

# Module1\_Introduction

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

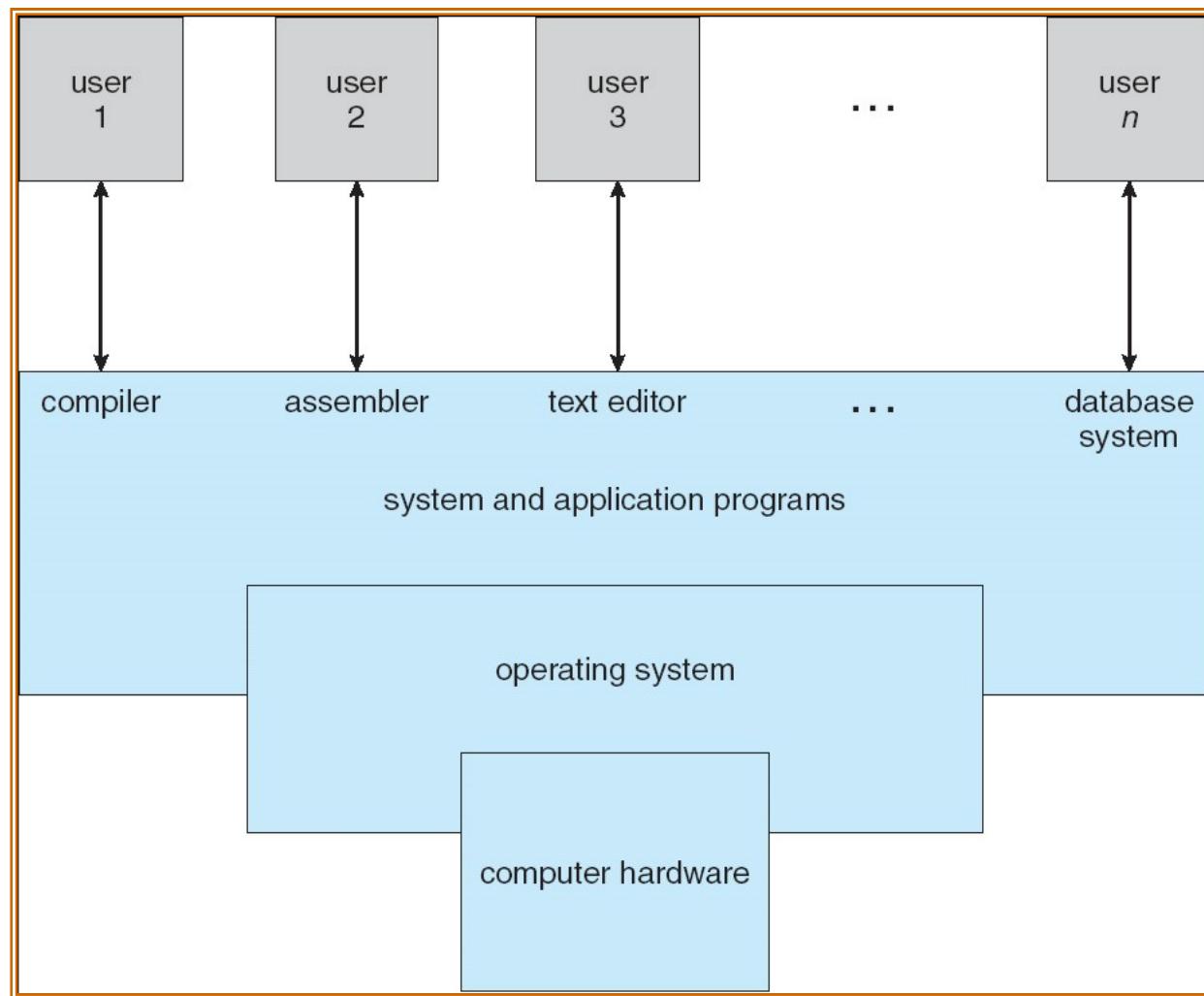
# What is an Operating System?

- A program that acts as an **intermediary** between a user of a computer and the computer hardware.
- Operating system goals:
  - Execute user programs and make **solving user problems easier**.
  - Make the computer system **convenient to use**.
- Use the computer hardware in an efficient manner.

# Computer System Structure

- Computer system can be divided into four components
  - **Hardware** – provides basic computing resources
    - CPU, memory, I/O devices
  - **Operating system**
    - Controls and coordinates use of hardware among various applications and users
  - **System and Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - **Users**
    - People, machines, other computers

# Four Components of a Computer System



# What Operating Systems Do

- The operating system **controls the hardware** and coordinates its use among the various application programs for the various users.
- We can also view a **computer system as consisting of hardware, software, and data.**
- The operating system provides the **means for proper use of these resources** in the operation of the computer system.
- An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an **environment** within which other programs can do useful work.
- To understand more fully the operating system's role, the concept of operating systems is explored in **two viewpoints:**
  - The user view
  - The system view

# User View

- The user's view of the computer varies according to the interface being used
  - **Single user computers** (e.g., PC, workstations). Such systems are designed for one user to monopolize its resources.
    - The goal is to **maximize the work** (or play) that the user is performing. the operating system is designed mostly for **ease of use** and **good performance**.
  - **Multi user computers** (e.g., mainframes, computing servers). These users share resources and may exchange information.
    - The operating system in such cases is designed to **maximize resource utilization** -- to assure that all available CPU time, memory, and I/O are used efficiently and that no individual users takes more than their fair share.

# User View (Cont'd)

- **Handheld computers** (e.g., smartphones and tablets). The user interface for mobile computers generally features a **touch screen**. The systems are resource poor, optimized for usability and battery life.
- **Embedded computers** (e.g., computers in home devices and automobiles) The user interface may have numeric keypads and may turn indicator lights on or off to show status. The operating systems are designed primarily to run without user intervention.

# System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. There are two different views:

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

# 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

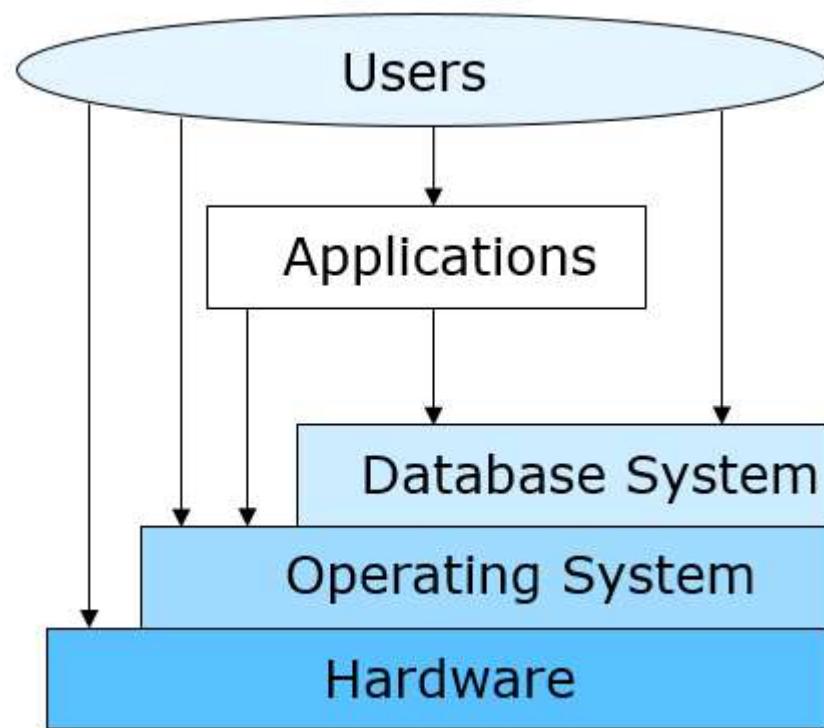
# Operating System Definition (Cont'd)

- Operating systems exist to offer a reasonable way to solve the problem of creating a usable computing system.
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- Since bare hardware alone is not particularly easy to use, application programs are developed.
  - These programs require certain common operations, such as those controlling the I/O devices.
  - The common functions of controlling and allocating resources are brought together into one piece of software: the **operating system**.

# Operating System Definition (Cont.)

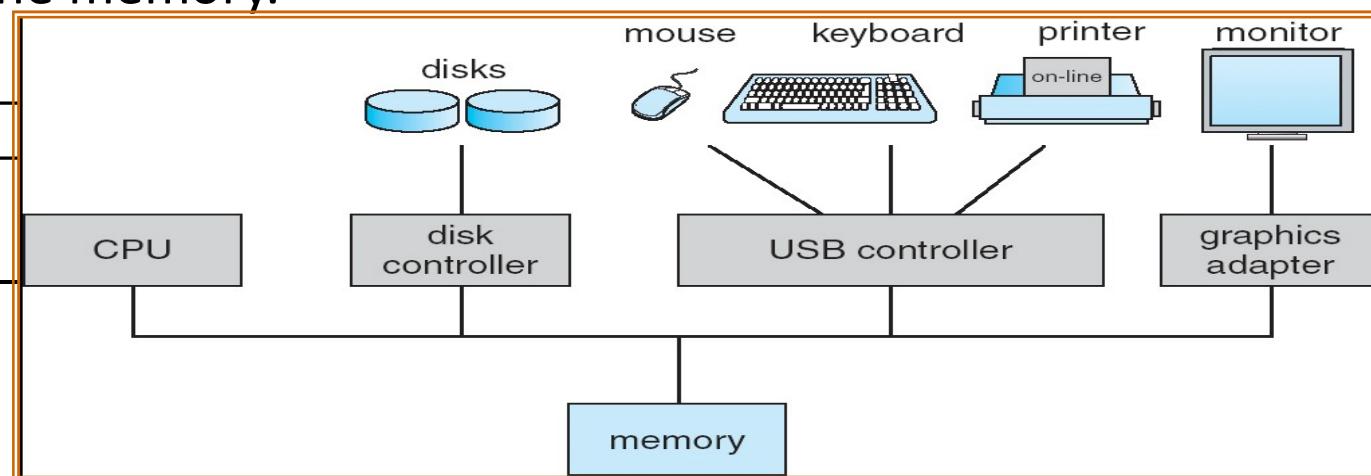
- No universally accepted definition
- A simple viewpoint is that it includes everything a vendor ships when you order the operating system. The features that are included vary greatly across systems:
  - Some systems take up less than a **megabyte of space** and lack even a full-screen editor
  - Some systems require **gigabytes of space** and are based entirely on graphical windowing systems.
- “The one program running at all times on the computer” is the **OS kernel**.
- Along with the kernel, there are two other types of programs:
  - **System programs**, which are associated with the operating system but are not necessarily part of the kernel.
  - **Application programs**, which include all programs not associated with the operation of the system.
- Kernel is the **central core/ fundamental part** of modern OS

# Evolution of Computer Systems



# Computer System Organization

- Computer-system
  - One or more CPUs, **device controllers** connect through common bus providing access to shared memory
- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays).  
**Each device controller has a local buffer.**
- CPU moves data from/to main memory to/from local buffers.
- The **CPU and the device controllers can execute in parallel**, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.



# 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*.

# Computer Startup

- **Bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM /EEPROM/Flash memory, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

# Interrupts

- There are two types of interrupts:
  - **Hardware Interrupt** -- a device may trigger an interrupt by sending a signal to the CPU, usually by way of the system bus.
  - **Software Interrupt** -- a program may trigger an interrupt by executing a special operation called a **system call**.
- A software-generated interrupt (sometimes called **trap** or **exception**) is caused either by an error (e.g., divide by zero) or a user request (e.g., an I/O request).
- An operating system is **interrupt driven**.

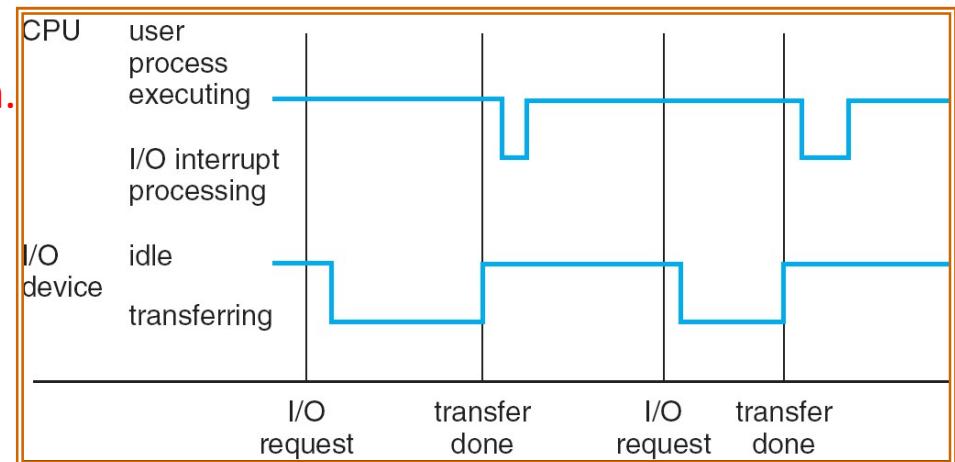
# Interrupts

- Once the kernel is loaded and executing, it can start providing services to the system and its users.
- Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running.
- On UNIX, the first system process is **init** and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.
- The occurrence of an event is usually signaled by an **interrupt**.

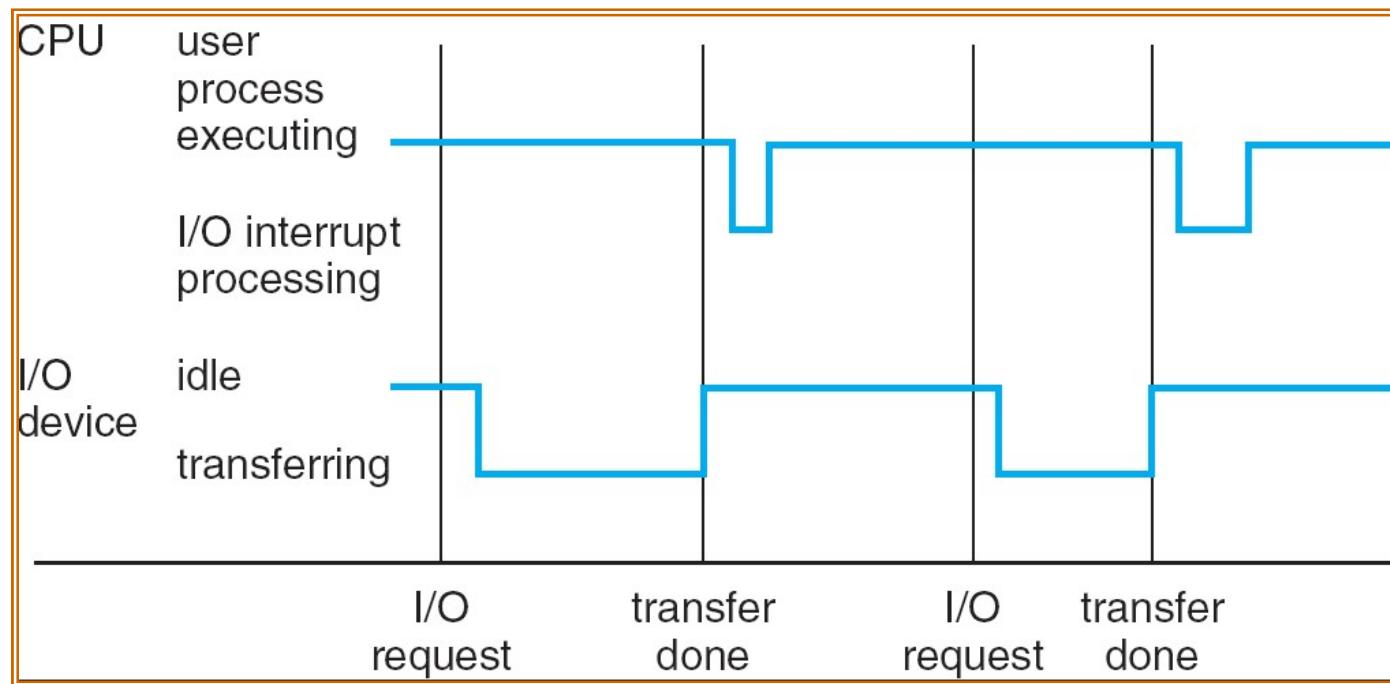
```
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
$ ps -aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
runner        1  3.9  0.0 1385280 12680 ?        SLs1 02:58  0:00 /inject/init
runner       13  0.0  0.0  20192  3924 pts/0    Ss 02:58  0:00 bash --norc
runner       18  0.0  0.0  36092  3372 pts/0    R+ 02:58  0:00 ps -aux
$
```

# Interrupt Handling

- When an interrupt occurs, the operating system
  - preserves the state of the CPU by storing the registers and the program counter
  - Determines which type of interrupt has occurred and transfers control to the appropriate interrupt-service routine (ISR)
- An interrupt-service routine is a collection of routines (modules), each of which is responsible for handling one particular interrupt (e.g., from a printer, from a disk)
- The transfer is generally through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are **disabled** while another interrupt is being processed to prevent a **loss of interrupt**.
- A **trap** is a software-generated interrupt caused either by an error or a user request.
- An operating system is **interrupt driven**.



# Interrupt Timeline



# Intel Pentium processor event-vector table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

# Storage Structure

- Main memory – the only large storage media that the CPU can access directly
  - **Random access**
  - **volatile**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
  - **Hard 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.
  - **Solid-state disks** – faster than hard disks, nonvolatile
- Tertiary storage
  - **Magnetic Tapes**
  - **Optical Disks**

# Storage Definition

- The **basic unit of computer storage** is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits.
- A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage.
- A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes.

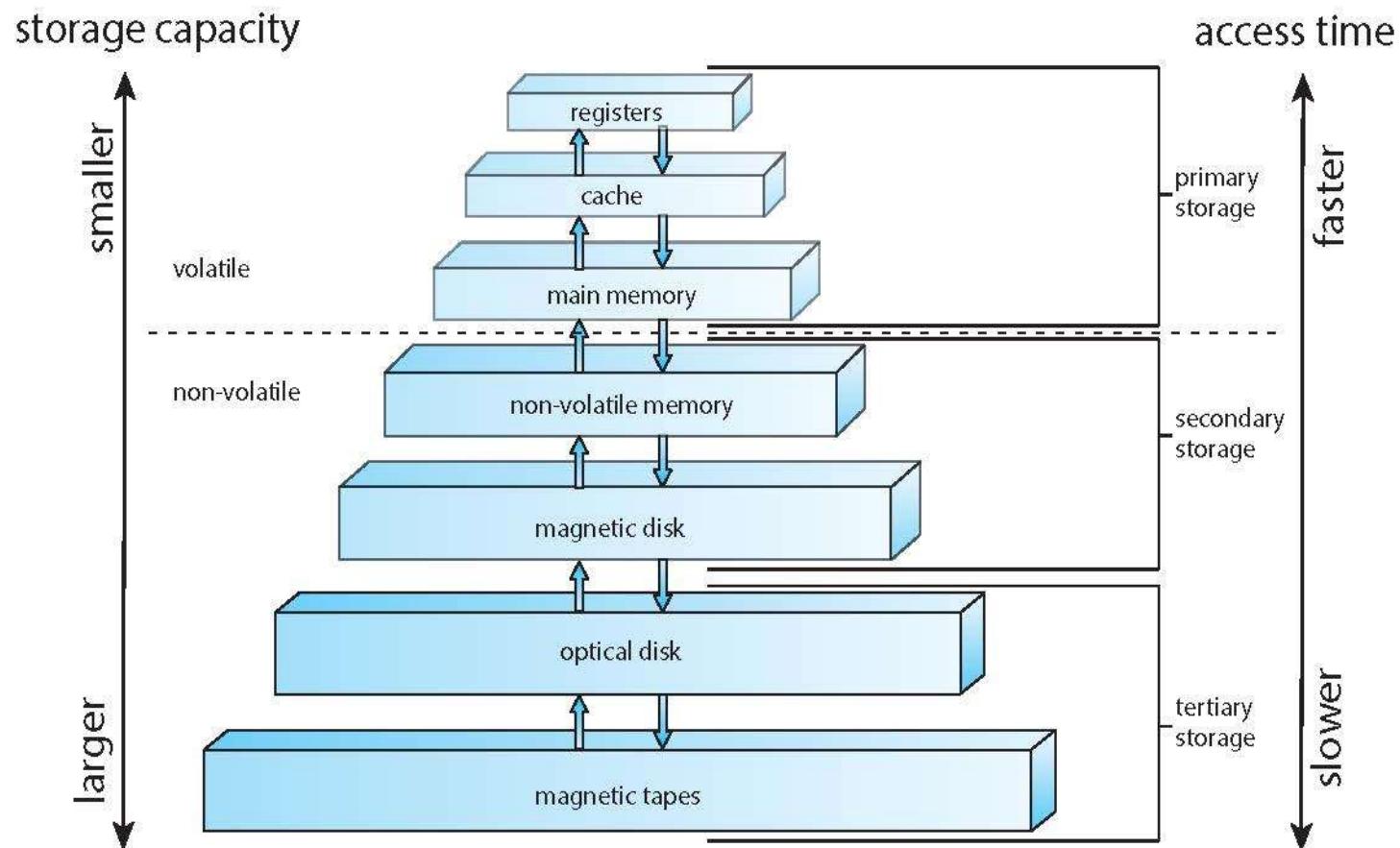
# Storage Definition (Cont.)

- Computer storage is generally measured and manipulated in bytes and collections of bytes.
  - A **kilobyte**, or **KB**, is 1,024 bytes
  - a **megabyte**, or **MB**, is  $1,024^2$  bytes
  - a **gigabyte**, or **GB**, is  $1,024^3$  bytes
  - a **terabyte**, or **TB**, is  $1,024^4$  bytes
  - a **petabyte**, or **PB**, is  $1,024^5$  bytes
  - exabyte, zettabyte, yottabyte
- Computer manufacturers often round off these numbers and say that **a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.** Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

# Storage Hierarchy

- Storage systems are organized in hierarchy.
  - Speed
  - Cost
  - Volatility
- *Caching* – copying information into faster storage system; **main memory can be viewed as a last *cache* for secondary storage.**

# Storage-Device Hierarchy



# Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use is copied from slower to faster storage temporarily
- **Faster storage (cache) checked first to determine if information is there**
  - If it is there, information is used directly from the cache (fast)
  - If not, data is copied to cache and used from there
- Cache is smaller than storage being cached
  - Cache management: important design problem
    - Cache size and replacement policy

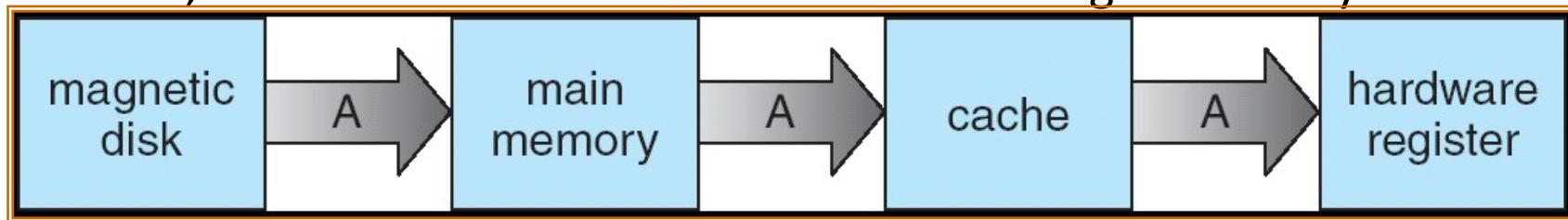
# Performance of Various Levels of Storage

- Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

# Migration of Integer A from Disk to Register

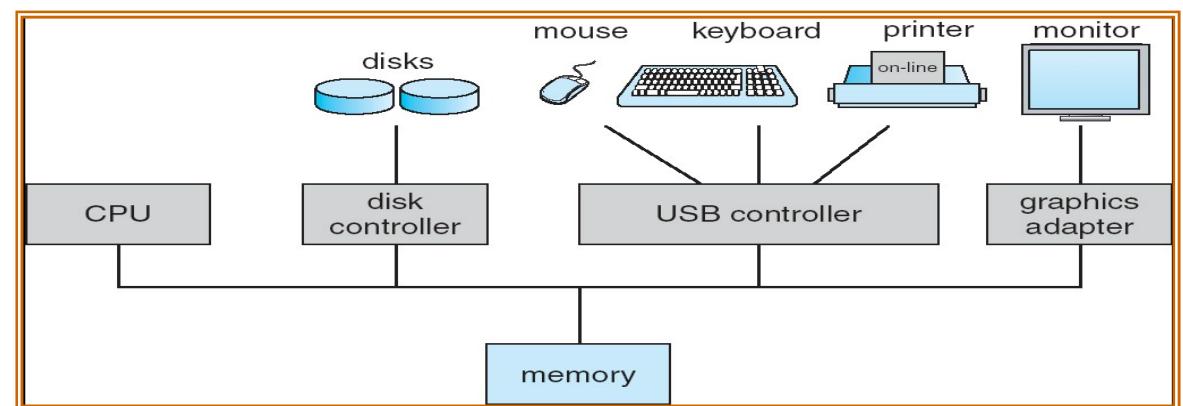
- Multitasking environments must be careful to **use most recent value**, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment -situation is even more complex
  - Several copies of a datum can exist

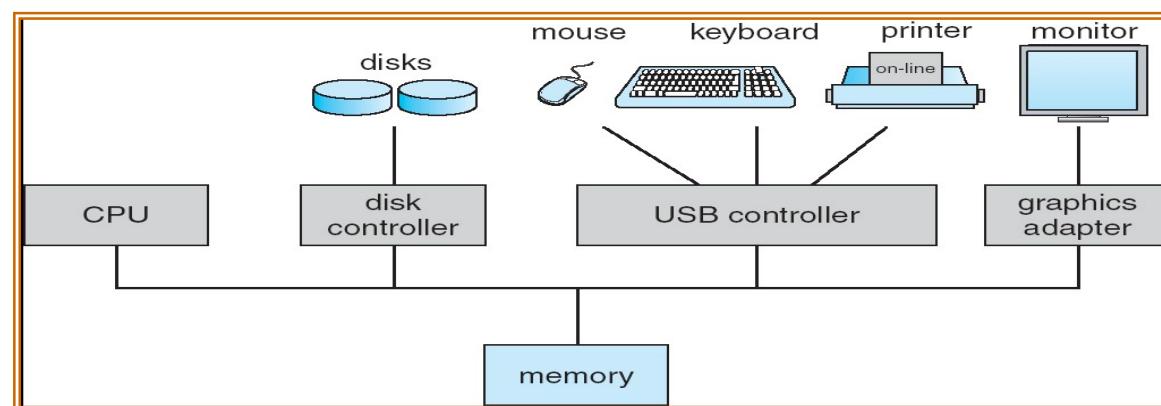
# I/O Structure

- A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each **device controller** is in charge of a specific type of device. More than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller.
- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel



# I/O Structure (Cont'd )

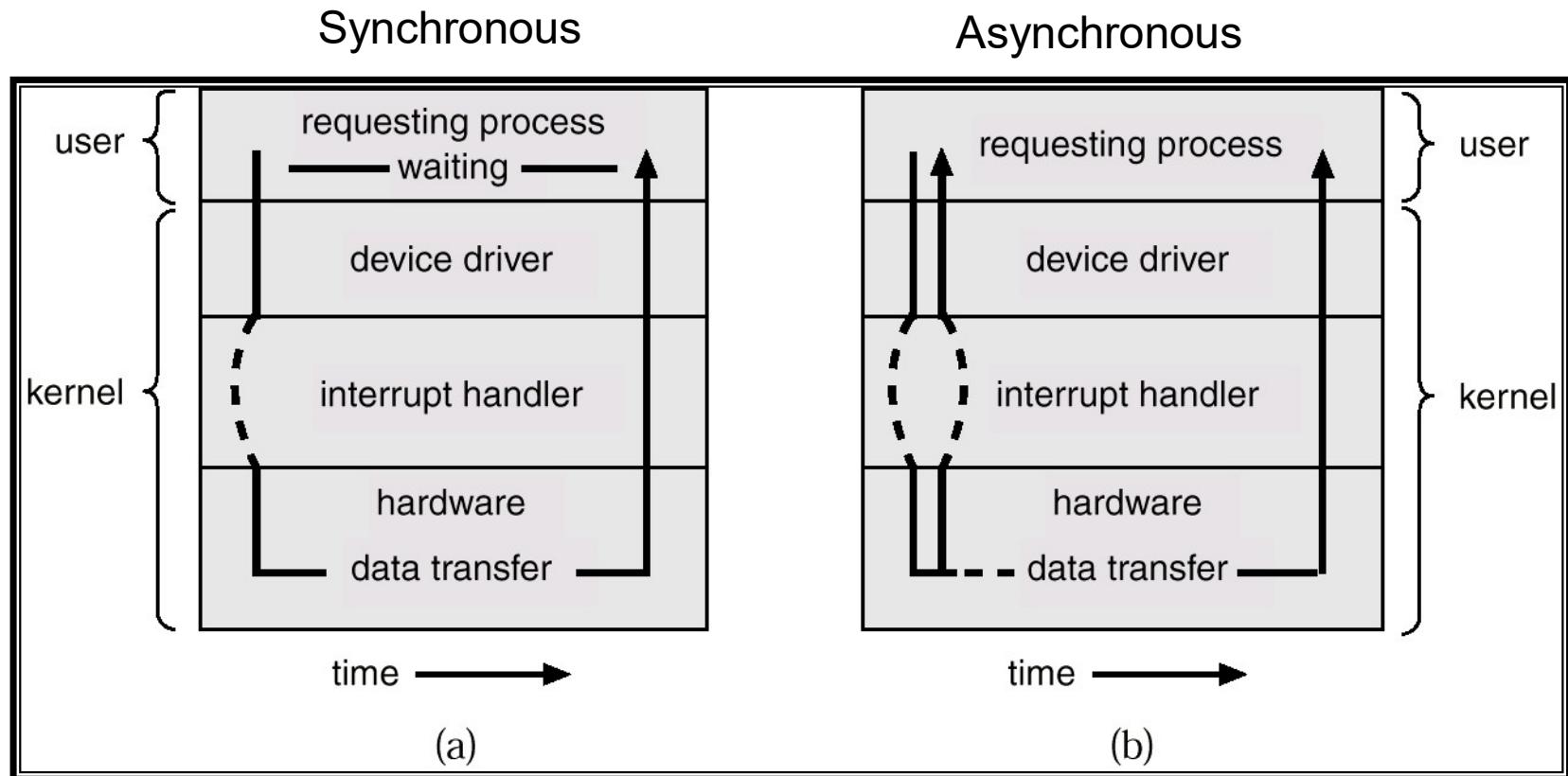
- To start an I/O operation, the device **driver loads the appropriate registers** within the device controller.
- The device controller, in turn, **examines the contents of these registers to determine what action to take** (such as “read” a character from the keyboard).
- The controller **starts the transfer of data** from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation.
- The device **driver then returns control to the operating system**, possibly returning the data or a pointer to the data if the operation was a read.
- For other operations, the device driver returns status information.



# I/O Structure Cont'd

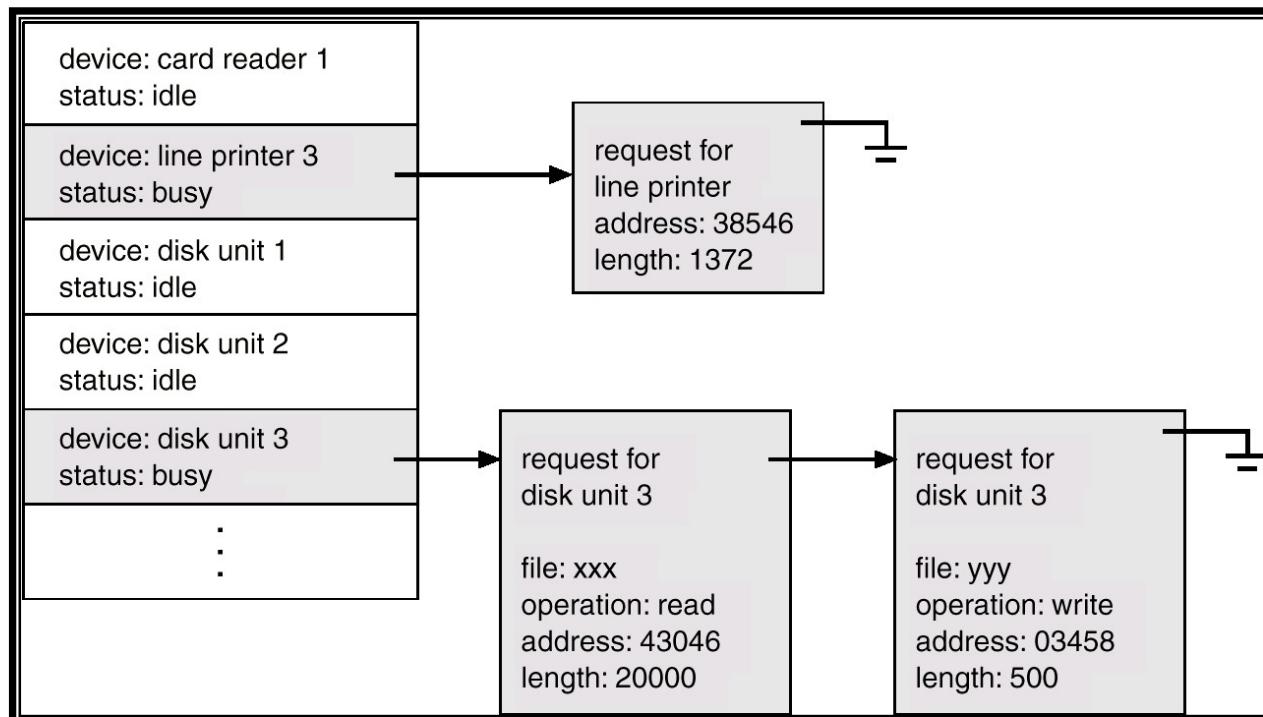
- After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop
  - one I/O request may be outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
  - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

# Two I/O Methods

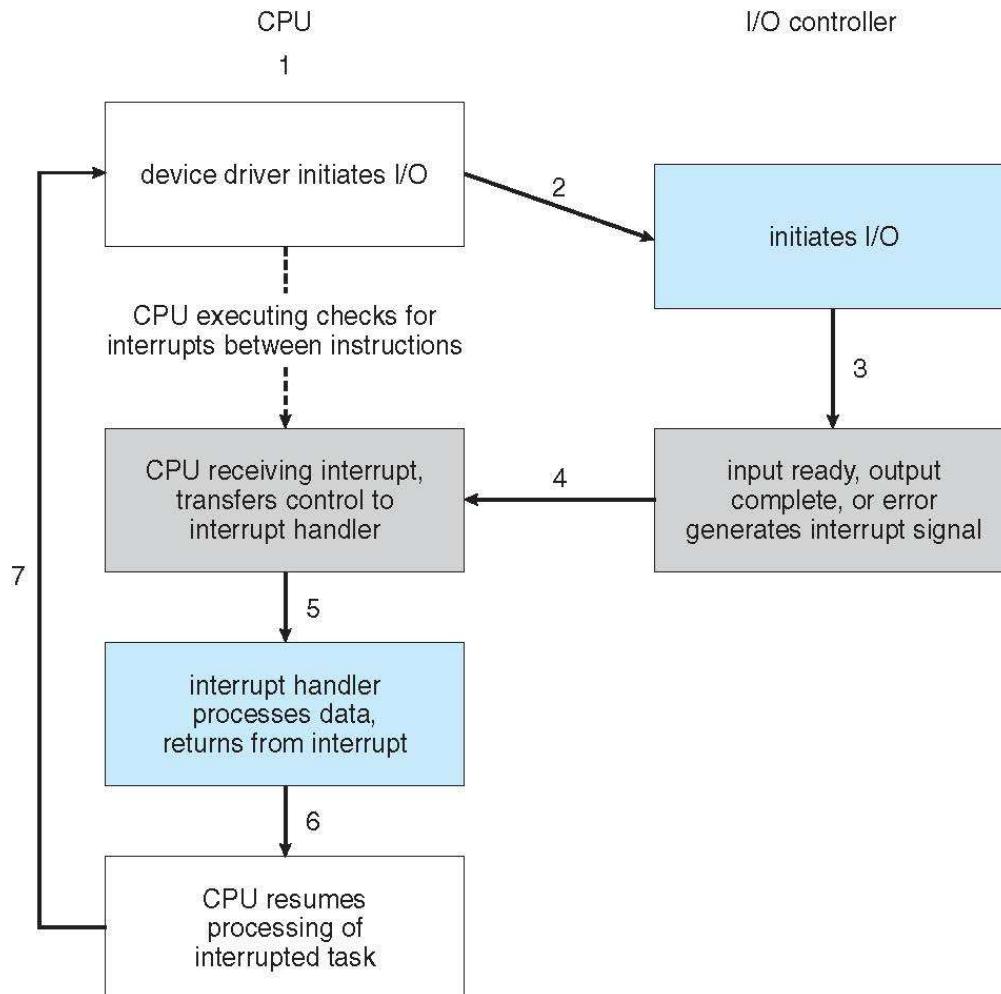


# Device-Status Table

- *Device-status table* contains entry for each I/O device indicating its type, address, and state.
- Operating system **indexes into I/O device table** to determine device status and to modify table entry to include interrupt



# Interrupt-driven I/O cycle



# Direct Memory Access Structure

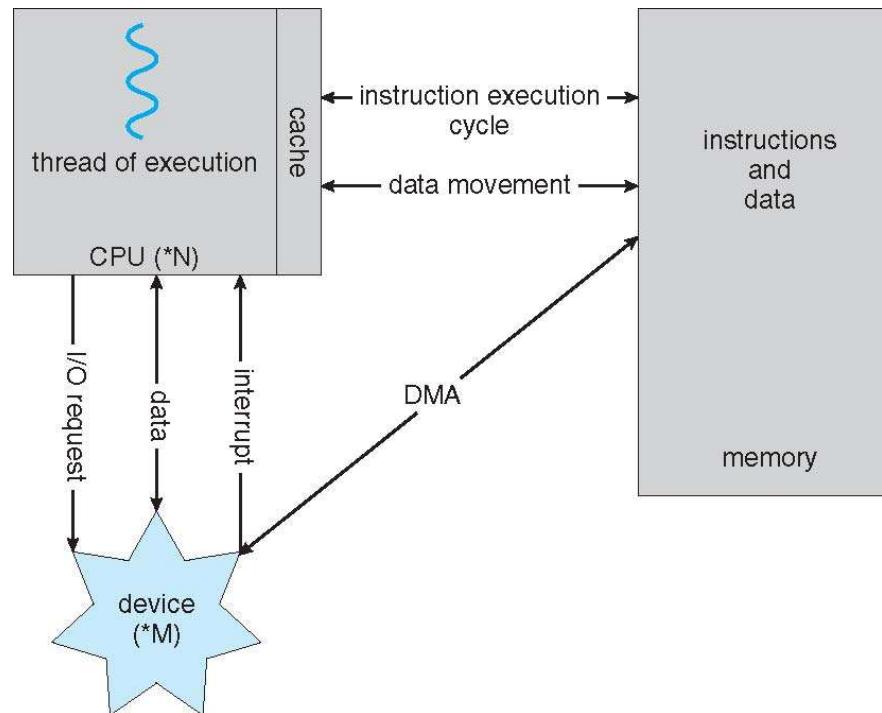
- **Interrupt-driven I/O** is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O.
- To solve this problem, **direct memory access** (DMA) is used.
  - After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.
  - Only one interrupt is generated per block, to tell the device driver that the operation has completed. While the device controller is performing these operations, the CPU is available to accomplish other work.
- Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective.

# Direct Memory Access Structure (Cont'd)

- 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.

# Working of a Modern Computer

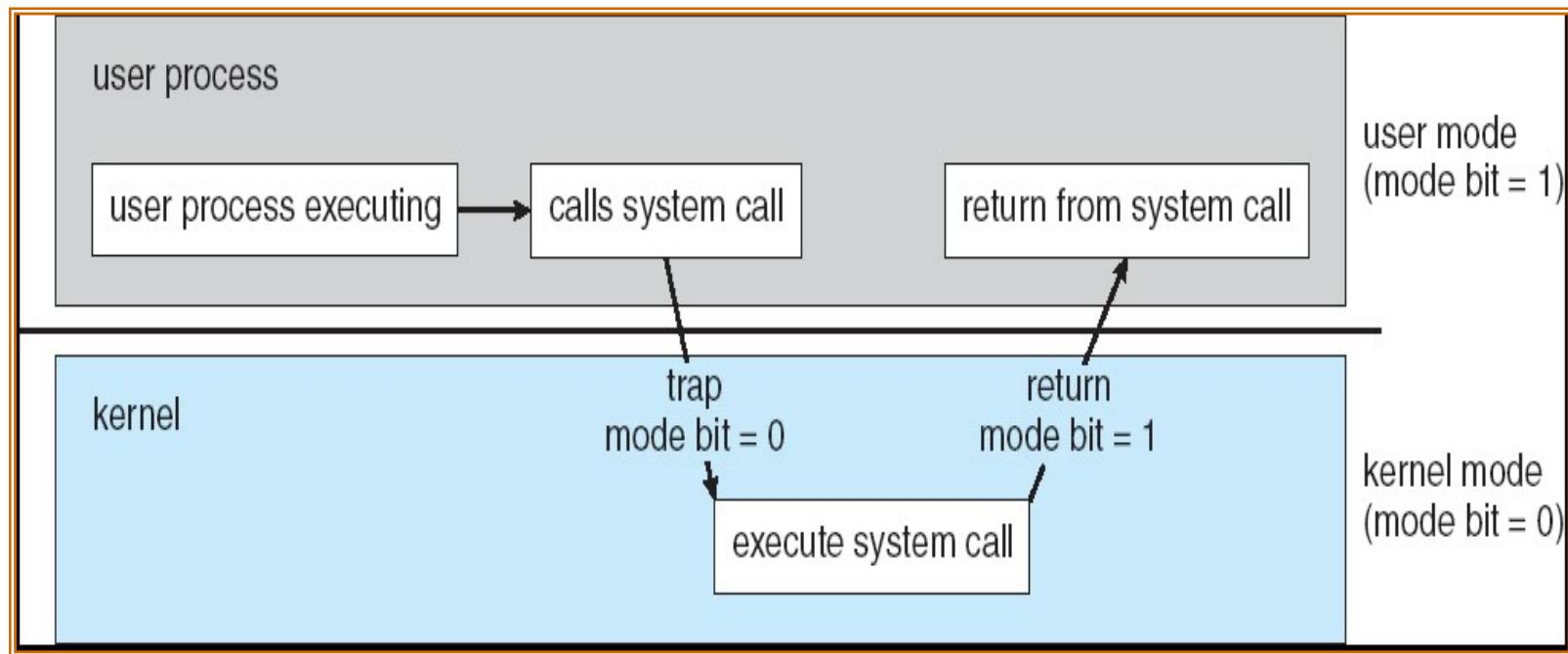
- A Von Neumann architecture and a depiction of the interplay of all components of a computer system.



# Modes of Operation

- A mechanism that allows the OS to protect itself and other system components
- Two modes:
  - **User mode**
  - **Kernel mode**
- Mode bit (0 or 1) 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
  - Systems call by a user asking the OS to perform some function changes from user mode to kernel mode.
  - Return from a system call resets the mode to user mode.
- **Dual-mode** operation allows OS to protect itself and other system components
  - Software error or request creates **exception or trap**
    - Division by zero, request for operating system service
  - Other process problems include infinite loop, processes modifying each other or the operating system

# Transition from User to Kernel Mode



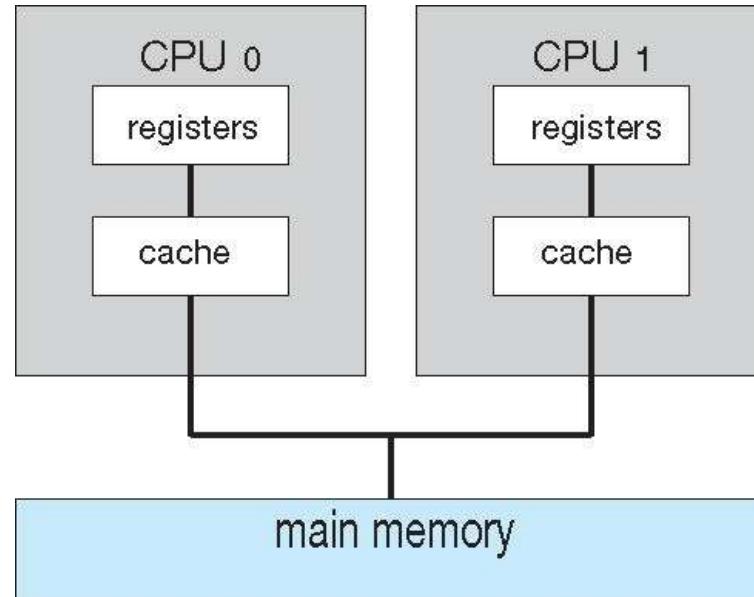
# Timer

- To prevent a process being in infinite loop (process hogging resources), a **timer** is used, which is a hardware device.
- Timer is a counter that is decremented by the physical clock.
- Timer is set to interrupt the computer after some time period
- Operating system sets the counter (privileged instruction)
- When counter reaches the value zero, and interrupt is generated.
- The OS sets up the value of the counter before scheduling a process to regain control or terminate program that exceeds the allotted time.

# Computer-System Architecture

- Single general-purpose processor
  - Most systems have special-purpose processors as well
- Multiprocessors systems
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include:
    - Increased throughput
    - Economy of scale
    - Increased reliability – graceful-degradation/fault-tolerance
  - Two types:
    - Symmetric Multiprocessing – each processor performs all tasks
    - Asymmetric Multiprocessing – each processor is assigned a specific task.

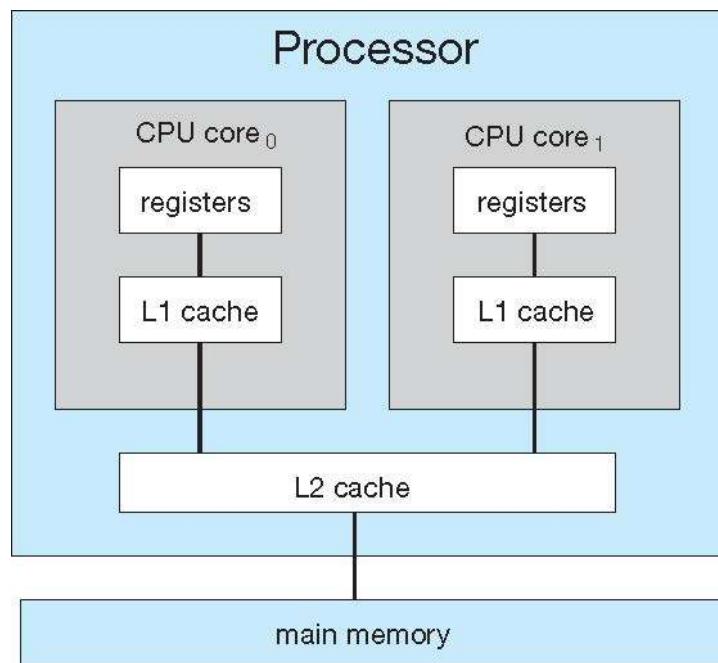
# Symmetric Multiprocessing Architecture



# Multicore Systems

- Most CPU design now includes multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- Multicore systems can be more efficient than multiple chips with single cores because:
  - On-chip communication is faster than between-chip communication.
  - One chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for laptops as well as mobile devices.
- Multicore systems are multiprocessor systems, not all multiprocessor systems are multicore.

## A dual-core with two cores placed on the same chip

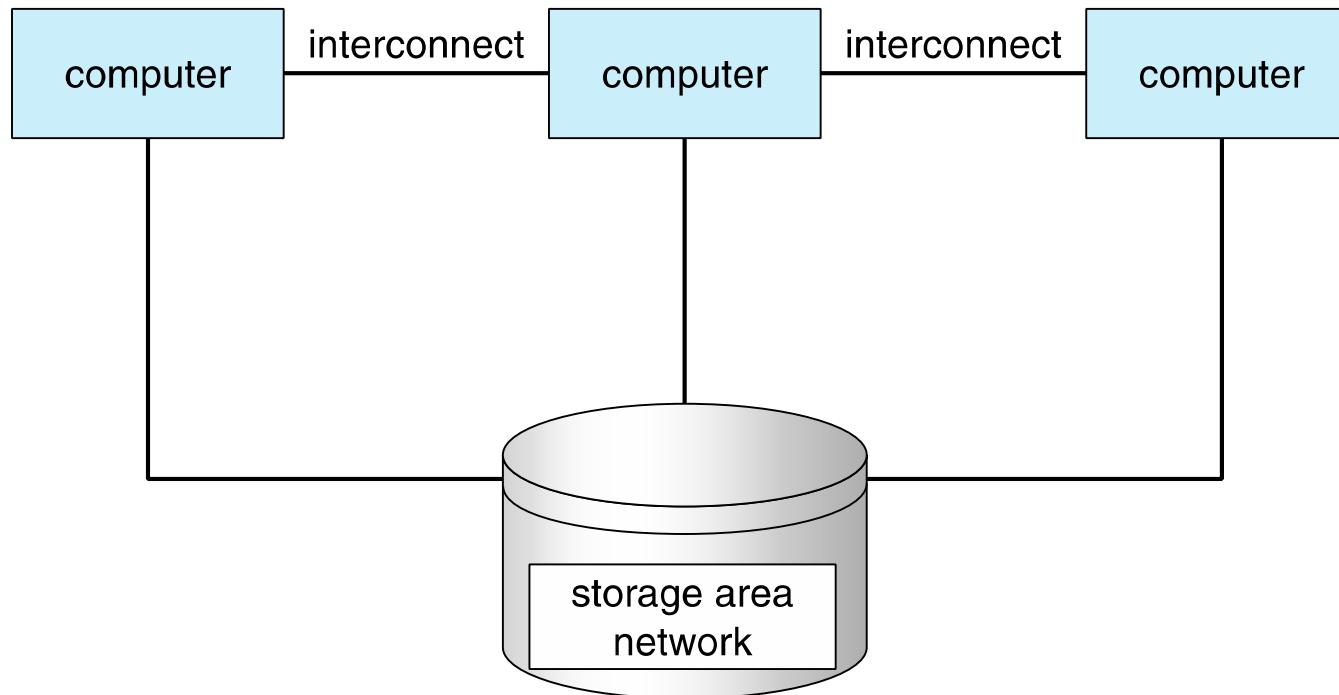


# Clustered Systems

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
- Some clusters are for **high-performance computing (HPC)**
  - Applications must be written to use **parallelization**
- Some have **distributed lock manager (DLM)** to avoid conflicting operations

# Clustered Systems



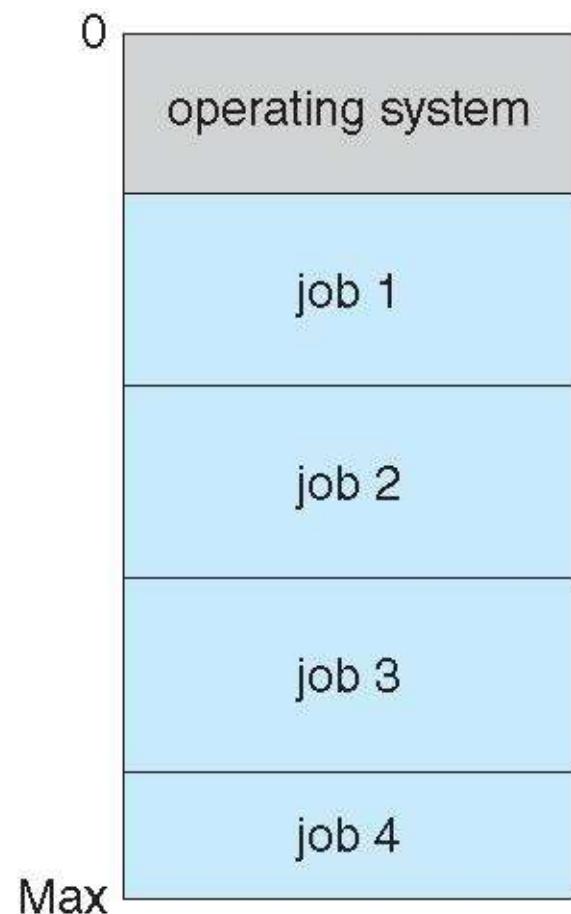
# Multiprogrammed System

- 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
- Batch systems:
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
- Interactive systems:
  - Logical extension of batch systems -- CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

# Interactive Systems

- **Response time** should be < 1 second
- Each user has at least one program executing in memory. Such a program is referred to as a **process**
- If several processes are ready to run at the same time, we need to have **CPU scheduling**.
- If processes do not fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory

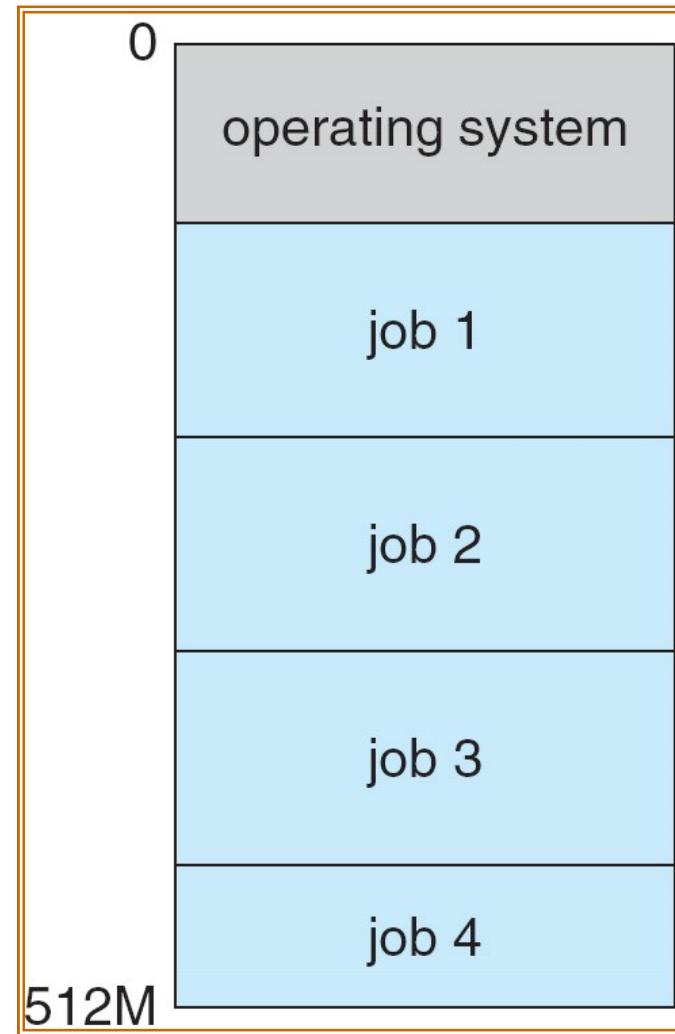
# Memory Layout for Multiprogrammed System



# Operating System 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  $\Rightarrow$  **process**
  - If several jobs are ready to run at the same time  $\Rightarrow$  **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



# 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
- Process **termination** requires **reclaim** of any **reusable resources**
- **Single-threaded** process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- **Multi-threaded** process has **one program counter per thread**
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

# 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
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

# Memory Management

- All **data** in memory before and after processing
- All **instructions** in memory in order to execute
- Memory management determines what is in memory when
  - Optimizing CPU utilization and computer response to users
- 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

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- 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

- Usually disks are used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- 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
  - Varies between WORM (write-once, read-many-times) and RW (read-write)

# I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including
    - **buffering** (storing data temporarily while it is being transferred),
    - **caching** (storing parts of data in faster storage for performance),
    - **spooling** (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - **Drivers** for specific hardware devices

# Protection and Security

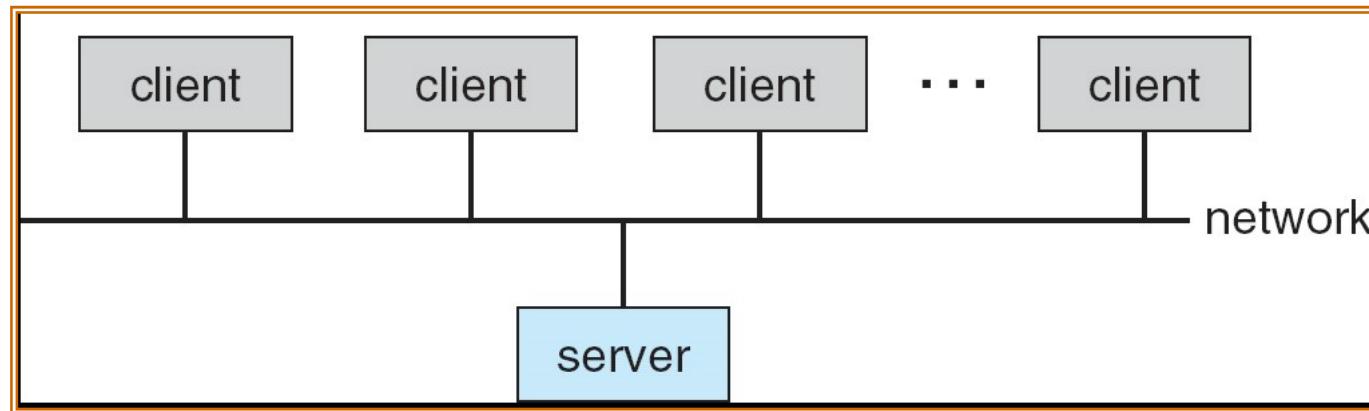
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights

# Computing Environments

- Traditional computer
  - Office environment
    - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
    - Now portals allowing networked and remote systems access to same resources
  - Home networks
    - Used to be single system, then modems
    - Now firewalled, networked

# Computing Environments (Cont.)

- Client-Server Computing
  - Dumb terminals supplanted by smart PCs
  - Many systems are now **servers**, responding to requests generated by **clients**
    - ▶ **Compute-server** provides an interface to client to request services (i.e. database)
    - ▶ **File-server** provides interface for clients to store and retrieve files



# Peer-to-Peer Computing

- A model of distributed system
- P2P **does not distinguish clients and servers**
  - Instead **all nodes are considered peers**
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via *discovery protocol*
  - Examples include *Napster* and *Gnutella*

# Web-Based Computing

- Web has become ubiquitous
- PCs are most prevalent devices
- More devices are networked to allow web access
- New category of devices to manage web traffic among similar servers: **load balancers**

# Module1\_Operating-System Structures

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- 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

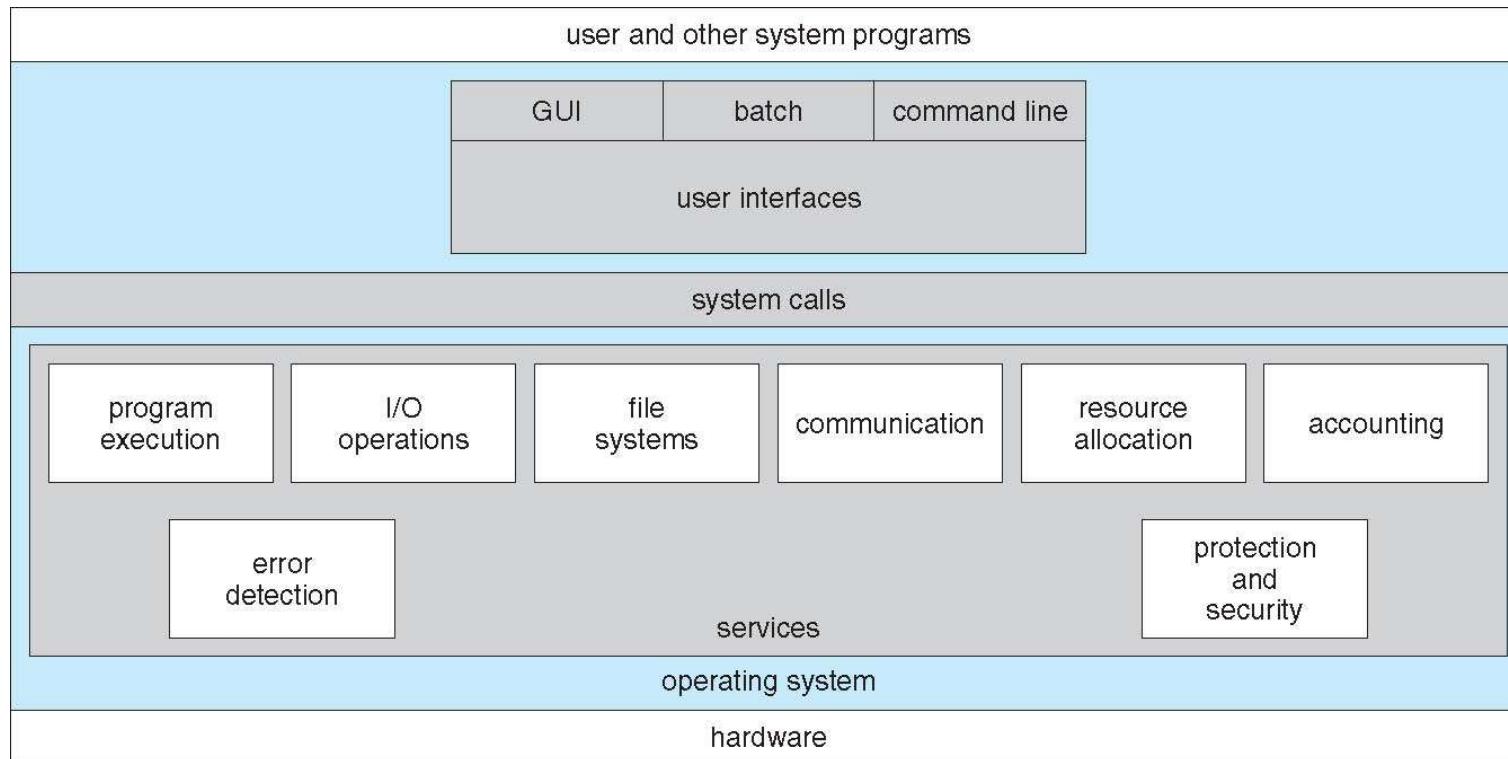
# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (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 to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

# Operating System Services (Cont.)

- Another set of **OS functions** exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs are 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

