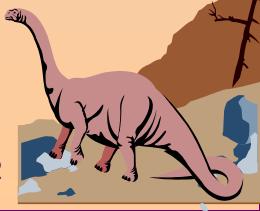




# Chapter 12: File System Implementation

- File System Structure
- File System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS



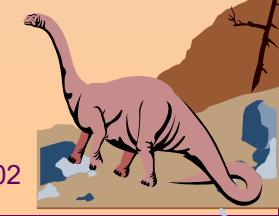
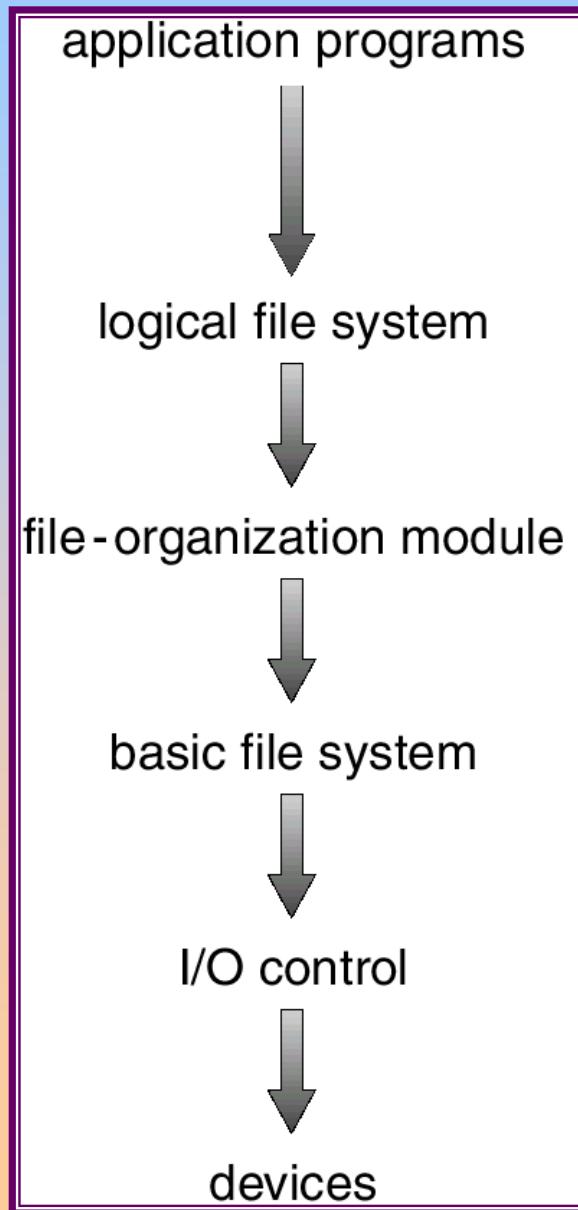


# File-System Structure

- File structure
    - ✖ Logical storage unit
    - ✖ Collection of related information
  - File system resides on secondary storage (disks).
  - File system organized into layers.
  - *File control block* – storage structure consisting of information about a file.
- 



# Layered File System





# A Typical File Control Block

file permissions

file dates (create, access, write)

file owner, group, ACL

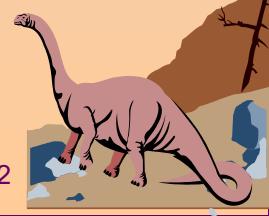
file size

file data blocks



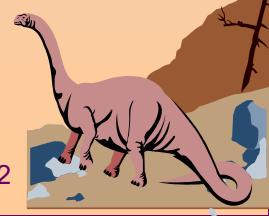
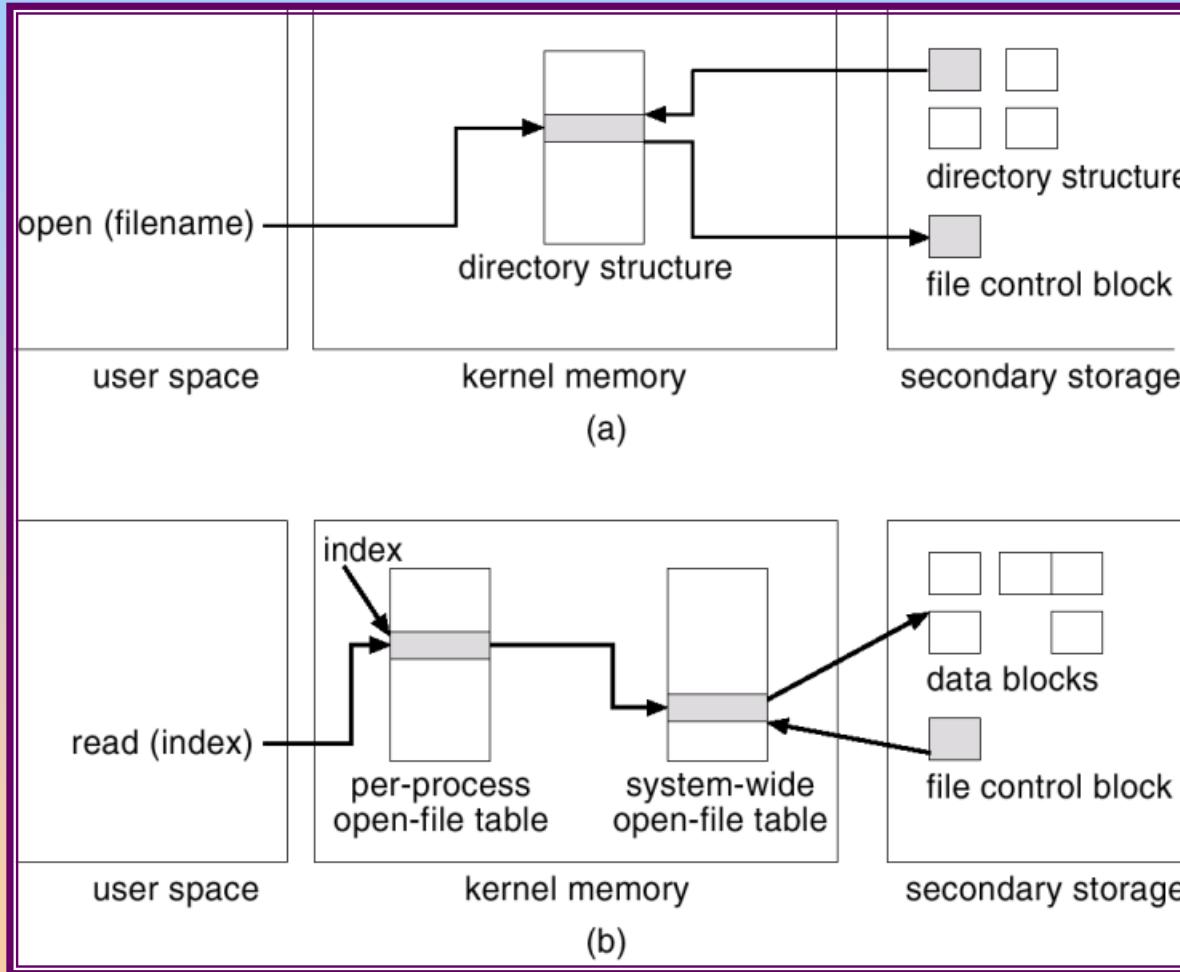
# In-Memory File System Structures

- The following figure illustrates the necessary file system structures provided by the operating systems.
- Figure 12-3(a) refers to opening a file.
- Figure 12-3(b) refers to reading a file.





# In-Memory File System Structures

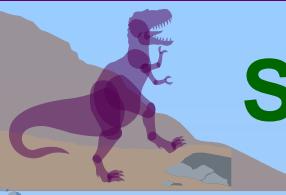




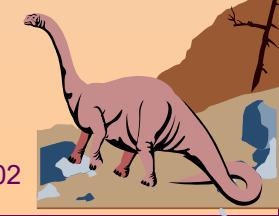
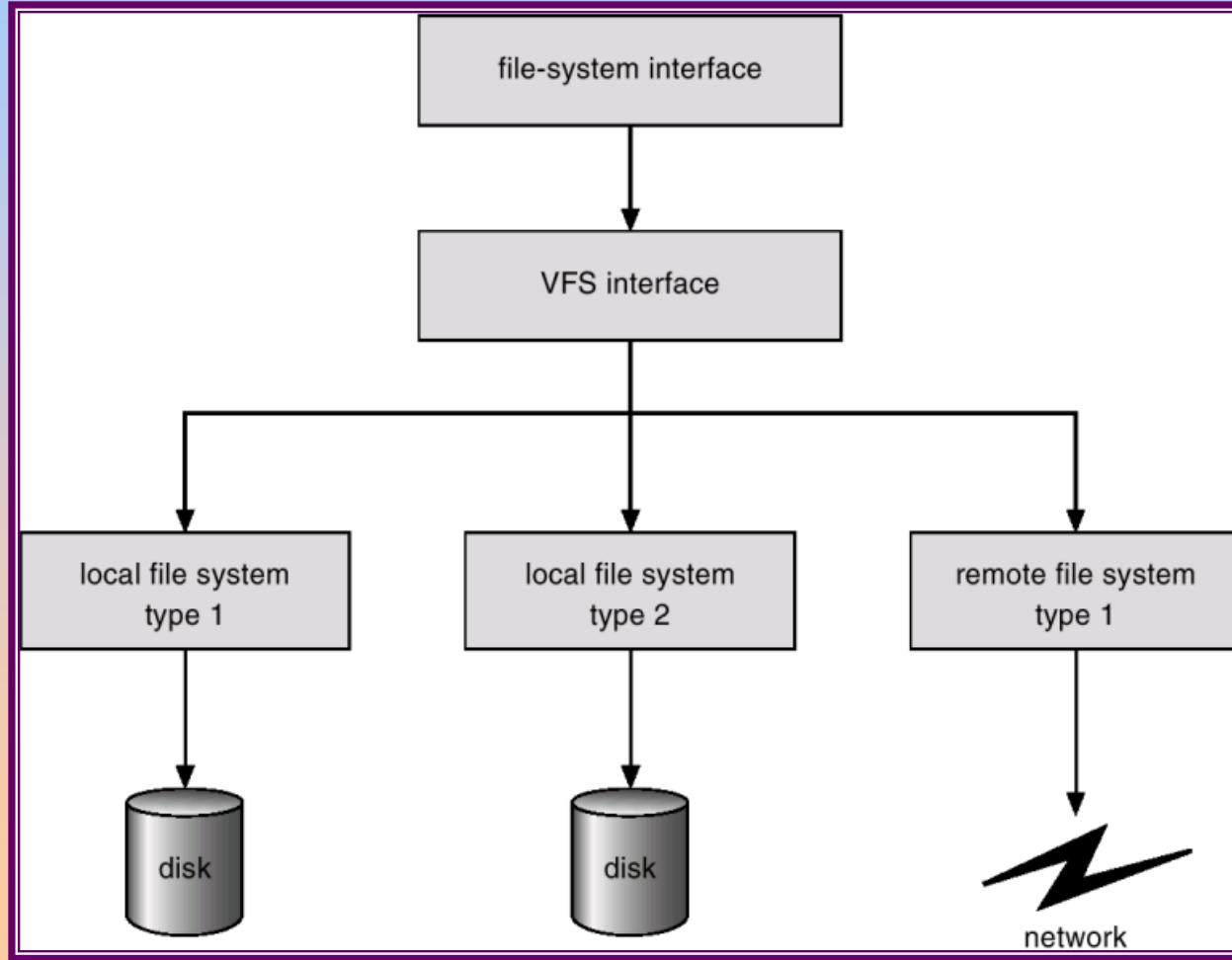
# Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.





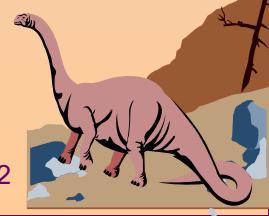
# Schematic View of Virtual File System





# Directory Implementation

- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
  
- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size





# Allocation Methods

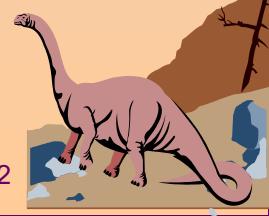
- An allocation method refers to how disk blocks are allocated for files:
- Contiguous allocation
- Linked allocation
- Indexed allocation





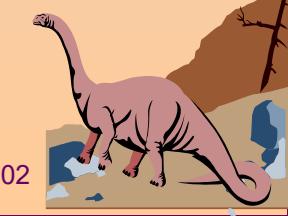
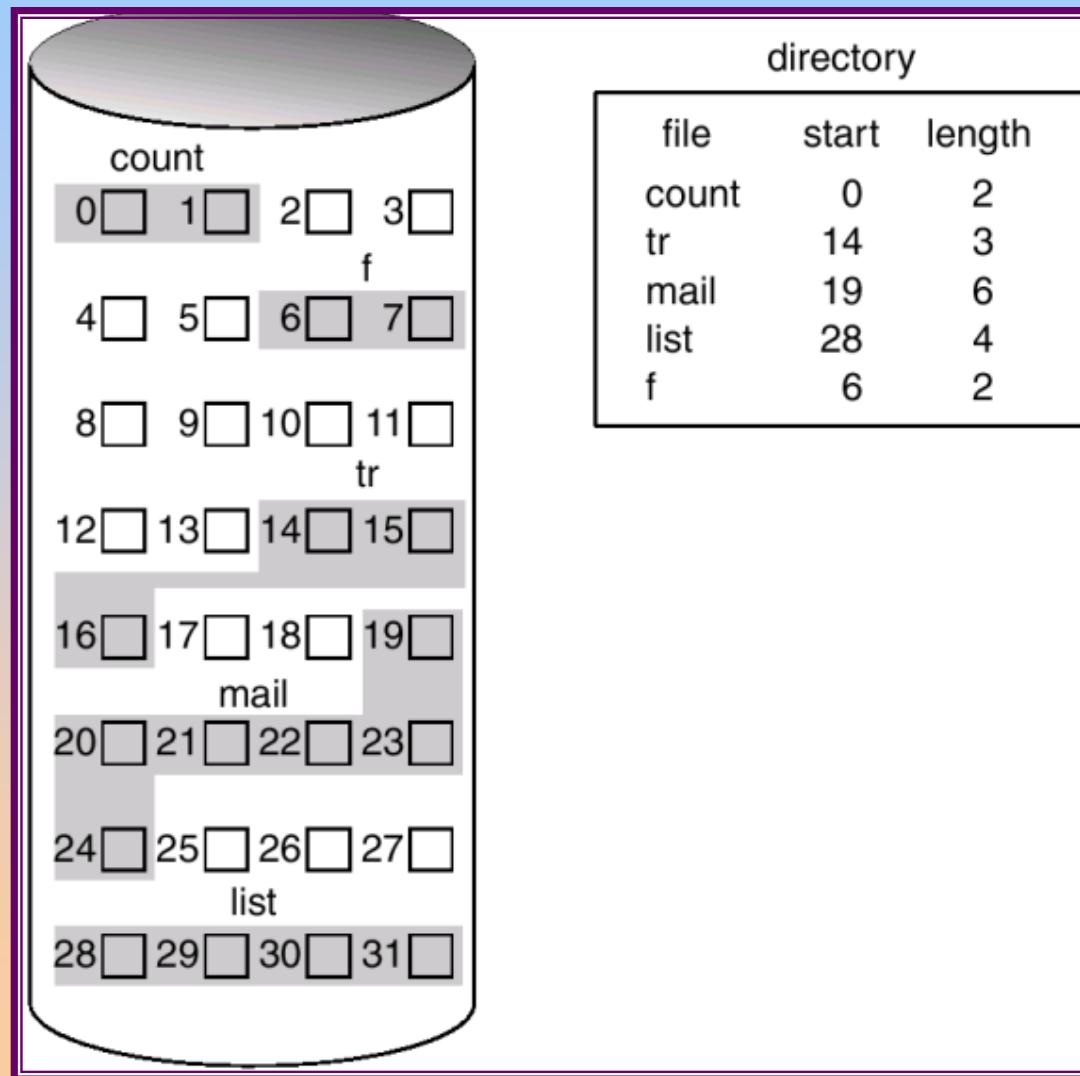
# Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- Simple – only starting location (block #) and length (number of blocks) are required.
- Random access.
- Wasteful of space (dynamic storage-allocation problem).
- Files cannot grow.





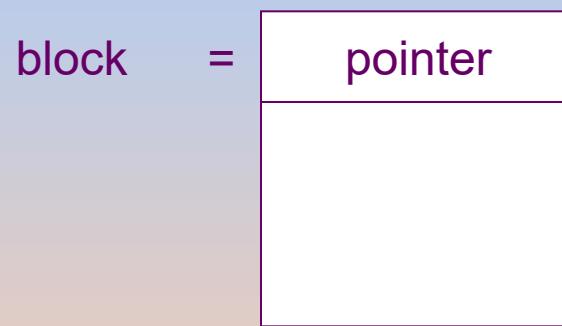
# Contiguous Allocation of Disk Space





# Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.





# Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping



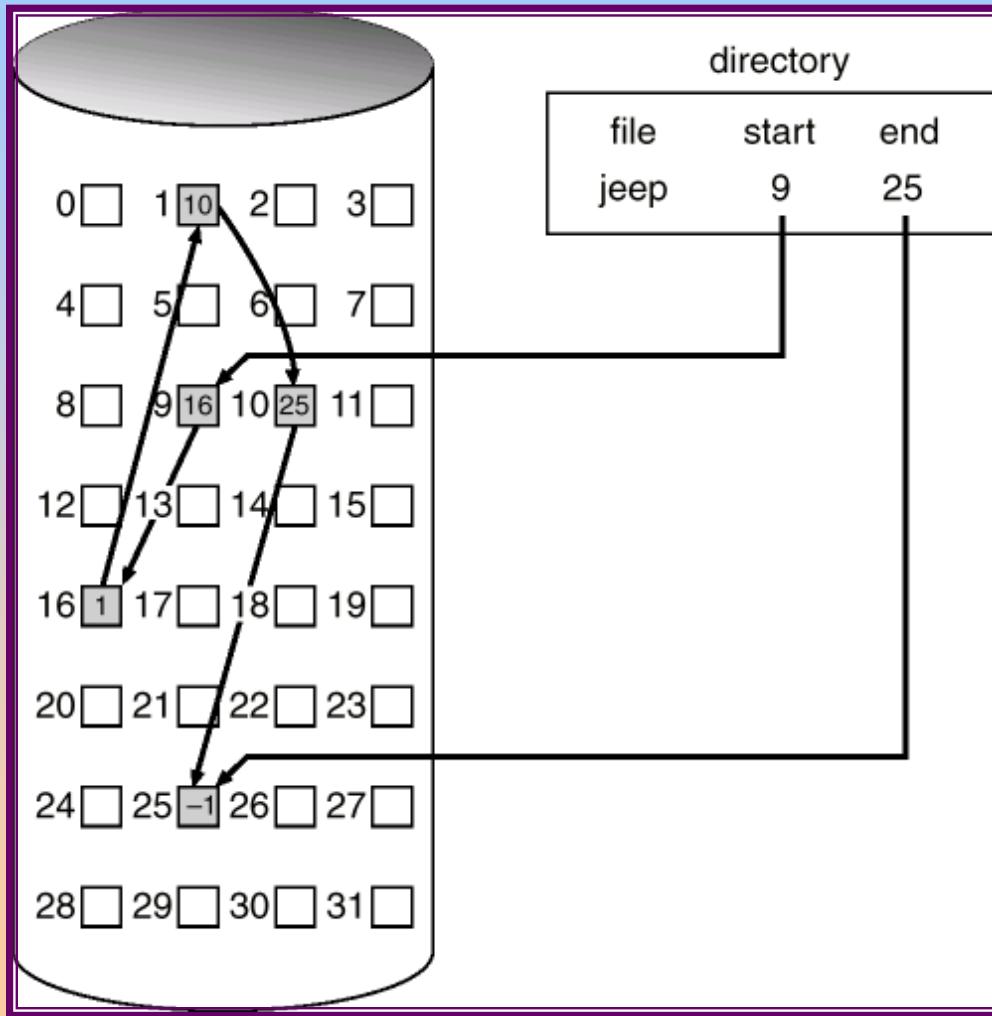
Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.



