorithm Optimization Criteria

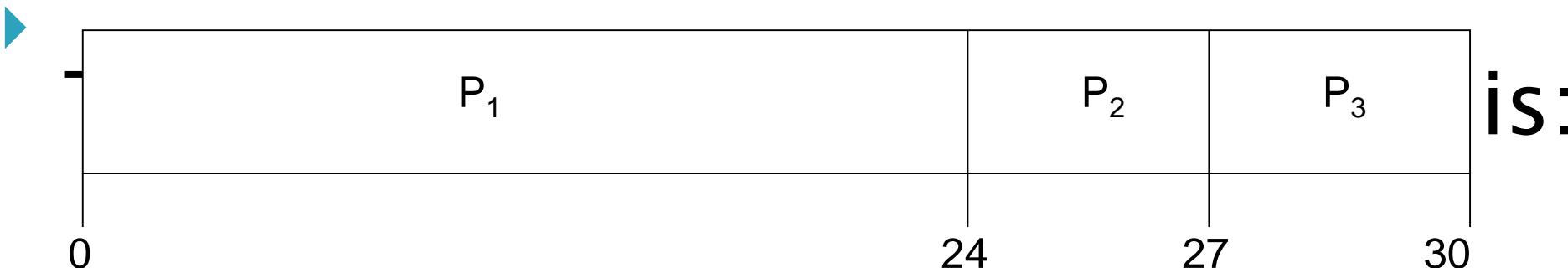
- ▶ Max CPU utilization
- ▶ Max throughput
- ▶ Min turnaround time
- ▶ Min waiting time
- ▶ Min response time

# First-Come, First-Served (FCFS) Scheduling

Process Burst Time

$P_1$	24
$P_2$	3
$P_3$	3

- ▶ Suppose that the processes arrive in the order:  $P_1$  ,  $P_2$  ,  $P_3$



# Example FCFS

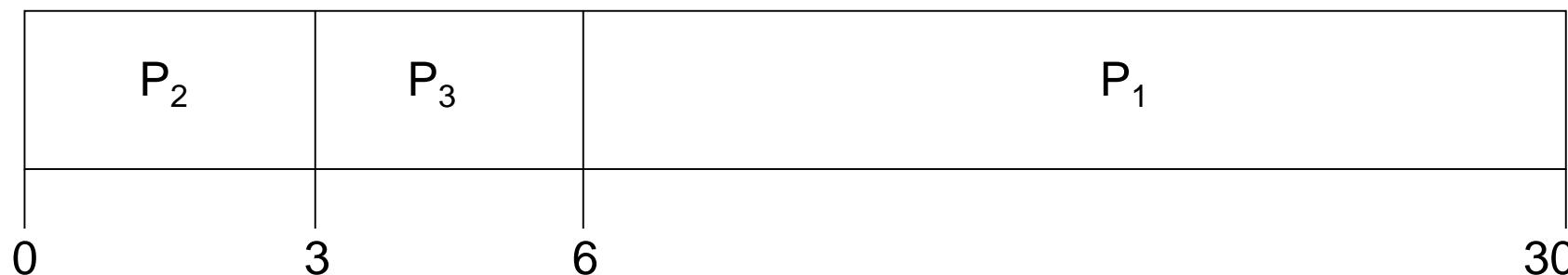
- ▶ Covered in class

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- ▶ The Gantt chart for the schedule is:



- ▶ Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- ▶ Average waiting time:  $(6 + 0 + 3)/3 = 3$
- ▶ Much better than previous case
- ▶ **Convoy effect** – short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

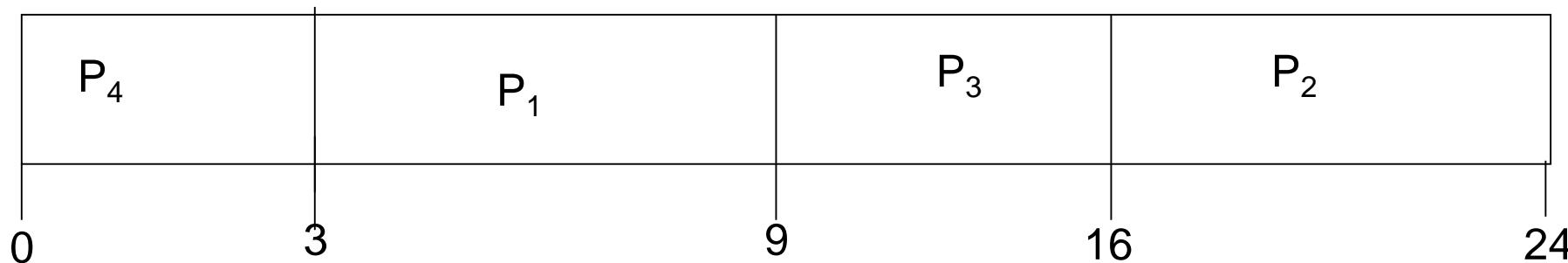
# Shortest-Job-First (SJF) Scheduling

- ▶ Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- ▶ SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- ▶ SJF scheduling chart



- ▶ Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# Example of SJF

- ▶ Covered in class

# Determining Length of Next CPU Burst

- ▶ Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- ▶ Can be done by using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

- ▶  $t_n$  = most recent info
- ▶  $T_n$  = Past history
- ▶ If  $\alpha=0$  then recent history has no effect
- ▶ If  $\alpha=1$ , then recent CPU burst matters
- ▶ Preemptive version called **shortest-remaining-time-first**

# Shortest-remaining-time-first

- ▶ Now we add the concepts of varying arrival times and preemption to the analysis:
- ▶ the process with the smallest amount of time remaining until completion is selected to execute.

# Example of SRTF

- ▶ Covered in class

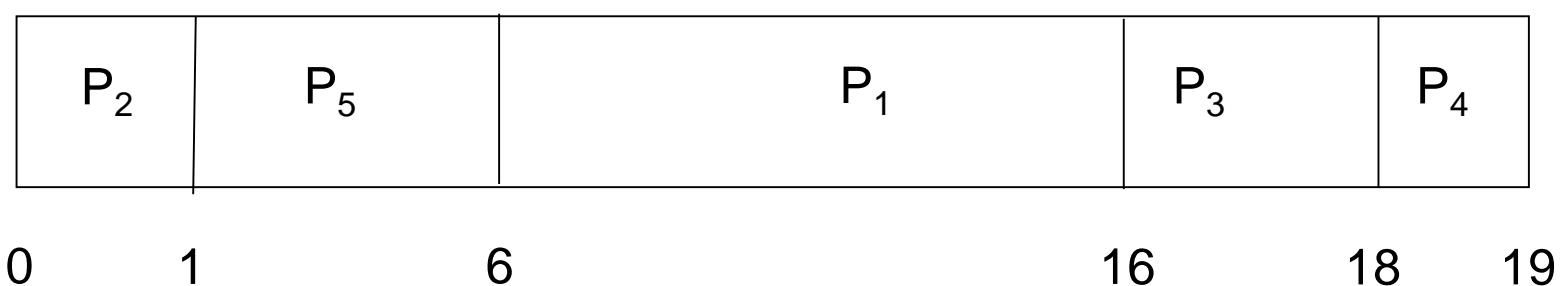
# Priority Scheduling

- ▶ A priority number (integer) is associated with each process
- ▶ The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Non-preemptive
- ▶ Problem  $\equiv$  **Starvation** – low priority processes may never execute
- ▶ Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- ▶ Priority scheduling Gantt Chart



- ▶ Average waiting time = 8.2 msec

# Example Priority

- ▶ Covered in class

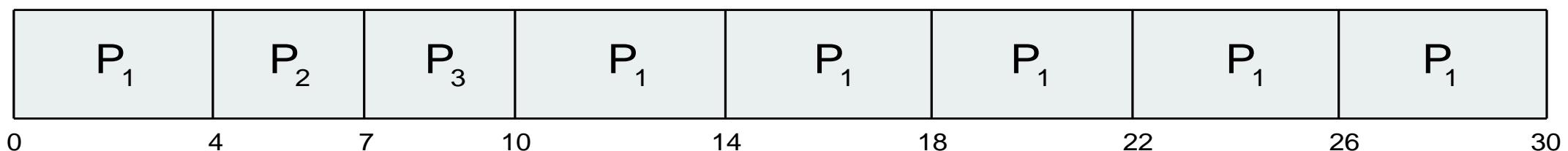
# Round Robin (RR)

- ▶ Each process gets a small unit of CPU time (**time quantum**  $q$ ), usually 10–100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- ▶ If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- ▶ Timer interrupts every quantum to schedule next process
- ▶ Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- ▶ The Gantt chart is:



- ▶ *Completion time* =  $P1=30; P2=7; P3=10$
- ▶  $TAT = P1 = (30-0); P2 = (7-4); P3 = (10-7)$
- ▶ *Avg. TAT* =

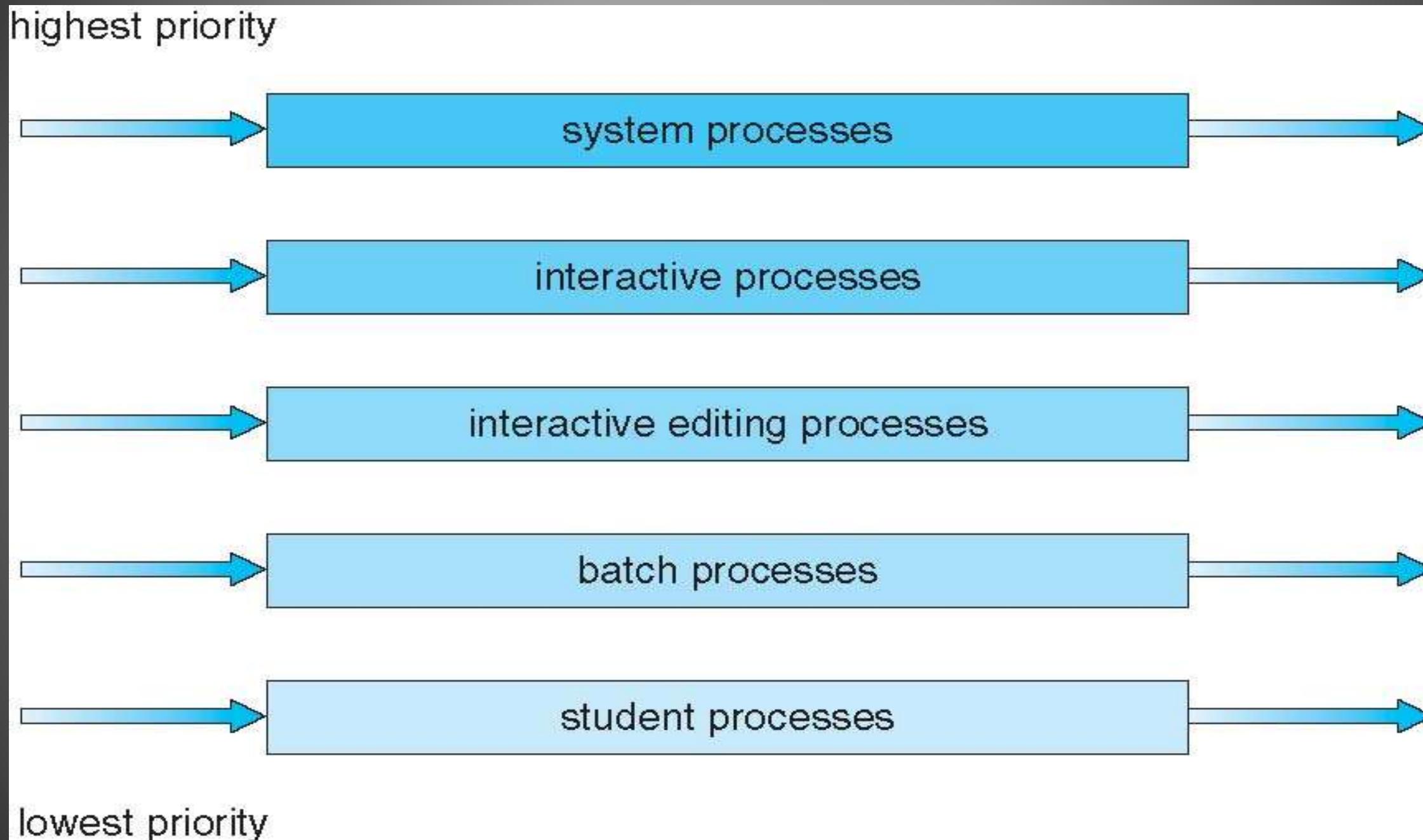
# Example RR

- ▶ Covered in class

# Multilevel Queue

- ▶ Process is assigned to one queue based on memory size, priority, process type.
- ▶ Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- ▶ Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- ▶ Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

# Multilevel Queue Scheduling

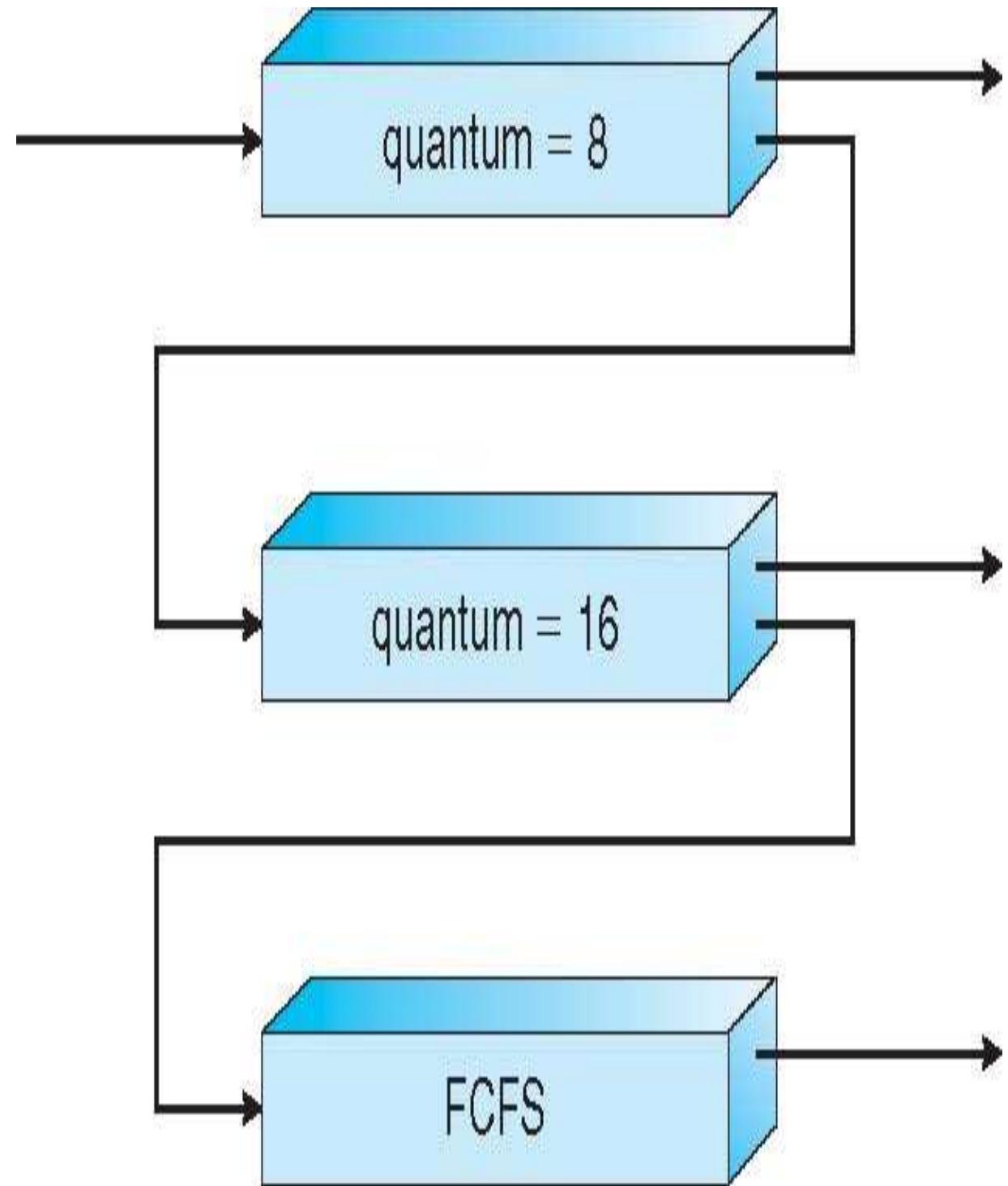


# Multilevel Feedback Queue

- ▶ Idea is to separate process according to CPU burst time
- ▶ If a process uses too much CPU time, it is moved to low priority
- ▶ If a process waits too long (aging) in low priority then it is moved to high priority
- ▶ Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

# Example of Multilevel Feedback Queue

- ▶ Three queues:
  - $Q_0$  – RR with time quantum 8 milliseconds
  - $Q_1$  – RR time quantum 16 milliseconds
  - $Q_2$  – FCFS
- ▶ Scheduling
  - A new job enters queue  $Q_0$  which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
  - At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue  $Q_2$



# Multiple-Processor Scheduling

- ▶ CPU scheduling more complex when multiple CPUs are available
- ▶ **Homogeneous processors** within a multiprocessor
- ▶ **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- ▶ **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

# Multiple-Processor Scheduling

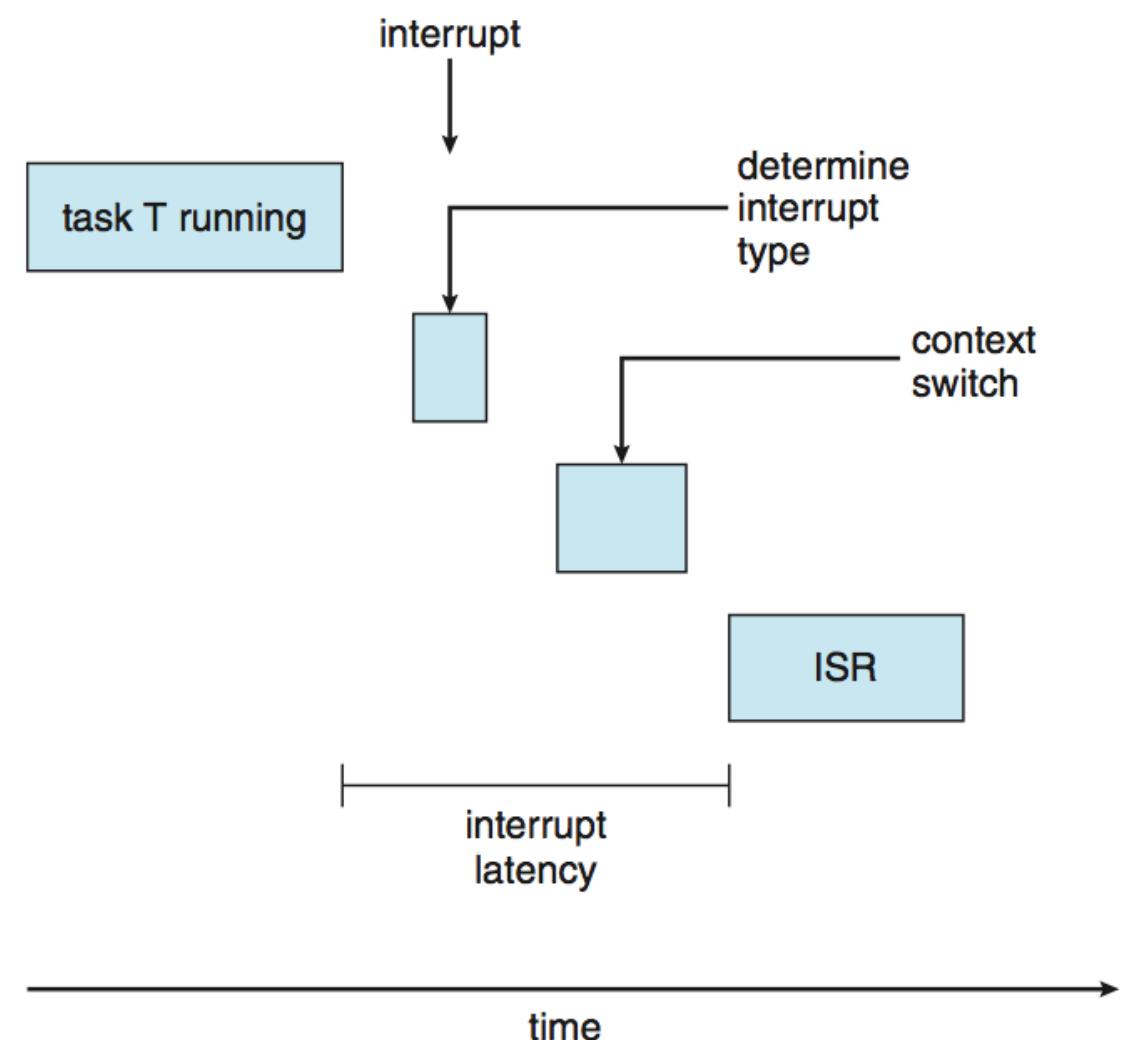
- ▶ **Processor affinity** – process has affinity for processor on which it is currently running
- ▶ **soft affinity:** When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so known as **soft affinity**.
- ▶ **hard affinity:** allowing a process to specify a subset of processors on which it may run.

# Multiple-Processor Scheduling – Load Balancing

- ▶ If SMP, need to keep all CPUs loaded for efficiency
- ▶ **Load balancing** attempts to keep workload evenly distributed
- ▶ **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- ▶ **Pull migration** – idle processors pulls waiting task from busy processor

# Real-Time CPU Scheduling

- ▶ **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- ▶ **Hard real-time systems** – task must be serviced by its deadline
- ▶ Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



# Threads

Course Instructor: Nausheen Shoaib

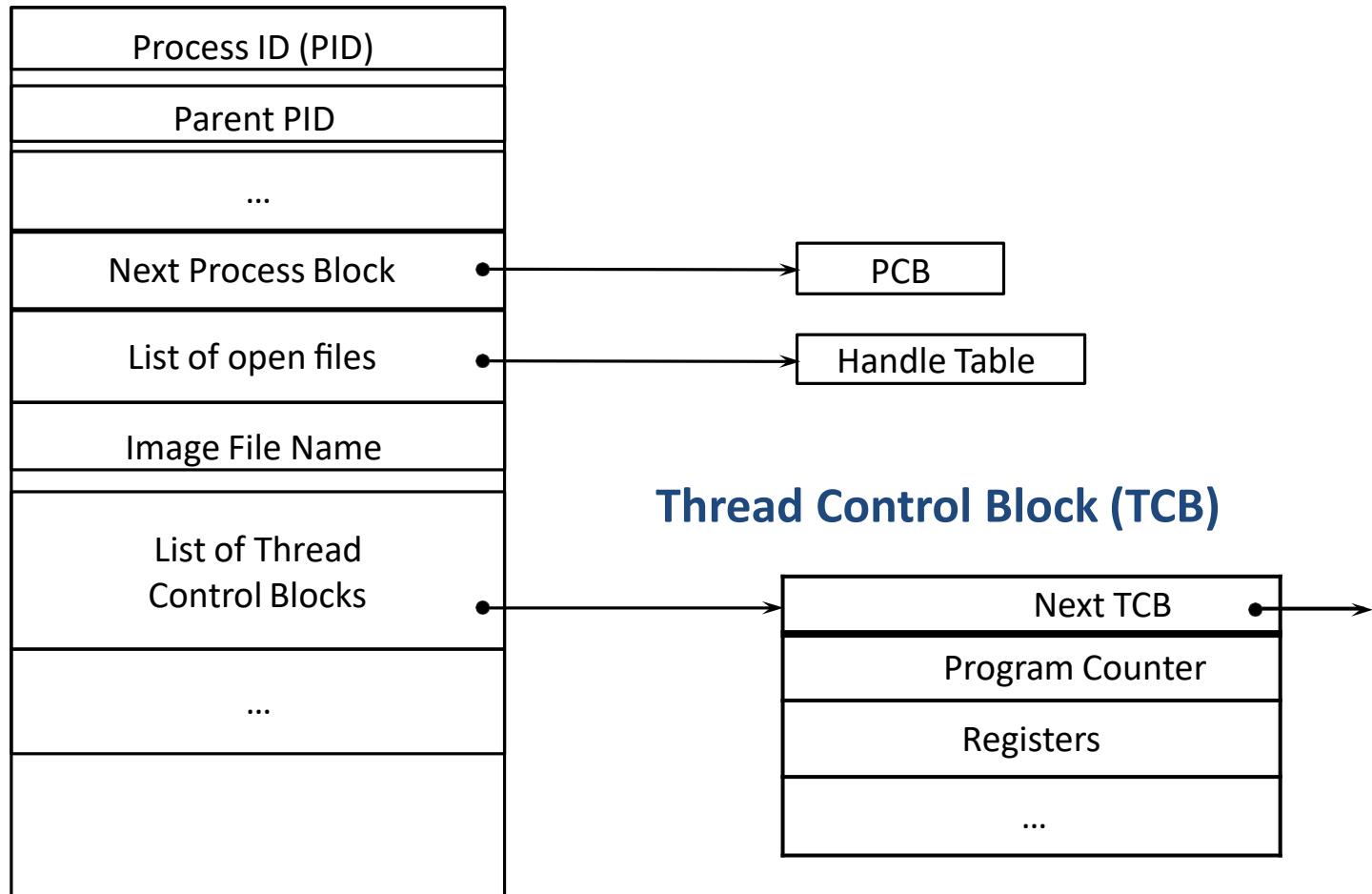
# Process Concept

- Classically, processes are executed programs that have ...
  - **Resource Ownership**
    - Process includes a virtual address space to hold the process image
    - Operating system prevents unwanted interference between processes
  - **Scheduling/Execution**
    - Process follows an execution path that may be interleaved with other processes
    - Process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the operating system
  - Today, the unit of dispatching is referred to as a **thread** or **lightweight process**
  - The unit of resource ownership remains the **process** or **task**