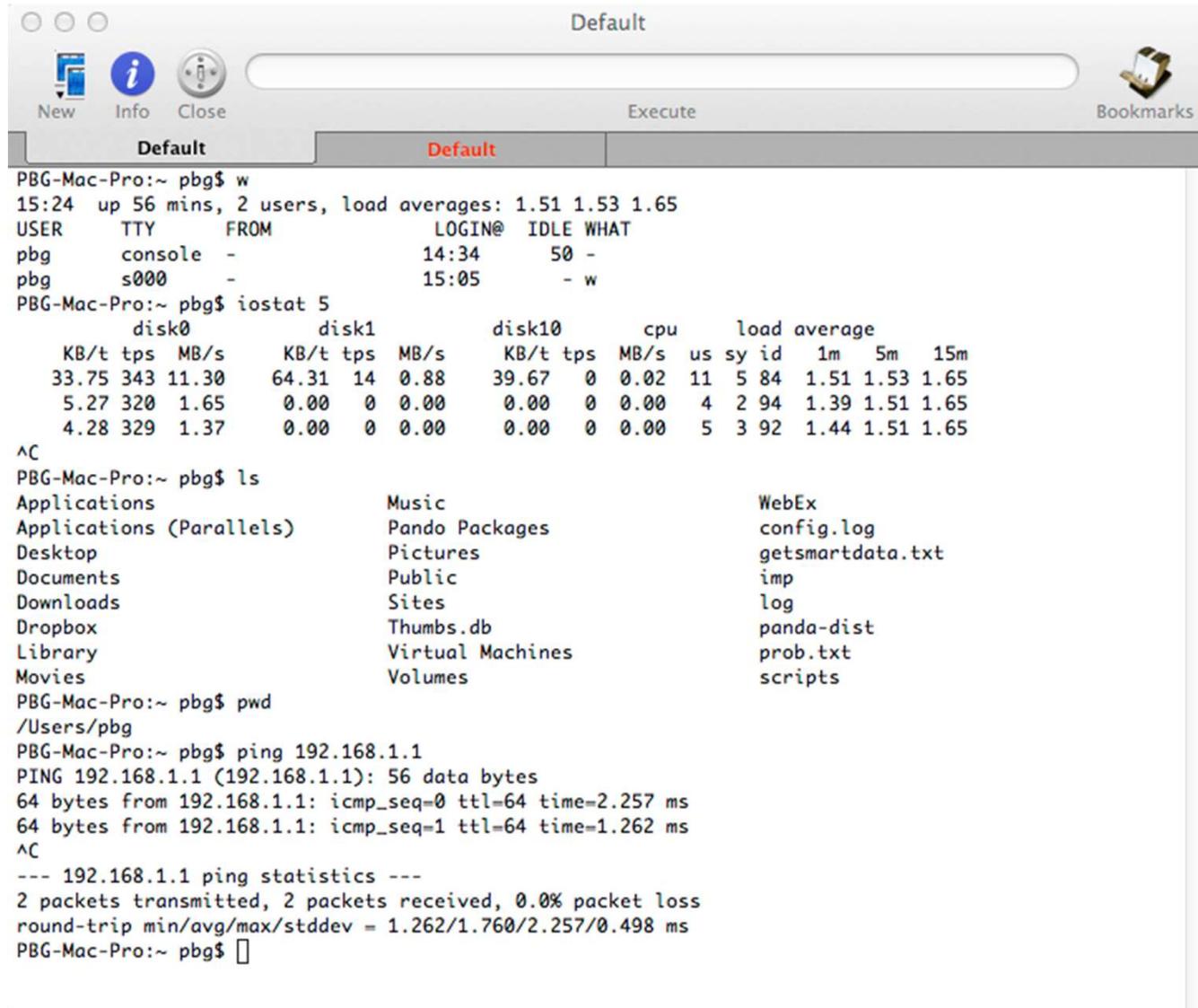


# User Operating System Interface - CLI

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

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

# Bourne Shell Command Interpreter



The screenshot shows a Mac OS X terminal window titled "Default". The window has standard OS X window controls (minimize, maximize, close) at the top left, and a toolbar with "New", "Info", "Close", "Execute", and "Bookmarks" buttons at the top right. The main pane displays a command-line session:

```
PBG-Mac-Pro:~ pbg$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER      TTY      FROM           LOGIN@ IDLE WHAT
pbg      console -          14:34      50 -
pbg      s000   -          15:05      - w

PBG-Mac-Pro:~ pbg$ iostat 5
              disk0      disk1      disk10     cpu    load average
              KB/t tps MB/s    KB/t tps MB/s    KB/t tps MB/s us sy id  1m   5m   15m
 33.75 343 11.30   64.31 14  0.88   39.67 0  0.02 11  5 84  1.51 1.53 1.65
  5.27 320 1.65   0.00 0  0.00   0.00 0  0.00  4  2 94  1.39 1.51 1.65
  4.28 329 1.37   0.00 0  0.00   0.00 0  0.00  5  3 92  1.44 1.51 1.65

^C
PBG-Mac-Pro:~ pbg$ ls
Applications           Music           WebEx
Applications (Parallels) Packages       config.log
Desktop                Pictures        getsmartdata.txt
Documents              Public          imp
Downloads              Sites           log
Dropbox                Thumbs.db      panda-dist
Library                Virtual Machines prob.txt
Movies                 Volumes         scripts

PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg

PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```

# 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**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

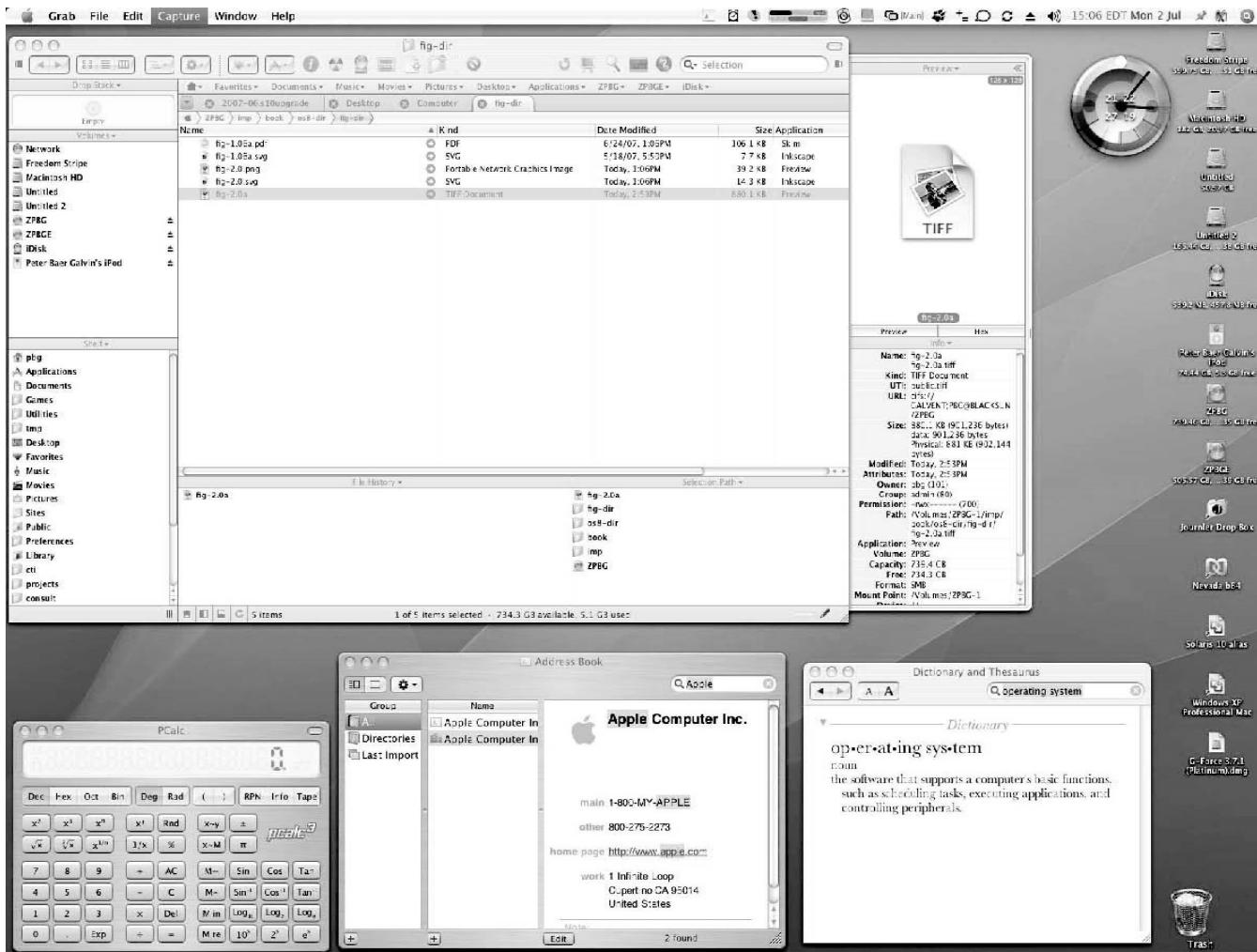
# Touchscreen Interfaces

## Touchscreen devices require new interfaces

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



# The Mac OS X GUI

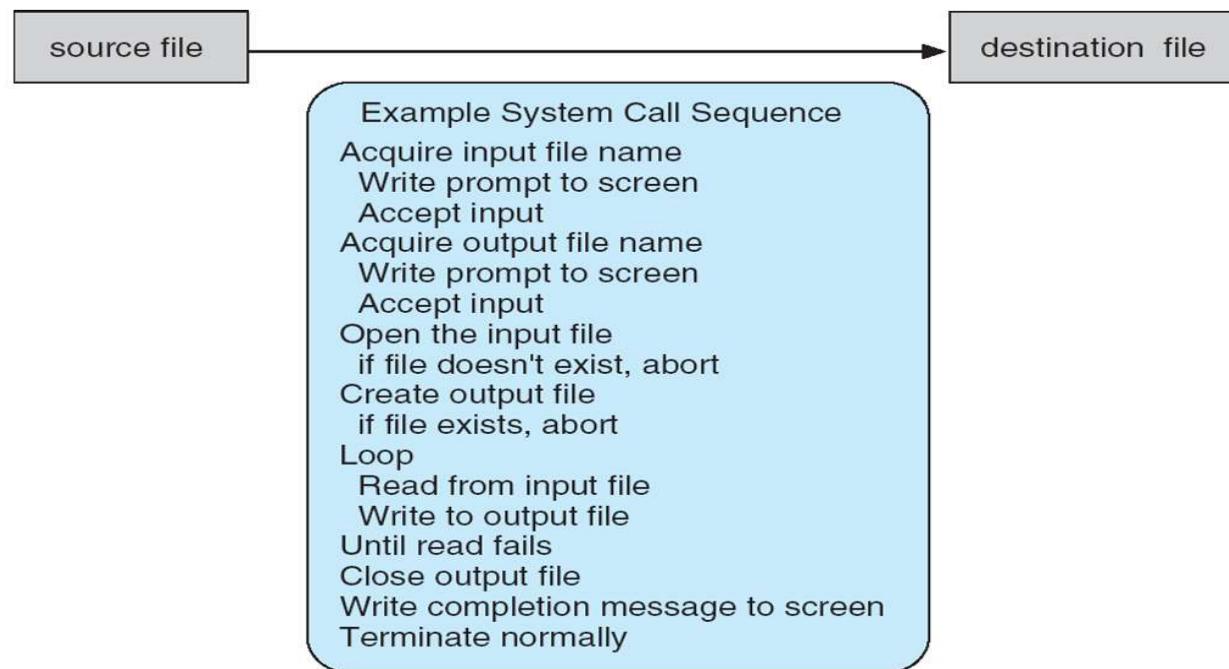


# System Calls

- 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 file to another file



# 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:

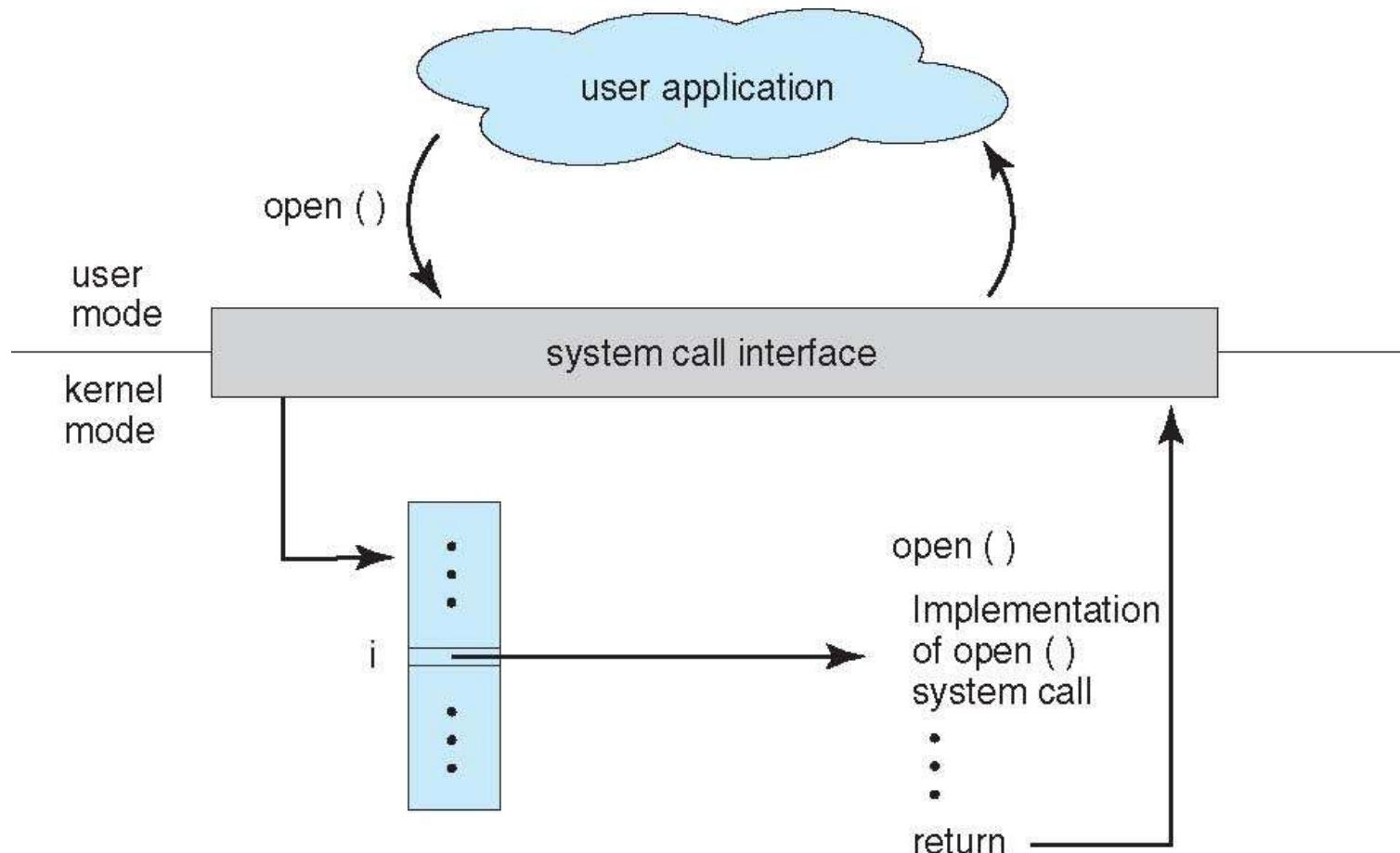
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

# System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface **invokes the intended system call in OS kernel** and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Needs to obey API and understand what OS will do as a result of call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

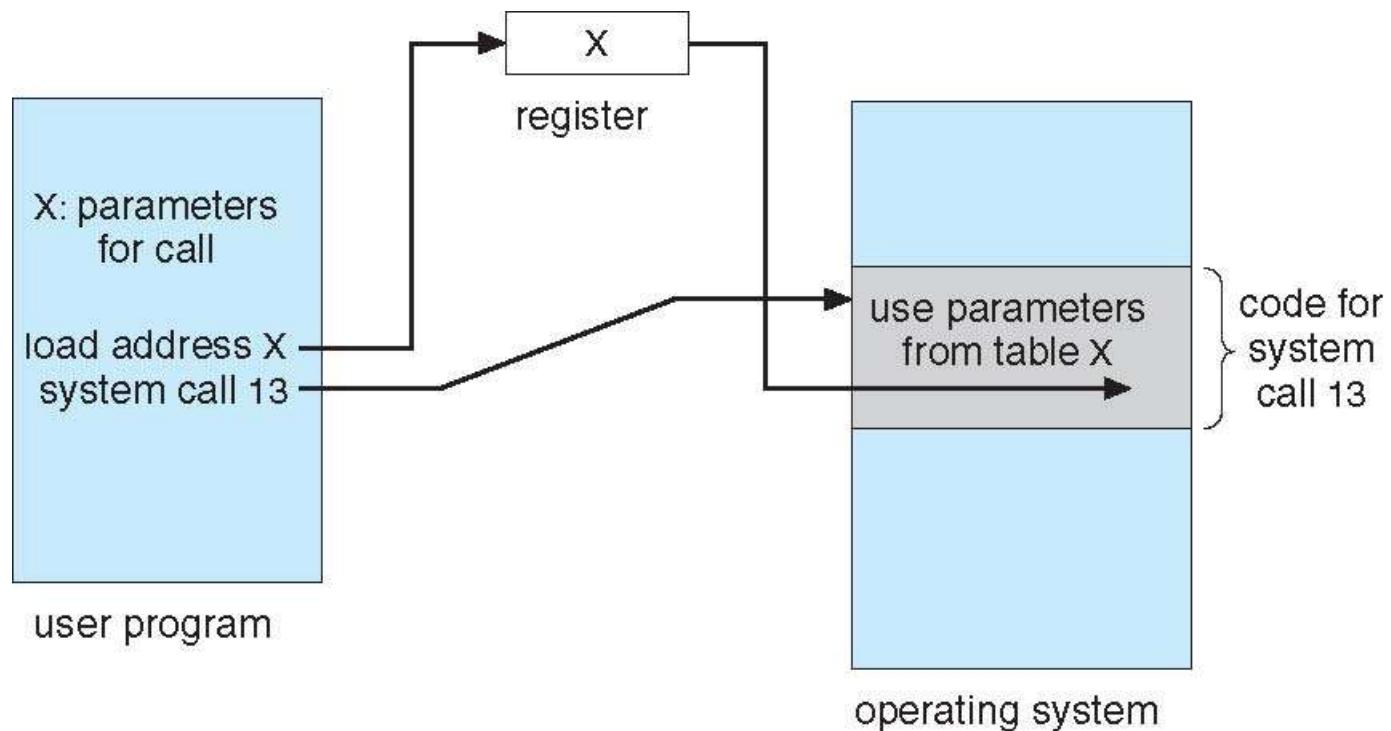
# API – System Call – OS Relationship



# System Call Parameter Passing

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

# Parameter Passing via Table



# Examples of Windows and Unix System Calls

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

# Types of System Calls

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

# Types of System Calls

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

# Types of System Calls (Cont.)

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

# Types of System Calls (Cont.)

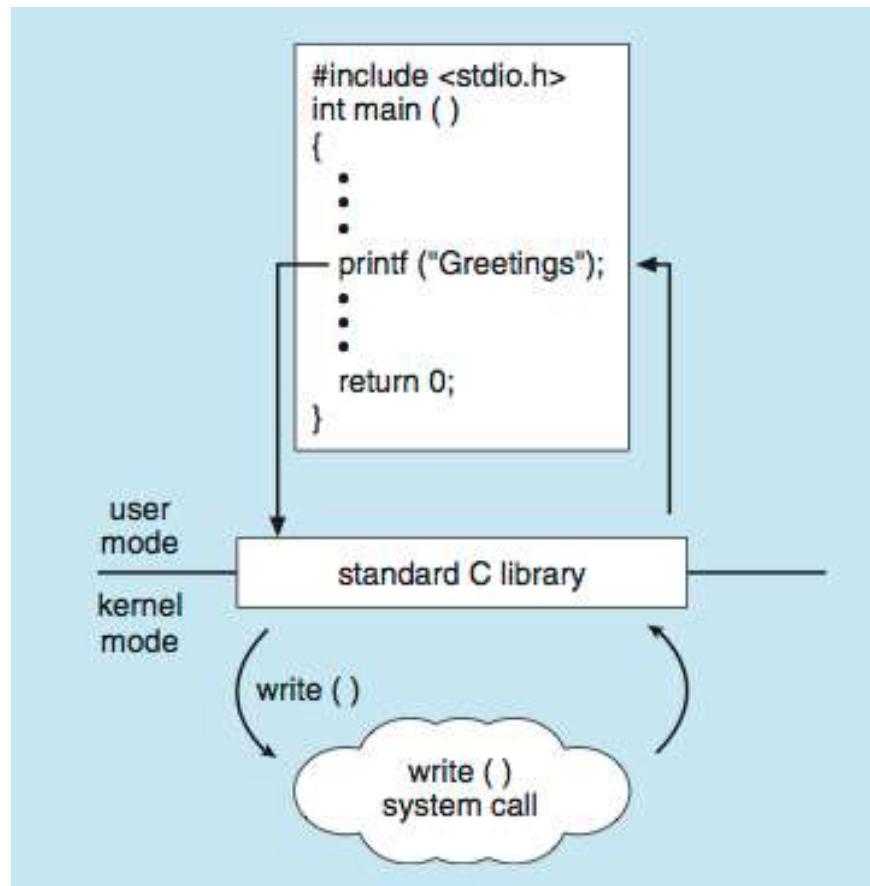
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

# Examples of Windows and Unix System Calls

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

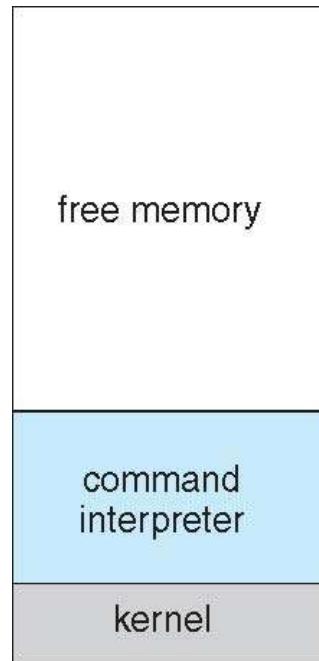
# Standard C Library Example

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



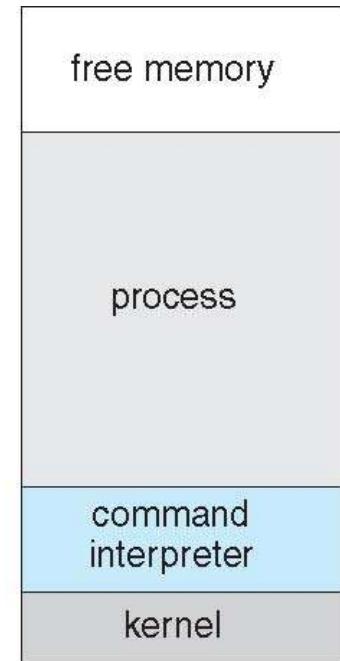
# Example: MS-DOS

- 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

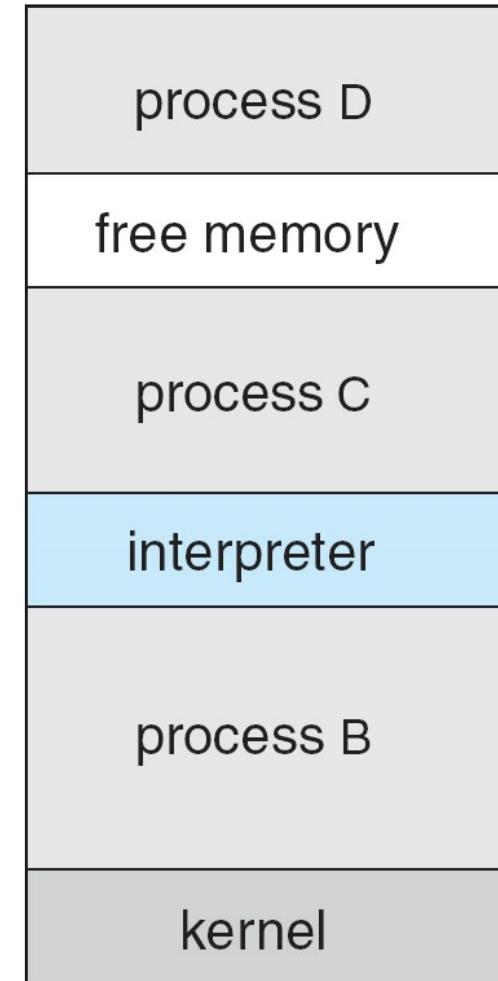


(b)

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
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
- Most users' view of the operating system is **defined by system programs**, not the actual system calls

# System Programs

- Provide a **convenient environment for program development and execution**
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **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
  - Typically, these programs format and print the output to the terminal or other output devices

# 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**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

# System Programs (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services, subsystems, daemons**
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered as part of OS
  - Launched by command line, mouse click, finger poke

# Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- 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

## 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)
- Specifying and designing an OS is a highly creative task of **software engineering**

# Implementation

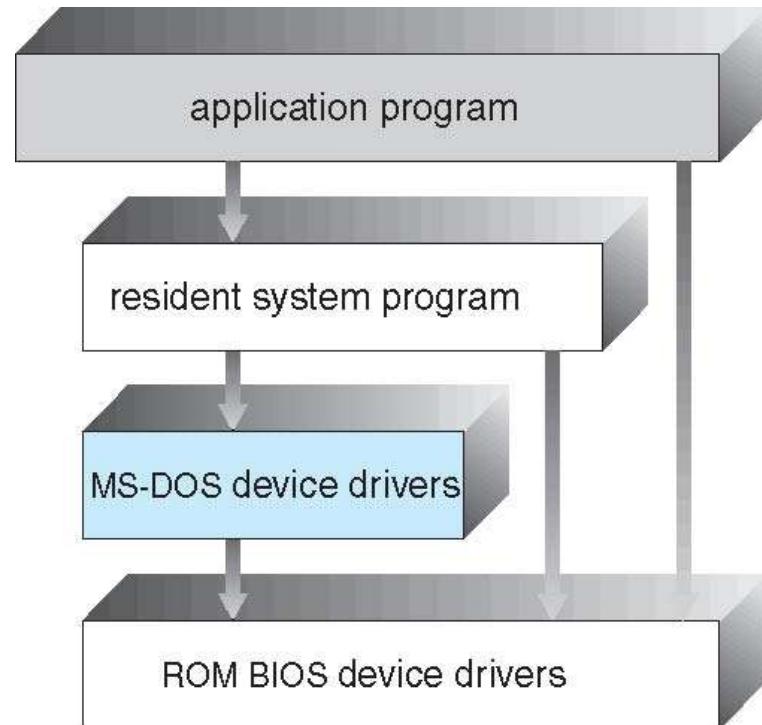
- Much variation
  - Early OSes are written in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
-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

# Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure
  - Simple structure – MS-DOS
  - Complex - UNIX
  - Layered – an abstraction
  - Microkernel -Mach

# Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



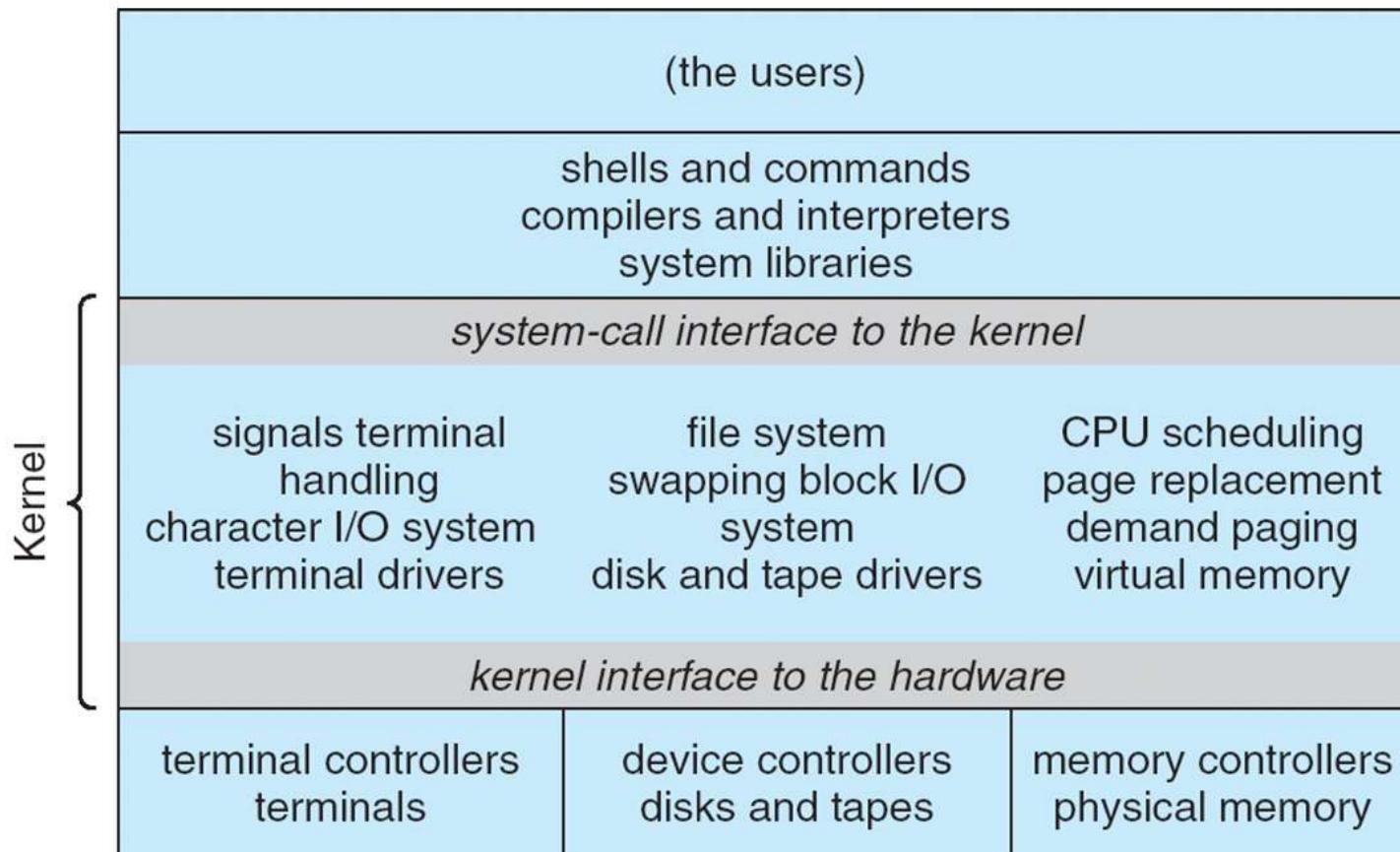
# Non Simple Structure -- UNIX

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- **Systems programs**
- **The kernel**
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

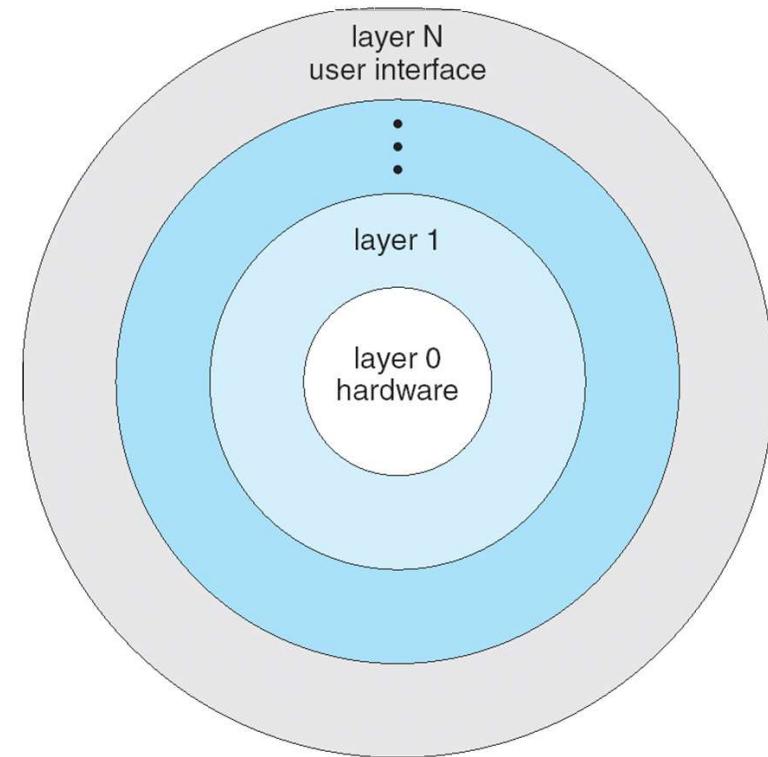
# Traditional UNIX System Structure

Beyond simple but not fully layered



# Layered Approach

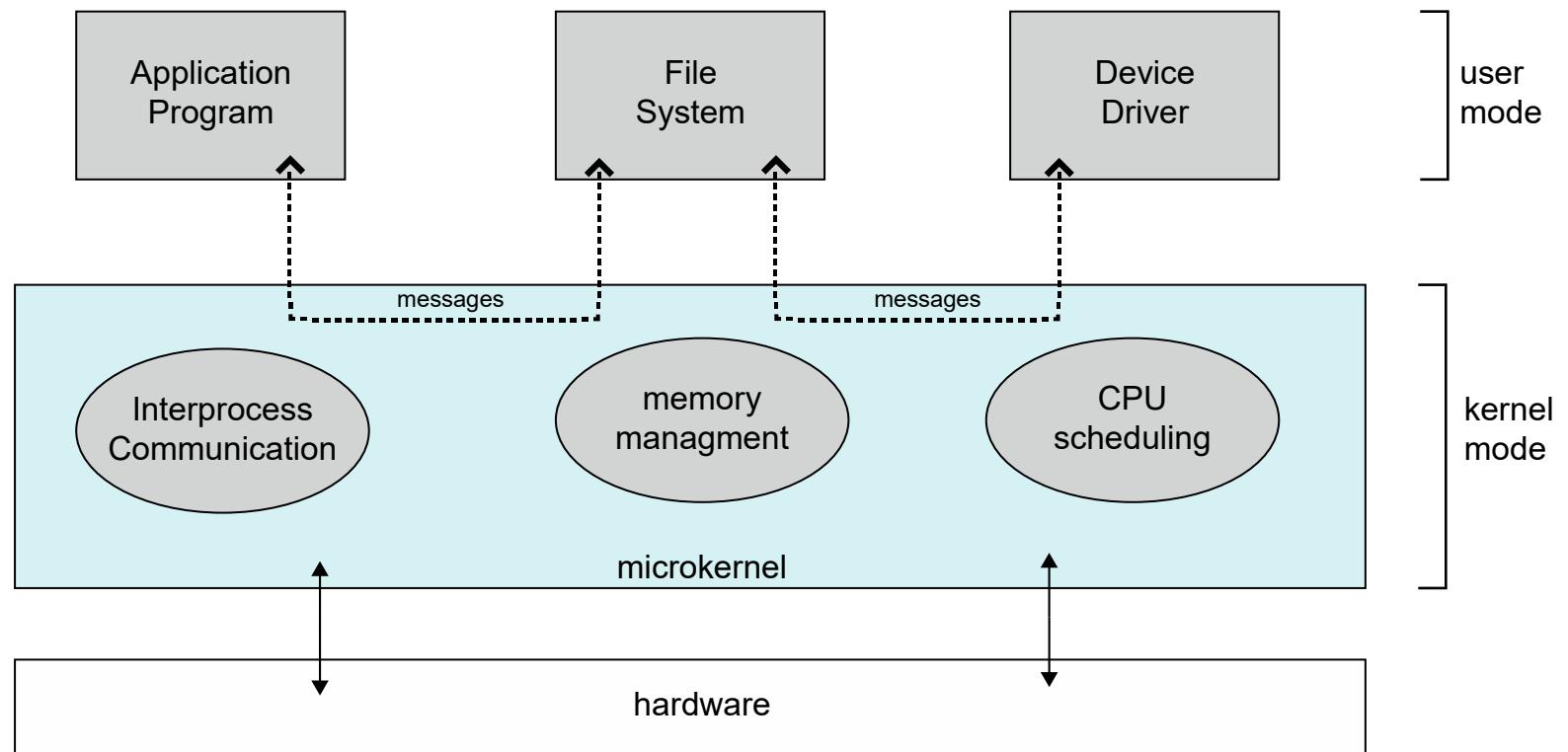
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



# Microkernel System Structure

- Moves as much from the kernel space into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

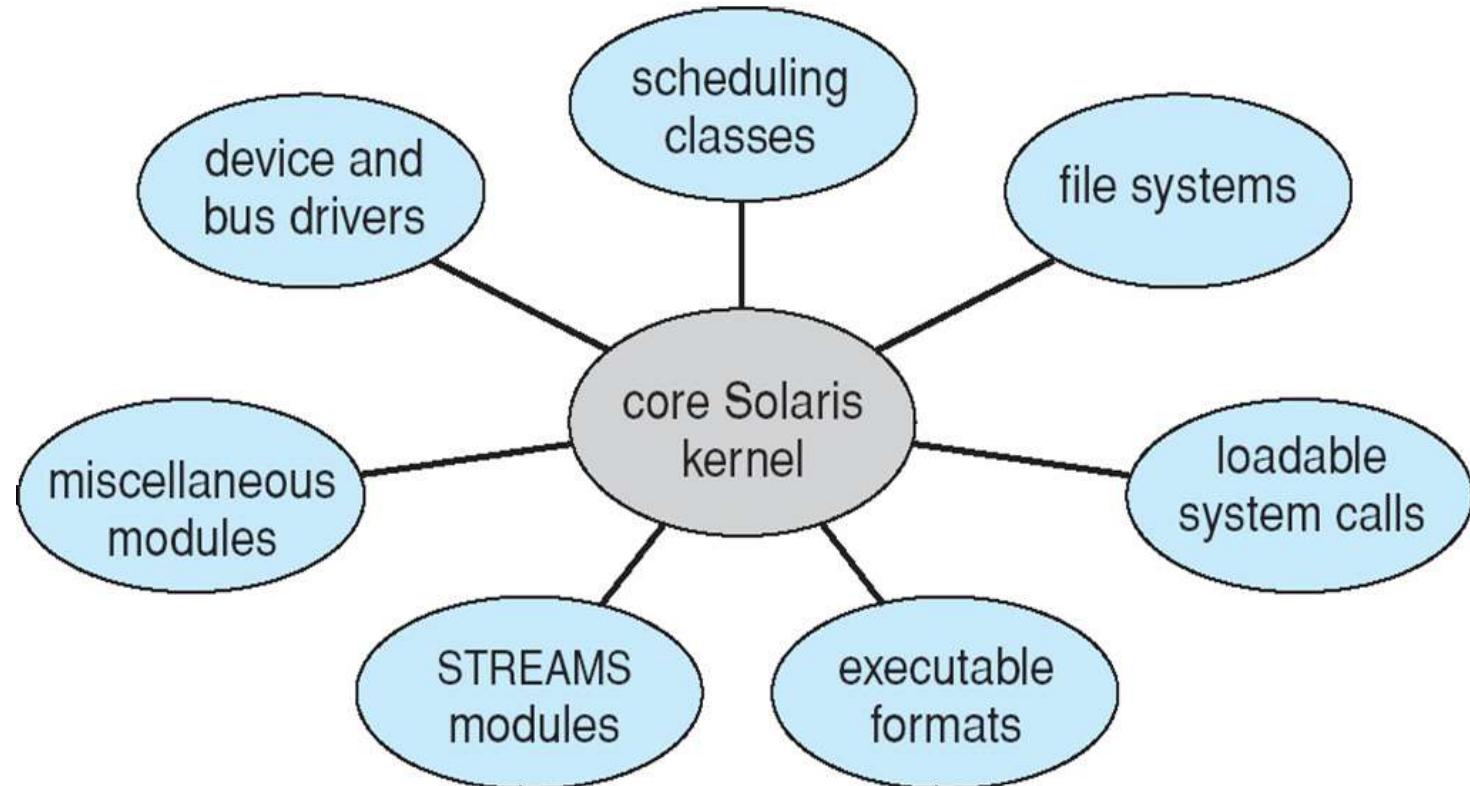
# Microkernel System Structure



# Modules

- Many modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each module is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, etc

# Solaris Modular Approach



# Hybrid Systems

- Most modern operating systems are actually not one pure models
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem *personalities*
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

# Mac OS X Structure

graphical user interface

Aqua

application environments and services

Java

Cocoa

Quicktime

BSD

kernel environment

Mach

BSD

I/O kit

kernel extensions

# iOS

- Apple mobile OS for *iPhone, iPad*
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel

Cocoa Touch

Media Services

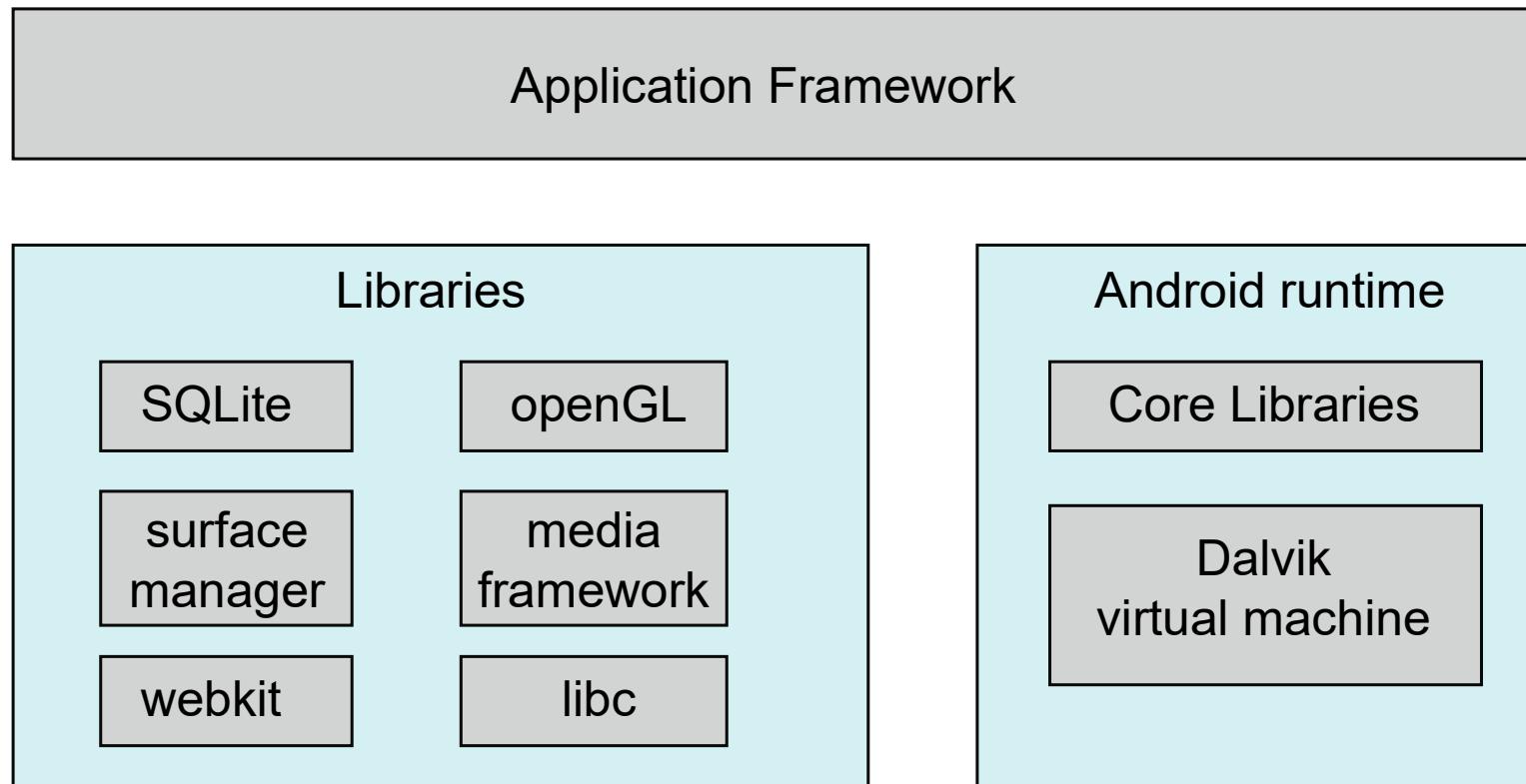
Core Services

Core OS

# Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable that runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

# Android Architecture



# Module2\_Processes

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

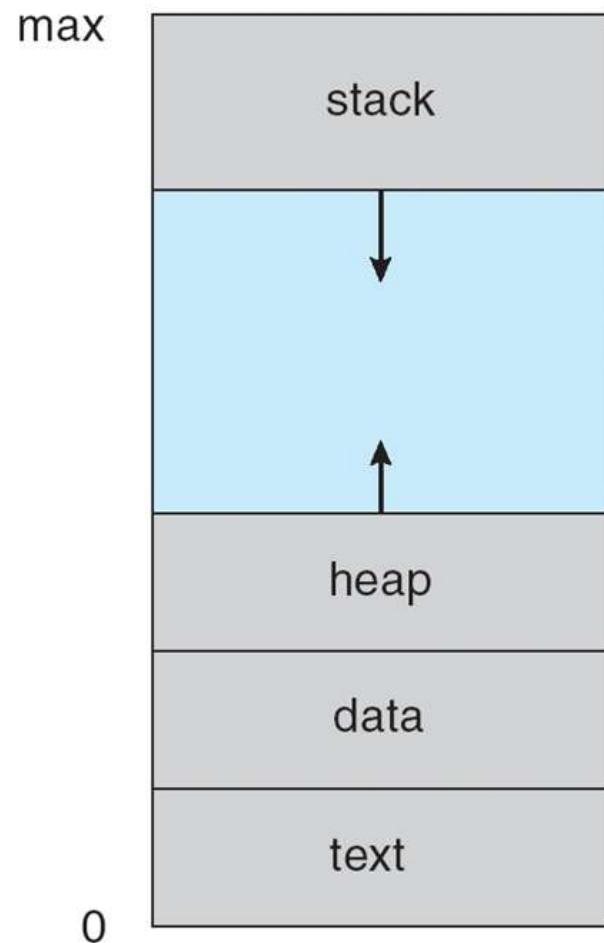
# Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section/code 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

# Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file is loaded into memory
- Execution of program is started via GUI mouse clicks, command line entry of its name, etc..
- One program can be several processes
  - Consider multiple users executing the same program

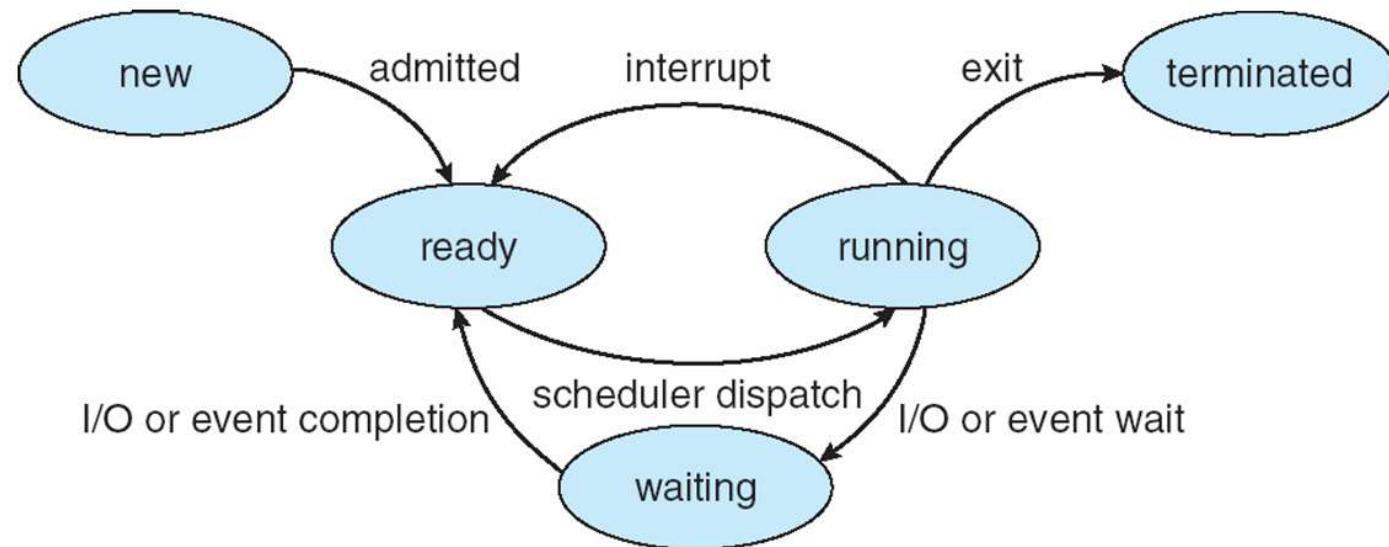
# Process in Memory



# 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

# Process States - Diagram



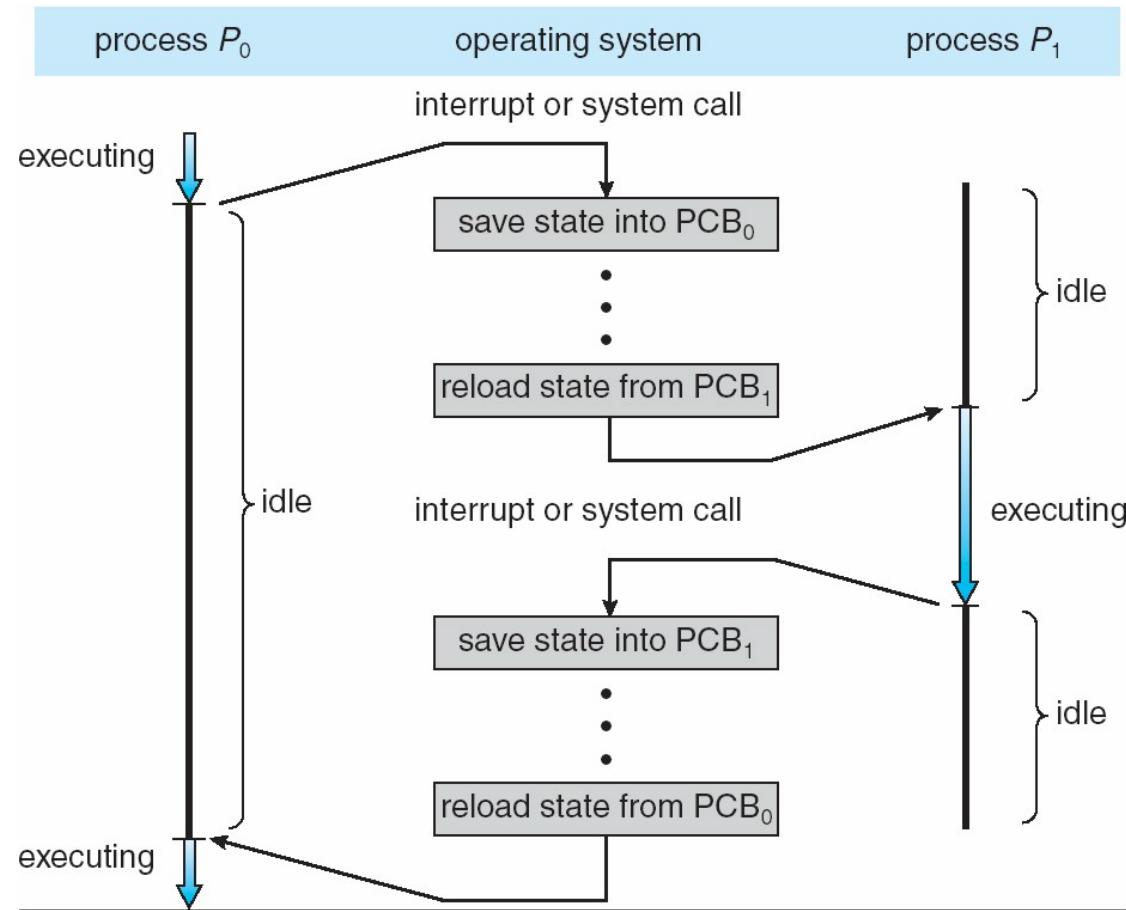
State Transition Diagram of a Process

# Process Control Block (PCB)

- Information associated with each process (also called **task control block**) is stored in PCB
- Process state – running, waiting, etc
  - Process number – ID of process
  - Program counter – location of next instruction to 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



# CPU Switch From Process to Process



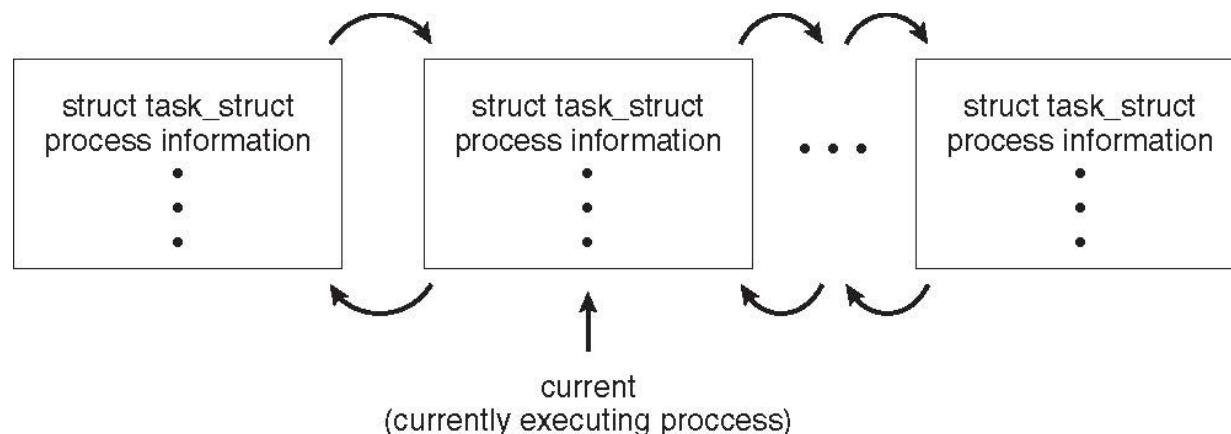
# Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
  - Must then have storage for thread details, multiple program counters in PCB

# Process Representation in Linux

Represented by the C structure `task_struct`

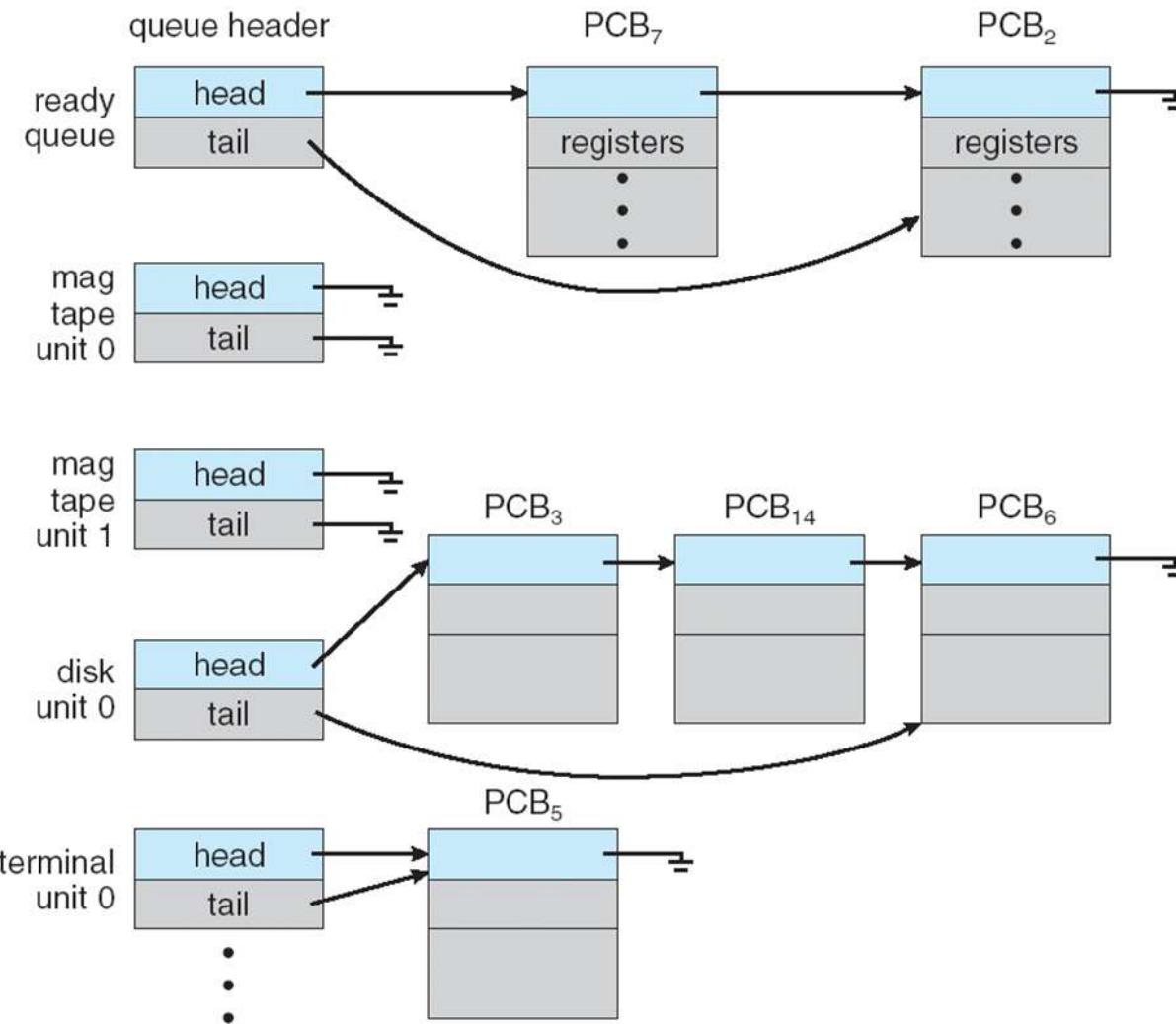
```
pid_t pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```



# Process Scheduling

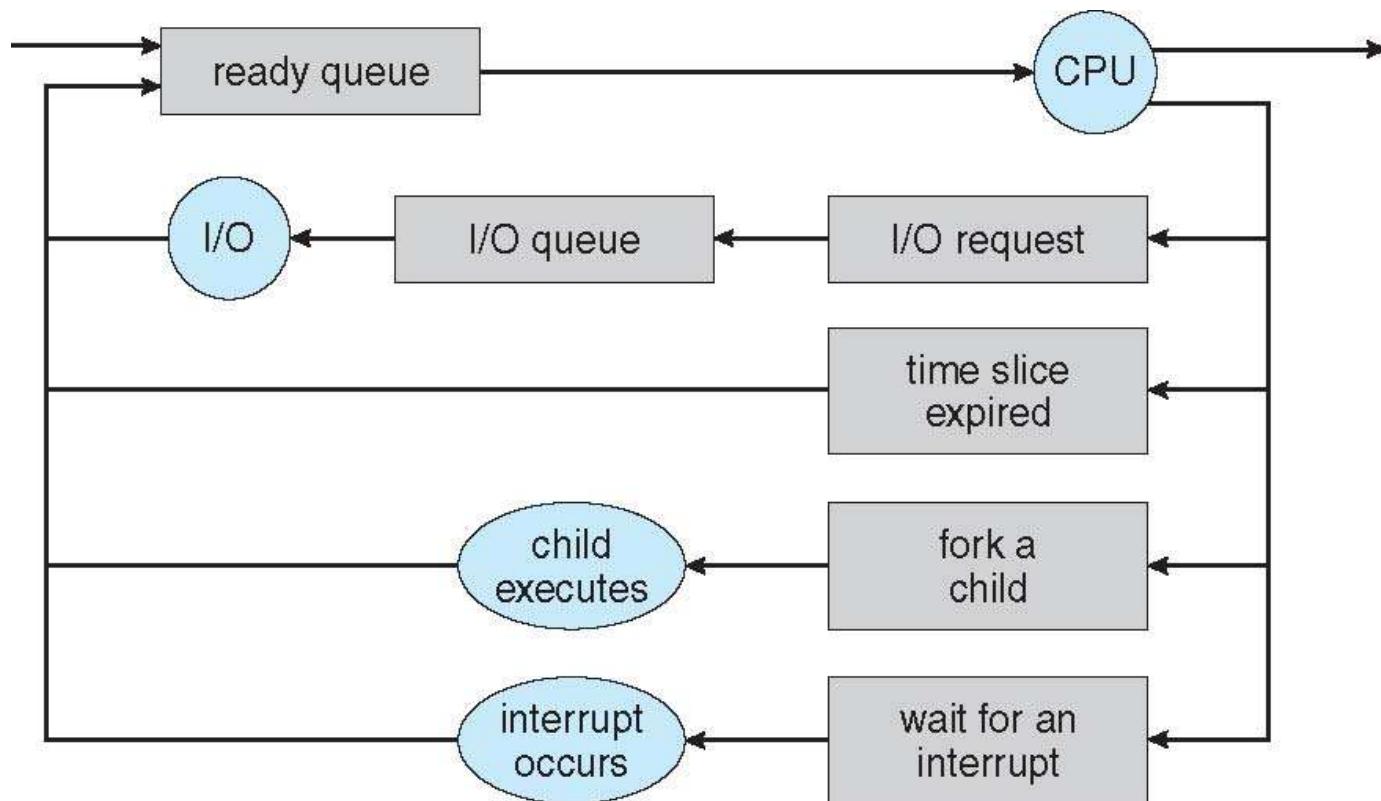
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing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

# Ready Queue And Various I/O Device Queues



# Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows

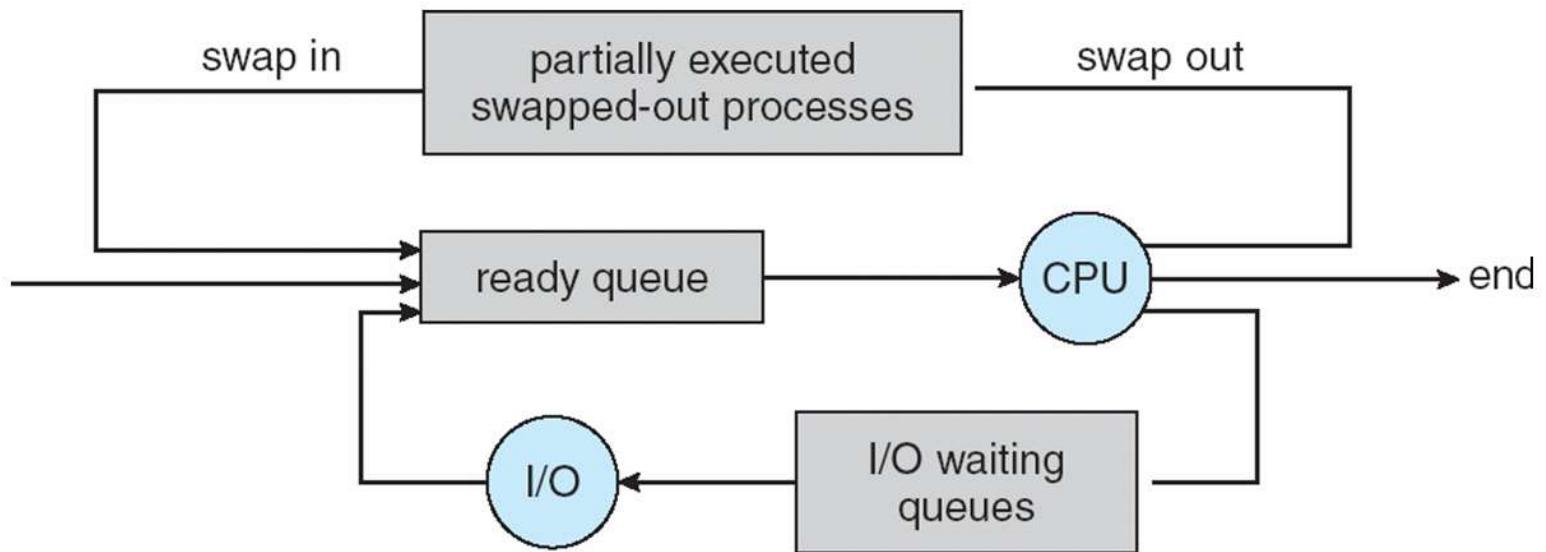


# Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)  $\Rightarrow$  (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 exist
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts exist
- Long-term scheduler strives for good ***process mix***

# 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
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

# 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 is represented in the PCB**
- Context-switch time is an overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

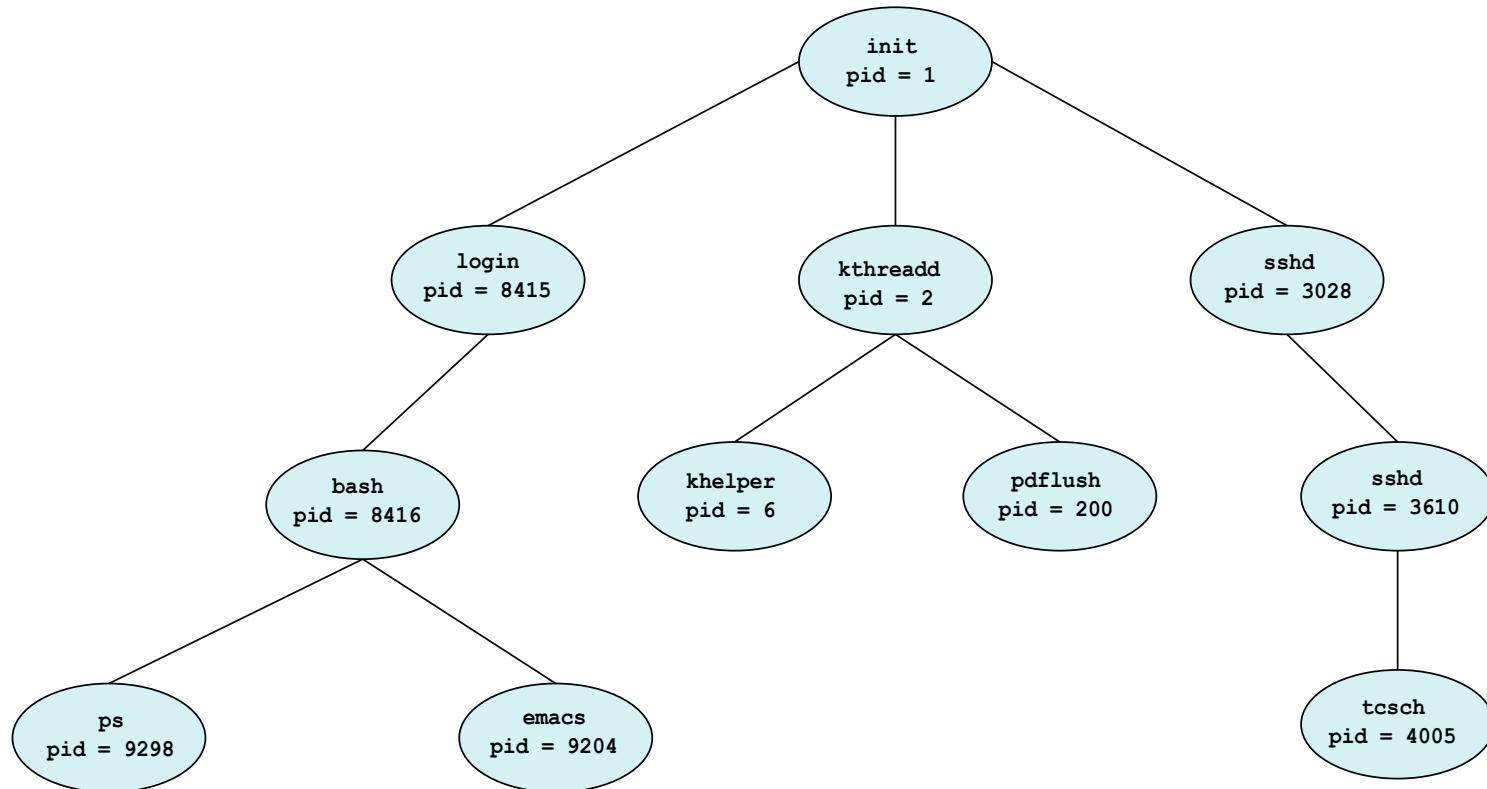
# Operations on Processes

- System must provide mechanisms for:
  - process creation
  - process termination

# Process Creation

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

# A Tree of Processes in Linux

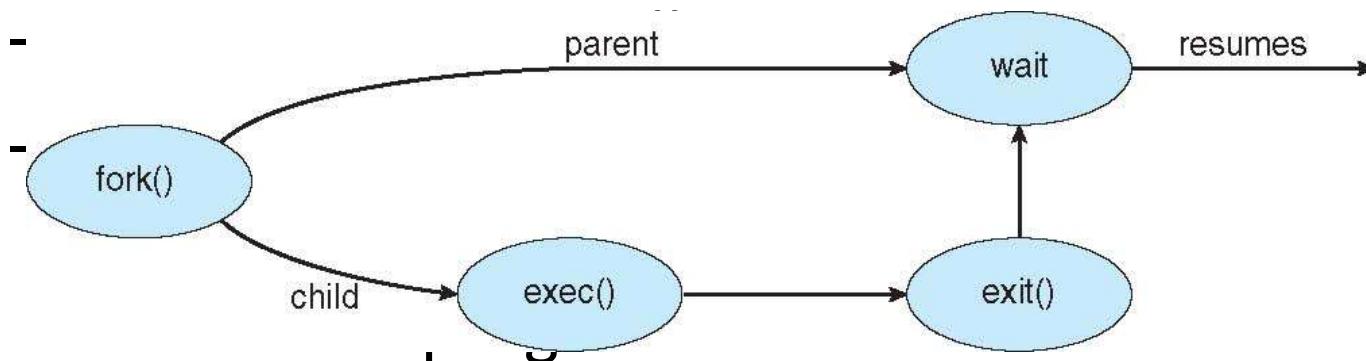


\$ps -aux

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
runner	1	0.0	0.0	1385548	13068	?	S	lsl 19:47	0:00	/inject/init
runner	12	0.0	0.0	20192	3848	pts/0	Ss	19:58	0:00	bash --norc
runner	18	0.0	0.0	36092	3384	pts/0	R+	19:58	0:00	ps -aux

# Process Creation (Cont.)

- Address space
  - Child is a duplicate of parent
  - Child has a program loaded into it (using exec system calls)
- UNIX examples
  -



# C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

# Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process's resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

# Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call . The call returns status information and the pid of the terminated process  
`pid = wait(&status);`
- If parent is not waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait` , process is an **orphan**

# Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



```

// fork() system call examples

//Example .....  

#include<stdio.h>  

#include<unistd.h>  

int main(){  

    fork();  

    printf("Hello\n");  

    return 0;  

}  

//Result : Hello will be output two times ( by parent process and child process)  

/*Note: fork() system call creates a child process. Child process is a copy of the parent process by  

default. Child process goes live after the successful fork() system call. Hence all statements after a fork  

system call will be executed twice i.e by the child and parent processes.  

*/  

//Example .....  

#include<stdio.h>  

#include<unistd.h>  

int main(){  

    fork(); //Will be executed by parent process leading to child 1 creation  

    fork(); // Will be executed by parent process leading to child 2 creation and also executed by  

           //child 1 leading to child11 process  

    printf("Hello\n");  

    return 0;  

}  

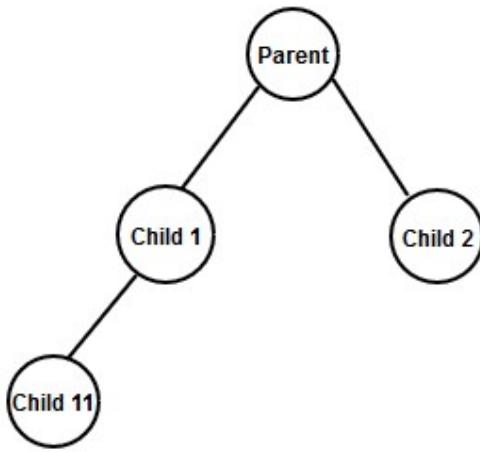
/* Result : Hello will be output four times ( by parent, child1 (created out of first fork), child2 (created  

out of second fork) and child11 (created out of second fork that is executed by child1). Plz refer the  

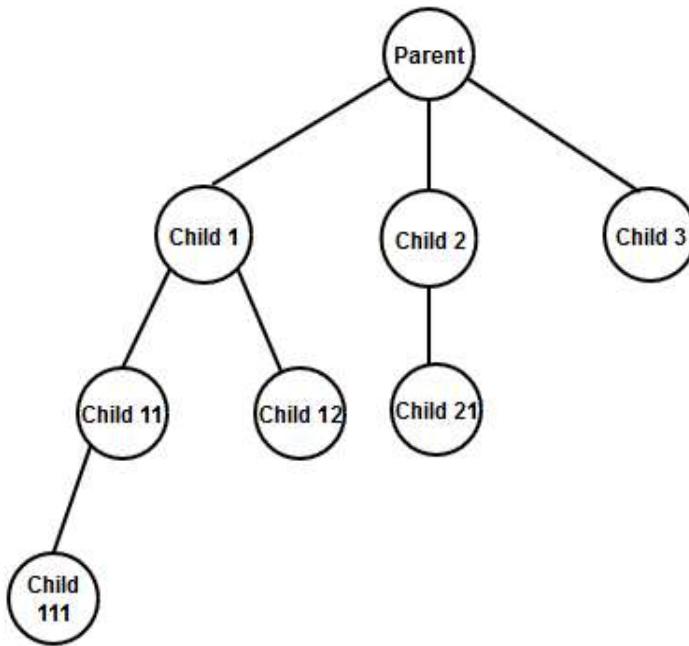
diagram below.  

*/

```



```
//Example .....  
#include<stdio.h>  
#include<unistd.h>  
int main(){  
    fork();  
    fork();  
    fork();  
    printf("Hello\n");  
    return 0;  
  
}  
/* Result : Hello will be output eight times as there are eight processes running as given below in the  
diagram. In general if there are 'n' calls to fork then there will be  $2^n$  processes created and running.  
*/
```



//Example .....

```

#include<stdio.h>
#include<unistd.h>
int main(){
    printf("Hello 1\n"); //Will be executed by parent process only
    fork();
    printf("Hello 2\n"); //Will be executed by parent and child processes
    return 0;
}
  
```

//Example .....