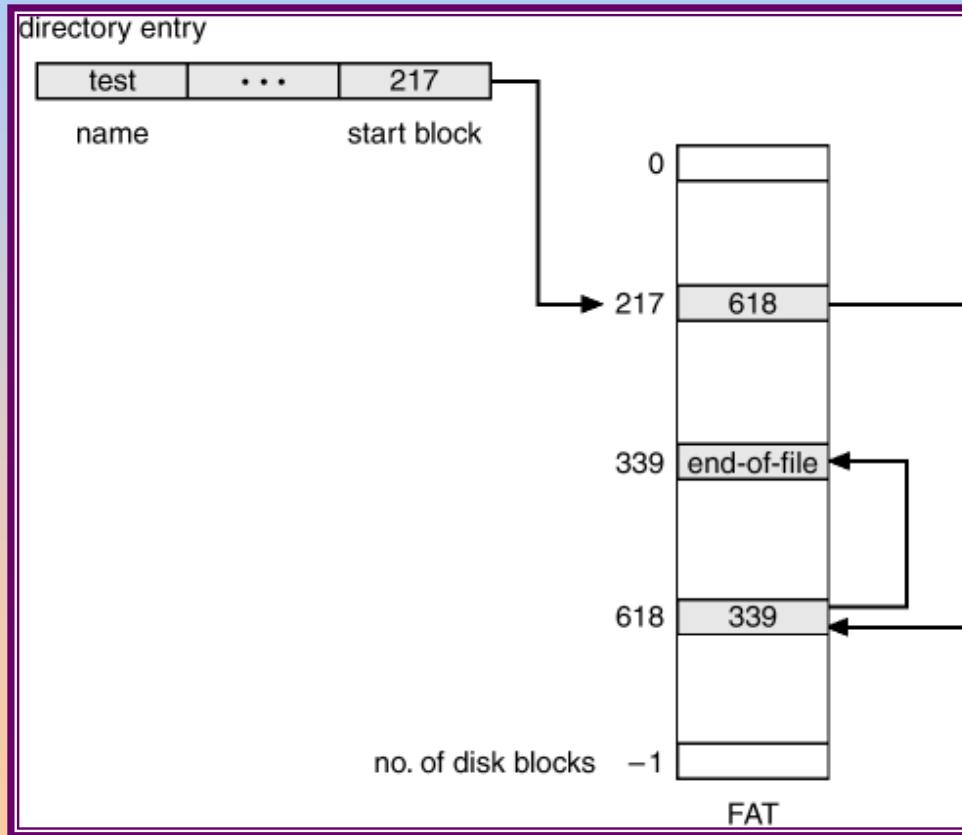
# Linked Allocation





# File-Allocation Table

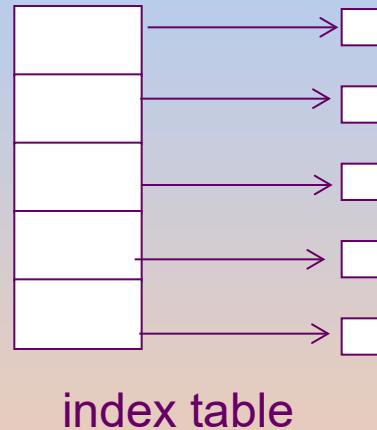
The information of files is maintained in a table by the OS - File Allocation Table





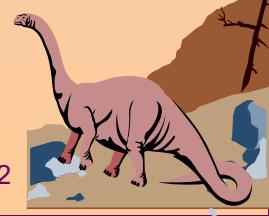
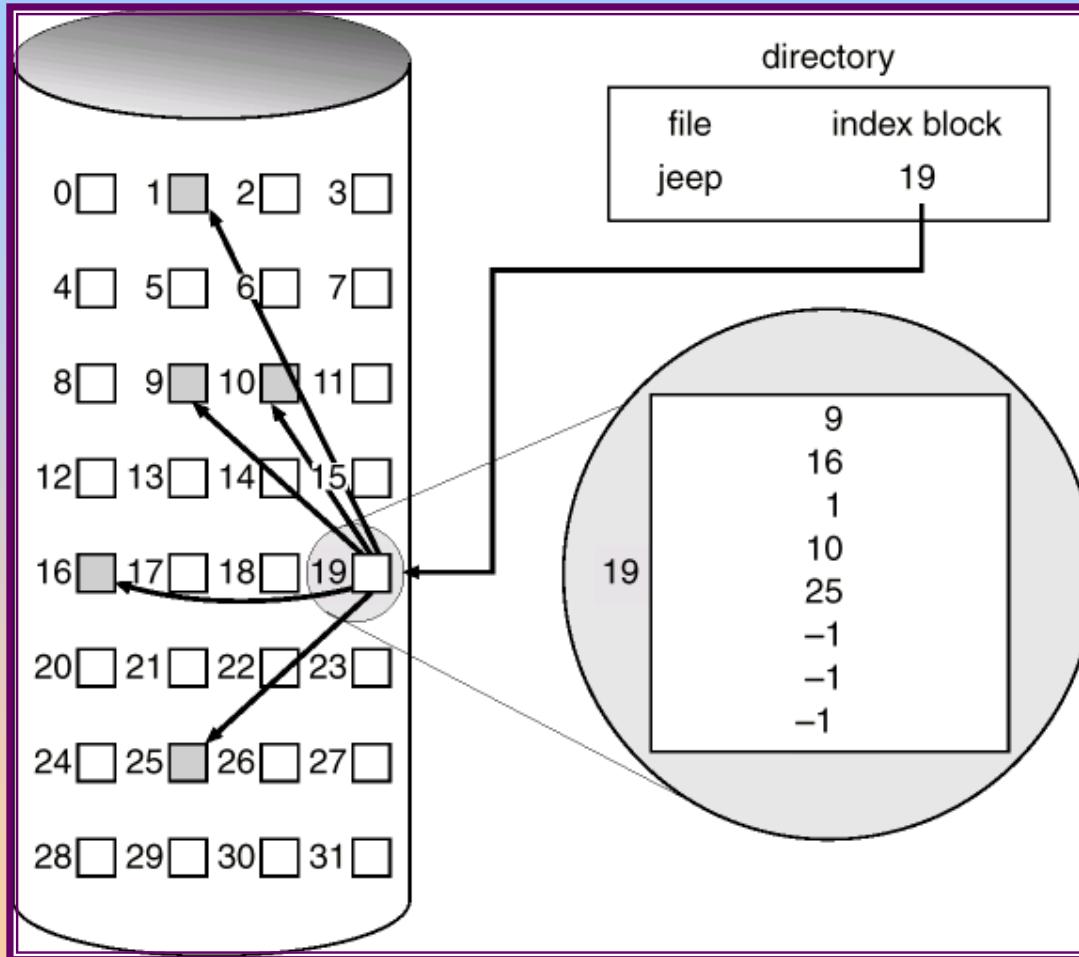
# Indexed Allocation

- Brings all pointers together into the *index block*.
- Logical view.





# Example of Indexed Allocation





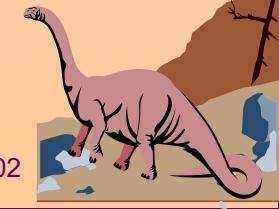
# Indexed Allocation (Cont.)

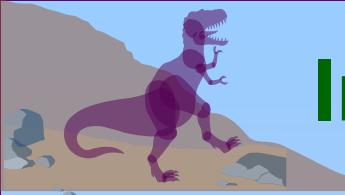
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



Q = displacement into index table

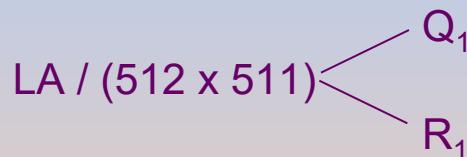
R = displacement into block





# Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words).
- Linked scheme – Link blocks of index table (no limit on size).



$Q_1$  = block of index table

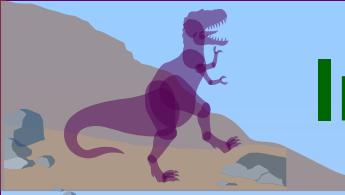
$R_1$  is used as follows:



$Q_2$  = displacement into block of index table

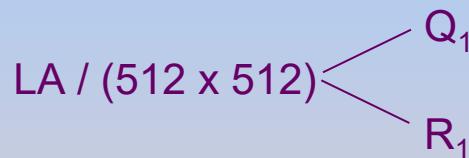
$R_2$  displacement into block of file:





# Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is  $512^3$ )



$Q_1$  = displacement into outer-index

$R_1$  is used as follows:



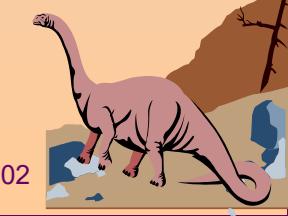
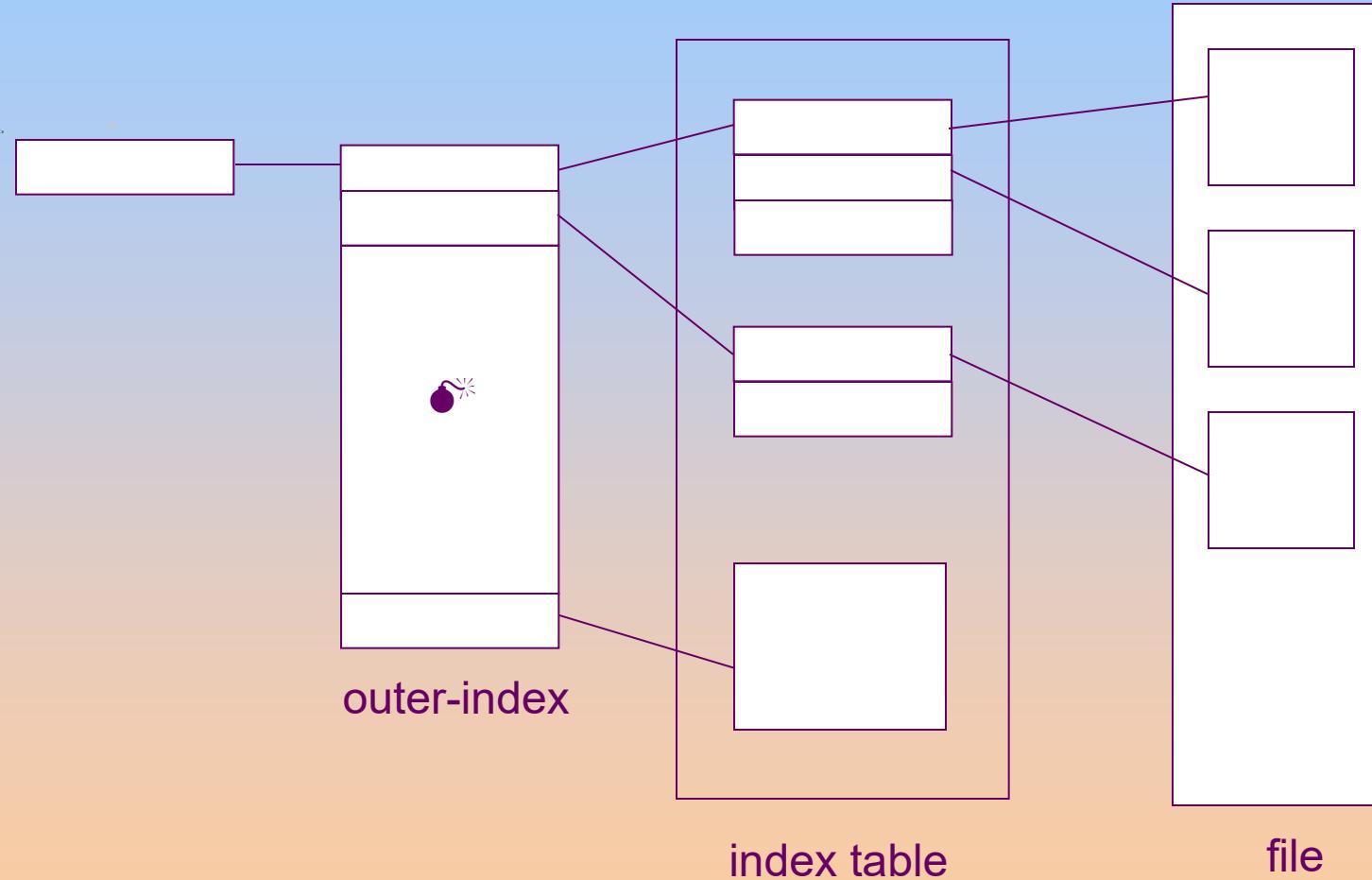
$Q_2$  = displacement into block of index table

$R_2$  displacement into block of file:



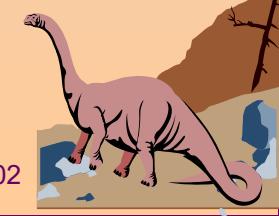
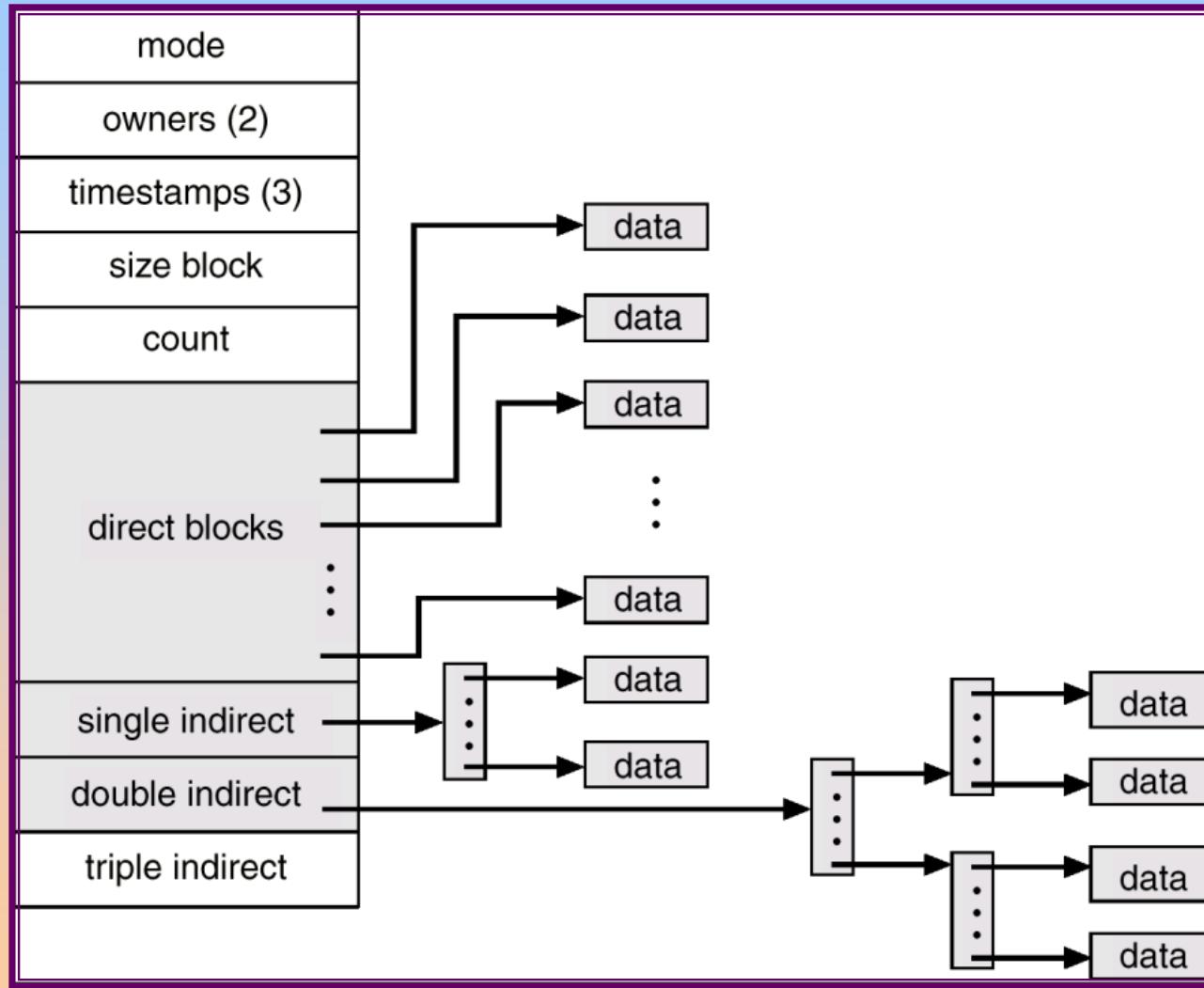


# Indexed Allocation – Mapping (Cont.)





# Combined Scheme: UNIX (4K bytes per block)





# Byte Capacity of a File

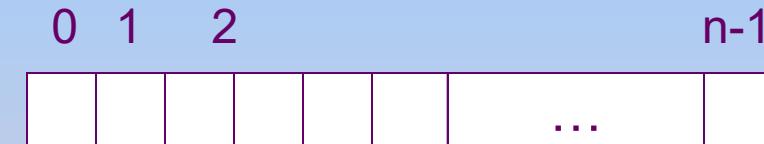
- System V UNIX.
- Assume that
  - ☞ Run with 13 entries
  - ☞ 1 logical block : 1K bytes = 1024 bytes
  - ☞ Block number address : a 32 bit (4byte) integer