# Control Blocks

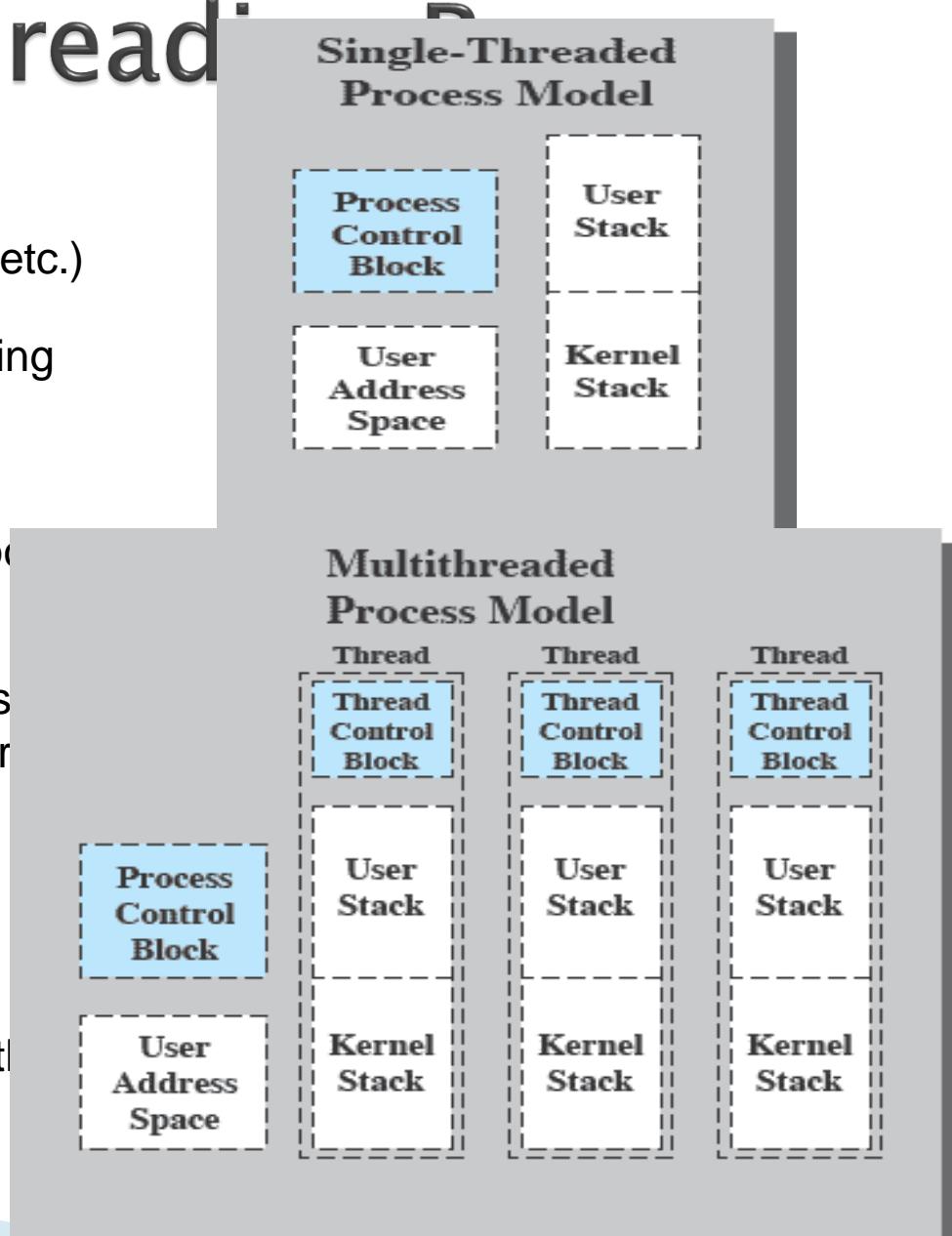
- Information associated with each process: **Process Control Block**
  - Memory management information
  - Accounting information
- Information associated with each thread: **Thread Control Block**
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Pending I/O information

# Control Blocks



# Single & Multithreaded Process Model

- Each **thread** has
  - An execution state (Running, Ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of the process (all threads of a process share this)
- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process



# Process Vs. Threads

S.N.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

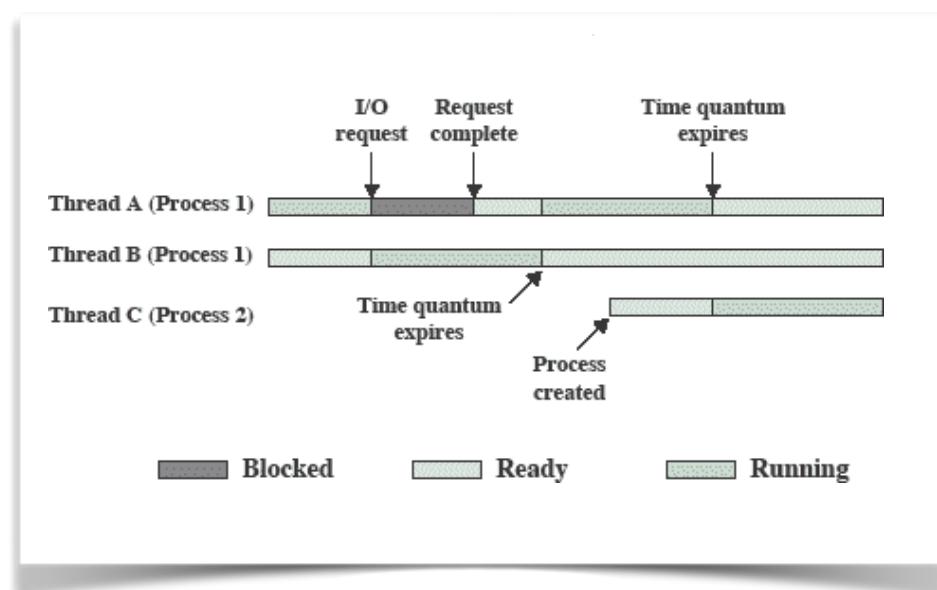
# Why Multithreading

- **Advantages**

- **Better responsiveness** - dedicated threads for handling user events
- **Simpler resource sharing** - all threads in a process share same address space
- Utilization of **multiple cores** for parallel execution
- Faster creation and termination of activities

- **Disadvantages**

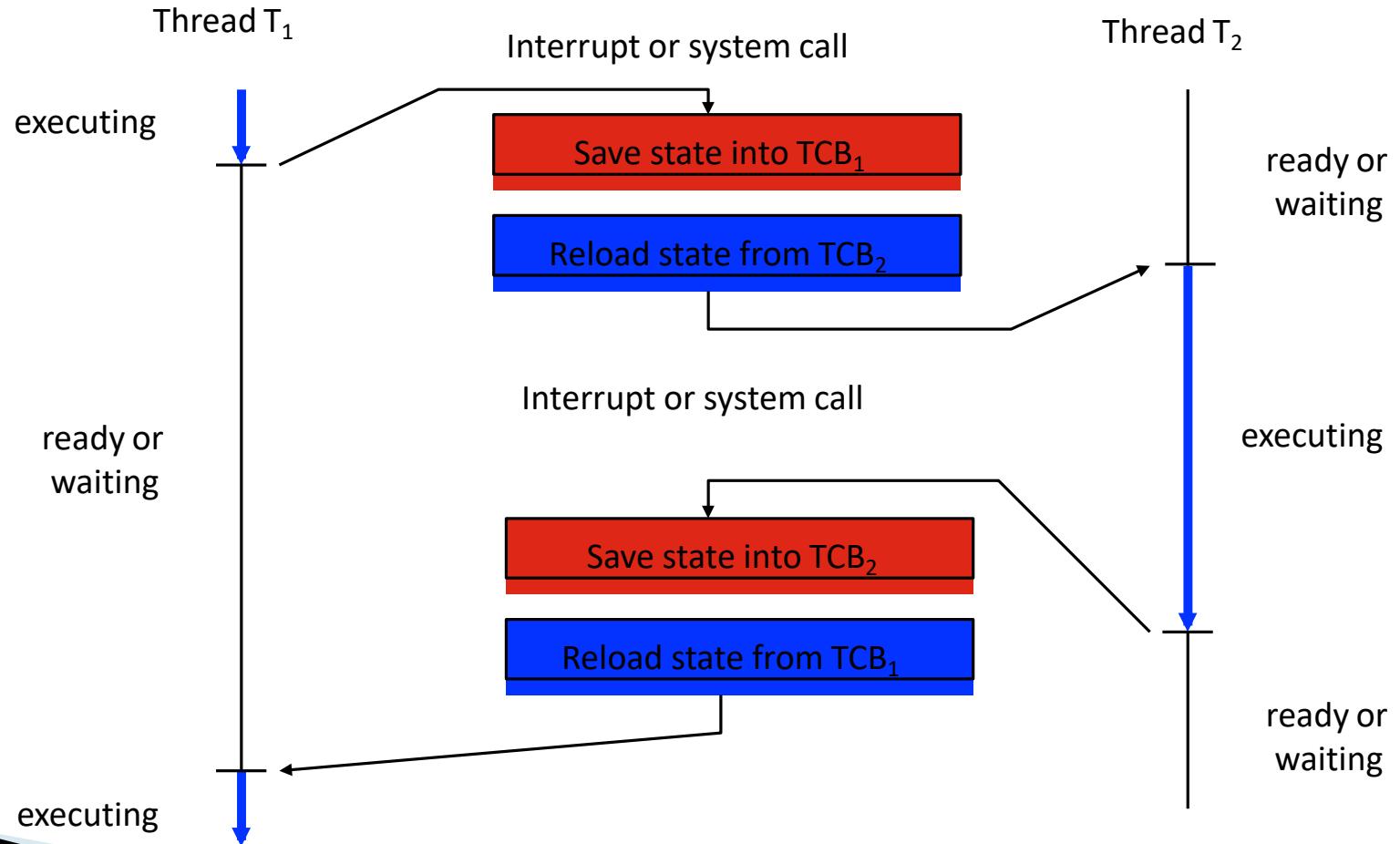
- Coordinated termination
- Signal and error handling
- Reentrant vs. non-reentrant system calls: reentrant if it can be interrupted in the middle of its execution, and then be safely called again



# Thread States

- The typical states for a thread are **running**, **ready**, **blocked**
- Typical **thread operations** associated with a change in thread state are:
  - **Spawn**: a thread within a process may spawn another thread
    - Provides instruction pointer and arguments for the new thread
    - New thread gets its own register context and stack space
  - **Block**: a thread needs to wait for an event
    - Saving its user registers, program counter, and stack pointers
  - **Unblock**: When the event for which a thread is blocked occurs
  - **Finish**: When a thread completes, its register context and stacks are deallocated.

# Thread Dispatching



# Threads

## Threads share....

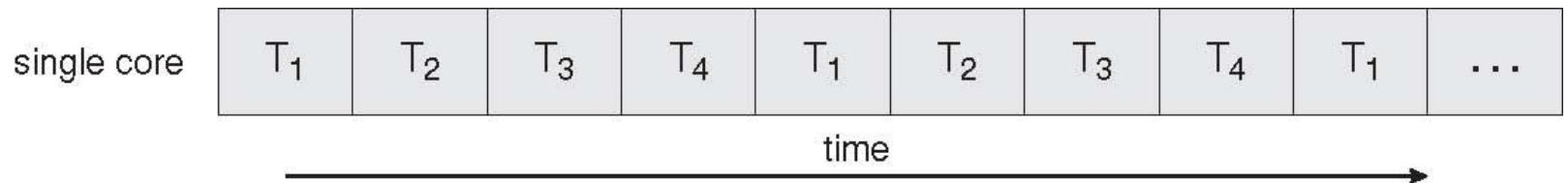
- ▶ Global memory
- ▶ Process ID and parent process ID
- ▶ Controlling terminal
- ▶ Process credentials (user )
- ▶ Open file information
- ▶ Timers
- ▶ .....

## Threads specific Attributes....

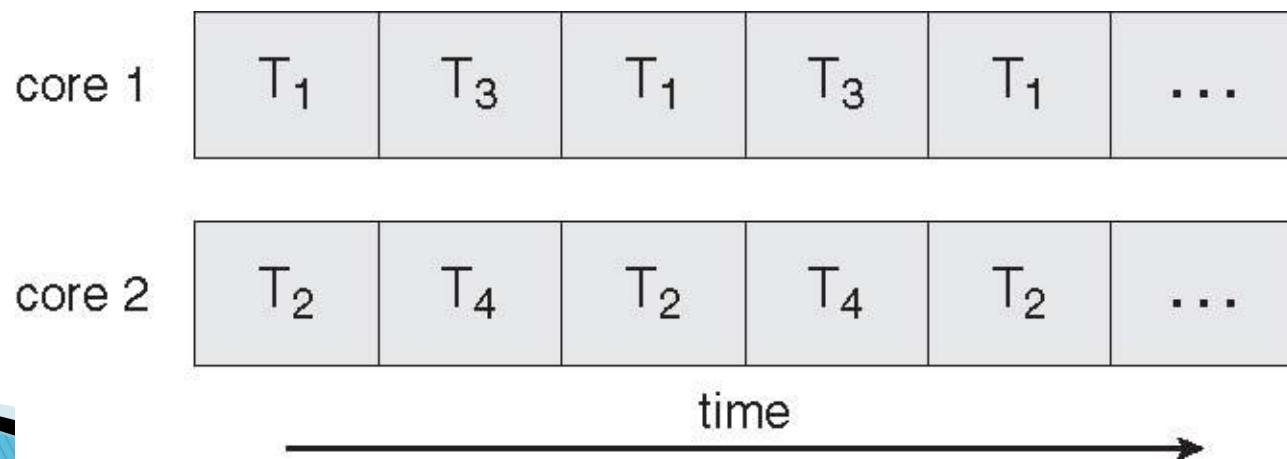
- Thread ID
- Thread specific data
- CPU affinity
- Stack (local variables and function call linkage information)
- .....

# Multicore Programming

## Concurrent Execution on a Single-core System



## Parallel Execution on a Multicore System



# Multicore Programming

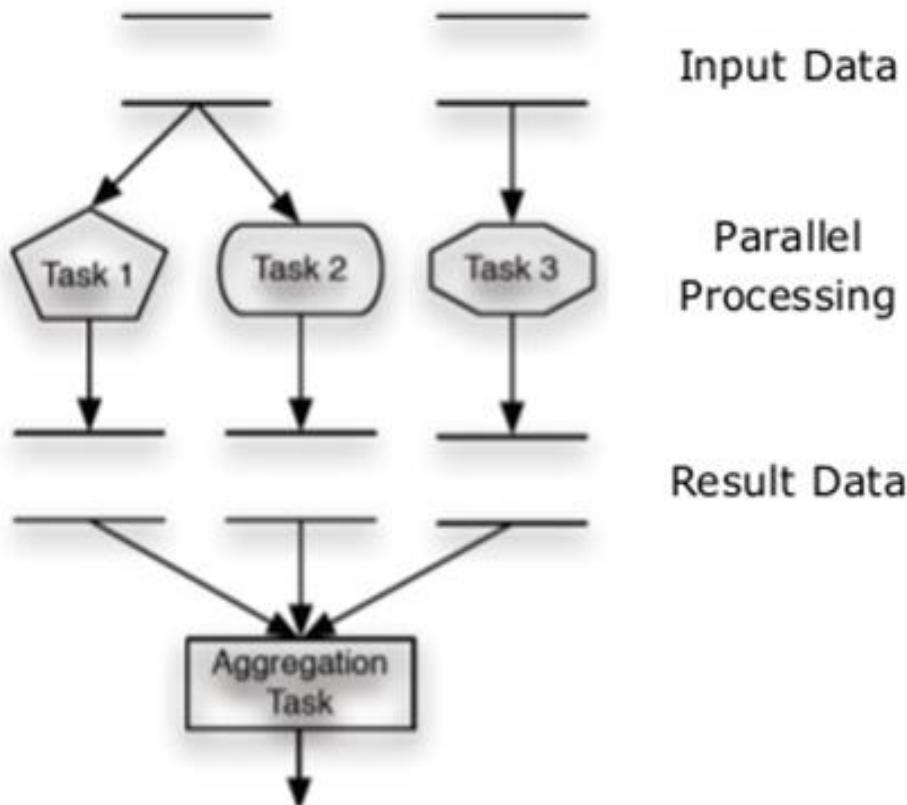
- ▶ Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
    - What tasks can be separated to run on different processors
  - **Balance**
    - Balance work on all processors
  - **Data splitting**
    - Separate data to run with the tasks
  - **Data dependency**
    - Watch for dependences between tasks
  - **Testing and debugging**
    - Harder!!!!

# Types of Parallelism

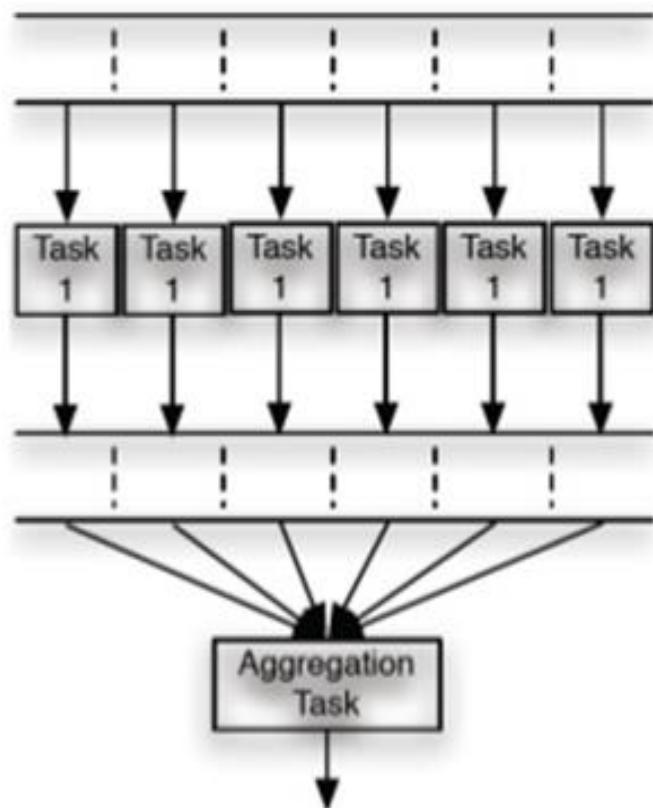
- ▶ **Data Parallelism:** focus on distributing data across different parallel computing nodes
- ▶ **Task Parallelism:** focus on distributing execution processes(threads) across different parallel computing nodes

# Types of Parallelism

Task Parallelism



Data Parallelism



# Data vs. Task Parallelism

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum <u>load balance</u> on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

# Amdahl's Law

- gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved

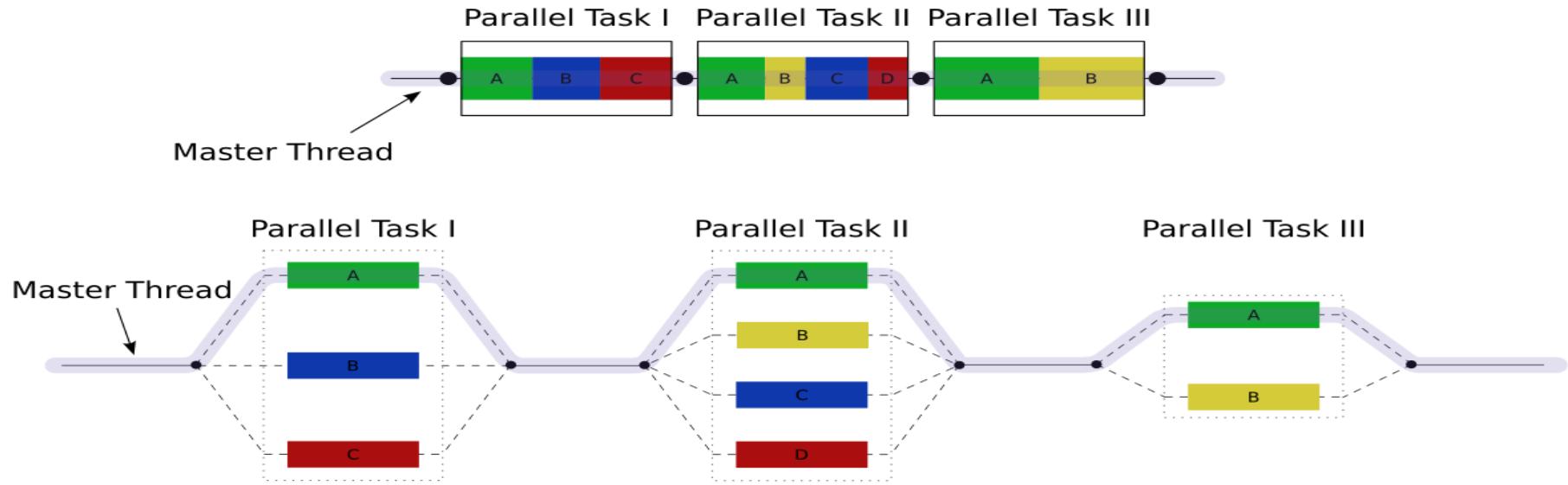
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Where S = portion of program executed serially  
N = Processing Cores

# Amdahl's Law Example

- ▶ we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores?
- ▶  $S=25\%=0.25$ ,  $N= 2$
- ▶ If we add two additional cores , calculate speedup?

# Fork – Join Model



```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve part
        join all subtasks spawned in previous loop
        combine results from subtasks
```

# Multithreading Models

- Support provided at either

- User level -> **user threads**

- Supported above the kernel and managed without kernel support

- Kernel level -> **kernel threads**

- Supported and managed directly by the operating system

What is the relationship between user and kernel threads?

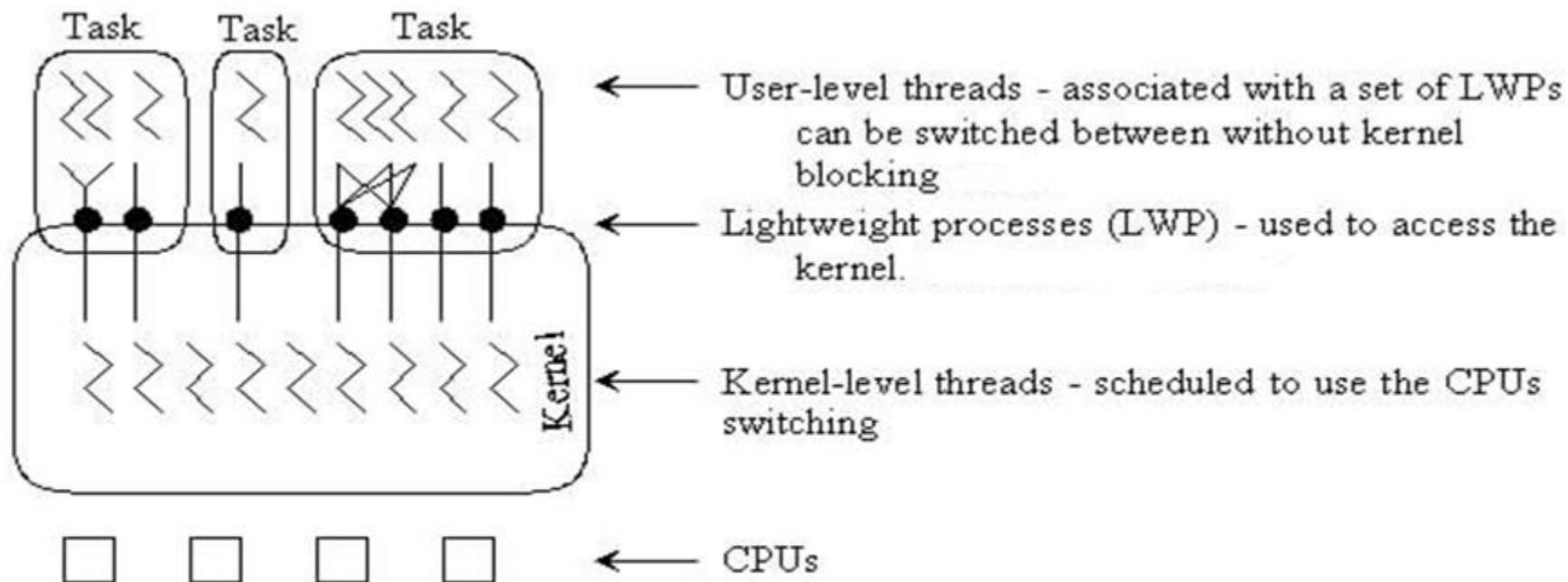
# User Threads

- ▶ Thread management done by user-level threads library
- ▶ Three primary thread libraries:
  - POSIX [Pthreads](#)
  - Win32 threads
  - Java threads

# Kernel Threads

- ▶ Supported by the Kernel
- ▶ Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# User vs. Kernel Thread