//General flow in assigning task to child process

```

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(){
    pid_t pid;
  
```

```

// Statements here will be executed by parent only .....
pid = fork(); //Creates a child process

if ( pid < 0 ) {

// This block is to check if fork has failed (unsuccessful) if so we return or exit immediately

perror("fork"); //Prints the error message behind the failure of fork

exit( -1);

}

if ( pid == 0 ) {

/* These statements will be executed by child. Because when child executes this "if" condition it
becomes true since fork returns 0 to the child process

*/
}

else {

/* These statements will be executed by parent. Because when parent executes this "if"
condition it becomes false since fork returns a non-zero positive value (which is the ID of the child
process) upon successful child process creation

*/
}

// Statements here will be executed by parent and child .....

return 0;
}

//Example .....

```

```

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main(){

pid_t pid;

pid = fork();

if ( pid < 0 ) {

perror("fork"); //Prints the error message behind the failure of fork

exit( -1);
}

```

```

    }

    if ( pid == 0 ) {
        printf("Child Process is executing\n");
    }
    else {
        printf("Parent Process is executing\n");
    }

    return 0;
}

//Example .....
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {

    pid_t pid;
    pid =fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0) {
        execvp("/bin/ls","ls",NULL); // Child process is executing ls command
    }
    else {
        wait (NULL); //Parent waits for child to complete its task
        printf("Child Completed");
    }

    return 0;
}

```

```
//Example .....  
  
#include <sys/types.h>  
#include <stdio.h>  
#include <unistd.h>  
  
int main()  
{  
    pid_t pid;  
  
    printf("Before fork system call");  
  
    fflush(stdout); //flushes out the contents of output buffer to standard output device  
  
    pid = fork();  
  
    if( pid < 0 ) {  
        fprintf(stderr, "Fork Failed");  
  
        return 1;  
    }  
  
    printf("Hello ...\\n");  
  
    return 0;  
}
```

```
//Example .....  
  
//Check the behaviour of the program as given firstly  
// Secondly by uncommenting fflush(stdout)  
  
#include <sys/types.h>  
#include <stdio.h>  
#include <unistd.h>  
  
int main()  
{
```

```

pid_t pid;

printf("Before fork system call");

//    fflush(stdout);

pid = fork();

if( pid < 0 ) {

    fprintf(stderr, "Fork Failed");

    return 1;

}

printf("Hello ...\\n");

return 0;

}

//Example ......

//Execution context switching between Parent and Child Processes

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int main()

{

    pid_t pid;

    printf("Parent Process is executing before fork()\\n");

    pid = fork();

    if (pid < 0) {

        fprintf(stderr, "Fork Failed");

        return 1;

    }

    else if (pid == 0) {

        printf("Child Process is executing..BEFORE SLEEP\\n");

        sleep(1);

        printf("Child Process is executing..AFTER SLEEP\\n\\n\\n");

    }

}

```

```

else {
    printf("Parent Process is executing\n");
}
return 0;
}

//Creating an Orphan Process

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;int status;
pid = fork();
if (pid < 0) {
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) {
    sleep(2);
    printf("\nChild Process is executing\n");
}
else
{
    return 0;
    printf("\nParent Process is executing\n");
}
return 0;
}

```

# Module3\_CPU\_Scheduling

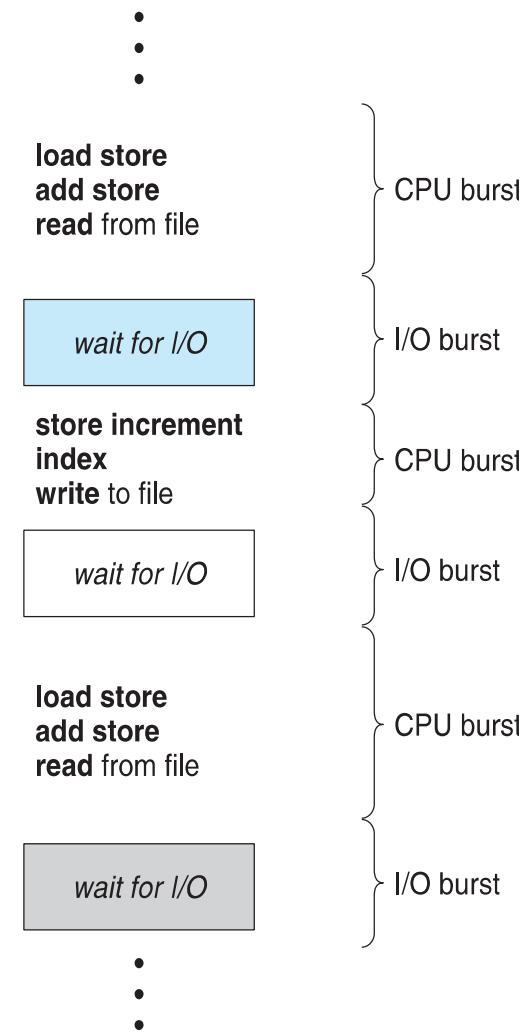
Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Objectives

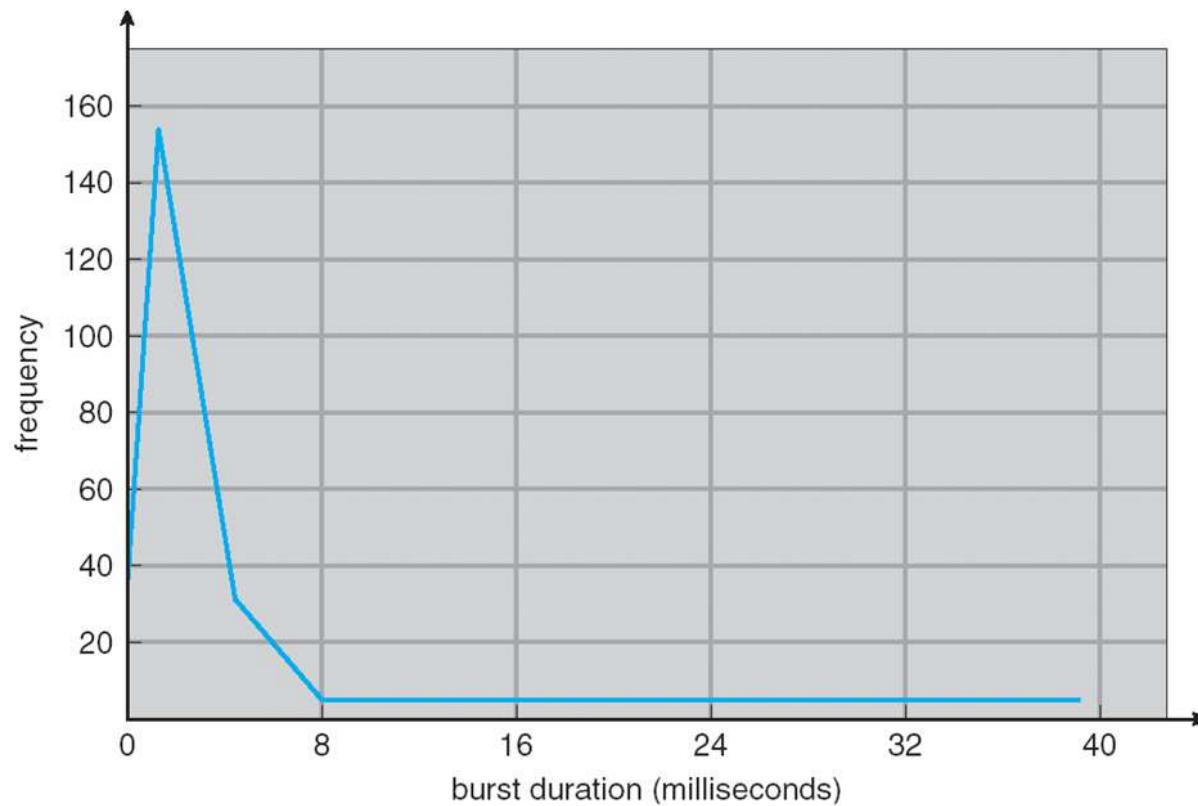
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

# Basic Concepts

- Maximum CPU utilization is 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
- **CPU Burst Time:** The amount of time a process executes before it goes to wait state

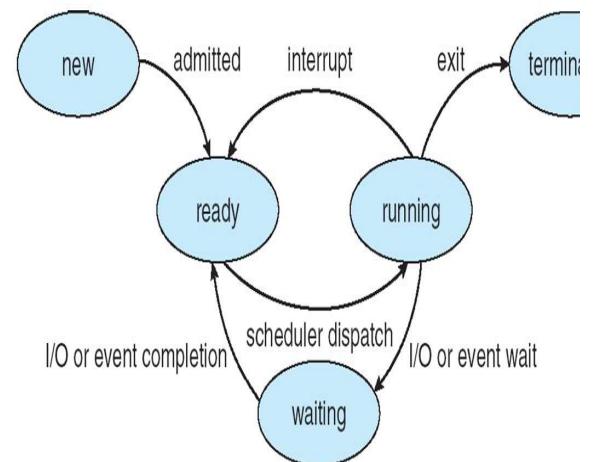


# Histogram of CPU-burst Times



# CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



# 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 taken by 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
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time taken from when a request was submitted until the first response is produced, not output (for time-sharing environment)

# Scheduling Algorithm Optimization Criteria

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

uptime -> Linux command  
system calls  
time  
`sched_getscheduler()`  
`sched_setscheduler()`

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

Example 1: Consider the following processes with their burst time. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

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

### Example 1: Continued

The Gantt Chart for the schedule is:



Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	24	24	24	0
P2	3	27	27	24
P3	3	30	30	27

- Waiting time for P<sub>1</sub> = 0; P<sub>2</sub> = 24; P<sub>3</sub> = 27
- Average waiting time:  $(0 + 24 + 27)/3 = 17$ 
  - Turnaround Time = Completion Time - Arrival Time
  - Waiting Time = Turn Around Time - Burst Time

# FCFS Scheduling (Cont.)

## Example 1: Continued

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

Process ID	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

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
- Performance of FCFS is dynamic
  - **Convoy effect** - short process behind long process
    - Consider one CPU-bound and many I/O-bound processes

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

- Example 2

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P <sub>1</sub>	0	7
P <sub>2</sub>	0	4
P <sub>3</sub>	0	1
P <sub>4</sub>	0	4

❑ Gantt chart

Schedule: P<sub>1</sub>

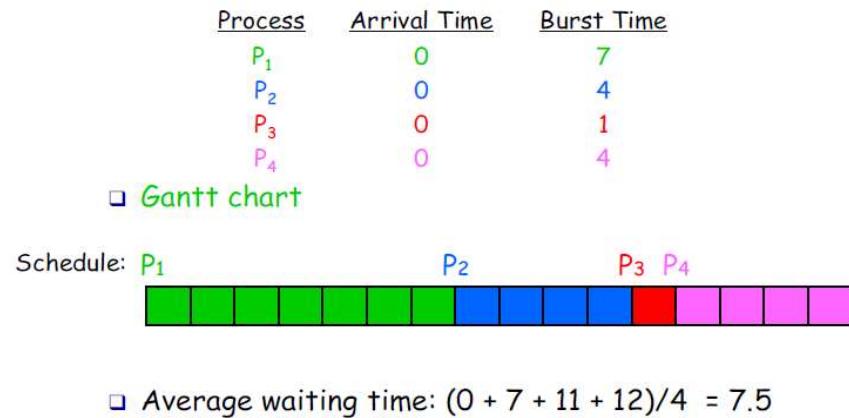


❑ Average waiting time:  $(0 + 7 + 11 + 12)/4 = 7.5$

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

### Example 2: Continued

The Gantt Chart for the schedule is:



Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	7	7	7	0
P2	4	11	11	7
P3	1	12	12	11
P4	4	16	16	12

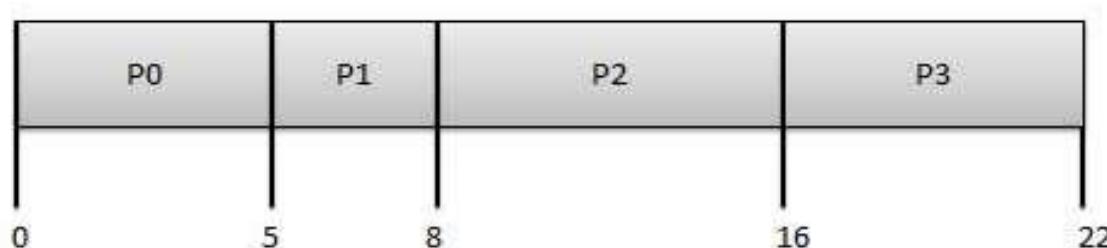
- Average waiting time:  $(0 + 7 + 11 + 12)/4 = 7.5$

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

Example 3: Consider the following processes with their **burst time and arrival time**. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	BurstTime	ArrivalTime
P0	5	0
P1	3	1
P2	8	2
P3	6	3

The **Gantt Chart** for the schedule is:



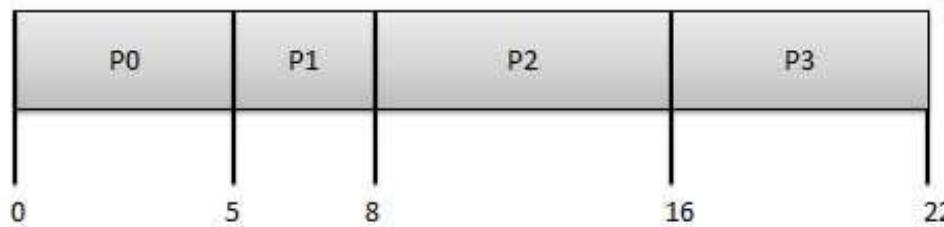
- Average Waiting Time:  $(0+4+6+13) / 4 = 5.75$

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

Example 3 (cont'd):

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P0	5	0	5	5	0
P1	3	1	8	7	4
P2	8	2	16	14	6
P3	6	3	22	19	13

The Gantt Chart for the schedule is:



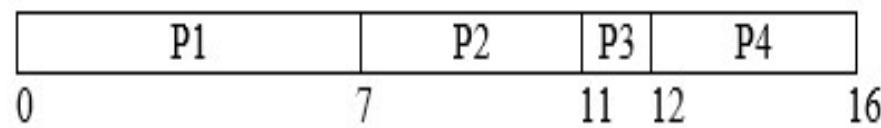
- Average Waiting Time:  $(0+4+6+13) / 4 = 5.75$

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

Example 4: Consider the following processes with their burst time and arrival time. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

The Gantt Chart for the schedule is:



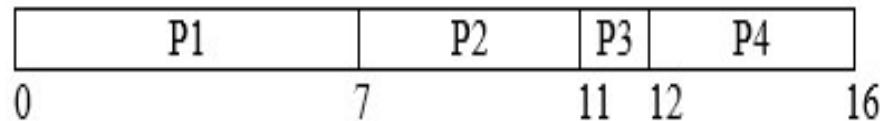
- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

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

Example 4: (continued)

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P1	7	0	7	7	0
P2	4	2	11	9	5
P3	1	4	12	8	7
P4	4	5	16	11	7

The Gantt Chart for the schedule is:



- Average waiting time =  $(0 + 5 + 7 + 7)/4 = 4.75$

# Shortest-Job-First (SJF) Scheduling

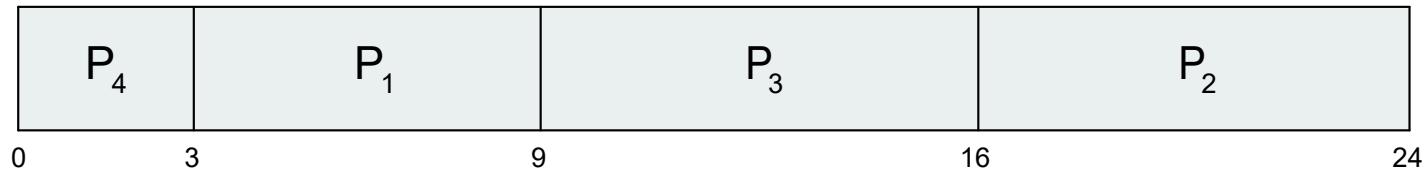
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

# SJF – Example 1

Example 1: Consider the following processes with their burst time. Use SJF to schedule the processes and compute the average waiting time.

Process ID	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



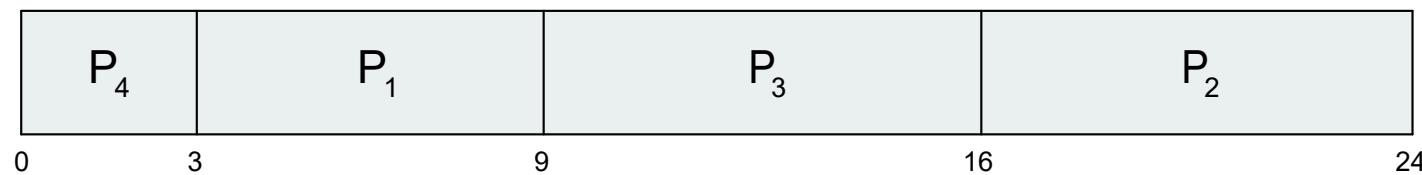
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# SJF – Example 1 (cont'd)

- Scheduling Table

Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	6	9	9	3
P2	8	24	24	16
P3	7	16	16	9
P4	3	3	3	0

- SJF scheduling chart



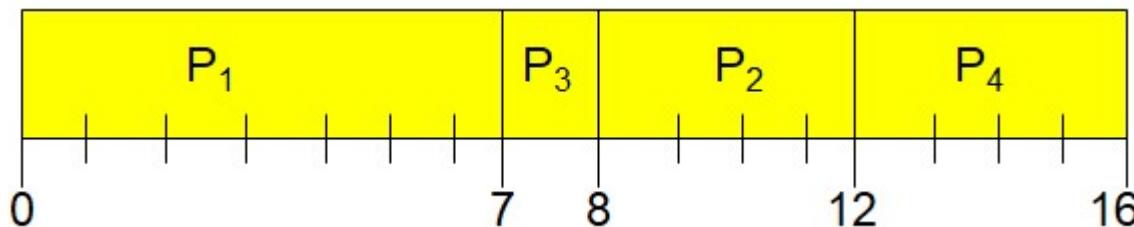
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

## SJF Scheduling - Example

Example 2: Consider the following processes with their burst time and arrival time. Use SJF to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

The Gantt Chart for the schedule is:



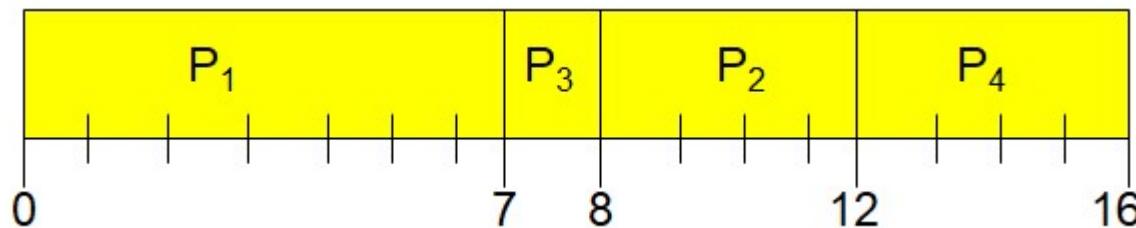
- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

## SJF Scheduling - Example

### Example 2: (continued)

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P1	7	0	7	7	0
P2	4	2	12	10	6
P3	1	4	8	4	3
P4	4	5	16	11	7

The Gantt Chart for the schedule is:

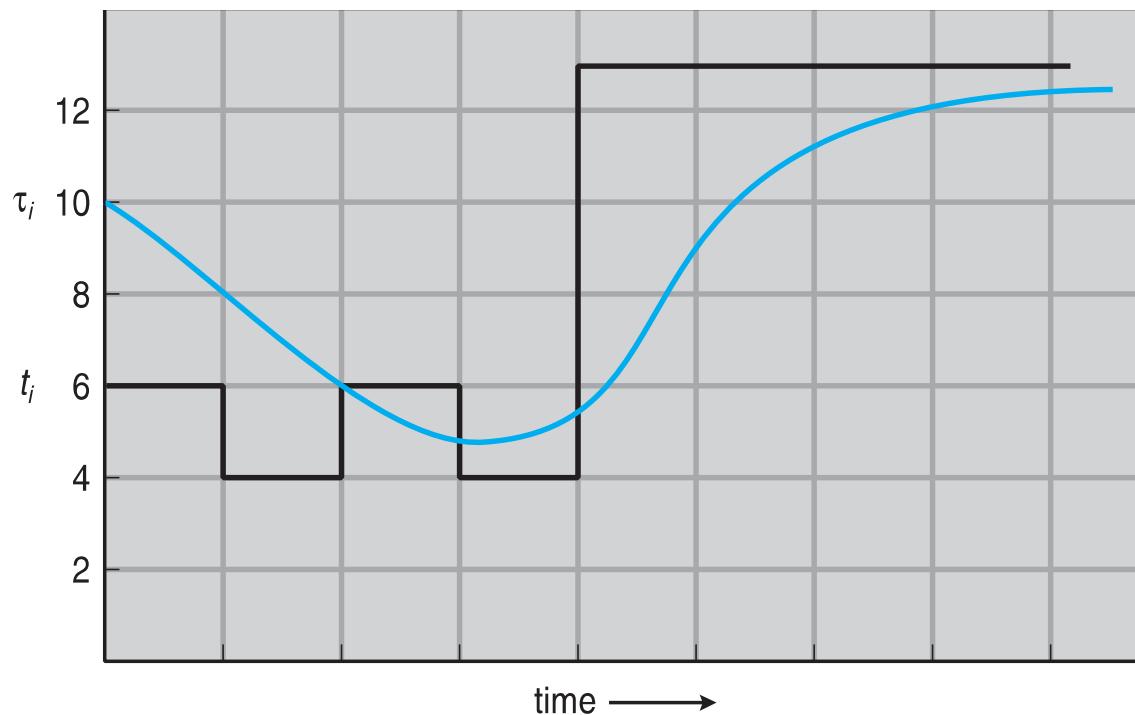


- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# 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
  1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define:  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
- Commonly,  $\alpha$  set to  $\frac{1}{2}$
- Preemptive version of SJF is called **Shortest-Remaining-Time-First (SRTF)**

## Prediction of the Length of the Next CPU Burst



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12

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

# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- $\alpha = 1$ 
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

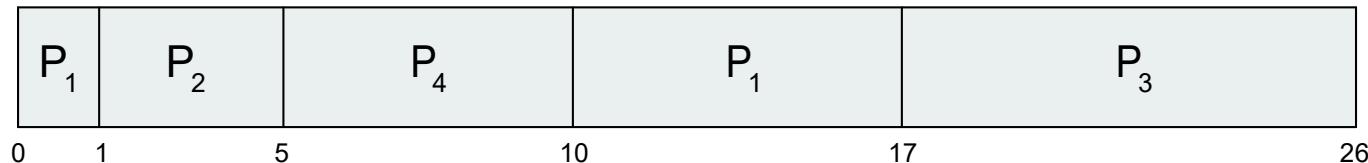
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **Shortest-Remaining-Time-First (SRTF)**
- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Completion Time</u>	<u>Wait Time</u>
$P_1$	0	8	17	9
$P_2$	1	4	5	0
$P_3$	2	9	26	15
$P_4$	3	5	10	2

- *Preemptive SJF Gantt Chart*

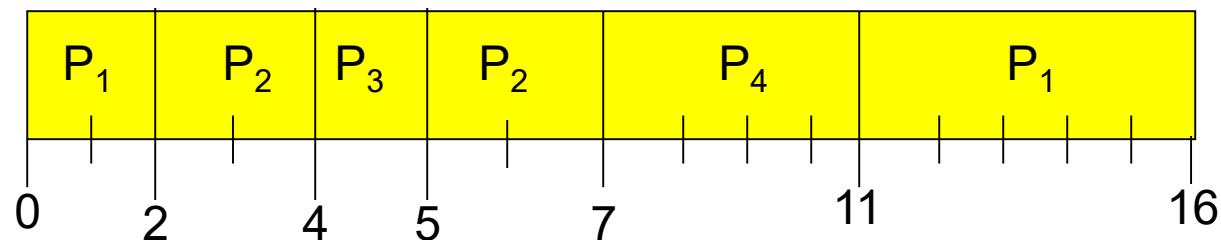


- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec

# Example of Preemptive SJF

Process	Arrival Time	Burst Time
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

- SJF (preemptive)



- Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

# Example of Preemptive SJF

- Example:

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

Process ID	Burst Time	Arrival Time	Turn Around Time	Wait Time
P1	7	0	16	9
P2	4	2	5	1
P3	1	4	1	0
P4	4	5	6	2

# 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
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

## Priority Scheduling - Example

Example 1: Consider the following processes with their burst time and priority. Use priority to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

# Priority Scheduling - Example

- Example 1 (cont'd) :

Process ID	Burst Time	Priority	TurnAround Time	Wait Time
P1	10	3	16	6
P2	1	1	1	0
P3	2	4	18	16
P4	1	5	19	18
P5	5	2	6	1

- Priority scheduling Gantt Chart:



- Average waiting time =  $(6+0+16+18+1)/5 = 8.2$  msec

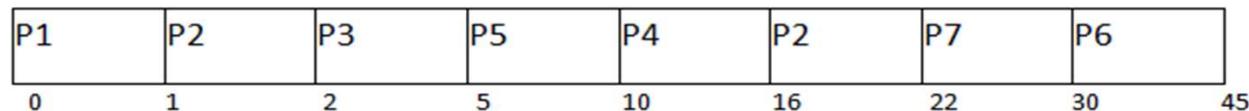
## Priority Scheduling - Example

Example 2: Consider the following processes with their burst time and priority. Use priority to schedule the processes and compute the average waiting time.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	6	8

# Priority Scheduling - Example

- Example 2 (continued): Gantt Chart



Process ID	Priority	Arrival Time	Burst Time	Completion Time	Turn Around Time	Wait Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

- Average Waiting Time =  $(0+14+0+7+1+25+16)/7 = 63/7 = 9$  units

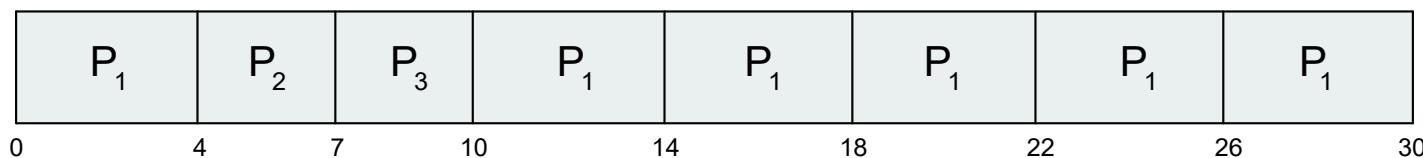
# Round Robin (RR)

- 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

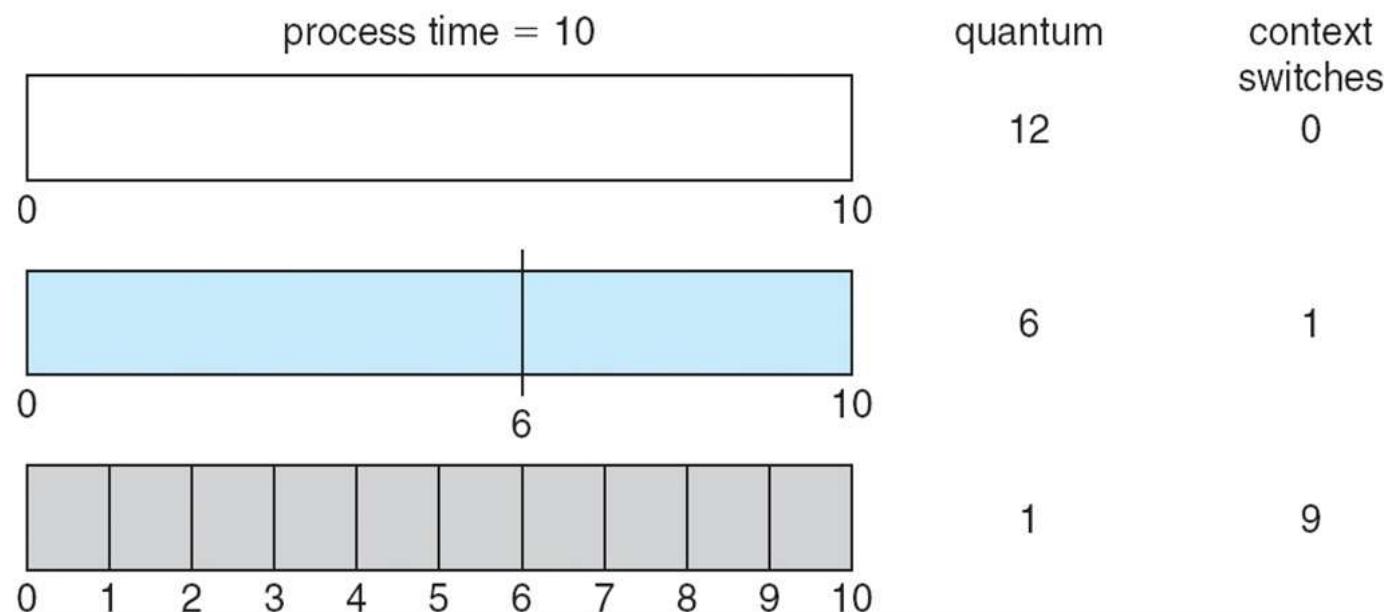
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

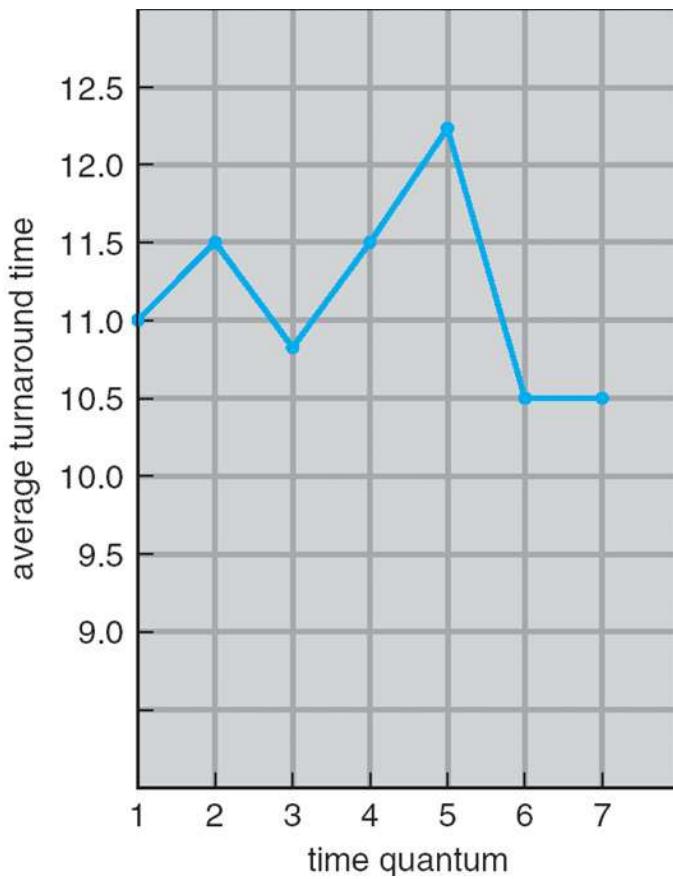


- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



## Turnaround Time Varies With The Time Quantum



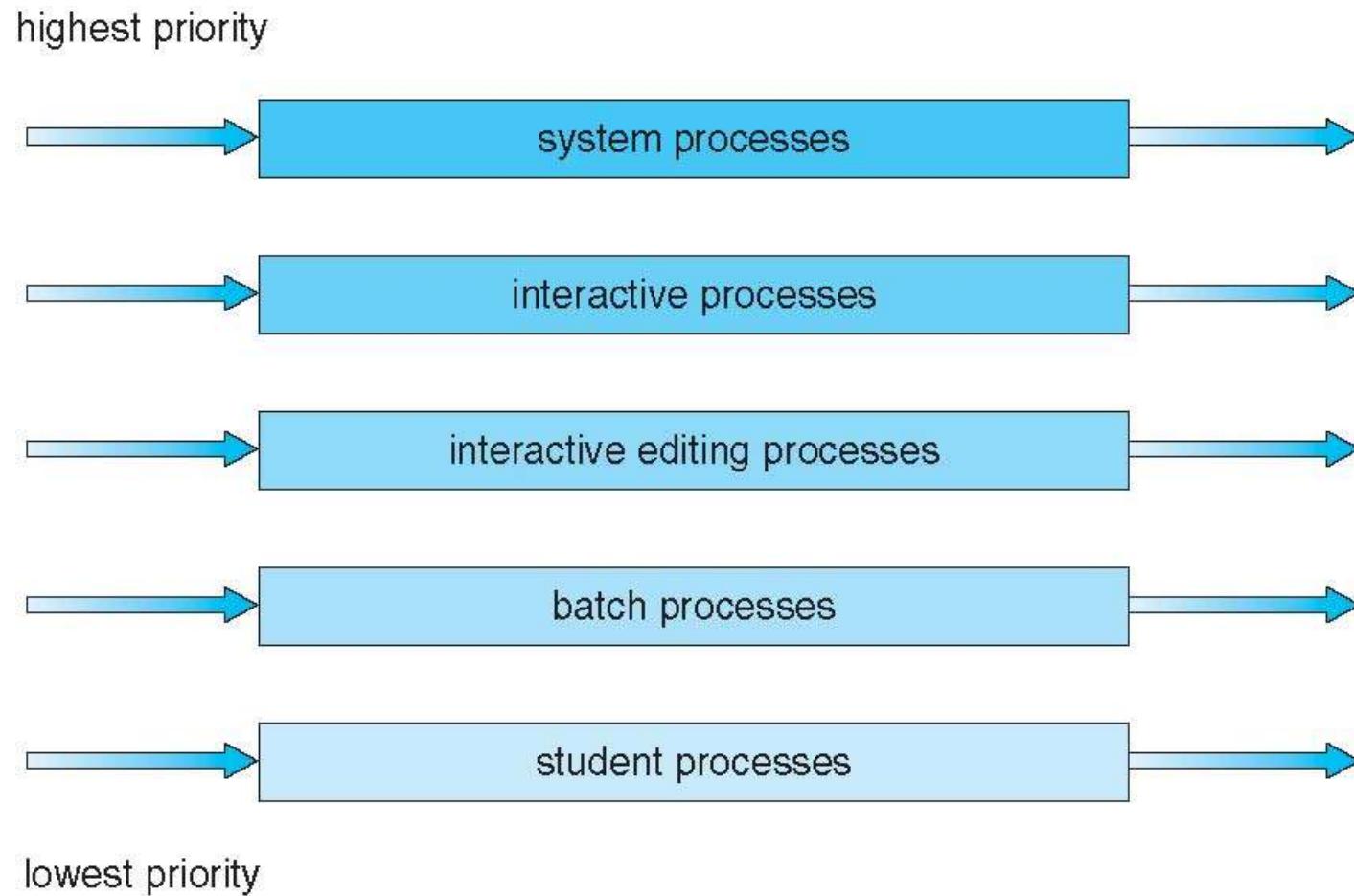
process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts  
should be shorter than  $q$

# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process is permanently in a given queue
- 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

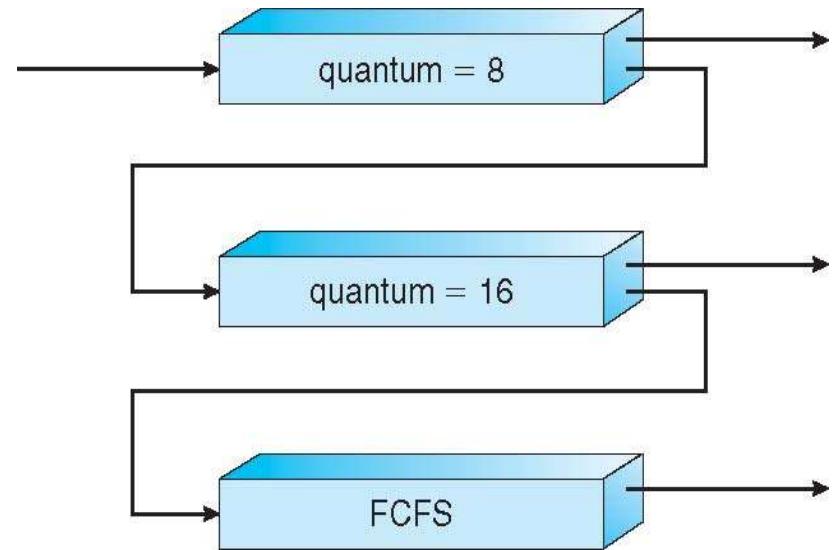


# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler is 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 by 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 by FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue  $Q_2$



# References

For more practice problems

- <https://www.gatevidyalay.com/round-robin-round-robin-scheduling-examples/>

# CPU Scheduling - Algorithms - CS2005

## Module - 3

Example 1:

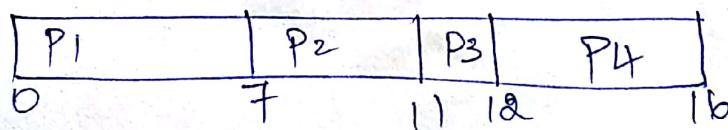
FCFS, SJF, Priority and RoundRobin Scheduling

Consider the following system of processes with their burst times. Present the schedule of processes through a Gantt chart and compute the average waiting time and average turn-around time. Assume that all processes have arrived at the same time,  $t = 0$ .

FCFS scheduling:

Process ID	Burst Time	Wait Time	Turn-Around Time
P <sub>1</sub>	7	0	7
P <sub>2</sub>	4	7	11
P <sub>3</sub>	1	11	12
P <sub>4</sub>	4	12	16

Gantt Chart



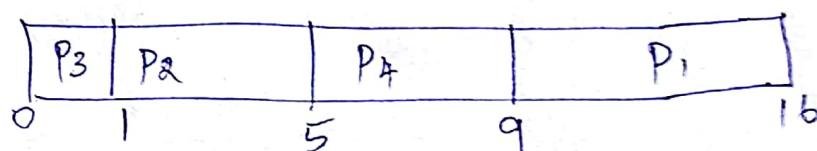
$$\text{Average Waiting Time} = \frac{(0+7+11+12)}{4}$$

$$= \frac{30}{4} = 7.5 \text{ time units}$$

$$\text{Average Turn-Around Time} = \frac{46}{4} \text{ time units}$$

SJF	Process ID	Burst Time	Turnaround Time	Waiting Time
Scheduling	P <sub>1</sub>	7	16	9
	P <sub>2</sub>	4	5	1
	P <sub>3</sub>	1	1	0
	P <sub>4</sub>	4	9	5

Grantl chart

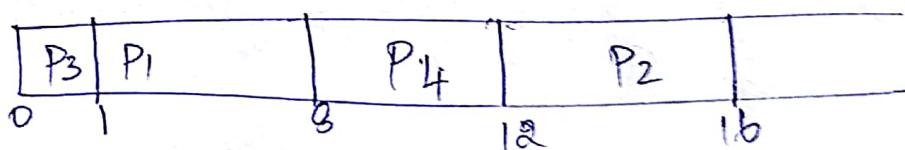


$$\text{Average Waiting Time} = \frac{15}{4} = 3.75 \text{ time unit}$$

$$\text{Average Turnaround Time} = \frac{31}{4} = 7.75 \text{ time unit}$$

Priority Scheduling : Priority is {2, 1, 4, 1, 2}

Process ID	Burst Time	Priority	Turnaround Time	Wait Time
P <sub>1</sub>	7	2	8	1
P <sub>2</sub>	4	4	16	12
P <sub>3</sub>	1	1	1	0
P <sub>4</sub>	4	2	12	8



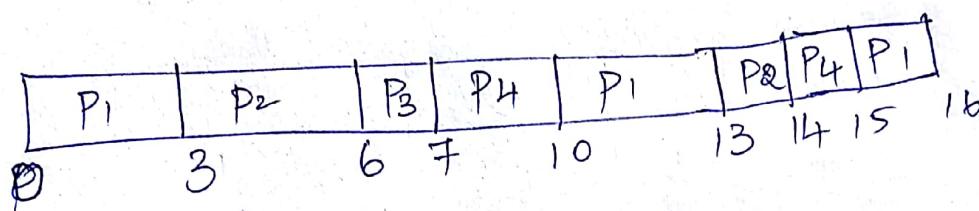
$$\text{Average Waiting Time} = \frac{21}{4} = 5.25 \text{ time unit}$$

$$\text{Average Turnaround Time} = \frac{37}{4} = 9.25 \text{ time unit}$$

## Round-Robin (RR) Scheduling

Time Quantum = 3 time units

Process ID	Burst Time	Turn Around Time	Waiting Time
P <sub>1</sub>	7	16	9
P <sub>2</sub>	4	14	10
P <sub>3</sub>	1	7	6
P <sub>4</sub>	4	15	11



$$\text{Average Waiting Time} = \frac{(9+10+6+1)}{4} = \frac{36}{4}$$

$$= 9 \text{ time unit}$$

$$\text{Average Turnaround Time} = \frac{52}{4} = 13 \text{ time unit}$$

∴ For the given system of 4 processes burst time, priority and time quantum = 3 time units, SJF algorithm has suffered from the least average waiting time of 3.75 time unit.

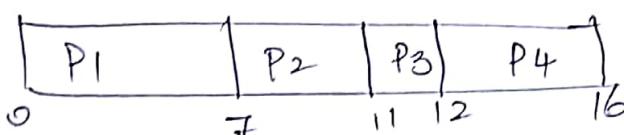
— X —

Example 2: Consider the system of processes as given in Example 1 with different arrival times of processes and apply all algorithms. Further provide your conclusions on average waiting time.

### PCFS scheduling:

Process ID	Burst Time	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	0	7	0
P <sub>2</sub>	4	1	10	6
P <sub>3</sub>	1	2	10	9
P <sub>4</sub>	4	3	13	9

### Gantt chart:

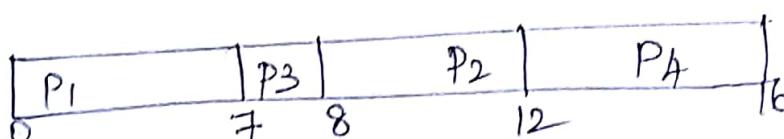


$$\text{Average Waiting Time} = \frac{(0+6+9+9)}{4} = 6 \text{ time units}$$

### SJF scheduling (without preemption)

Process ID	Burst Time	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	0	7	0
P <sub>2</sub>	4	1	11	7
P <sub>3</sub>	1	2	6	5
P <sub>4</sub>	4	3	13	9

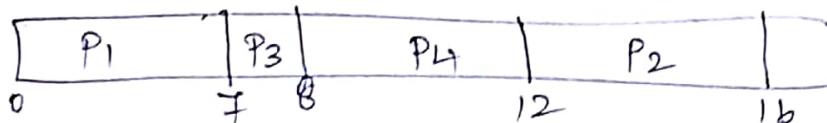
### Gantt Chart:



$$\text{Average Waiting Time} = \frac{(0+7+5+9)}{4} = 5.25 \text{ time units}$$

## Priority Scheduling (non-preemptive) :

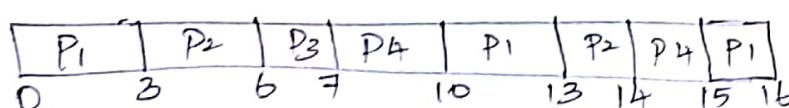
Process ID	Burst Time	Priority	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	2	0	7	0
P <sub>2</sub>	4	4	1	15	11
P <sub>3</sub>	1	1	2	6	5
P <sub>4</sub>	4	2	3	9	5



$$\text{Average Waiting Time} = (0+11+5+5)/4 = 5.25 \text{ time units}$$

Round Robin Scheduling with Time Quantum = 3 time units :

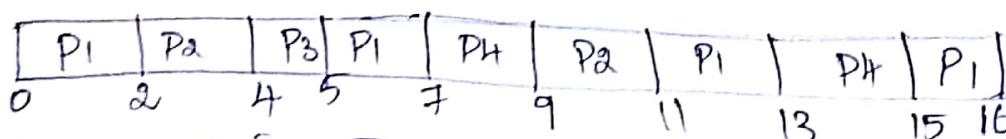
Process ID	Burst Time	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	0	16	9
P <sub>2</sub>	4	1	13	9
P <sub>3</sub>	1	2	5	4
P <sub>4</sub>	4	3	12	8



$$\text{Average Waiting Time} = (9+9+4+8)/4 = 7.5 \text{ time units}$$

Round-Robin Scheduling with Time Quantum = 2 time units :

Process ID	Burst Time	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	0	16	9
P <sub>2</sub>	4	1	10	6
P <sub>3</sub>	1	2	3	2
P <sub>4</sub>	4	3	12	8



$$\text{Average Waiting Time} = (9+6+2+8)/4 = 25/4 = 6.25 \text{ time units}$$

Rough Block:

Arrival/Re-arrival  $\rightarrow 0 \ 1 \ 2 \ 2 \ 3 \ 4 \ 7 \ 9 \ 13$

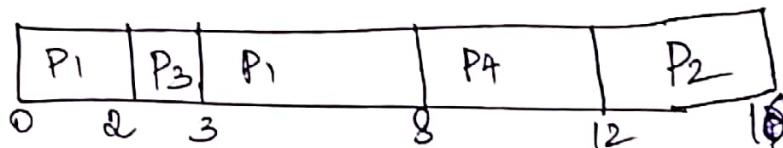
Time  $\rightarrow P_1 \ P_2 \ P_3 \ P_1 \ P_4 \ P_2 \ P_1 \ P_4 \ P_1$

Remaining Burst  $\rightarrow 7 \ 4 \ 1 \ 5 \ 4 \ 2 \ 3 \ 2 \ 1$

## Priority Scheduling with preemption:

Process ID	Burst Time	Priority	Arrival Time	Turnaround Time	Wait Time
P <sub>1</sub>	7	2	0	8	1
P <sub>2</sub>	4	4	1	15	11
P <sub>3</sub>	1	1	2	1	0
P <sub>4</sub>	4	2	3	9	5

Grant chart :

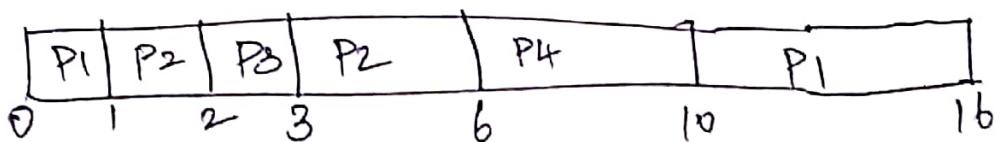


$$\text{Average Waiting Time} = (1+11+0+5)/4 = 17/4 = 4.25 \text{ time units}$$

Shortest Remaining Time First/Next (SRTF/SRTN) | SJF with preemption ) !

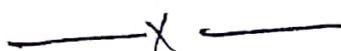
Process ID	Burst Time	Arrival Time	Turnaround Time	Waiting Time
P <sub>1</sub>	7	0	16	9
P <sub>2</sub>	4	1	5	1
P <sub>3</sub>	1	2	1	0
P <sub>4</sub>	4	3	7	3

Grant chart :



$$\text{Average Waiting Time} = (9+1+0+3)/4 = 13/4$$

= 3.25 time units



# Module3\_CPU\_Scheduling

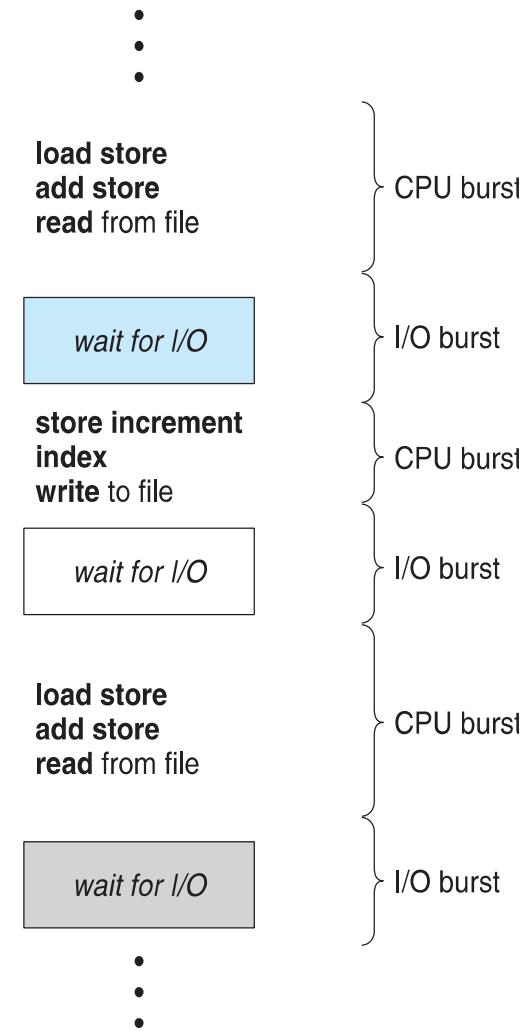
Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Objectives

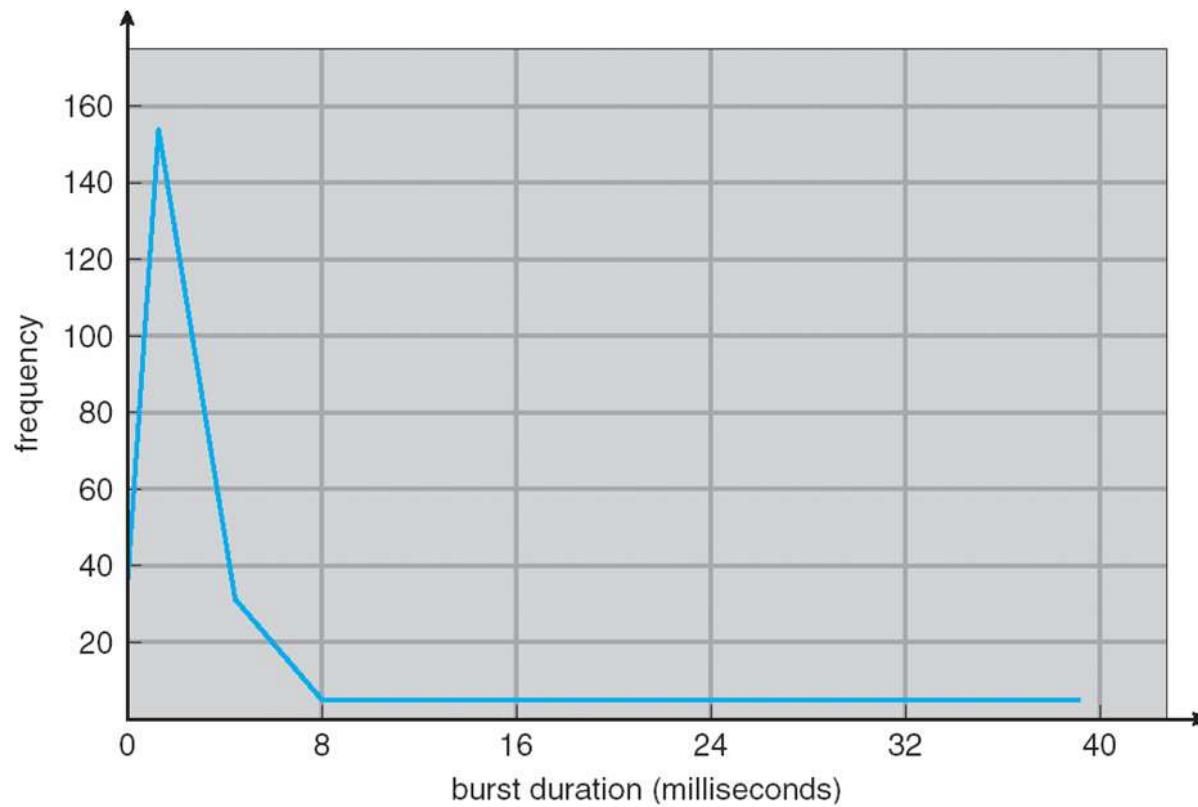
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

# Basic Concepts

- Maximum CPU utilization is 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
- **CPU Burst Time:** The amount of time a process executes before it goes to wait state

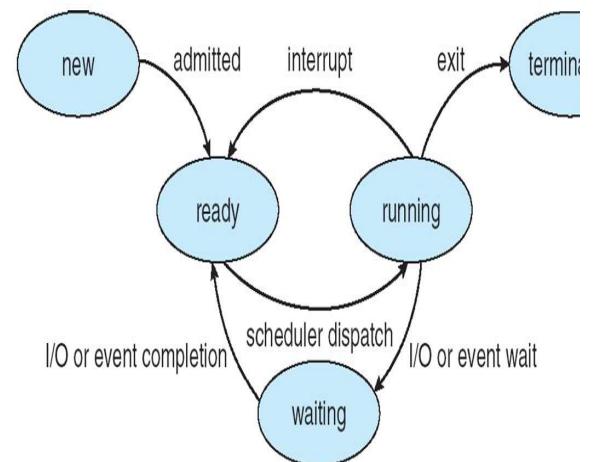


# Histogram of CPU-burst Times



# CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



# 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 taken by 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
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time taken from when a request was submitted until the first response is produced, not output (for time-sharing environment)

# Scheduling Algorithm Optimization Criteria

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

uptime -> Linux command  
system calls  
time  
`sched_getscheduler()`  
`sched_setscheduler()`

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

Example 1: Consider the following processes with their burst time. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

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

### Example 1: Continued

The Gantt Chart for the schedule is:



Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	24	24	24	0
P2	3	27	27	24
P3	3	30	30	27

- Waiting time for P<sub>1</sub> = 0; P<sub>2</sub> = 24; P<sub>3</sub> = 27
- Average waiting time:  $(0 + 24 + 27)/3 = 17$ 
  - Turnaround Time = Completion Time - Arrival Time
  - Waiting Time = Turn Around Time - Burst Time

# FCFS Scheduling (Cont.)

## Example 1: Continued

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

Process ID	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

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
- Performance of FCFS is dynamic
  - **Convoy effect** - short process behind long process
    - Consider one CPU-bound and many I/O-bound processes

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

- Example 2

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P <sub>1</sub>	0	7
P <sub>2</sub>	0	4
P <sub>3</sub>	0	1
P <sub>4</sub>	0	4

❑ Gantt chart

Schedule: P<sub>1</sub>

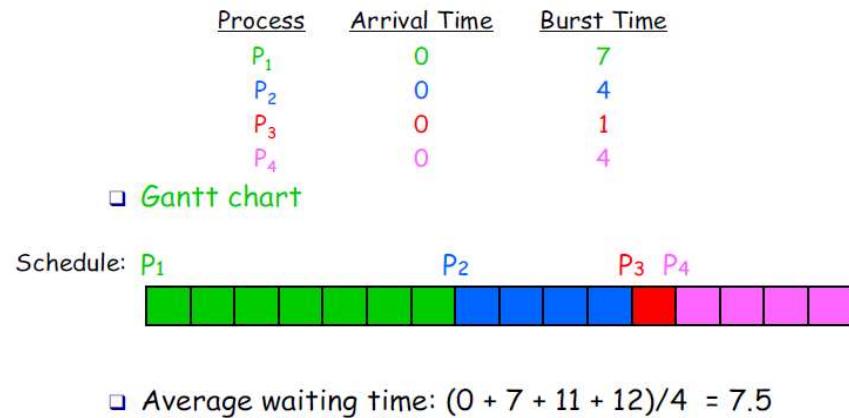


❑ Average waiting time:  $(0 + 7 + 11 + 12)/4 = 7.5$

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

### Example 2: Continued

The Gantt Chart for the schedule is:



Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	7	7	7	0
P2	4	11	11	7
P3	1	12	12	11
P4	4	16	16	12

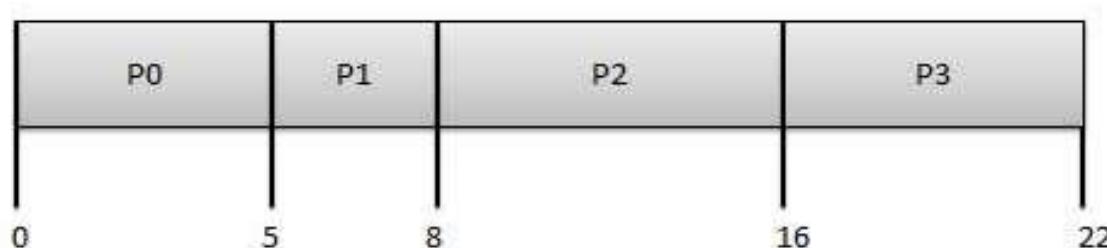
- Average waiting time:  $(0 + 7 + 11 + 12)/4 = 7.5$

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

Example 3: Consider the following processes with their **burst time and arrival time**. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	BurstTime	ArrivalTime
P0	5	0
P1	3	1
P2	8	2
P3	6	3

The **Gantt Chart** for the schedule is:



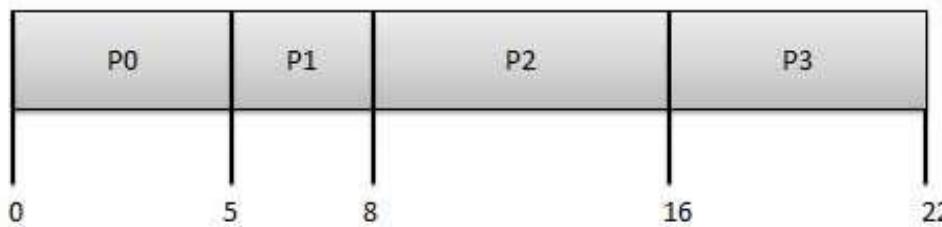
- Average Waiting Time:  $(0+4+6+13) / 4 = 5.75$

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

Example 3 (cont'd):

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P0	5	0	5	5	0
P1	3	1	8	7	4
P2	8	2	16	14	6
P3	6	3	22	19	13

The Gantt Chart for the schedule is:



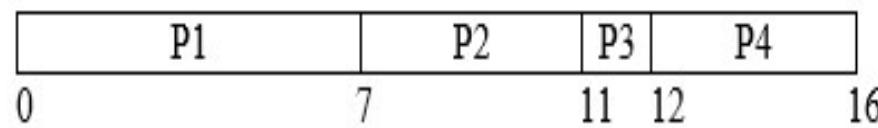
- Average Waiting Time:  $(0+4+6+13) / 4 = 5.75$

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

Example 4: Consider the following processes with their burst time and arrival time. Use FCFS to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

The Gantt Chart for the schedule is:



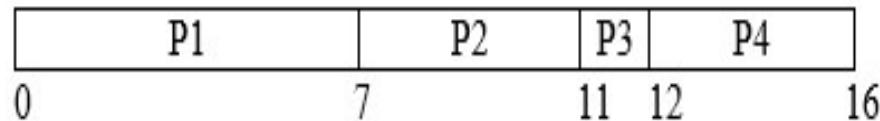
- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

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

Example 4: (continued)

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P1	7	0	7	7	0
P2	4	2	11	9	5
P3	1	4	12	8	7
P4	4	5	16	11	7

The Gantt Chart for the schedule is:



- Average waiting time =  $(0 + 5 + 7 + 7)/4 = 4.75$

# Shortest-Job-First (SJF) Scheduling

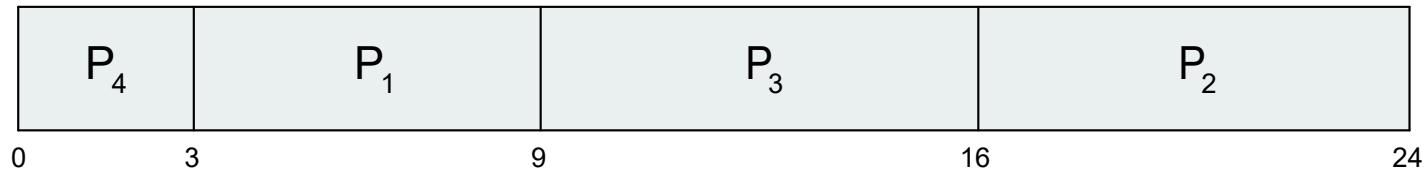
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

# SJF – Example 1

Example 1: Consider the following processes with their burst time. Use SJF to schedule the processes and compute the average waiting time.