- 1 Indirect block can hold up to 256 block numbers (1024byte / 4byte)
  - 10 direct blocks with 1K bytes data each=10K bytes
  - 1 indirect block with 256 direct blocks=  $1K \times 256 = 256K$  bytes
  - 1 double indirect block with 256 indirect blocks= $256K \times 256 = 64M$  bytes
  - 1 triple indirect block with 256 double indirect blocks= $64M \times 256 = 16G$ 
    - ☞ The maximum number of bytes that could be held in a file is calculated is 16 G bytes,
  - Size of a file : 4G ( $2^{32}$ ), if file size field in inode is 32bits



# Free-Space Management

## ■ Bit vector ( $n$ blocks)



$\text{bit}[i] = \begin{cases} \text{white square} & 0 \Rightarrow \text{block}[i] \text{ free} \\ \text{black square with cross} & 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$

## Block number calculation

(number of bits per word) \*  
(number of 0-value words) +  
offset of first 1 bit





# Free-Space Management (Cont.)

- Bit map requires extra space. Example:
  - block size =  $2^{12}$  bytes
  - disk size =  $2^{30}$  bytes (1 gigabyte)
  - $n = 2^{30}/2^{12} = 2^{18}$  bits (or 32K bytes)
- Easy to get contiguous files
- Linked list (free list)
  - ☞ Cannot get contiguous space easily
  - ☞ No waste of space
- Grouping
- Counting



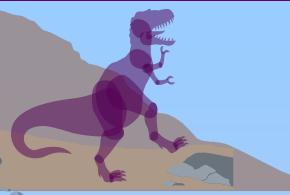


# Free-Space Management (Cont.)

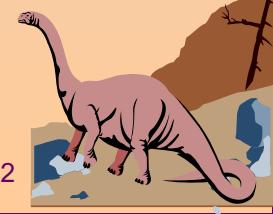
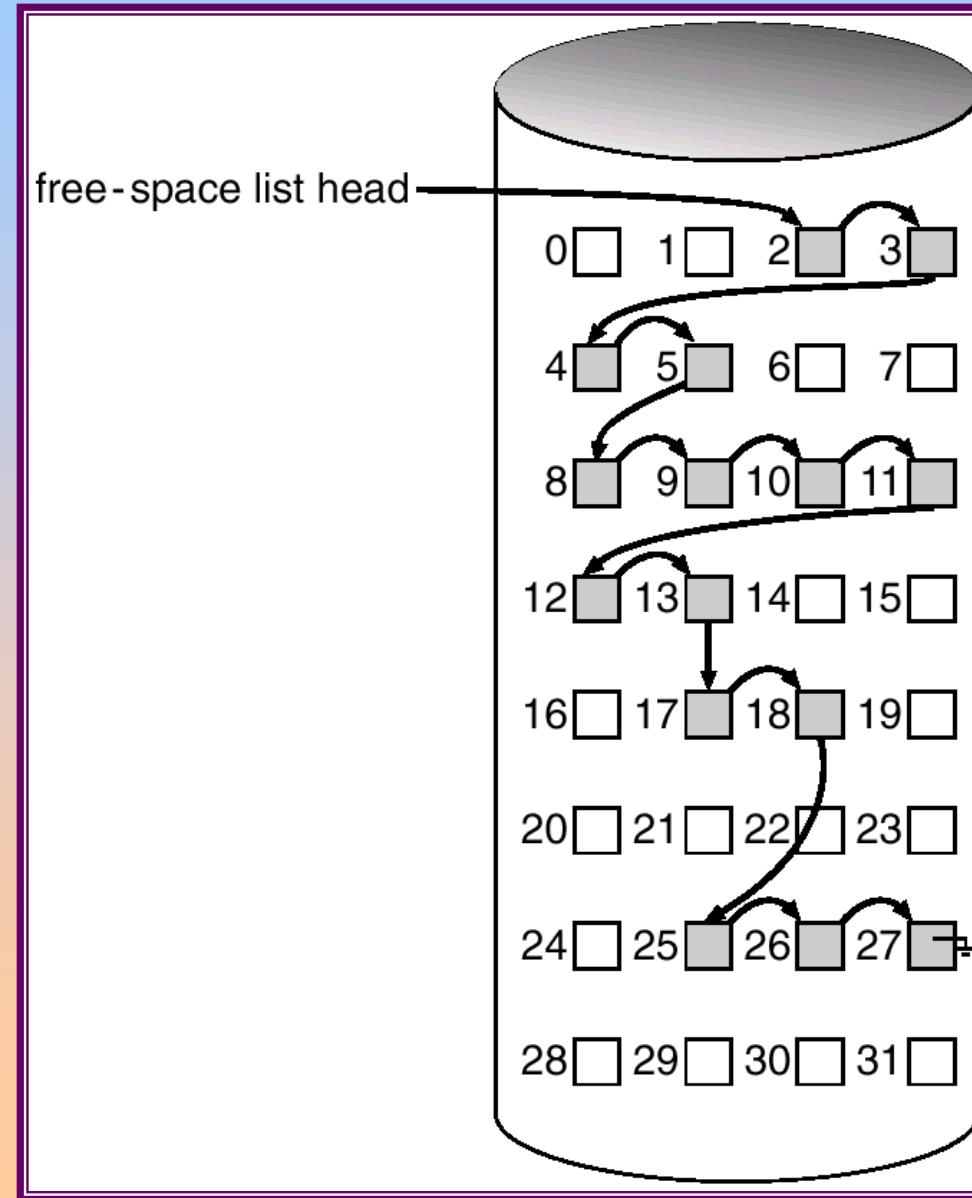
- Need to protect:

- ☞ Pointer to free list
  - ☞ Bit map
    - ☞ Must be kept on disk
    - ☞ Copy in memory and disk may differ.
    - ☞ Cannot allow for  $\text{block}[i]$  to have a situation where  $\text{bit}[i] = 1$  in memory and  $\text{bit}[i] = 0$  on disk.
  - ☞ Solution:
    - ☞ Set  $\text{bit}[i] = 1$  in disk.
    - ☞ Allocate  $\text{block}[i]$
    - ☞ Set  $\text{bit}[i] = 1$  in memory





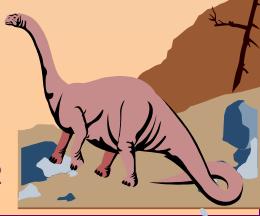
# Linked Free Space List on Disk





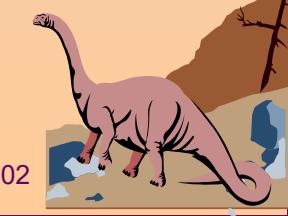
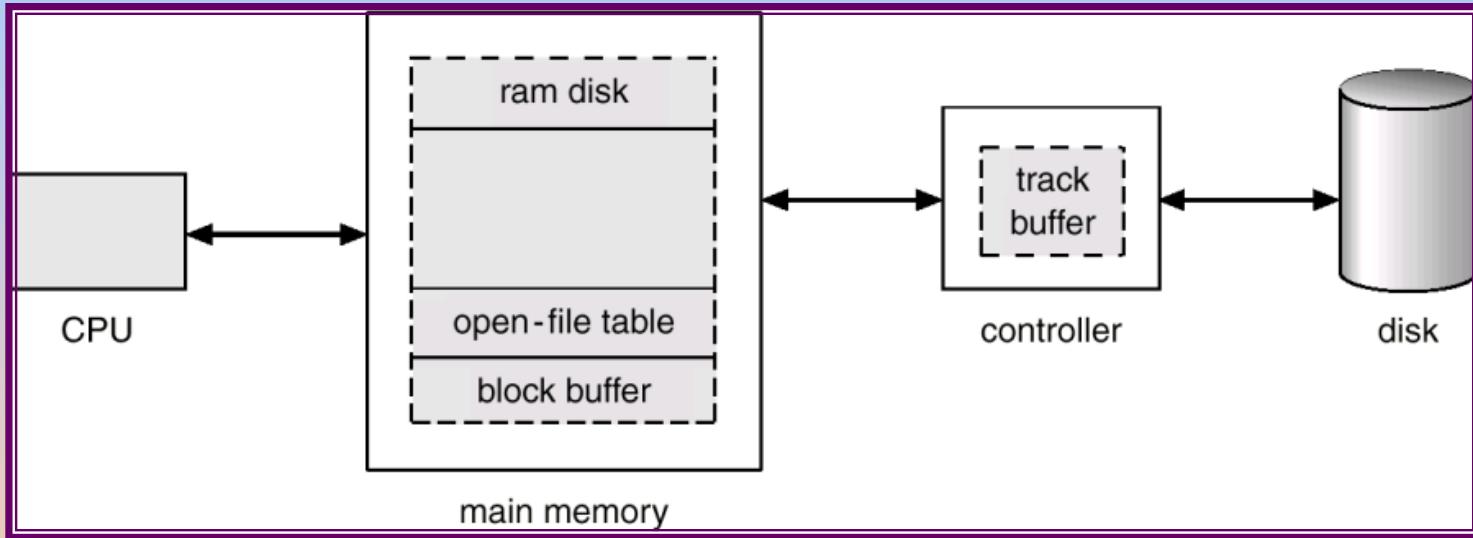
# Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
  
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.





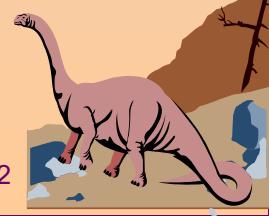
# Various Disk-Caching Locations





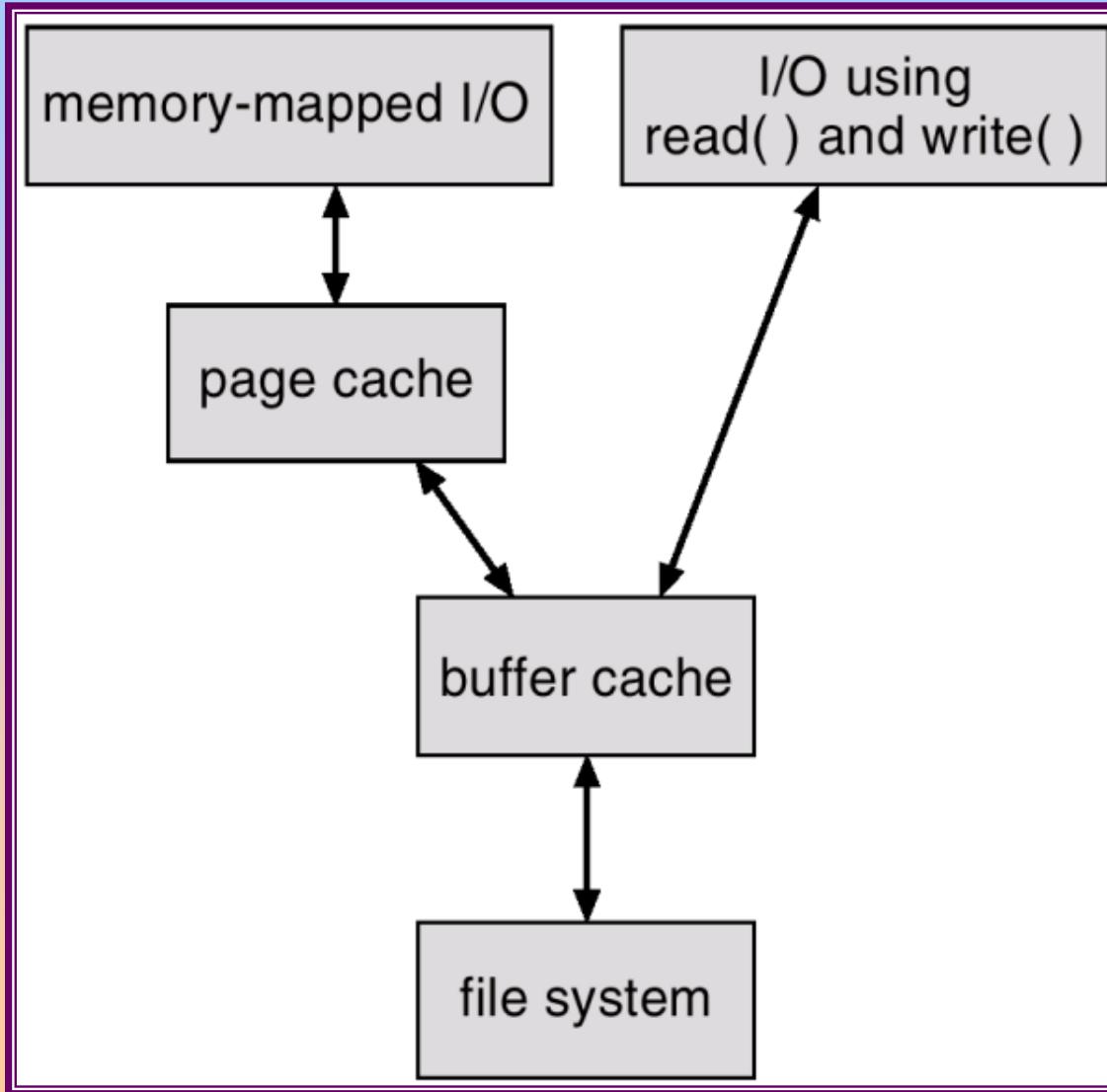
# Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.
- This leads to the following figure.





# I/O Without a Unified Buffer Cache





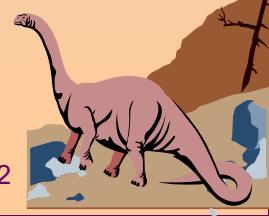
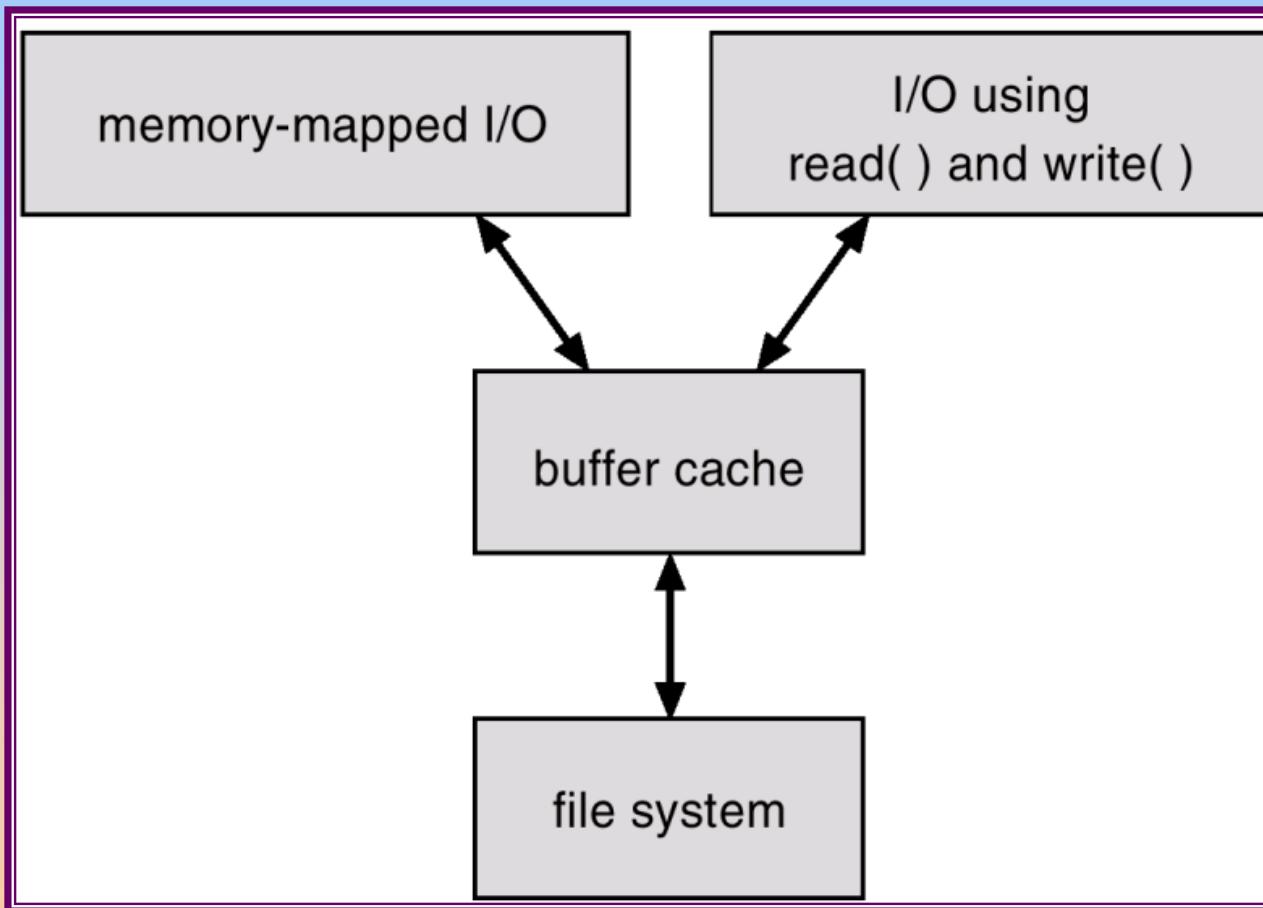
# Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.



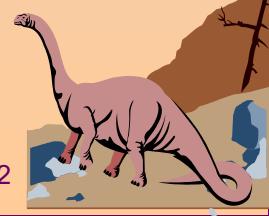


# I/O Using a Unified Buffer Cache





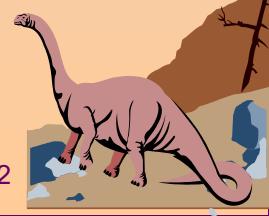
# Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
  - Use system programs to *back up* data from disk to another storage device (floppy disk, magnetic tape).
  - Recover lost file or disk by *restoring* data from backup.
- 



# Log Structured File Systems

- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**.
- All transactions are written to a **log**. A transaction is considered **committed** once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.





# The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet).





# NFS (Cont.)

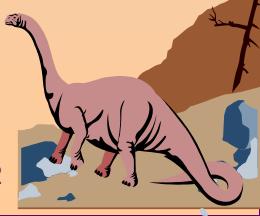
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner.
  - ☞ A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory.
  - ☞ Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided. Files in the remote directory can then be accessed in a transparent manner.
  - ☞ Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory.