# Multithreading Models

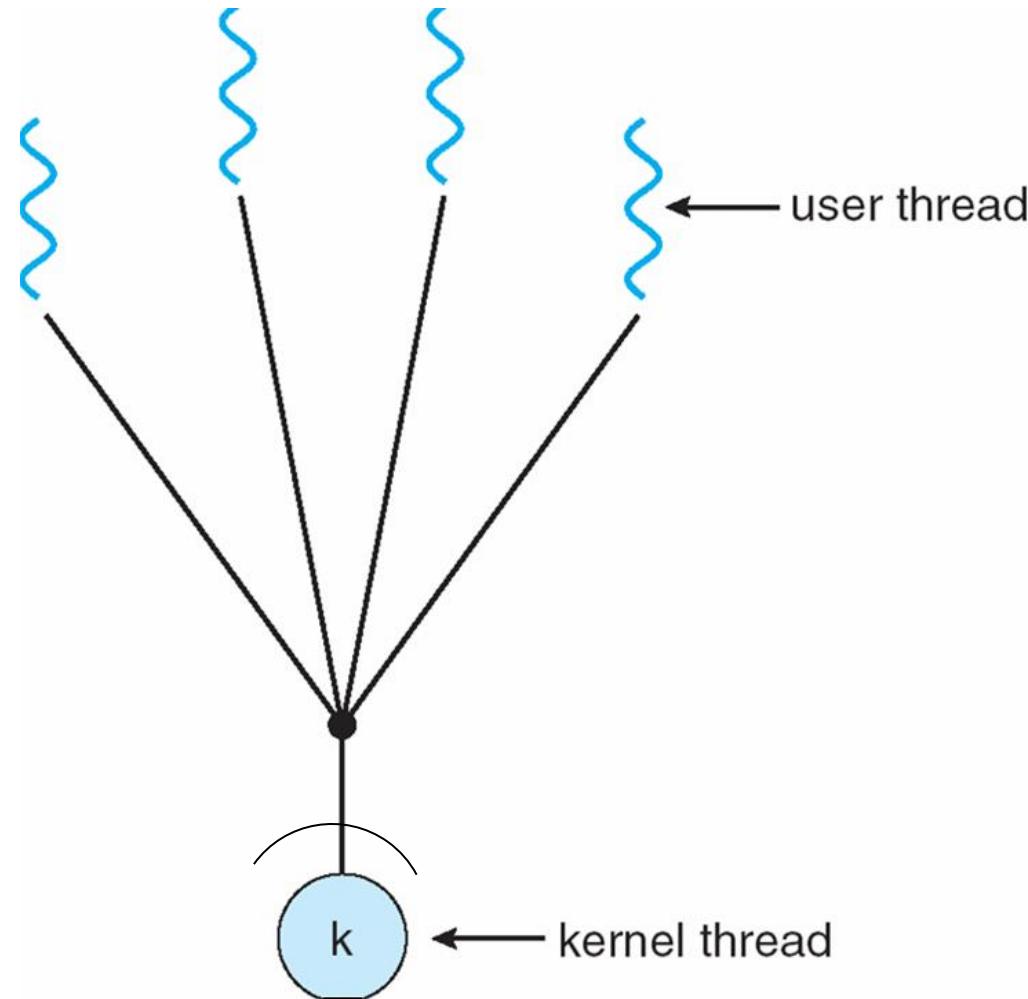
User Thread – to - Kernel Thread

- ▶ Many-to-One
- ▶ One-to-One
- ▶ Many-to-Many

# Many-to-One

Many user-level threads mapped to single kernel thread

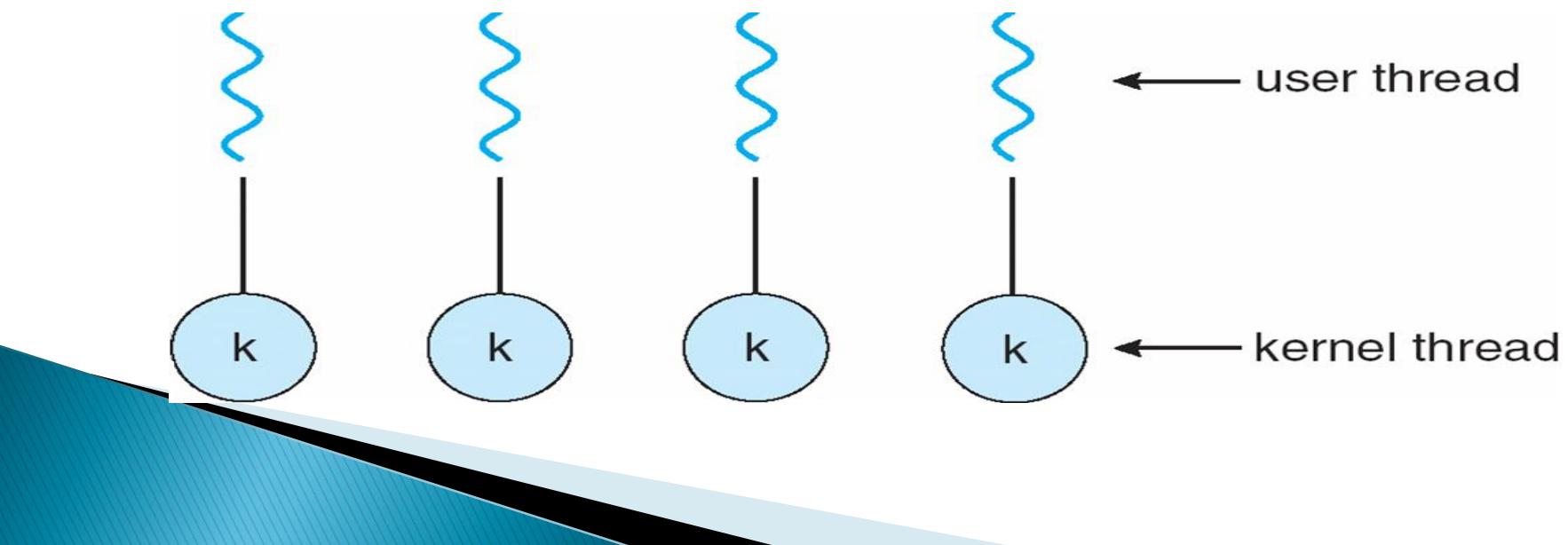
- ▶ Only one thread can access the kernel at a time,
- ▶ multiple threads are unable to run in parallel on multicore systems.
- ▶ the entire process will block if a thread makes a blocking system call



# One-to-One

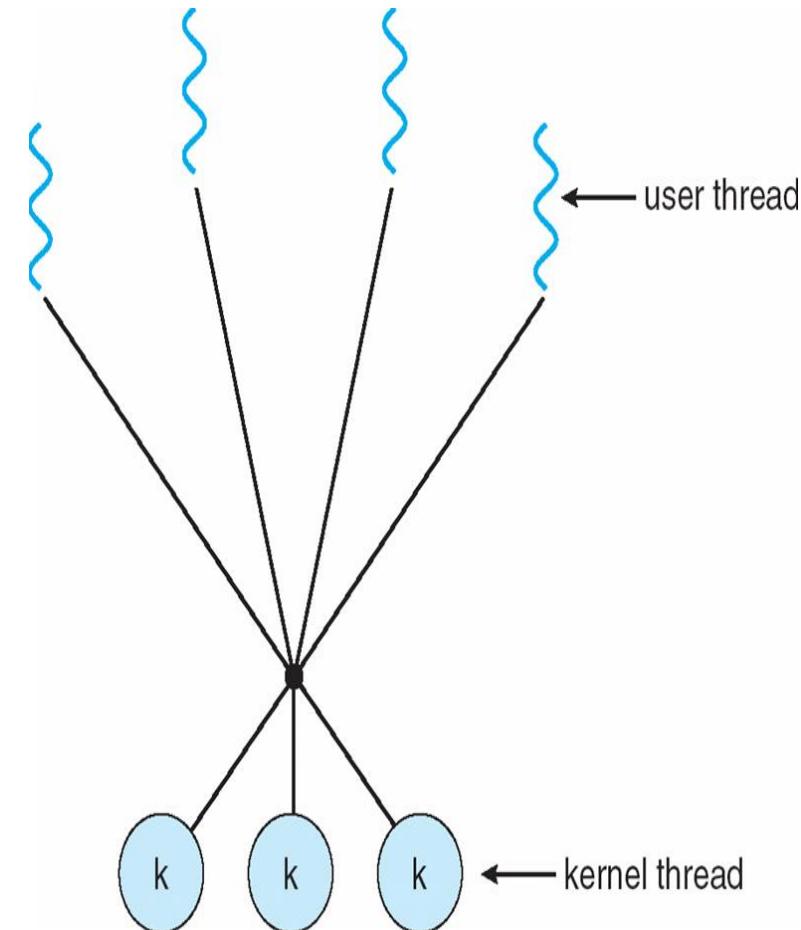
Each user-level thread maps to kernel thread

- ▶ more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- ▶ Allows multiple threads to run in parallel on multiprocessors.
- ▶ drawback is, creating a user thread requires creating the corresponding kernel thread



# Many-to-Many Model

- ▶ multiplexes many user-level threads to a smaller or equal number of kernel threads
- ▶ developers can create as many user threads as necessary, and the corresponding
- ▶ kernel threads can run in parallel on a multiprocessor.
- ▶ When thread performs a blocking system call, the kernel can schedule another thread for execution.



# 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

# POSIX Compilation on Linux

On Linux, programs that use the Pthreads API must be compiled with

*-pthread* or *-lpthread*

```
gcc -o thread -lpthread thread.c
```

# POSIX: Thread Creation

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start)(void *), void *arg);
```

- ❖ *\*thread* Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.
- ❖ *attr* Is the thread attribute object specifying the attributes for the thread that is being created. If *attr* is NULL, the thread is created with default attributes.
- ❖ *start* Is the main function for the thread; the thread begins executing user code at this address.
- ❖ *arg* Is the argument passed to *start*.

# POSIX: Thread ID

```
#include <pthread.h>

pthread_t pthread_self()
```

**returns :** ID of current (this) thread

# POSIX: Wait for Thread Completion

```
#include <pthread.h>
```

```
pthread_join (thread, NULL)
```

**returns :** 0 on success, some error code on failure.

# POSIX: Thread Termination

```
#include <pthread.h>

Void pthread_exit (return_value)
```

Threads terminate in one of the following ways:

- The thread's start functions performs a return specifying a return value for the thread.
- Thread receives a request asking it to terminate using `pthread_cancel()`
- Thread initiates termination `pthread_exit()`
- Main process terminates

```
▶ int main()
▶ {
▶     pthread_t thread1, thread2; /* thread variables */
▶     thdata data1, data2;      /* structs to be passed to threads */
▶
▶     /* initialize data to pass to thread 1 */
▶     data1.thread_no = 1;
▶     strcpy(data1.message, "Hello!");
▶
▶     /* initialize data to pass to thread 2 */
▶     data2.thread_no = 2;
▶     strcpy(data2.message, "Hi!");
▶
▶     /* create threads 1 and 2 */
▶     pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &data1);
▶     pthread_create (&thread2, NULL, (void *) &print_message_function, (void *) &data2);
▶
▶     /* Main block now waits for both threads to terminate, before it exits
▶        If main block exits, both threads exit, even if the threads have not
▶        finished their work */
▶     pthread_join(thread1, NULL);
▶     pthread_join(thread2, NULL);
▶
▶     exit(0);
▶ }
```

Example code but not complete

# Signal Handling

- ▶ Signals are used in UNIX systems to notify a process that a particular event has occurred
- ▶ A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- ▶ Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

# Thread Pools

- ▶ Create a number of threads in a pool where they await work
- ▶ Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

# Thread Scheduling

- ▶ In systems that support user and kernel-level threads, kernel-level threads are scheduled by the OS
  - Kernel-level threads instead of processes are scheduled
- User-level threads are managed by a thread library
  - To run on the CPU, the user-level thread must be mapped on an associated kernel-level thread

# Thread Scheduling

- ▶ **Contention Scope:**
- ▶ On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process contention scope (PCS)**,
- ▶ (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.

# What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct multithreaded, shared memory parallelism
- Three main API components
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable & Standardized
  - API exist both C/C++ and Fortan 90/77
  - Multi platform Support (Unix, Linux etc.)

# OpenMP Compilation

- GCC

```
bash: $ gcc -fopenmp hi-omp.c -o hi-omp.x
```

# OpenMP Directives

```
#pragma omp parallel default(shared) private(beta,pi)
```

## **#pragma omp barrier**

Each thread waits at the barrier until all threads have reached it.

## **#pragma omp for**

Distributes the iterations of a loop over multiple threads

# OpenMP threads

- ▶ **Thread Creation:**
- ▶ `omp_get_num_threads()`

Returns number of threads in parallel region

Returns 1 if called outside parallel region

- ▶ **Thread Id:**
- ▶ `omp_get_thread_num()`
- ▶ Returns id of thread in team Value between  $[0, n-1]$  // where  $n = \# \text{threads}$  Master thread always has id 0

# Open MP Example