Process ID	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



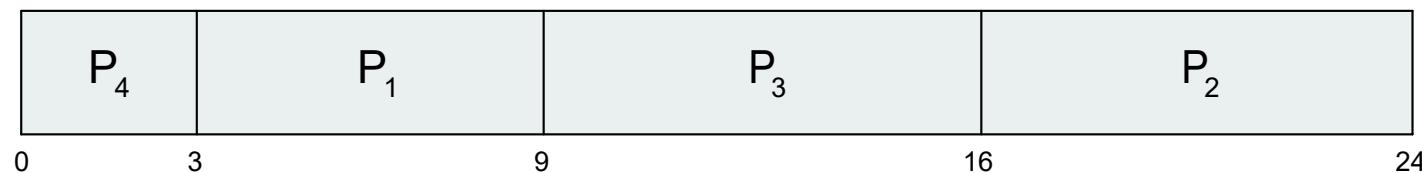
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# SJF – Example 1 (cont'd)

- Scheduling Table

Process ID	Burst Time	Completion Time	TurnAround Time	Wait Time
P1	6	9	9	3
P2	8	24	24	16
P3	7	16	16	9
P4	3	3	3	0

- SJF scheduling chart



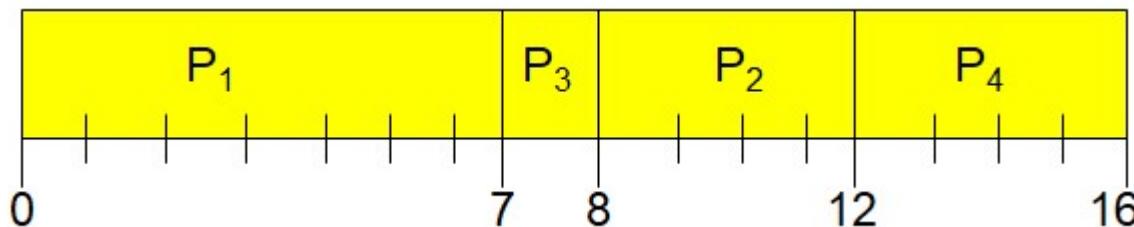
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

## SJF Scheduling - Example

Example 2: Consider the following processes with their burst time and arrival time. Use SJF to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

The Gantt Chart for the schedule is:



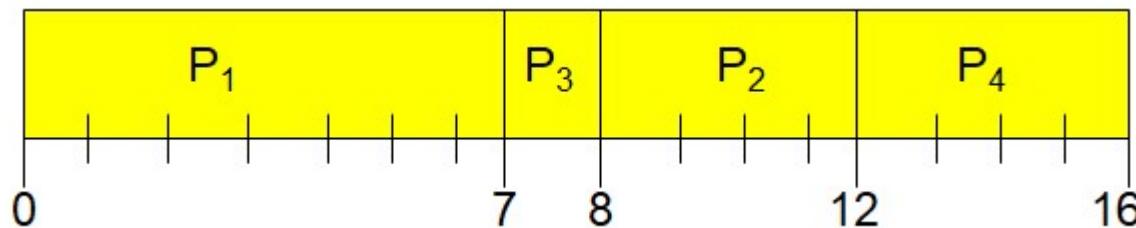
- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

## SJF Scheduling - Example

### Example 2: (continued)

Process ID	Burst Time	Arrival Time	Completion Time	TurnAround Time	Wait Time
P1	7	0	7	7	0
P2	4	2	12	10	6
P3	1	4	8	4	3
P4	4	5	16	11	7

The Gantt Chart for the schedule is:

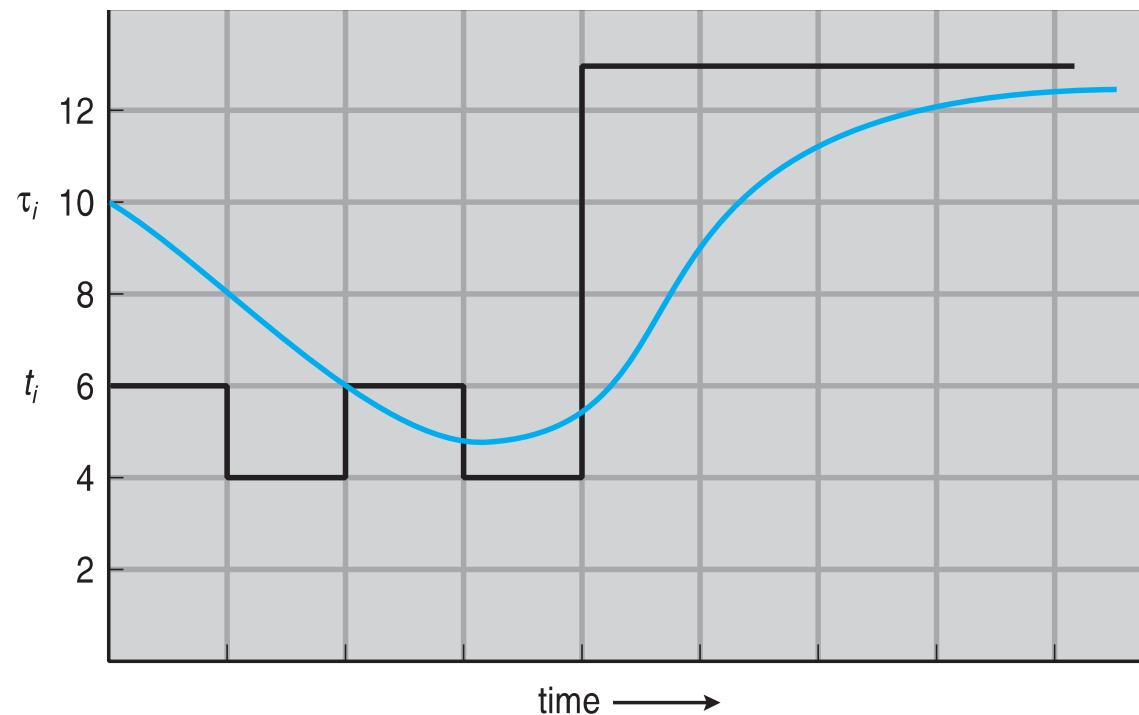


- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# 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
  1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define:  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
- Commonly,  $\alpha$  set to  $\frac{1}{2}$
- Preemptive version of SJF is called **Shortest-Remaining-Time-First (SRTF)**

## Prediction of the Length of the Next CPU Burst



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12

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

# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- $\alpha = 1$ 
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

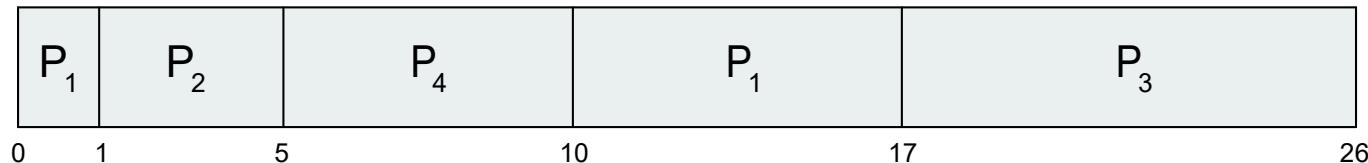
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **Shortest-Remaining-Time-First (SRTF)**
- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Completion Time</u>	<u>Wait Time</u>
$P_1$	0	8	17	9
$P_2$	1	4	5	0
$P_3$	2	9	26	15
$P_4$	3	5	10	2

- *Preemptive SJF Gantt Chart*

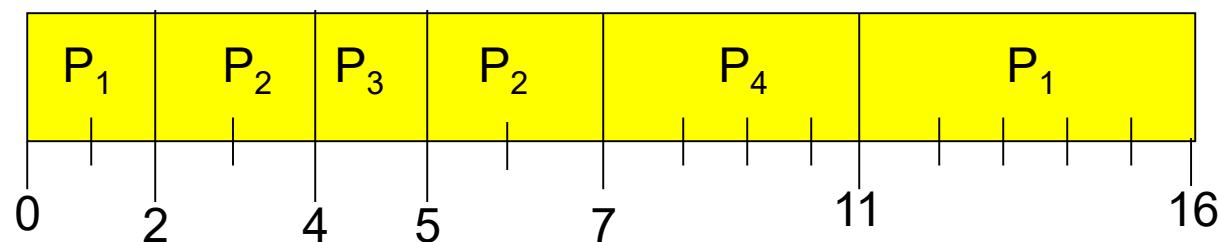


- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec

# Example of Preemptive SJF

Process	Arrival Time	Burst Time
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

- SJF (preemptive)



- Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

# Example of Preemptive SJF

- Example:

Process ID	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5

Process ID	Burst Time	Arrival Time	Turn Around Time	Wait Time
P1	7	0	16	9
P2	4	2	5	1
P3	1	4	1	0
P4	4	5	6	2

# 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
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

## Priority Scheduling - Example

Example 1: Consider the following processes with their burst time and priority. Use priority to schedule the processes and compute the average waiting time.

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

# Priority Scheduling - Example

- Example 1 (cont'd) :

Process ID	Burst Time	Priority	TurnAround Time	Wait Time
P1	10	3	16	6
P2	1	1	1	0
P3	2	4	18	16
P4	1	5	19	18
P5	5	2	6	1

- Priority scheduling Gantt Chart:



- Average waiting time =  $(6+0+16+18+1)/5 = 8.2$  msec

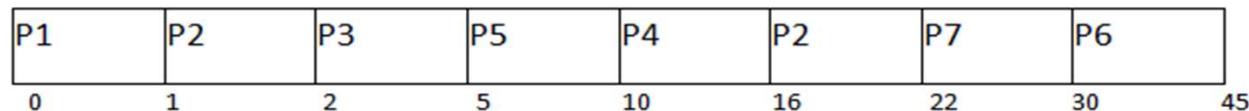
## Priority Scheduling - Example

Example 2: Consider the following processes with their burst time and priority. Use priority to schedule the processes and compute the average waiting time.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	6	8

# Priority Scheduling - Example

- Example 2 (continued): Gantt Chart



Process ID	Priority	Arrival Time	Burst Time	Completion Time	Turn Around Time	Wait Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

- Average Waiting Time =  $(0+14+0+7+1+25+16)/7 = 63/7 = 9$  units

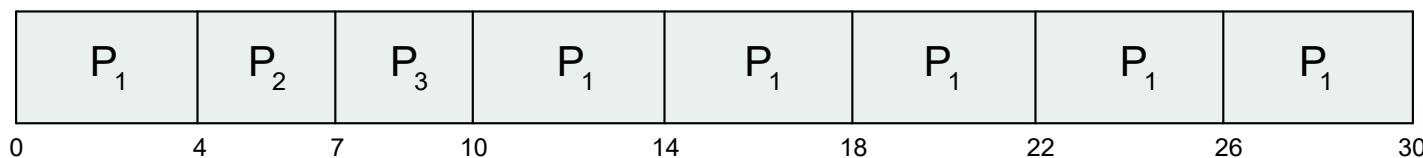
# Round Robin (RR)

- 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

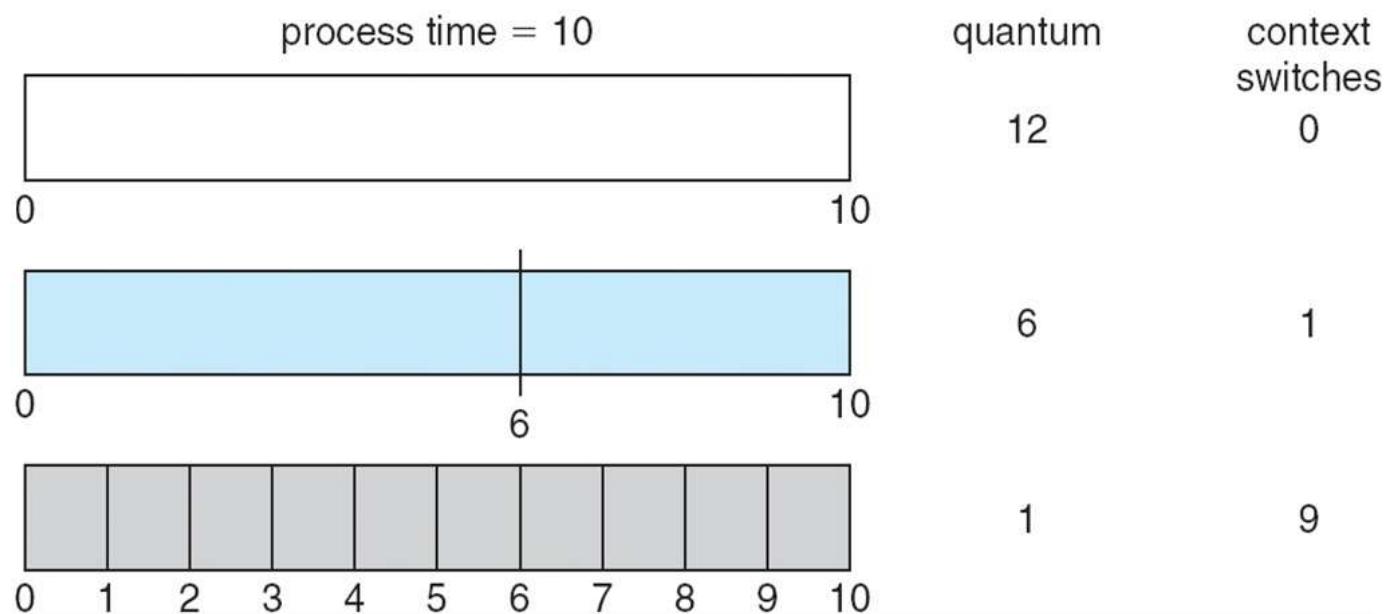
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

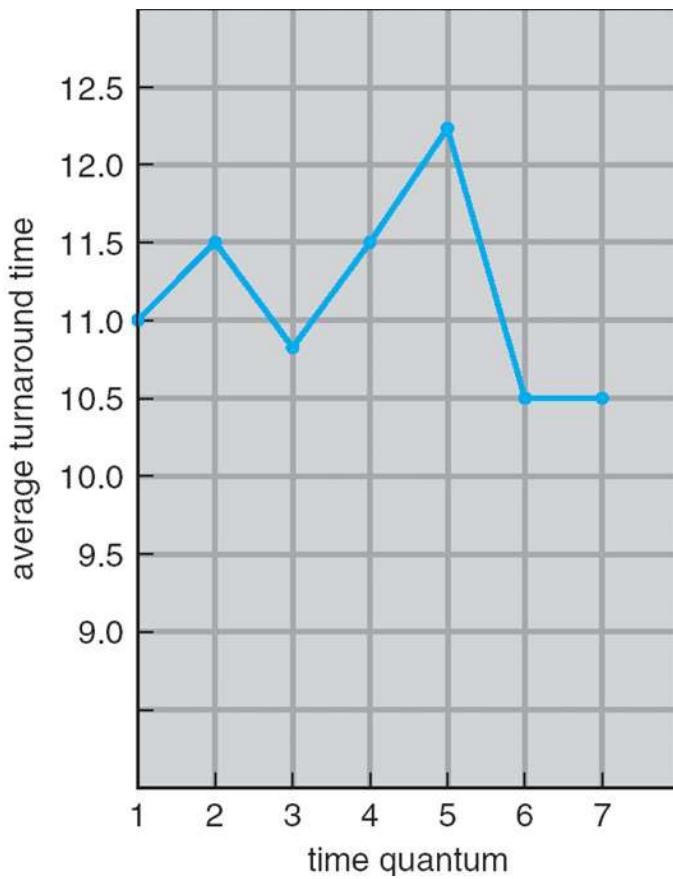


- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



## Turnaround Time Varies With The Time Quantum



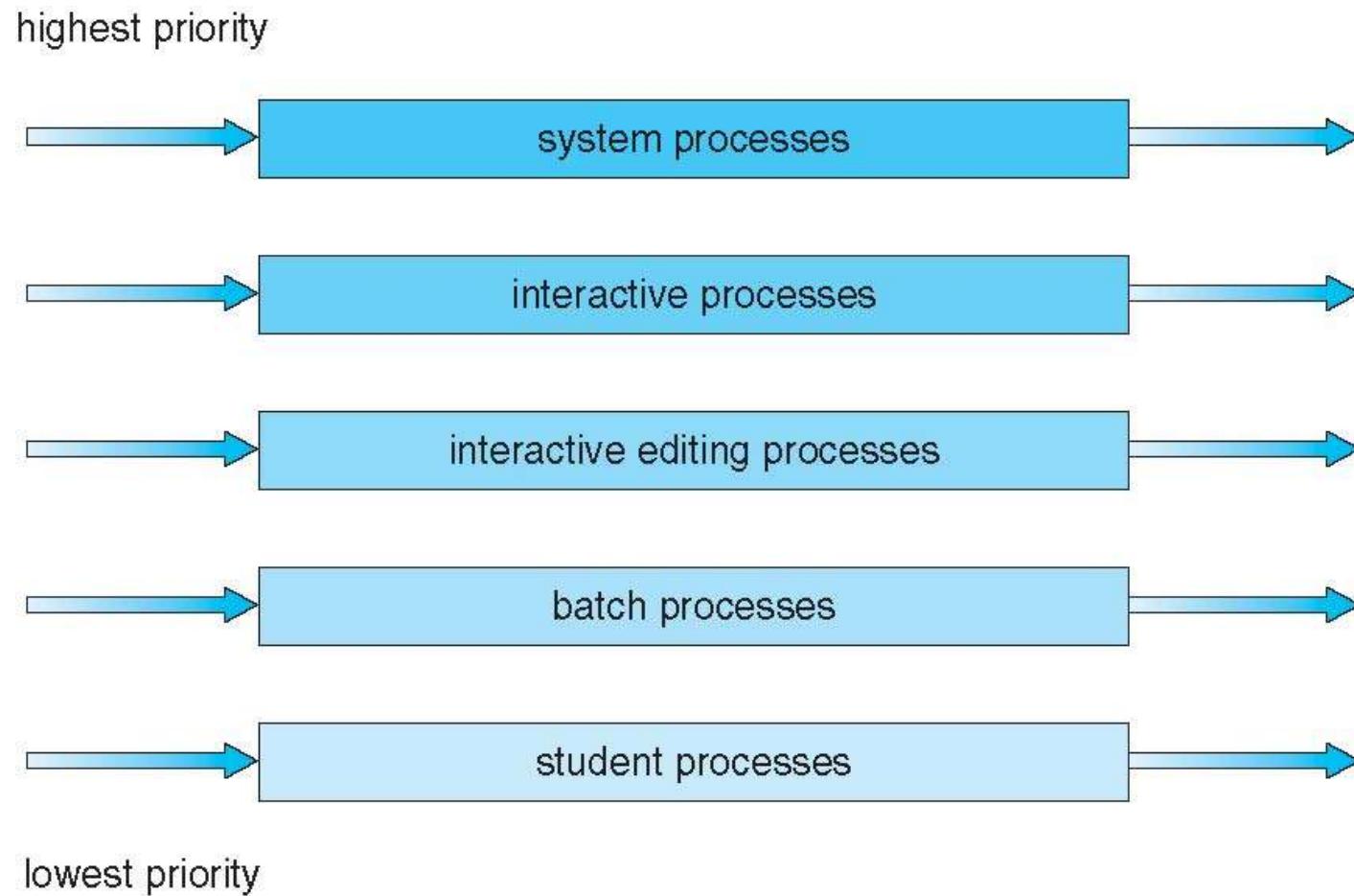
process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts  
should be shorter than  $q$

# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process is permanently in a given queue
- 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

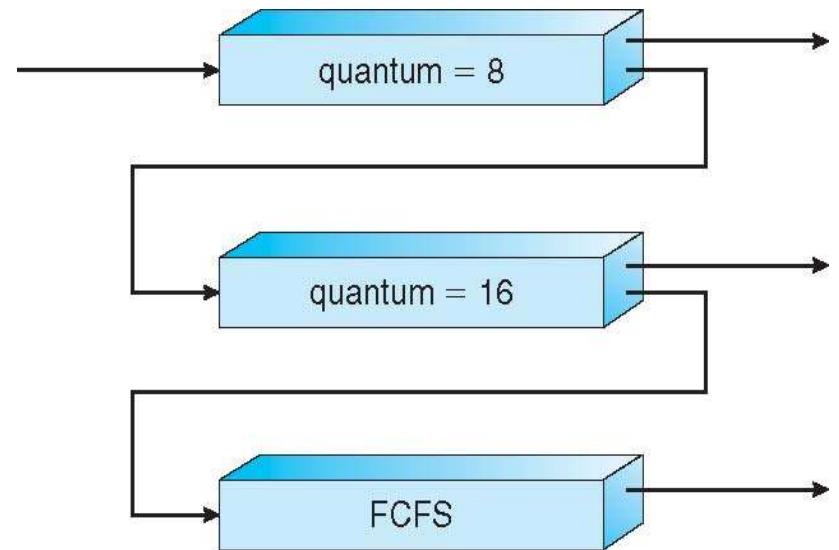


# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler is 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 by 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 by 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
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

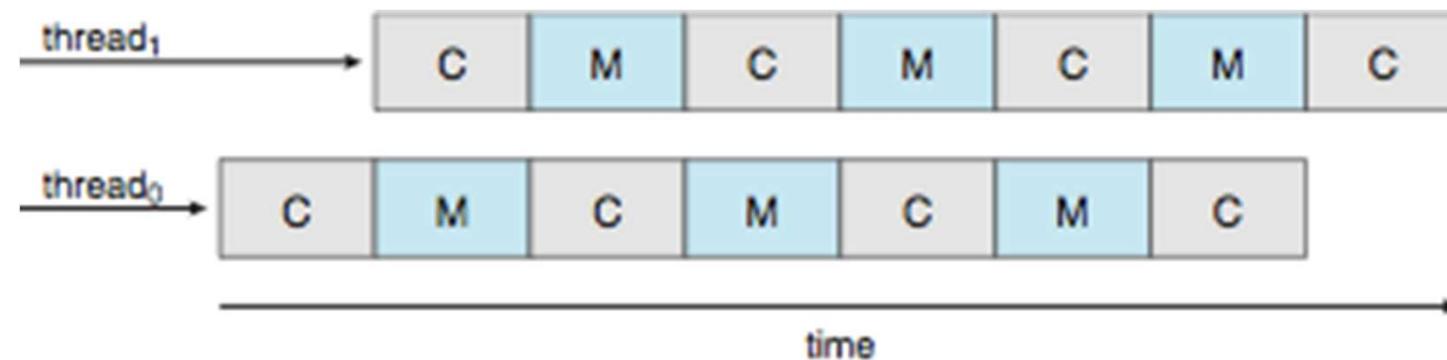
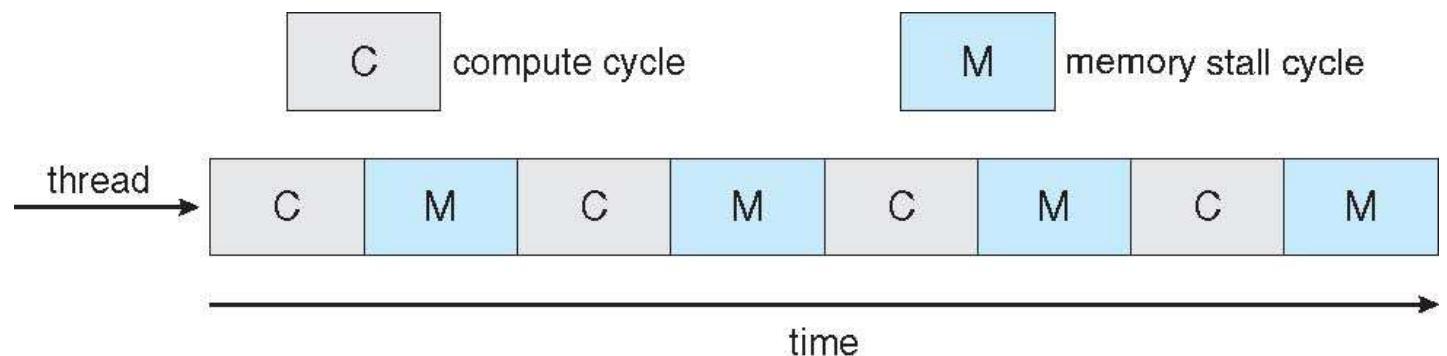
## Multiple-Processor Scheduling – Load Balancing

- 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

# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core is also growing
  - Takes advantage of **memory stall** to make progress on another thread while memory retrieve happens

# Multithreaded Multicore System



# Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

# Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

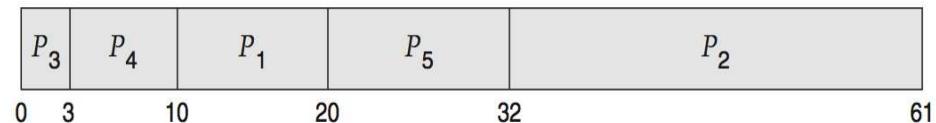
Process	Burst Time
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12

# Deterministic Evaluation

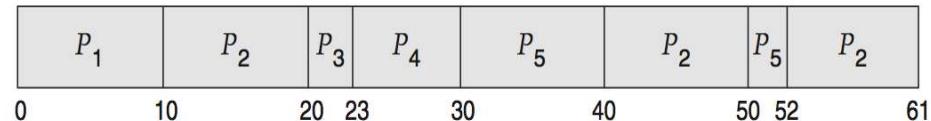
- ❑ For each algorithm, calculate minimum average waiting time
- ❑ Simple and fast, but requires exact numbers for input, applies only to those inputs
  - ❑ FCFS is 28ms:



- ❑ Non-preemptive SJF is 13ms:



- ❑ RR is 23ms:



# Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

# Little's Formula

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:  
$$n = \lambda \times W$$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on an average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

# Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

# References

For more practice problems

- <https://www.gatevidyalay.com/round-robin-round-robin-scheduling-examples/>

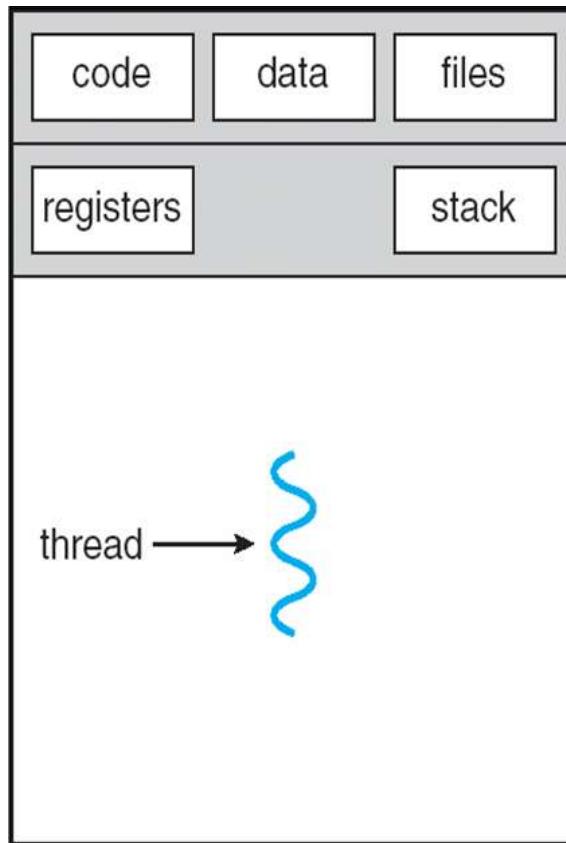
# Module2\_Threads

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Thread Overview

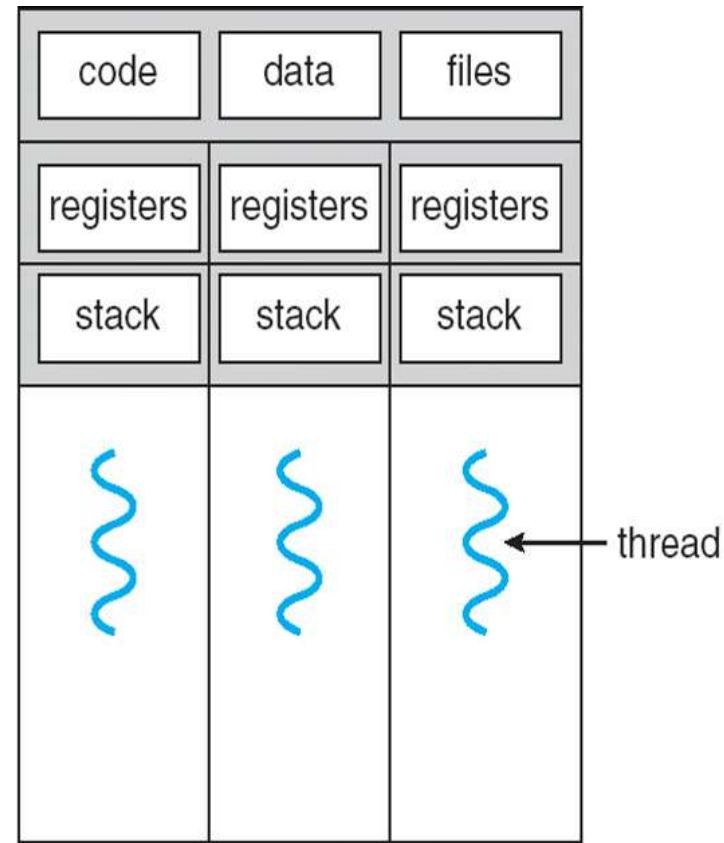
- Thread: A **fundamental unit of CPU utilization** that forms the basis of multithreaded computer systems
- Threads are mechanisms that permit an application to perform multiple tasks concurrently.
- Thread is a basic unit of CPU utilization
  - Thread ID
  - Program counter
  - Register set
  - Stack
- A single program can contain multiple threads
  - Threads share with other threads belonging to the same process (hence a thread is a Light Weight Process (LWP))
    - Code, data, open files

# Single and Multithreaded Processes



single-threaded process

**heavyweight** process



multithreaded process

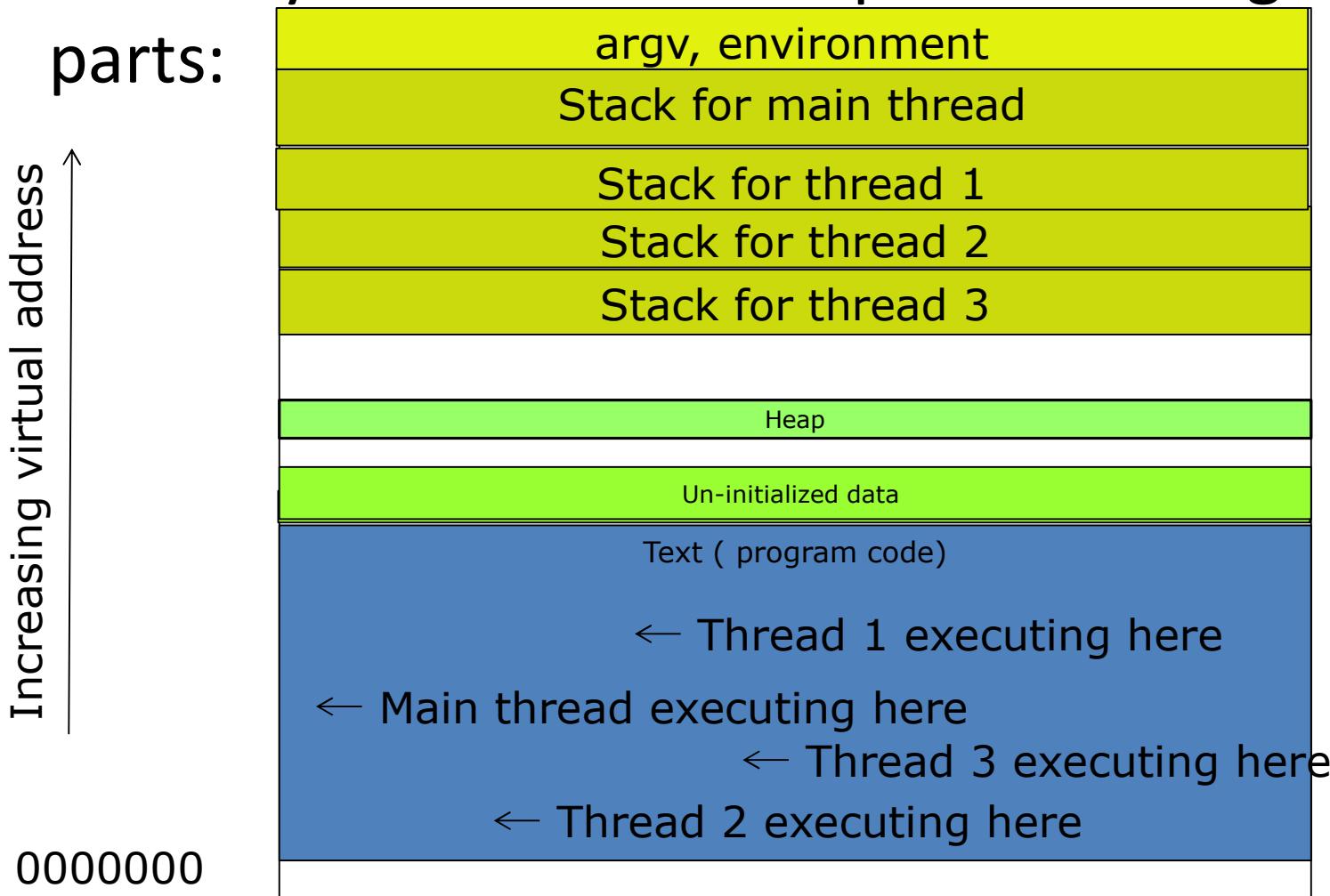
**lightweight** process

Traditional (**heavyweight**) process has a single thread of control

# Threads in Memory

Memory is allocated for a process in segments or

parts:



# 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)

# Benefits

- **Responsiveness**

Interactive application can delegate background functions to a thread and keep running

- **Resource Sharing**

Several threads can access the same address space

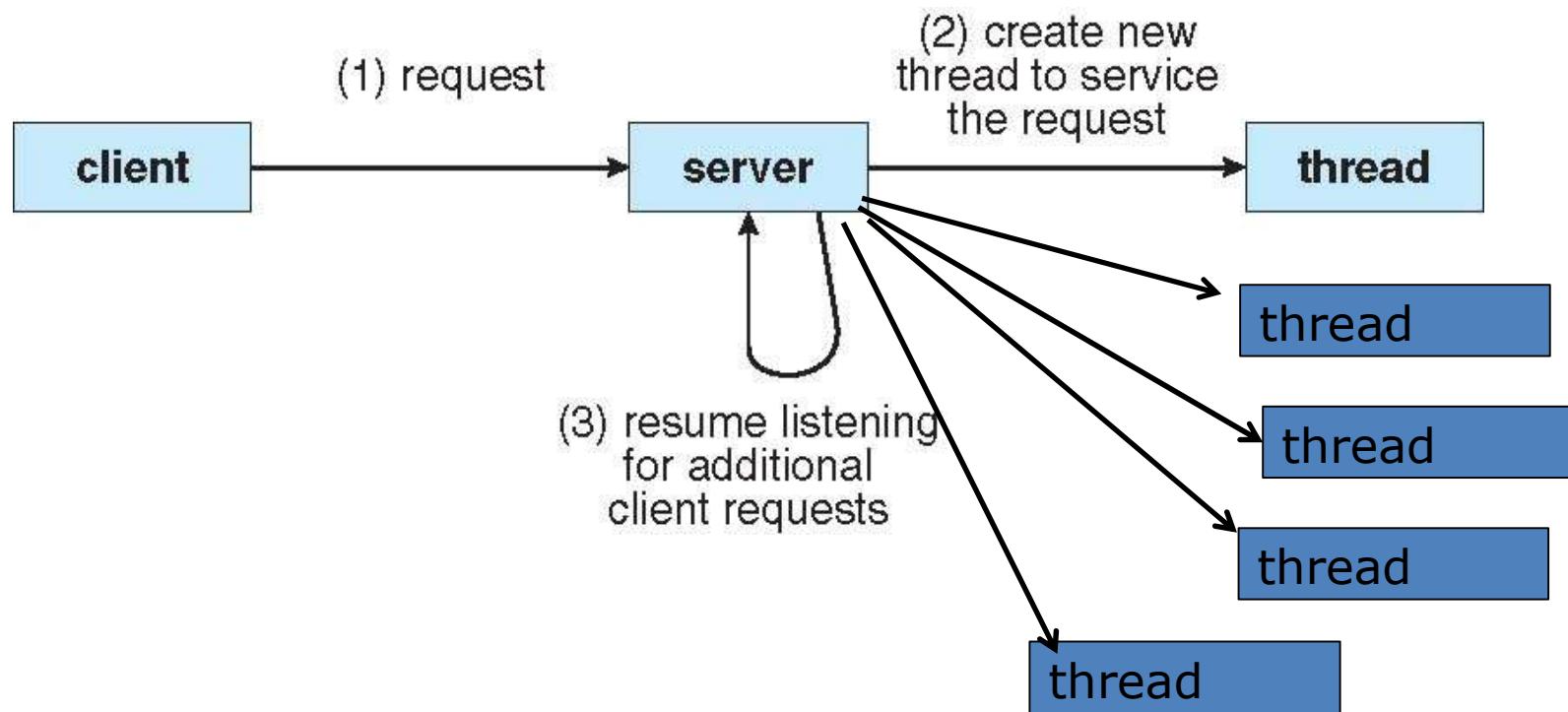
- **Economy**

Allocating memory and new processes is costly. Threads are much ‘cheaper’ to initiate.

- **Scalability**

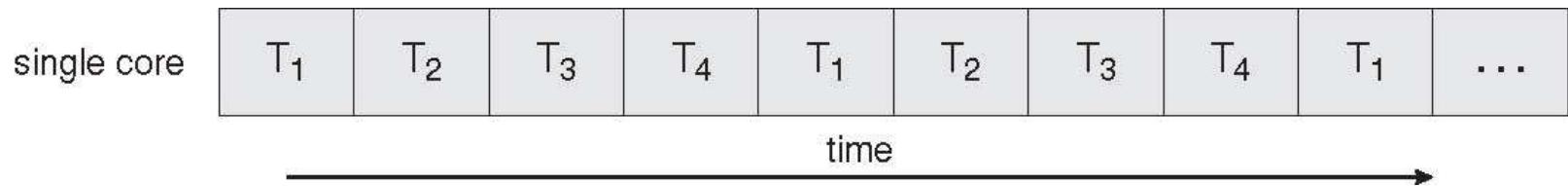
Use threads to take advantage of multiprocessor architecture

# Multithreaded Server Architecture

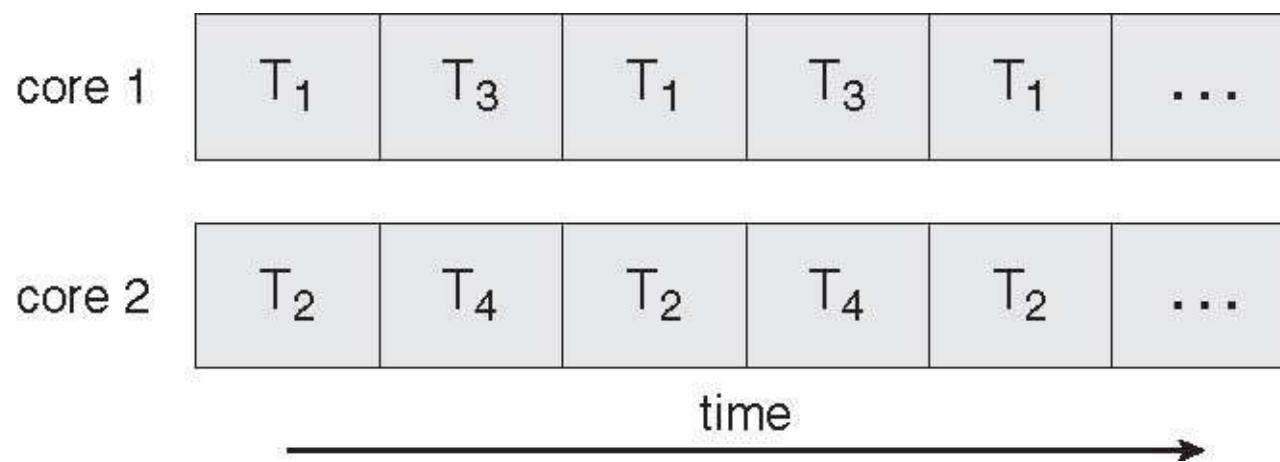


# Threads and Multicore Programming

## Concurrent Execution on a Single-core System



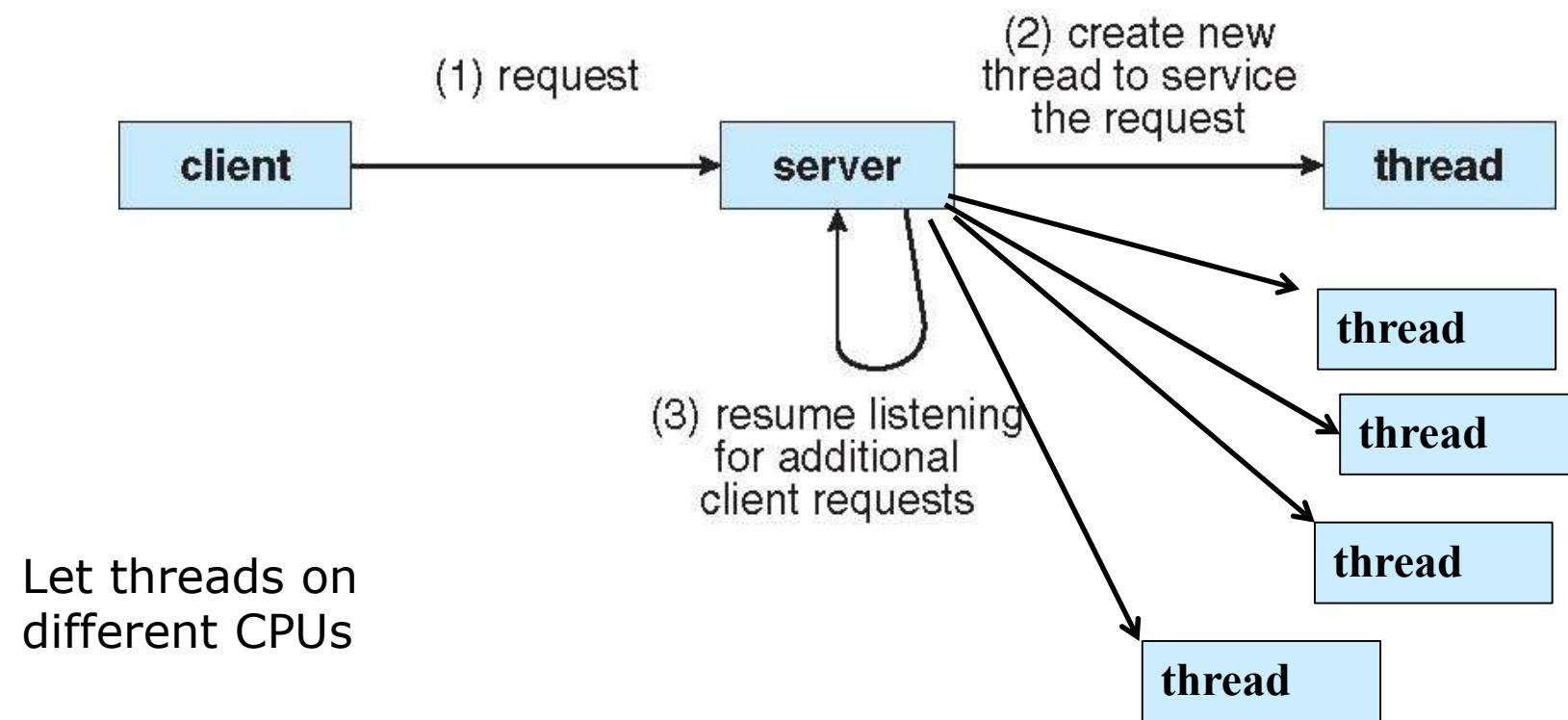
## Parallel Execution on a Multicore System



# Threads and Multicore Programming

- Multicore systems place pressure on programmers. Some of the 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!!!!

# Threads Assist Multicore Programming



# 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

# 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

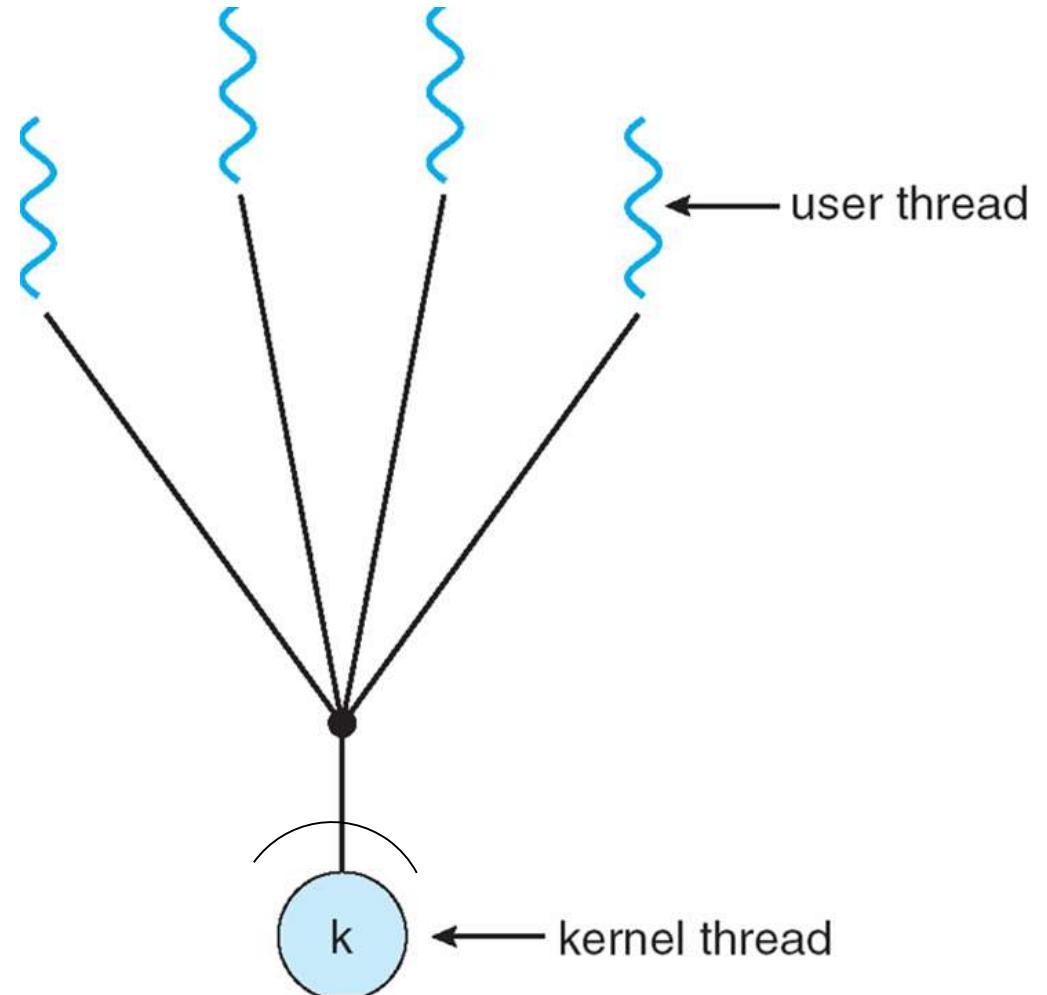
# Multithreading Models

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

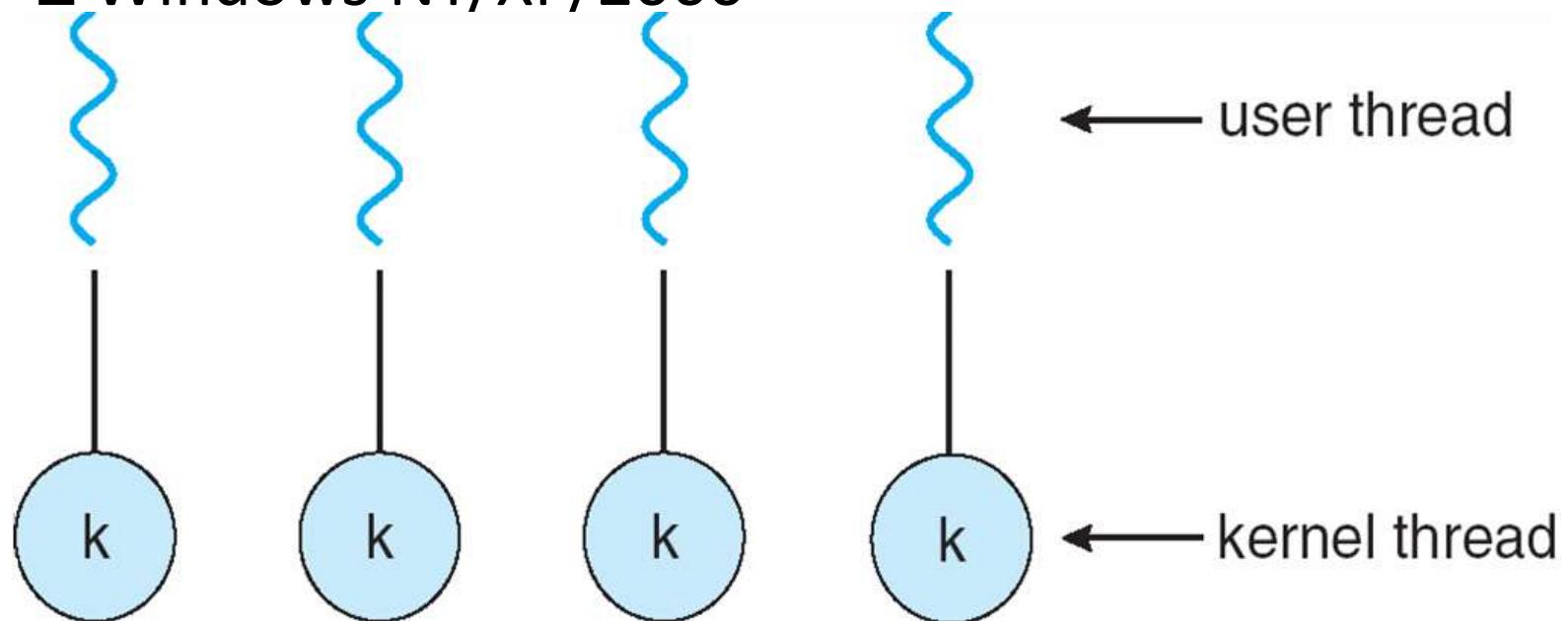


# One-to-One

Each user-level thread maps to kernel thread

## □ Examples

### □ Windows NT/XP/2000



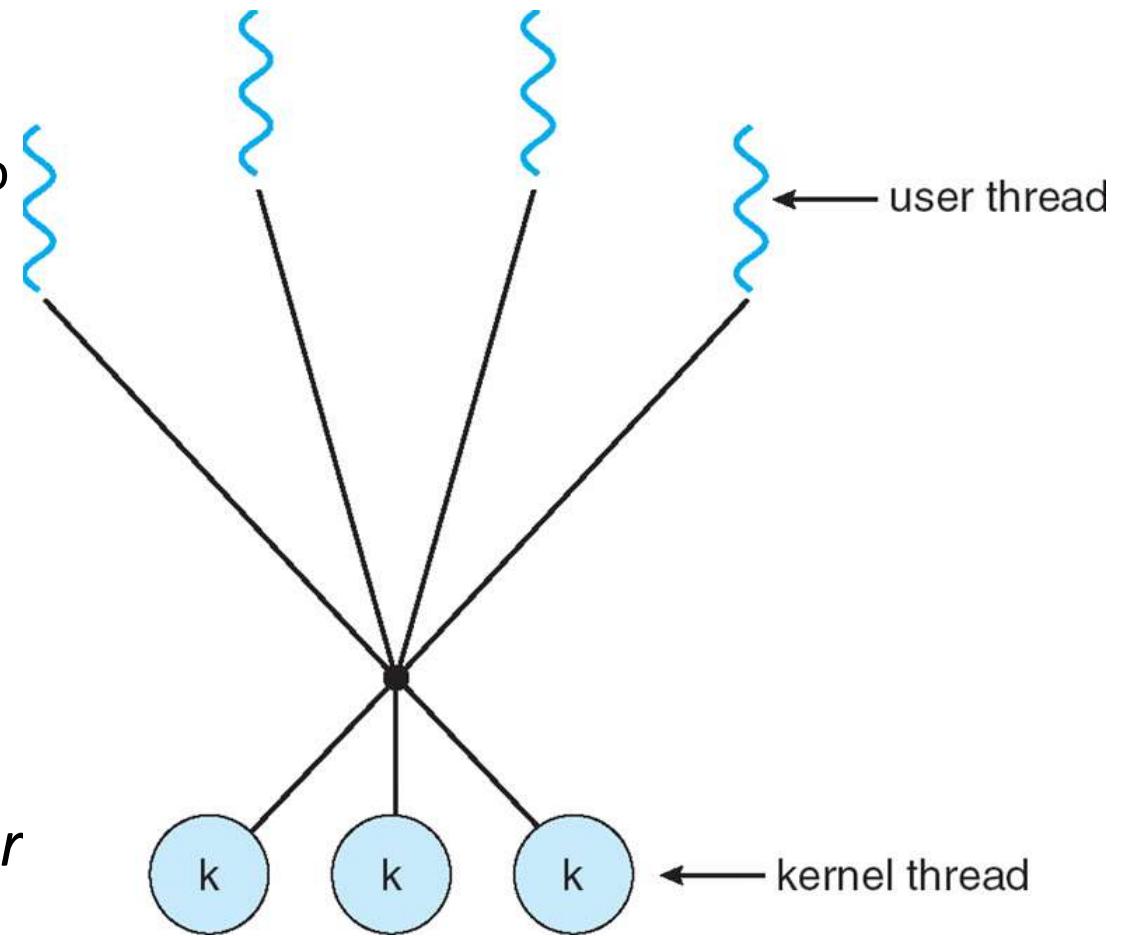
# Many-to-Many Model

Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads

## □ Example

□ Windows NT/2000 with the *ThreadFiber* package



# Thread Libraries

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

# Thread Libraries

- Three main thread libraries in use :
  - **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***

# POSIX: Thread Creation

```
#include <pthread.h>

pthread_create (thread, attr, start_routine, arg)
```

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

Literally:

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

# 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.

- The thread calling `pthread_join` waits for the completion of the thread passed to it as the first parameter.
- Second parameter is a pointer to the return value of the thread that joins the caller.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

# 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 using pthread\_exit()**
- **Main process terminates**

# Thread Cancellation

Terminating a thread before it has finished

Having made the cancellation request, *pthread\_cancel()* returns immediately; that is it does not wait for the target thread to terminate.

So, what happens to the target thread?

What happens and when it happens depends on the thread's cancellation state and type.

# Thread Cancellation

Thread attributes that indicate cancellation state and type

- Two general states
  - Thread cannot be cancelled.
    - PTHREAD\_CANCEL\_DISABLE
  - Thread can be cancelled
    - PTHREAD\_CANCEL\_ENABLE
    - Default

# Thread Cancellation

- When thread is cancelable, there are two general types
  - **Asynchronous cancellation** terminates the target thread immediately
    - PTHREAD\_CANCEL\_ASYNCHRONOUS
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - PTHREAD\_CANCEL\_DEFERRED
    - Cancel when thread reaches 'cancellation point'

# Thread Termination: Cleanup Handlers

Functions automatically executed when the thread is canceled.

- Clean up global variables
- Unlock code or data held by thread
- Close files
- Commit or rollback transactions

# Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
- Thread pools
- Thread safety
- Thread-specific data

# Semantics of *fork()*, *exec()*, *exit()*

Does **fork()** duplicate only the calling thread or all threads?

## □ Threads and ***exec()***

With *exec()*, the calling program is replaced in memory. All threads, except the one calling *exec()*, vanish immediately. No thread-specific data destructors or cleanup handlers are executed.

## □ Threads and ***exit()***

If any thread calls *exit()* or the main thread does a return, ALL threads immediately vanish. No thread-specific data destructors or cleanup handlers are executed.

# Semantics of *fork()*, *exec()*, *exit()*

## □ Threads and *fork()*

When a multithread process calls *fork()*, only the calling thread is replicated. All other threads vanish in the child. No thread-specific data destructors or cleanup handlers are executed.

Problems:

- The global data may be inconsistent:
  - Was another thread in the process of updating it?
  - Data or critical code may be locked by another thread. That lock is copied into child process, too.
- Memory leaks
- Thread (other) specific data not available

**Recommendation: In multithreaded application, use *fork()* only after *exec()***

# 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 Safety

A function is *thread-safe* if it can be safely invoked by multiple threads at the same time.

Example of a non safe function:

```
static int glob = 0;

static void Incr (int loops)
{
    int loc, j;
    for (j = 0; j<loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
}
```

Employs global or static values that are shared by all threads

# POSIX compilation

On Linux, programs that use the Pthreads API must be compiled with ***-pthread***. The effects of this option include the following:

- `_REENTRANT` preprocessor macro is defined. This causes the declaration of a few reentrant functions to be exposed.
- The program is linked with the *libpthread* library  
(the equivalent of ***-lpthread***)
  - The precise options for compiling a multithreaded program vary across implementations (and compilers).

From M. Kerrisk, [The Linux Programming Interface](#)

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

<http://codebase.eu/tutorial/posix-threads-c/>

# Threads vs. Processes

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

# Example 1

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h>
void * thread1_fun(void * a) {
    printf("Thread 1 is running.....\n\n");
    printf("Thread 1's ID is %ld\n", pthread_self());
}
void * thread2_fun(void *address) {
    printf("Thread 2 is running.....\n\n");
    printf("Thread 2's ID is %ld\n", pthread_self());
}
int main() {
    pthread_t thread1, thread2;
    printf("Main Thread has started running.....\n\n");
    printf("Main Thread's ID is %ld\n", pthread_self());
```

```
if ( pthread_create( &thread1, NULL, thread1_fun, NULL
) < 0 )
{
    perror("pthread_create");
    exit( 1 );
}

if ( pthread_create( &thread2, NULL, thread2_fun, NULL
) < 0 )
{
    perror("pthread_create");
    exit( 1 );
}

pthread_join( thread1 , NULL);
pthread_join( thread2 , NULL);
printf("Main Thread is about to terminate\n");

return 0;
}
```

# Example 2

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h>
void * thread1_fun(void *num) {
    int *n = (int *) num;
    int i;
    long fact =1;
    for( i = 1 ; i <= *n; i++)
        fact*= i;
    printf("The factorial of %d is %ld\n", *n ,
fact);
}
void * thread2_fun(void *address) {
    char *str = (char *) address;
    int len = 0;
    for( len = 0 ; str[len] != '\0' ; len++ );
    printf("The string length of %s is %d\n",
str , len);
}
```

```
int main() {
    int n =5 ;
    char str[50];
    strcpy(str, "VIT Vellore");
    pthread_t thread1, thread2;
    if ( pthread_create( &thread1, NULL, thread1_fun,
(void *) &n ) <0 )
    {
        perror("pthread_create");
        exit( 1 );
    }

    if ( pthread_create( &thread2, NULL, thread2_fun,
(void *) str ) <0 )
    {
        perror("pthread_create");
        exit( 1 );
    }
    pthread_join( thread1 , NULL);
    pthread_join( thread2 , NULL);
    return 0;
}
```

# Example 3

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h> //Same as the previous program but thread1_fun
returns a value
long * fact;
void * thread1_fun(void *num) {
    int *n = (int *) num;
    int i;
    *fact=1;
    for( i = 1 ; i <= *n; i++)
        *fact*= i;
    printf("The factorial of %d is %ld\n", *n , *fact);
    return fact;
}
void * thread2_fun(void *address) {
    char *str = (char *) address;
    int len = 0;
    for( len = 0 ; str[len] != '\0' ; len++ );
    printf("The string length of %s is %d\n", str , len);
}
```

```
int main() {
    int n =5 ;
    char str[50];
    strcpy(str, "VIT Vellore");
    fact = (long *) malloc(sizeof(long));
    pthread_t thread1, thread2;
    if ( pthread_create( &thread1, NULL, thread1_fun, (void *) &n ) <0 )
    {
        perror("pthread_create");
        exit( 1 );
    }
    if ( pthread_create( &thread2, NULL, thread2_fun, (void *) str ) <0 )
    {
        perror("pthread_create");
        exit( 1 );
    }
    pthread_join( thread1 , (void **)&fact);
    pthread_join( thread2 , NULL);

    printf("The factorial of %d is %ld\n", n , *fact);

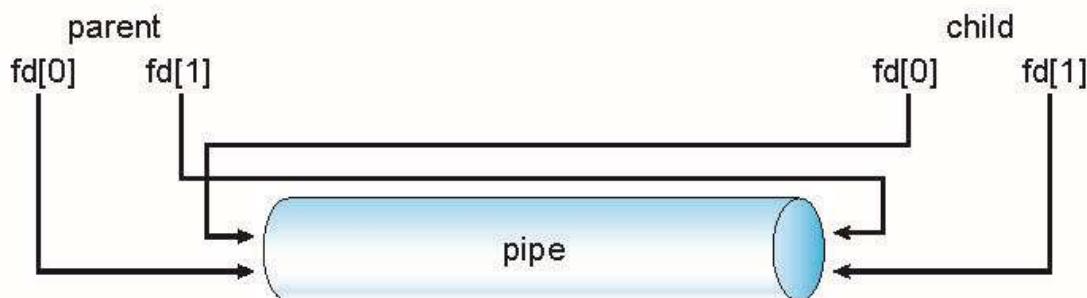
    return 0;
}
```

# Pipe

- 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** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.

## 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
- Windows calls these **anonymous pipes**
- Unix and Windows code samples are available in textbook



## 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

Examples

```
// Parent sends a message to child using a pipe

#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>

int main( ){

    char msg1[100], msg2[100];

    pid_t pid;
    int pipeDesc[2];
    strcpy(msg1, "Hello Child");

    if( pipe( pipeDesc ) < 0 ) {
        perror("pipe");
        return 1;
    }

    pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }

    if ( pid > 0 ) {
        close( pipeDesc [0] );
        printf("Parent writes: %s\n", msg1);
        write ( pipeDesc[1], msg1, strlen(msg1)+1);
    }
    else {
        close( pipeDesc [1] );
    }
}
```

```

        read ( pipeDesc[0], msg2, sizeof( msg2 ) );
        printf("\nChild read: %s\n", msg2);
    }

    return 0;
}

//.....
// Parent sends a message to child using a pipe

#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>

int main( ){

    char msg1[100], msg2[100];
    pid_t pid;
    int pipeDesc[2];

    if( pipe( pipeDesc ) < 0 ) {
        perror("pipe");
        return 1;
    }

    pid = fork() ;
    if (pid < 0 ) {
        perror("fork");
        return 1;
    }

    if ( pid > 0 ) {
        close( pipeDesc [0] );
        printf("Input a line of text...\n");
        scanf("%[^\\n]", msg1);
        printf("Parent writes: %s\n", msg1);
    }
}

```

```

        write ( pipeDesc[1], msg1, strlen(msg1)+1);

    }

else {

    close( pipeDesc [1] );

    read ( pipeDesc[0], msg2, sizeof( msg2 ) );

    printf("\nChild read: %s\n", msg2);

}

return 0;

}

//.....  

// Implementation of the command "ls - s | sort " using pipe  

#include<stdio.h>  

#include<stdlib.h>  

#include<unistd.h>  

int main() {  

    int pid;  

    int fd[2];  

    pipe(fd);  

    pid=fork();  

    if( pid < 0 ) {  

        perror( "fork");  

        exit( -1 );  

    }  

    if( pid==0) {  

        close(1);
        dup (fd[1]);
    }
}
```

```
    close(fd[0]);
    close(fd[1]);
    // execl("/usr/bin/who","who",(char *) 0);
    execl("/bin/ls", "ls", "-s", (char *) 0);

}

else {
    close(0);
    dup (fd[0]);
    close(fd[0]);
    close(fd[1]);
    execl("/usr/bin/sort","sort",(char *) 0);
}

}
```

```

//Pthread Examples.....  