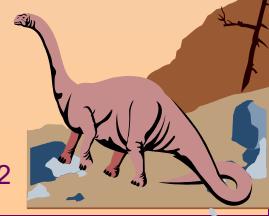
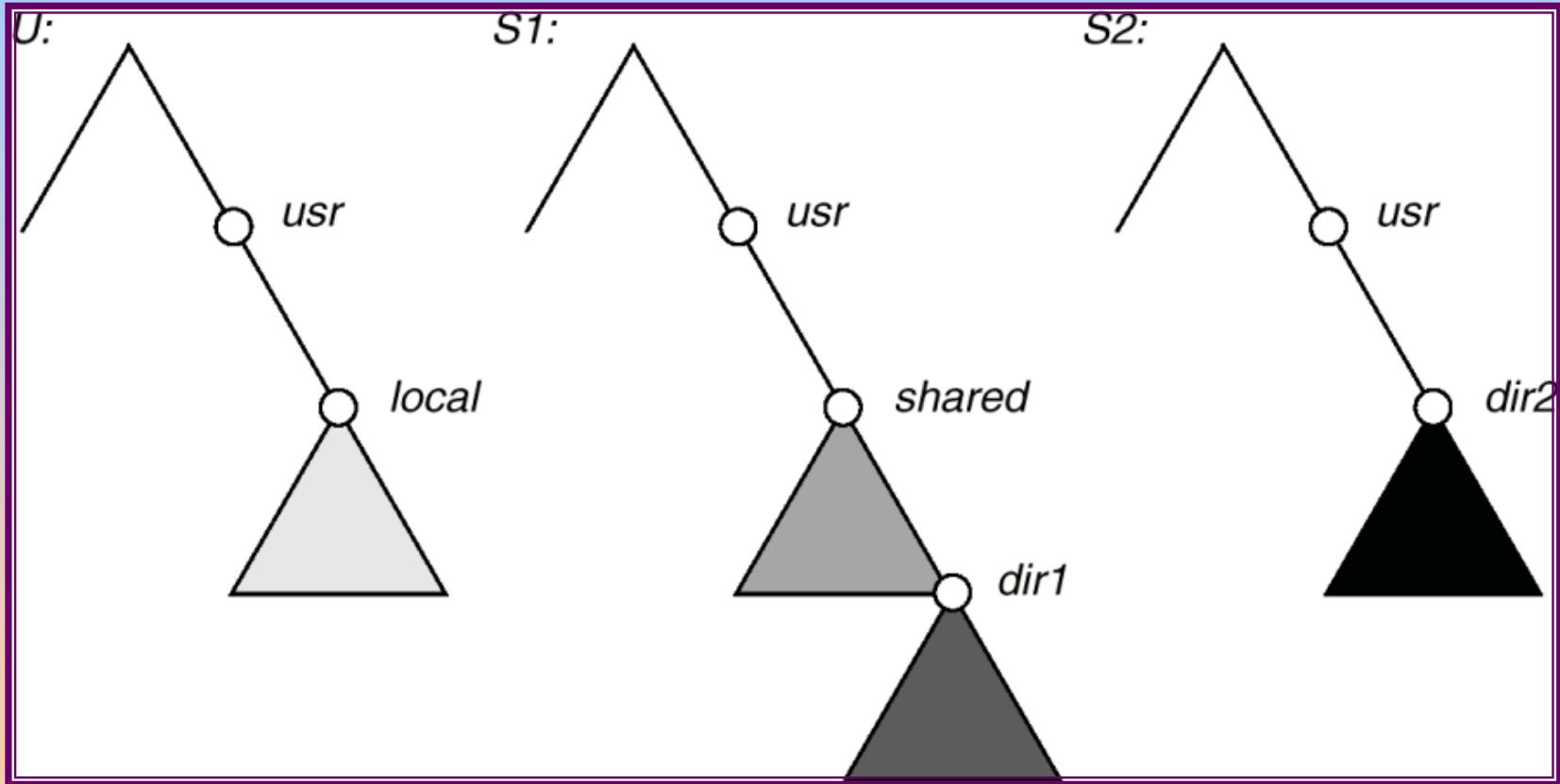
# NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media.
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services.



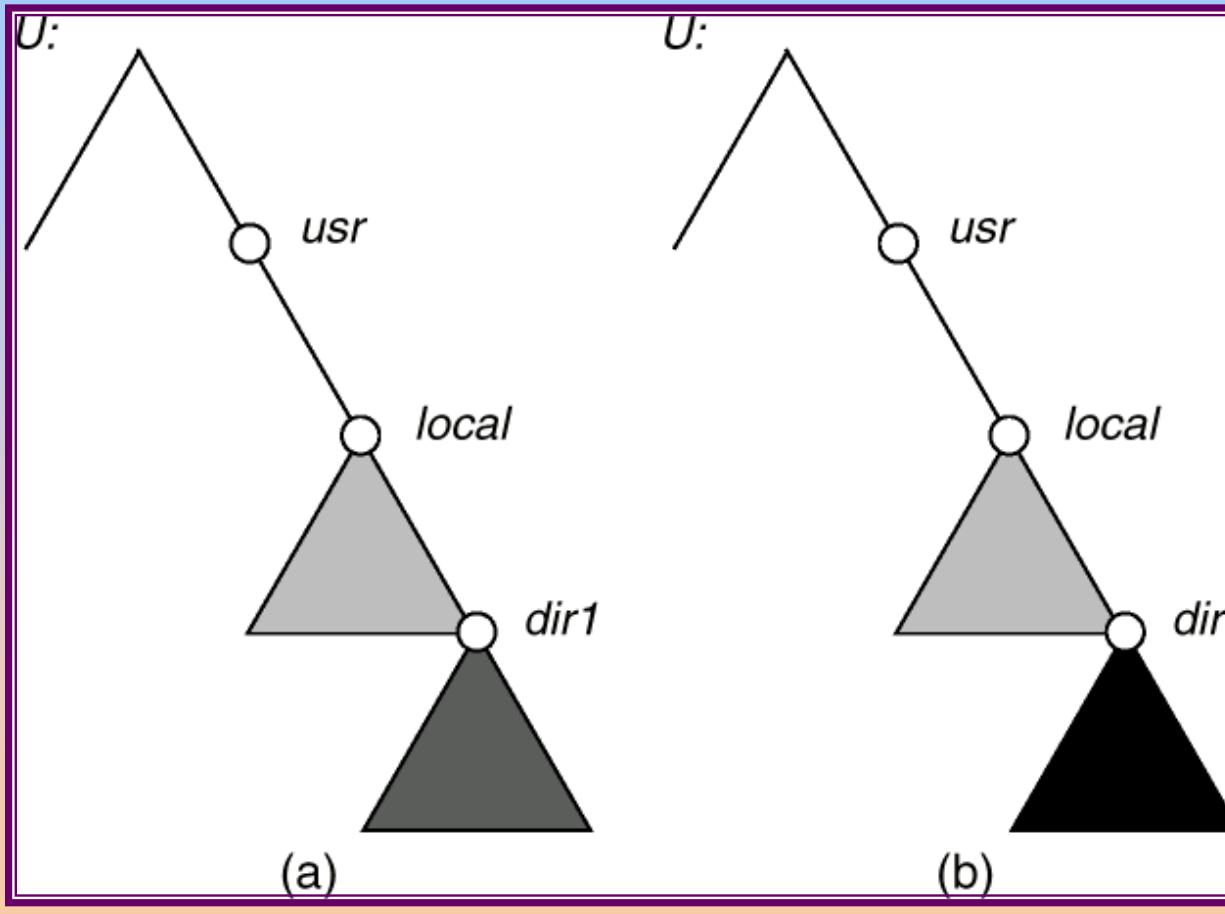


# Three Independent File Systems



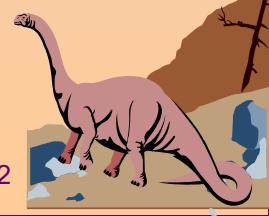


# Mounting in NFS



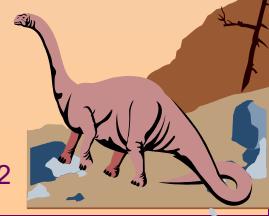
Mounts

Cascading mounts



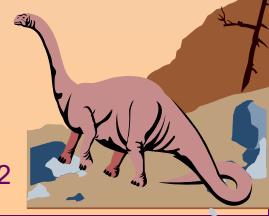


# NFS Mount Protocol

- Establishes initial logical connection between server and client.
  - Mount operation includes name of remote directory to be mounted and name of server machine storing it.
    - ☞ Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine.
    - ☞ *Export list* – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them.
  - Following a mount request that conforms to its export list, the server returns a *file handle*—a key for further accesses.
  - File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system.
  - The mount operation changes only the user's view and does not affect the server side.
- 



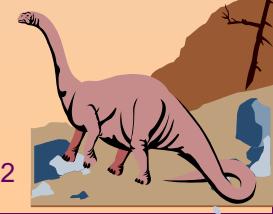
# NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
    - ☞ searching for a file within a directory
    - ☞ reading a set of directory entries
    - ☞ manipulating links and directories
    - ☞ accessing file attributes
    - ☞ reading and writing files
  - NFS servers are *stateless*; each request has to provide a full set of arguments.
  - Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching).
  - The NFS protocol does not provide concurrency-control mechanisms.
- 



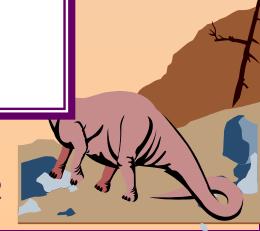
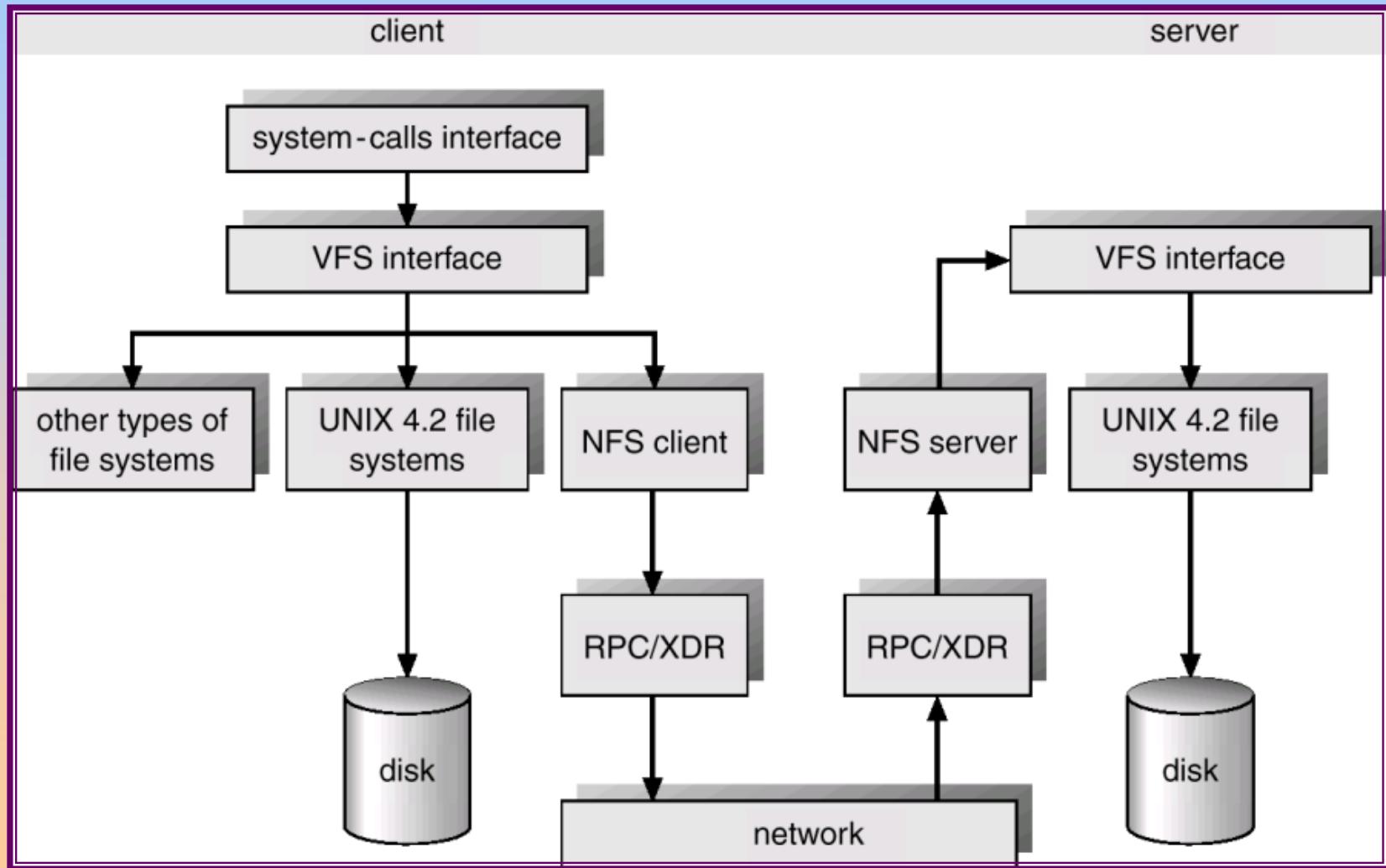
# Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and file descriptors).
- *Virtual File System (VFS)* layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  - ☞ The VFS activates file-system-specific operations to handle local requests according to their file-system types.
  - ☞ Calls the NFS protocol procedures for remote requests.
- NFS service layer – bottom layer of the architecture; implements the NFS protocol.





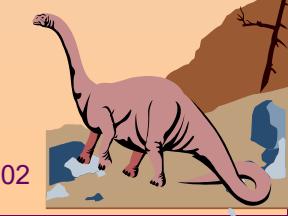
# Schematic View of NFS Architecture





# NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode.
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names.





# NFS Remote Operations

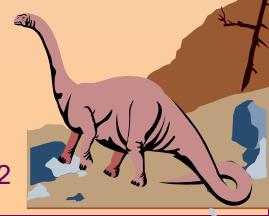
- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files).
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. Cached file blocks are used only if the corresponding cached attributes are up to date.
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server.
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.





# Chapter 14: Mass-Storage Systems

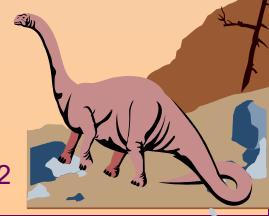
- Disk Structure
- Disk Scheduling
- Disk Management
- Swap-Space Management
- RAID Structure
- Disk Attachment
- Stable-Storage Implementation
- Tertiary Storage Devices
- Operating System Issues
- Performance Issues





# Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- 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.
  - ☞ Disk uses device drivers to interact with the OS





# 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.
  - Access time has two major components
    - ☞ *Seek time* is the time for the disk arm to move the heads to the cylinder containing the desired sector.
    - ☞ *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
  - 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.)

- Deals with the order in which disk access requests must be serviced
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

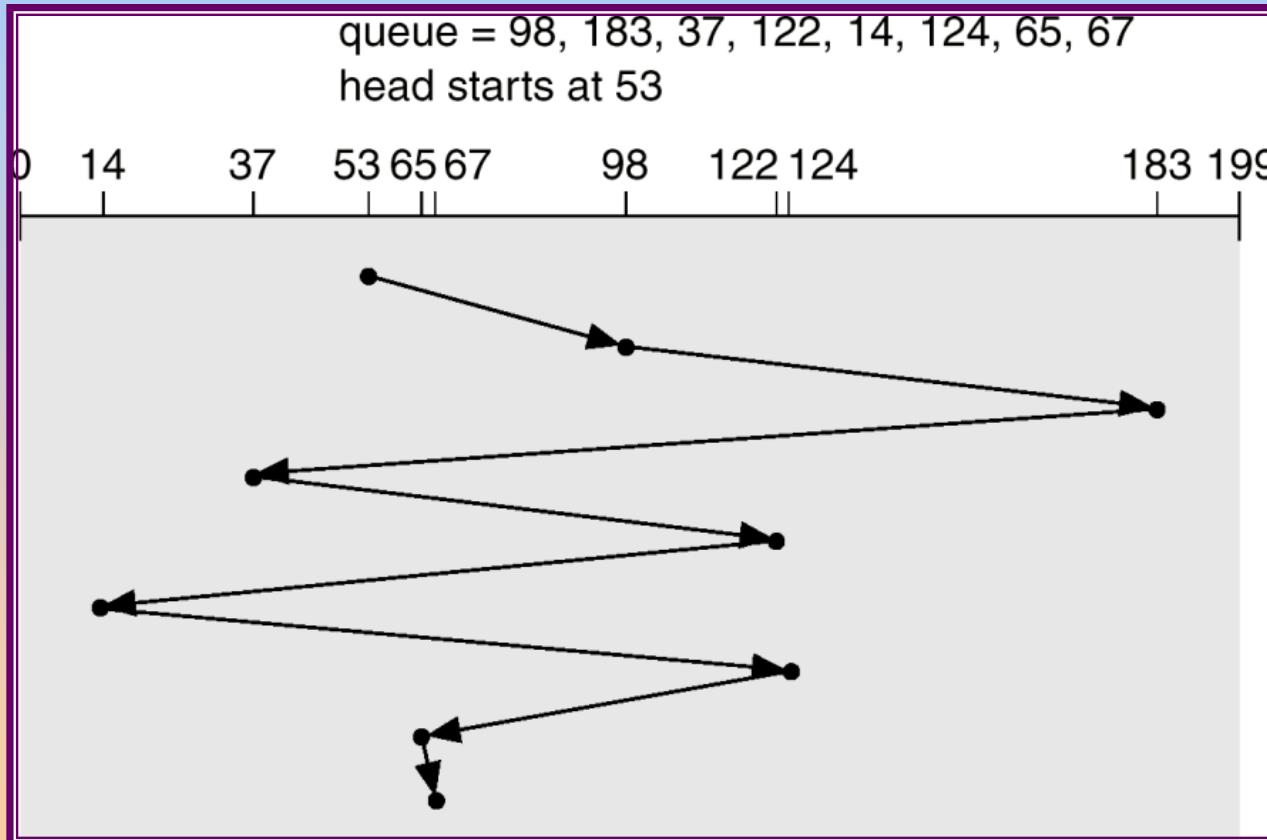
Head pointer 53





# FCFS

Illustration shows total head movement of 640 cylinders





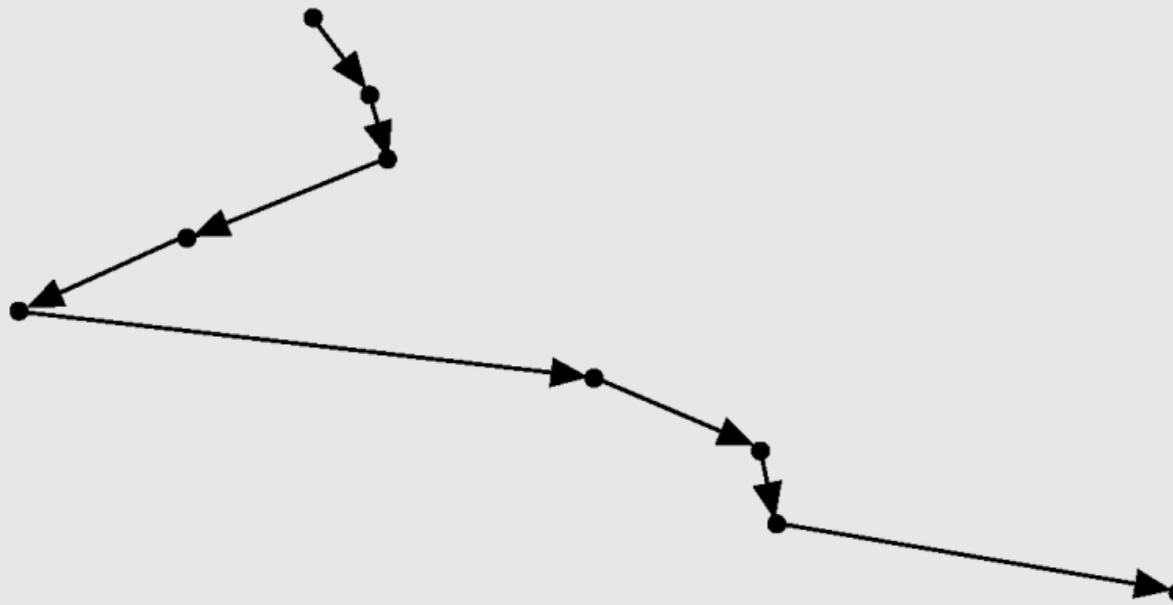
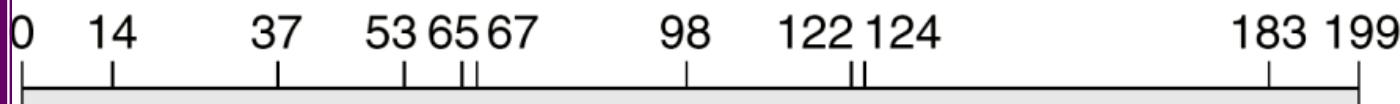
# Shortest Seek Time First