```
#include "omp.h" ← OpenMP include file
void main()
{
    #pragma omp parallel ← Parallel region with default
                           number of threads
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    } ← End of the Parallel region
}
```

## Sample Output:

hello(1) hello(0) wor  
world(0)  
hello (3) hello(2) wor  
world(2)

Runtime library function to  
return a thread ID.

# **Memory Management Strategies**

**Course Instructor: Nausheen Shoaib**

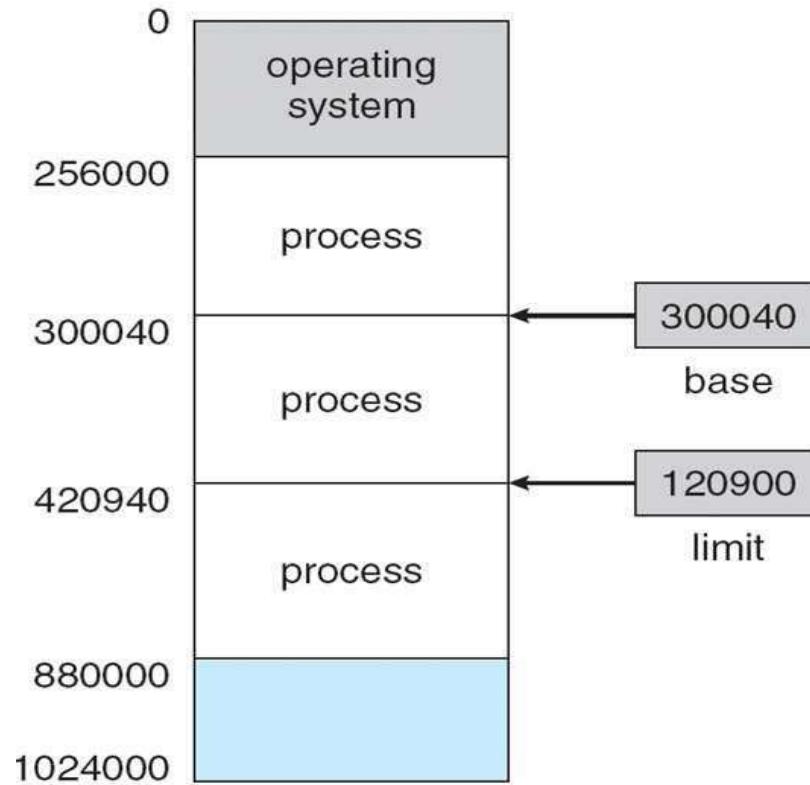
# Background

---

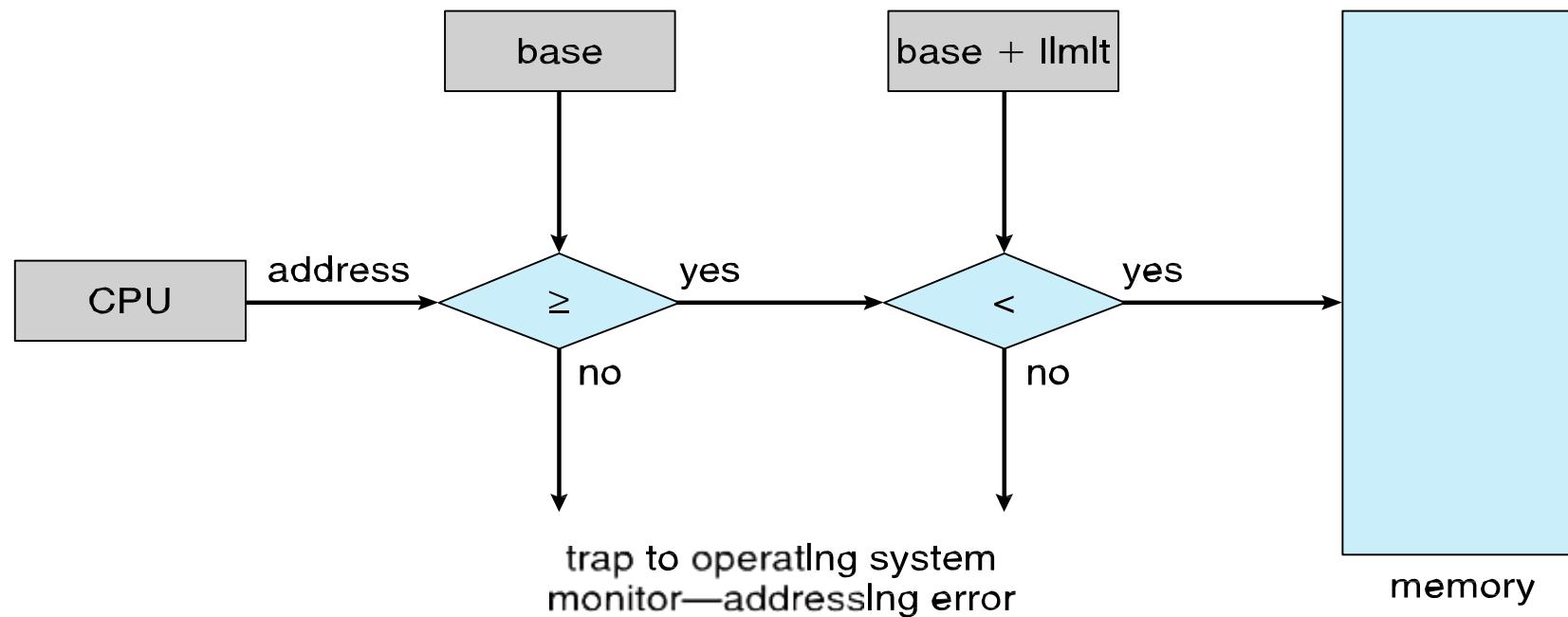
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage, CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

# Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



# Hardware Address Protection with Base and Limit Registers



# Address Binding

---

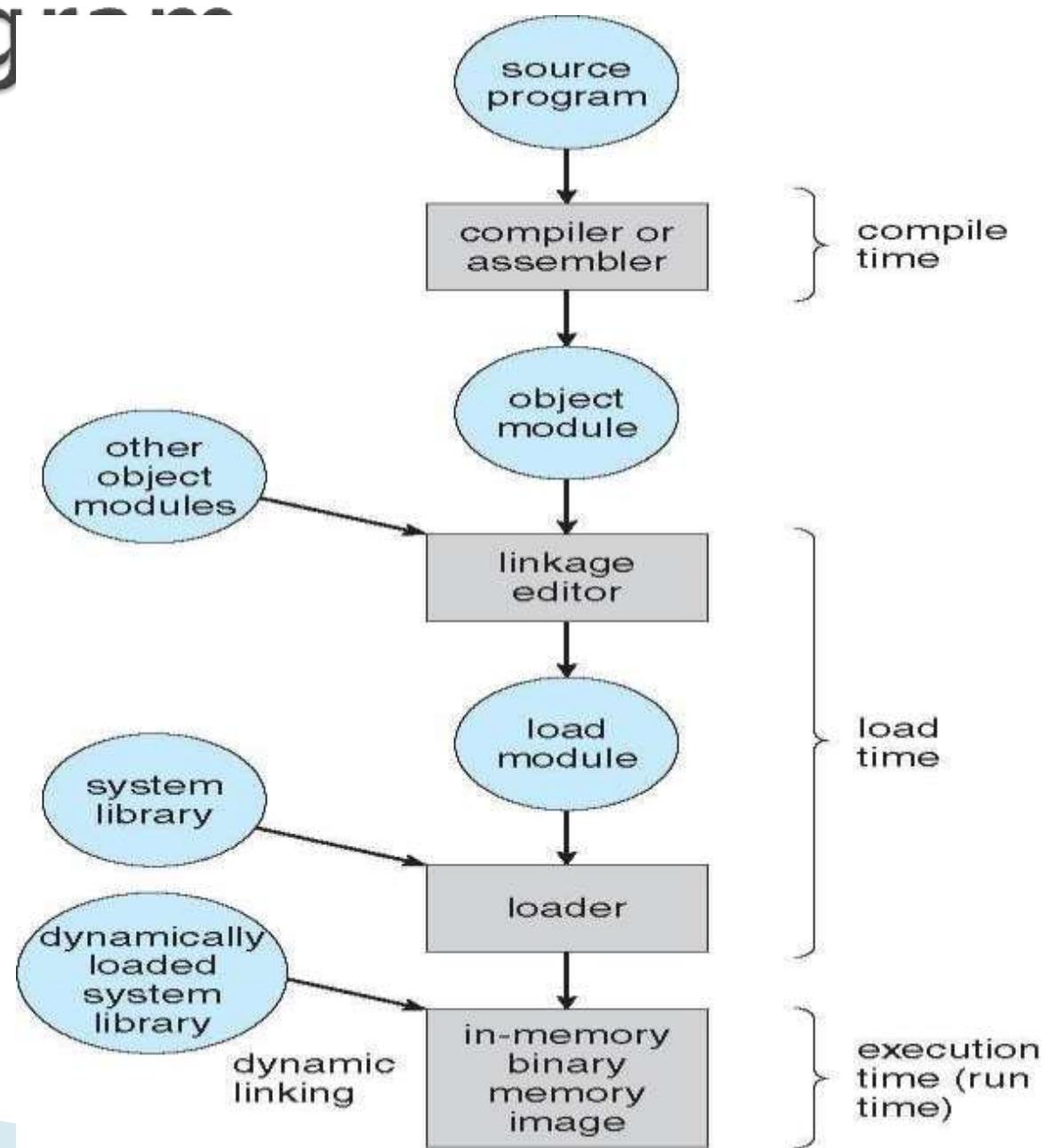
- Programs on disk, ready to be brought into memory to execute from an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
  - Linker or loader will bind relocatable addresses to absolute addresses
  - Each binding maps one address space to another

# Binding of Instructions and Data to Memory

---

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

# Multistep Processing of a User Program



# Logical vs. Physical Address Space

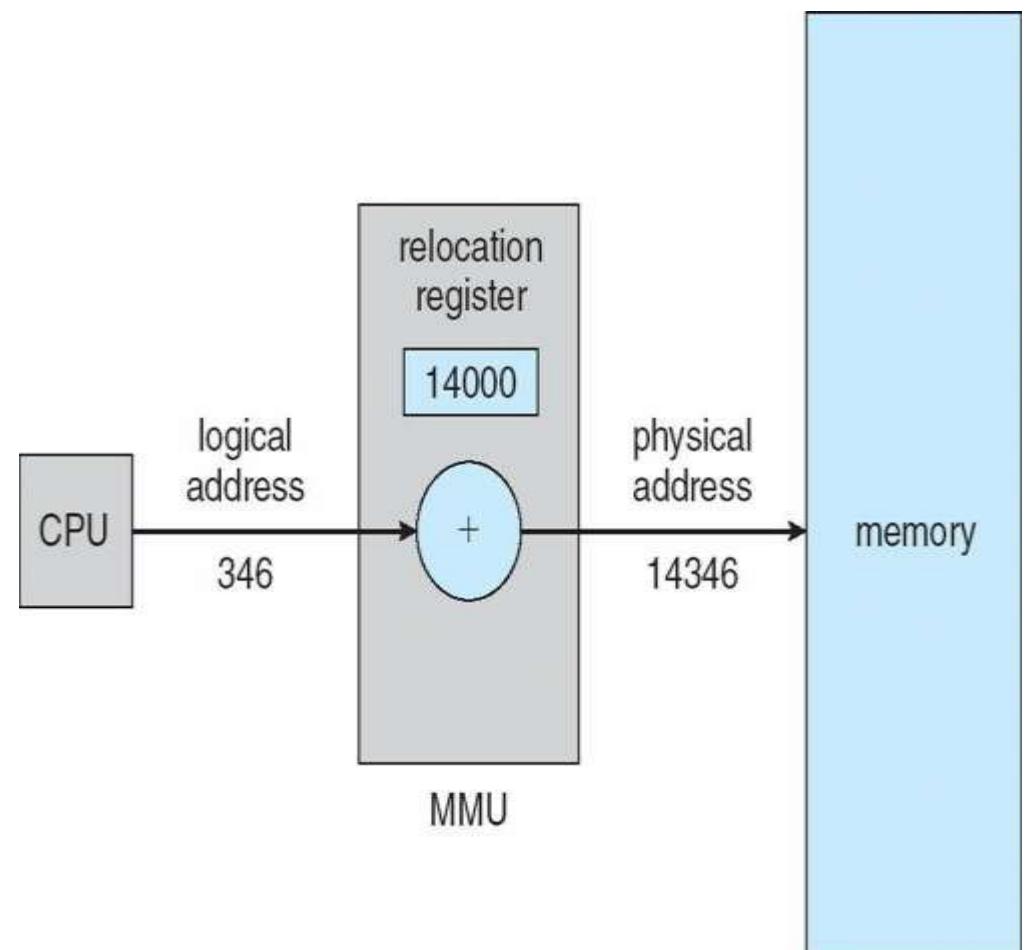
- The concept of a logical address space that is bound to a separate
  - 1)**Logical address** – generated by the CPU; also referred to as **virtual address**
  - 2)**Physical address** – address seen by the memory unit
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme**
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

## Memory-Management Unit (MMU)

- MMU is a hardware device that at run time maps virtual address to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real*/physical addresses
  - Execution-time binding occurs when reference is made to location in memory

# Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



# Static & Dynamic Linking

---

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** -linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries

# Swapping [1 / 2]

---

A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution

**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

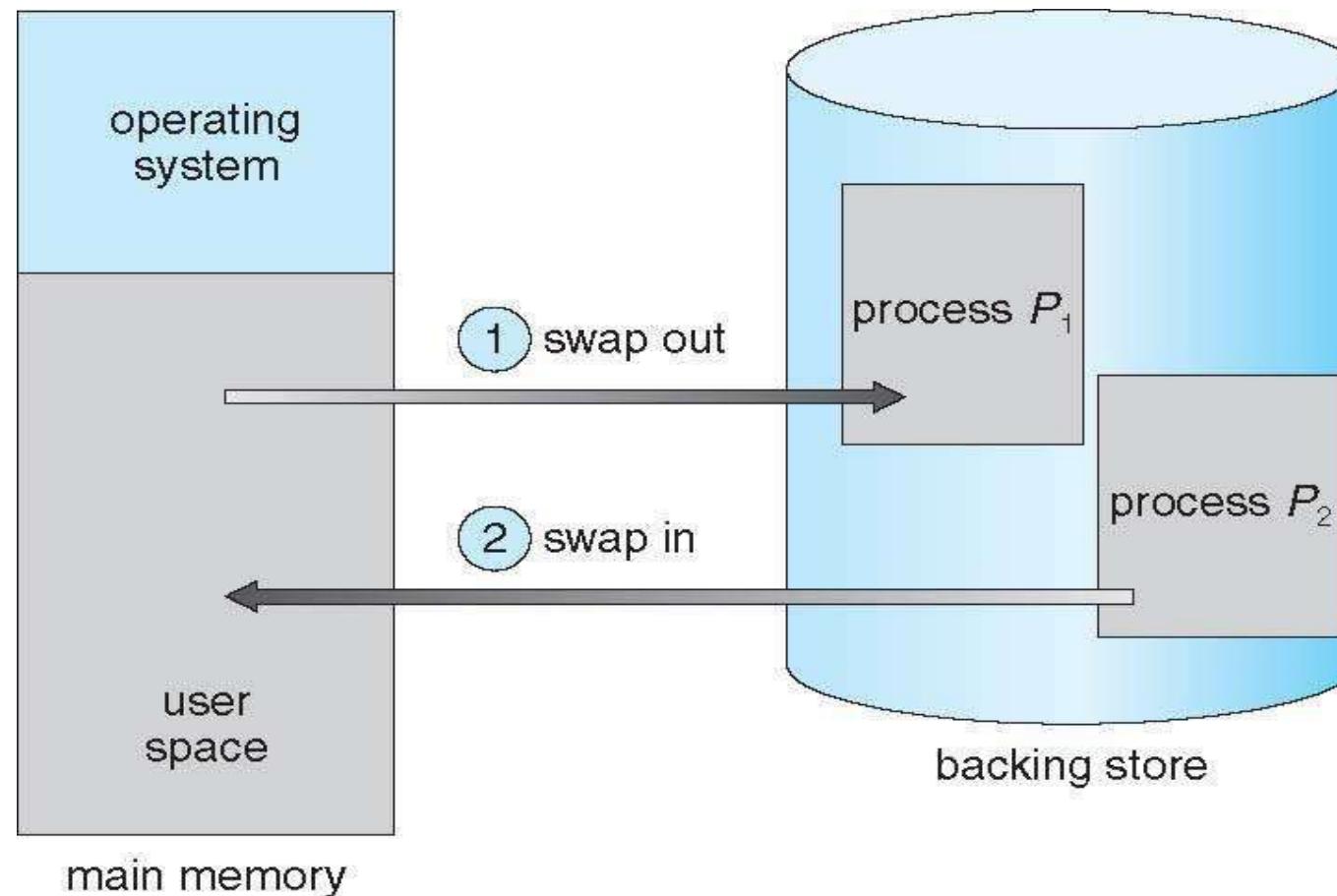
**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

# Swapping [2 / 2]

- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

# Schematic View of Standard Swapping



# Context Switch Time including Swapping

## [1 / 2]

---

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- Swap in Time= size of (process) / transfer rate
- 100MB process swapping to hard disk with transfer rate of 50MB/sec = 2 sec or 2000msec.
  - Swap out time = 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping time (Swap In +out) =4000ms (4 seconds)

# Context Switch Time including Swapping [2/2]

## Constraints of Swapping:

- Swap time can be reduced if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Other constraints are:
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - always transfer I/O to kernel space, then to I/O device
    - ▶ Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - ▶ Swap only when free memory extremely low

# Swapping on Mobile Systems

- ▶ Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- ▶ Instead use other methods to free memory if low
  - iOS *asks* apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restartf

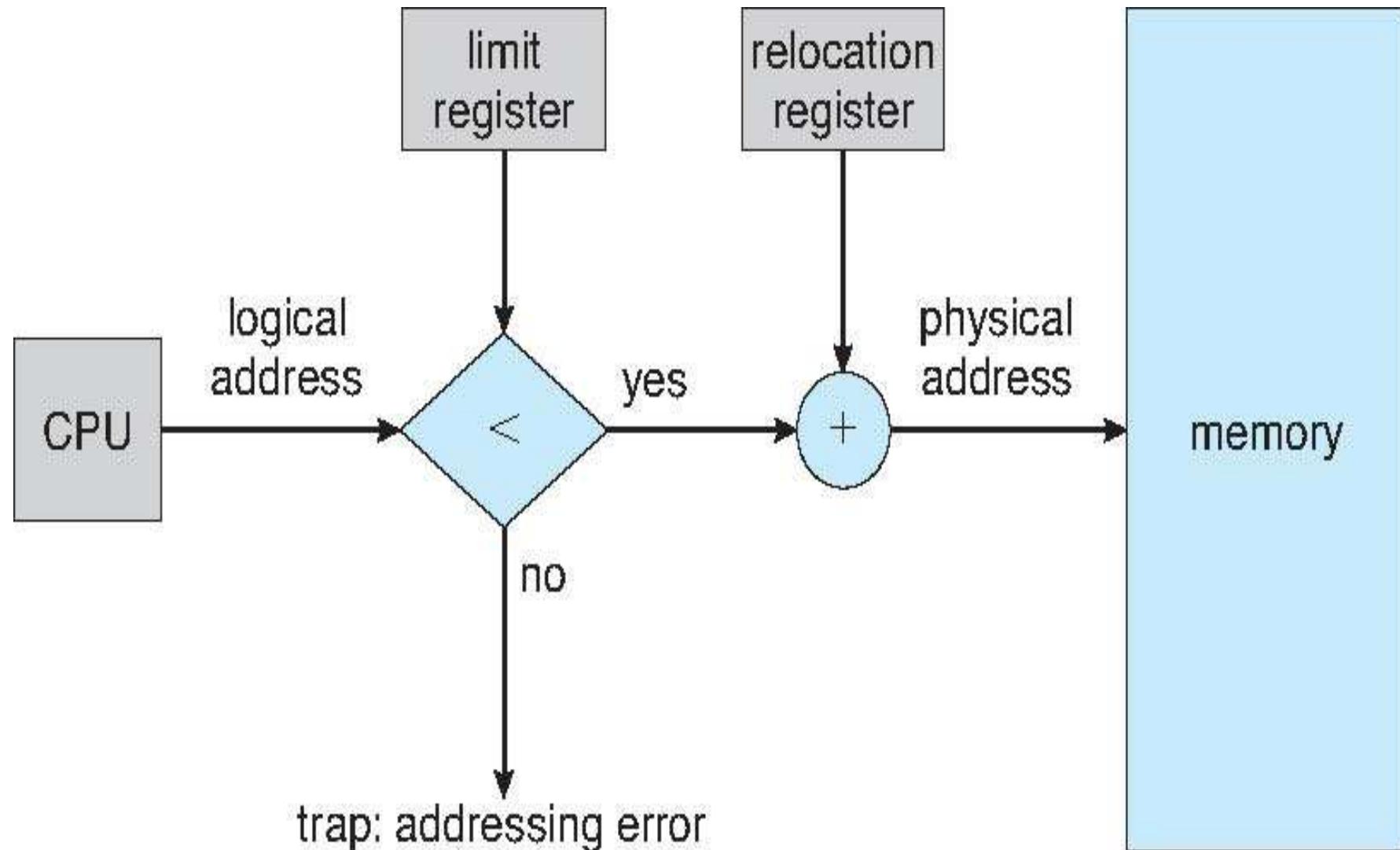
# Contiguous Allocation

- ▶ Main memory must support both OS and user processes
- ▶ Limited resource, must allocate efficiently
- ▶ Contiguous allocation is one early method
- ▶ Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

# Contiguous Allocation (Cont.)

- ▶ Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

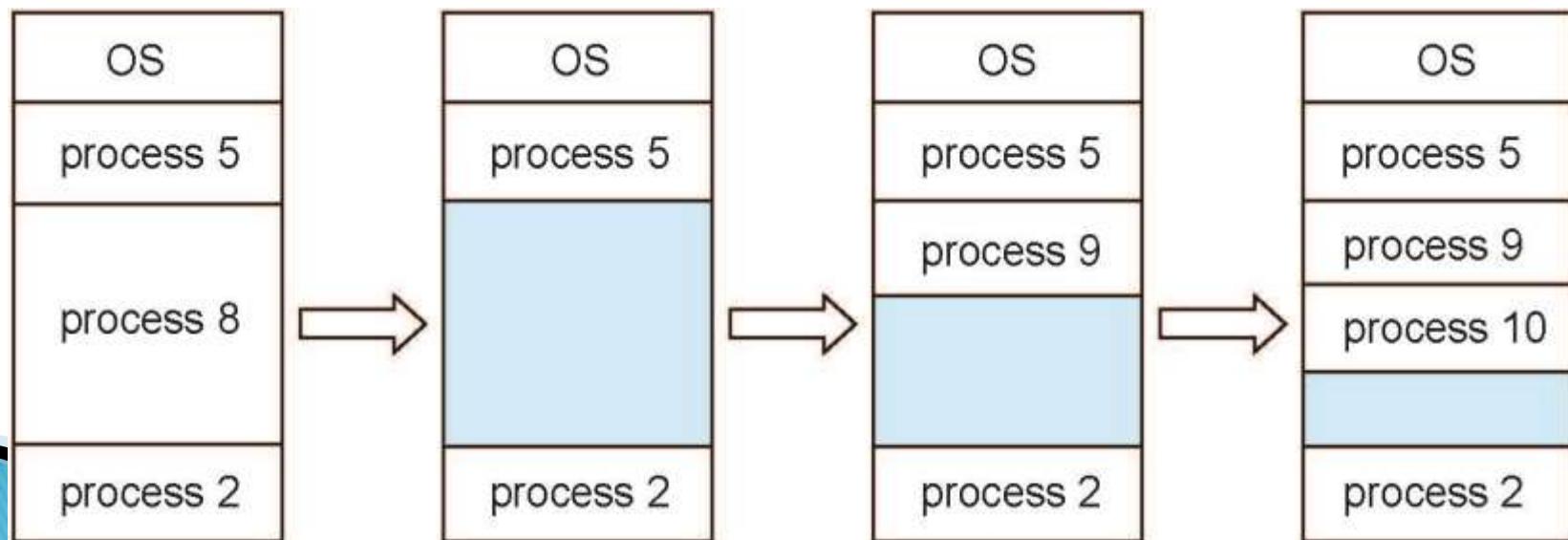
# Hardware Support for Relocation and Limit Registers



# Multiple-partition allocation

## ▶ Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    - a) allocated partitions
    - b) free partitions (hole)



# Dynamic Storage-Allocation Problem

How to satisfy a request of size  $n$  from a list of free holes?

- ▶ **First-fit**: Allocate the *first* hole that is big enough
- ▶ **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- ▶ **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

# Fragmentation

- ▶ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- ▶ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- ▶ First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

# Fragmentation (Cont.)

- ▶ Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- ▶ Now consider that backing store has same fragmentation problems

# Paging

- ▶ Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- ▶ Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- ▶ Divide logical memory into blocks of same size called **pages**
- ▶ Keep track of all free frames
- ▶ To run a program of size  $N$  pages, need to find  $N$  free frames and load program
- ▶ Set up a **page table** to translate logical to physical addresses
- ▶ Backing store likewise split into pages
- ▶ Still have Internal fragmentation

# Paging (Cont.)

- ▶ Calculating internal fragmentation

*Given:*

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes

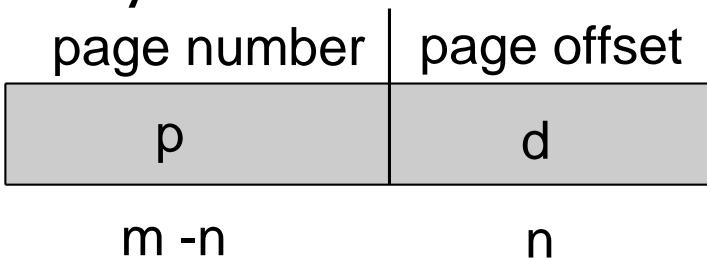
*Calculate:*

- Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size

- ▶ Process view and physical memory now very different
- ▶ By implementation process can only access its own memory

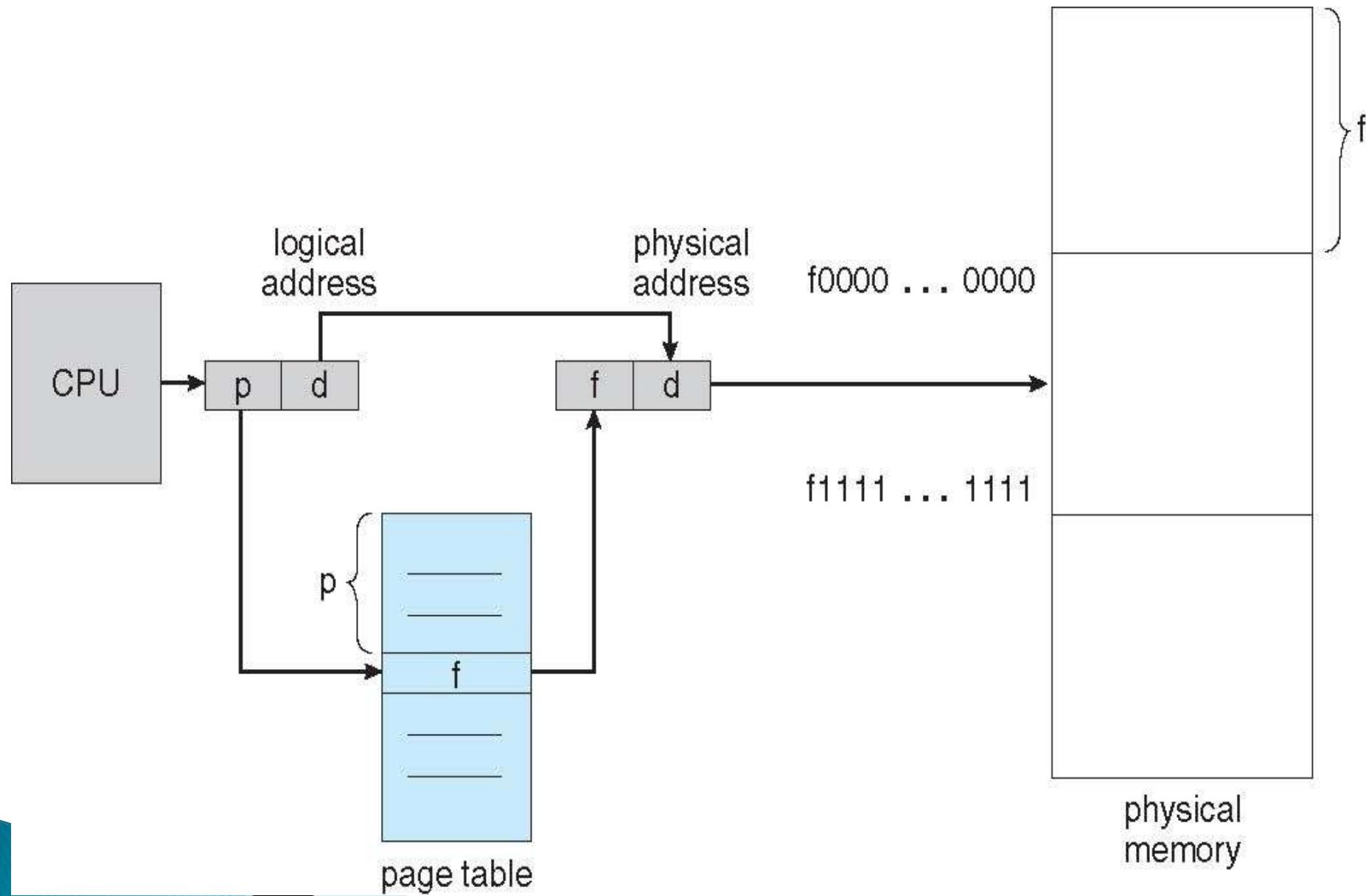
# Address Translation Scheme

- ▶ Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

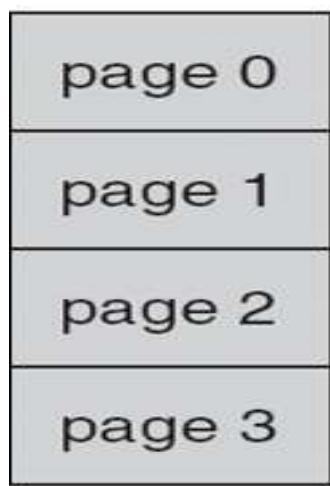


- For given logical address space  $2^m$  and page size  $2^n$

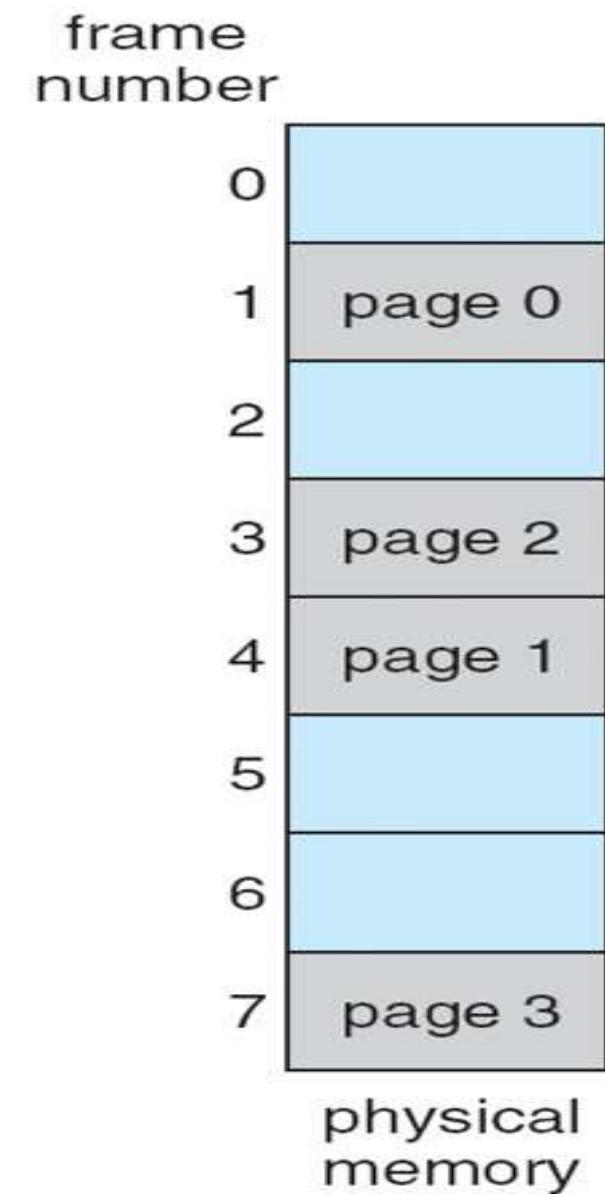
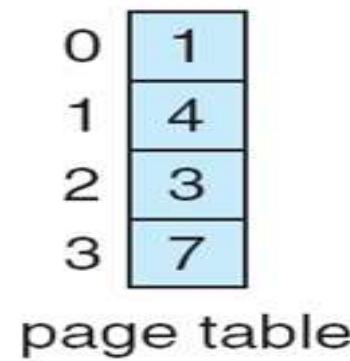
# Paging Hardware



# Paging Model of Logical and Physical Memory



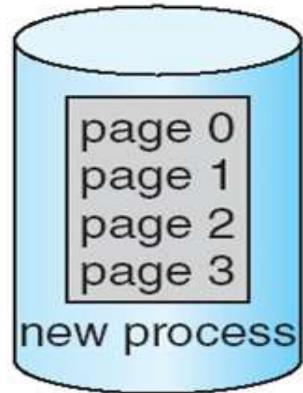
logical  
memory



# Free Frames

free-frame list

14  
13  
18  
20  
15

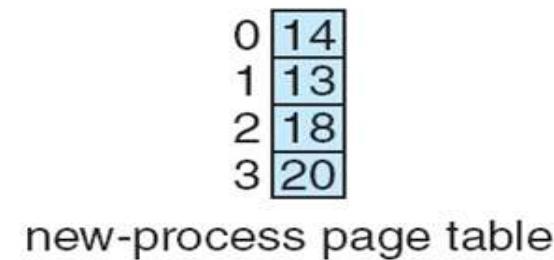
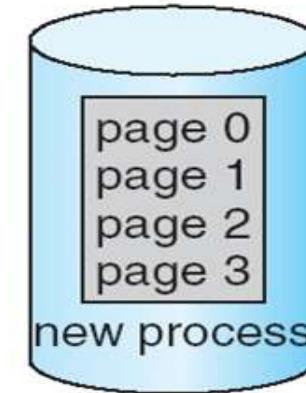


(a)

Before allocation

free-frame list