#include<stdio.h>  

#include<pthread.h>  

#include<stdlib.h>  

#include<string.h>  

void * thread1_fun(void * a) {  

    printf("Thread 1 is running.....\n\n");  

    printf("Thread 1's ID is %ld\n", pthread_self());  

}  

void * thread2_fun(void *address) {  

    printf("Thread 2 is running.....\n\n");  

    printf("Thread 2's ID is %ld\n", pthread_self());  

}  

int main() {  

    pthread_t thread1, thread2;  

    printf("Main Thread has started running.....\n\n");  

    printf("Main Thread's ID is %ld\n", pthread_self());  

    if ( pthread_create( &thread1, NULL, thread1_fun, NULL )<0 )  

    {  

        perror("pthread_create");  

        exit( 1 );  

    }  

    if ( pthread_create( &thread2, NULL, thread2_fun, NULL )< 0 )  

    {  

        perror("pthread_create");  

        exit( 1 );  

    }  

    pthread_join( thread1 , NULL);  

    pthread_join( thread2 , NULL);
}

```

```

printf("Main Thread is about to terminate\n");

return 0;
}

///////////////////////////////
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h>

void * thread1_fun(void *num) {
    int *n = (int *) num;
    int i;
    long fact=1;
    for( i = 1 ; i <= *n; i++)
        fact*= i;
    printf("The factorial of %d is %ld\n", *n , fact);
}

void * thread2_fun(void *address) {
    char *str = (char *) address;
    int len = 0;
    for( len = 0 ; str[len] != '\0' ; len++ );
    printf("The string length of %s is %d\n", str , len);
}

int main() {
    int n =5 ;
    char str[50];
    strcpy(str, "VIT Vellore");
    pthread_t thread1, thread2;
    if( pthread_create( &thread1, NULL, thread1_fun, (void *) &n ) <0 )
    {

```

```
perror("pthread_create");
exit( 1 );
}

if ( pthread_create( &thread2, NULL, thread2_fun, (void *) str ) <0 )
{
    perror("pthread_create");
    exit( 1 );
}

pthread_join( thread1 , NULL);
pthread_join( thread2 , NULL);

return 0;
}
```

# Module2\_InterProcessCommunication

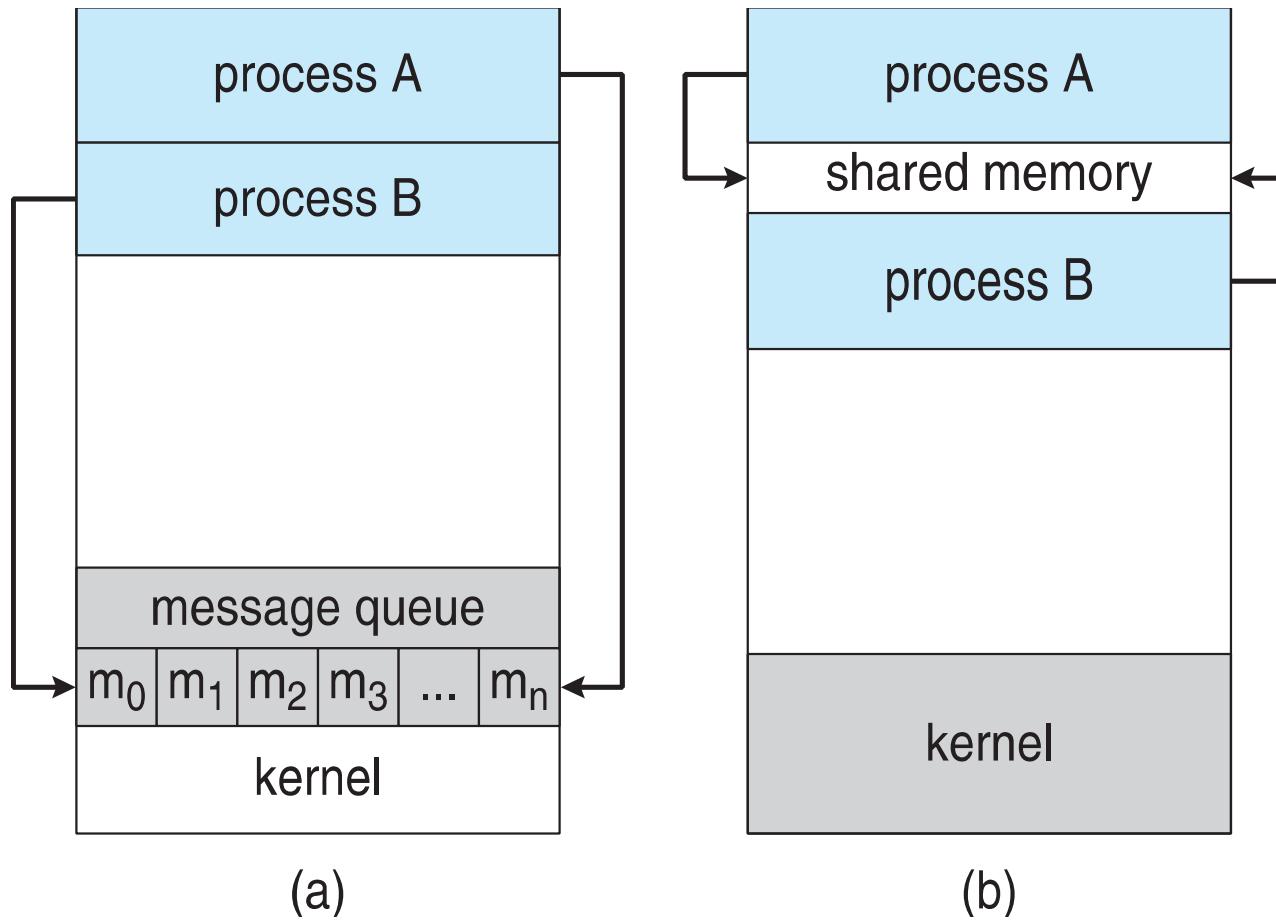
Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# 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

(a) Message passing. (b) shared memory.



# 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;
```

- Solution is correct, but can only use BUFFER\_SIZE-1 elements

# Bounded-Buffer – Producer

```
item next_produced;  
while (true) {  
    /* produce an item in next produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

# Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}
```

## Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in details in a later Module.

## Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send(message)**
  - **receive(message)**
- The *message size* is either **fixed** or **variable**

## Message Passing (Cont.)

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - 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?

## Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

# 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 is 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 (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message)** – send a message to mailbox A
  - receive(A, 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** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues
  - **Non-blocking receive** -- the receiver receives:
    - ❑ A valid message, or
    - ❑ Null message
- ❑ Different combinations possible
  - ❑ If both send and receive are blocking, we have a **rendezvous**

# Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link is full
  3. Unbounded capacity – infinite length  
Sender never waits

# Module3\_Deadlocks

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Deadlocks- Topics

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

# Objectives

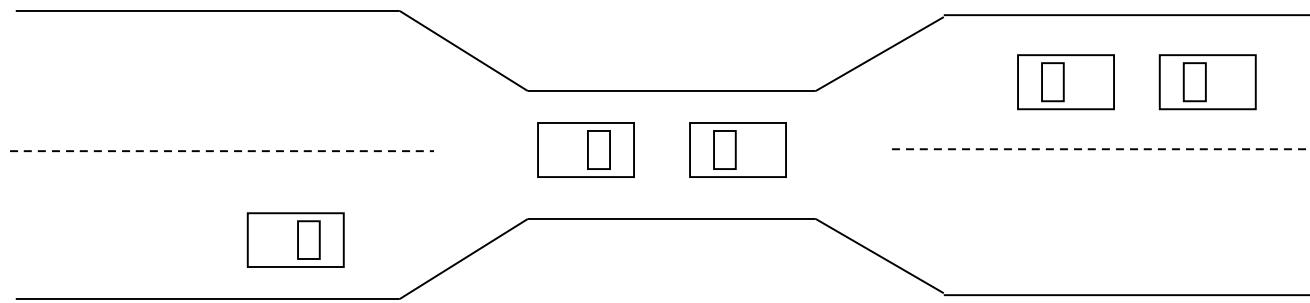
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - $P_1$  and  $P_2$  each hold one disk drive and each needs another one.
- Example
  - Semaphores  $A$  and  $B$ , initialized to 1

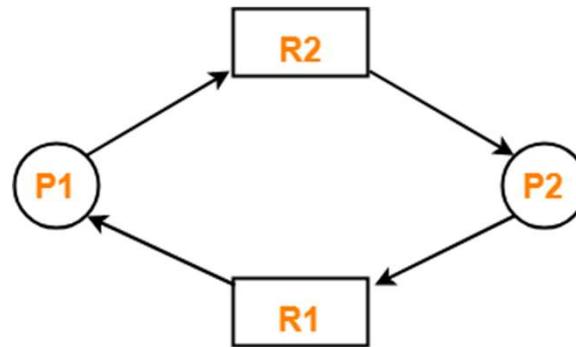
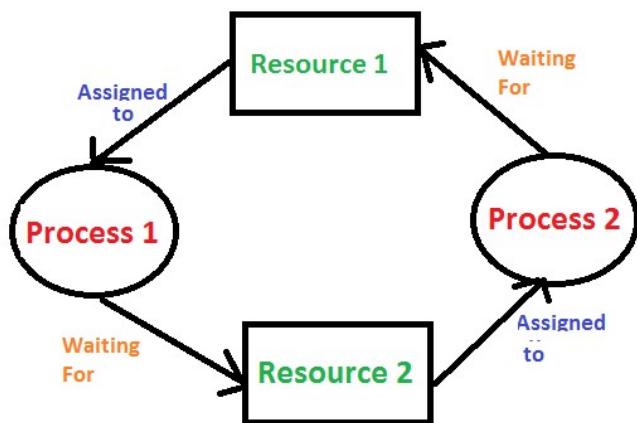
$P_0$	$P_1$
wait (A);	wait(B)
wait (B);	wait(A)

# Bridge Crossing Example

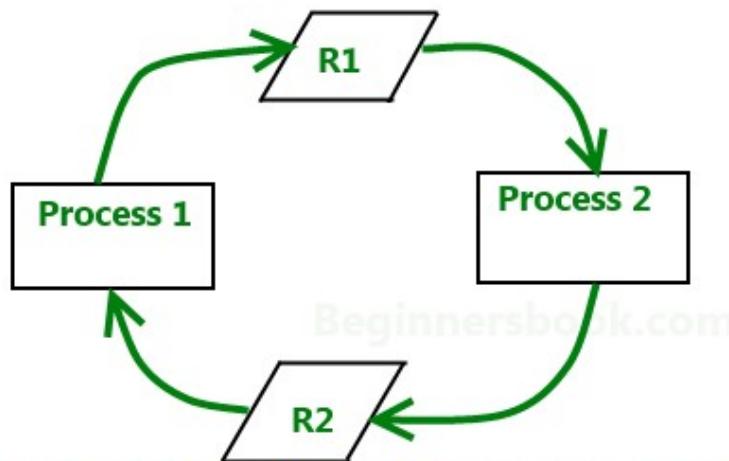


- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

# Examples



Example of a deadlock



Process P1 holds resource R2 and requires R1 while  
Process P2 holds resource R1 and requires R2.

# System Model

- System consists of **resources**
- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process **utilizes** a resource as follows:
  - **request**
  - **use**
  - **release**

# Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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

# Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.

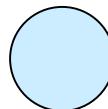
# Resource-Allocation Graph

A set of vertices  $V$  and a set of edges  $E$ .

- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

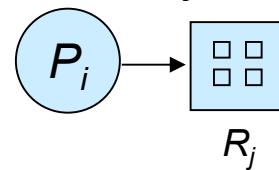
- Process



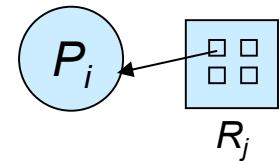
- Resource Type with 4 instances



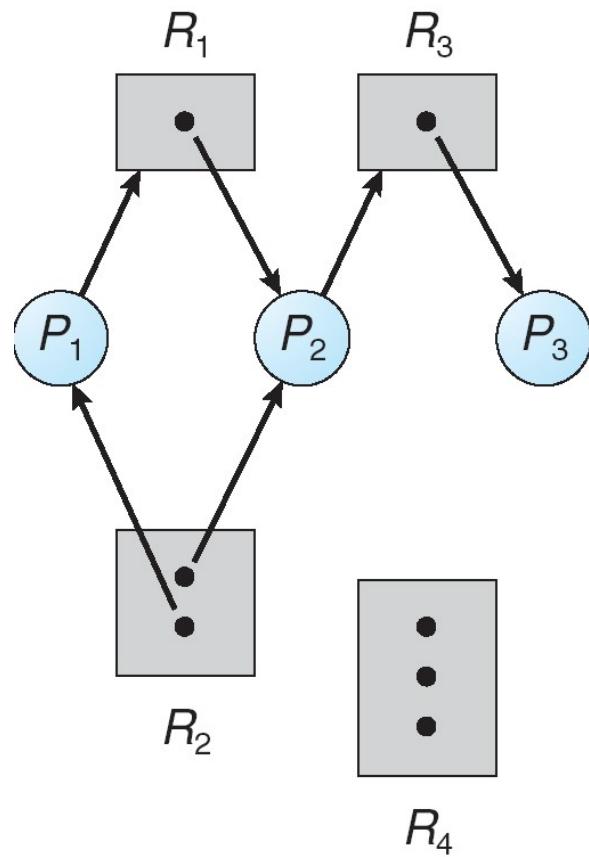
- $P_i$  requests instance of  $R_j$



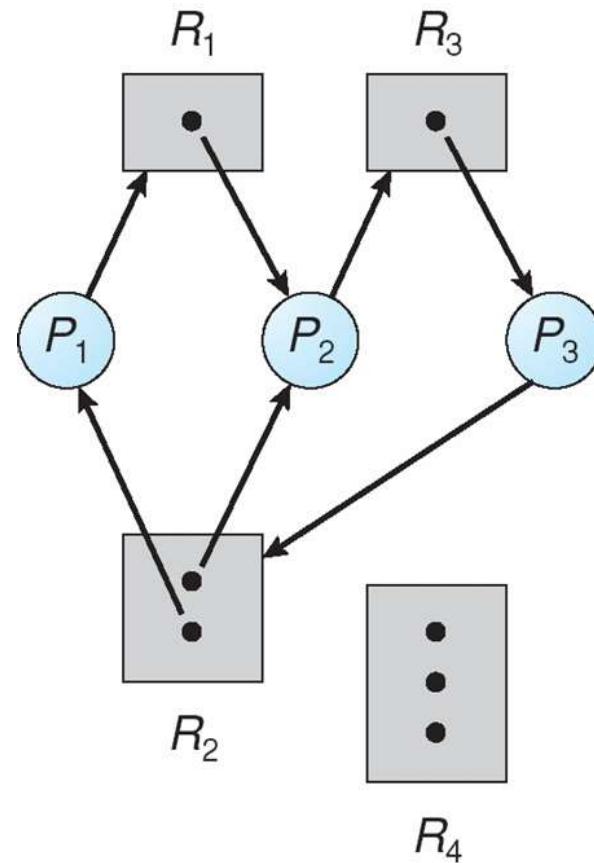
- $P_i$  is holding an instance of  $R_j$



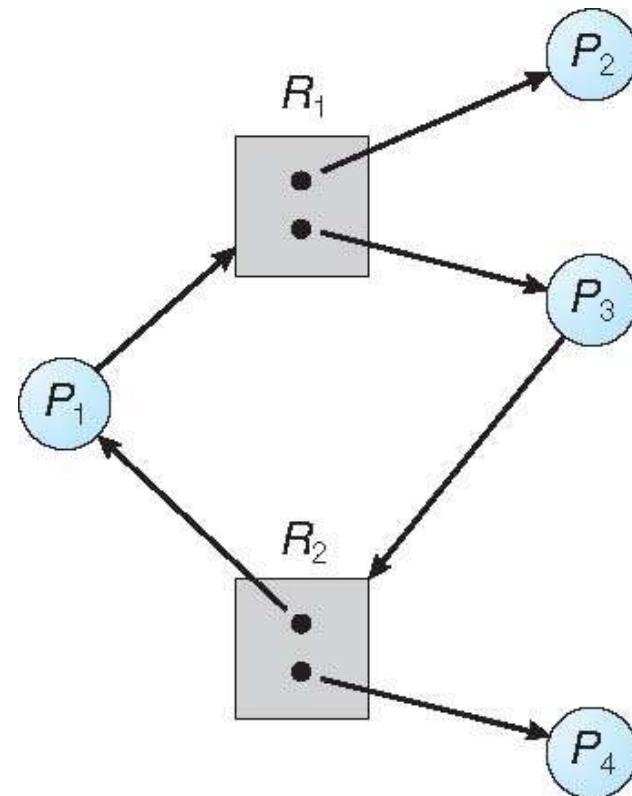
## Example of a Resource Allocation Graph



## Resource Allocation Graph With A Deadlock



# Graph With A Cycle But No Deadlock



# Basic Facts

- If graph contains no cycles  $\Rightarrow$  no deadlock
- If graph contains a cycle
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# Methods for Handling Deadlocks

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

# Deadlock Prevention

Restrain the ways request can be made

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

# Deadlock Prevention (Cont.)

- **No Preemption –**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then **all resources currently being held are released(preempted)**
    - Preempted resources are added to the list of resources for which the process is waiting
    - Process will be **restarted only when it can regain its old resources, as well as the new ones** that it is requesting  
(or)
  - If a process requests some resources, we first **check whether they are available.**
    - If they are, we allocate them.
    - If they are not, we **check whether they are allocated to some other process** that is waiting for additional resources.
      - If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
    - If the **resources are neither available nor held by a waiting process**, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

# Deadlock Prevention (Cont.)

- **Circular Wait** – One way to ensure that this condition never holds is to impose **a total ordering of all resource types** and to require that each process requests resources in an increasing order of enumeration.
  - Let  $R = \{R_1, R_2, \dots, R_m\}$  be the set of resource types.
  - Assign to each resource type a unique integer number, which allows to compare two resources and determine whether one precedes another in our ordering.
  - Formally, a one-to-one function  $F: R \rightarrow N$ , is defined where  $N$  is the set of natural numbers.
  - E.g., If the set of resource types  $R$  includes tape drives, disk drives, and printers, then the function  $F$  might be defined as follows:
    - $F(\text{tape drive}) = 1, F(\text{disk drive}) = 5, F(\text{printer}) = 12$
  - **Protocol to prevent deadlocks:** **Each process can request process resources only in an increasing order of enumeration.**
    - A process can initially request any number of instances of a resource type —say,  $R_i$ . After that, the process can request instances of resource type  $R_j$  if and only if  $F(R_j) > F(R_i)$ .
    - E.g: A process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
    - Alternatively, we can require that a process requesting an instance of resource type  $R_j$  must have released any resources  $R_i$  such that  $F(R_i) \geq F(R_j)$ .
  - If the two protocol mentioned above is used, then the circular-wait condition cannot hold.
    - Proof by contradiction: Assume that a circular wait exists. Let the set of processes involved in the circular wait be  $\{P_0, P_1, \dots, P_n\}$ , where  $P_i$  is waiting for a resource  $R_i$ , which is held by process  $P_{i+1}$ . ( $P_n$  is waiting for a resource  $R_n$  held by  $P_0$ .) Then, since process  $P_{i+1}$  is holding resource  $R_i$  while requesting resource  $R_{i+1}$ , we must have  $F(R_i) < F(R_{i+1})$  for all  $i$ . But this condition means that  $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$ . By transitivity,  $F(R_0) < F(R_0)$ , which is impossible. Therefore, there can be no circular wait in the system.

# Deadlock Example

```
/* Create and initialize the mutex locks */
Pthread_mutex_t first_mutex;
Pthread_mutex_t second_mutex;
pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);

/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

# Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A

# Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation **state** is defined by the number of available and allocated resources, and the maximum demands of the processes

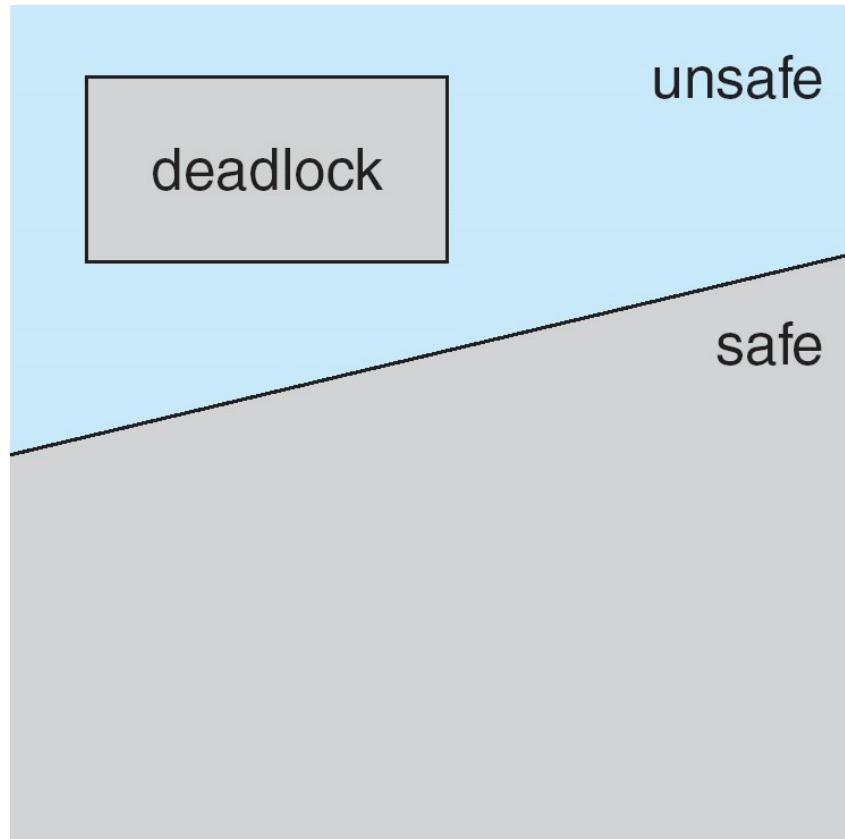
# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there **exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes** in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request could be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$ 
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

# Basic Facts

- 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.

# Safe, Unsafe, Deadlock State



# Safe, Unsafe, Deadlock State

- In an unsafe state, OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- The behavior of the processes controls unsafe states.
  - E.g , we consider a system with 12 magnetic tape drives and 3 processes: P0, P1, and P2. Consider the snapshot of the system at time t0 as given below in the table. Then there are three free tape drives.
  - At time t0, the system is in a safe state. The sequence **<P1, P0, P2>** satisfies the safety condition.
- A system can go from a safe state to an unsafe state.
  - Suppose that, at time t1, process P2 requests and is allocated one more tape drive.
  - The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock.
    - **Solution:** P2 should have been put under wait state, wait until either of the other processes had finished and released its resources

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

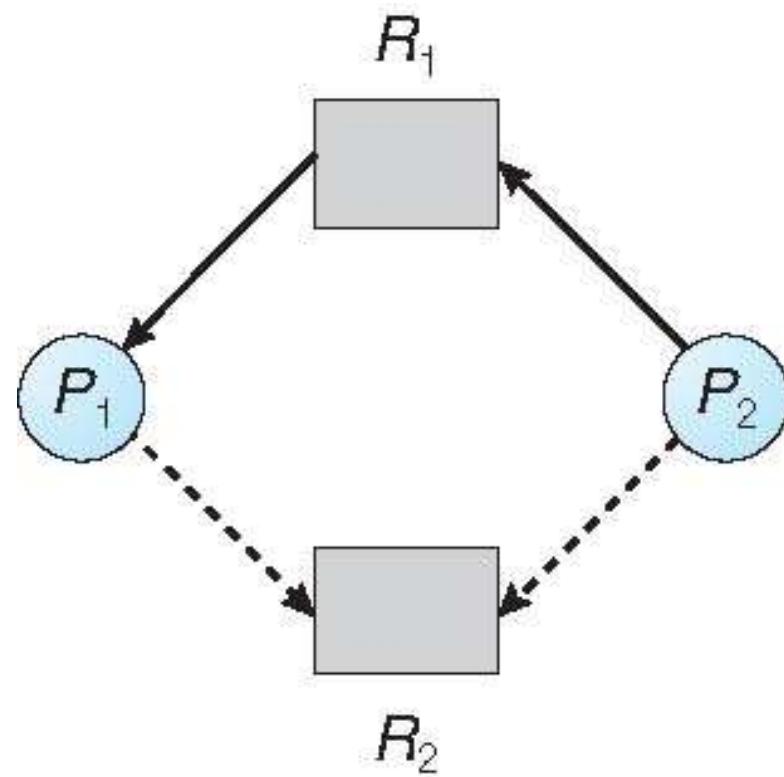
# Avoidance Algorithms

- Single instance of a resource type
  - Use a Resource-Allocation Graph (RAG)
- Multiple instances of a resource type
  - Use the Banker's algorithm

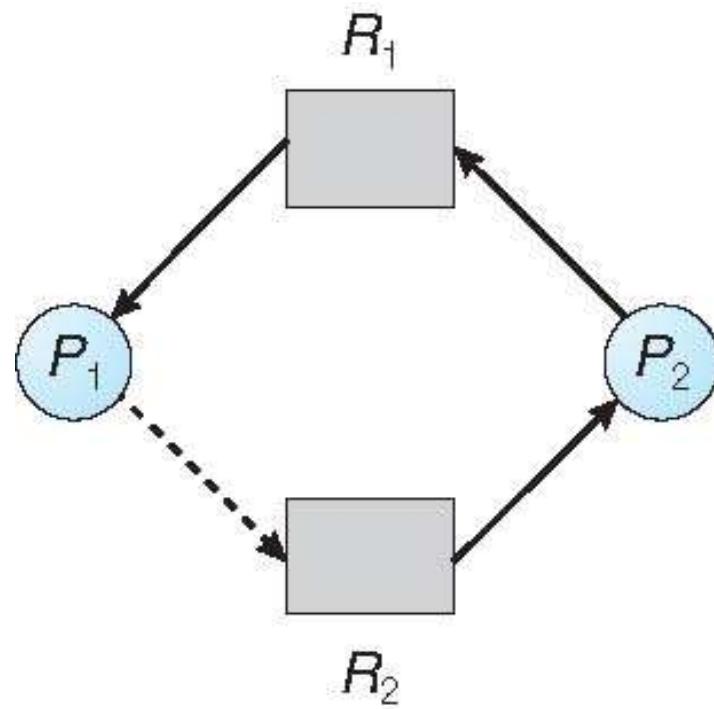
# Resource-Allocation Graph Scheme

- **Claim edge**  $P_i \rightarrow R_j$  indicated that process  $P_j$  may request resource  $R_j$ ; represented by a dashed line
- **Claim edge converts to request edge** when a process requests a resource
- **Request edge is converted to an assignment edge** when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

## Resource-Allocation Graph



## Unsafe State In Resource-Allocation Graph



## Resource-Allocation Graph Algorithm

- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

# 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

Time-varying data structures:

- **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

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

# Safety Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:

*Work = Available*

*Finish [i] = false for i = 0, 1, ..., n- 1*

2. Find an  $i$  such that both (a) & (b) are true:

(a) *Finish [i] = false*

(b)  $\text{Need}_i \leq \text{Work}$

If no such  $i$  exists, go to step 4

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$

*Finish[i] = true*

go to step 2

4. If *Finish [i] == true* for all  $i$ , then the system is in a safe state

## 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 Banker's Algorithm

- Consider a system with five processes P0 through P4 and three resource types A, B and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time t0 the following snapshot of the system has been taken: **Determine whether the system is in safe state.**
- 5 processes  $P_0$  through  $P_4$ ;  
3 resource types:  
 $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

# Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	<u>Need</u>		
	A	B	C
$P_0$	7	4	3
$P_1$	1	2	2
$P_2$	6	0	0
$P_3$	0	1	1
$P_4$	4	3	1

- The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety criteria
  - Also
    - The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria

## Example (Cont.)

- Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so **Request1 = (1,0,2)**  
– Can this request by  $P_1$  be granted?

# Example (cont'd): $P_1$ Request (1,0,2)

- Check if Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2)$ )  
 $\Rightarrow$  true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

- Executing safety algorithm shows that **sequence  $< P_1, P_3, P_4, P_0, P_2 >$  satisfies the safety requirement.**
  - Hence , the request by  $P_1$  can be granted immediately

## Example (cont'd)

- Can request for  $(3,3,0)$  by  $P_4$  be granted?
  - No, since the resources are not available (current availability is  $(2,3,0)$ ).
- Can request for  $(0,2,0)$  by  $P_0$  be granted?
  - No (though the resources are available, the resulting state is unsafe).

# Deadlock Detection

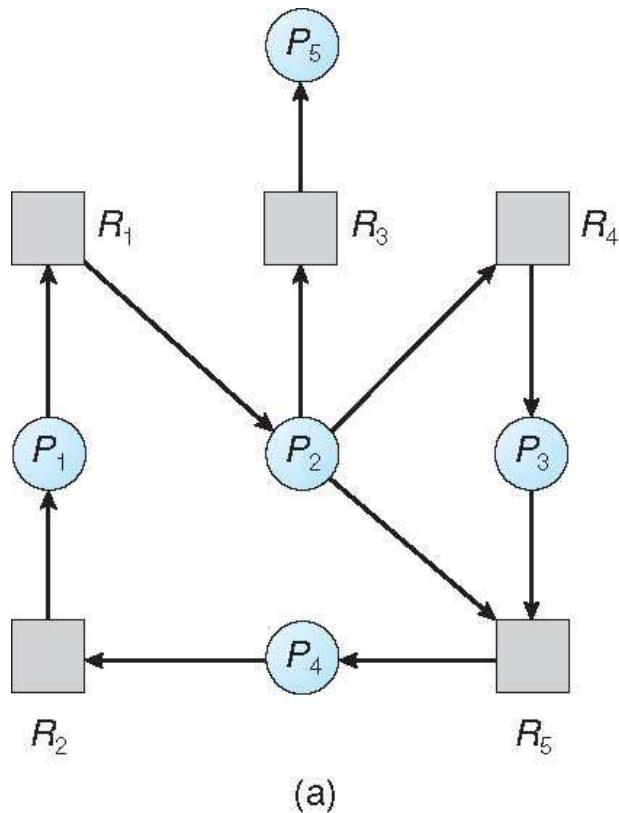
- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.
  - Allow the system to enter deadlock state
- In this environment, the system may provide
  - Detection algorithm
  - Recovery scheme

(At an additional overhead of run-time costs in maintaining the necessary information and executing the detection algorithm and also the potential losses inherent in recovering from a deadlock)

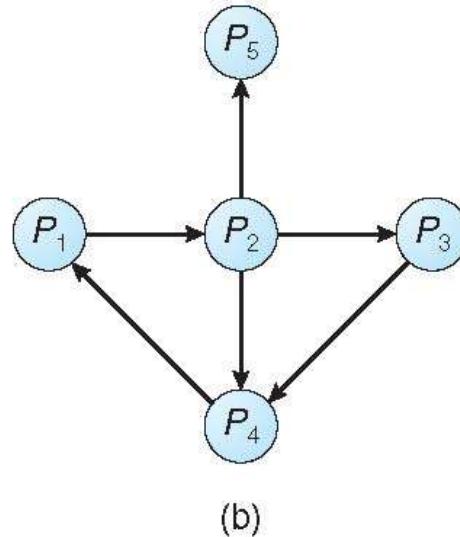
# Deadlock Detection - Single Instance of Each Resource Type

- Maintain **wait-for** graph (obtained from RAG, by removing resources nodes and collapsing edges appropriately)
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$ 
    - $P_i \rightarrow R_q$  and  $R_q \rightarrow P_j$
- Periodically invoke an algorithm that **searches for a cycle in the graph**. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph.

# Deadlock Detection – Resource Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

## Deadlock Detection -Several Instances of a Resource Type

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .

# Deadlock Detection Algorithm - Several Instances of a Resource Type

1. Let  $\text{Work}$  and  $\text{Finish}$  be vectors of length  $m$  and  $n$ , respectively  
Initialize:
  - (a)  $\text{Work} = \text{Available}$
  - (b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_i \neq 0$ , then  
 $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$
2. Find an index  $i$  such that both:
  - (a)  $\text{Finish}[i] == \text{false}$
  - (b)  $\text{Request}_i \leq \text{Work}$   
If no such  $i$  exists, go to step 4
3.  $\text{Work} = \text{Work} + \text{Allocation}_i$ ,  
 $\text{Finish}[i] = \text{true}$   
go to step 2
4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state

# Example of Detection Algorithm

- Consider a system with Five processes  $P_0$  through  $P_4$ ; three resource types  
A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time  $T_0$  (resource-allocation state):

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = true$  for all i.  
Hence, the system is not in a deadlocked state.  
Sequence  $\langle P_0, P_2, P_3, P_4, P_1 \rangle$  is another safe sequence.

## Example of Detection Algorithm(Cont'd)

- Suppose now that process  $P_2$  requests an additional instance of type **C**
  - **Request** matrix is modified as follows:

	<u>Request</u>		
	A	B	C
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

- State of system?
  - Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes requests
  - **Deadlock exists**, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$

# Detection-Algorithm Usage

- When, and how often, to invoke the algorithm depends on:
  - How often a deadlock is likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
    - How many processes will need to be rolled back?
      - one for each disjoint cycle

# Detection-Algorithm Usage (cont'd)

- If deadlocks occur frequently, then the detection algorithm should be invoked frequently
  - Resources allocated to deadlocked processes will be idle until the deadlock can be broken
  - Number of processes involved in the deadlock cycle may grow
- Deadlocks occur only when some process makes a request that cannot be granted immediately.
  - This request may be the final request that completes a chain of waiting processes.
- In the extreme case , we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
  - We can identify not only the deadlocked set of processes but also the specific process that “caused” the deadlock.
  - One request may create many cycles in the resource graph “caused” by the one identifiable process
    - Incurs considerable overhead in computation time.
- Alternatively, invoke the algorithm at defined intervals
  - Once per hour or Whenever CPU utilization drops below 40 percent.
- A deadlock eventually cripples system throughput and causes CPU utilization to drop.
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

# Recovery from Deadlock

- When a detection algorithm determines that **a deadlock exists**:
  - One possibility is to **inform the operator** that a deadlock has occurred and to let the operator deal with the deadlock **manually**.
  - Another possibility is to **let the system recover from the deadlock automatically**.
    - There are two options for breaking a deadlock.
      - **Abort one or more processes** to break the circular wait.
      - **Preempt some resources** from one or more of the deadlocked processes.

# Recovery from Deadlock: Process Termination

## 1. Abort all deadlocked processes

- Will break the deadlock cycle, but at great expense.
- Deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

## 2. Abort one process at a time until the deadlock cycle is eliminated

- Incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- In which order should we choose to abort? (determine which deadlocked process(es) to be terminated)
  - If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
  - If the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.
- Abort those processes whose termination will incur the minimum cost.
  - Many factors may affect which process is to be chosen, including:
    1. Priority of the process
    2. How long process has computed, and how much longer to completion
    3. Resources the process has used
    4. Resources process needs to complete
    5. How many processes will need to be terminated
    6. Is process interactive or batch?

The system reclaims all resources allocated to the terminated processes.

## Recovery from Deadlock: Resource Preemption

- To eliminate deadlocks using resource preemption
  - **successively preempt some resources from processes** and give these resources to other processes until the deadlock cycle is broken.
- If preemption is required to deal with deadlocks, then **three issues need to be addressed:**
  1. **Selecting a victim** – minimize cost
  2. **Rollback** – return to some safe state or restart process from that state
  3. **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

# Recovery from Deadlock: Resource Preemption

## 1. Selecting a victim ( Which resources and which processes are to be preempted?)

- determine the order of preemption leading to minimum cost.
- Cost factors may include parameters like the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

## 2. Rollback (If we preempt a resource from a process, what should be done with that process?)

- It cannot continue with its normal execution as it is missing some needed resource
- Roll back the process to some safe state and restart it from that state
- It is difficult to determine what a safe state is?
  - **Solution: Total rollback** - abort the process and then restart
- Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

## 3. Starvation (How do we ensure that starvation will not occur? How can we guarantee that resources will not always be preempted from the same process?)

- If victim selection is based on cost factors:
  - May be the same process is always picked as a victim
    - This process never completes its designated task
  - Solution: ensure that a process can be picked as a victim only a (small) finite number of times. **Include the number of rollbacks in the cost factor.**

# Banker's Algorithm example

Consider the System snapshot:

	Max	Allocation	Available
	A B C	A B C	A B C
P <sub>0</sub>	0 0 1	0 0 1	
P <sub>1</sub>	1 7 5	1 0 0	
P <sub>2</sub>	2 3 5	1 3 5	1 5 2
P <sub>3</sub>	0 6 5	0 6 3	

- How many resources are there of type (A,B,C)?
- What is the contents of the Need matrix?
- Is the system in a safe state? Why?

## Solution

- How many resources are there of type (A,B,C)? **(3,14,11)**
- What is the contents of the need matrix?

## Need Matrix:

	A	B	C
P <sub>0</sub>	0	0	0
P <sub>1</sub>	0	7	5
P <sub>2</sub>	1	0	0
P <sub>3</sub>	0	0	2

- Is the system in a safe state? Why?
- Yes, because the processes can be executed in the sequence P<sub>0</sub>, P<sub>2</sub>, P<sub>1</sub>, P<sub>3</sub>, even if each process asks for its maximum number of resources when it executes.

## Steps:

$$\text{Work} = \text{Available} = <1,5,2>$$

P<sub>0</sub> Need <= Work, hence Work= <1,5,2> + <0,0,1> = <1,5,3>, Safe Sequence = < P<sub>0</sub> >

P<sub>1</sub> Need is not <= Work

P<sub>2</sub> Need <= Work, hence Work= <1,5,3> + <1,3,5> = <2,8,8>, Safe Sequence = < P<sub>0</sub>, P<sub>2</sub> >

P<sub>1</sub> Need <= Work, hence Work= <2,8,8> + <1,0,0> = <3,8,8>, Safe Sequence = < P<sub>0</sub>, P<sub>2</sub>, P<sub>1</sub> >

P<sub>3</sub> Need <= Work, hence Work= <3,8,8> + <0,6,3> = <3,14,11>,

Safe Sequence = < P<sub>0</sub>, P<sub>2</sub>, P<sub>1</sub>, P<sub>3</sub> >

If a request from process  $P_1$  arrives for additional resources of  $(0,5,2)$ , can the Banker's algorithm grant the request immediately? Show the system state,

**Yes. Since**

the sequence  $\langle P_0, P_2, P_1, P_3 \rangle$  satisfies the safety constraint.

## Banker's Algorithm Example Solutions

### Exercise 1

Assume that there are 5 processes,  $P_0$  through  $P_4$ , and 4 types of resources. At  $T_0$  we have the following system state:

Max Instances of Resource Type A = 3 (2 allocated + 1 Available)

Max Instances of Resource Type B = 17 (12 allocated + 5 Available)

Max Instances of Resource Type C = 16 (14 allocated + 2 Available)

Max Instances of Resource Type D = 12 (12 allocated + 0 Available)

<u>Given Matrices</u>												
	<u>Allocation Matrix</u> (N0 of the allocated resources By a process)				<u>Max Matrix</u> Max resources that may be used by a process				<u>Available Matrix</u> Not Allocated Resources			
	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>
<b>P<sub>0</sub></b>	0	1	1	0	0	2	1	0	1	5	2	0
<b>P<sub>1</sub></b>	1	2	3	1	1	6	5	2				
<b>P<sub>2</sub></b>	1	3	6	5	2	3	6	6				
<b>P<sub>3</sub></b>	0	6	3	2	0	6	5	2				
<b>P<sub>4</sub></b>	0	0	1	4	0	6	5	6				
<b>Total</b>	<b>2</b>	<b>12</b>	<b>14</b>	<b>12</b>								

#### 1. Create the need matrix (max-allocation)

$$\text{Need}(i) = \text{Max}(i) - \text{Allocated}(i)$$

$$(i=0) \quad (0,2,1,0) - (0,1,1,0) = (0,1,0,0)$$

$$(i=1) \quad (1,6,5,2) - (1,2,3,1) = (0,4,2,1)$$

$$(i=2) \quad (2,3,6,6) - (1,3,6,5) = (1,0,0,1)$$

$$(i=3) \quad (0,6,5,2) - (0,6,3,2) = (0,0,2,0)$$

$$(i=4) \quad (0,6,5,6) - (0,0,1,4) = (0,6,4,2)$$

Ex. Process  $P_1$  has max of (1,6,5,2) and allocated by (1,2,3,1)

$$\text{Need}(p_1) = \text{max}(p_1) - \text{allocated}(p_1) = (1,6,5,2) - (1,2,3,1) = (0,4,2,1)$$

$$\text{Need Matrix} = \text{Max Matrix} - \text{Allocation Matrix}$$

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>
<b>P<sub>0</sub></b>	0	1	0	0
<b>P<sub>1</sub></b>	0	4	2	1
<b>P<sub>2</sub></b>	1	0	0	1
<b>P<sub>3</sub></b>	0	0	2	0
<b>P<sub>4</sub></b>	0	6	4	2

#### 2. Use the safety algorithm to test if the system is in a safe state or not?

##### a. We will first define work and finish:

Initially work = available = ( 1, 5 , 2, 0)  
**Finish** = False for all processes

Finish matrix	
<b>P<sub>0</sub></b>	False
<b>P<sub>1</sub></b>	False
<b>P<sub>2</sub></b>	False
<b>P<sub>3</sub></b>	False
<b>P<sub>4</sub></b>	False

Work vector			
<b>1</b>	<b>5</b>	<b>2</b>	<b>0</b>

##### b. Check the needs of each process [ $\text{needs}(p_i) \leq \text{Max}(p_i)$ ], if this condition is true:

- Execute the process , Change  $\text{Finish}[i] = \text{True}$
- Release the allocated Resources by this process
- Change The Work Variable = Allocated (pi) + Work

**need<sub>0</sub> (0,1,0,0) <= work(1,5,2,0)**

P0 will be  
executed because  
need(P0) <= Work  
P0 will be True

Finish matrix	
<b>P<sub>0</sub> - 1</b>	<b>True</b>
P <sub>1</sub>	False
P <sub>2</sub>	False
P <sub>3</sub>	False
P <sub>4</sub>	False

P0 will release the allocated resources(0,1,1,0)  
Work = Work  
(1,5,2,0)+Allocated(P0)  
(0,1,1,0) = 1,6,3,0

Work vector			
1	6	3	0

**Need<sub>1</sub> (0,4,2,1) <= work(1,6,3,0) Condition Is False P1 will Not be executed**

**Need<sub>2</sub> (1,0,0,1) <= work(1,6,3,0) Condition Is False P2 will Not be executed**

**Need<sub>3</sub> (0,0,2,0) <= work(1,6,3,0) P3 will be executed**

P3 will be  
executed because  
need(P3) <= Work  
P3 will be True

Finish matrix	
<b>P<sub>0</sub> - 1</b>	<b>True</b>
<b>P<sub>1</sub></b>	False
P <sub>2</sub>	False
<b>P<sub>3</sub> - 2</b>	<b>True</b>
P <sub>4</sub>	False

P3 will release the allocated resources (0,6,3,2)  
Work = Work  
(1,6,3,0)+Allocated(P3)  
(0,6,3,2) = 1,12,6,2

Work vector			
1	12	6	2

**Need<sub>4</sub> (0,6,4,2) <= work(1,12,6,2) P4 will be executed**

P4 will be  
executed because  
need(P4) <= Work  
P4 will be True

Finish matrix	
<b>P<sub>0</sub> - 1</b>	<b>True</b>
<b>P<sub>1</sub></b>	False
P <sub>2</sub>	False
<b>P<sub>3</sub> - 2</b>	<b>True</b>
<b>P<sub>4</sub> - 3</b>	<b>True</b>

P4 will release the allocated resources (0,0,1,4)  
Work = Work  
(1,12,6,2) + Allocated(P4)  
(0,0,1,4) = 1,12,7,6

Work vector			
1	12	7	6

**Need<sub>1</sub> (0,4,2,1) <= work(1,12,7,6) P1 will be executed**

P1 will be  
executed because  
need(P1) <= Work  
P1 will be True

Finish matrix	
<b>P<sub>0</sub> - 1</b>	<b>True</b>
<b>P<sub>1</sub> - 4</b>	<b>True</b>
P <sub>2</sub>	False
<b>P<sub>3</sub> - 2</b>	<b>True</b>
<b>P<sub>4</sub> - 3</b>	<b>True</b>

P1 will release the allocated resources (1,2,3,1)  
Work = Work  
(1,12,7,6) + Allocated(P1)  
(1,2,3,1) = 2,14,10,7

Work vector			
2	14	10	7

**Need<sub>2</sub> (1,0,0,1) <= work(2,14,10,7) P2 will be executed**

P2 will be  
executed because  
need(P2) <= Work  
P2 will be True

Finish matrix	
<b>P<sub>0</sub> - 1</b>	<b>True</b>
<b>P<sub>1</sub> - 4</b>	<b>True</b>
<b>P<sub>2</sub> - 5</b>	<b>True</b>
<b>P<sub>3</sub> - 2</b>	<b>True</b>
<b>P<sub>4</sub> - 3</b>	<b>True</b>

P2 will release the allocated resources (1,3,6,5)  
Work = Work  
(2,14,10,7) + Allocated(P1)  
(1,3,6,5) = 3,17,16,12

Work vector			
3	17	16	12

The system is in a safe state and the processes will be executed in the following order:

P0,P3,P4,P1,P2

### Exercise 2:

If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary.

- a. P1 requests (2,1,1,0)

We cannot grant this request, because we do not have enough available instances of resource A.

- b. P1 requests (0,2,1,0)

There are enough available instances of the requested resources, so first let's pretend to accommodate the request and see the system looks like:

	Allocation				Max				Available				Need Matrix
	A	B	C	D	A	B	C	D	A	B	C	D	
P <sub>0</sub>	0	1	1	0	0	2	1	0	1	3	1	0	
P <sub>1</sub>	1	4	4	1	1	6	5	2					
P <sub>2</sub>	1	3	6	5	2	3	6	6					
P <sub>3</sub>	0	6	3	2	0	6	5	2					
P <sub>4</sub>	0	0	1	4	0	6	5	6					

Now we need to run the safety algorithm:

Initially

Work vector	Finish matrix
1	P <sub>0</sub> False
3	P <sub>1</sub> False
1	P <sub>2</sub> False
0	P <sub>3</sub> False
	P <sub>4</sub> False

Let's first look at P<sub>0</sub>. Need<sub>0</sub> (0,1,0,0) is less than work, so we change the work vector and finish matrix as follows:

Work vector	Finish matrix
1	P <sub>0</sub> True
4	P <sub>1</sub> False
2	P <sub>2</sub> False
0	P <sub>3</sub> False
	P <sub>4</sub> False

Need<sub>1</sub> (0,2,1,1) is not less than work, so we need to move on to P<sub>2</sub>.

Need<sub>2</sub> (1,0,0,1) is not less than work, so we need to move on to P<sub>3</sub>.

Need<sub>3</sub> (0,0,2,0) is less than or equal to work.

Let's update work and finish:

Work vector	Finish matrix
1	P <sub>0</sub> True
10	P <sub>1</sub> False
5	P <sub>2</sub> False
2	P <sub>3</sub> True
	P <sub>4</sub> False

Let's take a look at Need<sub>4</sub> (0,6,4,2). This is less than work, so we can update work and finish:

Work vector	Finish matrix
1	P <sub>0</sub> True
10	P <sub>1</sub> False
6	P <sub>2</sub> False
6	P <sub>3</sub> True
	P <sub>4</sub> True

We can now go back to P<sub>1</sub>. Need<sub>1</sub> (0,2,1,1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P <sub>0</sub>	True
14	P <sub>1</sub>	True
10	P <sub>2</sub>	False
7	P <sub>3</sub>	True
	P <sub>4</sub>	True

Finally, Need<sub>2</sub> (1,0,0,1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order:

P<sub>0</sub>,P<sub>3</sub>,P<sub>4</sub>,P<sub>1</sub>,P<sub>2</sub>. We therefore can grant the resource request.

### Exercise 3

Assume that there are three resources, A, B, and C. There are 4 processes P<sub>0</sub> to P<sub>3</sub>. At T<sub>0</sub> we have the following snapshot of the system:

	Allocation			Max			Available			1.Create the need matrix.
	A	B	C	A	B	C	A	B	C	
P <sub>0</sub>	1	0	1	2	1	1	2	1	1	
P <sub>1</sub>	2	1	2	5	4	4				
P <sub>2</sub>	3	0	0	3	1	1				
P <sub>3</sub>	1	0	1	1	1	1				

2. Is the system in a safe state? Why or why not?

In order to check this, we should run the safety algorithm. Let's create the work vector and finish matrix:

	Work vector		Finish matrix			Need <sub>0</sub> (1,1,0) is less than work, so let's go ahead and update work and finish:
	2	P <sub>0</sub>	False	P <sub>1</sub>		
	1	P <sub>1</sub>	False			
	1	P <sub>2</sub>	False			
		P <sub>3</sub>	False			

Need<sub>1</sub> (3,3,2) is not less than work, so we have to move on to P<sub>2</sub>.

Need<sub>2</sub> (0,1,1) is less than work, let's update work and finish:

	Work vector		Finish matrix	
	6	P <sub>0</sub>	True	P <sub>1</sub>
	1	P <sub>1</sub>	False	
	2	P <sub>2</sub>	True	
		P <sub>3</sub>	False	

Need<sub>3</sub> (0,1,0) is less than work, we can update work and finish:

	Work vector		Finish matrix	
	7	P <sub>0</sub>	True	P <sub>1</sub>
	1	P <sub>1</sub>	False	
	3	P <sub>2</sub>	True	
		P <sub>3</sub>	True	

We now need to go back to P<sub>1</sub>. Need<sub>1</sub> (3,3,2) is not less than work, so we cannot continue. Thus, the system is not in a safe state.

# Module4\_Process Synchronization

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

# Objectives

- To present the concept of process synchronization.
- To introduce the **critical-section problem**, whose solutions can be used to **ensure the consistency of shared data**
- To present both **software and hardware solutions** of the critical-section problem
- To examine several **classical process-synchronization problems**
- To explore several **tools** that are used to solve process synchronization problems

# Background

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

Suppose that we wanted to provide a solution to the **Producer-Consumer** problem that **fills all the buffers**. We can do so by having **an integer counter that keeps track of the number of full buffers**. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

# Producer

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}  
//Old version of Producer  
item next_produced;  
while (true) {  
    /* produce an item in next produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

# Consumer

```
while (true) {  
    while (counter == 0);  
    /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next_consumed */  
}
```

# Race Condition - Example

- **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, is called a **race condition**.
  - **Solution:** Ensure that only one process at a time can be manipulating the variable **counter** which requires that the processes be synchronized in some way.

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “counter = 5” initially:

T0: producer executes <b>register1 = counter</b>	{register1 = 5}
T1: producer executes <b>register1 = register1 + 1</b>	{register1 = 6}
T2: consumer executes <b>register2 = counter</b>	{register2 = 5}
T3: consumer executes <b>register2 = register2 - 1</b>	{register2 = 4}
T4: producer executes <b>counter = register1</b>	{counter = 6 }
T5: consumer executes <b>counter = register2</b>	{counter = 4}

# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be
    - changing common variables
    - Updating a shared memory block
    - updating table
    - writing file, ... etc
  - When one process is in critical section, no other may be in its critical section
    - no two processes are executing in their critical sections at the same time.
- **Critical section problem** is to design protocol that the processes can use to cooperate
- Each process must ask permission to enter **critical section** in **entry section**, may follow critical section with **exit section**, then **remainder section**

# Critical Section

- General structure of process  $P_i$

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

# Solution to Critical-Section Problem

A solution to the critical-section problem must **satisfy the following three requirements:**

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.
  - Only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next
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

# Algorithm for Process P<sub>i</sub>

```
do {  
    while (turn == j);  
        critical section  
    turn = j;  
    remainder section  
} while (true);
```

Process 0, P0

do {

while (turn ==1);

critical section

turn =1;

remainder  
section

} while (true);

Process 1, P1

do {

while (turn == 0);

critical section

turn = 0;

remainder  
section

} while (true);

# Solution to Critical-Section Problem

A solution to the critical-section problem must **satisfy the following three requirements:**

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.
  - Only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next
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

# Dekker's Algorithm

- Dekker's algorithm is the first known correct solution to the mutual exclusion problem. The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra.

- Variables

```
wants_to_enter : array of 2 booleans  
turn : integer  
  
wants_to_enter[0] ← false  
wants_to_enter[1] ← false  
turn ← 0 // or 1
```

```
p0:  
    wants_to_enter[0] ← true  
    while wants_to_enter[1] {  
        if turn ≠ 0 {  
            wants_to_enter[0] ← false  
            while turn ≠ 0 {  
                // busy wait  
            }  
            wants_to_enter[0] ← true  
        }  
    }  
  
    // critical section  
    ...  
    turn ← 1  
    wants_to_enter[0] ← false  
    // remainder section
```

```
p1:  
    wants_to_enter[1] ← true  
    while wants_to_enter[0] {  
        if turn ≠ 1 {  
            wants_to_enter[1] ← false  
            while turn ≠ 1 {  
                // busy wait  
            }  
            wants_to_enter[1] ← true  
        }  
    }  
  
    // critical section  
    ...  
    turn ← 0  
    wants_to_enter[1] ← false  
    // remainder section
```

# Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- 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);
```

Process i,  $P_j$

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

Process j,  $P_j$

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

# Peterson's Solution (Cont.)

- To enter the critical section, process  $P_i$  first sets  $\text{flag}[i]$  to be true and then sets  $\text{turn}$  to the value  $j$ , thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time,  $\text{turn}$  will be set to both  $i$  and  $j$  at roughly the same time.
  - Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of  $\text{turn}$  determines which of the two processes is allowed to enter its critical section first.

# Peterson's Solution (Cont.)

- Provable that the three CS requirements 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:
    - Process  $P_i$  can be prevented from entering the critical section only if it is stuck in the while loop with the condition  $\text{flag}[j] == \text{true}$  and  $\text{turn} == j$ ; this loop is the only one possible.
    - If  $P_j$  is not ready to enter the critical section, then  $\text{flag}[j] == \text{false}$ , and  $P_i$  can enter its critical section.
    - If  $P_j$  has set  $\text{flag}[j]$  to true and is also executing in its while statement, then either  $\text{turn} == i$  or  $\text{turn} == j$ . If  $\text{turn} == i$ , then  $P_i$  will enter the critical section. If  $\text{turn} == j$ , then  $P_j$  will enter the critical section. However, once  $P_j$  exits its critical section, it will reset  $\text{flag}[j]$  to false, allowing  $P_i$  to enter its critical section.
  3. Bounded-waiting requirement is met
    - If  $P_j$  resets  $\text{flag}[j]$  to true, it must also set  $\text{turn}$  to  $i$ . Thus, **since  $P_i$  does not change the value of the variable  $\text{turn}$  while executing the while statement,  $P_i$  will enter the critical section (progress)** after at most one entry by  $P_j$  (bounded waiting).

# Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
  - Protecting critical regions via locks
- **Hardware features can make any programming task easier and improve system efficiency.**
- **Uniprocessors** – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on **multiprocessor systems**
    - Operating systems using this not broadly scalable
- Modern machines **provide special atomic hardware instructions**
  - **Atomic** = non-interruptible
  - Either **test** memory word and **set** value
  - Or **swap contents** of two memory words

## Solution to Critical-section Problem Using Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);  
  
//General flow as given above
```

# test\_and\_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

# Solution using test\_and\_set()

- Shared Boolean variable **lock**, initialized to FALSE
- Solution - Mutual-exclusion implementation with test and set():

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

# 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;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new\_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

# Solution using compare\_and\_swap

- Shared integer “lock” is initialized to 0.
- **Solution-Mutual-exclusion implementation with the compare and swap() instruction:**

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

## Bounded-waiting Mutual Exclusion with test\_and\_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

# Mutex Locks

- ❑ Previous solutions are complicated and generally inaccessible to application programmers
- ❑ OS designers build software tools to solve critical section problem
- ❑ Simplest is `mutex lock`
- ❑ Protect a critical section by first `acquire()` a lock then `release()` the lock
  - ❑ Boolean variable indicating if lock is available or not
- ❑ Calls to `acquire()` and `release()` must be atomic
  - ❑ Usually implemented via hardware atomic instructions
- ❑ But this solution requires **busy waiting**
  - ❑ This 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);`

# Semaphore

- **Synchronization tool** that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
  - Can only be accessed via two indivisible (atomic) operations
    - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait() operation**

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

- Definition of the **signal() operation**

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

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider two concurrently running processes  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

**P1 :**

```
s1;  
signal(synch);
```

**P2 :**

```
wait(synch);  
s2;
```

- Can implement a counting semaphore  $S$  as a binary semaphore

# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the 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
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - But many 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
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- 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
- ```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

## 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);  
    }  
}
```

# Semaphore Example

```
// WRT the modified wait and signal implementation:  
Semaphore *s1, *s2;  
s1 = new Semaphore(1); // s1->value = 1  
s2 = new Semaphore(2); // s2->value = 2  
// Assume a system of 3 cooperating processes and the pseudocode  
snippets given below:
```

| Process P1     | Process P2     | Process P3     |
|----------------|----------------|----------------|
| wait ( s2 );   |                |                |
| wait ( s1 );   |                |                |
| ....           |                |                |
| signal ( s1 ); | wait( s2 );    |                |
|                | wait ( s1 );   |                |
|                | ....           |                |
|                | signal ( s1 ); | wait ( s2 );   |
|                |                | wait ( s1 );   |
|                |                | ....           |
|                |                | signal ( s1 ); |

```
// After the execution of the snippets according to the given  
timeline , the state of s1 and s2 would be as follows  
// P3 waits on s2 ...  
s1->value = 1 , s1->list = NULL  
s2->value = -1 , s2->list -> P3
```

## Semaphore Example (cont'd)

```
// Adding to previous pseudocode snippets
```

| Process P1             | Process P2      | Process P3      |
|------------------------|-----------------|-----------------|
| wait ( s2 ) ;          |                 |                 |
| wait ( s1 ) ;          |                 |                 |
| ....                   |                 |                 |
| signal ( s1 ) ;        | wait( s2) ;     |                 |
|                        | wait ( s1 ) ;   |                 |
|                        | ....            |                 |
|                        | signal ( s1 ) ; | wait ( s2 ) ;   |
|                        |                 | wait ( s1 ) ;   |
|                        |                 | ....            |
|                        |                 | signal ( s1 ) ; |
| <b>signal ( s2 ) ;</b> |                 |                 |
|                        | signal ( s2 );  |                 |
|                        |                 | signal ( s2 );  |

```
// Now the state of s1 and s2 would be as follows
```

```
s1->value = 1 , s1->list = NULL  
s2->value = 2 , s2->list -> NULL
```

# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $s$  and  $q$  be two semaphores initialized to 1

$P_0$

```
wait(S);  
wait(Q);  
...  
signal(S);  
signal(Q);
```

$P_1$

```
wait(Q);  
wait(S);  
...  
signal(Q);  
signal(S);
```

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process (L) holds a lock needed by higher-priority process (H)
  - Solved through **priority-inheritance protocol**

# Classical Problems of Synchronization

- Classical problems used to test newly proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers 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);
```

# 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 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);
```

# Readers-Writers Problem Variations

- ***First*** variation – no reader is kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- They don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

# Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
        // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
        // think  
} while (TRUE);
```

- What is the problem with this algorithm?
  - Ans: Deadlock
  - If all five philosophers become hungry at the same time and each grabs his/her left chopstick. All the elements of chopstick will now be equal to 0.
  - When each philosopher tries to grab his/her right chopstick, he/she will be delayed forever.

## Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

# Monitors

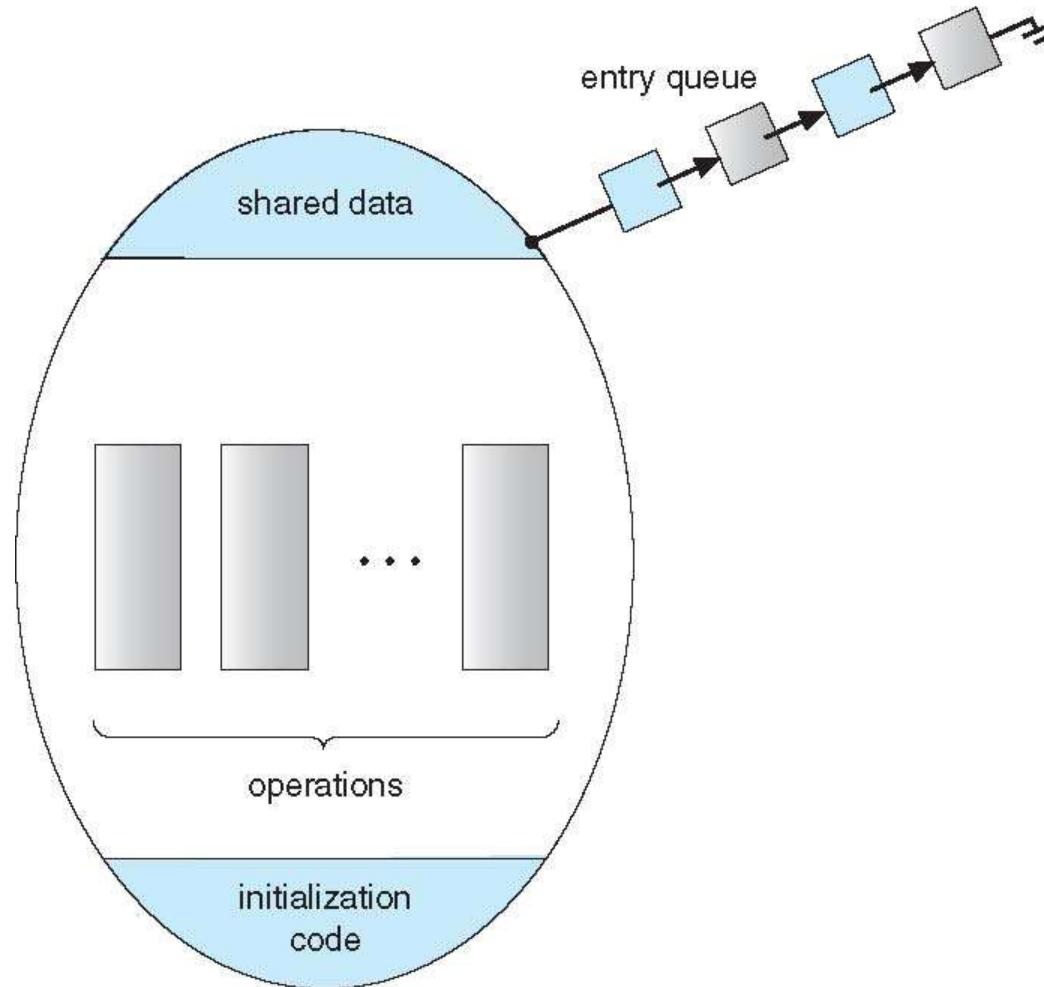
- A **high-level abstraction** that provides a convenient and effective mechanism **for process synchronization**
- **An Abstract data type**, internal variables are only accessible by code within the procedure
- **Only one process may be active within the monitor at a time.**
- But not powerful enough to model some synchronization schemes, hence **condition constructs** are introduced.