- 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.
- 

# SSTF (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

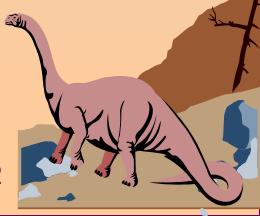
head starts at 53



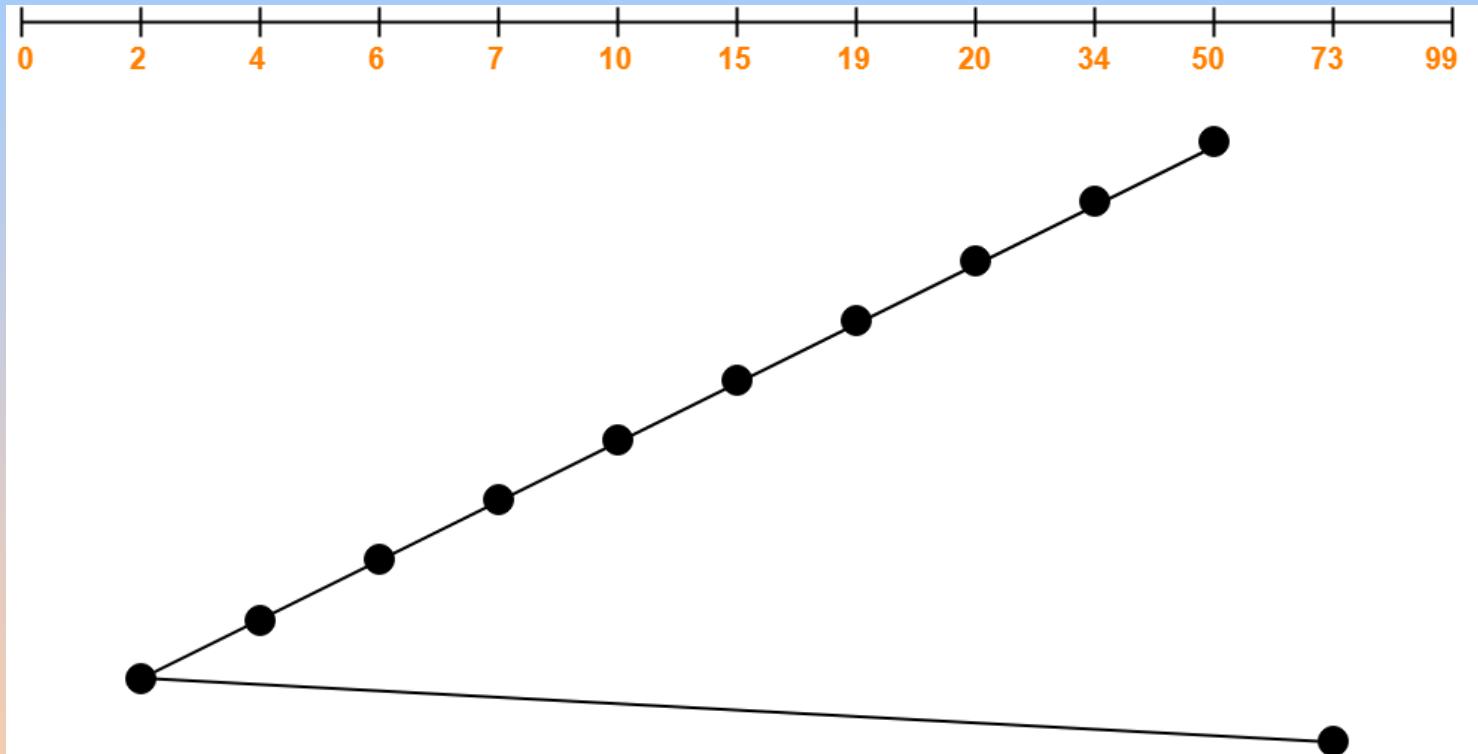


# SSTF

- Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence-
  - 4, 34, 10, 7, 19, 73, 2, 15, 6, 20
- Assuming that the head is currently at cylinder 50, what is the time **taken to satisfy all requests** if it takes 1 ms to move from one cylinder to adjacent one \_\_\_\_\_
- The cylinders are numbered from 0 to 99. The **total head movement** (in number of cylinders) incurred while servicing these requests is \_\_\_\_\_.



# SSTF





# SSTF

Total head movements incurred while servicing these requests

$$= (50 - 34) + (34 - 20) + (20 - 19) + (19 - 15) + (15 - 10) + (10 - 7) + \\(7 - 6) + (6 - 4) + (4 - 2) + (73 - 2)$$

$$= 16 + 14 + 1 + 4 + 5 + 3 + 1 + 2 + 2 + 71$$

$$= 119$$

Time taken for one head movement = 1 msec. So,

Time taken for 119 head movements

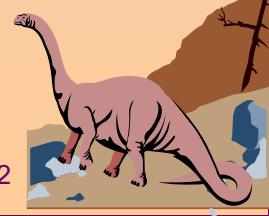
$$= 119 \times 1 \text{ msec}$$

$$= 119 \text{ msec}$$

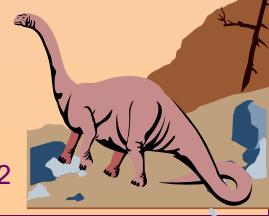
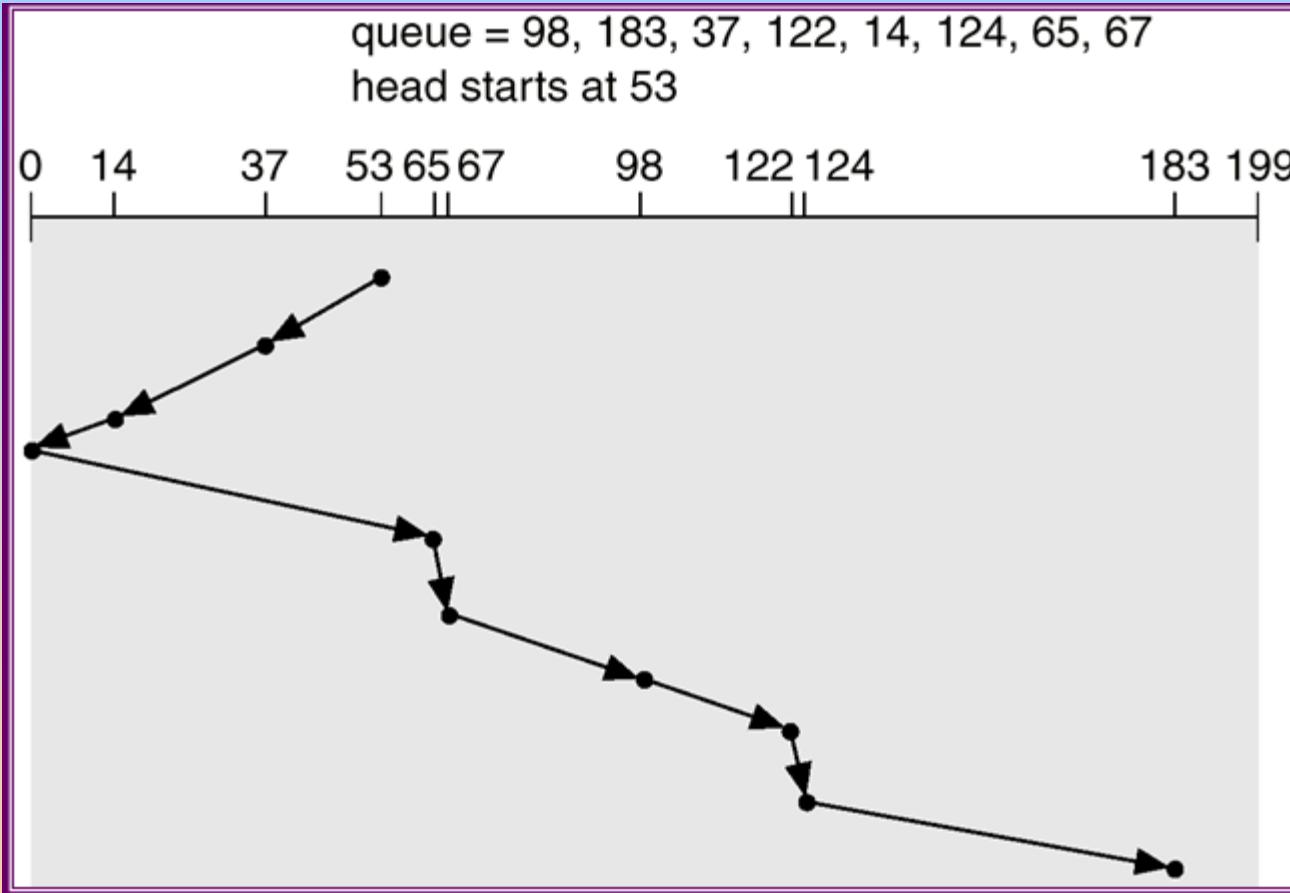




# SCAN

- 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.
  - Sometimes called the ***elevator algorithm***.
  - Illustration shows total head movement of 208 cylinders.
- 

# SCAN (Cont.)





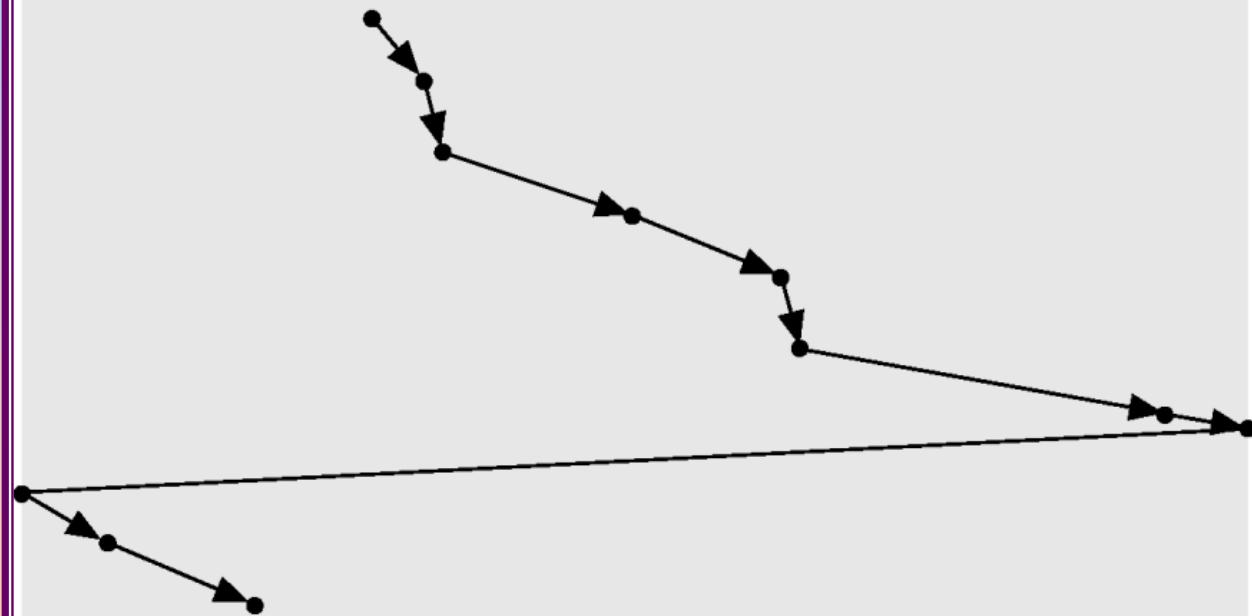
# C-SCAN

- 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.
- 

# C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53

0 14 37 53 65 67 98 122 124 183 199



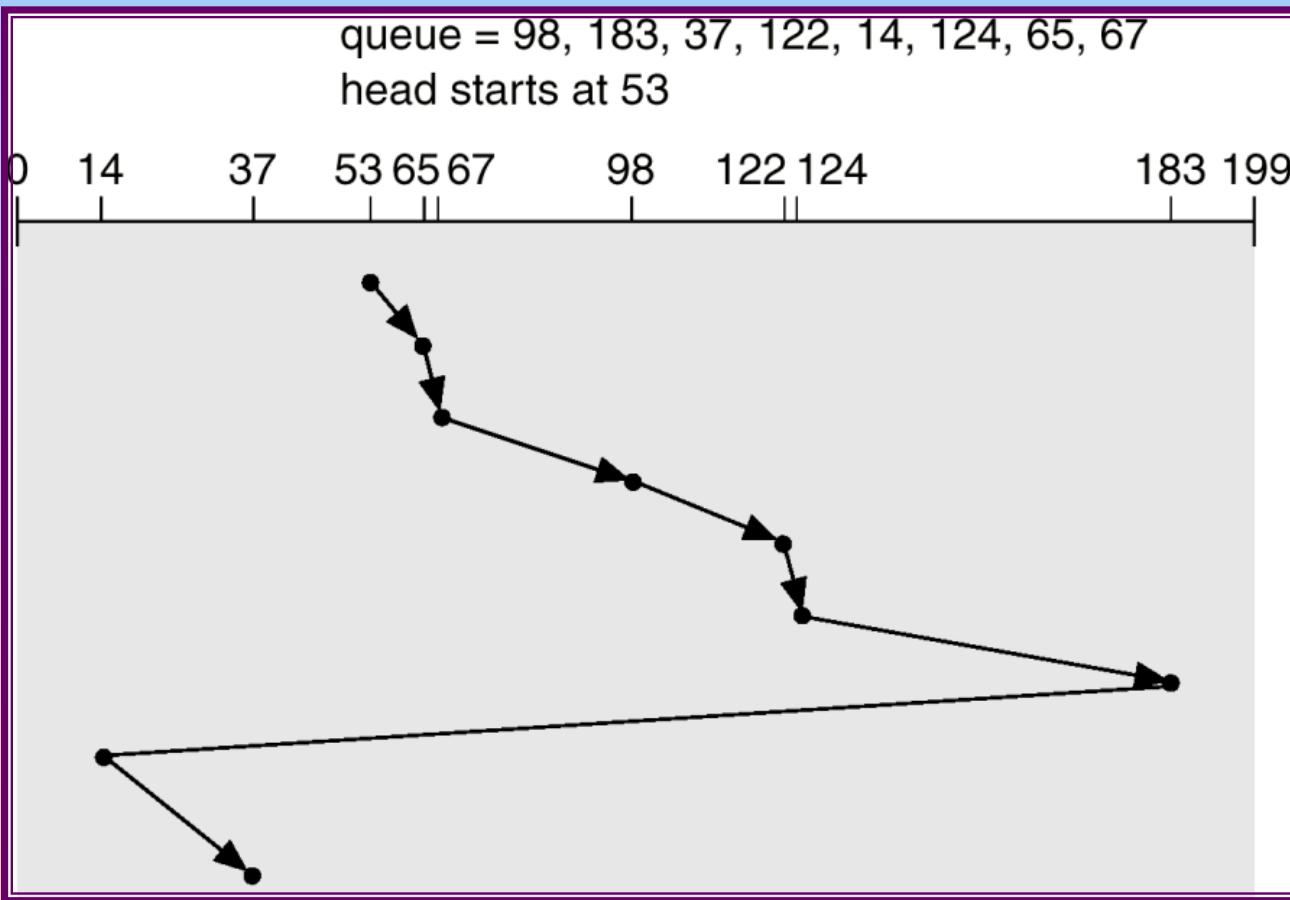


# C-LOOK

- 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.



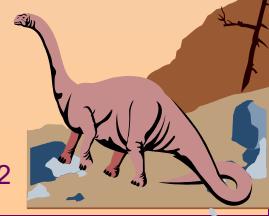
# C-LOOK (Cont.)





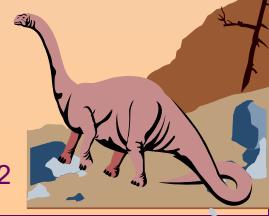
# Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or C-LOOK is a reasonable choice for the default algorithm.