15



(b)

After allocation



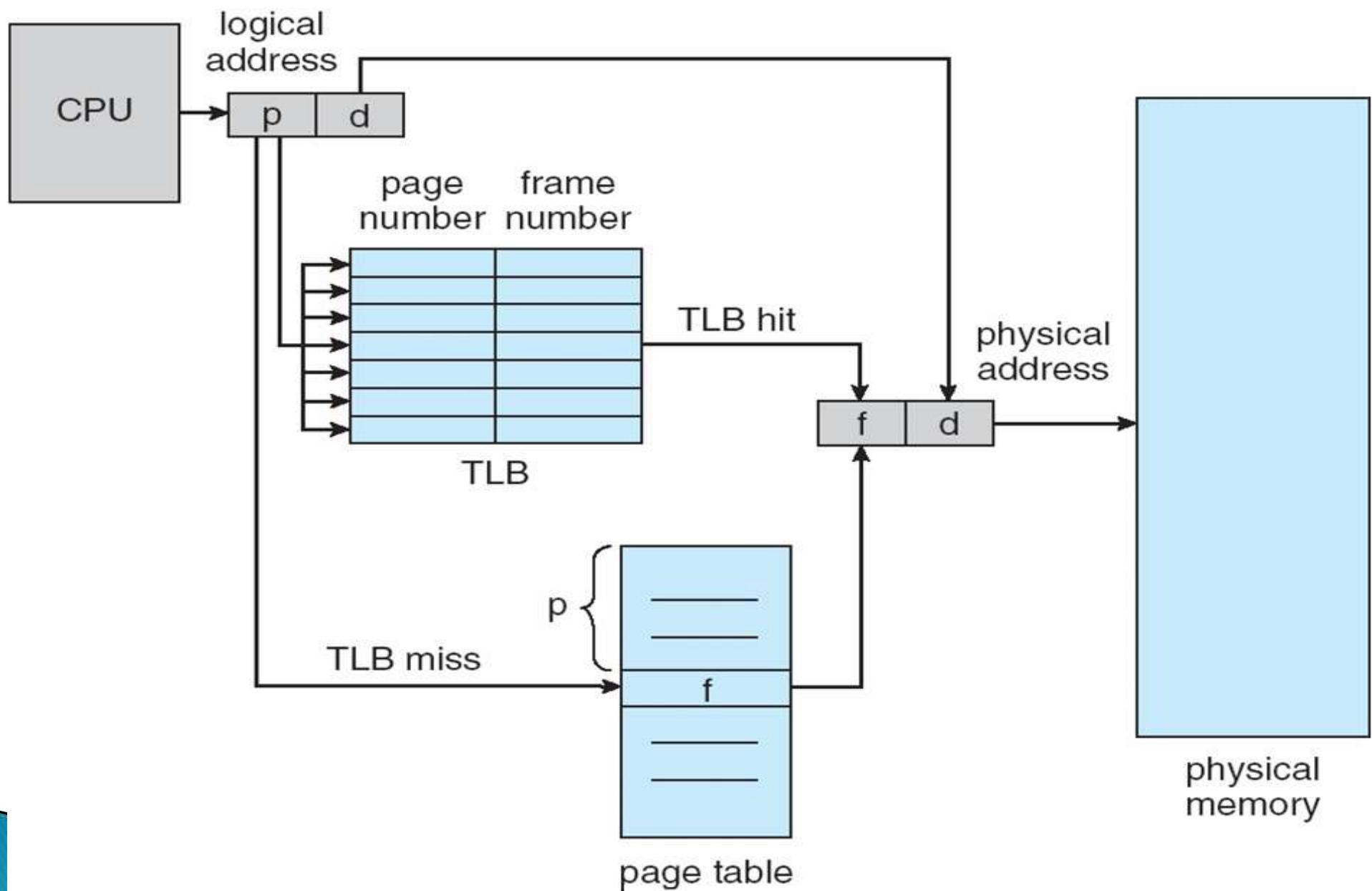
# Implementation of Page Table

- ▶ Page table is kept in main memory
- ▶ **Page-table base register (PTBR)** points to the page table
- ▶ **Page-table length register (PTLR)** indicates size of the page table
- ▶ In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- ▶ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

# Implementation of Page Table (Cont.)

- ▶ Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- ▶ TLBs typically small (64 to 1,024 entries)
- ▶ On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

# Paging Hardware With TLB



# Effective Access Time [1 / 2]

- ▶ Hit ratio =  $\alpha$ 
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- ▶ **Effective Access Time (EAT)**

$$EAT = \alpha \times \text{memory access time (hit)} + (1 - \alpha) \times \text{memory access time (failed)}$$

# Effective Access Time [2/2]

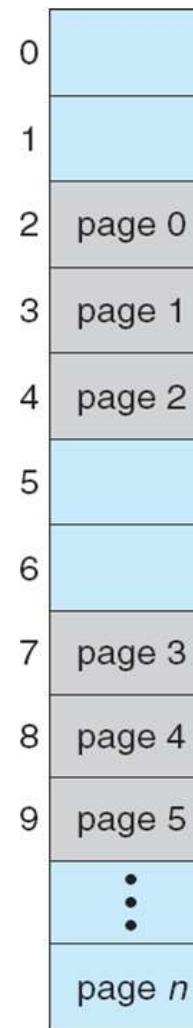
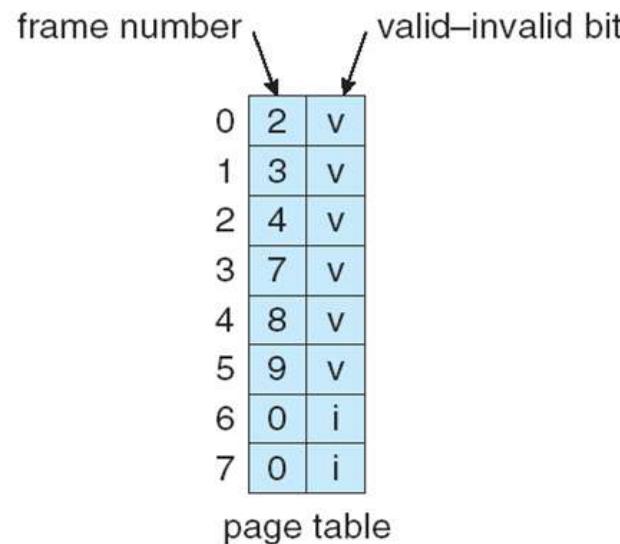
- ▶ Consider  $\alpha = 80\%$ , 100ns for memory access
  - $EAT = 0.80 \times 100 + 0.20 \times (100+100) = 120\text{ns}$
- ▶ Consider more realistic hit ratio  $\rightarrow \alpha = 99\%$  , 100ns for memory access
  - $EAT = 0.99 \times 100 + 0.01 \times (100+100) = 101\text{ns}$

# Memory Protection

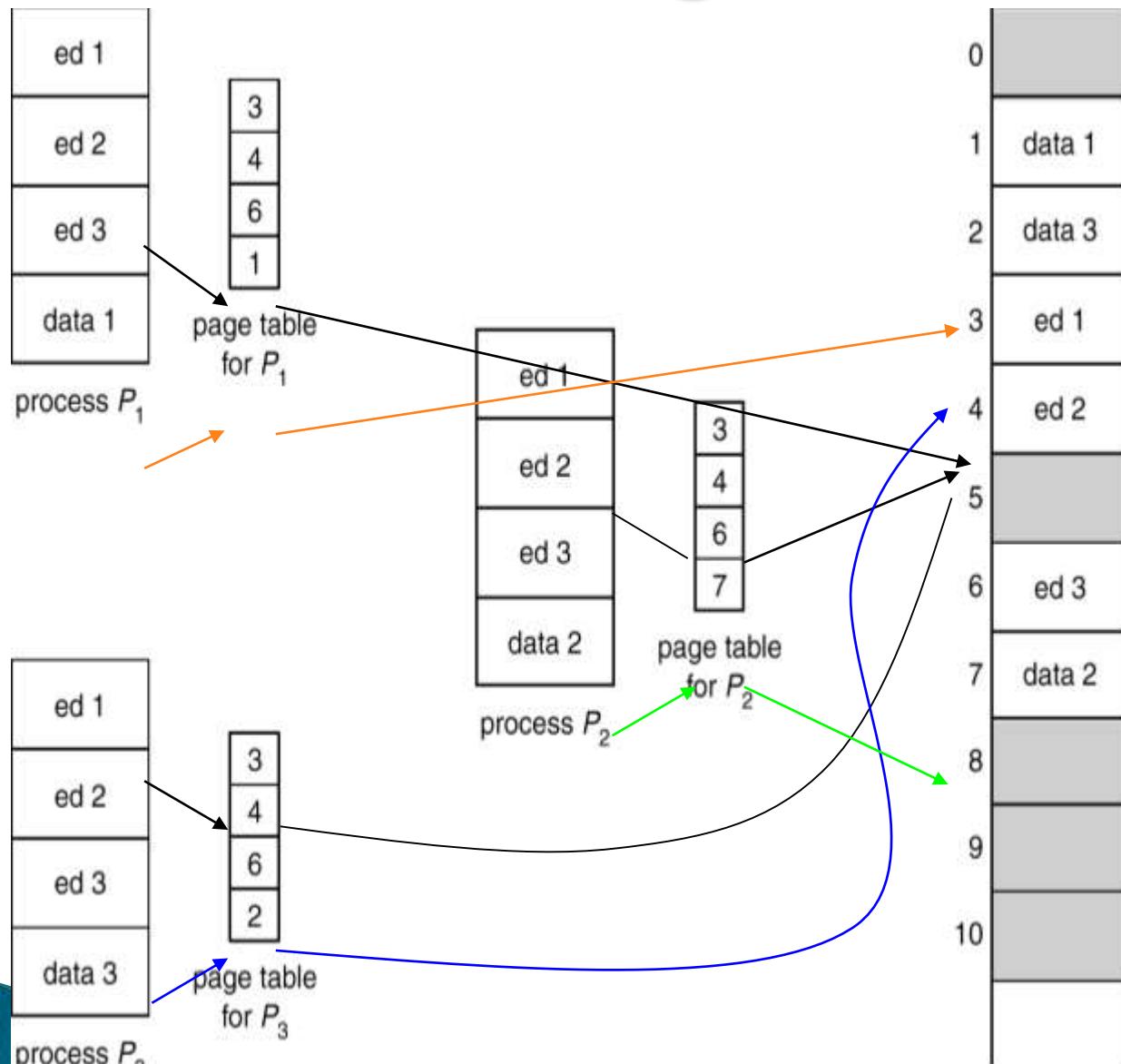
- ▶ Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
- ▶ **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register (PTLR)**

# Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	



# Shared Pages



- ▶ **Shared code**
  - Read-only (reentrant) code shared among processes
  - Shared code appeared in same location in the physical address space
- ▶ **Private code and data**
  - Each process keeps a separate copy of the code and data, (e.g., stack).
  - Private page can appear anywhere in the physical address space.
- ▶ **Copy on write**
  - Pages may be initially shared upon a fork
  - They will be duplicated upon a write

# **Virtual Memory**

## **Course Instructor: Nausheen Shoaib**

# Definition of Virtual Memory [1 / 2]

- ▶ **Virtual memory** is a **memory management** capability of an OS
- ▶ uses hardware and software to allow a computer to compensate for **physical memory** shortages by temporarily transferring data from random access **memory** (RAM) to disk storage.

# Definition of Virtual Memory [2 / 2]

- ▶ Virtual memory is not RAM (Random access memory) on memory chips.
- ▶ Virtual memory is an area of hard drive or other storage space, which OS uses as swap space (overflow) for the physical RAM.

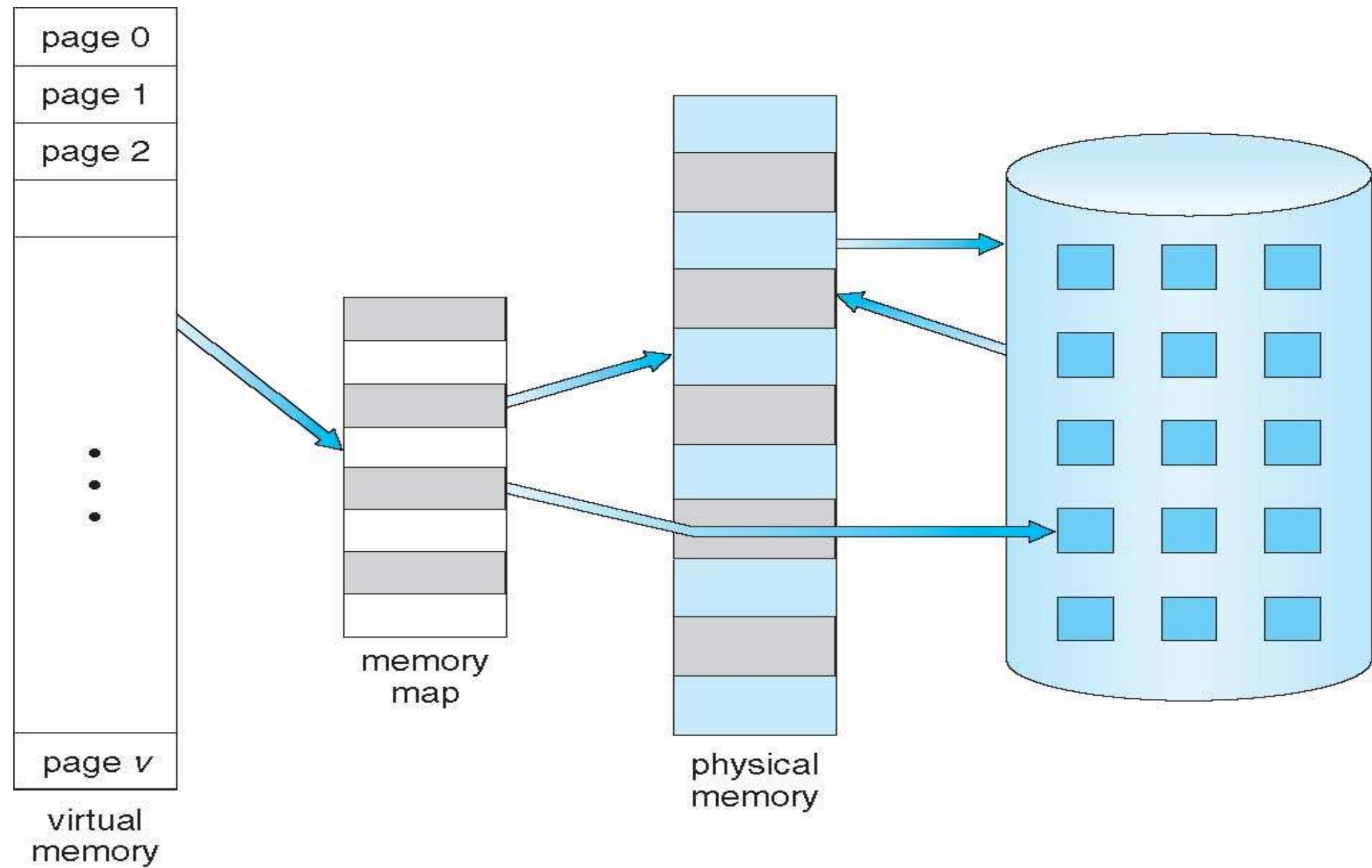
When the demands of the software and data exceed the physical RAM size, the operating system copies to hard disk blocks of data from RAM that have been seldom used, and releases that RAM for use by the current process.

- ▶ When the block (or Page) is required again, the OS makes room in RAM by copying another block to hard drive and copying in the data from the first block.

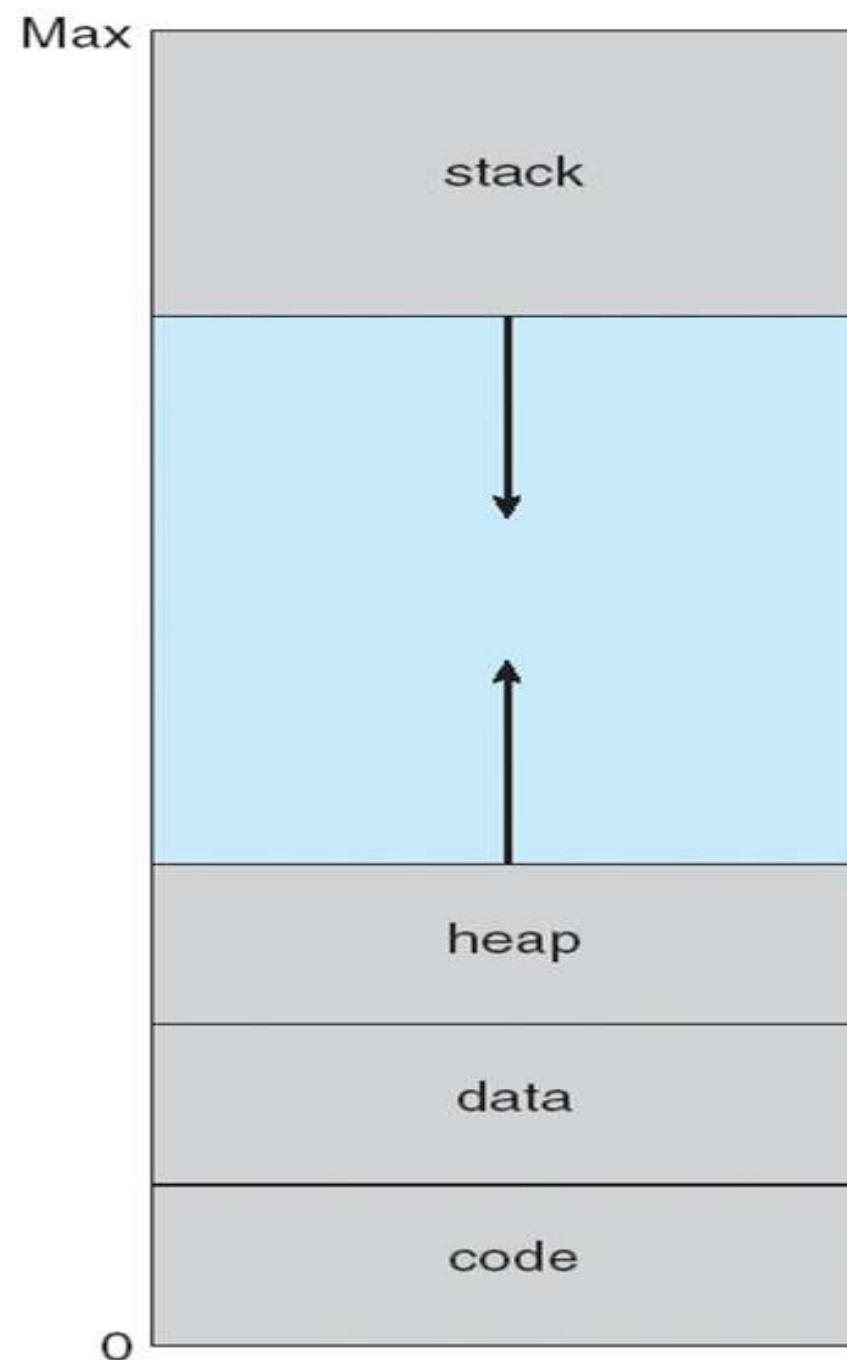
# Background

- ▶ **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- ▶ Virtual memory can be implemented via:
  - Demand paging

# Virtual Memory That is Larger Than Physical Memory



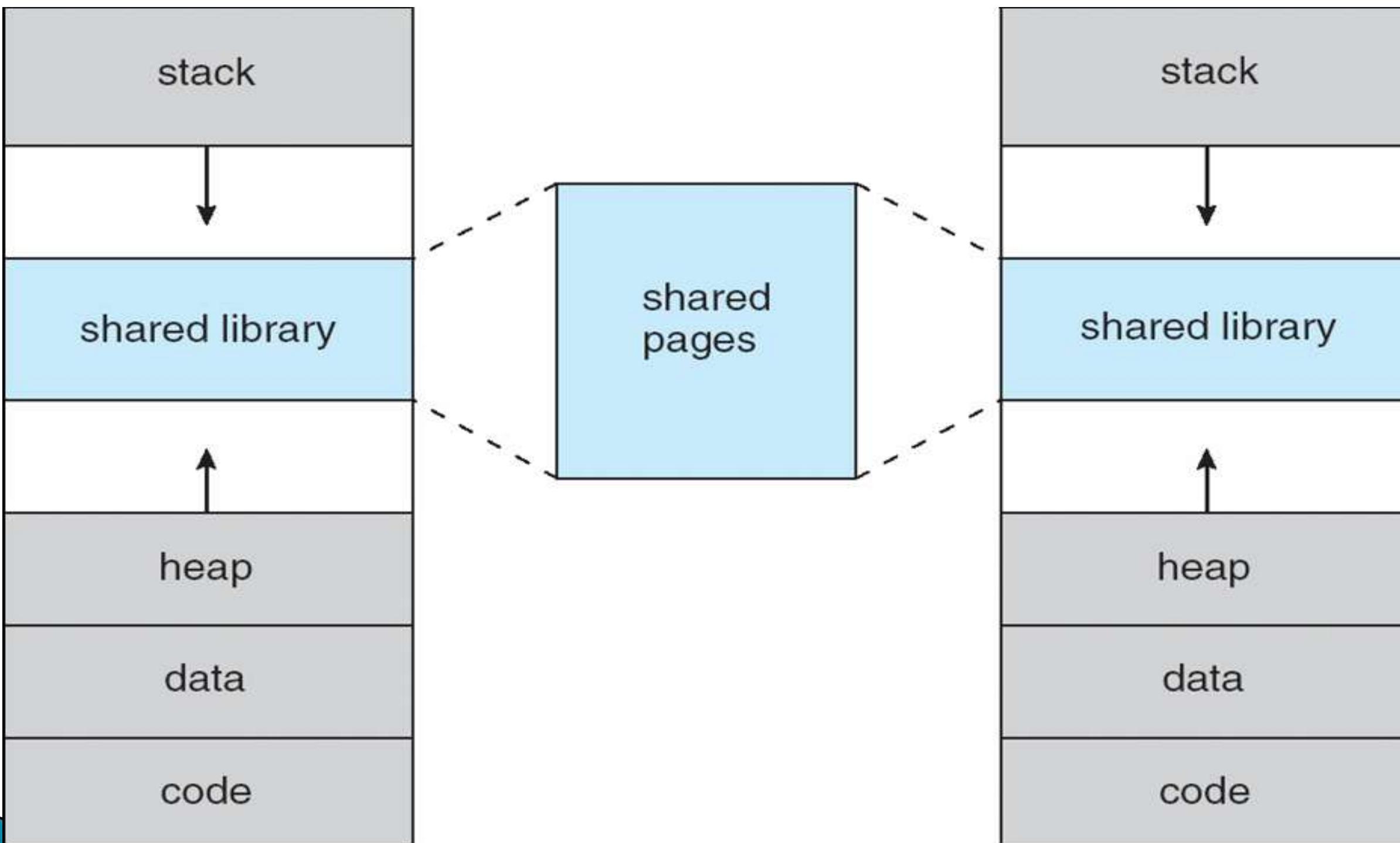
# Virtual-address Space



# Virtual Address Space

- ▶ Virtual address spaces that include holes are known as **sparse** address spaces.
- ▶ System libraries shared via mapping into virtual address space
- ▶ Shared memory by mapping pages read–write into virtual address space
- ▶ Pages can be shared during `fork()`, speeding process creation

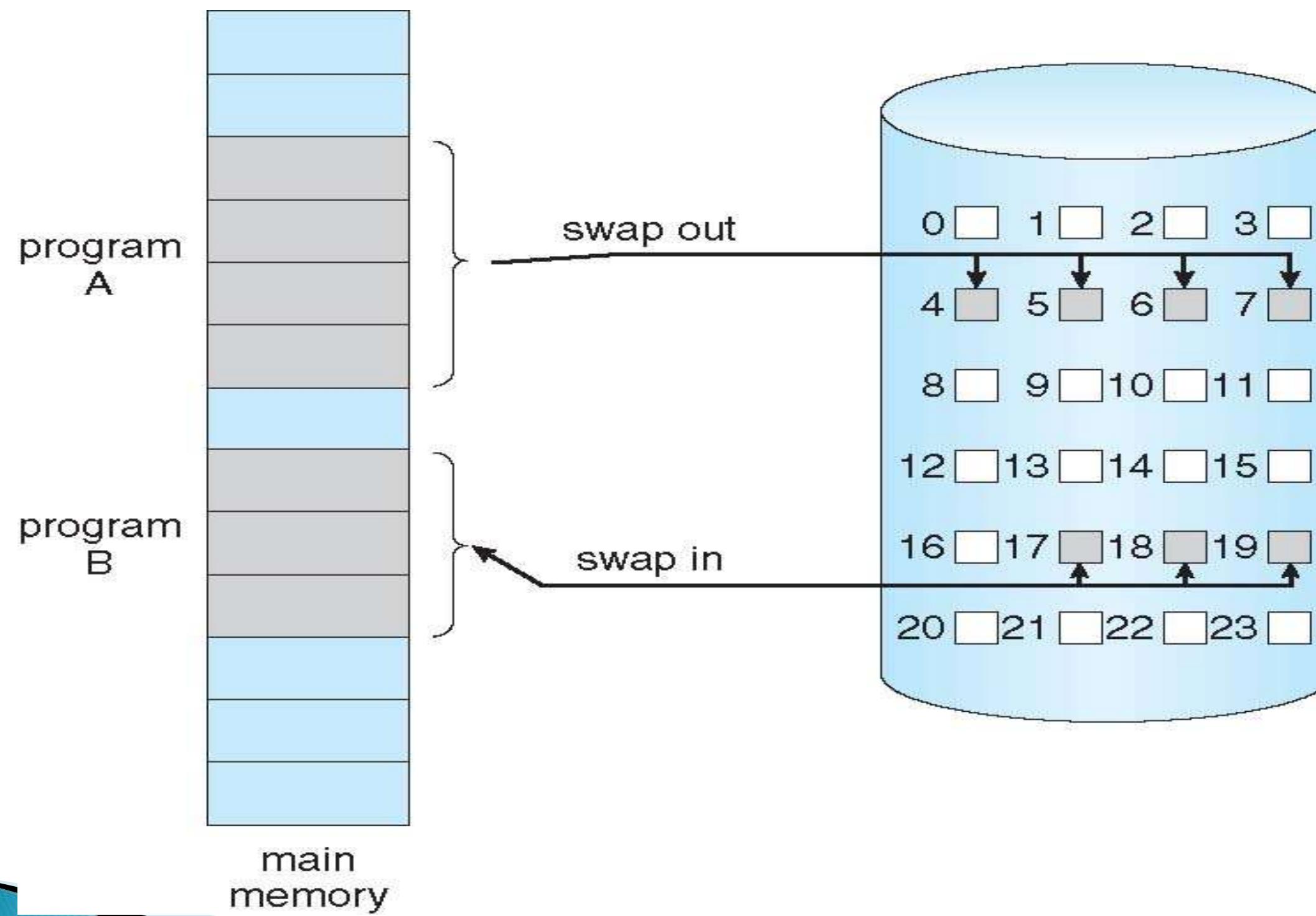
# Shared Library Using Virtual Memory



# Demand Paging

- ▶ Could bring entire process into memory at load time
- ▶ Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- ▶ Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- ▶ **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

# Transfer of a Paged Memory to Contiguous Disk Space



# Valid-Invalid Bit

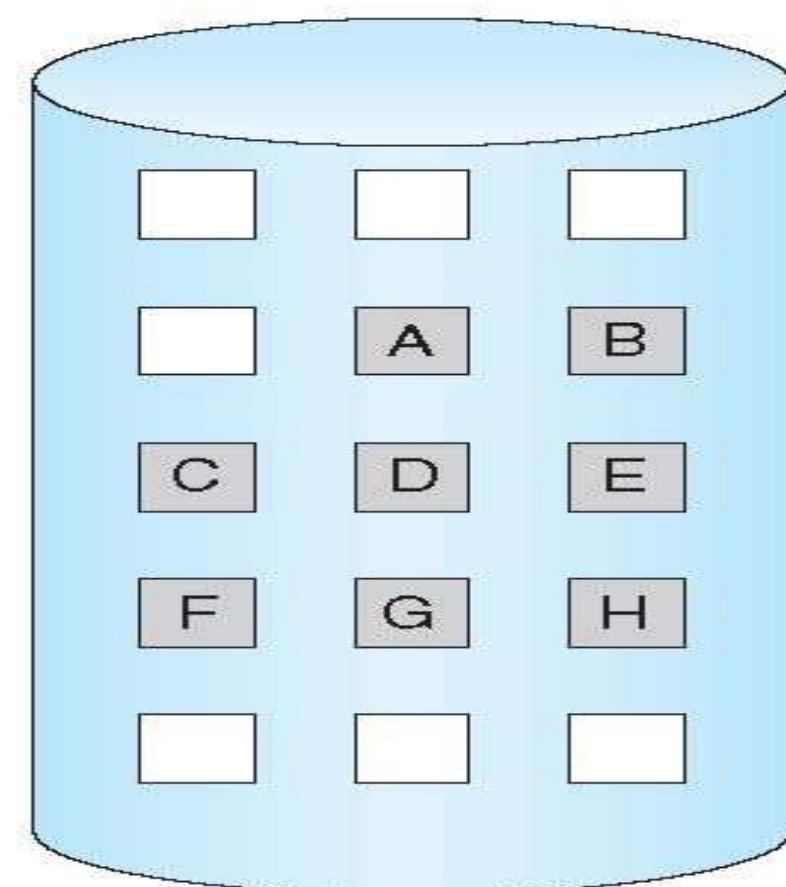
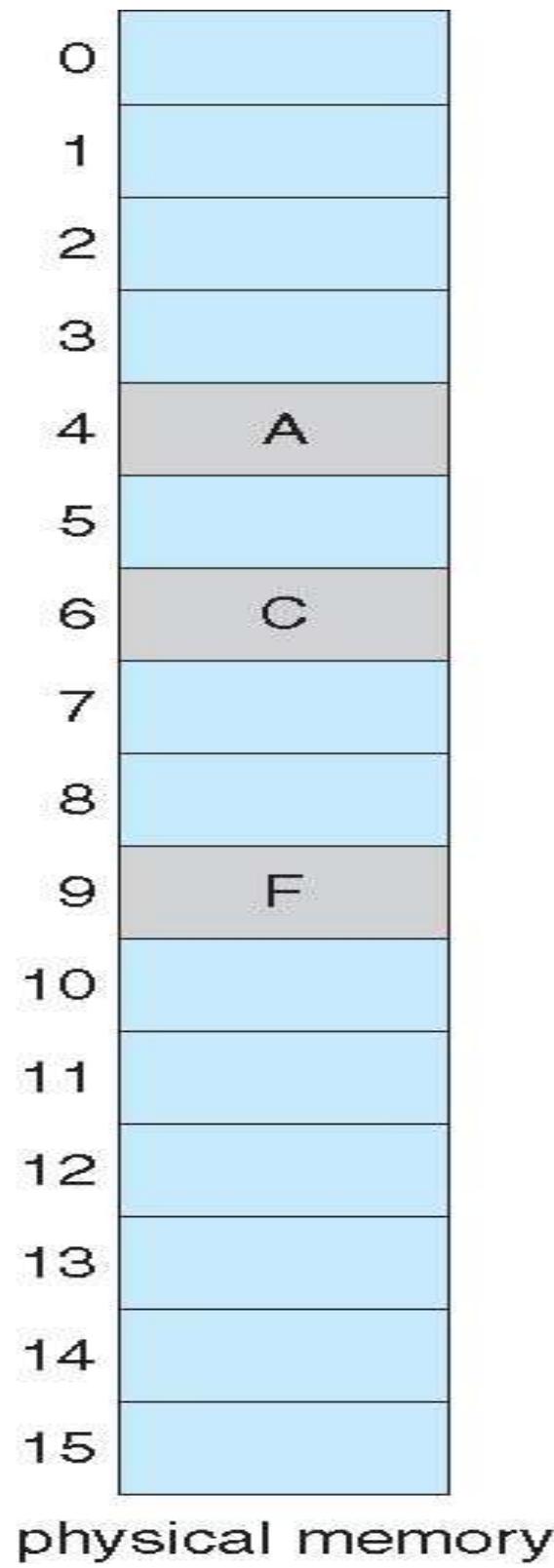
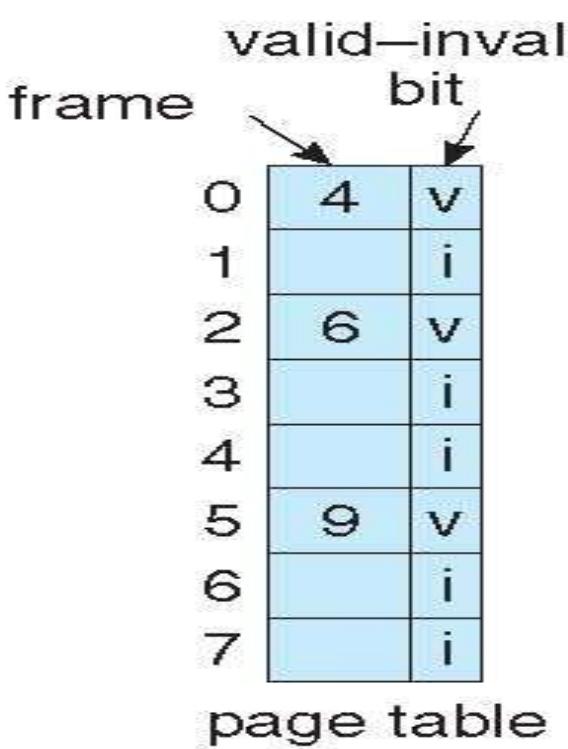
- ▶ With each page table entry a valid-invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- ▶ Initially valid-invalid bit is set to **i** on all entries
- ▶ Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
....	
	i
	i

page table

- ▶ During address translation, if valid-invalid bit in page table entry is **I** ⇒ page fault

# Page Table When Some Pages Are Not in Main Memory



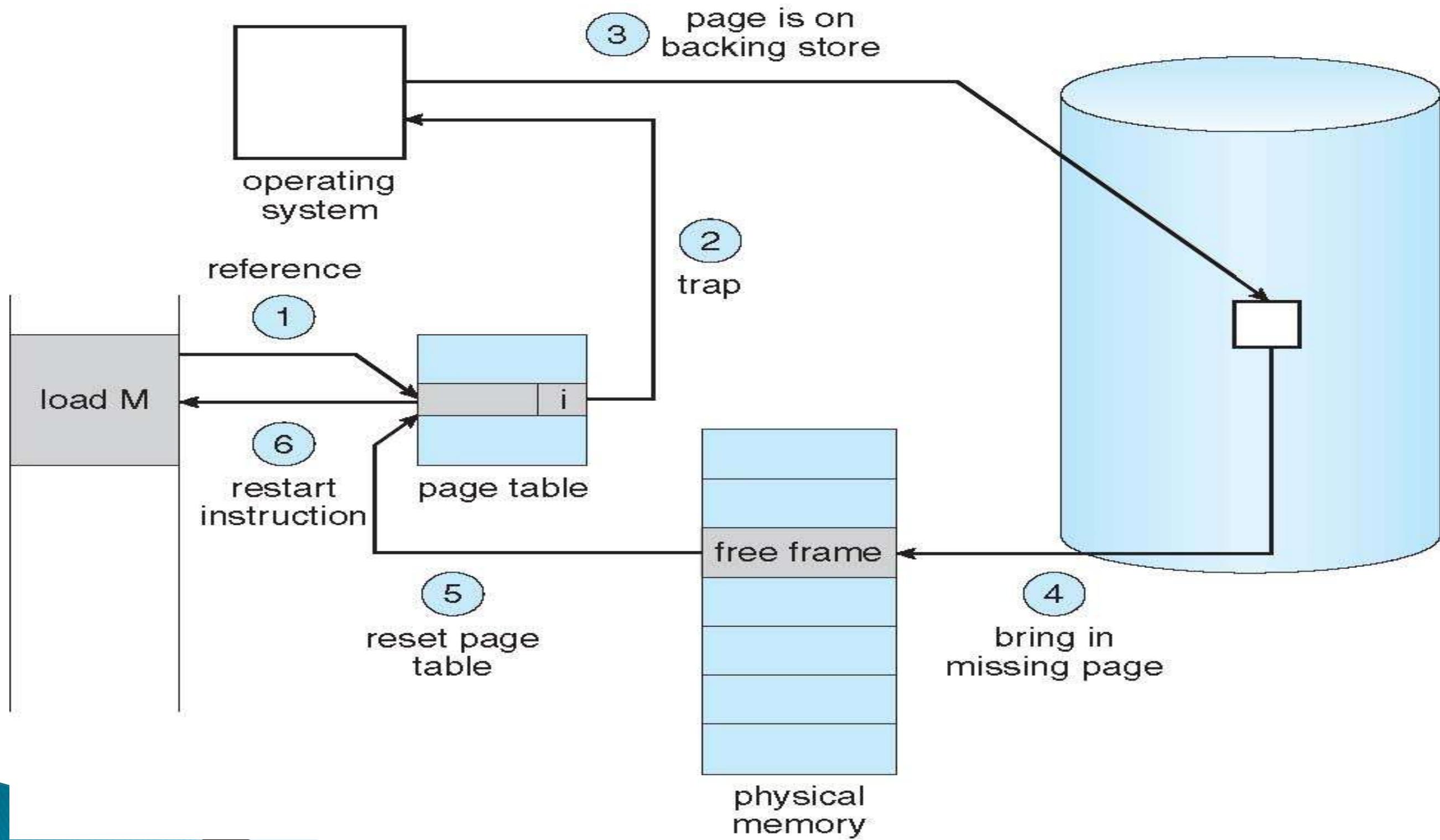
# Page Fault

- ▶ If there is a reference to a page, first reference to that page will trap to operating system:  
**page fault**
- 1. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
- 2. Get empty frame
- 3. Swap page into frame via scheduled disk operation
- 4. Reset tables to indicate page now in memory  
Set validation bit = **V**
- 5. Restart the instruction that caused the page fault

# Aspects of Demand Paging

- ▶ Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- ▶ Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- ▶ Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

# Steps in Handling a Page Fault



# Performance of Demand Paging (Cont.)

- ▶ Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault
- ▶ Effective Access Time (EAT)  
effective access time =  $(1 - p) \times ma$   
 $+ p \times$  page fault service time.

## Performance of Demand Paging (Cont.)

Memory Access Time = 200 nanoseconds

Average page fault service time = 8 millisecond

If one access out of 1,000 causes a page fault, then

$$p=1/1000=0.001$$

effective access time =  $(1 - p) \times ma + p \times \text{page fault time.}$

$$\text{EAT} = (1-0.001)*200+(0.001* 8e-6)$$

$$= 8199.8 \text{ nanosecond}$$

$$= 8.199 \text{ microsecond}$$

Millisecond =  $10e-3$

Microsecond =  $10e-6$

Nanosecond =  $10e-9$

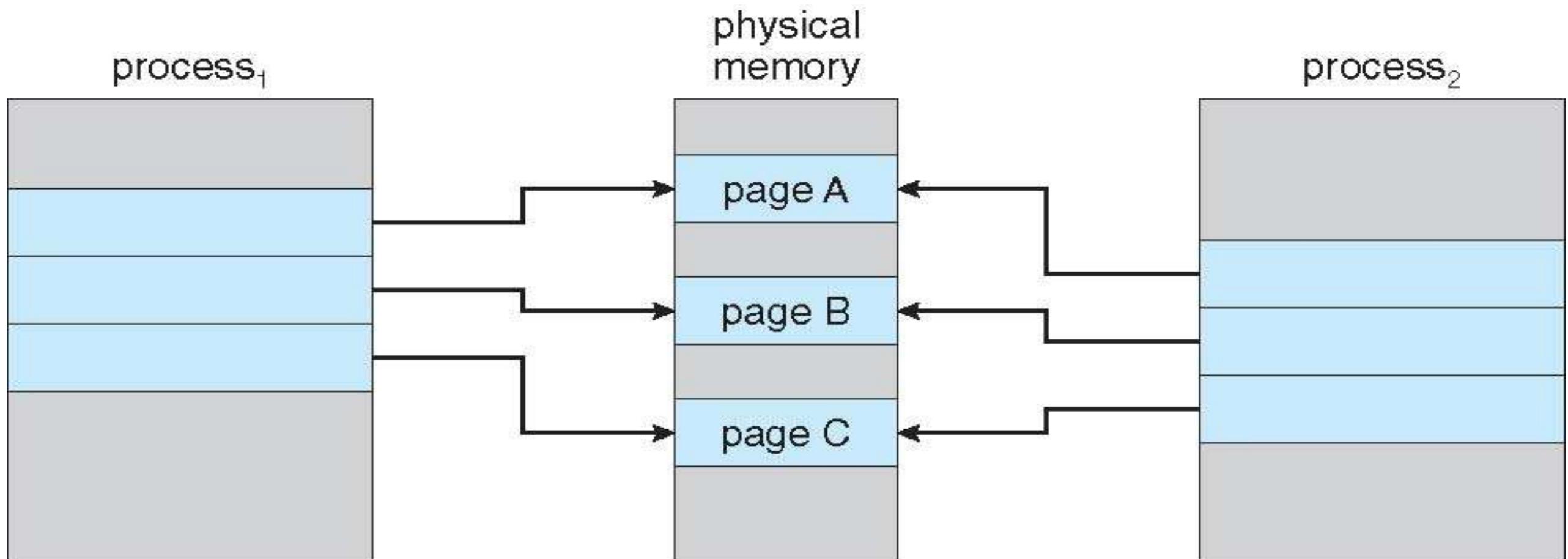
# Demand Paging Optimizations

- ▶ Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- ▶ Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD

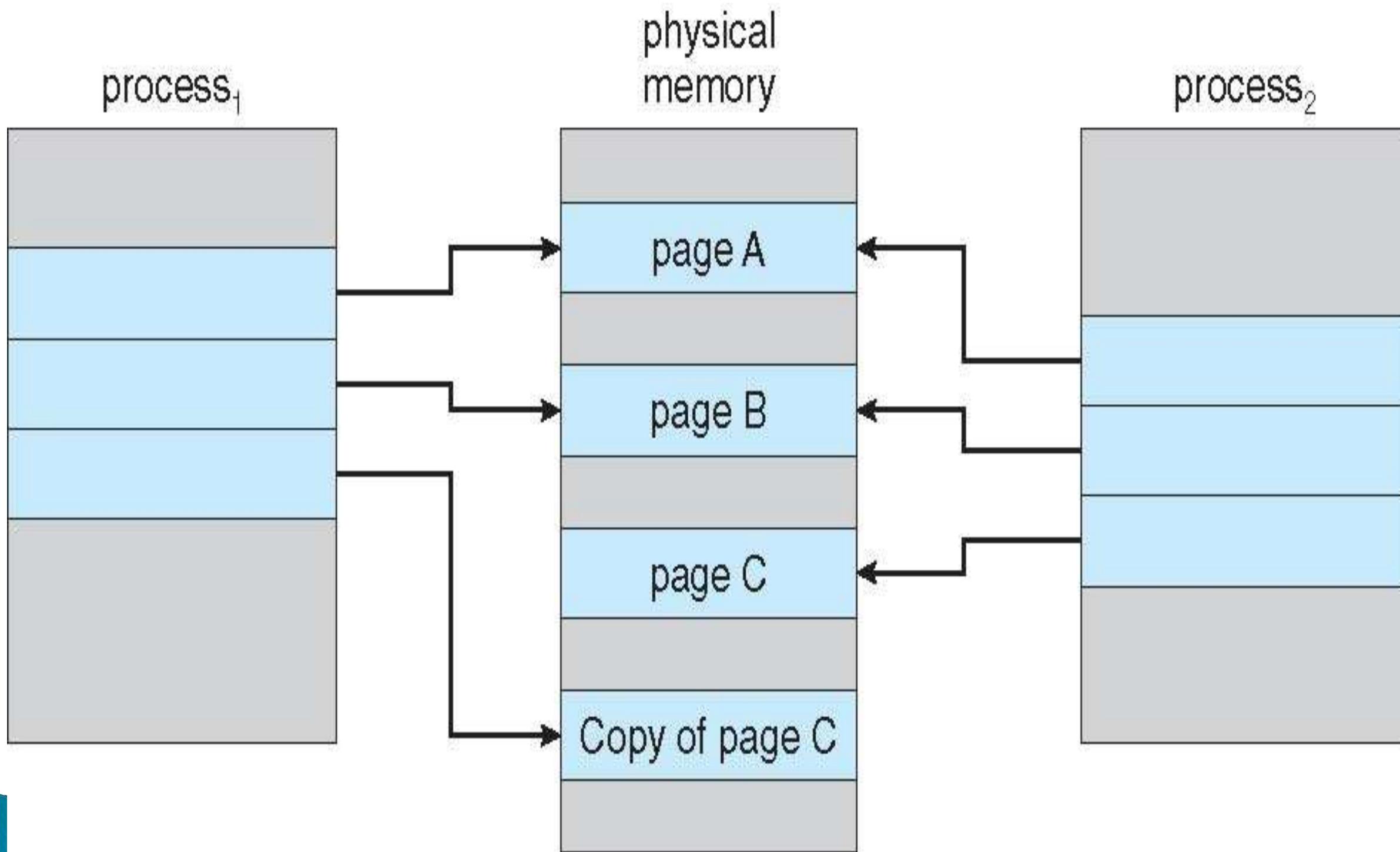