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
```

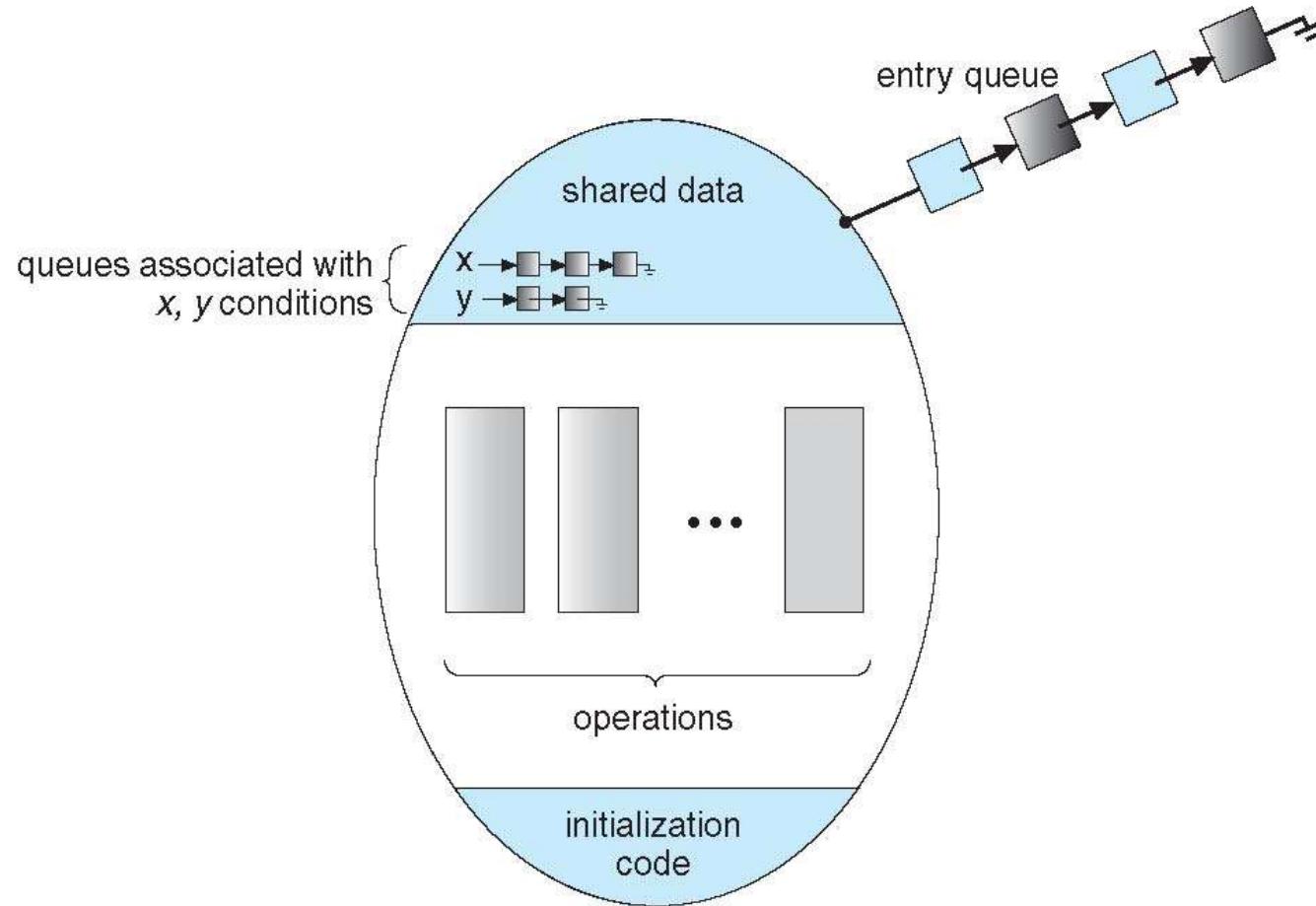
# Schematic view of a Monitor



# Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

# Monitor with Condition Variables



# Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in **Concurrent Pascal** compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }
    void putdown (int i) {
        state[i] = THINKING;
                    // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }
}
```

# Solution to Dining Philosophers (Cont.)

```
/*void test (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}
*/ initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```

# Solution to Dining Philosophers (Cont.)

- Each philosopher  $i$  invokes the operations `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup(i) ;`

**EAT**

`DiningPhilosophers.putdown(i) ;`

- No deadlock, but starvation is possible

# Examples

- Sleeping Barber's Problem
- Cigarette Smoker's Problem

# Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Each procedure *F* will be replaced by

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

# Monitor Implementation – Condition Variables

- For each condition variable  $x$ , we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation  $x.wait$  can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

# Monitor Implementation (Cont.)

- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```

# Resuming Processes within a Monitor

- If several processes queued on condition  $x$ , and  $x.signal()$  executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form  $x.wait(c)$ 
  - Where  $c$  is **priority number**
  - Process with lowest number (highest priority) is scheduled next

# Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);  
    ...  
access the resource;  
    ...  
  
R.release;
```

- Where R is an instance of type `ResourceAllocator`

# A Monitor to Allocate Single Resource

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

# Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
    - Timers notify one or more thread when time expired
    - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

# Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

# Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()  
{  
    /* read/write memory */  
}
```

# OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the **#pragma omp critical** directive is treated as a critical section and performed atomically.

# Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

# Memory Management

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

# Background

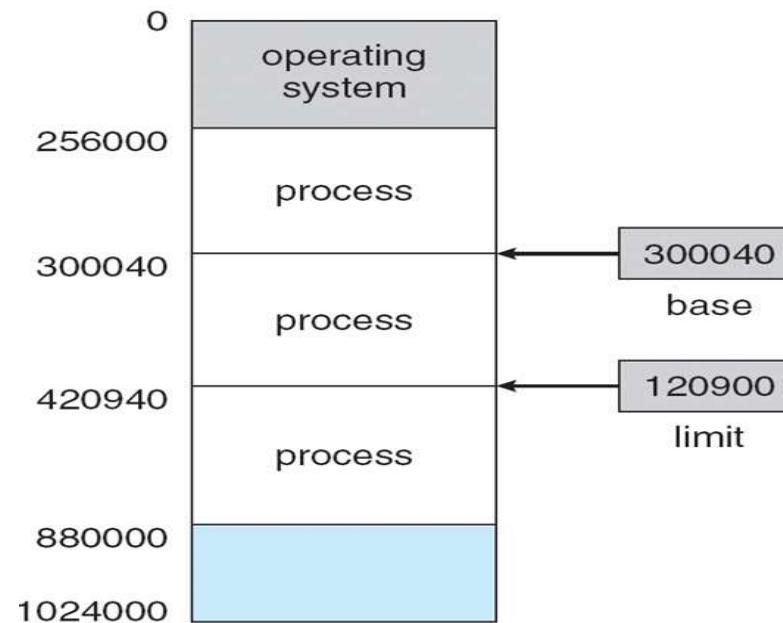
- Program must be brought (from disk) into memory and placed within a process for it to be run
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- Main memory and registers are the only storage which can be accessed by CPU directly

# Background

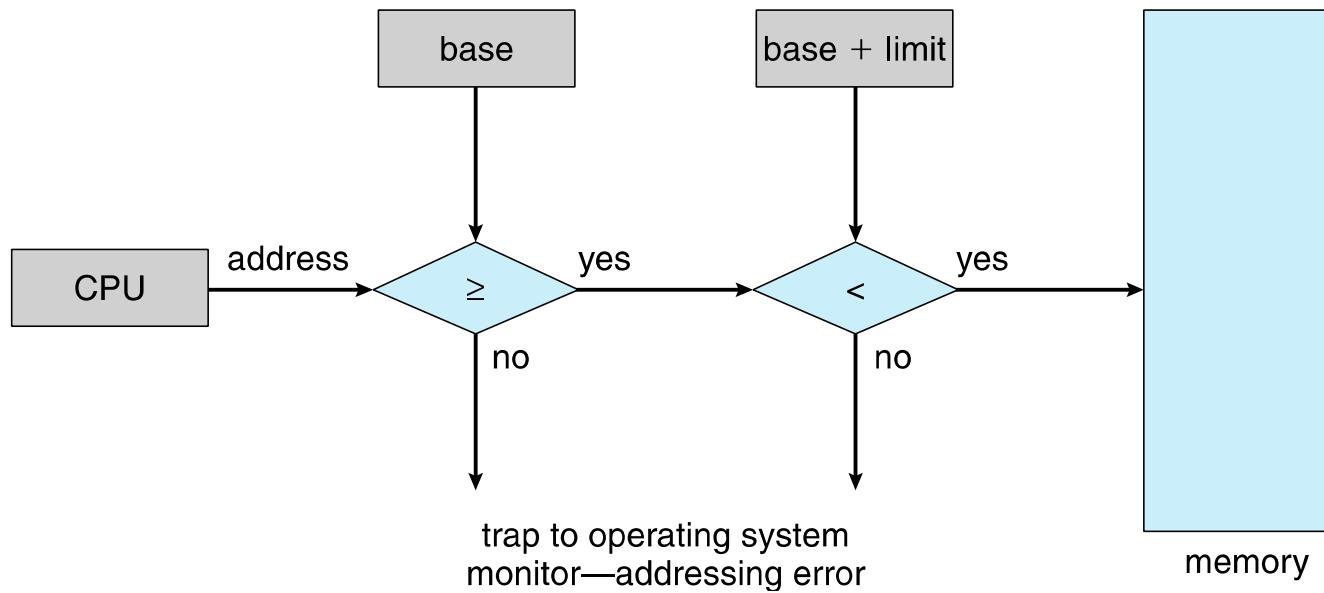
- Program must be brought (from disk) into memory and placed within a process for it to be run
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- Main memory and registers are the only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests 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 is required to ensure correct operation

# Base and Limit Registers

- A pair of registers (**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 registers for that user



# Hardware Address Protection



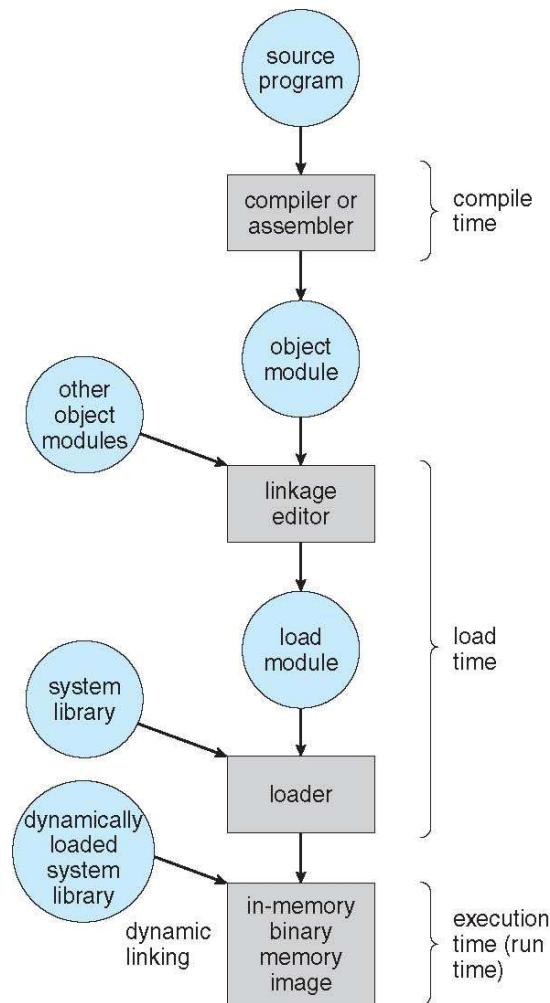
# Address Binding

- 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
  - How can it not be?
- 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**
    - i.e. “14 bytes from beginning of this module”
  - **Linker or loader** will bind relocatable addresses to **absolute addresses**
    - i.e. 74014
  - 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:** Compiler generates **relocatable code** if memory location is not known at compile time. Binding is delayed until load 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

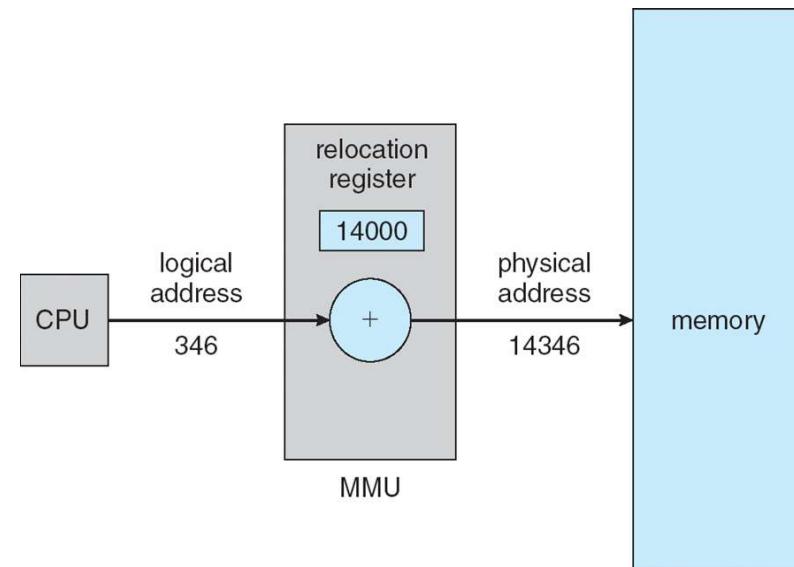
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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)

- Hardware device maps virtual to physical address at run time
- 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
  - Logical address bound to physical addresses

# 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



# Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded.
- 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.

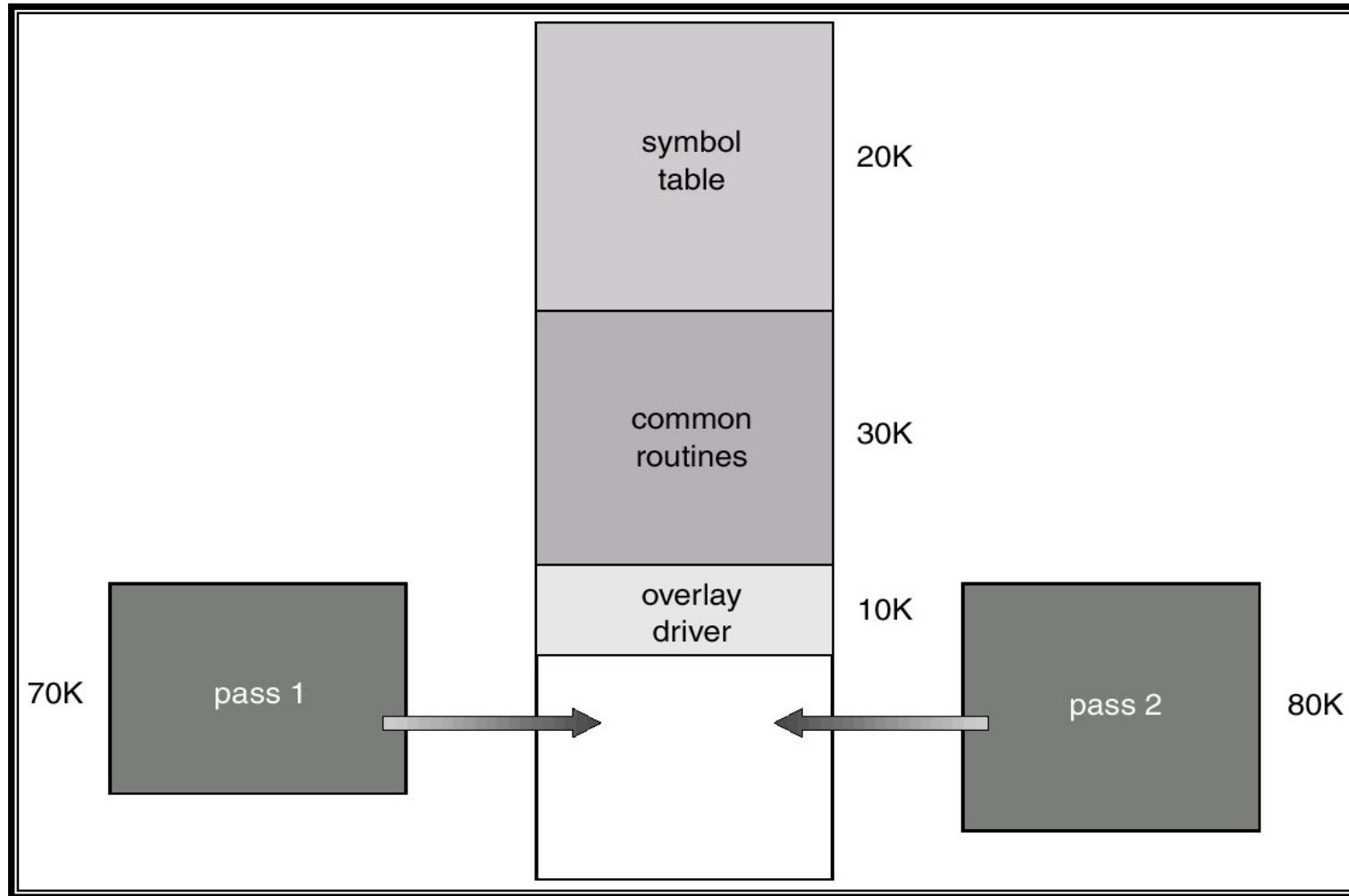
# Dynamic Linking

- **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
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System libraries also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

# Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex

# Overlays for a Two-Pass Assembler



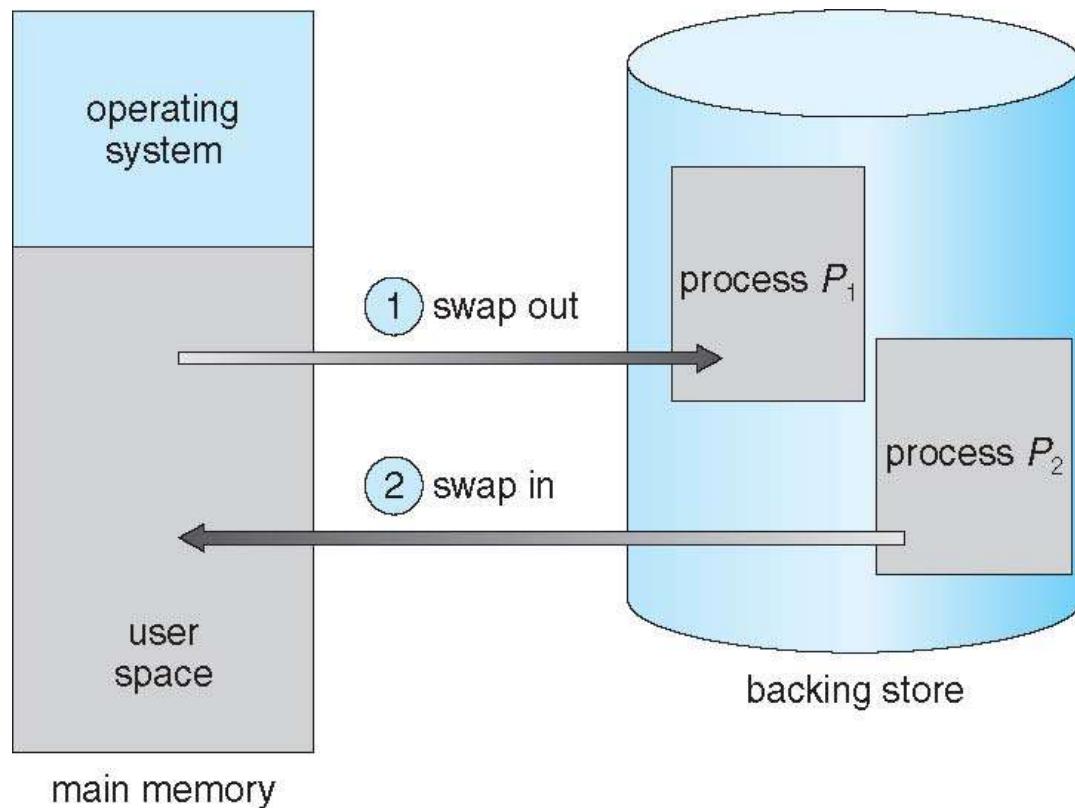
# Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **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
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

# Swapping (Cont.)

- 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 Swapping



# Context Switch Time including Swapping

- 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
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce 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()`

## Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or 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
  - But modified version common
    - 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 restart
  - Both OSes support paging

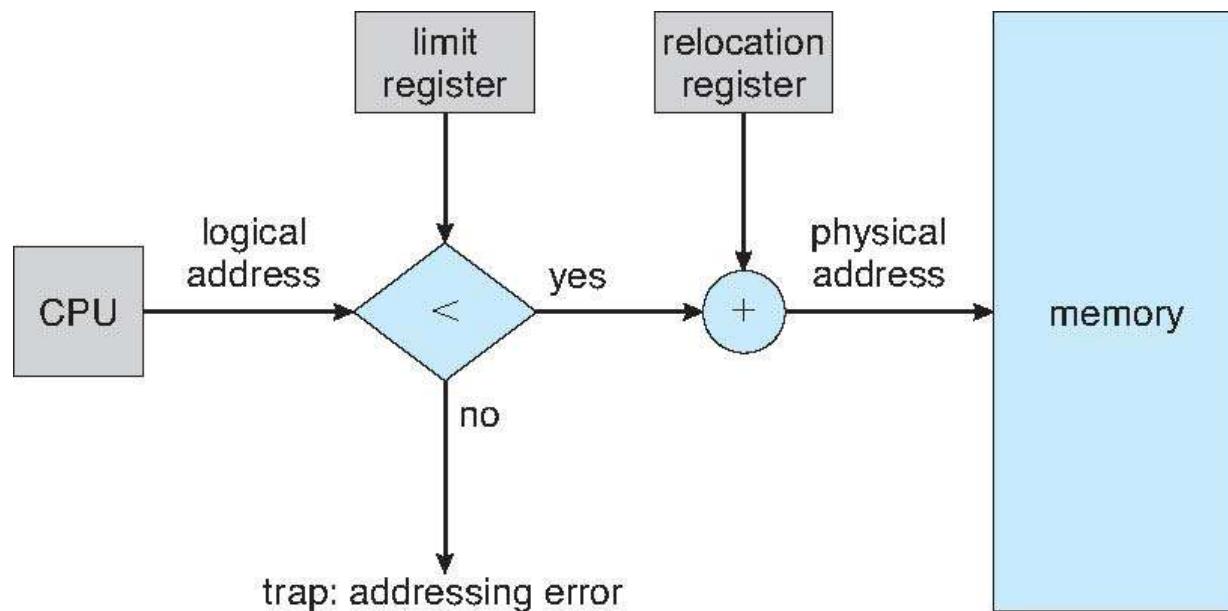
# Contiguous Allocation

- 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

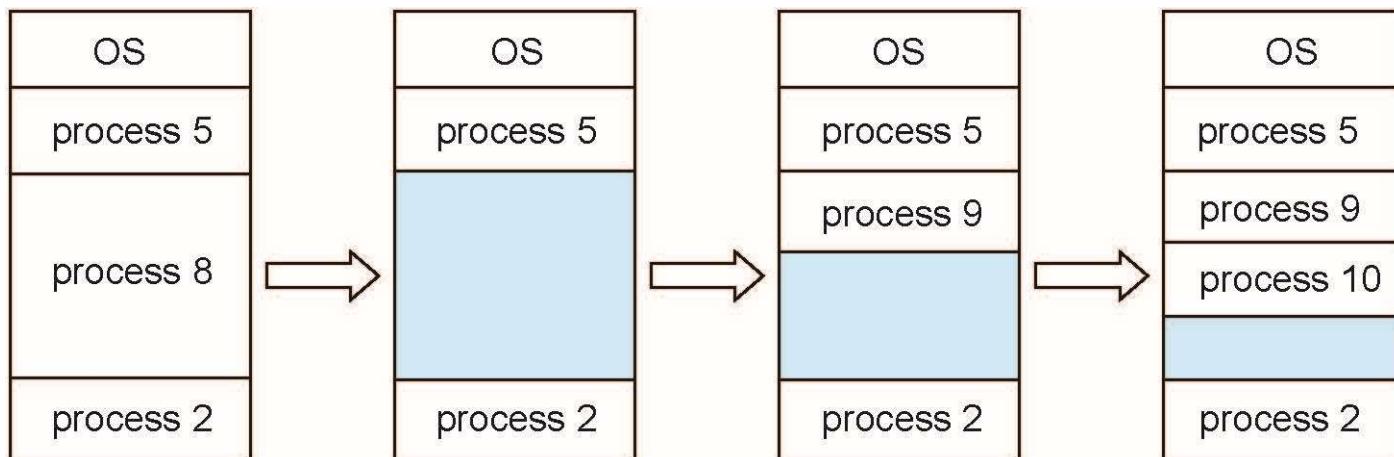
# Hardware Support for Relocation and Limit Registers



# 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
- **Next fit** is a modified version of first fit. It begins as first fit to find a free partition. When called next time it starts searching from where it left off, not from the beginning.
- **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
  - Internal fragmentation is unused allocated area
  - External fragmentation is un-allocated area and unused
- First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

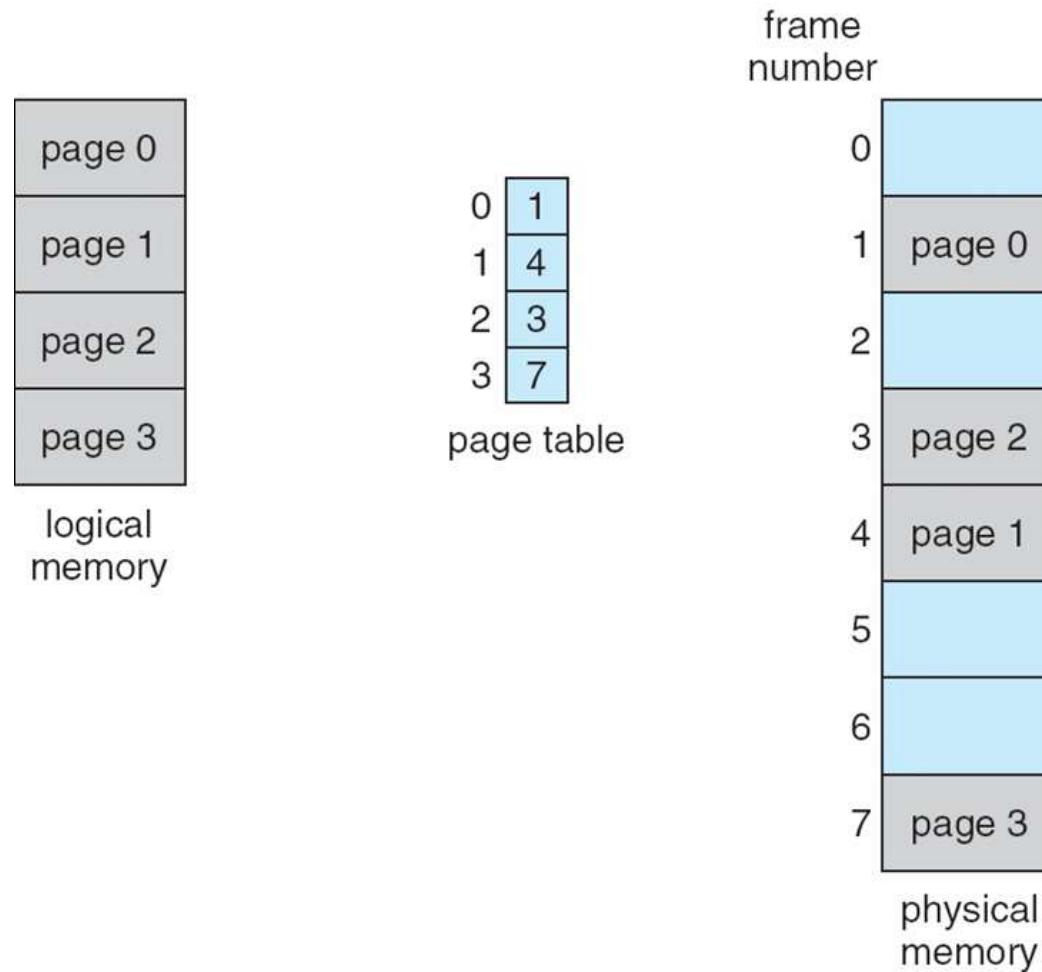
# Fragmentation (Cont.)

- 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

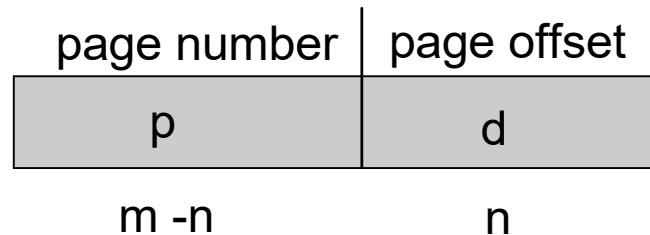
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, 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 is also split into pages
  - Still have Internal fragmentation

# Paging Model of Logical and Physical Memory



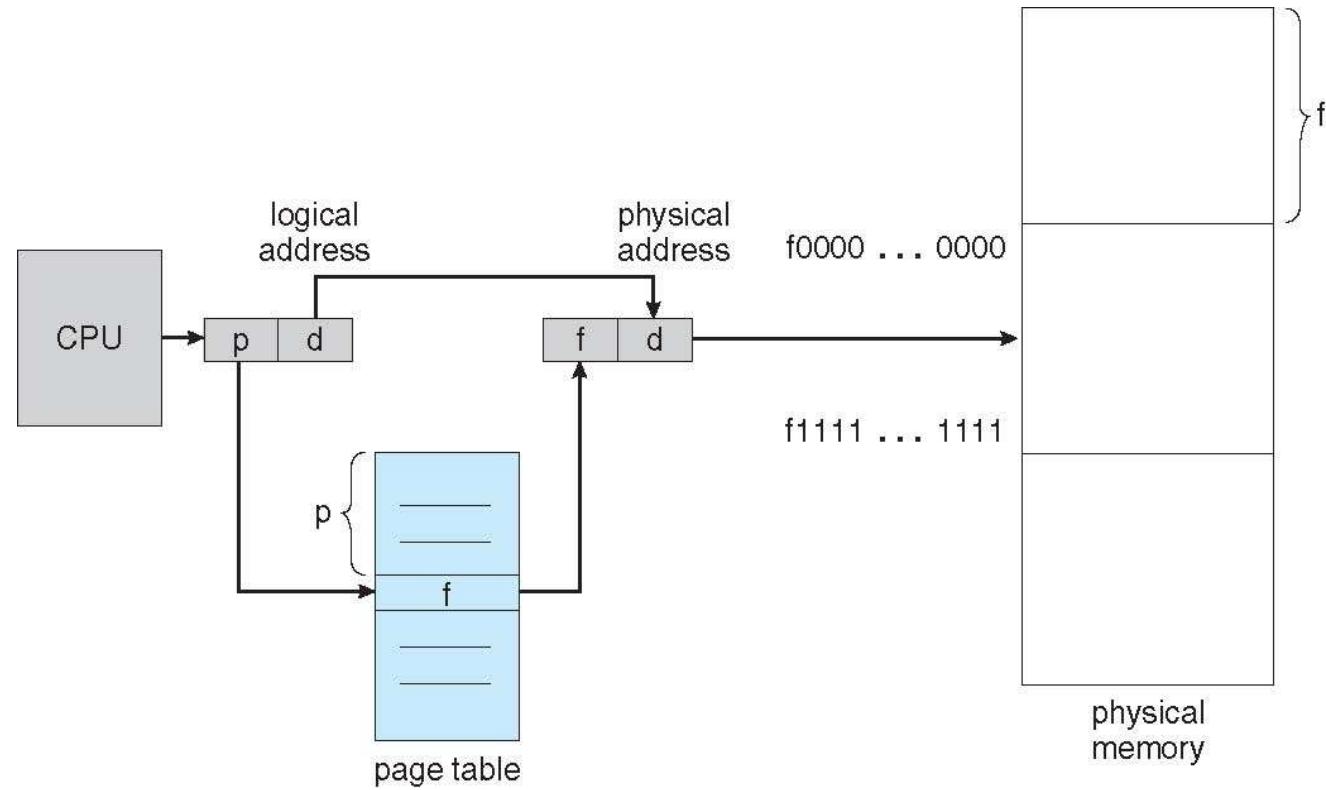
# Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number ( $p$ )** – used as an index into a **page table** which contains **base address**(or frame number) 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 a given logical address space  $2^m$  and page size  $2^n$

# Paging Hardware



# Paging Example

|    |   |
|----|---|
| 0  | a |
| 1  | b |
| 2  | c |
| 3  | d |
| 4  | e |
| 5  | f |
| 6  | g |
| 7  | h |
| 8  | i |
| 9  | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

logical memory

|   |   |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

|    |                  |
|----|------------------|
| 0  |                  |
| 4  | i<br>j<br>k<br>l |
| 8  | m<br>n<br>o<br>p |
| 12 |                  |
| 16 |                  |
| 20 | a<br>b<br>c<br>d |
| 24 | e<br>f<br>g<br>h |
| 28 |                  |

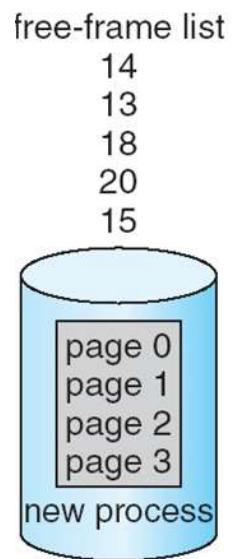
physical memory

$n=2$  and  $m=4$  32-byte memory and 4-byte pages

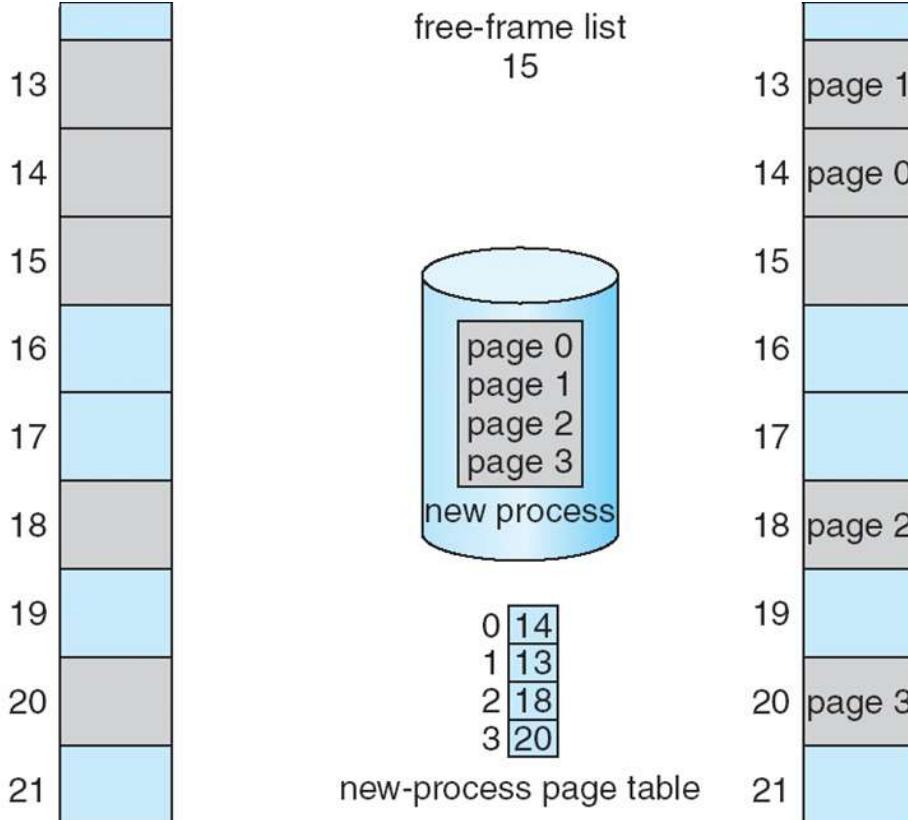
# Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - **Process size = 72,766 bytes**
    - 35 pages + 1,086 bytes
  - Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
  - **Worst case fragmentation = 1 frame – 1 byte** ( Process requires  $n$  pages + 1 byte)
  - On average fragmentation =  $1 / 2$  frame size
  - So small frame sizes desirable
  - Each page table entry takes memory to track
  - Page sizes grow over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory are very different
- By implementation process can only access its own memory

# Free Frames



(a)



(b)

Before allocation

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/PIDs)** 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

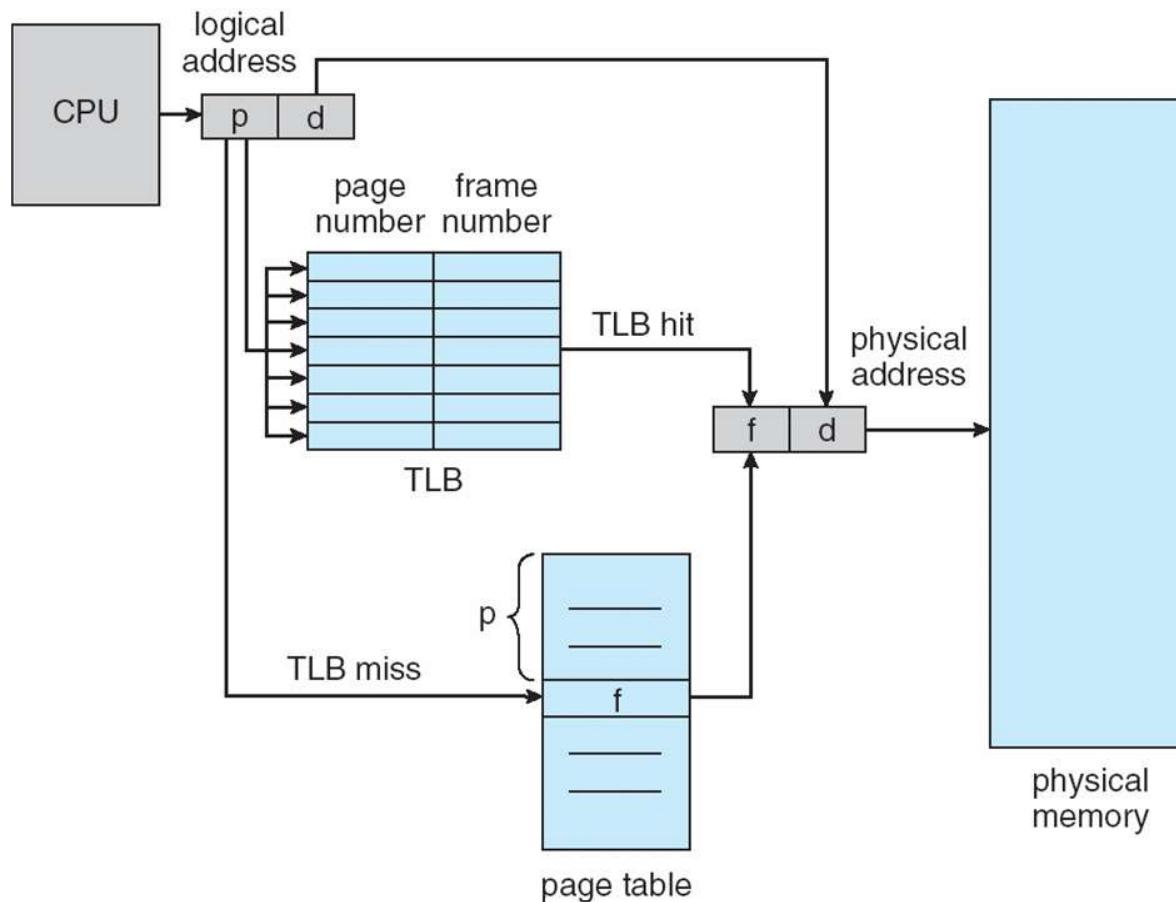
# Associative Memory

- Associative memory – parallel search

| Page # | Frame # |
|--------|---------|
|        |         |
|        |         |
|        |         |
|        |         |

- Address translation (p, d)
  - If p is in associative register, get frame #
  - Otherwise get frame # from page table in memory

# Paging Hardware With TLB



# Effective Access Time

- **Associative Lookup =  $\varepsilon$  time unit**
  - Can be < 10% of memory access time
- **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)**
$$EAT = (\text{memory access time} + \varepsilon) \alpha + (2 * \text{memory access time} + \varepsilon)(1 - \alpha)$$
- Assume memory cycle time is 1 microsecond
- **Effective Access Time (EAT)**
$$\begin{aligned} EAT &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

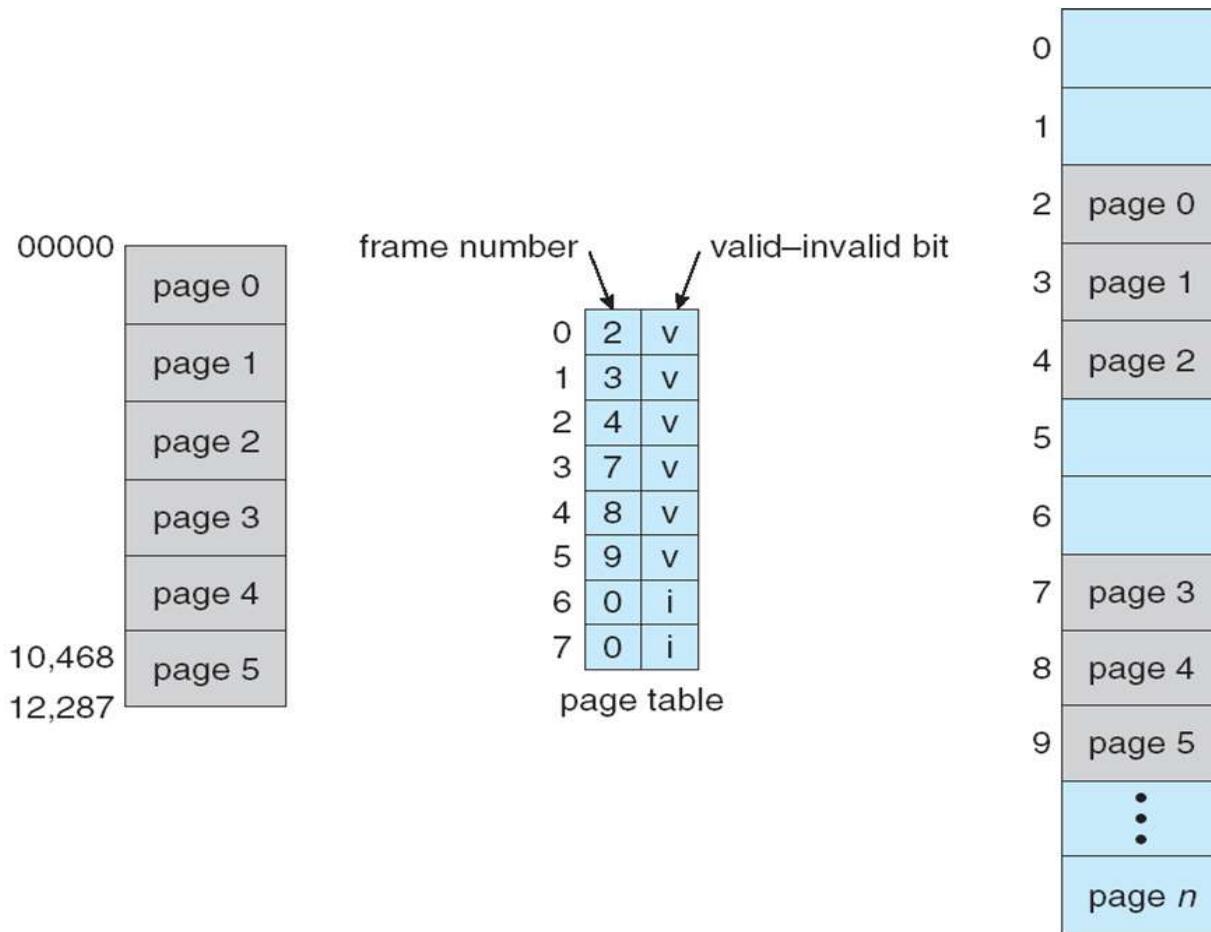
Example:

- Consider  $\alpha = 80\%$ ,  $\varepsilon = 20\text{ns}$  for TLB search, 100ns for memory access
  - $EAT = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$
- Consider more realistic hit ratio  $\rightarrow \alpha = 98\%$ ,  $\varepsilon = 20\text{ns}$  for TLB search, 100ns for memory access
  - $EAT = 0.98 \times 120 + 0.02 \times 220 = 122\text{ns}$

# Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if **read-only or read-write access** is allowed
  - Can also add more bits to indicate page **execute-only**, and so on
- **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’s logical address space
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel

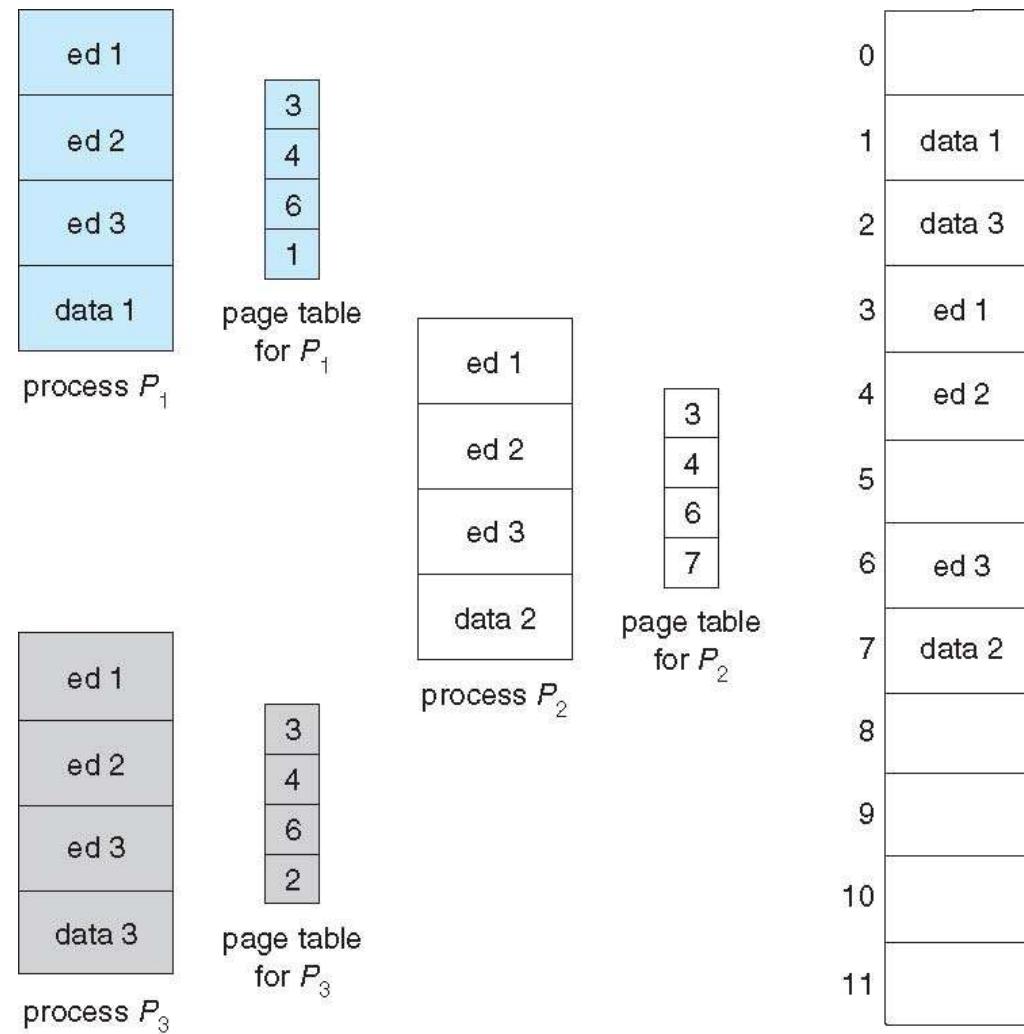
# Valid (v) or Invalid (i) Bit In A Page Table



# Shared Pages

- A process may have shared block of code, private code and data
- Shared code
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

# Shared Pages Example



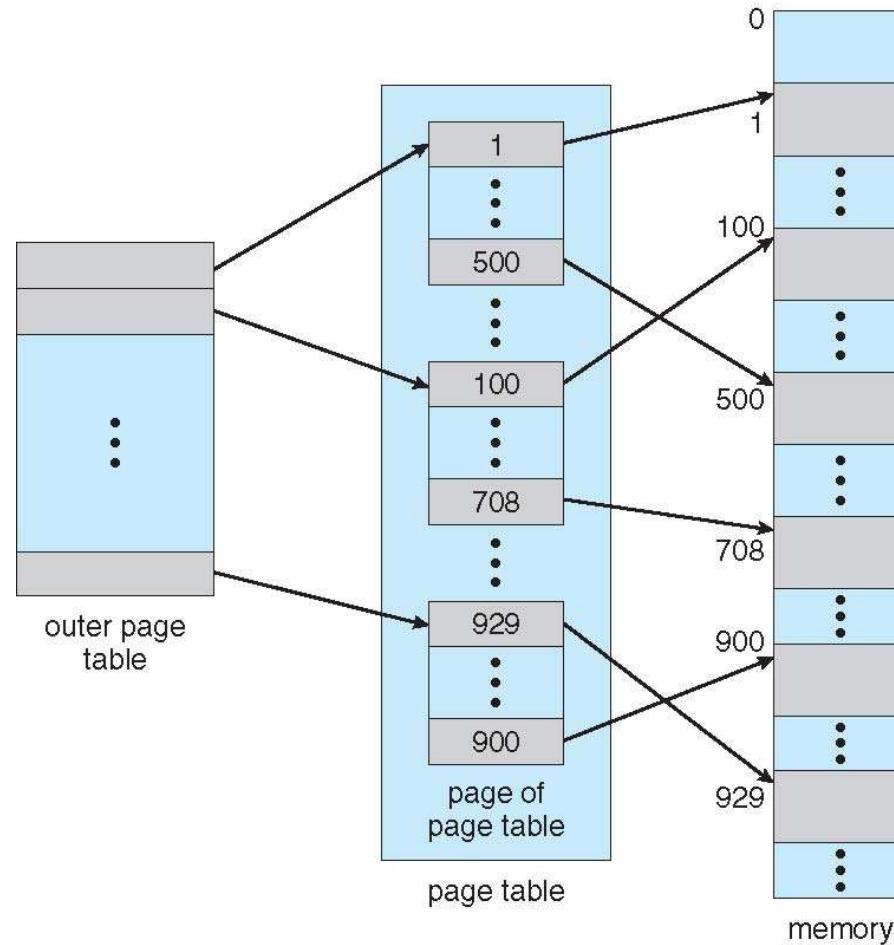
# Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a **32-bit logical address space** as on modern computers
  - **Page size of 4 KB ( $2^{12}$ )**
  - **Page table would have 1 million entries ( $2^{32} / 2^{12}$ )**
  - If each entry is 4 bytes  $\rightarrow$  **4 MB of physical address space / memory for page table alone**
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

# Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

# Two-Level Page-Table Scheme



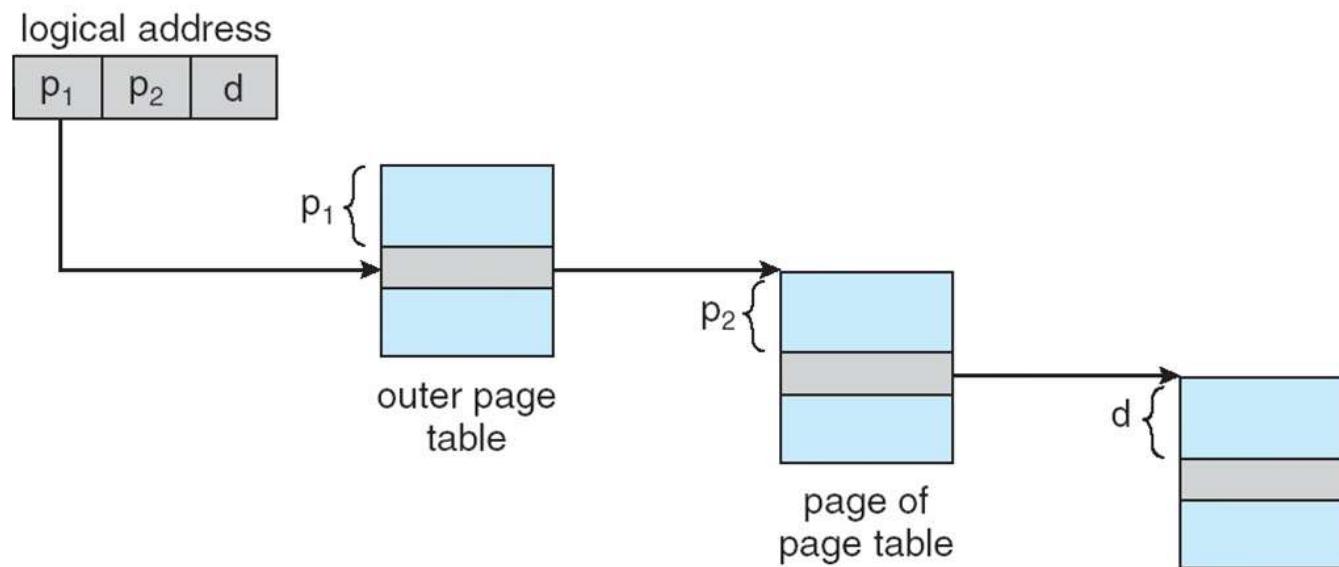
# Two-Level Paging Example

- A logical address (**on 32-bit machine with 1K page size**) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

| page number |       | page offset |
|-------------|-------|-------------|
| $p_1$       | $p_2$ | $d$         |
| 12          | 10    | 10          |

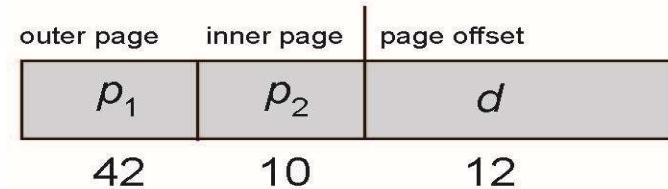
- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

# Address-Translation Scheme



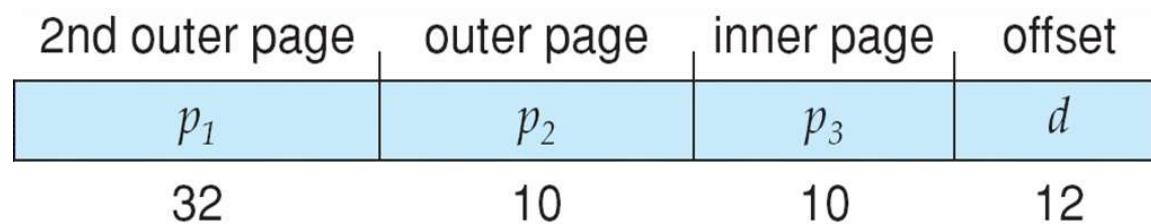
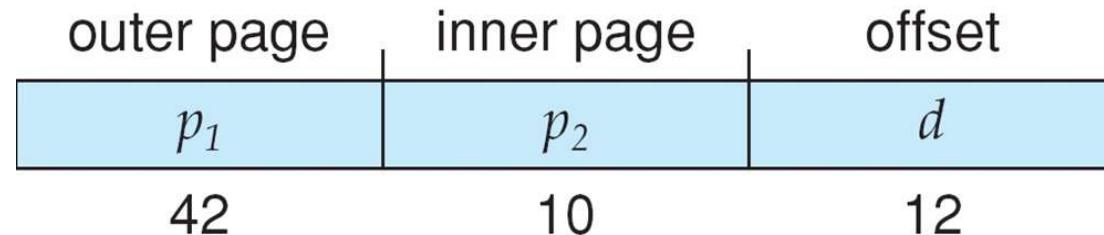
# 64-bit Logical Address Space

- Even two-level paging scheme is not sufficient
- If **page size is 4 KB ( $2^{12}$ )**
  - Then **page table has  $2^{52}$  entries**
  - If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
  - Address would look like



- Outer page table has  $2^{42}$  entries
- One solution is to add a 2<sup>nd</sup> outer page table
- But in the following example the 2<sup>nd</sup> outer page table is still  $2^{34}$  bytes in size
  - And possibly 4 memory access to get to one physical memory location

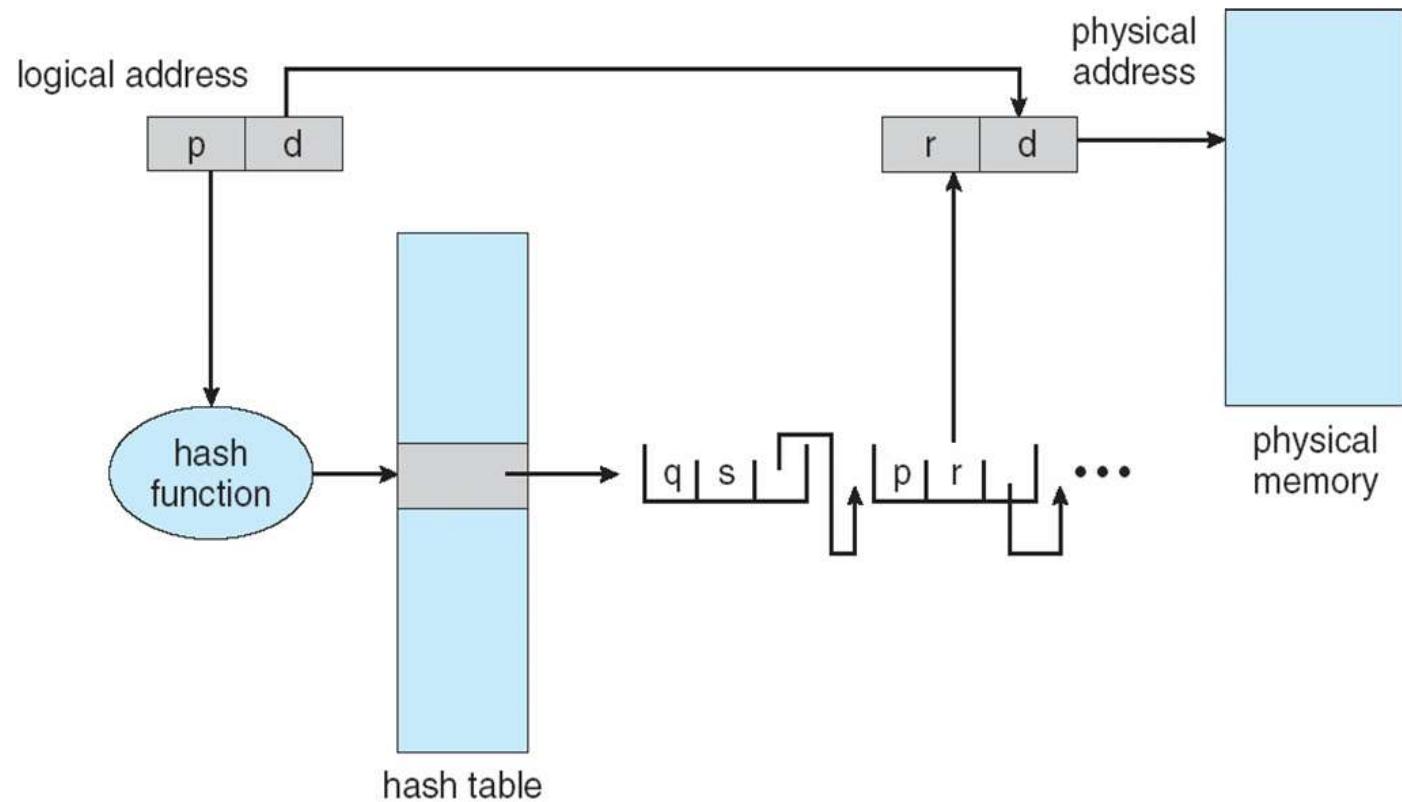
# Three-level Paging Scheme



# Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

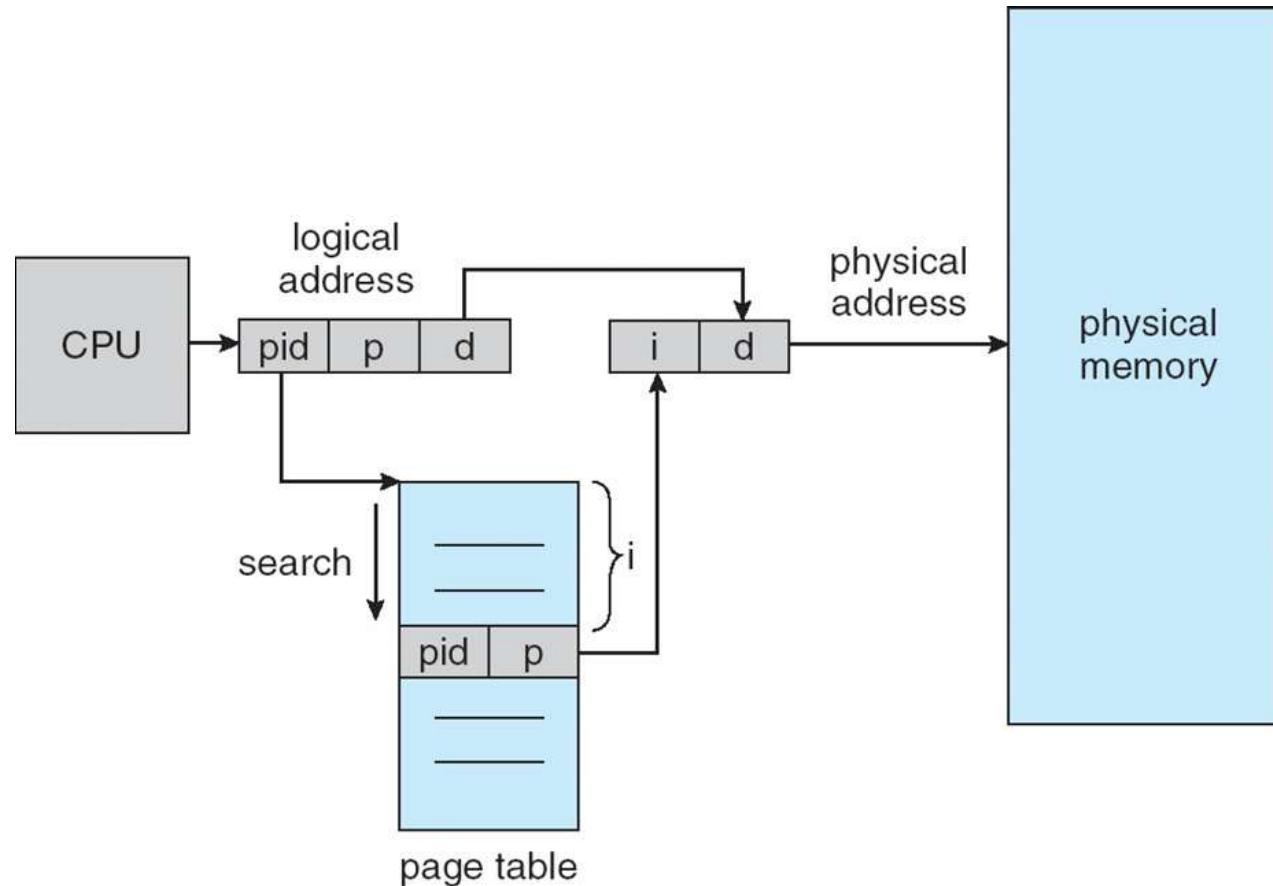
# Hashed Page Table



# Inverted Page Table

- One page table is used in a system to track the mapping of logical address to physical address
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

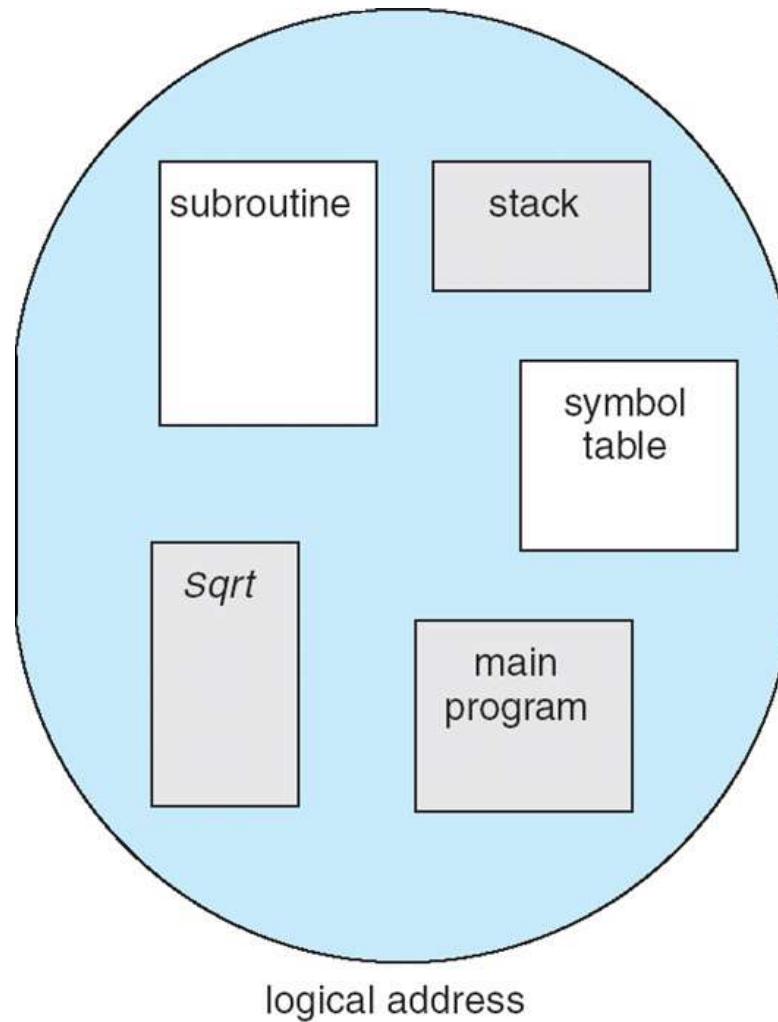
# Inverted Page Table Architecture



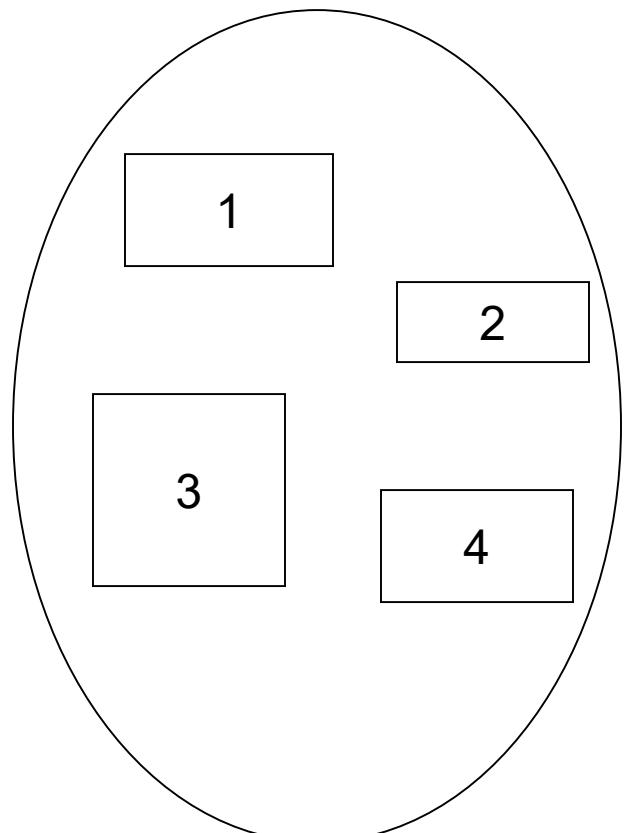
# Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays

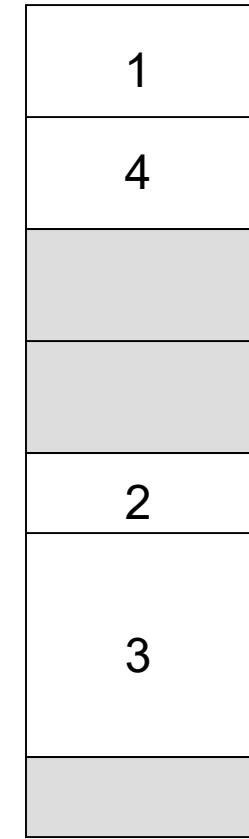
# User's View of a Program



# Logical View of Segmentation



user space



physical memory space

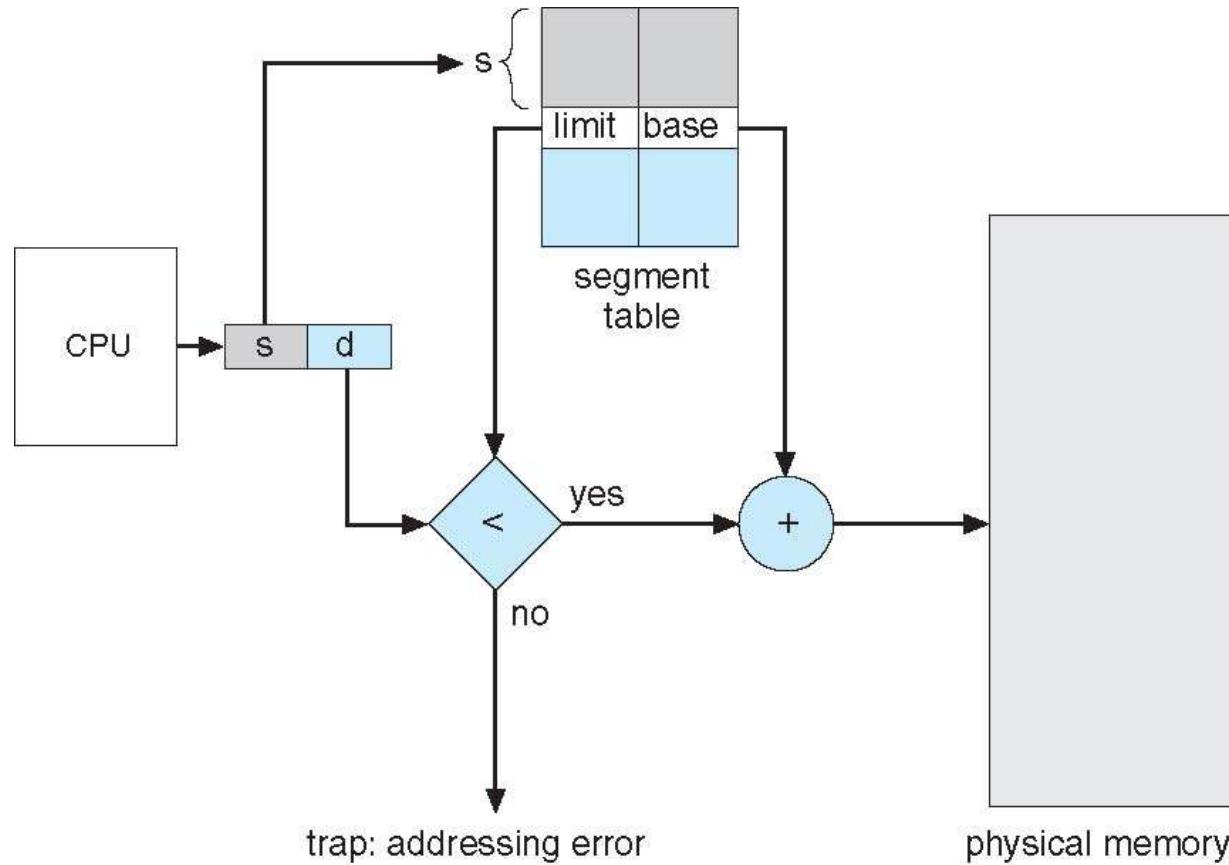
# Segmentation Architecture

- Logical address consists of a two tuple:  
 $\langle \text{segment-number}, \text{offset} \rangle$ ,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;  
segment number **s** is legal if  **$s < STLR$**

# Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0  $\Rightarrow$  illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

# Segmentation Hardware



# Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

# Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

## Example: The Intel 32 and 64-bit Architectures

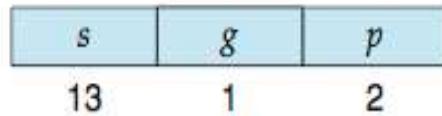
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

## Example: The Intel IA-32 Architecture

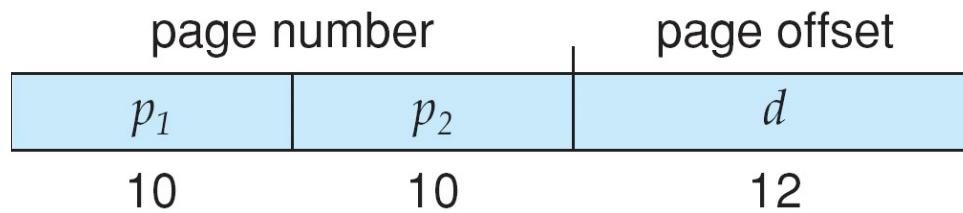
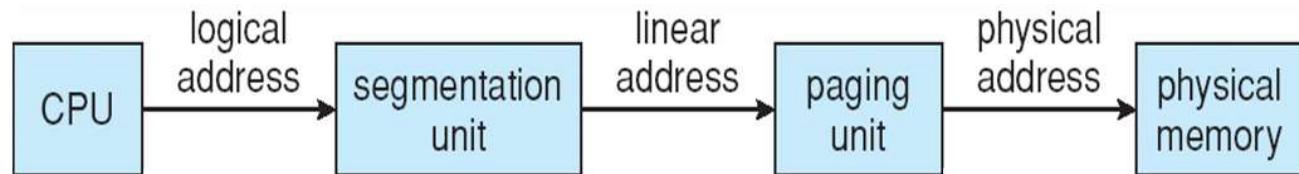
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)

## Example: The Intel IA-32 Architecture (Cont.)

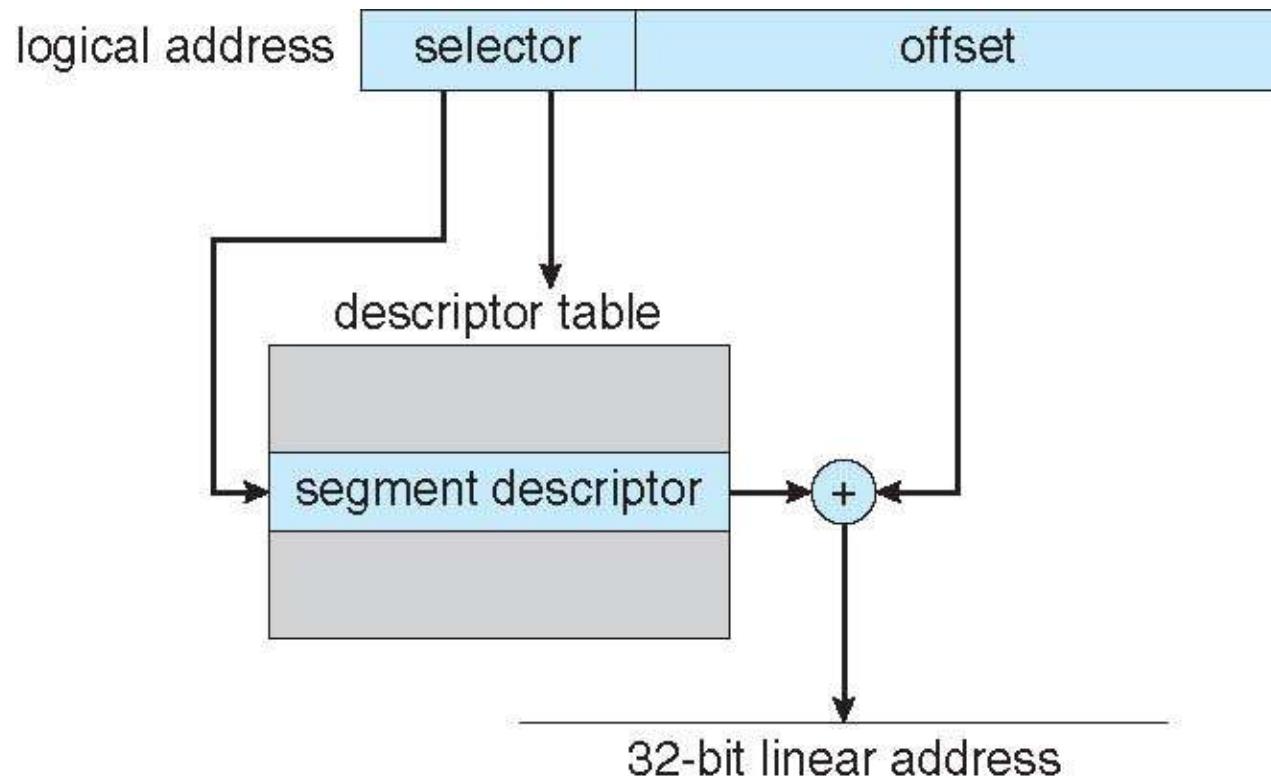
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB



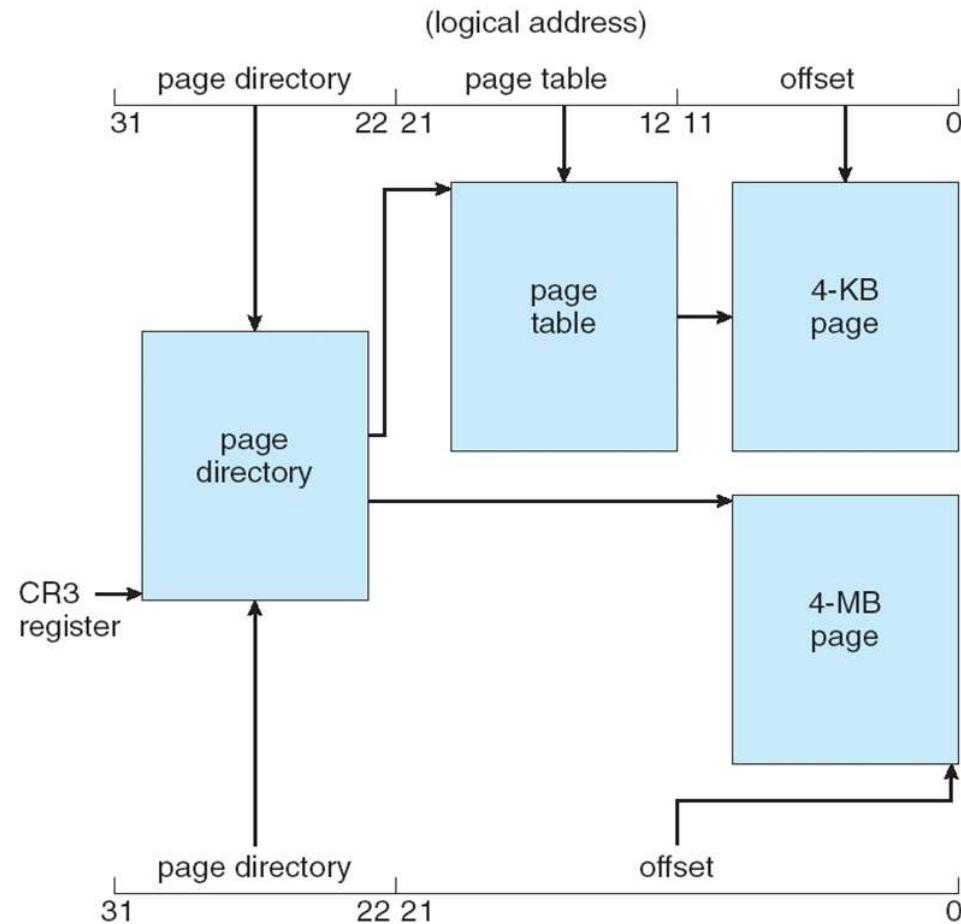
# Logical to Physical Address Translation in IA-32



# Intel IA-32 Segmentation

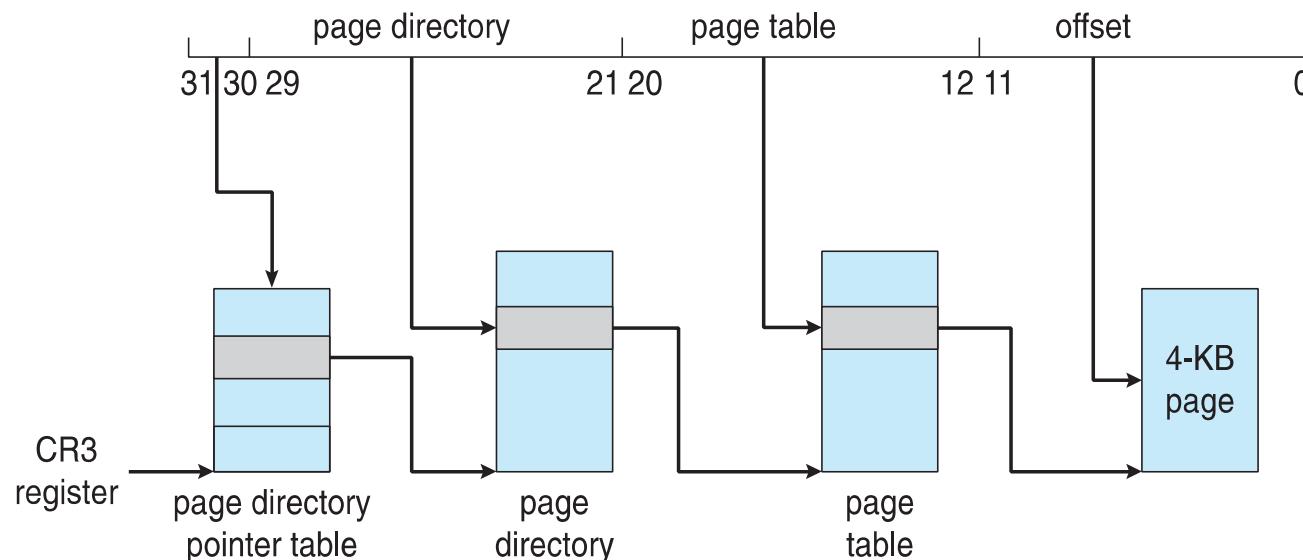


# Intel IA-32 Paging Architecture



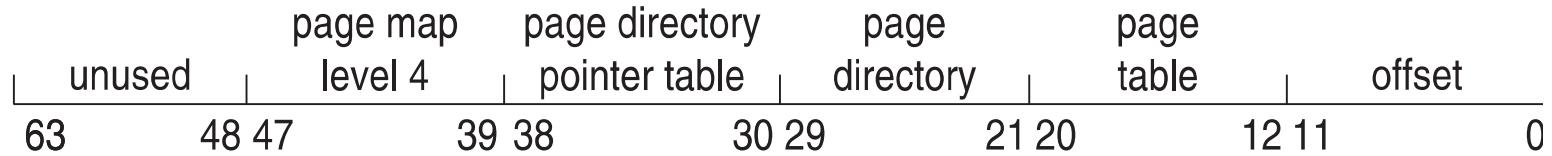
# Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



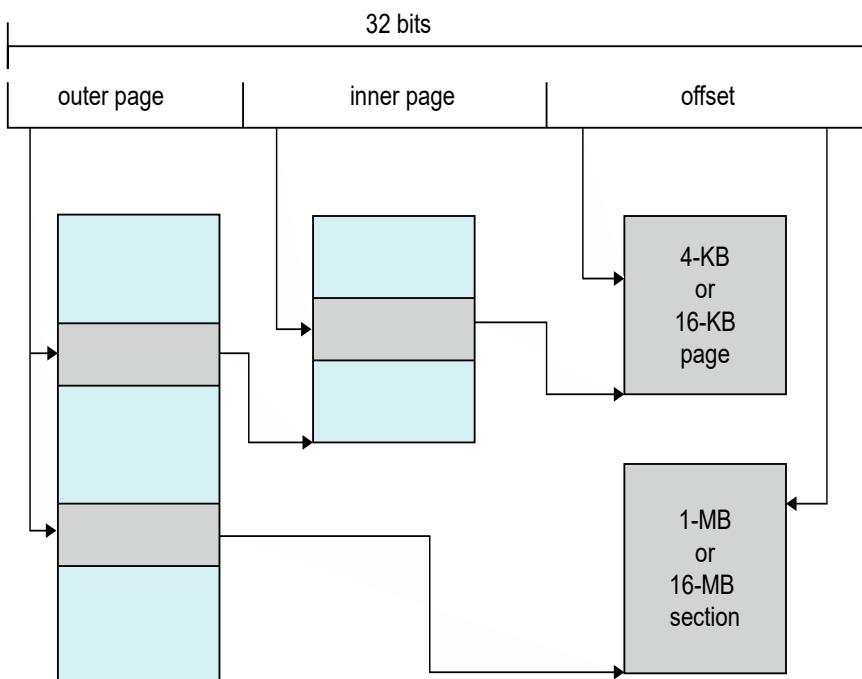
# Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



# Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



# Module6\_VirtualMemory

Reference: “OPERATING SYSTEM CONCEPTS”, ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE , Wiley publications.

# Background

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- Code needs to be in memory to execute, but entire program is rarely used
  - Error code, unusual routines, large data structures
- Entire program code is not needed at same time
- Consider the ability to execute partially-loaded program
  - Programmer is no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs can be run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

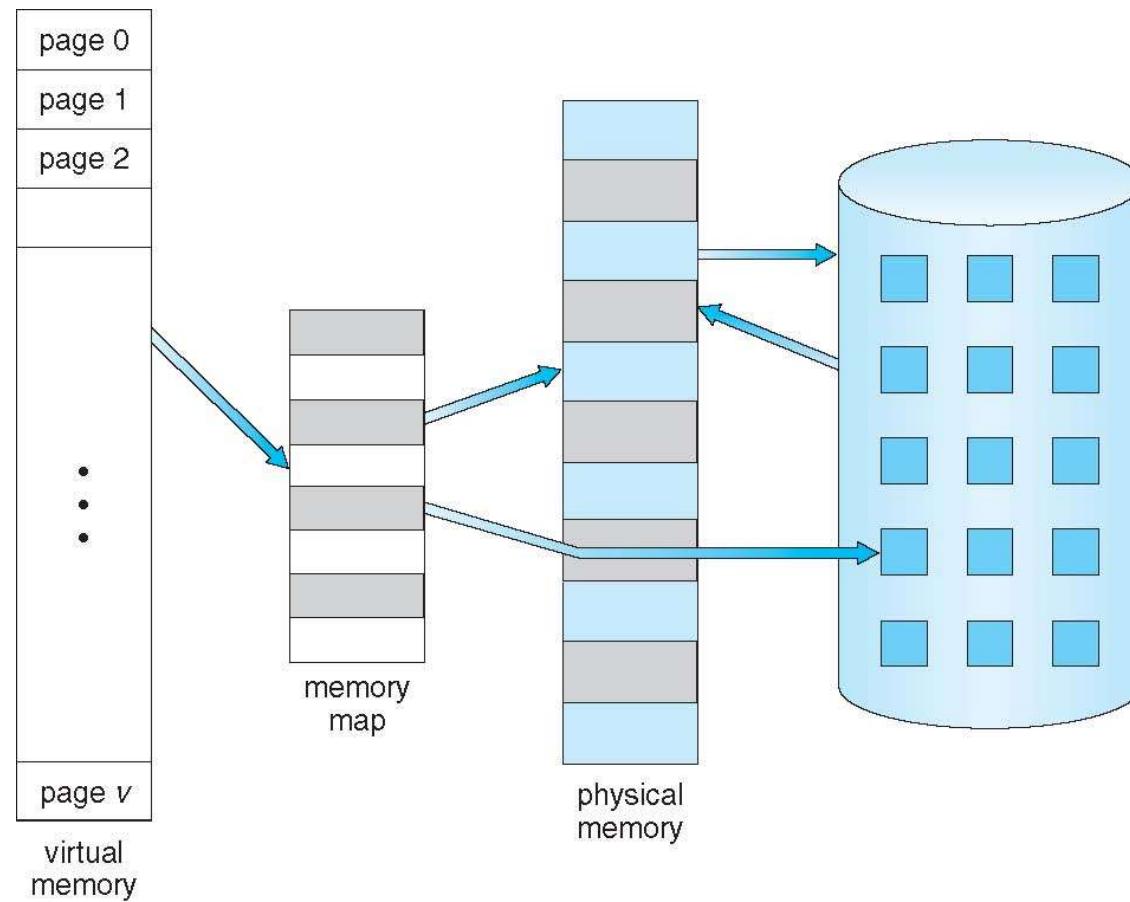
# Background (Cont.)

- **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

# Background (Cont.)

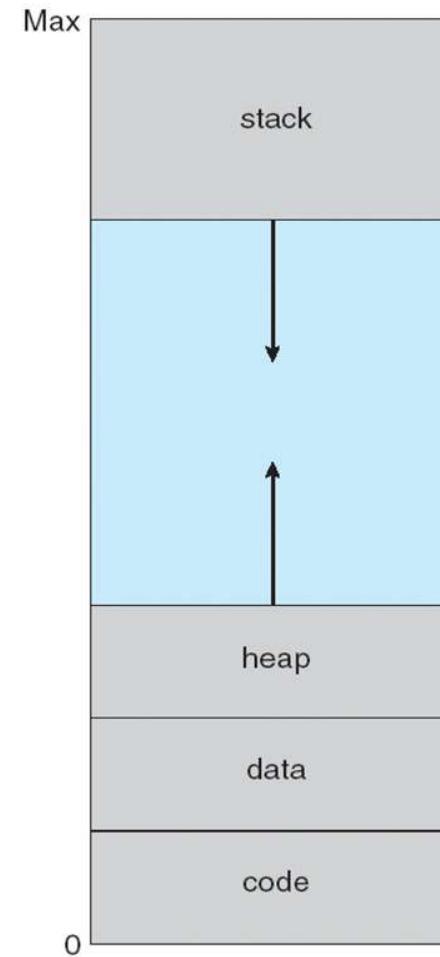
- Virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory
- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Physical memory organized as frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory that is Larger than Physical Memory

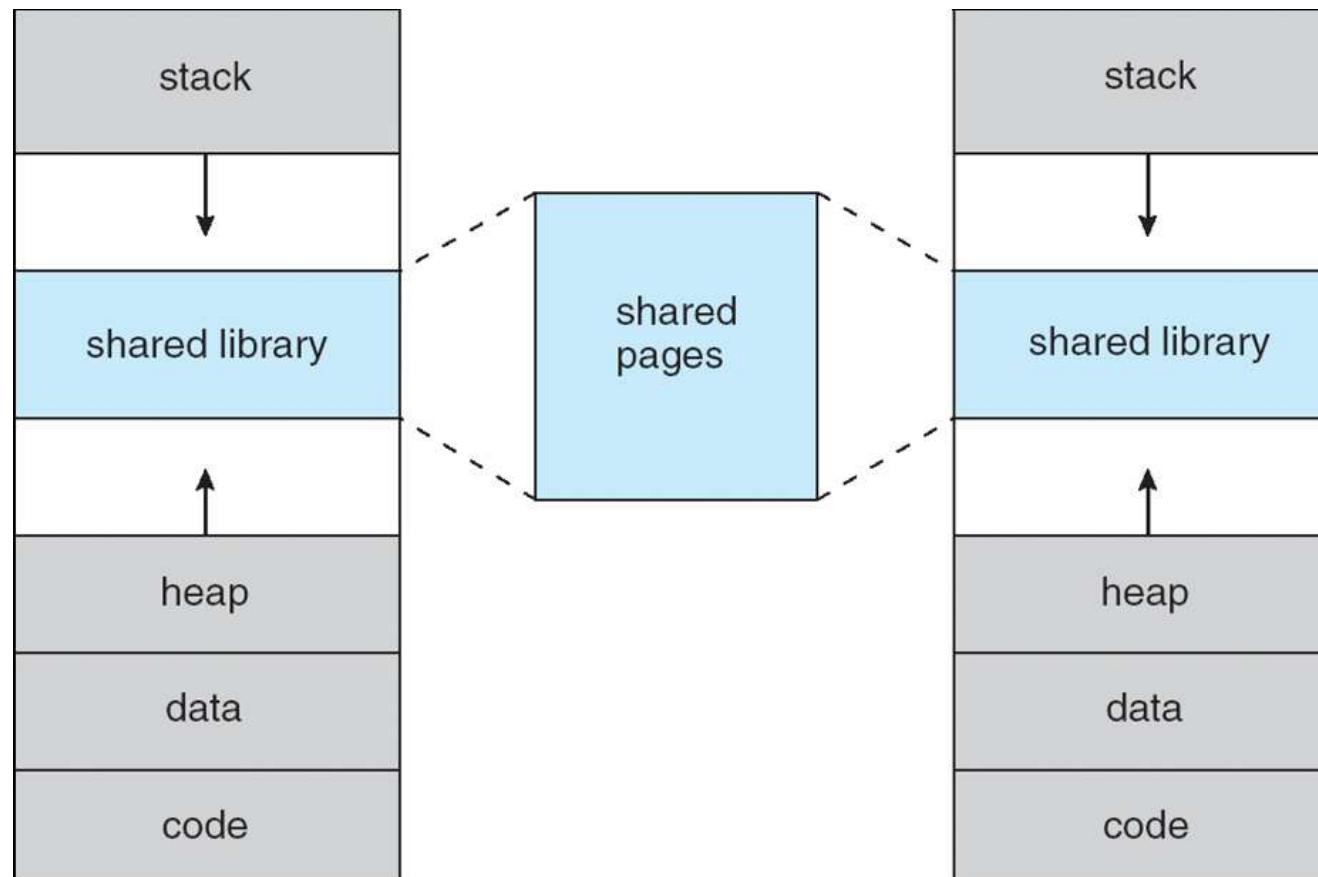


# Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - **Maximizes address space use**
  - Unused address space between the two is hole
    - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries are 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

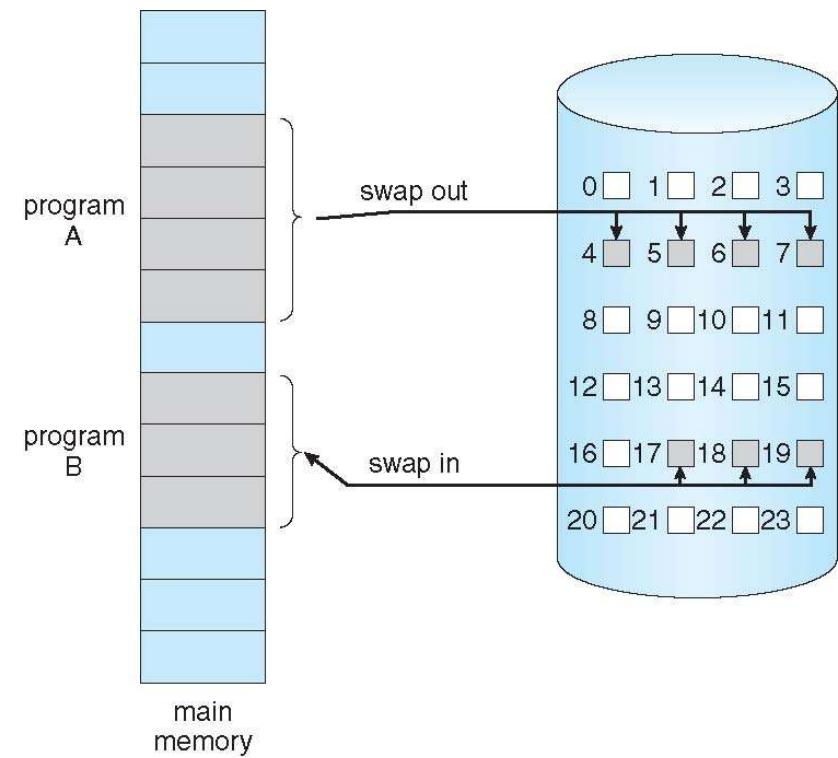


# Shared Library Using Virtual Memory



# 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
- Similar to paging system with swapping (diagram on right)
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



# Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, **pager brings in only those pages into memory**
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

# Valid-Invalid Bit

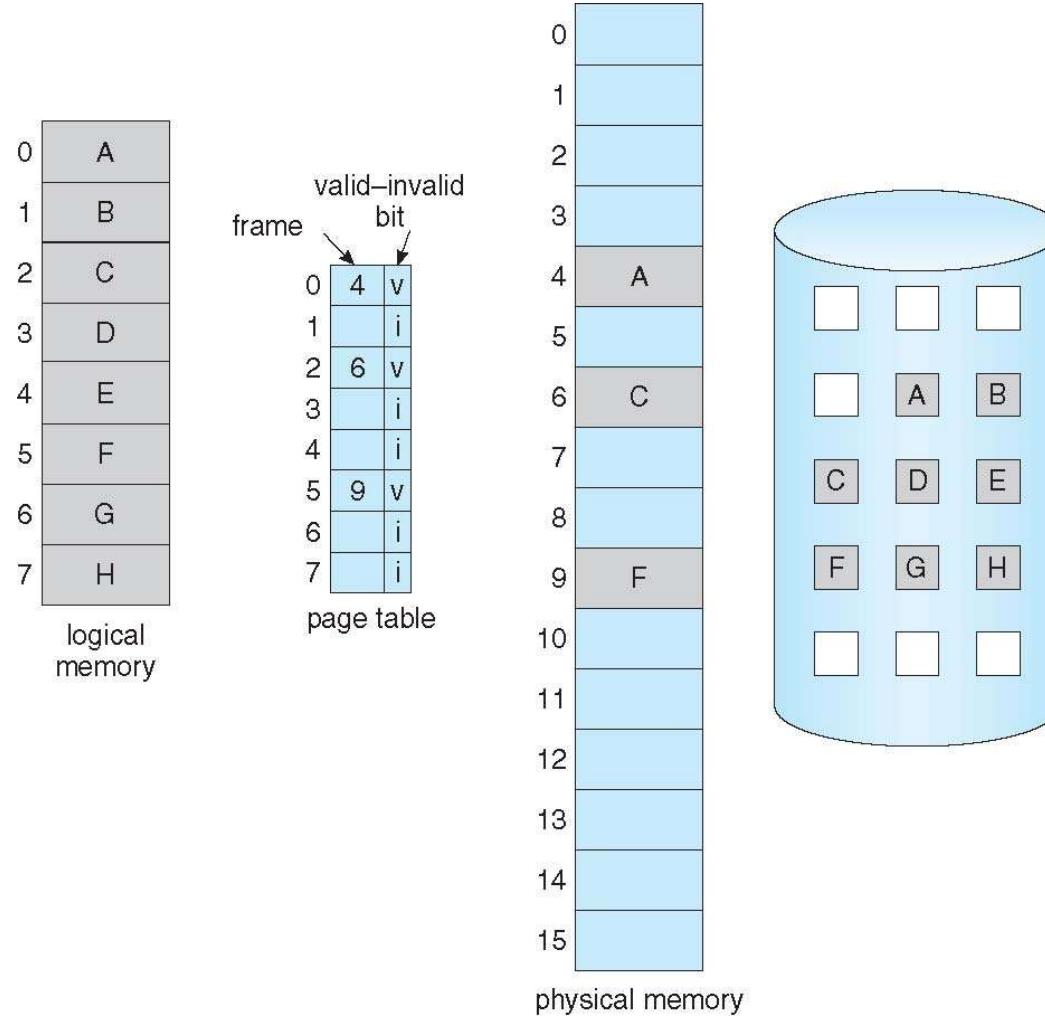
- With each page table entry a valid–invalid bit is associated (**v**  $\Rightarrow$  in-memory – **memory resident**, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to i on all entries**
- Example of a page table snapshot:

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

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i**  $\Rightarrow$  page fault

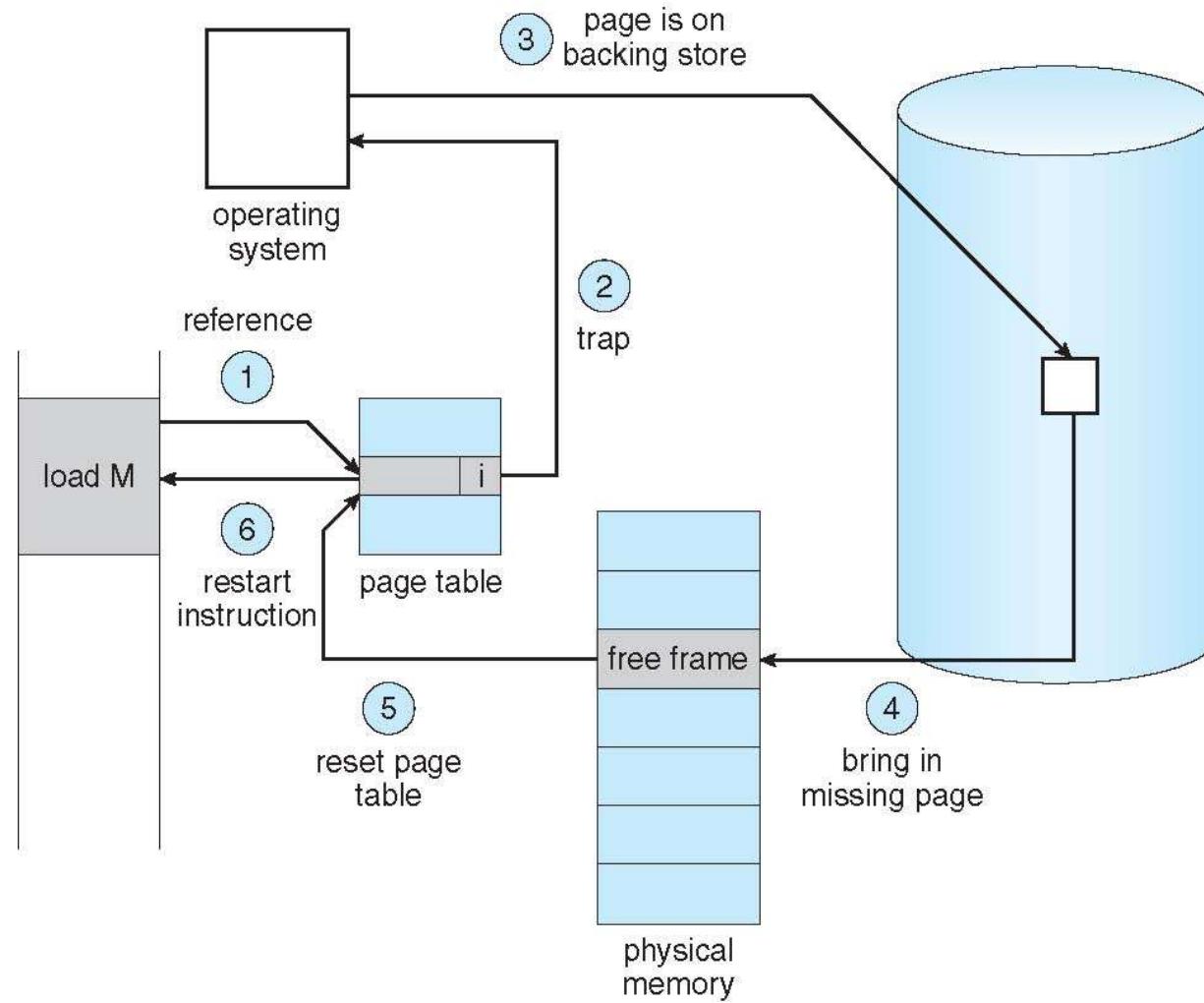
## Page Table When Some Pages Are Not in Main Memory



# 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 an internal table (part of PCB) to decide:
    - Invalid reference  $\Rightarrow$  abort
    - Just not in memory
  2. Find free 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

# Steps in Handling a Page Fault

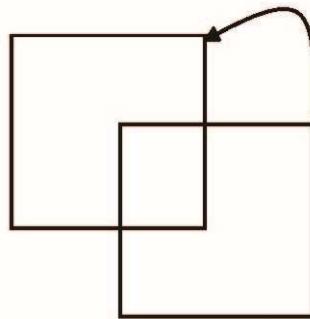


# Aspects of Demand Paging

- 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
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - 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

# Instruction Restart

- Consider an instruction that could access several different locations
  - block move



- auto increment/decrement location
- Restart the whole operation?
  - What if source and destination overlap?

# Performance of Demand Paging

- Stages in Demand Paging

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
  1. Wait in a queue for this device until the read request is serviced
  2. Wait for the device seek and/or latency time
  3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

# Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- 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)

$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + \text{swap page out} \\ & + \text{swap page in }) \end{aligned}$$

# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$   
 $= (1 - p) \times 200 + p \times 8,000,000$   
 $= 200 + p \times 7,999,800$  $EAT = (1 - p) \times \text{memory access}$   
 $+ p (\text{page fault overhead}$   
 $+ \text{swap page out}$   
 $+ \text{swap page in})$
- If one access out of 1,000 causes a page fault, then  
 $EAT = 8.2 \text{ microseconds.}$   
This is a slowdown by a factor of 40!!
- If we want performance degradation to be < 10 percent
  - $220 > 200 + 7,999,800 \times p$   
 $20 > 7,999,800 \times p$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses

# Demand Paging Example

- Memory access time for a system is given as 1 microseconds and the average page fault service time is given as 10 milliseconds. Let  $p=0.001$  then EAT ?

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + \text{swap page out} \\ & + \text{swap page in}) \end{aligned}$$

- Solution:**
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 1 \text{ microsecond} + p (10 \text{ milliseconds}) \\ &= (1 - 0.001) \times 1 \text{ microsecond} + 0.001 \times 10 \\ &\quad \text{milliseconds} \\ &= 0.999 + 0.001 * 10000 \text{ microseconds} \\ &= 0.999 + 10 \text{ microseconds} \\ &= 11 \text{ microseconds(approx)} \end{aligned}$$

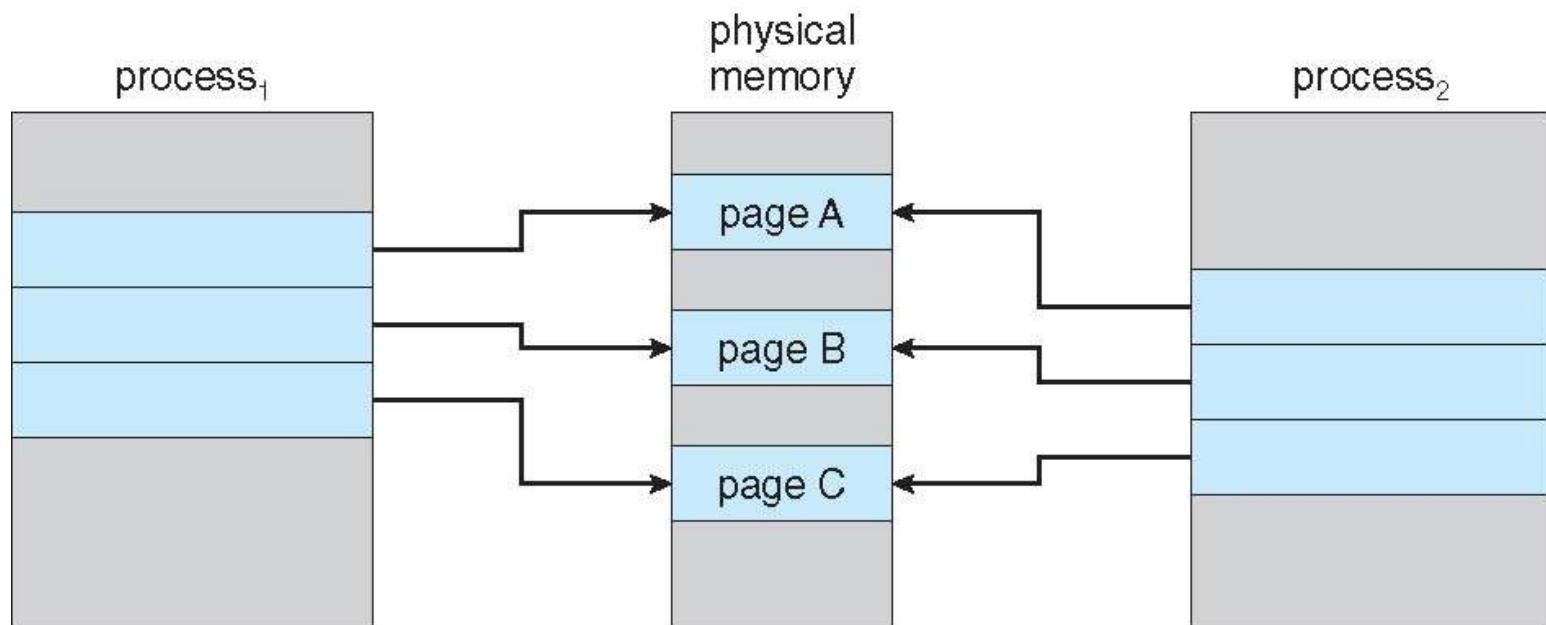
# Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- 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
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

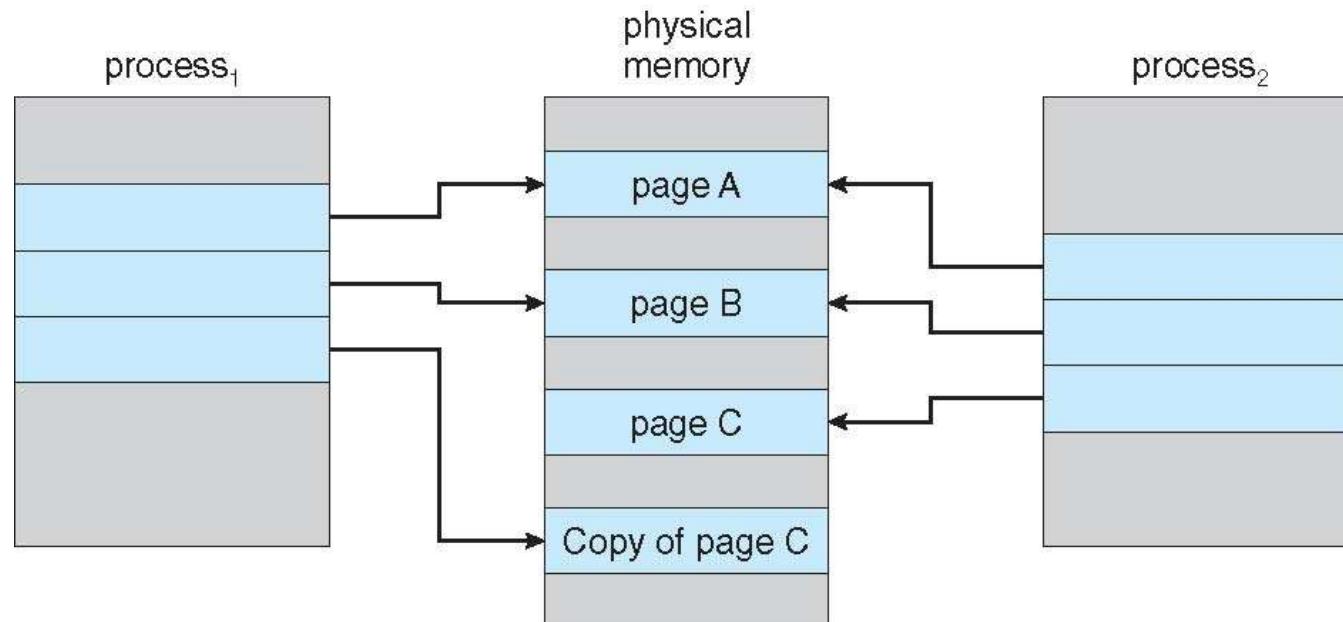
# Copy-on-Write

- 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
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- **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



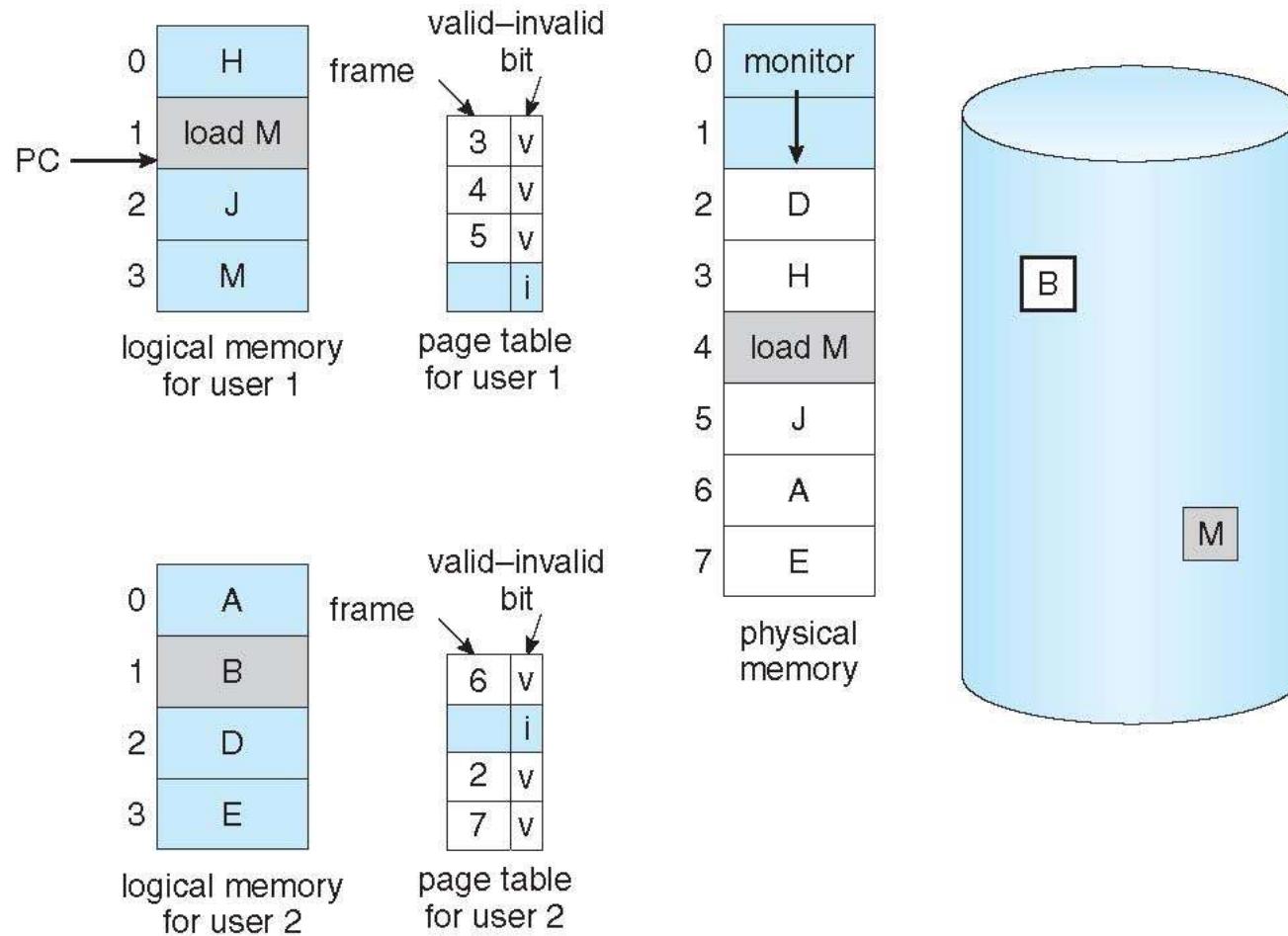
# What Happens if There is no Free Frame?

- 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

# Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

# Need For Page Replacement

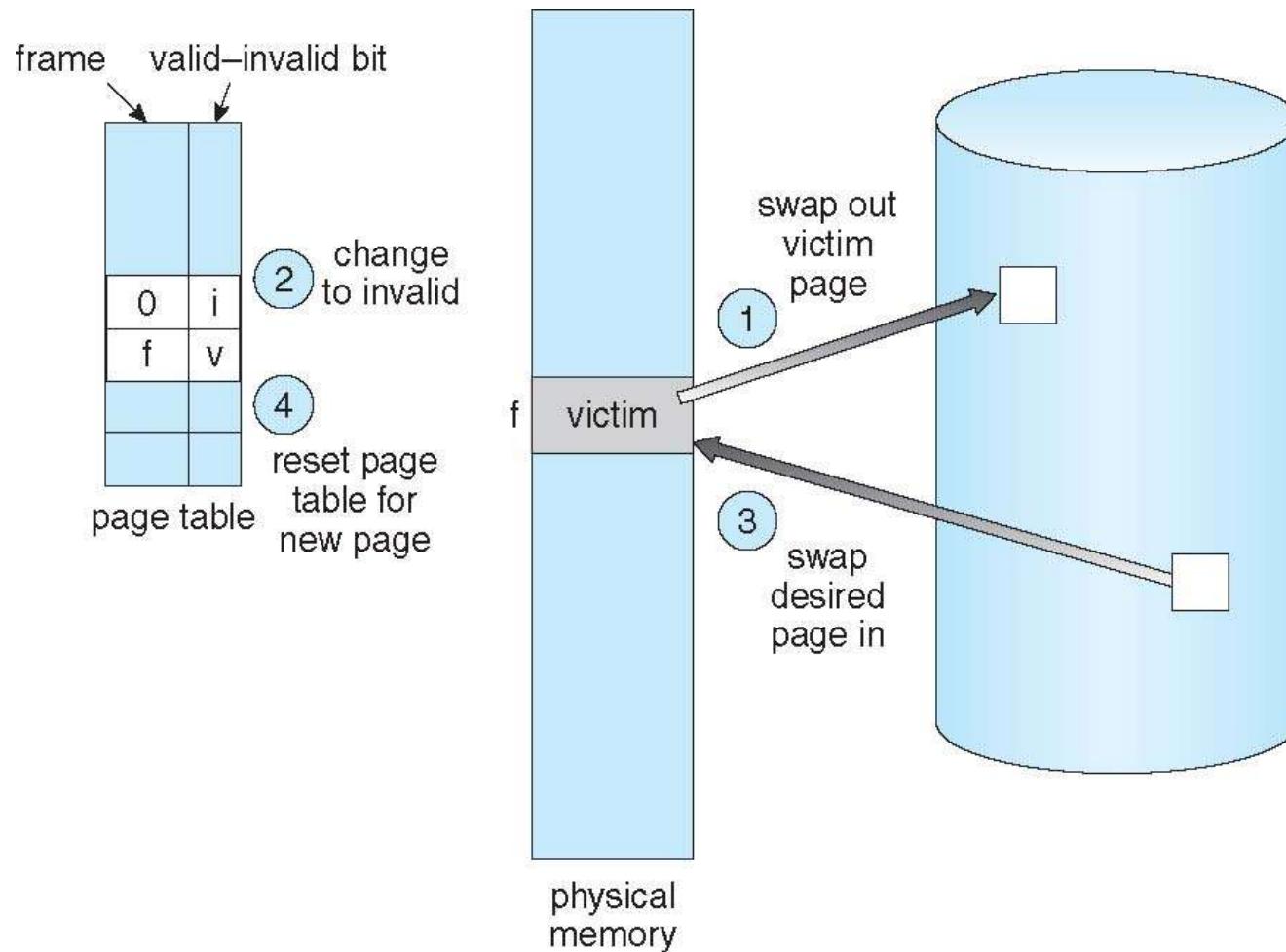


# Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

# Page Replacement

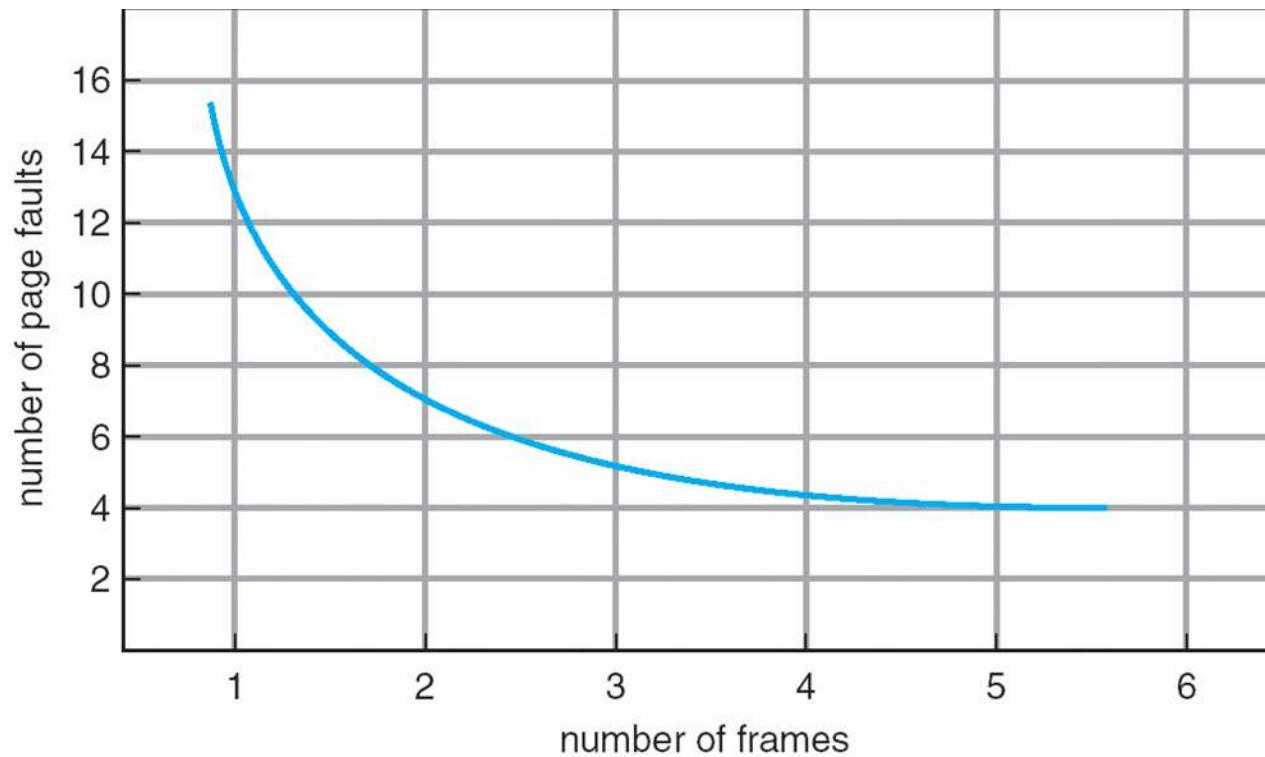


# Page and Frame Replacement Algorithms

- **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
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

## Graph of Page Faults Versus The Number of Frames



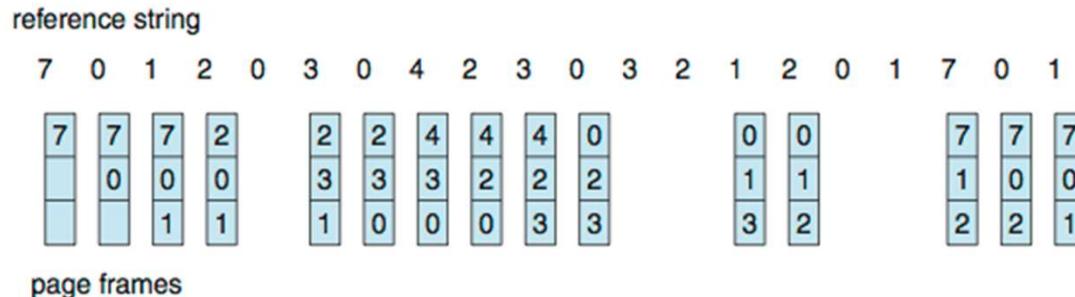
# Page and Frame Replacement Algorithms

- **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
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

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

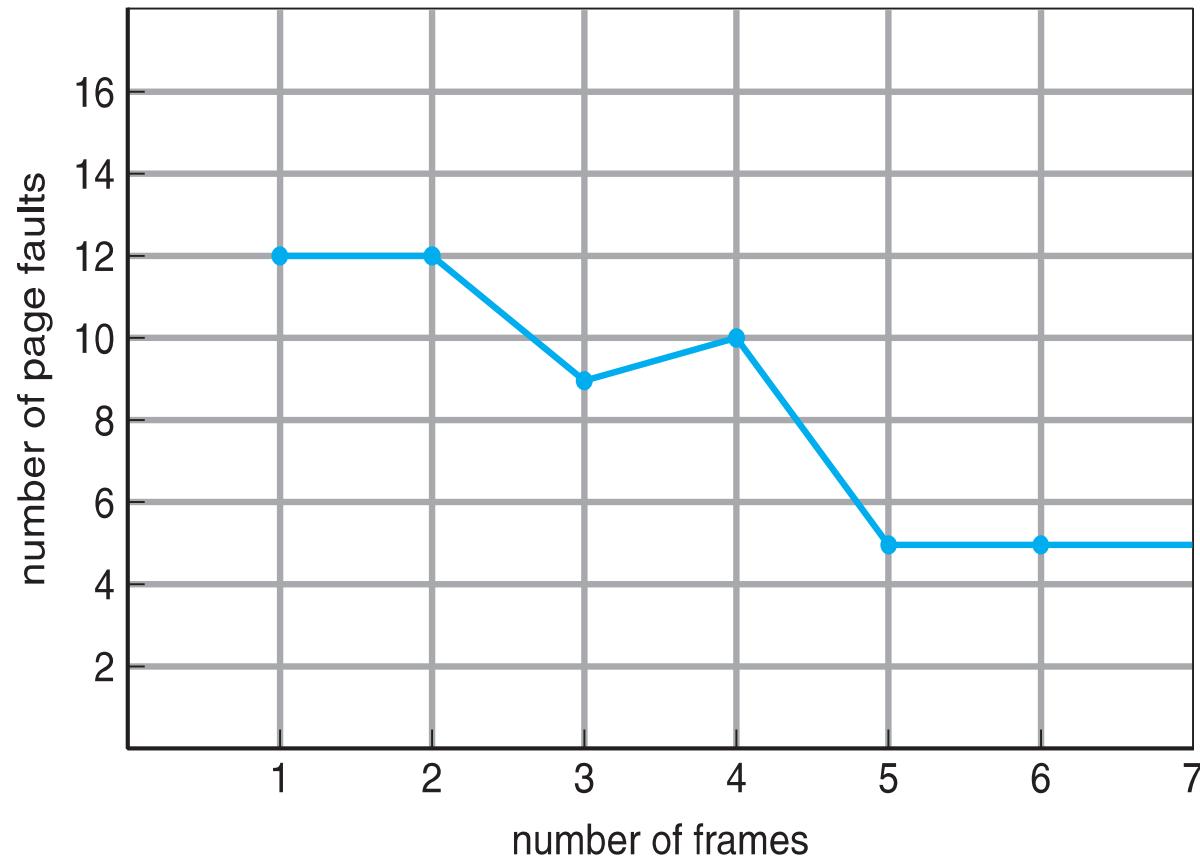
- Reference string:  
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

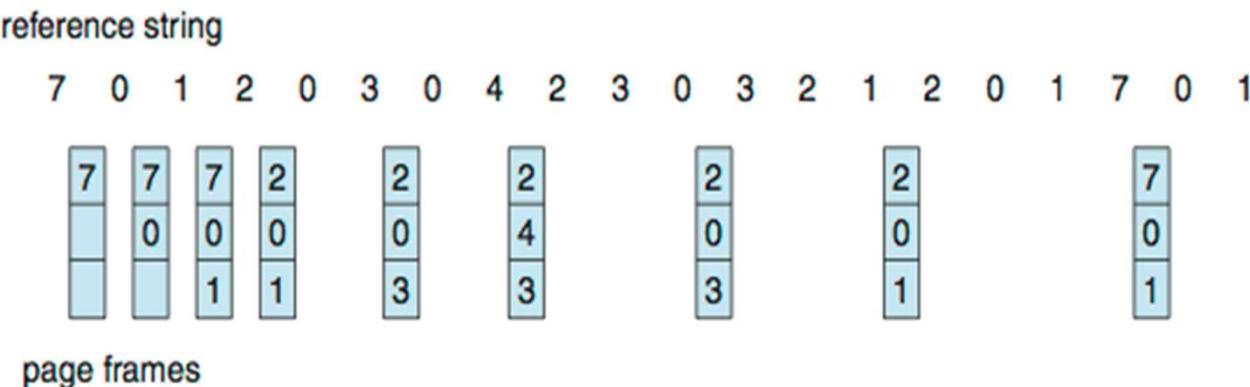
- Can vary by reference string: consider  
1,2,3,4,1,2,5,1,2,3,4,5
  - No of page faults is 9 and 10, given 3 and 4 frames respectively
  - **Adding more frames can cause more page faults!**
    - Belady's Anomaly
- How to track ages of pages?
  - Just use a FIFO queue

# FIFO Illustrating Belady's Anomaly



# Optimal Algorithm

- Replace page that will not be used for the longest period of time
  - 9 page faults is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

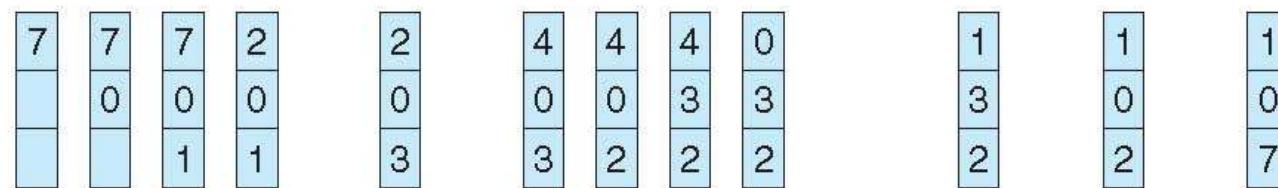


# Least Recently Used (LRU) Algorithm

- 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

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

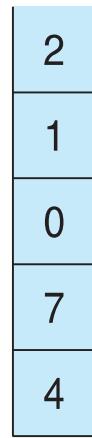
# LRU Algorithm (Cont.)

- 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 and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

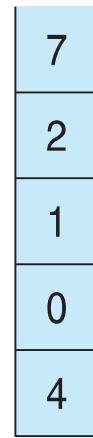
## Use Of A Stack to Record Most Recent Page References

reference string

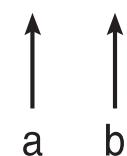
4 7 0 7 1 0 1 2 1 2 7 1 2



stack  
before  
a



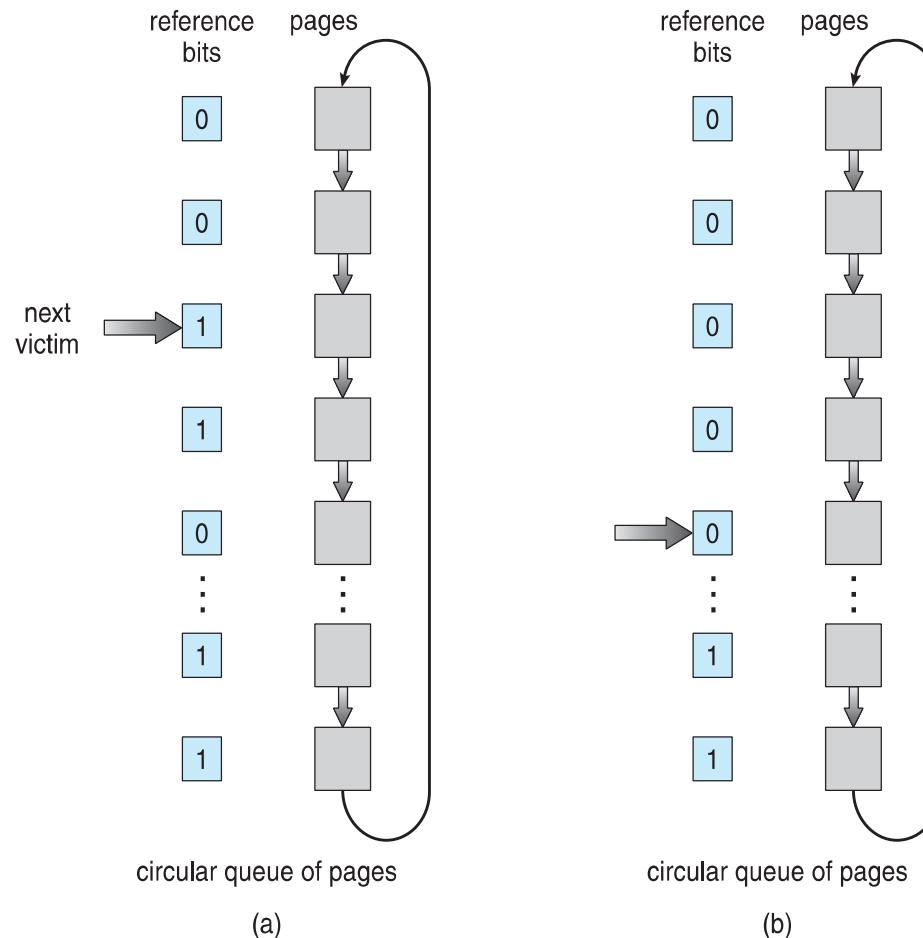
stack  
after  
b



# 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 page 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 (clock) Page-Replacement Algorithm



# Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- **Take ordered pair (reference, modify)**
  1. **(0, 0)** neither recently used nor modified – best page to replace
  2. **(0, 1)** not recently used but modified – not quite as good, must write out before replacement
  3. **(1, 0)** recently used but clean – probably will be used again soon
  4. **(1, 1)** recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used (LFU) Algorithm:** replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and is yet to be used

# Page-Buffering Algorithms

- System keeps a pool of free frames, always
  - When a page fault occurs, a victim frame is chosen as before.
  - Desired page is read into a free frame from the pool before the victim is written out.
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim (written out later)
- Possibly, keep list of modified pages
  - When the **paging device is idle, a modified page is selected and is written to the disk**. Its modify bit is then reset.
- Possibly, keep free frame contents intact and note what is in them
  - Remember which page was in each frame.
  - Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.
  - No I/O is needed in this case.
  - When a page fault occurs, we first check whether the desired page is in the free-frame pool.
  - If it is not, we must select a free frame and read into it.

# Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases (provides its own memory management and I/O buffering.)
- **Memory intensive applications can cause double buffering**
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk for special programs the **ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures.**
  - **The array is general called as Raw disk ( Raw I/O )**
  - Bypasses buffering, locking, demand paging, file name, directories, prefetching,etc ....

# Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** number of frames depends on the total frames in the system
- Two major allocation schemes
  - **fixed allocation**
  - **priority allocation**

# Fixed Allocation

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming and process sizes change
  - $s_i$  = size of process  $p_i$
  - $S = \sum s_i$
  - $m$  = total number of frames
  - $a_i$  = allocation for  $p_i$  =  $\frac{s_i}{S} \times m$
$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

# Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process  $P_i$  generates a page fault,
  - select for replacement of one of its frames
  - select for replacement of a frame from a process with lower priority number

# Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

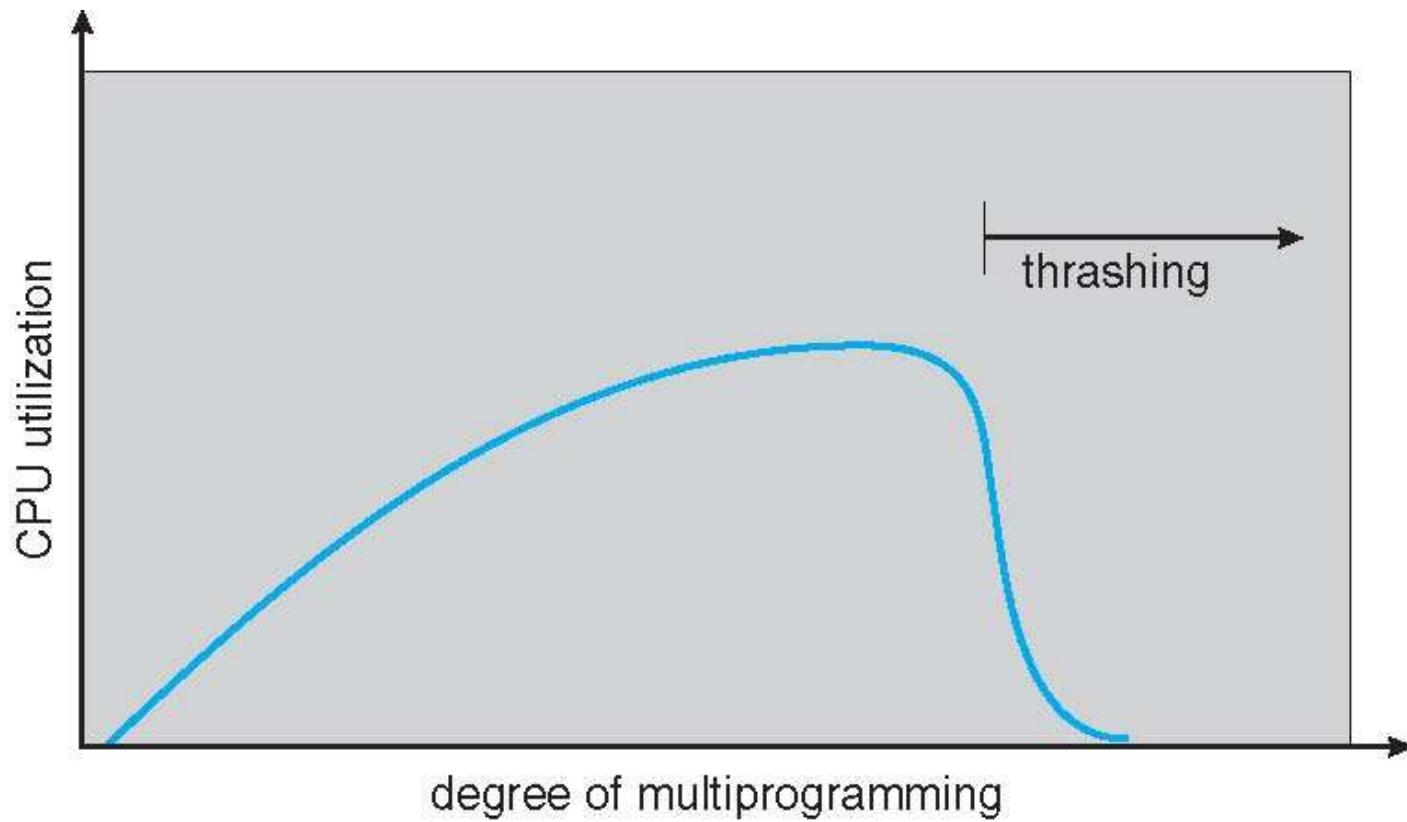
# Non-Uniform Memory Access

- All memory addresses are accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **Igroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the Igroup

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault happens
  - Replace existing frame
  - But one may quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process is added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out
  - **High paging activity is called thrashing.**
  - **A process is thrashing if it is spending more time paging than executing.**

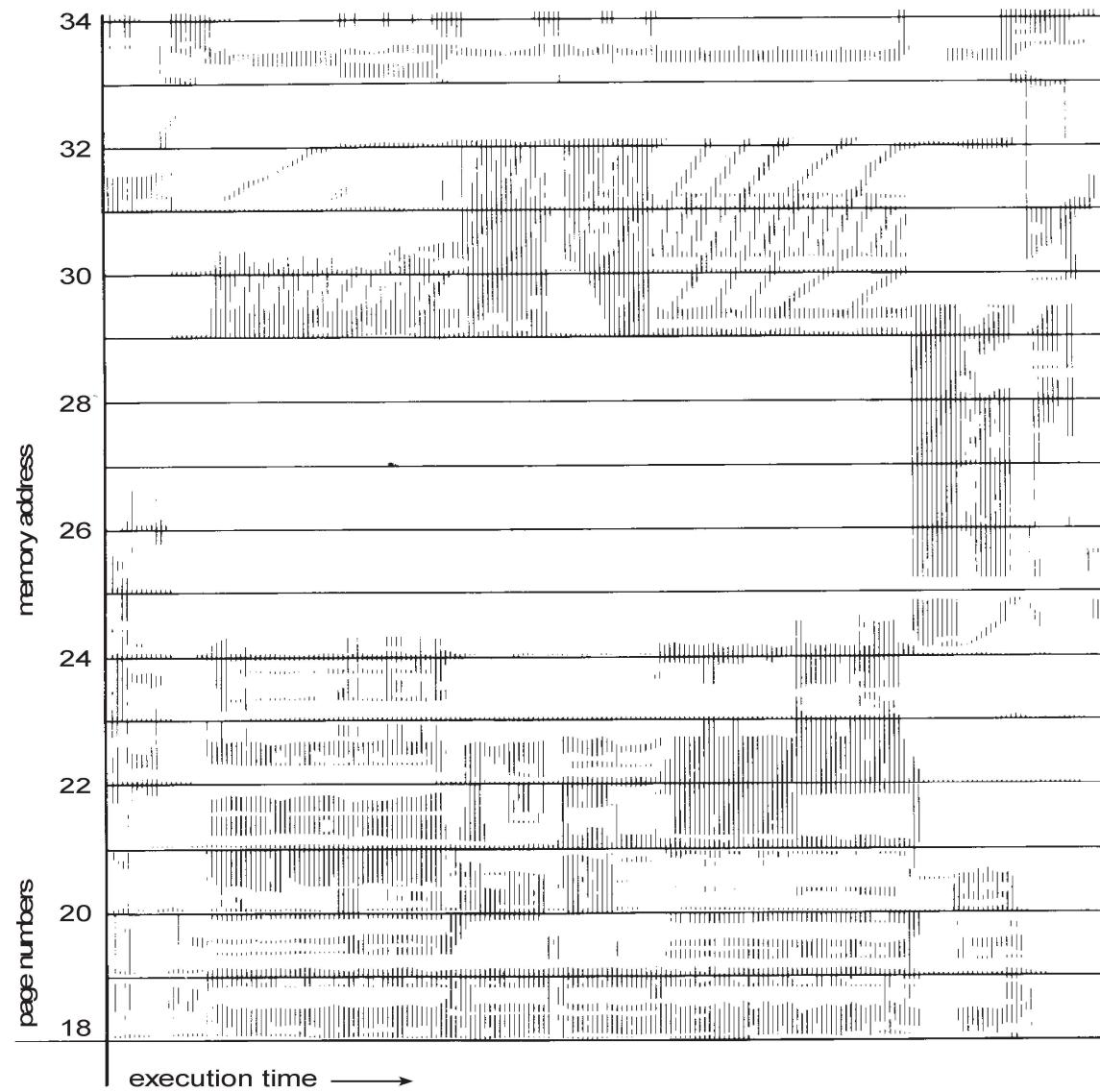
# Thrashing (Cont.)



# Thrashing

- To prevent thrashing, we must provide a process with as many frames as it needs.
  - But how do we know how many frames it “needs”?
  - Solution : Locality model The working-set model
  - Working set strategy starts by looking at how many frames a process is actually using.
  - This approach defines the **locality model** of process execution.
- Locality model states that, as a process executes, it moves from locality to locality.
  - A locality is a set of pages that are actively used together.
  - A program is generally composed of several different localities, which may overlap.
  - **Locality model**
  - Process migrates from one locality to another
- Why does thrashing occur?  
 $\Sigma$  size of locality > total memory size
  - Limit effects by using local or priority page replacement

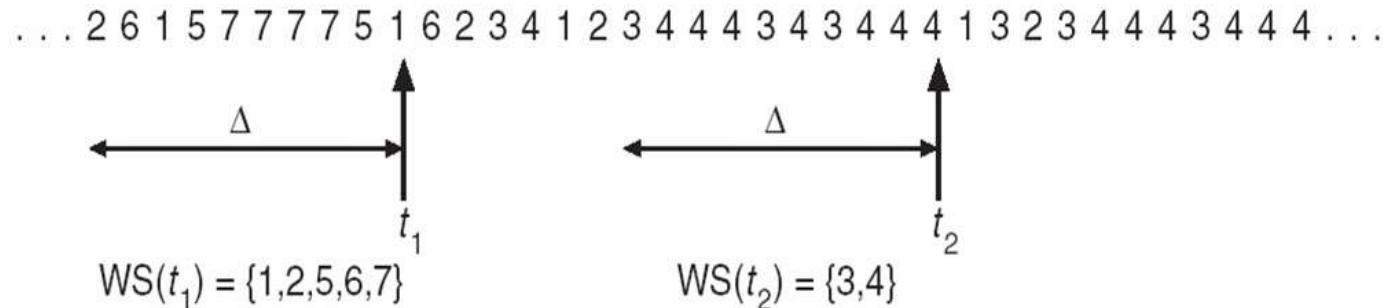
# Locality In A Memory-Reference Pattern



# Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
Example: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) =  
total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality
- if  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

page reference table

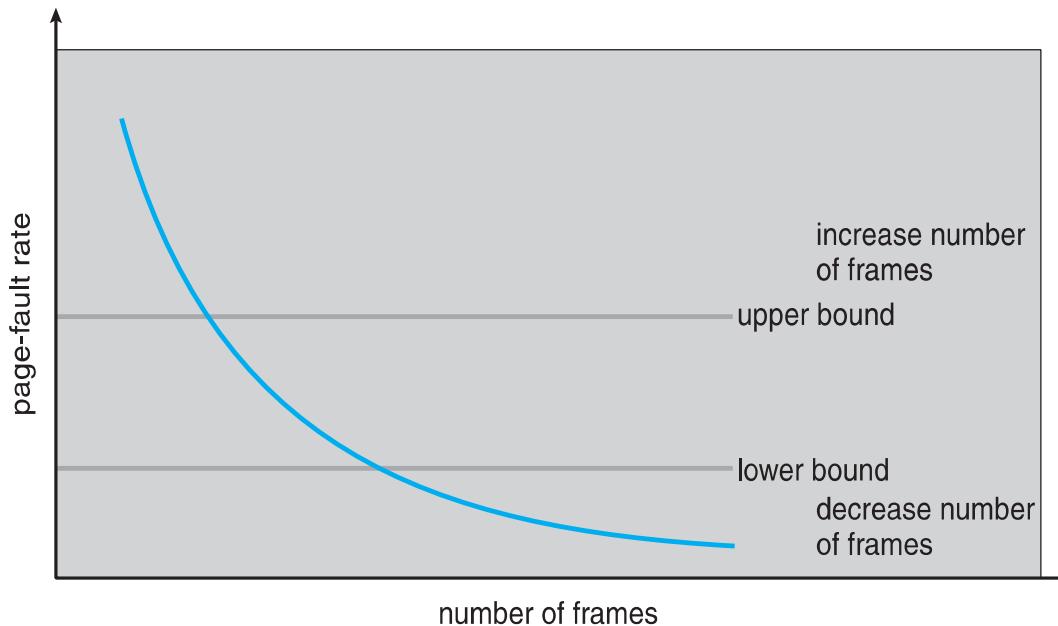


# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupt occurs copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

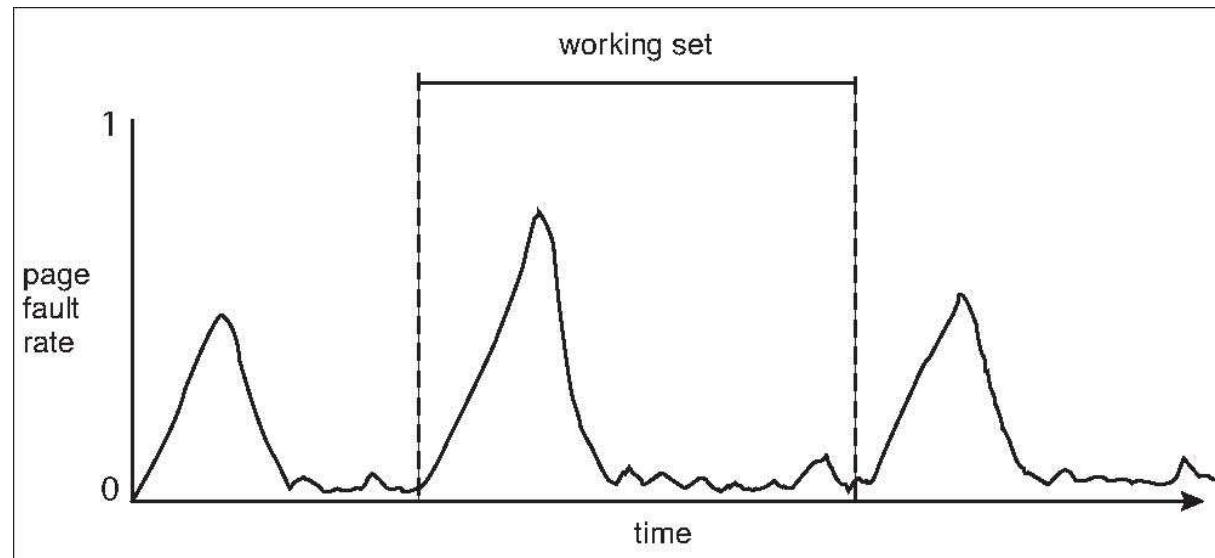
# Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



# Working Sets and Page Fault Rates

- n Direct relationship between working set of a process and its page-fault rate
- n Working set changes over time
- n Peaks and valleys over time



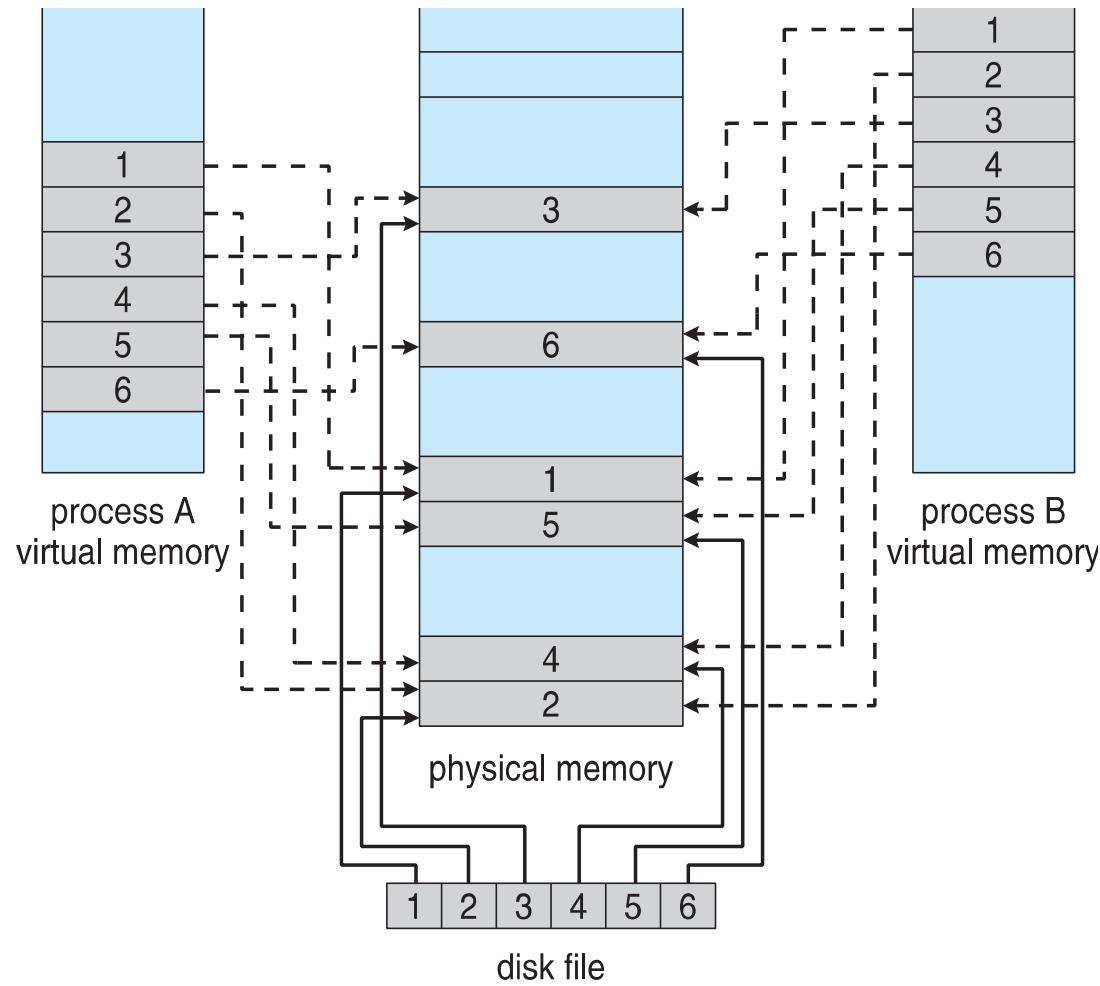
# Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

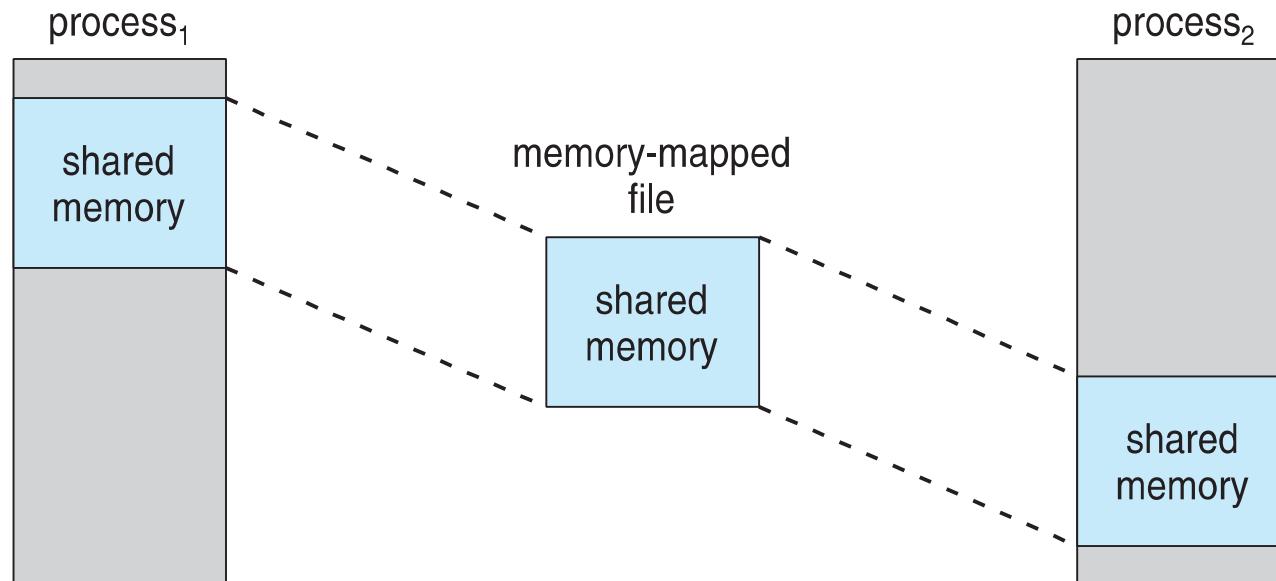
# Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
  - But map file into kernel address space
  - Process still does `read()` and `write()`
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

# Memory Mapped Files



# Shared Memory via Memory-Mapped I/O



# Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
  - Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
  - Producer create shared-memory object using memory mapping features
  - Open file via `CreateFile()`, returning a HANDLE
  - Create mapping via `CreateFileMapping()` creating a **named shared-memory object**
  - Create view via `MapViewOfFile()`
- Sample code in Textbook

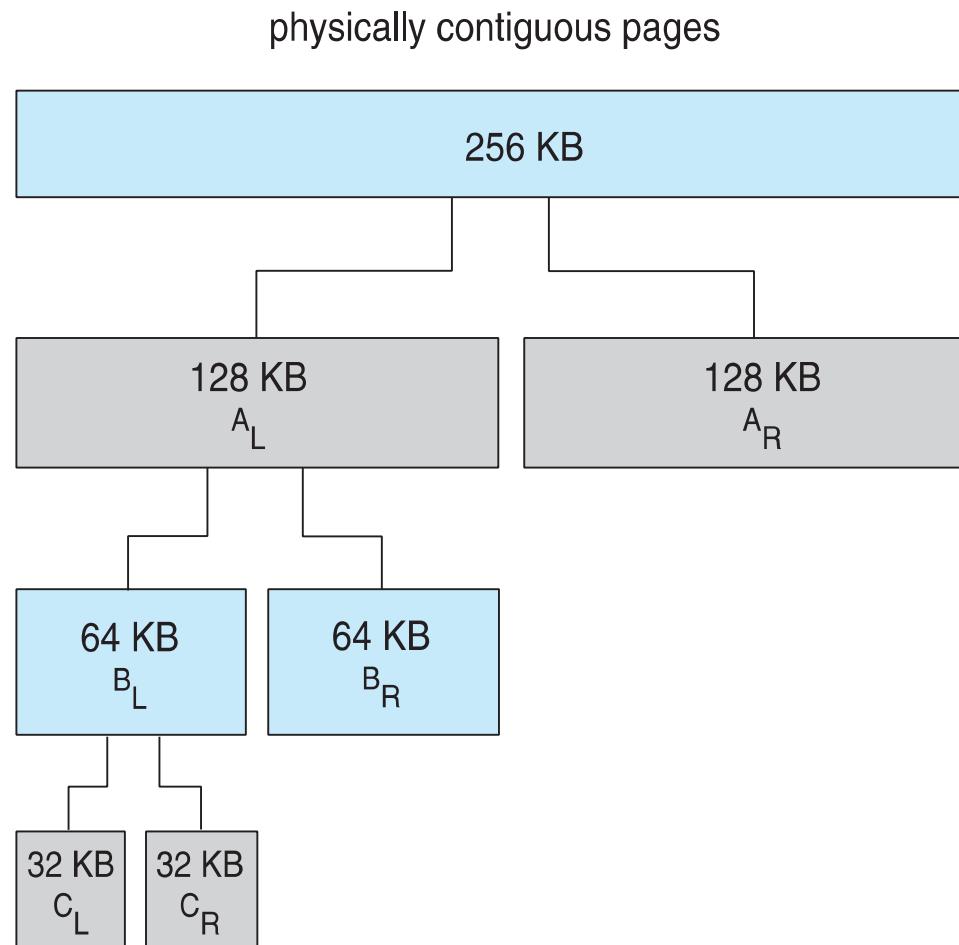
# Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

# Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into  $A_L$  and  $A_R$  of 128KB each
    - One further divided into  $B_L$  and  $B_R$  of 64KB
      - One further into  $C_L$  and  $C_R$  of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

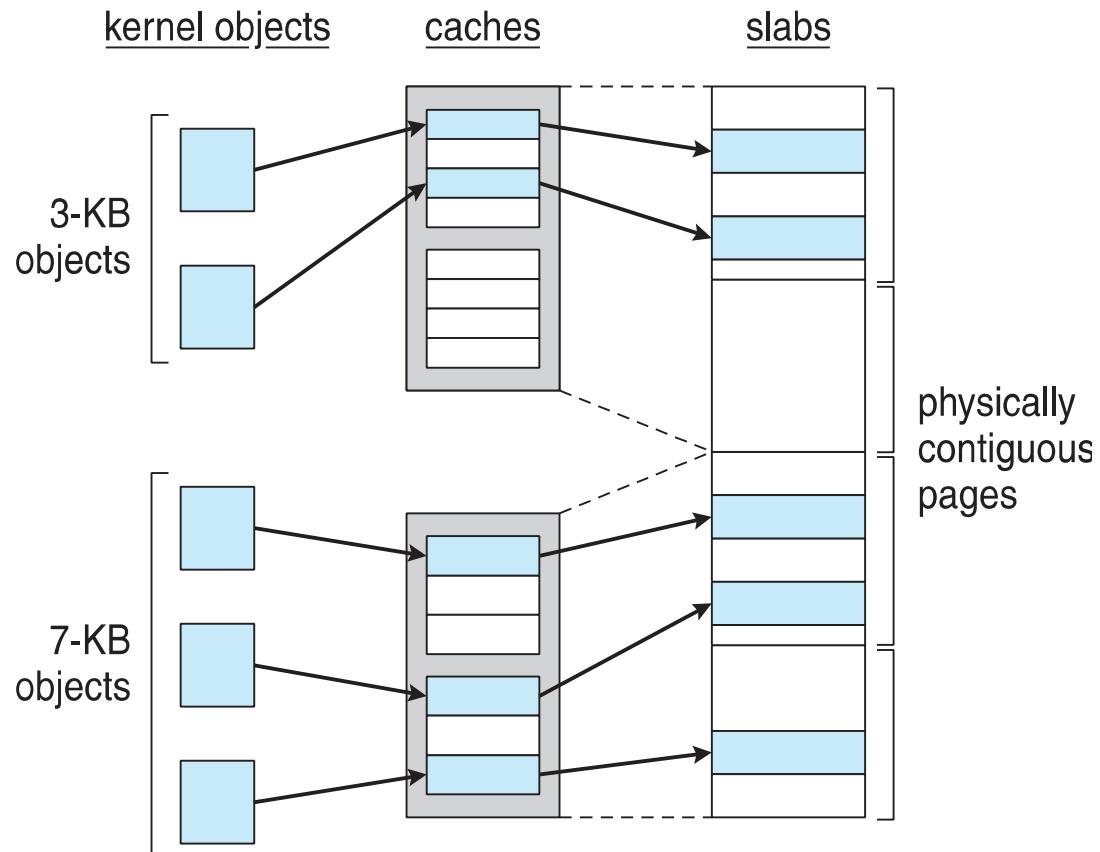
# Buddy System Allocator



# Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

# Slab Allocation



# Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

# Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

# Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume  $s$  pages are prepaged and  $\alpha$  of the pages is used
    - Is cost of  $s * \alpha$  save pages faults > or < than the cost of prepaging  $s * (1 - \alpha)$  unnecessary pages?
    - $\alpha$  near zero  $\Rightarrow$  prepaging loses

# Other Issues – Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range  $2^{12}$  (4,096 bytes) to  $2^{22}$  (4,194,304 bytes)
- On average, growing over time

# Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

# Other Issues – Program Structure

- Program structure

- int[128,128] data;
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128;
i++)
        data[i, j] = 0;
```