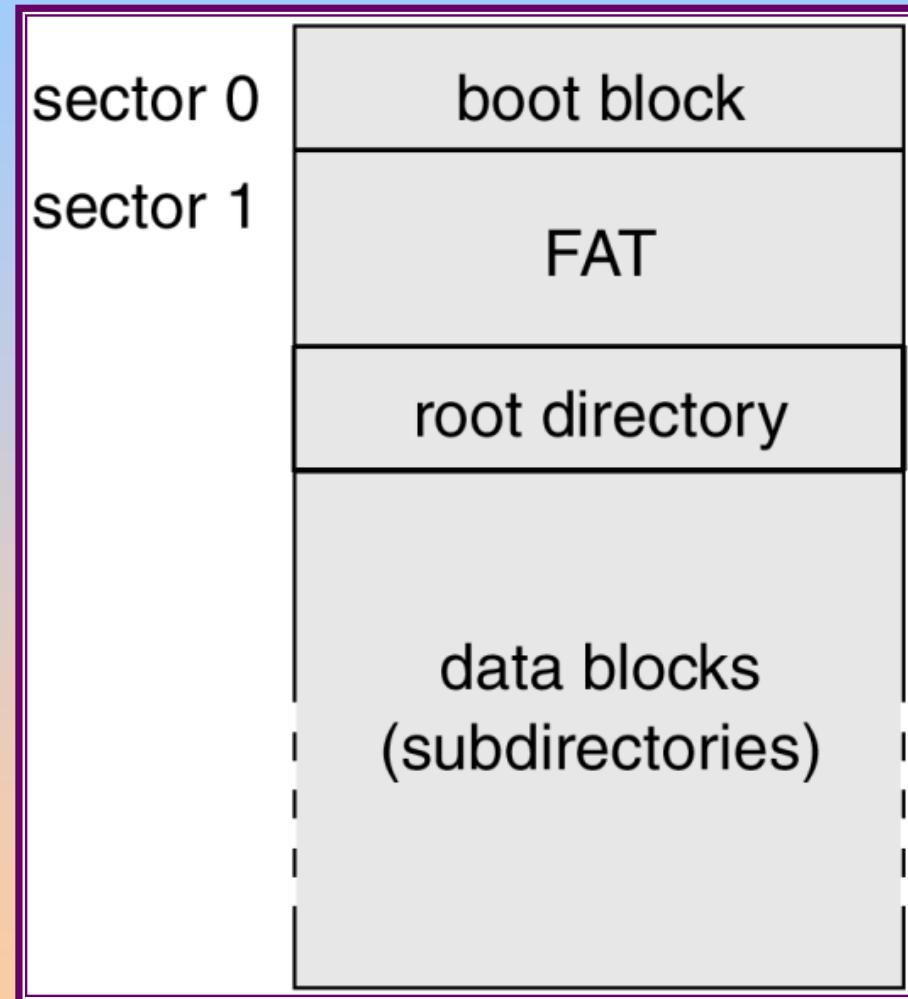




# Disk Management

- *Low-level formatting, or physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
  - To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
    - ☞ *Partition* the disk into one or more groups of cylinders.
    - ☞ *Logical formatting* or “making a file system”.
  - Boot block initializes system.
    - ☞ The bootstrap is stored in ROM.
    - ☞ *Bootstrap loader* program.
  - Methods such as *sector sparing* used to handle bad blocks.
- 

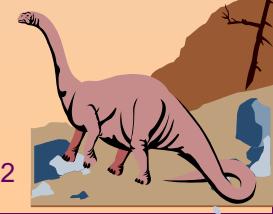
# MS-DOS Disk Layout





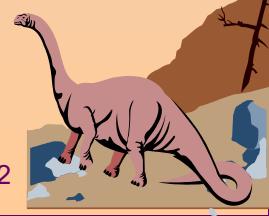
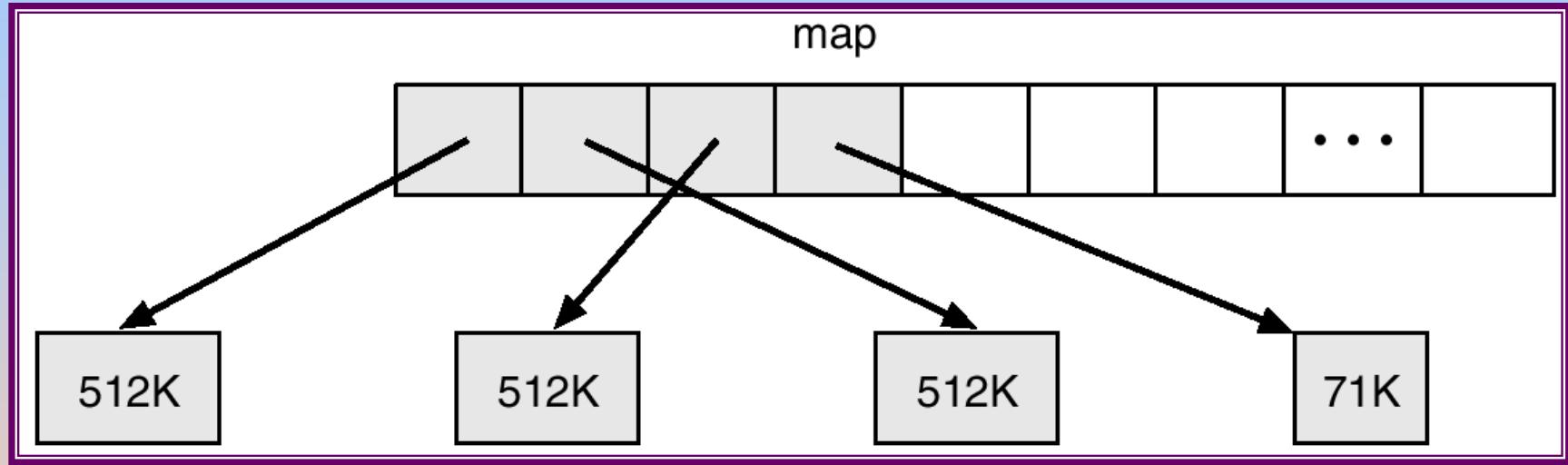
# Swap-Space Management

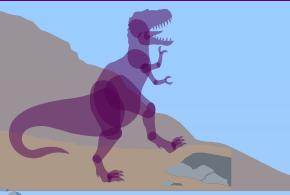
- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition.
- Swap-space management
  - ☞ 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment*.
  - ☞ Kernel uses *swap maps* to track swap-space use.
  - ☞ Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.



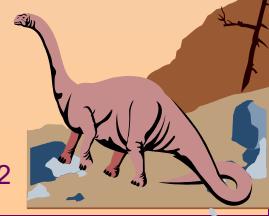
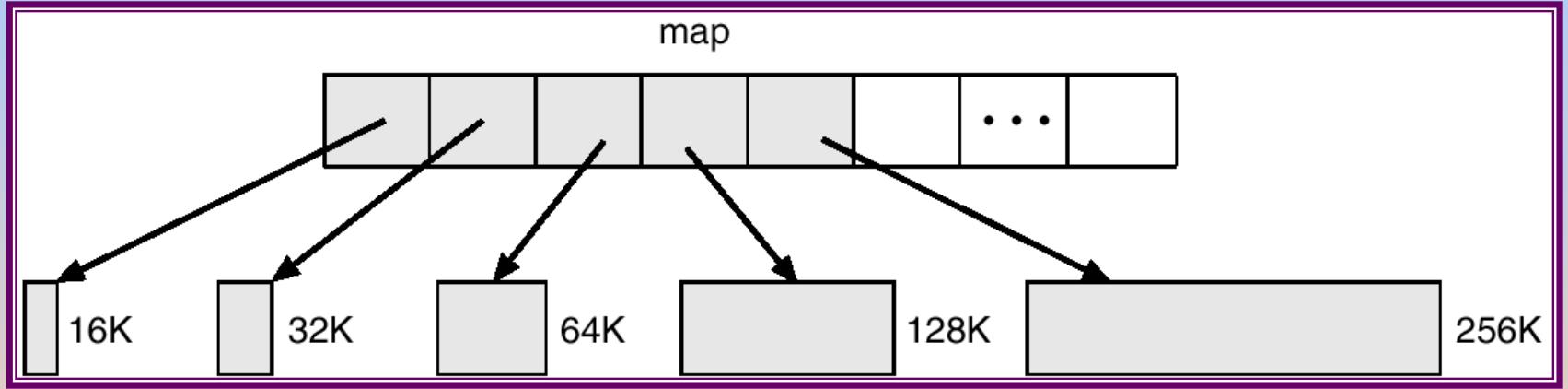


## 4.3 BSD Text-Segment Swap Map





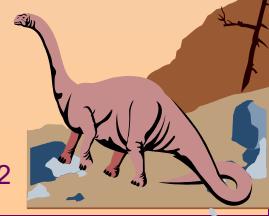
## 4.3 BSD Data-Segment Swap Map





# RAID Structure

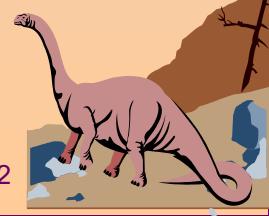
- RAID – multiple disk drives provides **reliability** via **redundancy**.
- RAID is arranged into six different levels.





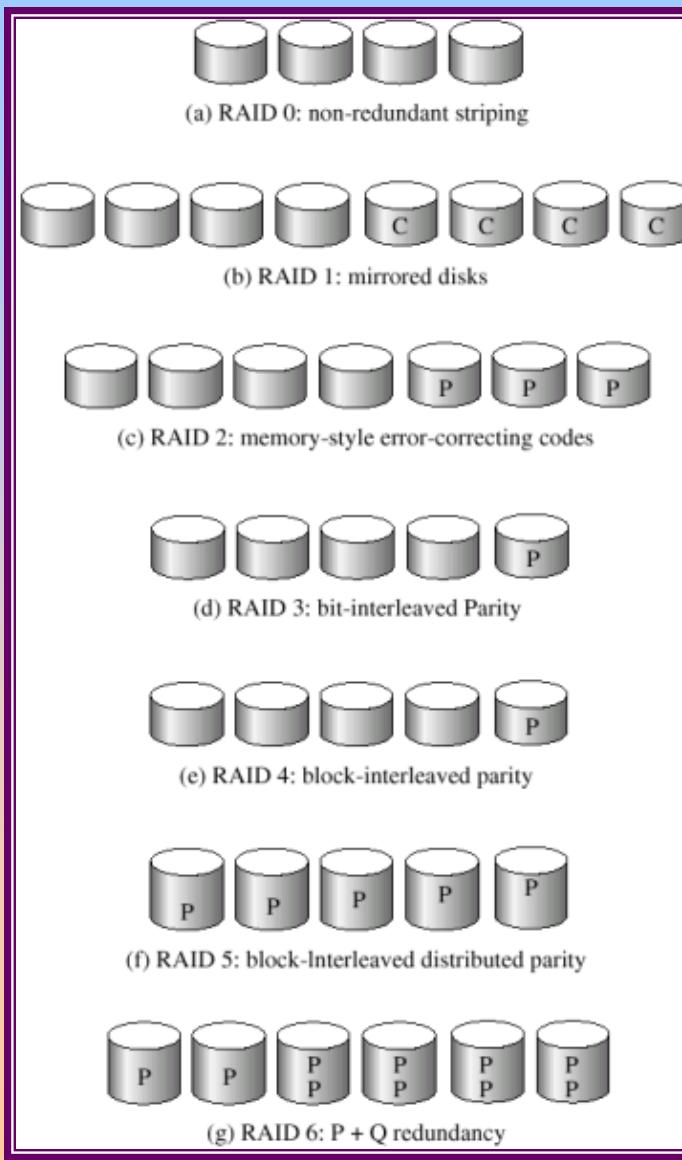
# RAID (cont)

- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
- Disk striping uses a group of disks as one storage unit.
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
  - ☞ *Mirroring or shadowing* keeps duplicate of each disk.
  - ☞ *Block interleaved parity* uses much less redundancy.



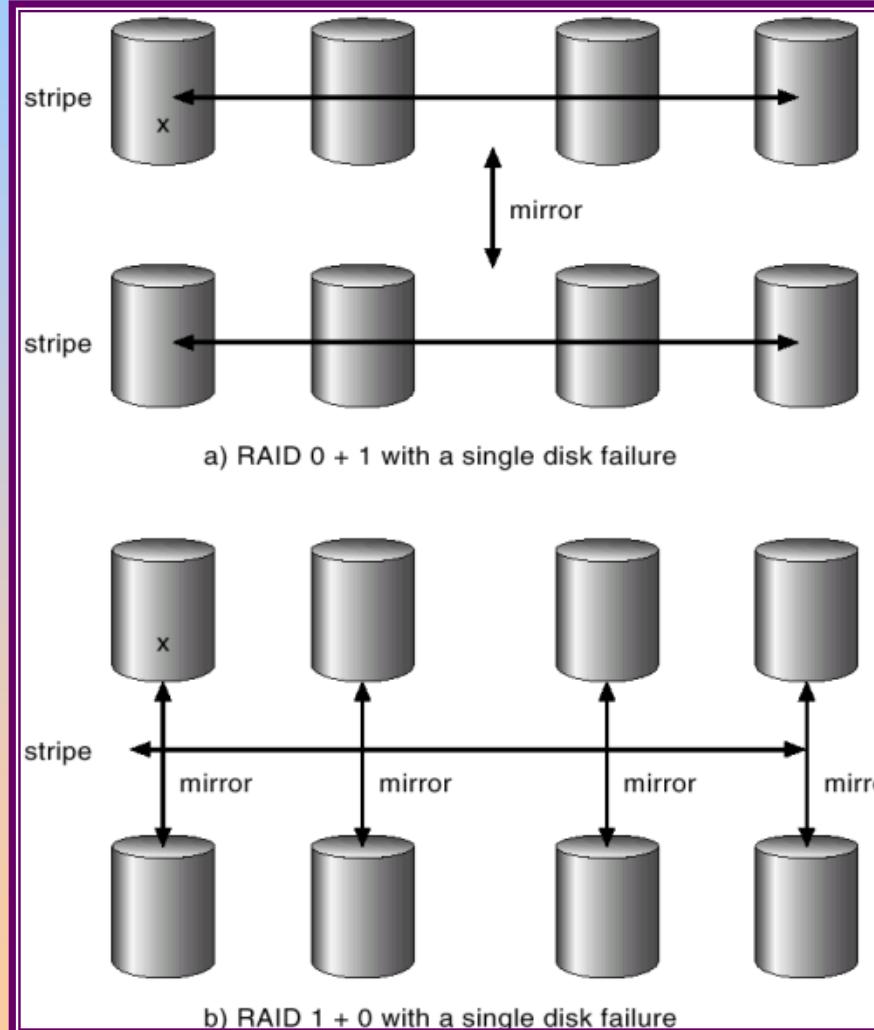


# RAID Levels



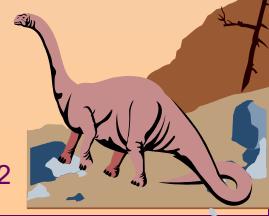


# RAID (0 + 1) and (1 + 0)



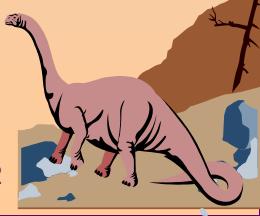
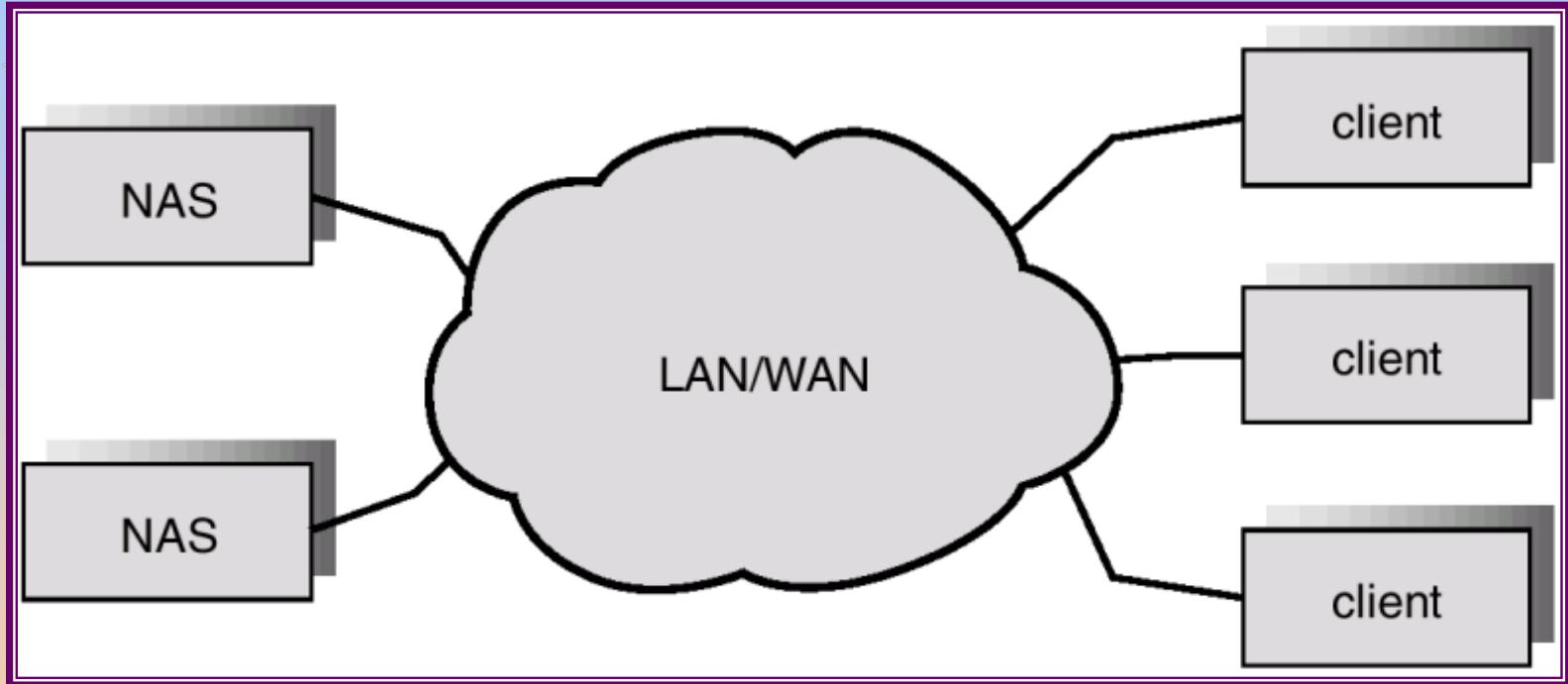