# Copy-on-Write

- ▶ **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- ▶ COW allows more efficient process creation as only modified pages are copied
- ▶ In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
- ▶ `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

# Before Process 1 Modifies Page C



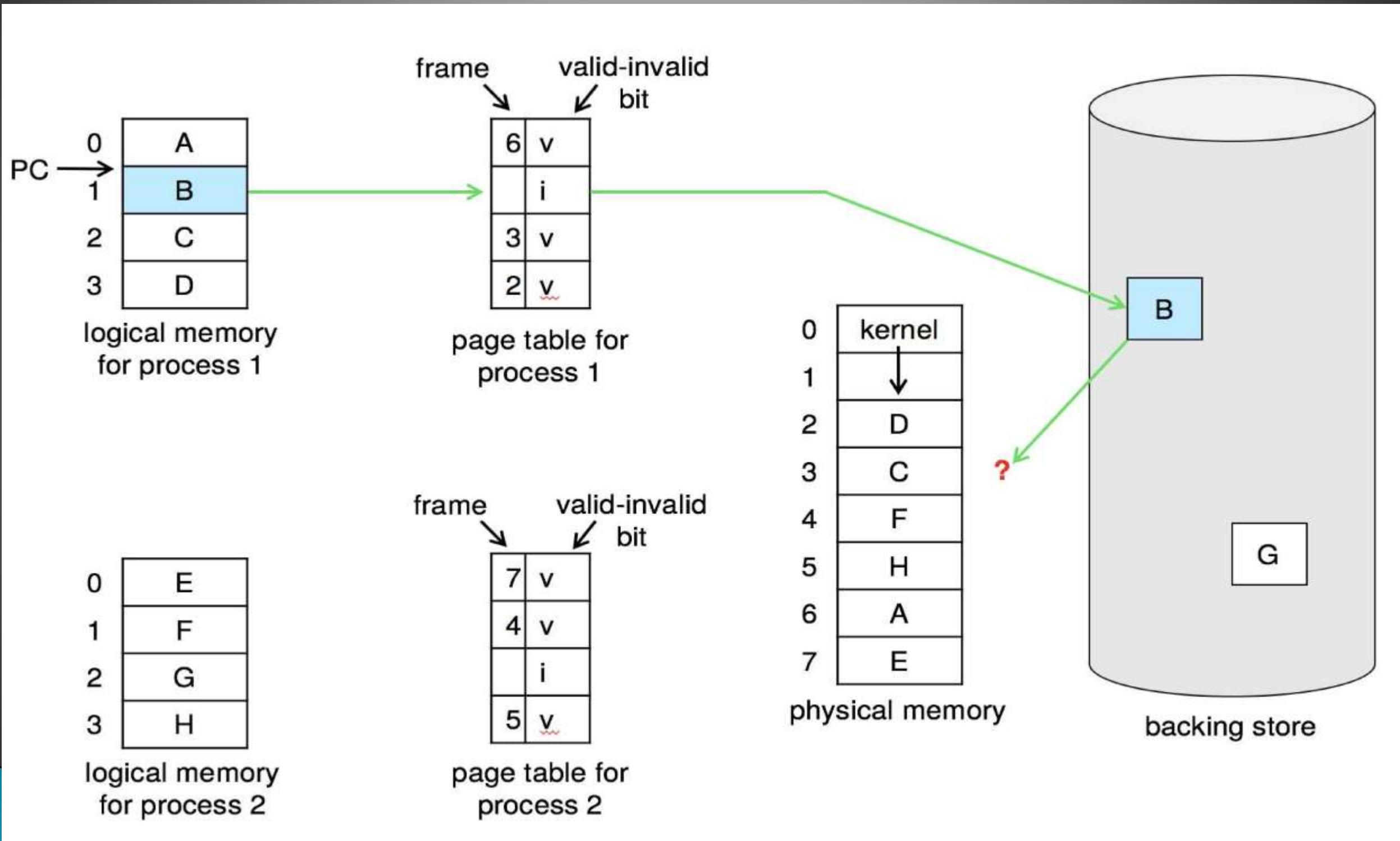
# After Process 1 Modifies Page C



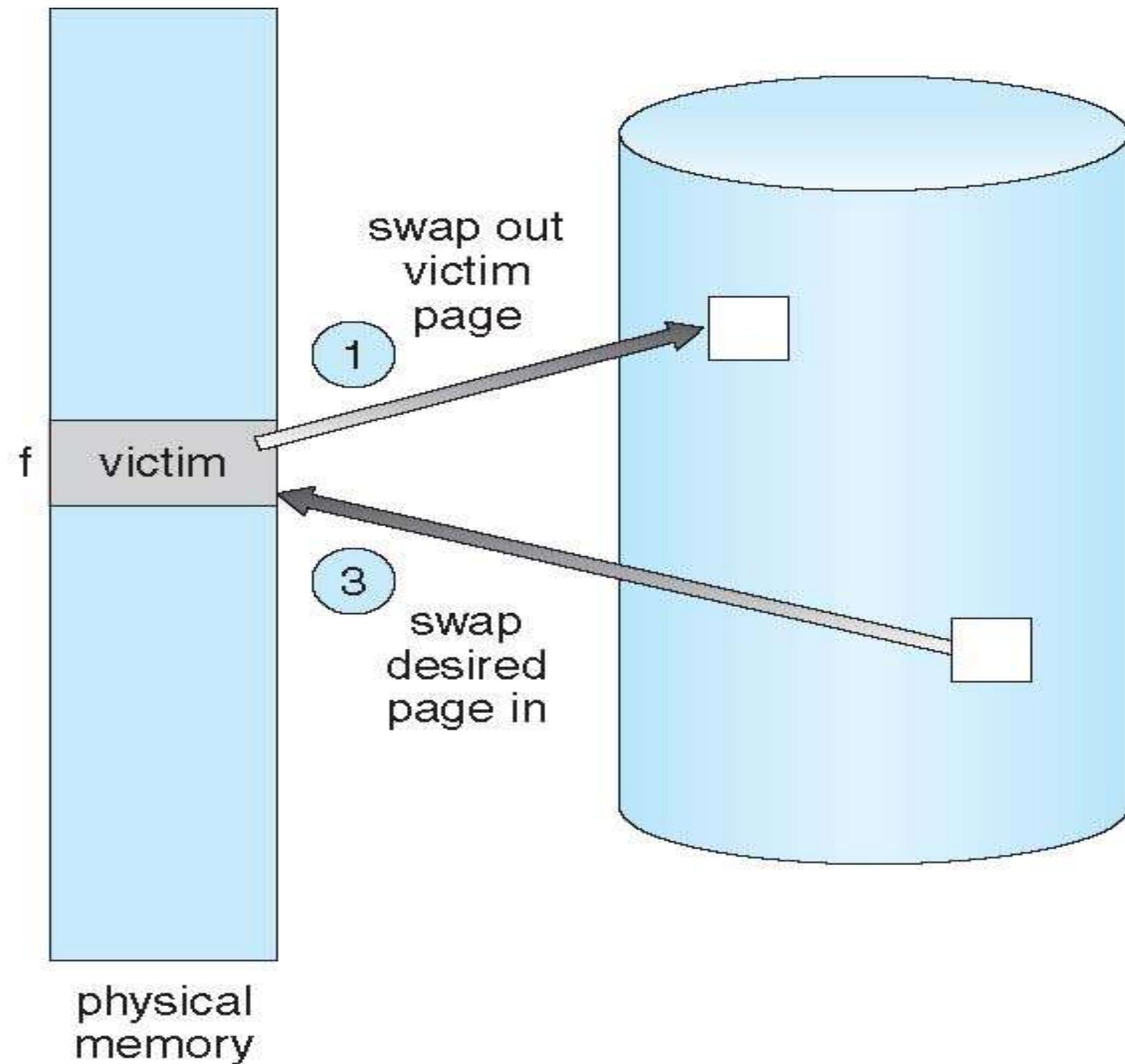
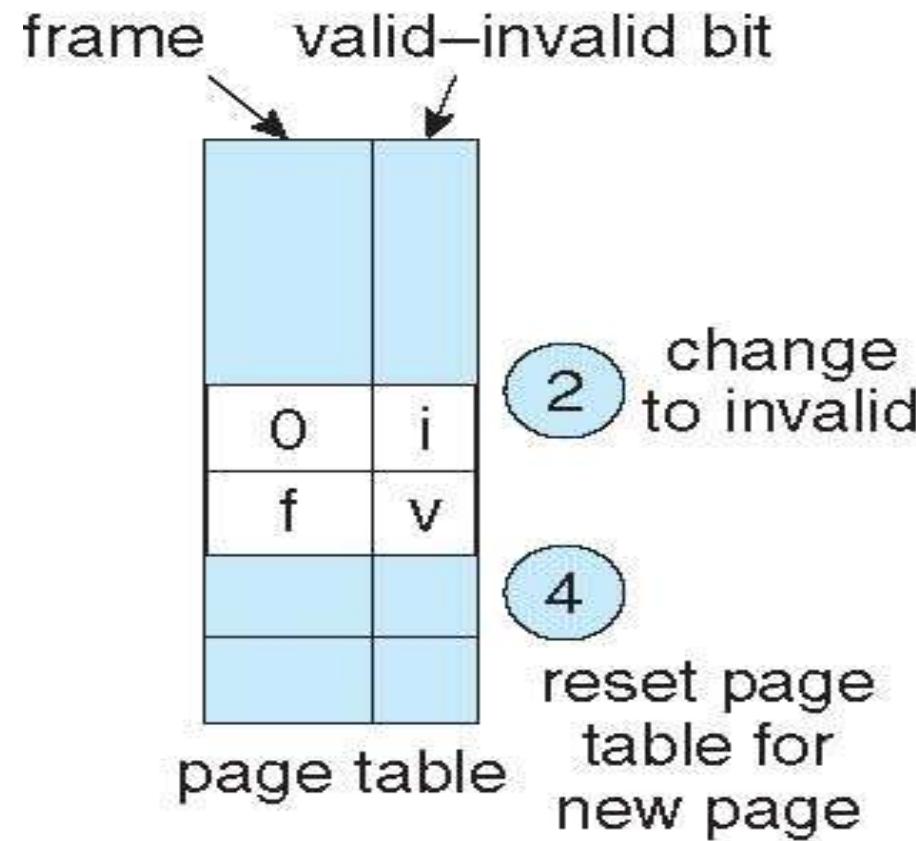
## What Happens if There is no Free Frame?

- ▶ Used up by process pages
  - ▶ Also in demand from the kernel, I/O buffers, etc
  - ▶ How much to allocate to each?
- 
- ▶ Page replacement – find some page in memory, but not really in use, page it out
    - Algorithm – terminate? swap out? replace the page?
    - Performance – want an algorithm which will result in minimum number of page faults
  - ▶ Same page may be brought into memory several times

# Need For Page Replacement



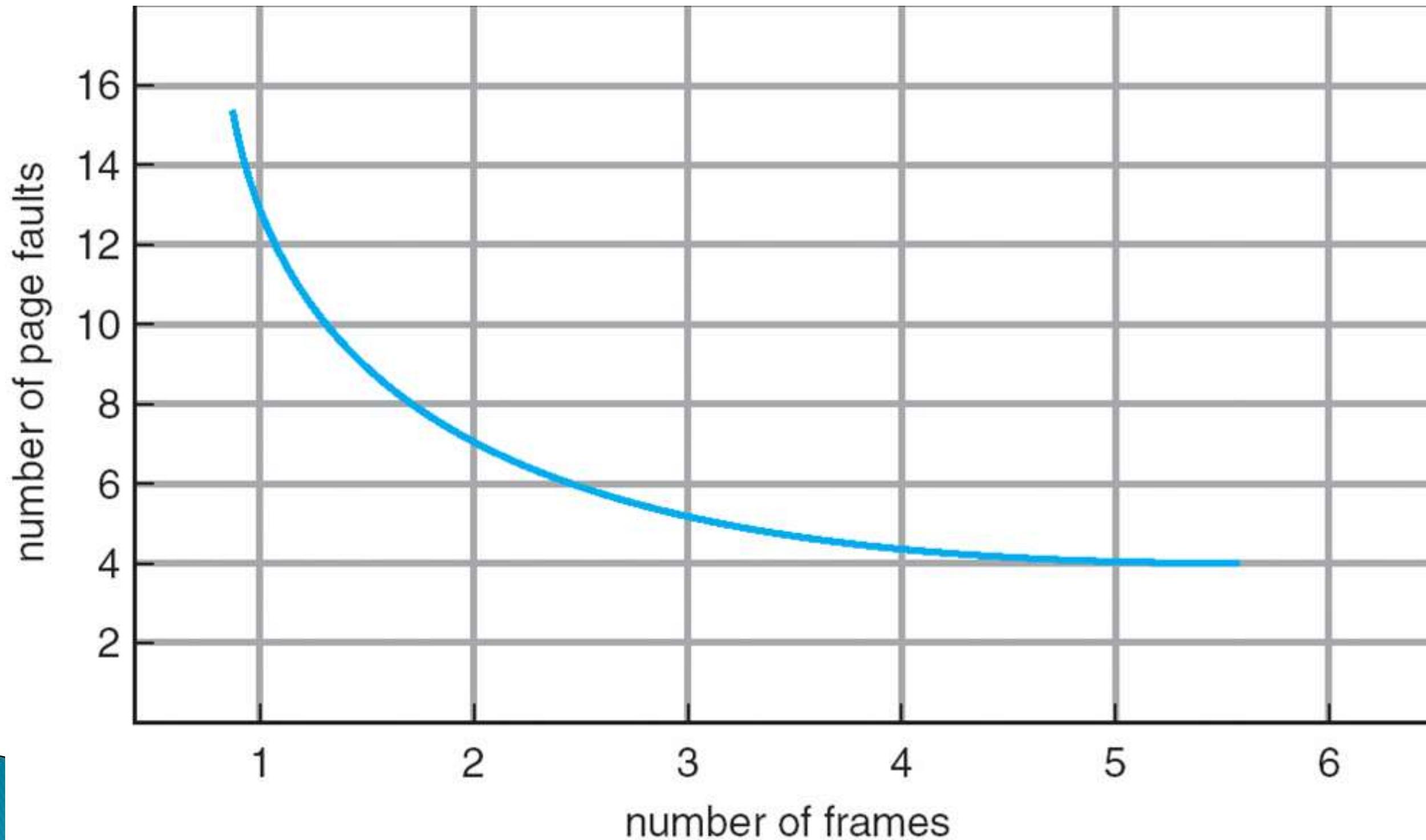
# Page Replacement



# Page and Frame Replacement Algorithms

- ▶ **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- ▶ **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

# Graph of Page Faults Versus The Number of Frames



# First-In-First-Out (FIFO) Algorithm

- ▶ Reference string:  
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- ▶ 3 frames (3 pages can be in memory at a time per process)

1	7	2	4	0	7	
2	0	3	2	1	0	
3	1	0	3	2	1	15 page faults

- ▶ Can vary by reference string: consider  
1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- ▶ How to track ages of pages?
  - Just use a FIFO queue

# FIFO Page Replacement Example

- ▶ Covered in Class

# Least Recently Used (LRU) Algorithm

- ▶ Use past knowledge rather than future
- ▶ Replace page that has not been used in the most amount of time
- ▶ Associate time of last use with each page

# LRU Algorithm (Cont.)

- ▶ Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- ▶ Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

# LRU Algorithm Example

- ▶ Covered in Class

# Optimal Algorithm

- ▶ Replace page that will not be used for longest period of time

# Optimal Page Replacement Example

- ▶ Covered in Class

# LRU Approximation Algorithms

- ▶ LRU needs special hardware and still slow
- ▶ **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- ▶ **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

# Second Chance Algorithm

- ▶ Covered in Class

# **Process Synchronization**

## **Course Instructor: Nausheen Shoaib**

# Multiple Processes

- ▶ Operating Systems is managing multiple processes:
  - Multiprogramming- The management of multiple processes within a uniprocessor system.
  - Multiprocessing- The management of multiple processes within a multiprocessor.
  - Distributed Processing- The management of multiple processes executing on multiple, distributed computer systems.e.g clusters
- ▶ Big Issue is Concurrency

# Concurrency [1 / 2]

- ▶ Concurrency encompasses a host of design issues, including :
  - communication among processes
  - sharing of and competing for resources (such as memory, files, and I/O access)
  - synchronization of the activities of multiple processes
  - allocation of processor time to processes

# **Concurrency [2 / 2]**

**Concurrency arises in:**

- ▶ **Multiple applications**
  - Sharing time
- ▶ **Structured applications**
  - Extension of modular design
- ▶ **Operating system structure**
  - OS themselves implemented as a set of processes or threads

# Difficulties of Concurrency

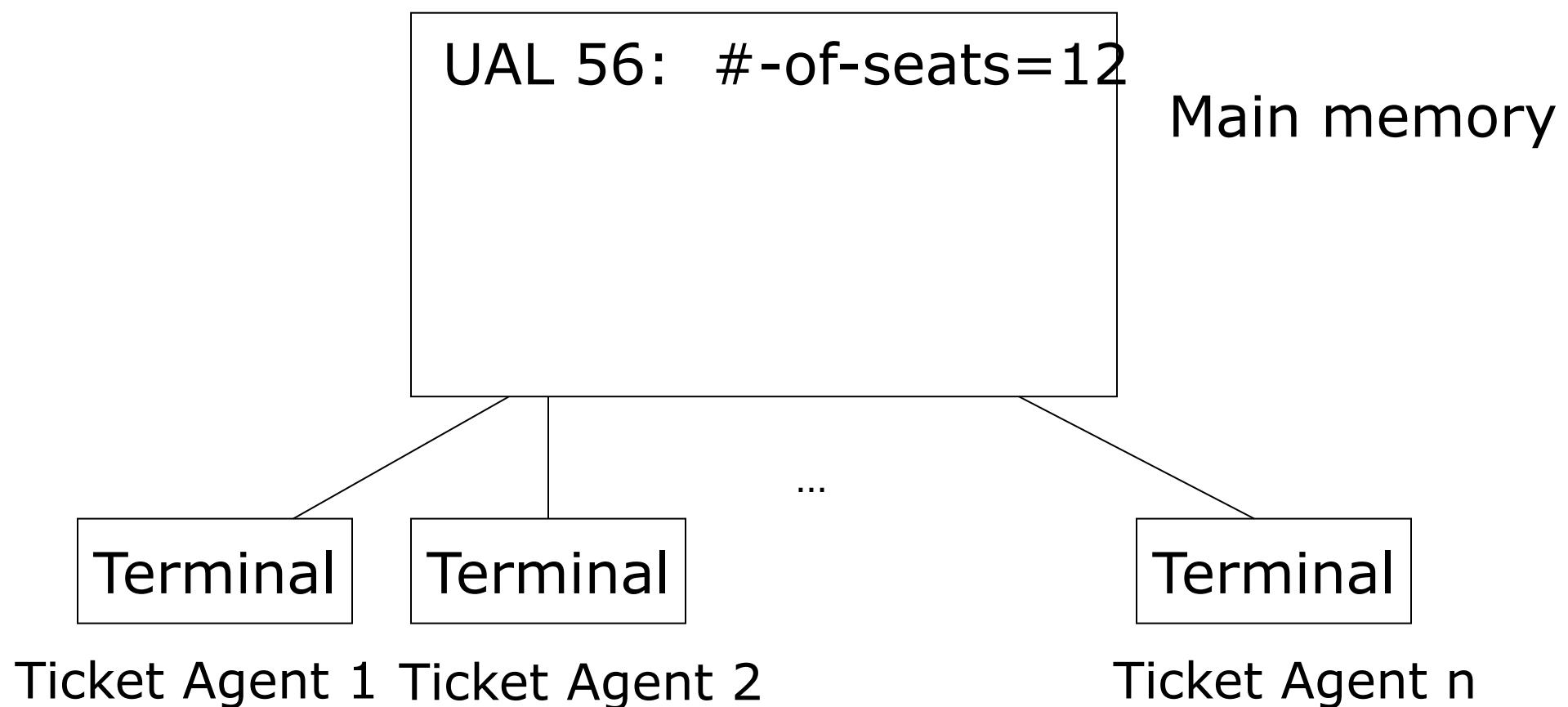
- ▶ Sharing of global resources
  - Writing a shared variable: the order of writes is important
  - Incomplete writes a major problem
- ▶ Optimally managing the allocation of resources
- ▶ Difficult to locate programming errors as results are not deterministic and reproducible.

# Race Condition

- ▶ A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

# Example for Race condition

- ▶ Suppose a customer wants to book a seat on UAL 56. Ticket agent will check the #‐of‐seats. If it is greater than 0, he will grab a seat and decrement #‐of‐seats by 1.



# Example for Race condition(cont.)

Ticket Agent 1

P1: LOAD #-of-seats  
P2: DEC 1  
P3: STORE #-of-seats

Ticket Agent 2

Q1: LOAD #-of-seats  
Q2: DEC 1  
Q3: STORE #-of-seats

Ticket Agent 3

R1: LOAD #-of-seats  
R2: DEC 1  
R3: STORE #-of-seats

Suppose instructions are interleaved as P1,Q1,R1,P2,Q2,R2,P3,Q3,R3

To solve the above problem, we must make sure that:

P1,P2,P3 must be completely executed before we execute Q1 or R1, or  
Q1,Q2,Q3 must be completely executed before we execute P1 or R1, or  
R1,R2,R3 must be completely executed before we execute P1 or Q1.

# The Critical Section / Region Problem

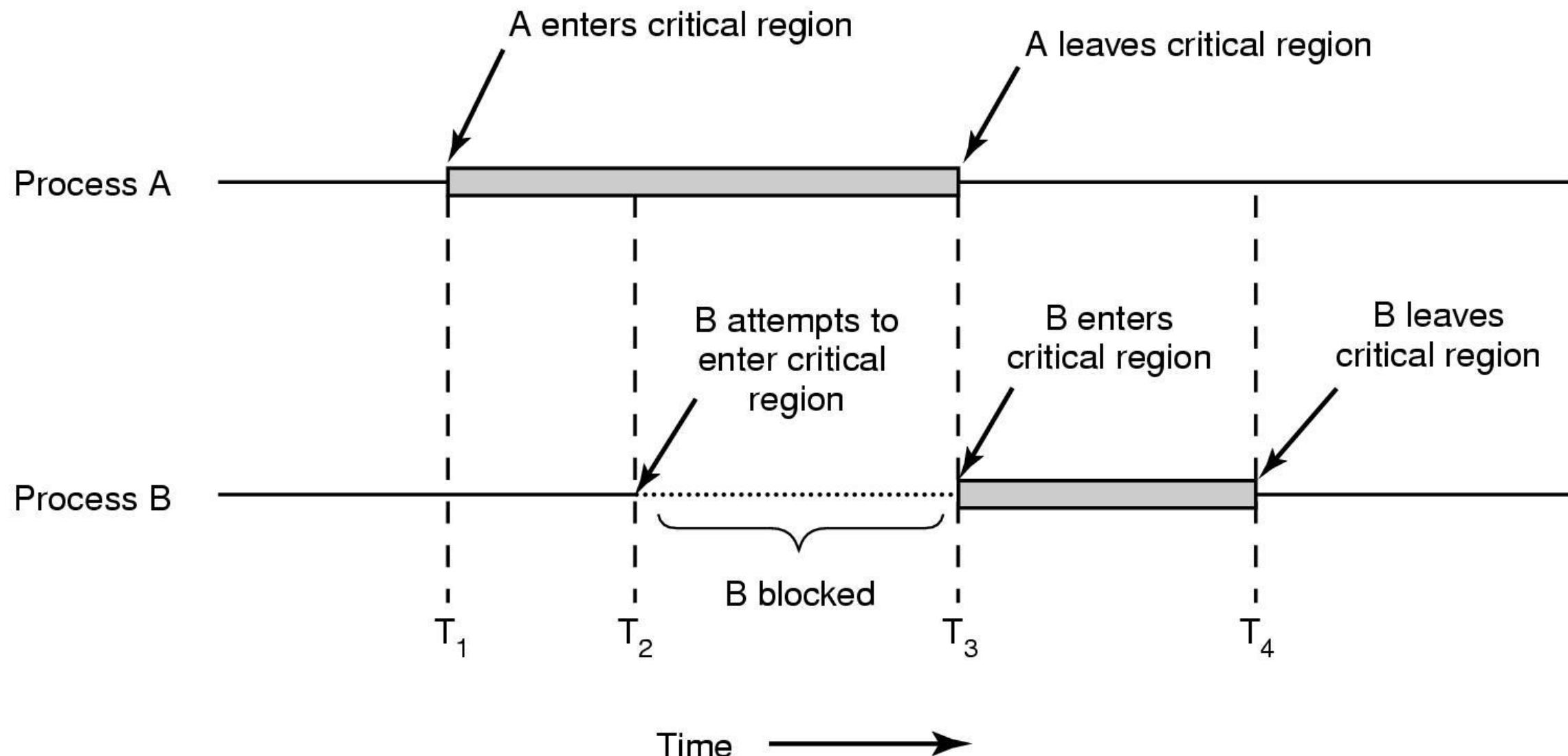
- ▶ Occurs in systems where multiple processes all compete for the use of shared data.
- ▶ Each process includes a section of code (**the critical section**) where it accesses this shared data.
- ▶ The problem is to ensure that **only one process at a time is allowed** to be operating in its critical section.

# Critical Regions (1)

Conditions required to avoid race condition:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

# Critical Regions (2)



Mutual exclusion using critical regions.



# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the  $n$  processes





# Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - ▶ Essentially free of race conditions in kernel mode





# Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process  $P_i$  is ready!





# Algorithm for Process $P_i$

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```





# Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

$P_i$  enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met



# Peterson's Solution and Modern Architecture

- ▶ Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- ▶ Understanding why it will not work is useful for better understanding race conditions.
- ▶ For single-threaded this is ok as the result will always be the same.
- ▶ For multithreaded the reordering may produce inconsistent or unexpected results!

# Modern Architecture Example

- ▶ Two threads share the data:

```
boolean flag = false;
```

```
int x = 0;
```

- ▶ Thread 1 performs

```
while (!flag)
```

```
;
```

```
    print x
```

- ▶ Thread 2 performs

```
    x = 100;
```

```
    flag = true
```

- ▶ What is the expected output?

# Modern Architecture Example (Cont.)

- ▶ However, since the variables `flag` and `x` are independent of each other, the instructions:

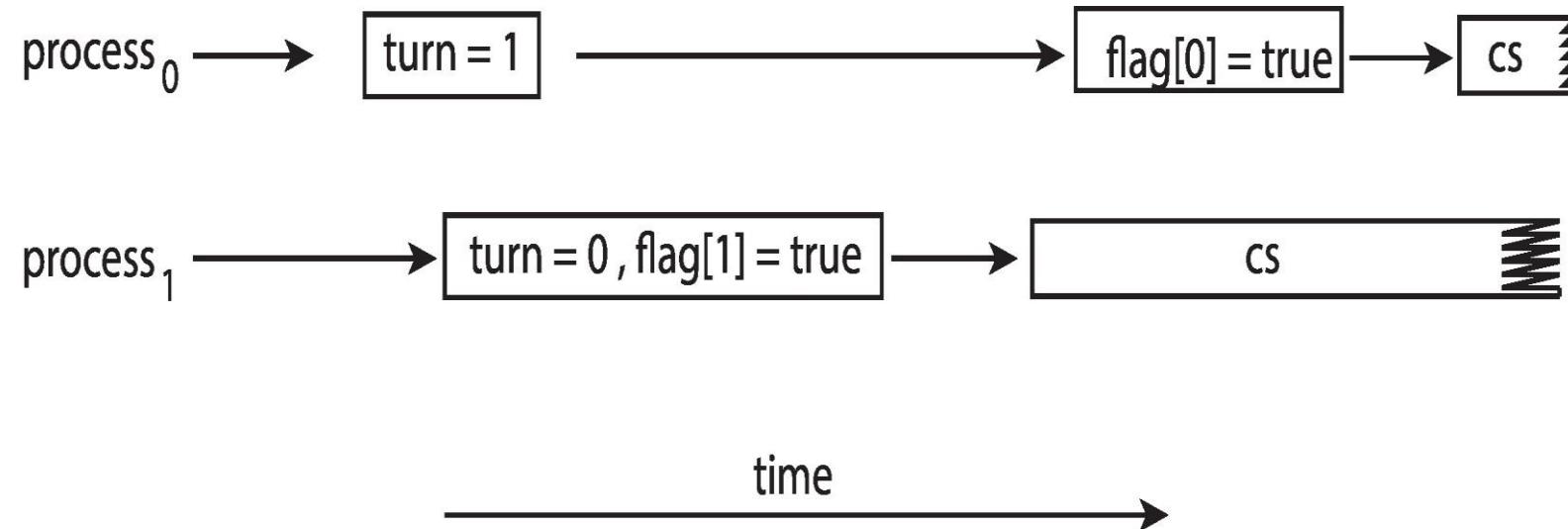
```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

- ▶ If this occurs, the output may be 0!

# Peterson's Solution Revisited

- ▶ The effects of instruction reordering in Peterson's Solution



- ▶ This allows both processes to be in their critical section at the same time!
- ▶ To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

# Memory Barrier

- ▶ **Memory model** are the memory guarantees a computer architecture makes to application programs.
- ▶ Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- ▶ A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

# Memory Barrier Instructions

- ▶ When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- ▶ Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

# Synchronization Hardware

- ▶ Many systems provide hardware support for implementing the critical section code.
- ▶ Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- ▶ We will look at three forms of hardware support:
  1. Hardware instructions
  2. Atomic variables

# Hardware Instructions

- ▶ Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptedly.)
  - Test-and-Set instruction
  - Compare-and-Swap instruction

# The test\_and\_set Instruction

- ▶ Definition

```
boolean test_and_set  
(boolean *target)  
{  
    boolean rv =  
*target;  
    *target = true;  
    return rv;  
}
```

# The test\_and\_set Instruction

- ▶ if two test and set() instructions are executed simultaneously(each on a different core), they will be executed sequentially in some arbitrary order.
- ▶ If the machine supports the test and set() instruction, then we can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

# Solution Using test\_and\_set()

- ▶ Shared boolean variable **lock**, initialized to **false**
- ▶ Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
    /* remainder section */  
}  
while (true);
```

# The compare\_and\_swap Instruction

- ▶ Definition

```
int compare_and_swap(int *value, int expected, int new_value)
```

```
{  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;
```

```
}
```

- ▶ Properties

- Executed atomically
- Returns the original value of passed parameter **value**
- Set the variable **value** the value of the passed parameter **new\_value** but only if **\*value == expected** is true. That is, the swap takes place only under this condition.

# The compare\_and\_swap Instruction

- ▶ if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.
- ▶ Mutual exclusion using CAS can be provided

# Solution using compare\_and\_swap

- ▶ Shared integer **lock** initialized to 0;
- ▶ Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1)
!= 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

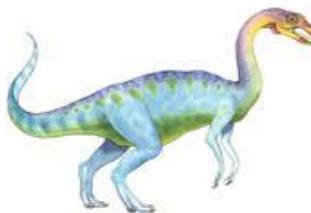
    /* remainder section */
}
```

# Atomic Variables

- ▶ Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- ▶ One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.

# Mutex Locks

- n OS designers build software tools to solve critical section problem
- n Simplest is mutex lock
- n Protect critical regions with it by first **acquire()** a lock then **release()** it
  - | Boolean variable indicating if lock is available or not
- n Calls to **acquire()** and **release()** must be atomic
  - | Usually implemented via hardware atomic instructions
- n But this solution requires **busy waiting**
- n **Busy Waiting means** busy-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available
- n Mutex lock therefore called a **spinlock**



# acquire() and release()

- `acquire() {`  
    `while (!available)`  
        `; /* busy wait */`  
    `available = false;`  
}
- `release() {`  
    `available = true;`  
}
- `do {`  
    `acquire lock`  
    `critical section`  
    `release lock`  
    `remainder section`  
`} while (true);`



# Pthreads Synchronization – Mutex Lock

```
#include <pthread.h>
Pthread_mutex_t mutex;

/* create the mutex lock */
Pthread_mutex_init(&mutex, NULL);
```

1<sup>st</sup> Arg: Mutex initilizer

2<sup>nd</sup> Arg: Attributes, NULL means no error checks will be performed

# Pthreads Synchronization – Mutex Lock

- ▶ The mutex is acquired and released with the `pthread_mutex_lock()`
- ▶ and `pthread_mutex_unlock()` functions.
- ▶ If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`.

# What is a Semaphore

- ▶ a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.

# Semaphore

- ▶ Synchronization tool that does not require busy waiting
- ▶ Semaphore  $S$  – integer variable
- ▶ Two standard operations modify  $S$ : `wait()` and `signal()`
- ▶ Less complicated
- ▶ Can only be accessed via two indivisible (atomic) operations

**Definition of `wait()` operation:**

```
wait (S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

**Definition of `signal()` operation:**

```
signal (S) {  
    S++;  
}
```

- ▶ when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

# Semaphore Usage

## Counting Semaphore:

- ▶ Semaphores which allow an arbitrary resource count are called **counting semaphores**,
- ▶ Counting semaphores can be used to control access to a given resource consisting of a finite number of instances
- ▶ Can implement a counting semaphore  $S$  as a binary semaphore

## Binary Semaphore:

- ▶ while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.
- ▶ Binary semaphores behave similarly to mutex locks.
- ▶ Can implement a counting semaphore  $S$  as a binary semaphore

# Semaphore Usage

- ▶ consider two concurrently running processes:  $P_1$  with a statement  $S_1$  and  $P_2$  with a statement  $S_2$ .
- ▶ It is required that  $S_2$  be executed only after  $S_1$  has completed. We can implement this scheme readily by letting  $P_1$  and  $P_2$  share a common semaphore  $\text{synch}$ , initialized to 0.

$P_1$ :

```
 $S_1;$   
signal(synch);
```

$P_2$ :

```
wait(synch);  
 $S_2;$ 
```

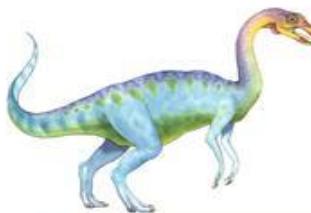
- ▶ Because  $\text{synch}$  is initialized to 0,  $P_2$  will execute  $S_2$  only after  $P_1$  has invoked  $\text{signal}(\text{synch})$ , which is after statement  $S_1$  has been executed.

# Semaphore Implementation

- ▶ Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- ▶ Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
- ▶ Note that applications may spend lots of time in critical sections and therefore this is not a good solution

# Semaphore Implementation with no Busy waiting

- ▶ With each semaphore there is an associated waiting queue
- ▶ Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue



## Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```



# POSIX – Semaphores

```
#include <semaphore.h>
Sem_t sem;
/* Create the semaphore and initialize it to 1 */
```

```
Sem_init(&sem, 0, 1);
```

The `sem_init()` function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

# POSIX – Semaphores

```
/* acquire the semaphore */
```

```
Sem_wait(&sem);
```

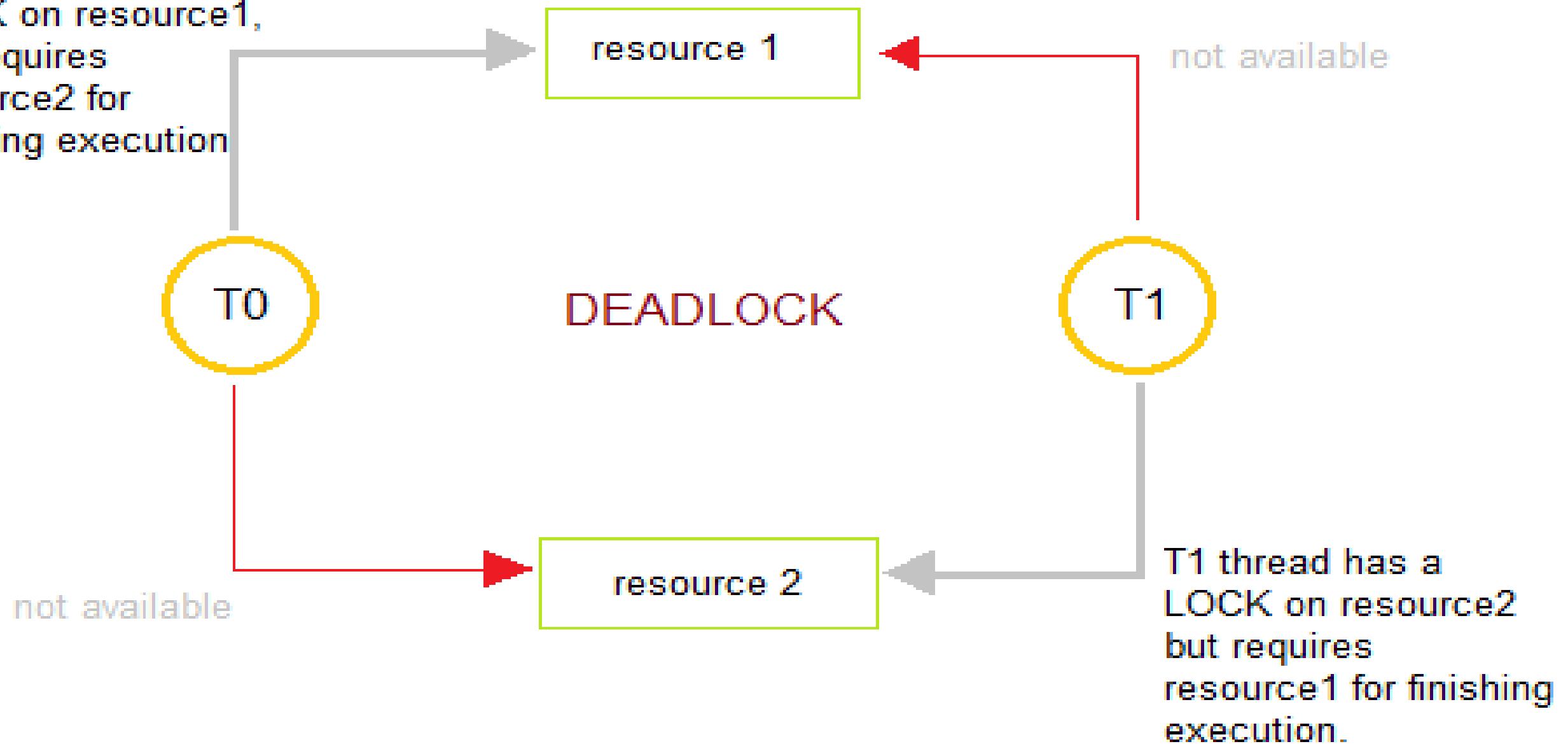
```
/* critical section */
```