128 x 128 = 16,384 page faults

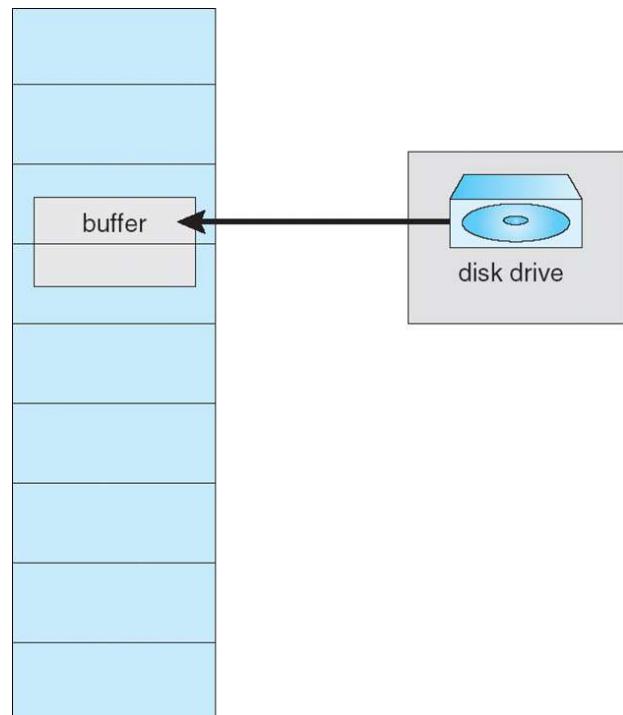
- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

128 page faults

# Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



# Operating System Examples

- Windows
- Solaris

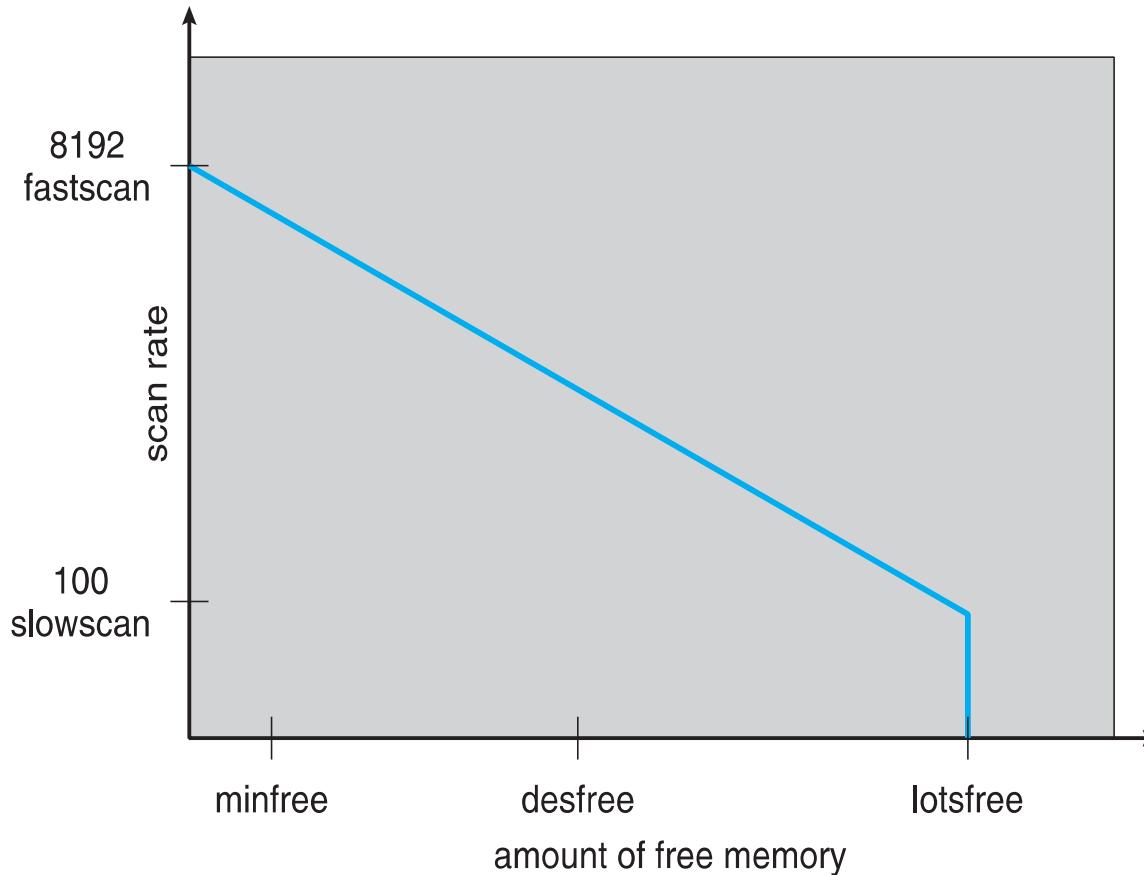
# Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

# Solaris

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to begin swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

# Solaris 2 Page Scanner



# Page Replacement Algorithms in Operating Systems

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

**Page Fault** – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

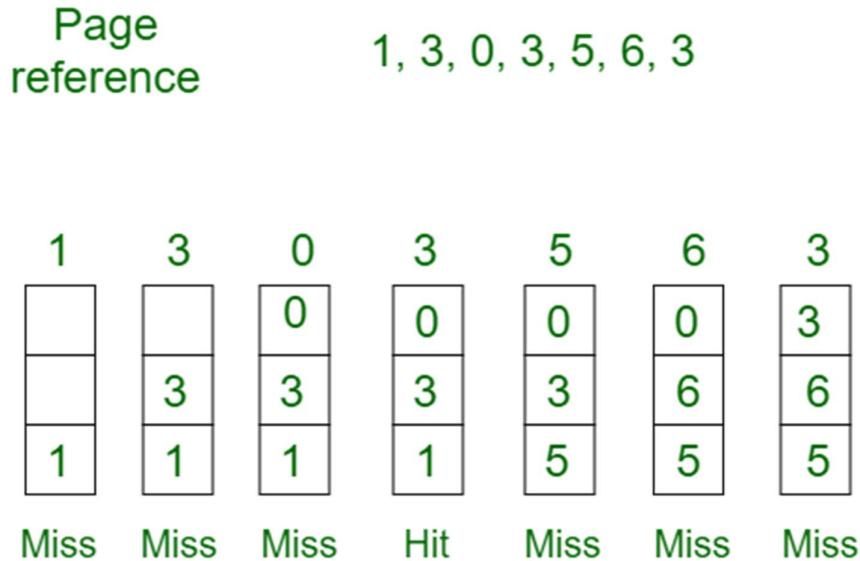
Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

## Page Replacement Algorithms:

- **First In First Out (FIFO)** –

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

**Example-1** Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames. Find number of page faults.



### Total Page Fault = 6

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots  $\rightarrow$  **3 Page Faults**.  
when 3 comes, it is already in memory so  $\rightarrow$  **0 Page Faults**.

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1.  $\rightarrow$  **1 Page Fault**.

6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3  $\rightarrow$  **1 Page Fault**.

Finally when 3 comes it is not available so it replaces 0  $\rightarrow$  **1 page fault**

**Belady's anomaly** – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

- **Optimal Page replacement –**

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

**Example-2:** Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Page reference    7,0,1,2,0,3,0,4,2,3,0,3,2,3    No. of Page frame - 4

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 7 | 7 | 0 | 1 | 0 | 1 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

Miss    Miss    Miss    Miss    Hit    Miss    Hit    Miss    Hit    Hit    Hit    Hit    Hit    Hit

**Total Page Fault = 6**

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → 4 Page faults

0 is already there so → 0 Page fault.

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. → 1 Page fault.

0 is already there so → 0 Page fault..

4 will takes place of 1 → 1 Page Fault.

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

- **Least Recently Used –**

In this algorithm page will be replaced which is least recently used.

**Example-3** Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.

| Page reference       | 7,0,1,2,0,3,0,4,2,3,0,3,2,3 |      |      |     |      |     |      |     |     |     |     |     |     |     |     | No. of Page frame - 4 |
|----------------------|-----------------------------|------|------|-----|------|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----------------------|
| 7                    | 0                           | 1    | 2    | 0   | 3    | 0   | 4    | 2   | 3   | 0   | 3   | 2   | 3   | 0   | 3   | 2                     |
|                      |                             |      |      |     |      |     |      |     |     |     |     |     |     |     |     |                       |
|                      | 0                           | 0    | 1    | 0   | 0    | 1   | 4    | 2   | 4   | 2   | 4   | 2   | 4   | 2   | 4   | 2                     |
| 7                    | 7                           | 7    | 7    | 7   | 3    | 3   | 3    | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3                     |
| Miss                 | Miss                        | Miss | Miss | Hit | Miss | Hit | Miss | Hit                   |
| Total Page Fault = 6 |                             |      |      |     |      |     |      |     |     |     |     |     |     |     |     |                       |

Here LRU has same number of page fault as optimal but it may differ according to question.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots  $\rightarrow$  **4 Page faults**

0 is already there so  $\rightarrow$  **0 Page fault**.

when 3 came it will take the place of 7 because it is least recently used  $\rightarrow$  **1 Page fault**

0 is already in memory so  $\rightarrow$  **0 Page fault**.

4 will take place of 1  $\rightarrow$  **1 Page Fault**

Now for the further page reference string  $\rightarrow$  **0 Page fault** because they are already available in the memory.

## Belady's Anomaly in Page Replacement Algorithms

In Operating System, process data is loaded in fixed sized chunks and each chunk is referred to as a page. The processor loads these pages in the fixed sized chunks of memory called frames. Typically the size of each page is always equal to the frame size.

A page fault occurs when a page is not found in the memory, and needs to be loaded from the disk. If a page fault occurs and all memory frames have been already allocated, then replacement of a page in memory is required on the request of a new page. This is referred to as demand-paging. The choice of which page to replace is specified by a page replacement algorithms. The commonly used page replacement algorithms are FIFO, LRU, optimal page replacement algorithms etc.

Generally, on increasing the number of frames to a process' virtual memory, its execution becomes faster as less number of page faults occur. Sometimes the reverse happens, i.e. more number of page faults occur when more frames are allocated to a process. This most unexpected result is termed as **Belady's Anomaly**.

**Bélády's anomaly** is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

This phenomenon is commonly experienced in following page replacement algorithms:

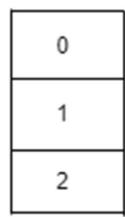
1. First in first out (FIFO)
2. Second chance algorithm
3. Random page replacement algorithm

### **Reason of Belady's Anomaly –**

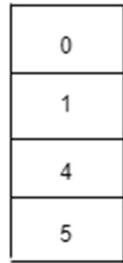
The other two commonly used page replacement algorithms are Optimal and LRU, but Belady's Anomaly can never occur in these algorithms for any reference string as they belong to a class of stack based page replacement algorithms.

A **stack based algorithm** is one for which it can be shown that the set of pages in memory for  $N$  frames is always a subset of the set of pages that would be in memory with  $N + 1$  frames. For LRU replacement, the set of pages in memory would be the  $n$  most recently referenced pages. If the number of frames increases then these  $n$  pages will still be the most recently referenced and so, will still be in the memory. While in FIFO, if a page named  $b$  came into physical memory before a page –  $a$  then priority of replacement of  $b$  is greater than that of  $a$ , but this is not independent of the number of page frames and hence, FIFO does not follow a stack page replacement policy and therefore suffers Belady's Anomaly.

**Example:** Consider the following diagram to understand the behaviour of a stack-based page replacement algorithm



Number of frames = 3



Number of frames = 4

As the set of pages in memory with 4 frames is not a subset of memory with 3 frames, the property of stack algorithm fails in FIFO.

The diagram illustrates that given the set of pages i.e.  $\{0, 1, 2\}$  in 3 frames of memory is not a subset of the pages in memory –  $\{0, 1, 4, 5\}$  with 4 frames and it is a violation in the property of stack based algorithms. This situation can be frequently seen in FIFO algorithm.

### **Belady's Anomaly in FIFO –**

Assuming a system that has no pages loaded in the memory and uses the FIFO Page replacement algorithm. Consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

**Case-1:** If the system has 3 frames, the given reference string on using FIFO page replacement algorithm yields a total of 9 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

|    |    |    |    |    |    |    |   |   |    |    |   |
|----|----|----|----|----|----|----|---|---|----|----|---|
| 1  | 1  | 1  | 2  | 3  | 4  | 1  | 1 | 1 | 2  | 5  | 5 |
|    | 2  | 2  | 3  | 4  | 1  | 2  | 2 | 2 | 5  | 3  | 3 |
|    |    | 3  | 4  | 1  | 2  | 5  | 5 | 5 | 3  | 4  | 4 |
| PF | X | X | PF | PF | X |

**Case-2:** If the system has 4 frames, the given reference string on using FIFO page replacement algorithm yields a total of 10 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

|    |    |    |    |   |   |    |    |    |    |    |    |
|----|----|----|----|---|---|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1 | 1 | 2  | 3  | 4  | 5  | 1  | 2  |
|    | 2  | 2  | 2  | 2 | 2 | 3  | 4  | 5  | 1  | 2  | 3  |
|    |    | 3  | 3  | 3 | 3 | 4  | 5  | 1  | 2  | 3  | 4  |
|    |    |    | 4  | 4 | 4 | 5  | 1  | 2  | 3  | 4  | 5  |
| PF | PF | PF | PF | X | X | PF | PF | PF | PF | PF | PF |

It can be seen from the above example that on increasing the number of frames while using the FIFO page replacement algorithm, the number of **page faults increased** from 9 to 10.

**Note –** It is not necessary that every string reference pattern cause Belady anomaly in FIFO but there are certain kind of string references that worsen the FIFO performance on increasing the number of frames.

#### Why Stack based algorithms do not suffer Anomaly –

All the stack based algorithms never suffer Belady Anomaly because these type of algorithms assigns a priority to a page (for replacement) that is independent of the number of page frames. Examples of such policies are Optimal, LRU and LFU. Additionally these algorithms also have a good property for simulation, i.e. the miss (or hit) ratio can be computed for any number of page frames with a single pass through the reference string.

In LRU algorithm every time a page is referenced it is moved at the top of the stack, so, the top  $n$  pages of the stack are the  $n$  most recently used pages. Even if the number of frames are incremented to  $n+1$ , top of the stack will have  $n+1$  most recently used pages.

Similar example can be used to calculate the number of page faults in LRU algorithm. Assuming a system that has no pages loaded in the memory and uses the LRU Page replacement algorithm. Consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

**Case-1:** If the system has 3 frames, the given reference string on using LRU page replacement algorithm yields a total of 10 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

|    |    |    |    |    |    |    |   |   |    |    |    |
|----|----|----|----|----|----|----|---|---|----|----|----|
| 1  | 2  | 3  | 4  | 1  | 2  | 5  | 1 | 2 | 3  | 4  | 5  |
| 1  | 2  | 3  | 4  | 1  | 2  | 5  | 1 | 2 | 3  | 4  |    |
|    | 1  | 2  | 3  | 4  | 1  | 2  | 5 | 1 | 2  | 3  |    |
| PF | X | X | PF | PF | PF |

**Case-2:** If the system has 4 frames, the given reference string on using LRU page replacement algorithm, then total 8 page faults occur. The diagram shows the pattern of the page faults in the example.

|    |    |    |    |   |   |    |   |   |    |    |    |
|----|----|----|----|---|---|----|---|---|----|----|----|
| 1  | 2  | 3  | 4  | 1 | 2 | 5  | 1 | 2 | 3  | 4  | 5  |
| 1  | 2  | 3  | 4  | 1 | 2 | 5  | 1 | 2 | 3  | 4  |    |
|    | 1  | 2  | 3  | 4 | 1 | 2  | 5 | 1 | 2  | 3  |    |
|    |    | 1  | 2  | 3 | 4 | 1  | 2 | 5 | 1  | 2  |    |
| PF | PF | PF | PF | X | X | PF | X | X | PF | PF | PF |

## Practise Problems:

### Problem-01:

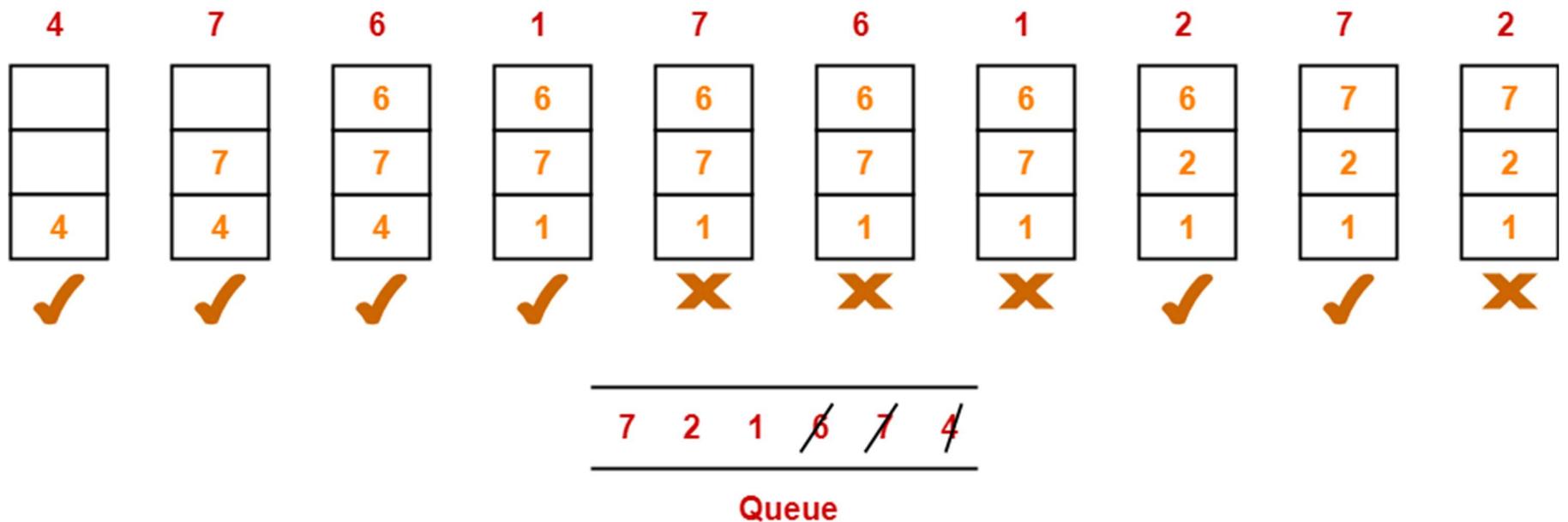
A system uses 3 page frames for storing process pages in main memory. It uses the First in First out (FIFO) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

### Solution-

Total number of references = 10



Total number of page faults occurred = 6

Calculating Hit ratio-

Total number of page hits

= Total number of references – Total number of page misses or page faults

=  $10 - 6$

= 4

Thus, Hit ratio

= Total number of page hits / Total number of references

=  $4 / 10$

= 0.4 or 40%

#### Calculating Miss ratio-

Total number of page misses or page faults = 6

Thus, Miss ratio

= Total number of page misses / Total number of references

=  $6 / 10$

= 0.6 or 60%

**Alternatively,**

Miss ratio

$$= 1 - \text{Hit ratio}$$

$$= 1 - 0.4$$

$$= 0.6 \text{ or } 60\%$$

Problem-02:

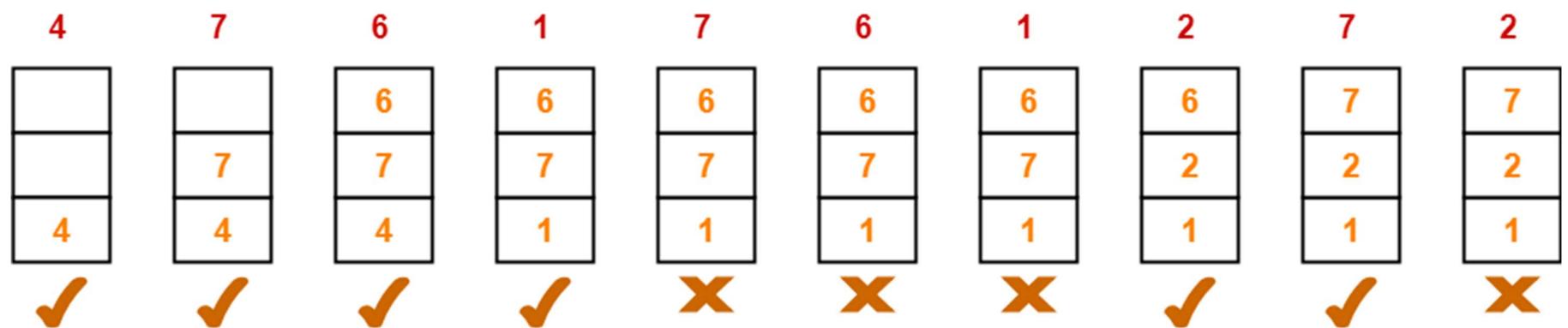
A system uses 3 page frames for storing process pages in main memory. It uses the Least Recently Used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

Solution-

Total number of references = 10



Total number of page faults occurred = 6

In the similar manner as above-

- Hit ratio = 0.4 or 40%
- Miss ratio = 0.6 or 60%

Problem-03:

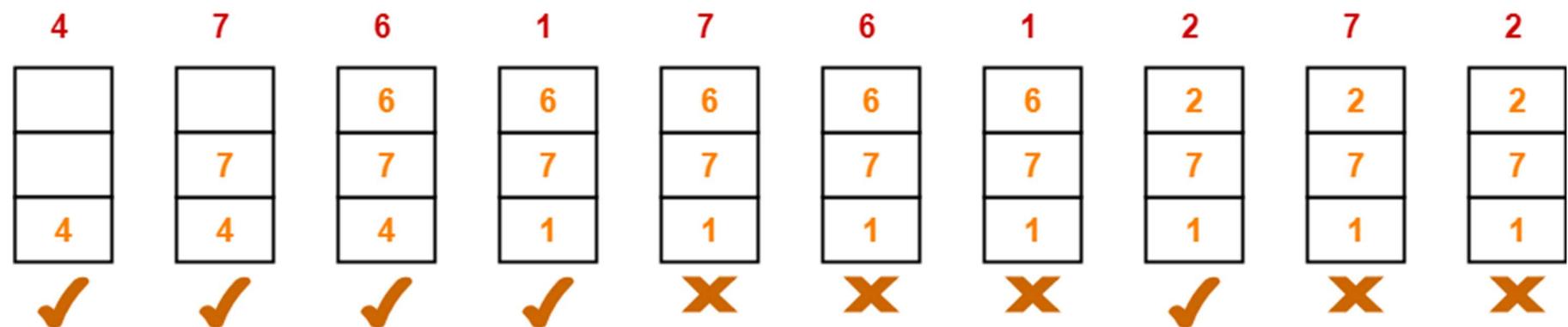
A system uses 3 page frames for storing process pages in main memory. It uses the Optimal page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

Solution-

Total number of references = 10



Total number of page faults occurred = 5

In the similar manner as above-

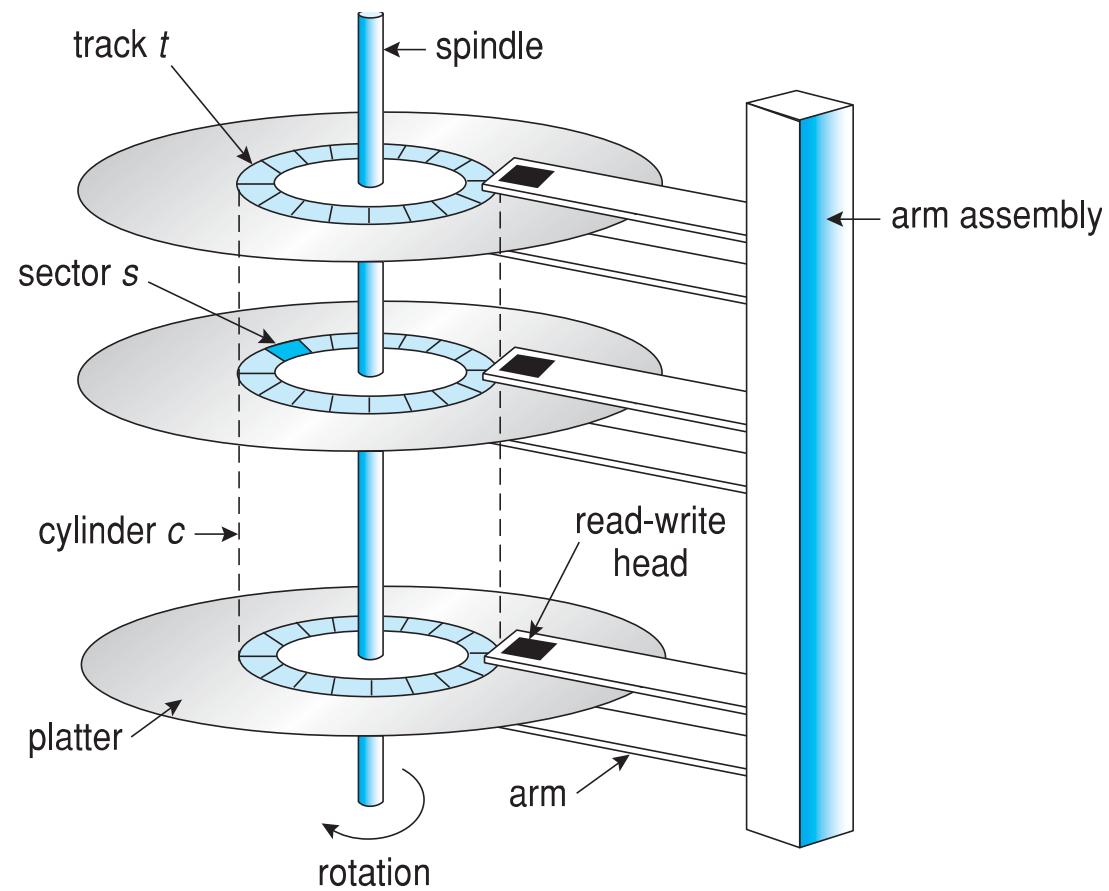
- Hit ratio = 0.5 or 50%
- Miss ratio = 0.5 or 50%

# Mass Storage Systems

# Overview of Mass Storage Structure

- **Magnetic disks** provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is the time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
- Disks can be removable
- Drive attached to computer via **I/O bus**

# Moving-head Disk Mechanism



# Hard Disks

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
  - Average latency =  $\frac{1}{2}$  latency

| Spindle [rpm] | Average latency [ms] |
|---------------|----------------------|
| 4200          | 7.14                 |
| 5400          | 5.56                 |
| 7200          | 4.17                 |
| 10000         | 3                    |
| 15000         | 2                    |

(From Wikipedia)

# The First Commercial Disk Drive



1956  
IBM RAMDAC computer  
included the IBM Model  
350 disk storage system

5M (7 bit) characters  
50 x 24" platters  
Access time = < 1 second

# Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address mapping should be easy
    - Except for bad sectors

# Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time  $\approx$  seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

# Disk Scheduling (Cont.)

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes **input or output mode, disk address, memory address, number of sectors to transfer**
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk must queue the requests
  - Optimization algorithms only make sense when a queue exists

# Disk Scheduling (Cont.)

- Drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- We illustrate scheduling algorithms with a request queue as given below(0-199)

98, 183, 37, 122, 14, 124, 65, 67

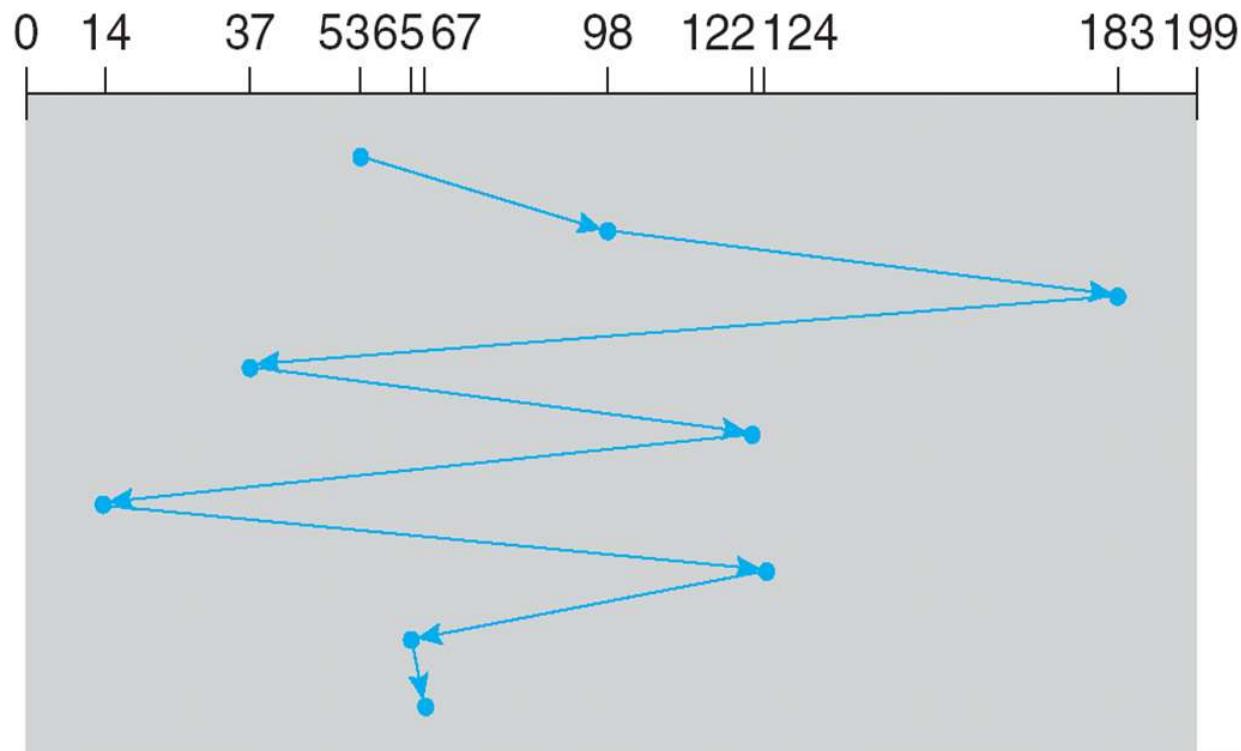
Head pointer 53

# FCFS

Illustration shows total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

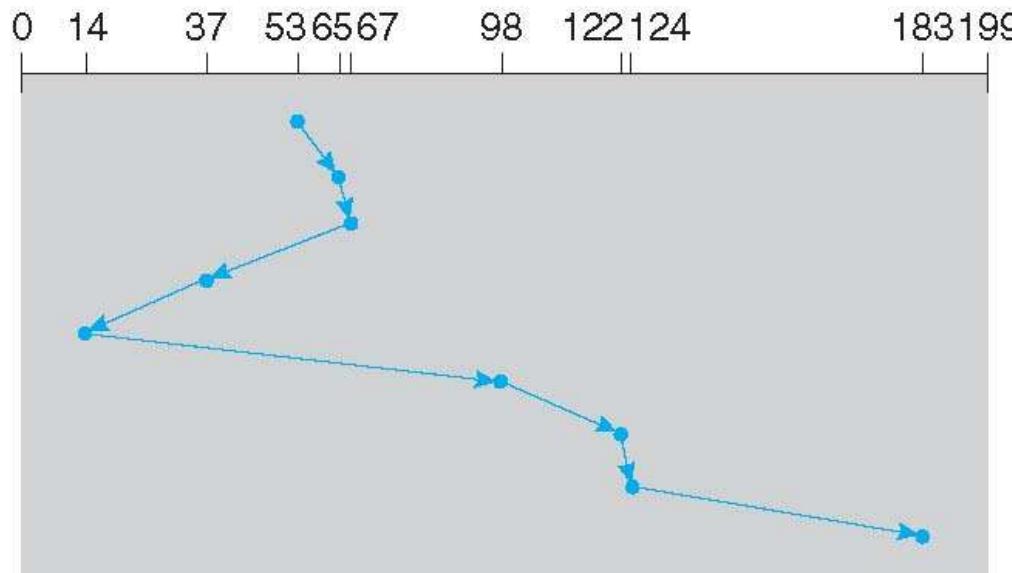


# SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



# SCAN

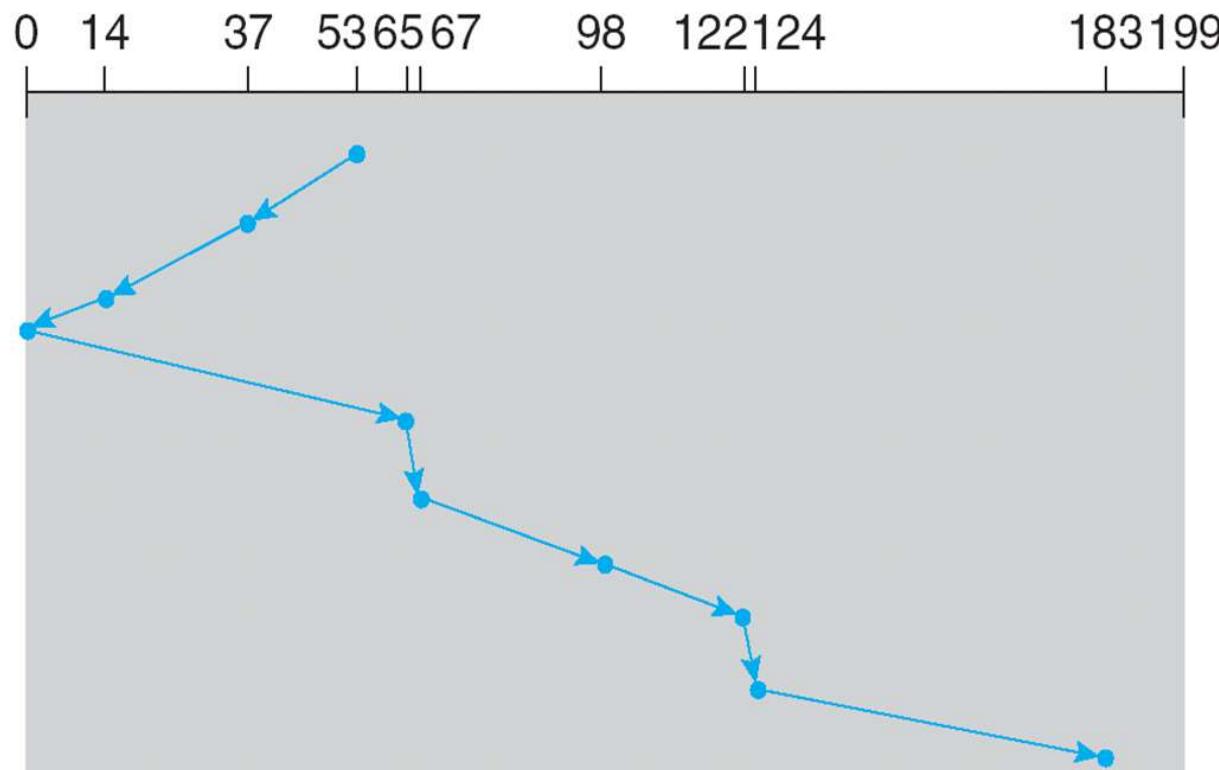
- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** is sometimes called the **elevator algorithm**

# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



# C-SCAN

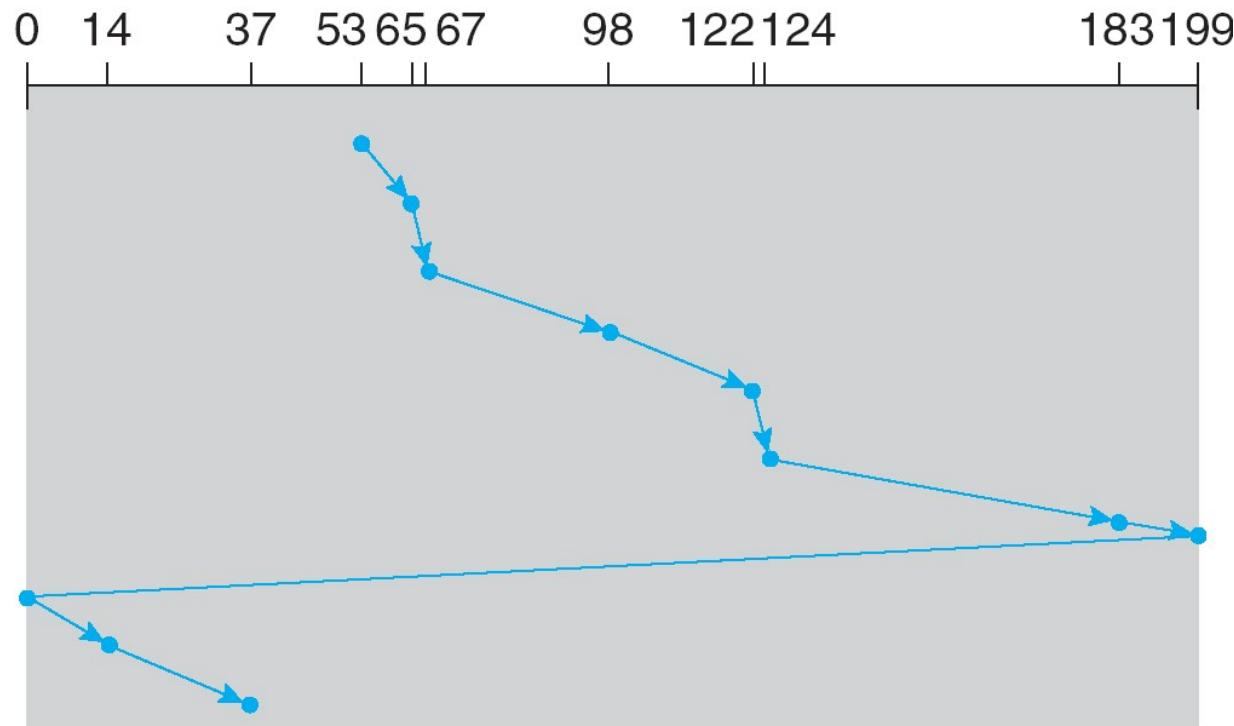
- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

# C-SCAN

- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



# C-LOOK

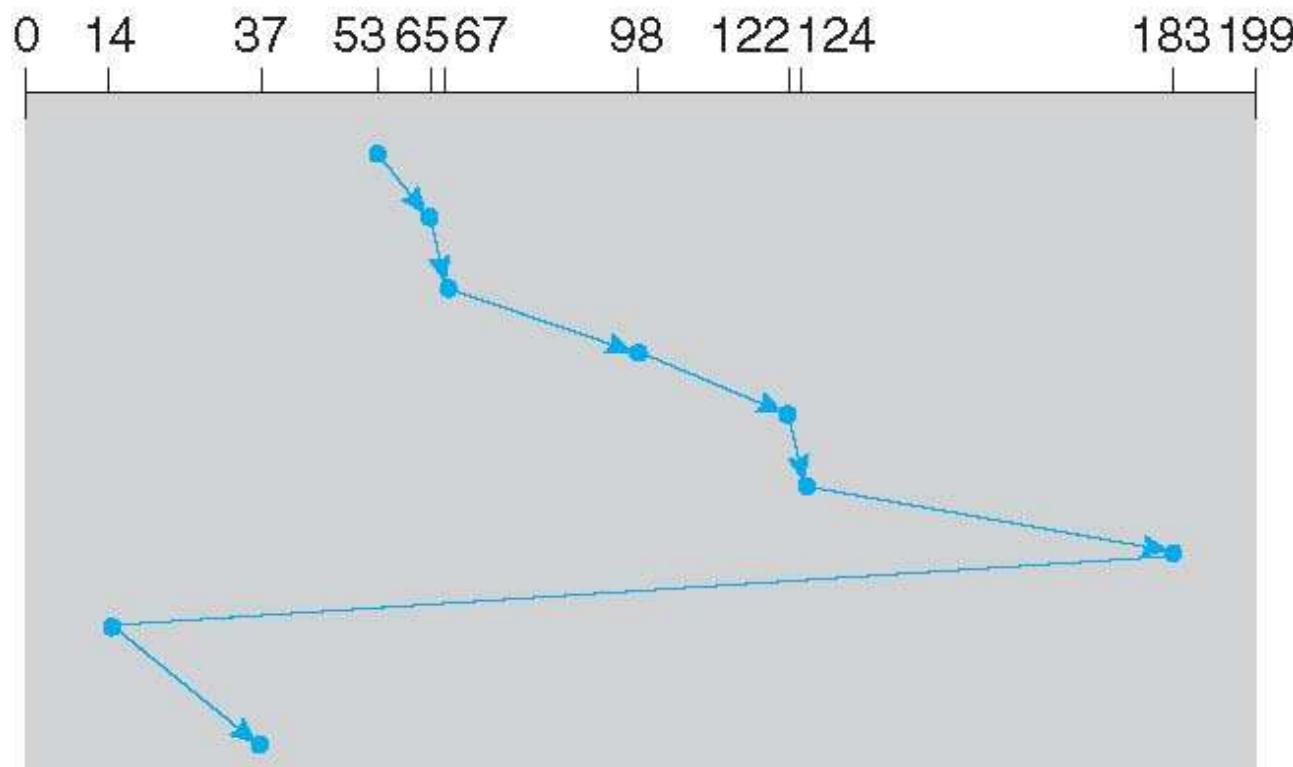
- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders?

# C-LOOK

- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Example:

Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the **previous request was at cylinder 125**. The queue of pending requests, in FIFO order, is:

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for FCFS, SSTF, SCAN, LOOK, C-SCAN and C-LOOK.

# Example (cont'd)

Solution:

Work Queue is 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

The FCFS schedule is 143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. The total seek distance is 7081(7703).

The SSTF schedule is 143, 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774. The total seek distance is 1745.

The SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86. The total seek distance is 9769.

The C-SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 0, 86, 130. The total seek distance is 9985.

The LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86. The total seek distance is 3319.

The C-LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 86, 130. The total seek distance is 3363.