# Disk Attachment

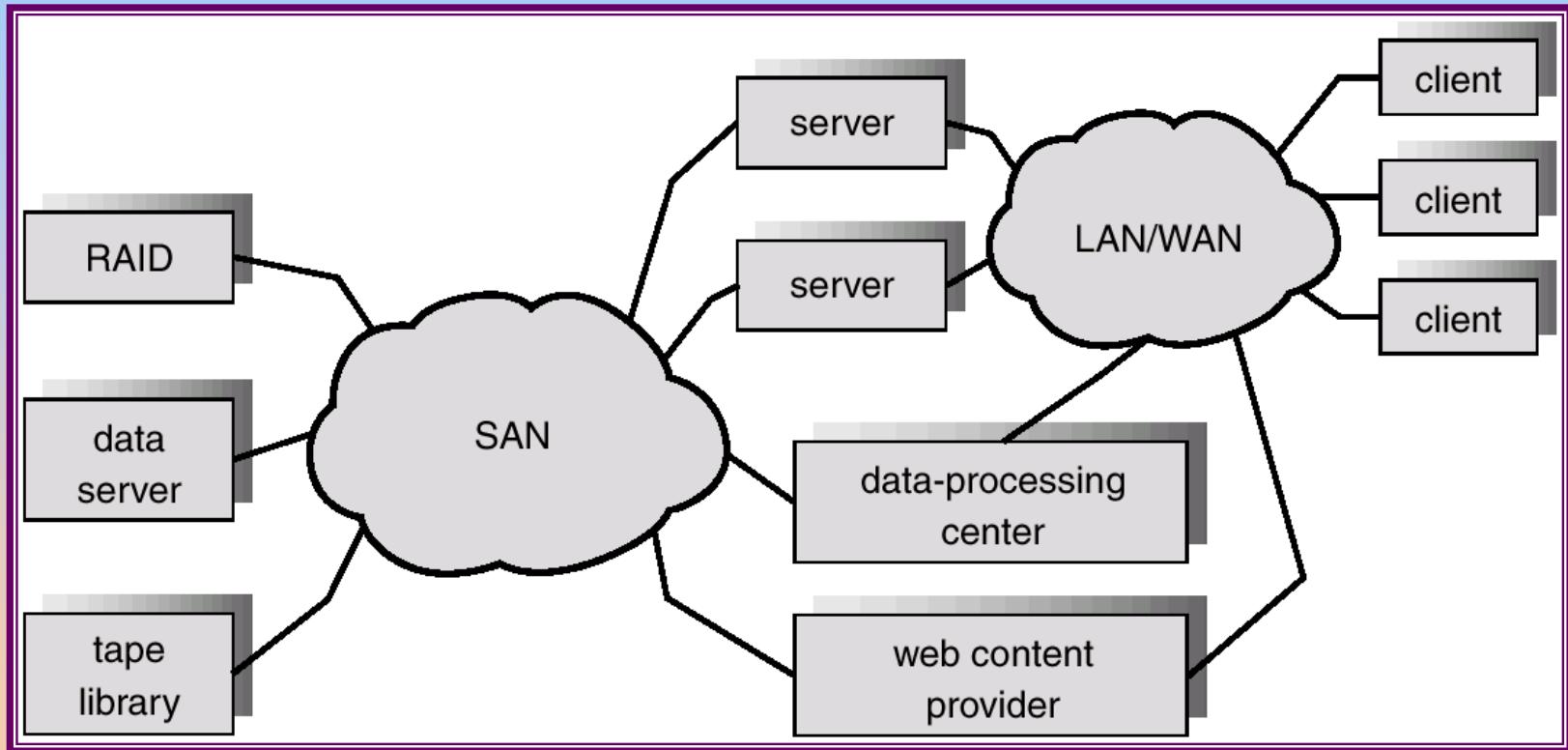
- Disks may be attached one of two ways:
    1. **Host attached** via an I/O port
    2. **Network attached** via a network connection
- 



# Network-Attached Storage

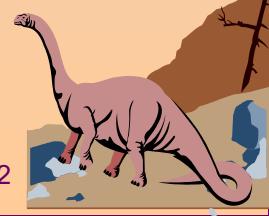


# Storage-Area Network



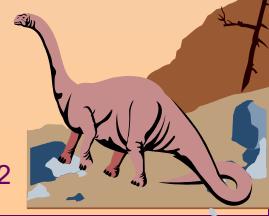


# Stable-Storage Implementation

- Write-ahead log scheme requires stable storage.
  - To implement stable storage:
    - ☞ Replicate information on more than one nonvolatile storage media with independent failure modes.
    - ☞ Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.
- 



# Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage.
  - Generally, tertiary storage is built using *removable media*
  - Common examples of removable media are floppy disks and CD-ROMs; other types are available.
- 



# Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.
    - ☞ Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.
    - ☞ Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.
- 



# Removable Disks (Cont.)

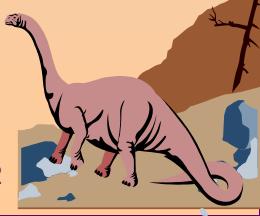
- A magneto-optic disk records data on a rigid platter coated with magnetic material.
  - ☞ Laser heat is used to amplify a large, weak magnetic field to record a bit.
  - ☞ Laser light is also used to read data (Kerr effect).
  - ☞ The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.
- Optical disks do not use magnetism; they employ special materials that are altered by laser light.





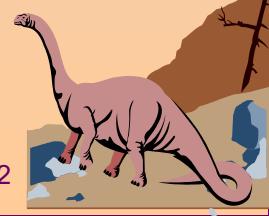
# WORM Disks

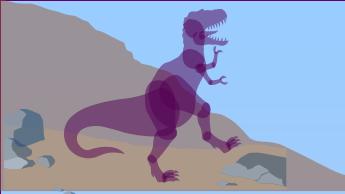
- The data on read-write disks can be modified over and over.
- WORM (“Write Once, Read Many Times”) disks can be written only once.
- Thin aluminum film sandwiched between two glass or plastic platters.
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.
- Very durable and reliable.
- *Read Only* disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded.





# Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
  - Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
  - Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
    - ☞ stacker – library that holds a few tapes
    - ☞ silo – library that holds thousands of tapes
  - A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.
- 



# Operating System Issues

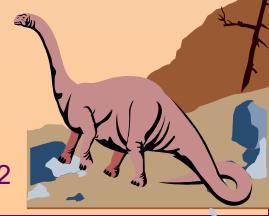
- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications
- For hard disks, the OS provides two abstraction:
  - ☞ Raw device – an array of data blocks.
  - ☞ File system – the OS queues and schedules the interleaved requests from several applications.

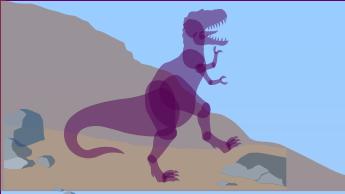




# Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device.
- Usually the tape drive is reserved for the exclusive use of that application.
- Since the OS does not provide file system services, the application must decide how to use the array of blocks.
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.

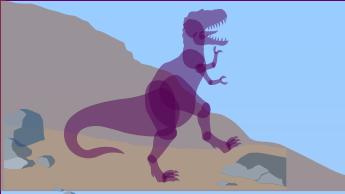




# Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- **locate** positions the tape to a specific logical block, not an entire track (corresponds to **seek**).
- The **read position** operation returns the logical block number where the tape head is.
- The **space** operation enables relative motion.
- Tape drives are “append-only” devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written.





# File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.





# Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system.
  - ☞ Small and frequently used files remain on disk.
  - ☞ Large, old, inactive files are archived to the jukebox.
- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

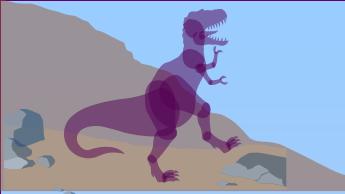




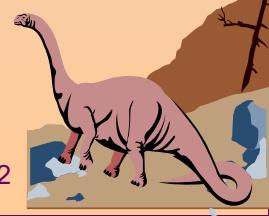
# Speed

- Two aspects of speed in tertiary storage are bandwidth and latency.
- Bandwidth is measured in bytes per second.
  - ☞ Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time.  
Data rate when the data stream is actually flowing.
  - ☞ Effective bandwidth – average over the entire I/O time, including **seek** or **locate**, and cartridge switching.  
Drive's overall data rate.





# Speed (Cont.)

- Access latency – amount of time needed to locate data.
    - ☞ Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds.
    - ☞ Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds.
    - ☞ Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk.
  - The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives.
  - A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.
- 



# Reliability

- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
  - An optical cartridge is likely to be more reliable than a magnetic disk or tape.
  - A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.
- 

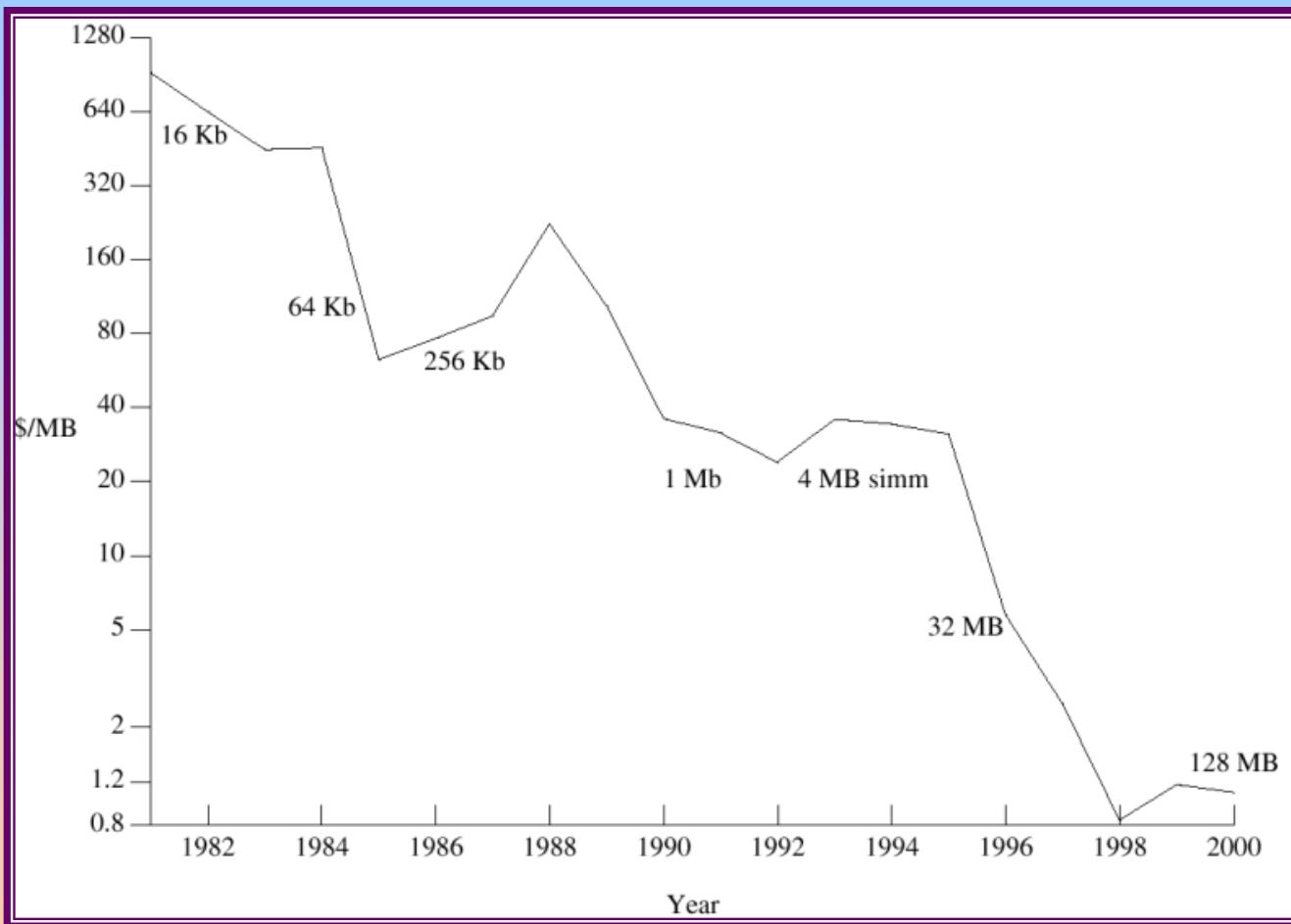


# Cost

- Main memory is much more expensive than disk storage
- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.
- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.

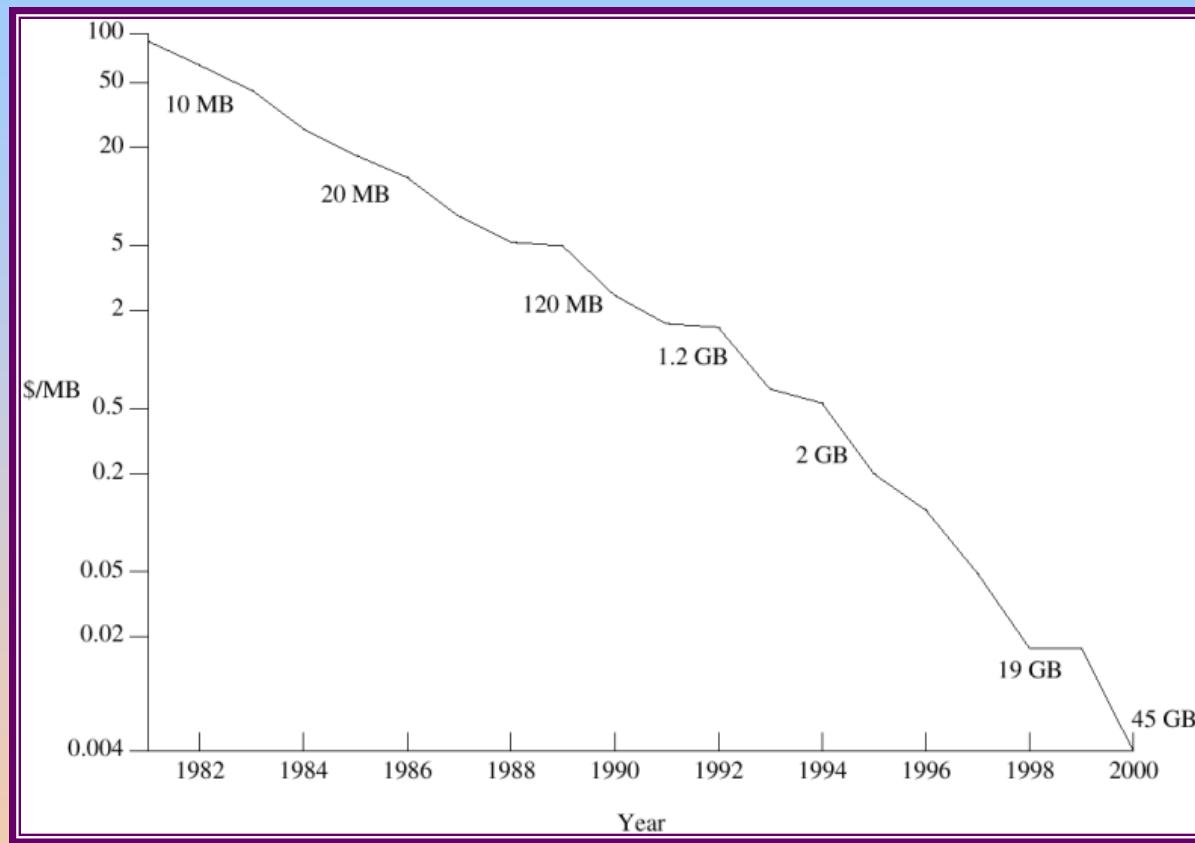


# Price per Megabyte of DRAM, From 1981 to 2000



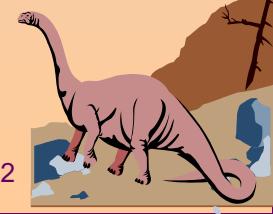
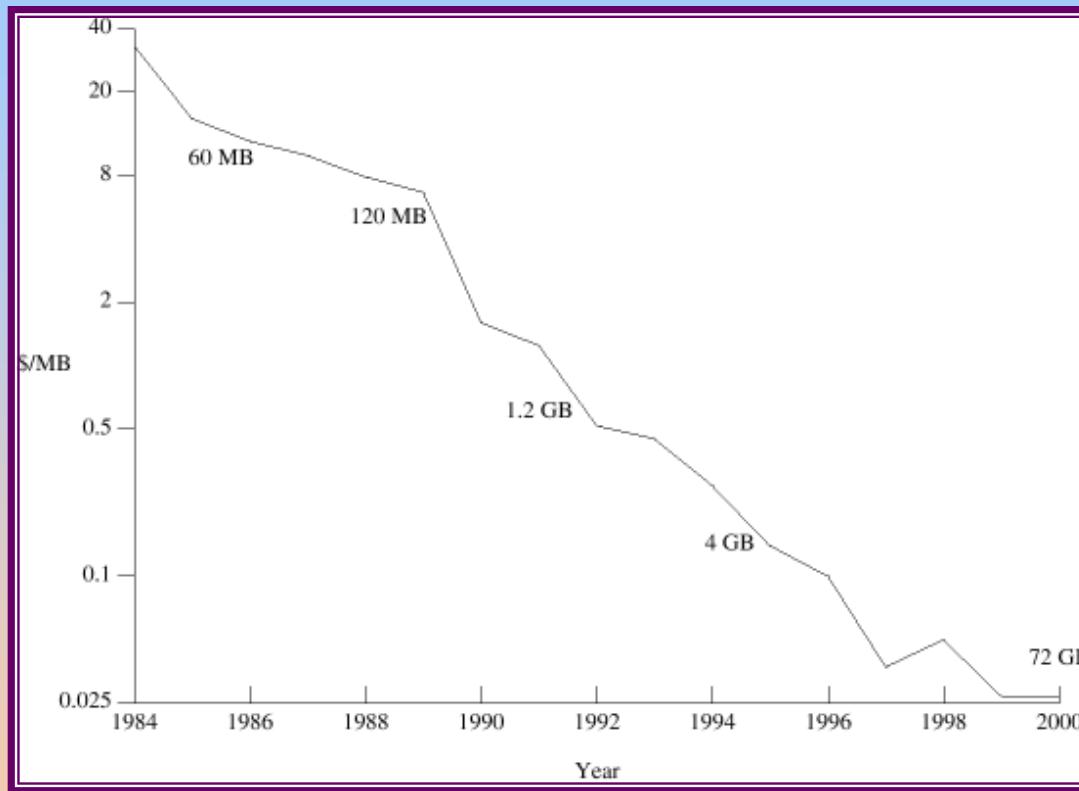


# Price per Megabyte of Magnetic Hard Disk, From 1981 to 2000

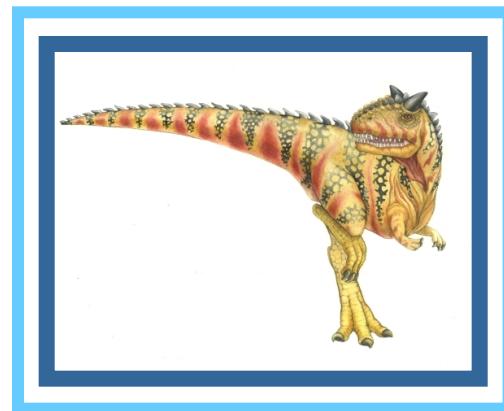




# Price per Megabyte of a Tape Drive, From 1984-2000



# Chapter 8: Main Memory Management





# Introduction

- Memory consists of large array of words or bytes each with its own address
- The CPU fetches instructions from memory according to the value of program counter
- A typical instruction-execution cycle fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory.
- After the instruction has been executed on the operands, results may be stored back in memory.