```
/* release the semaphore */
```

```
Sem_post(&sem);
```

# Deadlock

T0 thread has a  
LOCK on resource1,  
but requires  
resource2 for  
finishing execution



# Starvation

- ▶ **Starvation** is the name given to the indefinite postponement of a process because it requires some resource before it can run, but the resource, though available for allocation, is never allocated to this process.

# Deadlock Vs. Starvation

- ▶ Deadlock refers to the situation when processes are stuck in circular waiting for the resources.
- ▶ On the other hand, starvation occurs when a process waits for a resource indefinitely.

# Priority Inversion

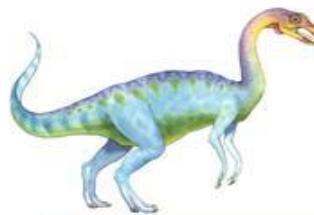
- ▶ **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

# Priority Inheritance Protocol

- ▶ when a job blocks one or more high-priority jobs, it ignores its original priority assignment and executes its critical section at an elevated priority level.
- ▶ After executing its critical section and releasing its locks, the process returns to its original priority level

# Classical Problems of Synchronization

- ▶ Classical problems used to test newly-proposed synchronization schemes
  - Bounded Buffer Problem
  - Dining–Philosophers Problem
  - Readers and Writers Problem

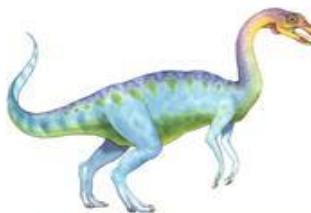


# Bounded-Buffer Problem

---

- $n$  buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value  $n$



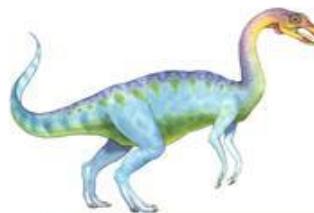


# Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```





# Bounded Buffer Problem (Cont.)

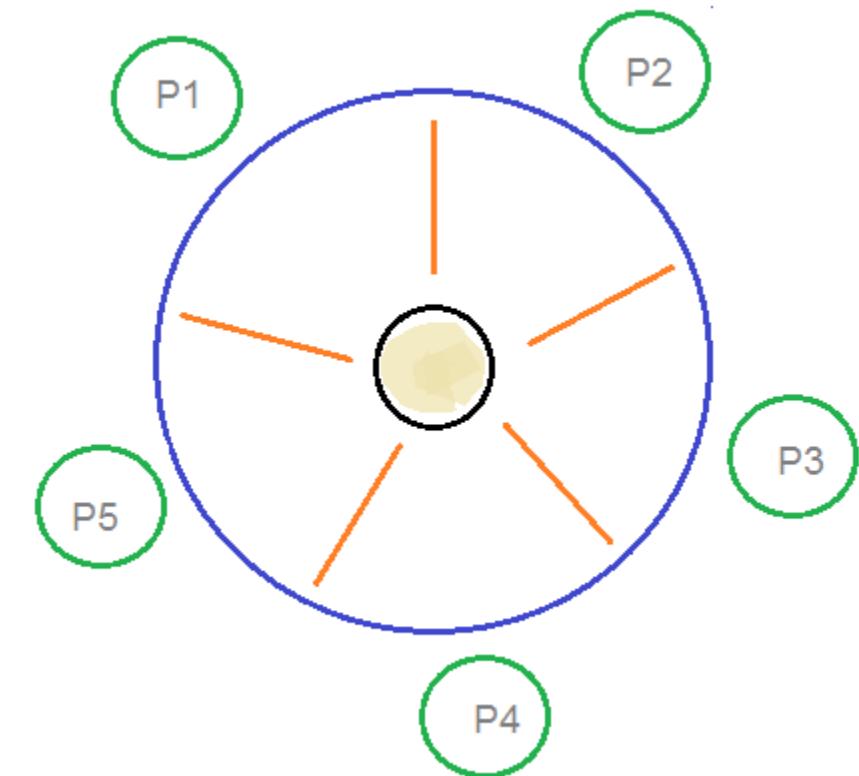
- The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```



# Dining-Philosophers Problem

- Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice.



# Dining-Philosophers Solution

- ▶ it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time

# Dining-Philosophers Solution 1

- ▶ An array of five semaphores, **stick[5]**, for each of the five chopsticks.
- ▶ The code for each philosopher looks like:

```
while(TRUE) {  
    wait(stick[i]);  
    wait(stick[(i+1) % 5]); // mod is used because if i=5, next  
                           // chopstick is 1 (dining table is circular)  
    /* eat */  
    signal(stick[i]);  
    signal(stick[(i+1) % 5]);  
    /* think */  
}
```

# Dining-Philosophers Solution 1

- ▶ When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick.
- ▶ Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down

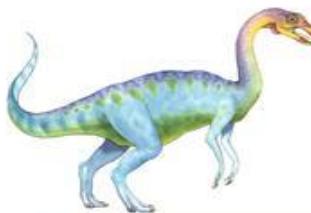
# Dining-Philosophers Problem

- ▶ But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs:

*Deadlock implies starvation but starvation does not imply deadlock*

# Dining-Philosophers Solution 2

- ▶ Two Possible solutions are :
- ▶ A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- ▶ Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.



# Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0





# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```





# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```



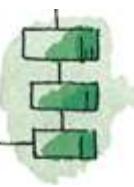
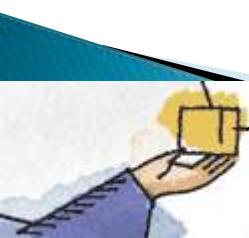
# **Deadlock**

**Course Instructor: Nausheen Shoaib**



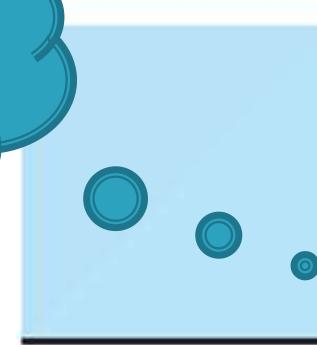
# Deadlock

- ▶ A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
  - Typically involves processes competing for the same set of resources
- ▶ No efficient solution

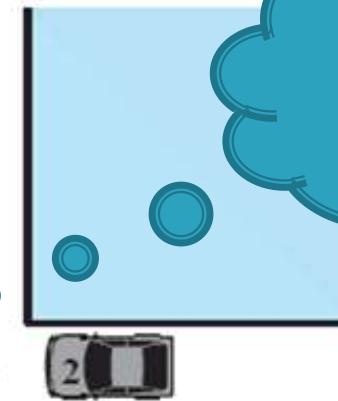


# Potential Deadlock

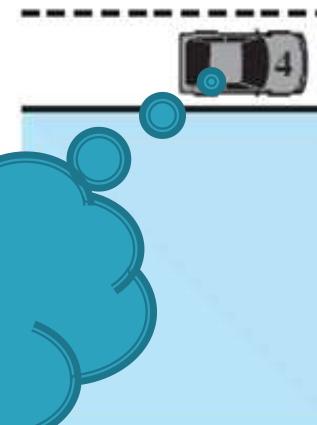
I need  
quad C  
and B



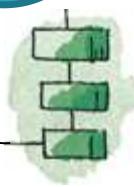
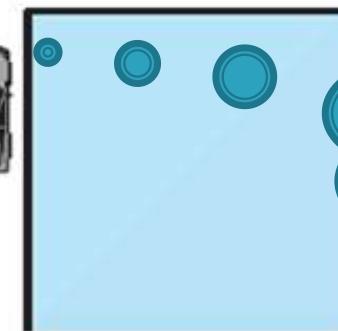
I need  
quad B  
and C



I need  
quad D  
and A

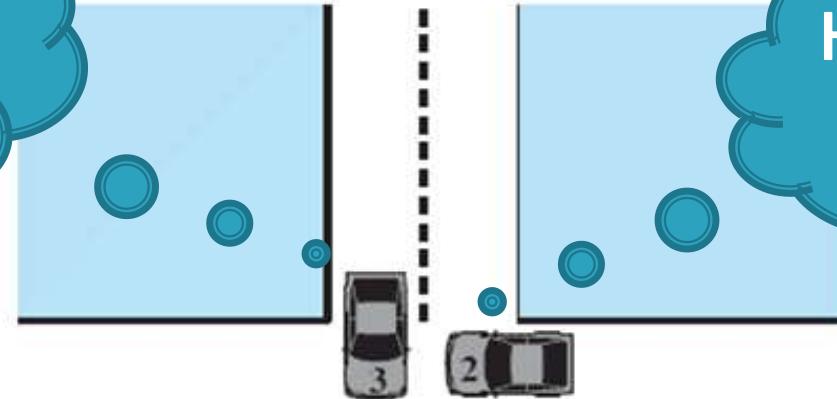


I need  
quad A  
and B

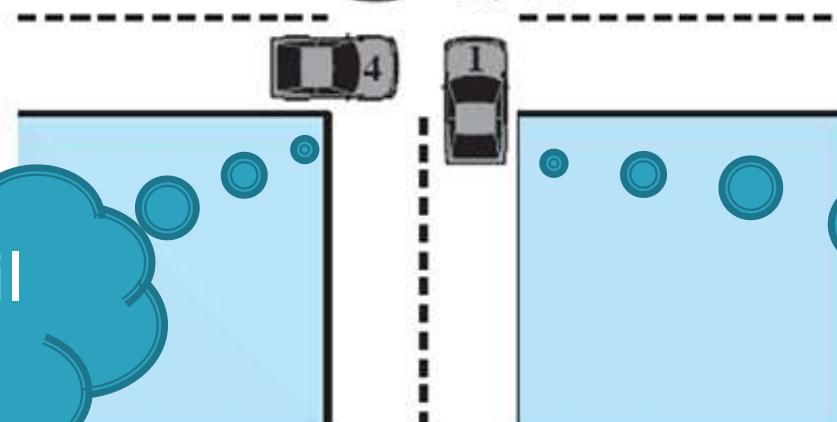


# Actual Deadlock

HALT until  
D is free

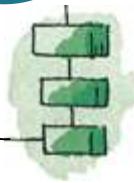


HALT until  
C is free



HALT until  
A is free

HALT until  
B is free



# Resource Allocation Graphs

- ▶ Directed graph that depicts a state of the system of resources and processes

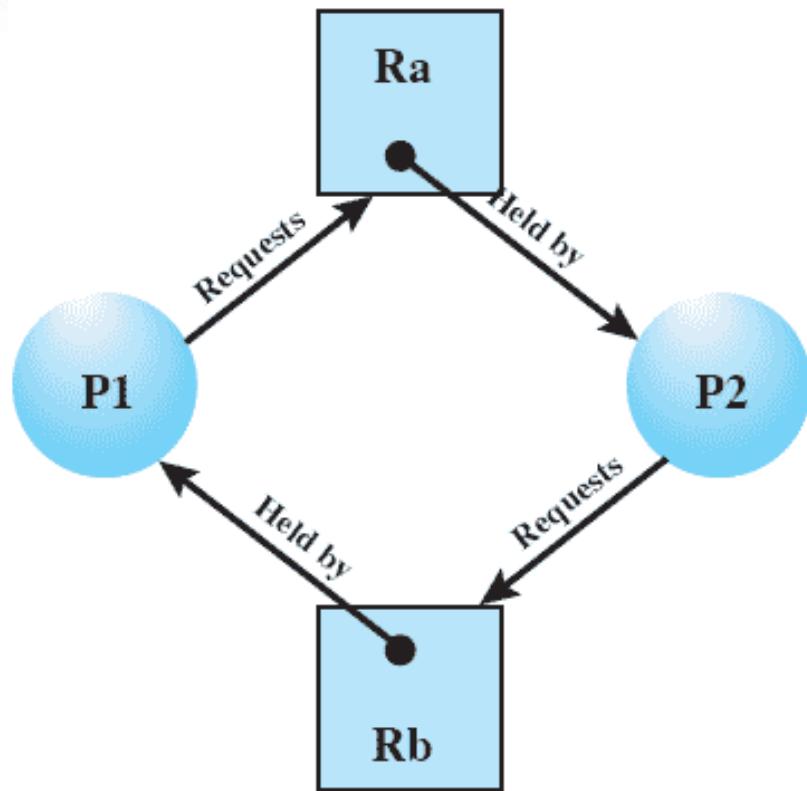


(a) Resource is requested

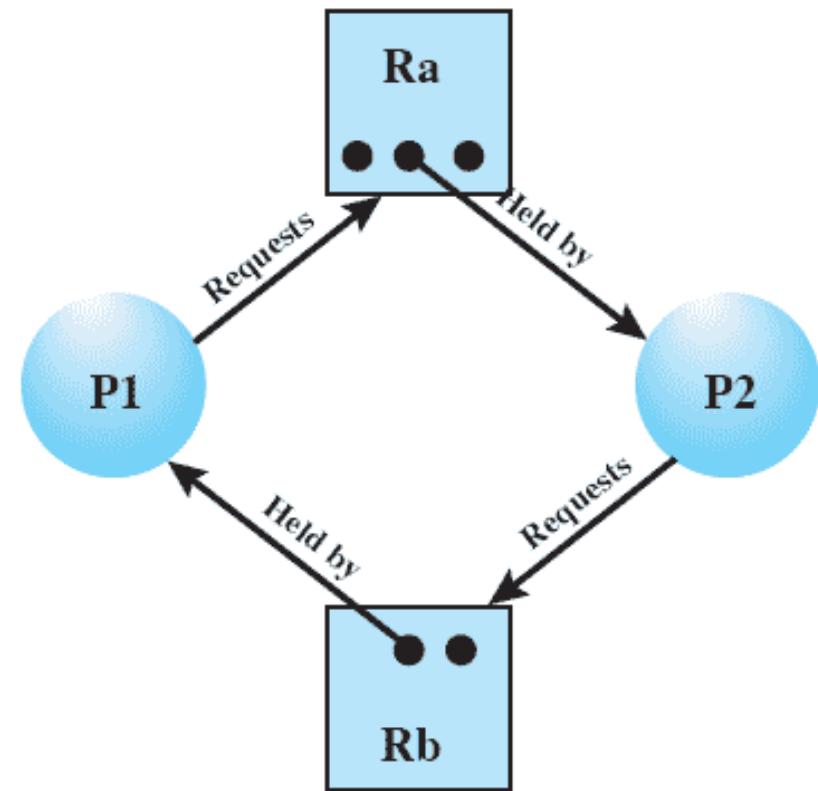


(b) Resource is held

# Resource Allocation Graphs of deadlock



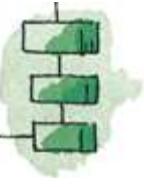
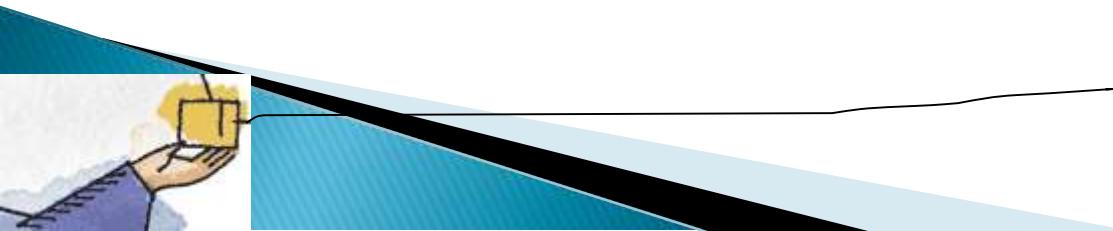
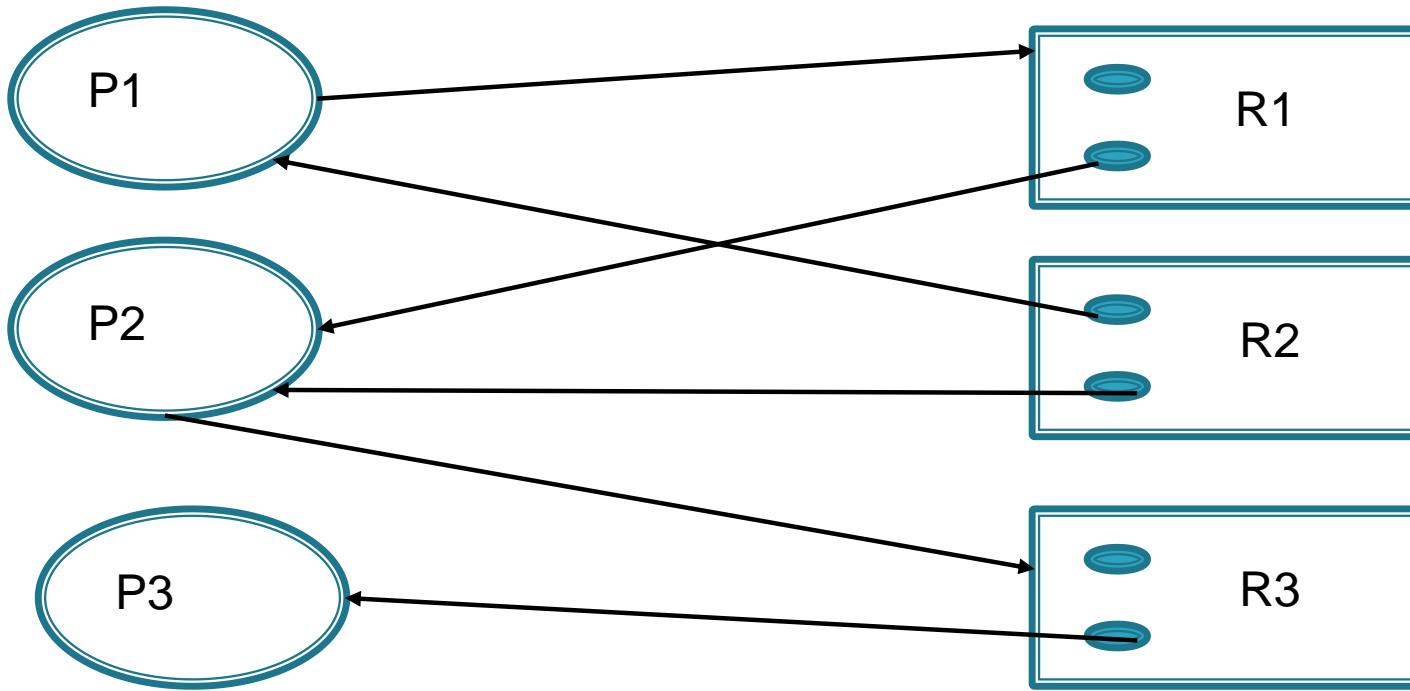
(c) Circular wait



(d) No deadlock

# Resource Allocation Graphs of deadlock

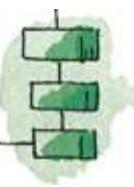
- ▶ Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**





# Dealing with Deadlock

- ▶ Three general approaches exist for dealing with deadlock.
  - Prevent deadlock
  - Avoid deadlock
  - Detect Deadlock

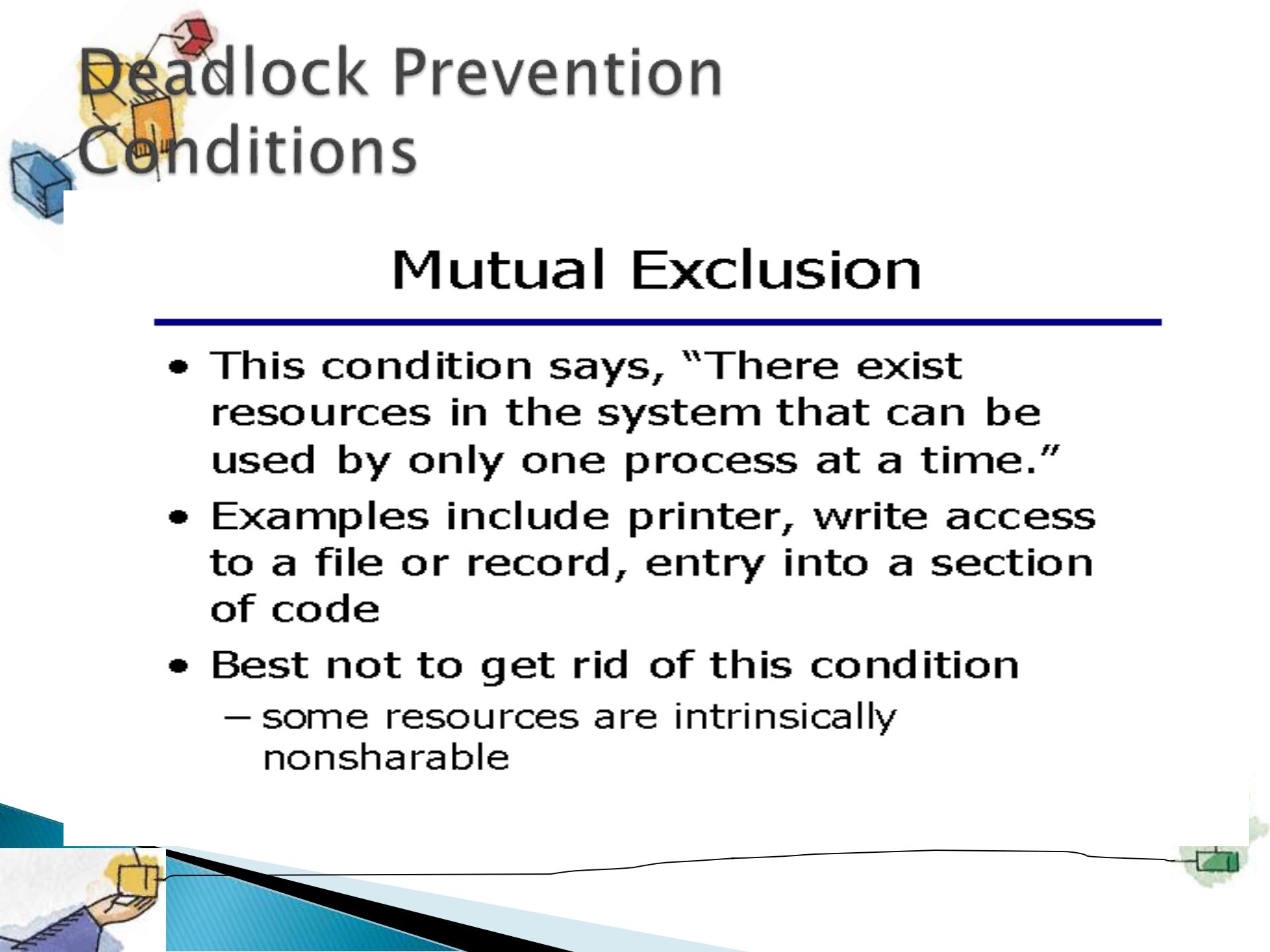


# Deadlock Prevention Strategy

- ▶ Design a system in such a way that the possibility of deadlock is excluded.
- ▶ Two main methods
  - Indirect – prevent one of the three necessary conditions from occurring
  - Direct – prevent circular waits

# Deadlock Prevention Conditions

- ▶ Mutual Exclusion
- ▶ Hold and Wait
- ▶ No Preemption
- ▶ Circular Wait



# Deadlock Prevention Conditions

## Mutual Exclusion

---

- This condition says, “There exist resources in the system that can be used by only one process at a time.”
- Examples include printer, write access to a file or record, entry into a section of code
- Best not to get rid of this condition
  - some resources are intrinsically nonsharable

# Deadlock Prevention Conditions

## Hold and Wait (1/2)

---

- This condition says, “Some process holds one resource while waiting for another.”
- To attack the hold and wait condition:
  - Force a process to acquire all the resources it needs before it does anything; if it can’t get them all, get none
- Each philosopher tries to get both chopsticks, but if only one is available, put it down and try again later

# Deadlock Prevention Conditions

## No Preemption (1/2)

---

- This condition says, “Once a process has a resource, it will not be forced to give it up.”
- To attack the no preemption condition:
  - If a process asks for a resource not currently available, block it but also take away all of its other resources
  - Add the preempted resources to the list of resource the blocked process is waiting for

# Deadlock Prevention Conditions

## Circular Wait (1/2)

---

- This condition says, “A is blocked waiting for B, B for C, C for D, and D for A”
- Note that the number of processes is actually arbitrary
- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order

# Deadlock Avoidance

- ▶ A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- ▶ Requires knowledge of future process requests

# Two Approaches to Deadlock Avoidance

- ▶ Process Initiation Denial
- ▶ Resource Allocation Denial

# Process

## Initiation Denial

- ▶ A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- ▶ Not optimal,
  - Assumes the worst: that all processes will make their maximum claims together.

# Resource

## Allocation Denial

- ▶ Referred to as the banker's algorithm
  - A strategy of resource allocation denial
- ▶ Consider a system with fixed number of resources
  - ***State*** of the system is the current allocation of resources to process
  - ***Safe state*** is where there is at least one sequence that does not result in deadlock
  - ***Unsafe state*** is a state that is not safe

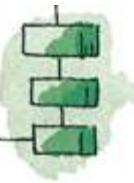
# Basic Facts for deadlock avoidance

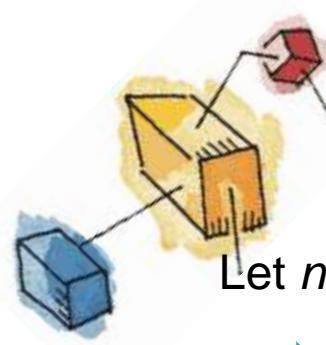
- ▶ If a system is in safe state  $\Rightarrow$  no deadlocks
- ▶ If a system is in unsafe state  $\Rightarrow$  possibility of deadlock
- ▶ Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state.



# Banker's Algorithm

- ▶ Multiple instances
- ▶ Each process must a priori claim maximum use
- ▶ When a process requests a resource it may have to wait
- ▶ When a process gets all its resources it must return them in a finite amount of time



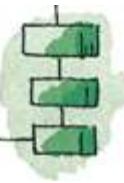


# Data Structures for the Banker's Algorithm

Let  $n$  = number of processes, and  $m$  = number of resources types.

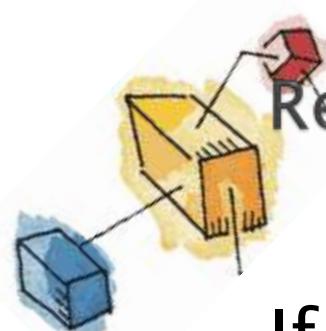
- ▶ **Available:** Vector of length  $m$ . If  $\text{available}[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- ▶ **Max:**  $n \times m$  matrix. If  $\text{Max}[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- ▶ **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- ▶ **Need:**  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$



# Example of Banker's Algorithm

- ▶ Discussed in Class



## Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $P_i$ .  
If  $\text{Request}_{i,j} = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

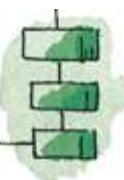
1. If  $\text{Request}_i \leq \text{Need}_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3.  
Otherwise  $P_i$  must wait, since resources are not available
3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- If safe  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored





# Example of Resource Request Algorithm

- ▶ Discussed in Class