# Background

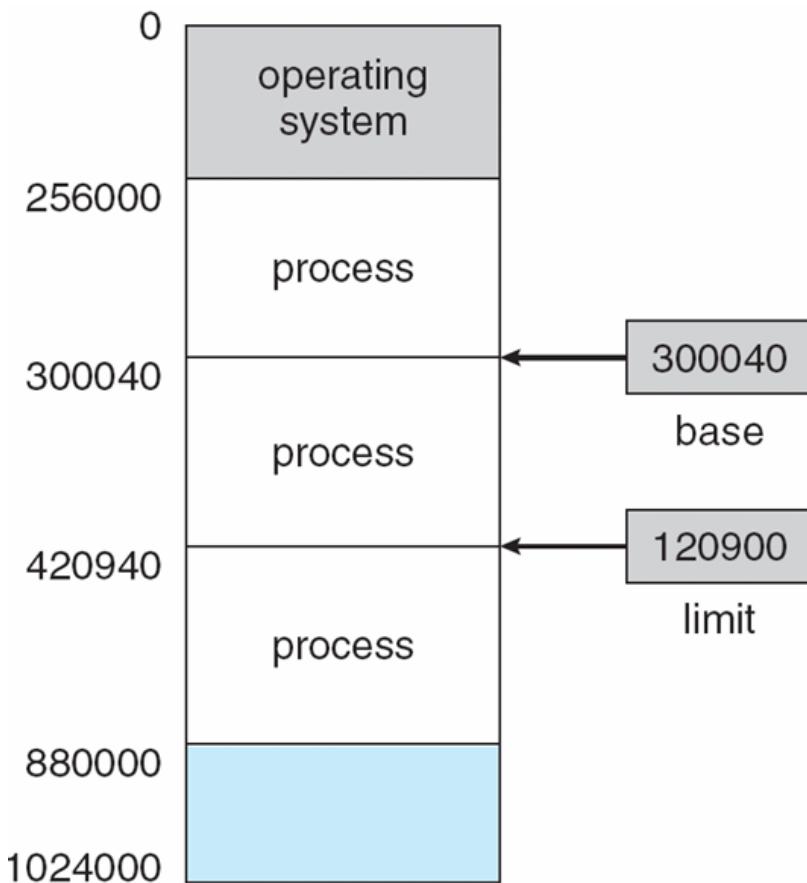
- Program must be brought (from disk) into memory and placed within a process for it to be run
- **Main memory and registers** are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock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





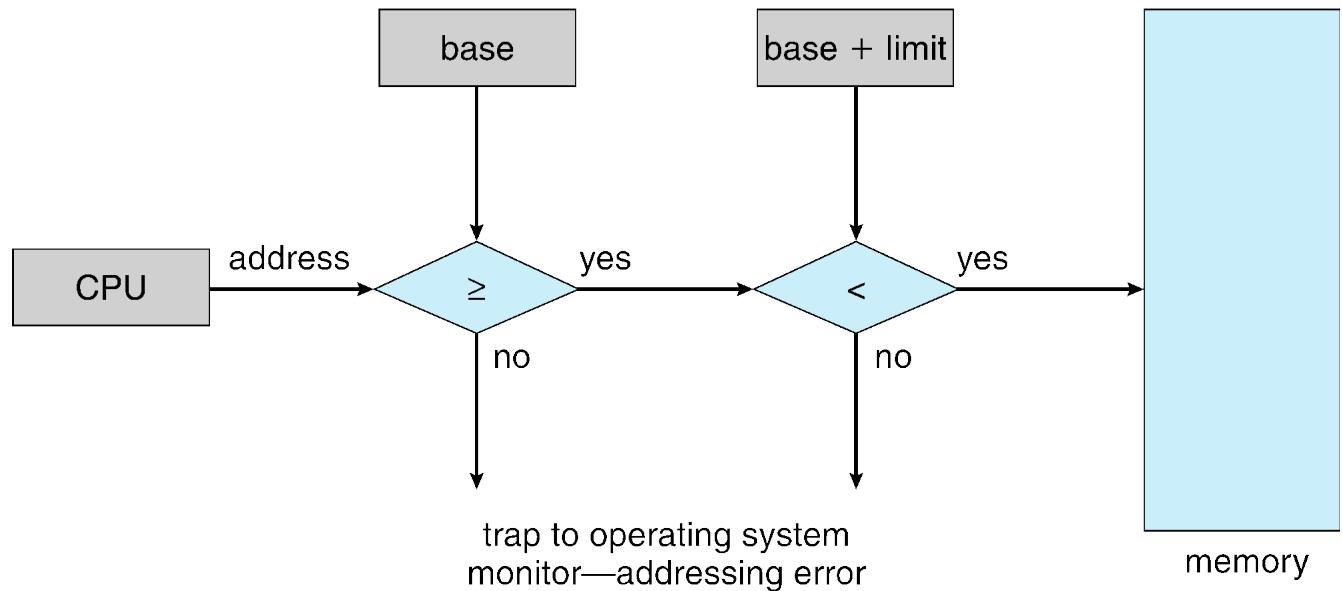
# Base and Limit Registers

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





# Hardware Address Protection





## Address representation

- Address are represented in different ways during different steps
  - **Source program:** addresses are symbolic like variables
  - **Compiler:** binds symbolic addresses to **relocatable**
    - ▶ Ex: 14 bytes from the beginning of this module
  - **Loader:** binds relocatable addresses to **absolute**, Ex. 74014





# Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at different steps
  - **Compile time:** if at some later time the starting location changes, then recompiling is necessary
  - **Load time:** binding relocatable addresses (from compile time) to absolute addresses
  - **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)





# Logical vs. Physical Address Space

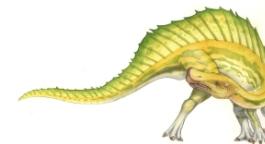
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses corresponding to the program logical addresses





# Memory-Management Unit (MMU)

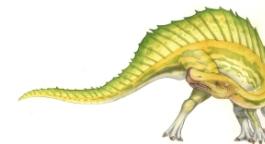
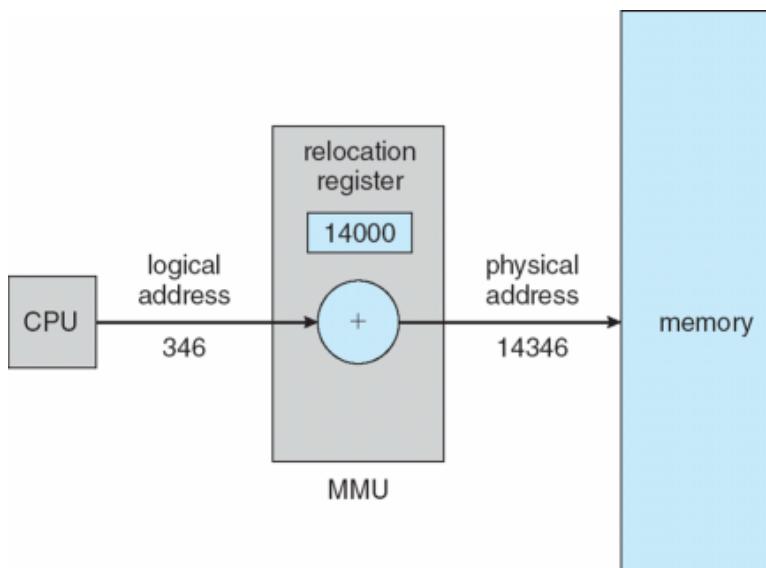
- Hardware device that at run time maps virtual to physical address
  - Usually a part of CPU
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* 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





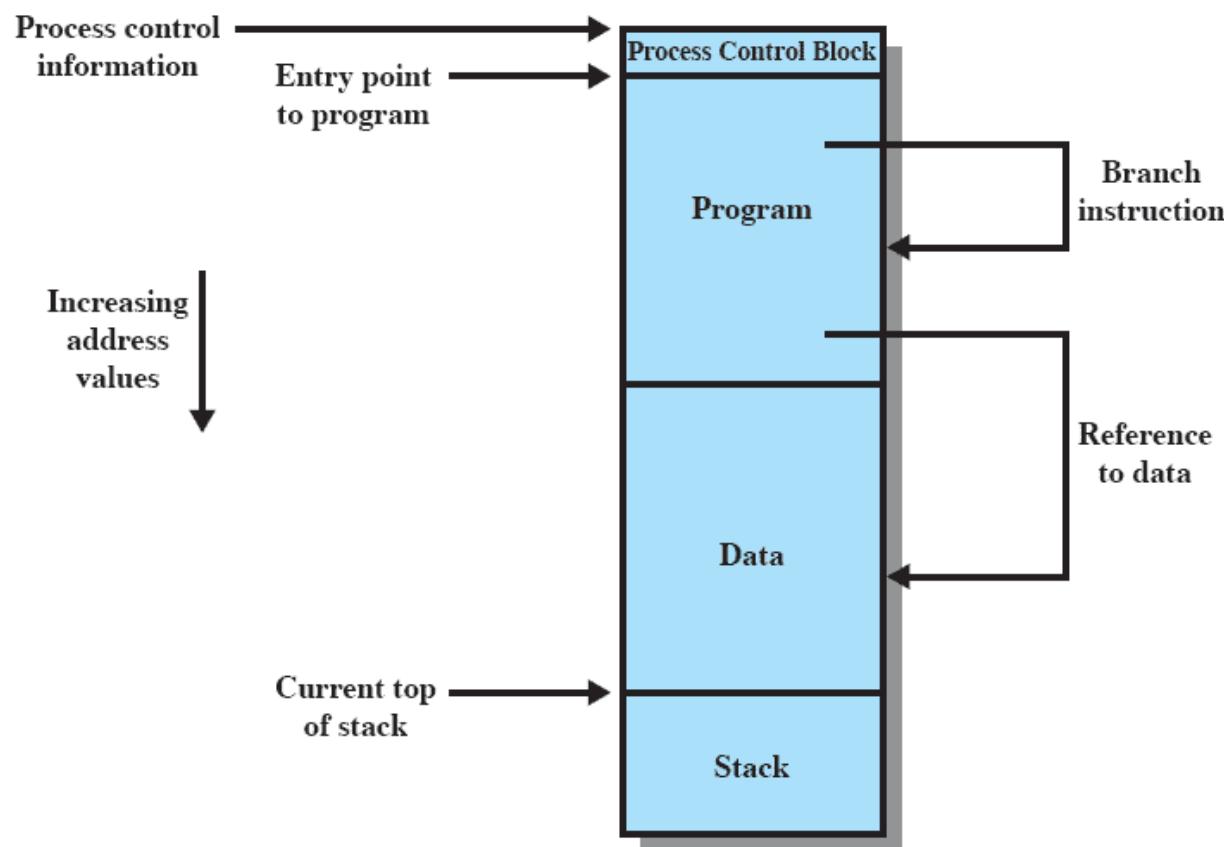
# Dynamic Linking

- **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 also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed





# Addressing



**Figure 7.1 Addressing Requirements for a Process**



# Memory Management Requirements

## Relocation

- Programmer cannot know where the program will be loaded in memory when it is executed
- A program may be *relocated* (often) in main memory due to swapping
- Memory references in code (for both instructions and data) must be translated to actual physical memory addresses

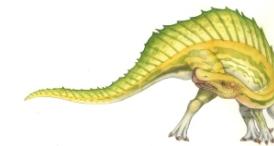




# Memory Management Requirements

## Protection

- Processes should not reference memory locations in another process without permission
- Cannot do this at compile time, because we do not know where the program will be loaded in memory
- address references must be checked at run time by hardware





# Memory Management Requirements

## Sharing

- must allow several processes to access a common portion of data or program without compromising protection
- cooperating processes may need to share access to the same data structure
- Program text can be shared by several processes executing the same program for a different user
- Saves memory over separate copy for each





# Memory Management Requirements

## Physical Organization

- Memory hierarchy: (several types of memory: from slow and large to small and fast.)
- main memory for program and data currently in use, secondary memory is the long term store for swapping and paging
- moving information between levels of memory is a major concern of memory and file management (OS)
  - should be transparent to the application programmer





# Simple Memory Management

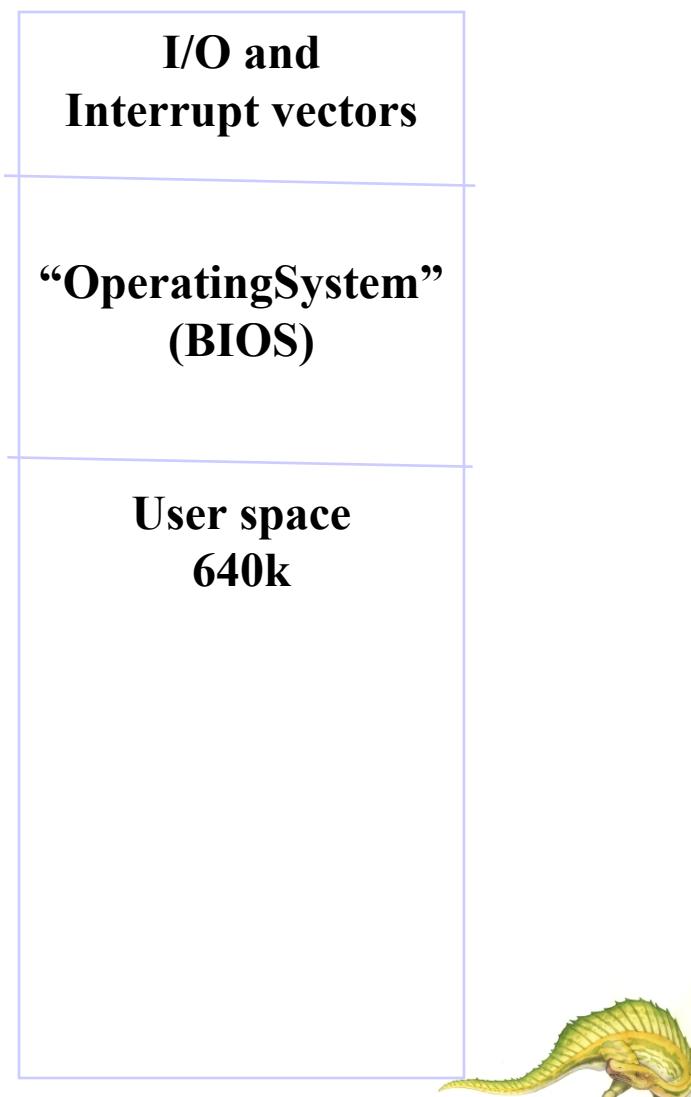
- We will look at the following simple memory management techniques :
  - fixed partitioning
  - dynamic partitioning





# Fixed Partitioning (Single Process)

- If only one process allowed (e.g. MS-DOS):
  - only one user partition
  - and one for the OS
- Not many decisions to make:
  - program either fits or it doesn't





# Fixed Partitioning

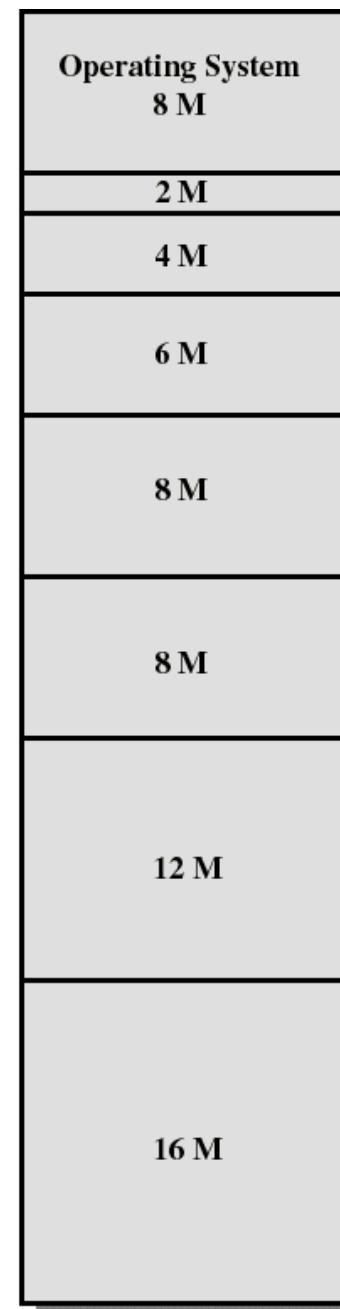
- Partition main memory into a set of non overlapping regions called partitions

- Partitions can be of equal or unequal sizes

..both pictures show partitioning of 64M main memory



Equal-size partitions



Unequal-size partitions



# Fixed Partitioning

- any program smaller than a partition can be loaded into the partition
- if all partitions are occupied, the operating system can swap a process out of a partition to make room
- a program may be too large to fit in a partition.





# Fixed Partitioning

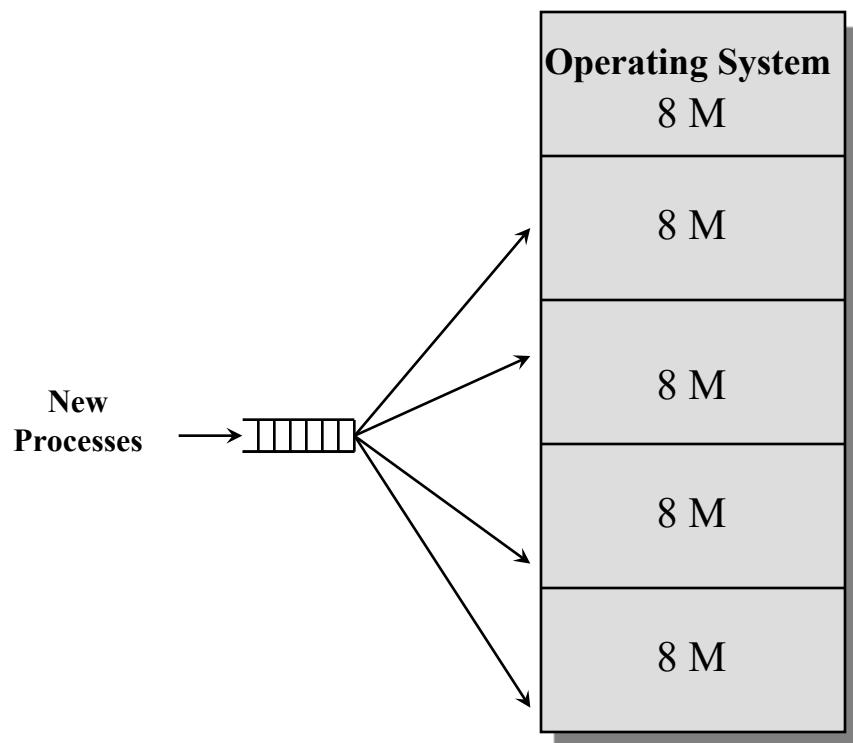
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition.
- This leaves holes of unused memory inside the partitions: *internal fragmentation*.
- Unequal-size partitions can help
  - put small programs in small partitions
  - but there will still be holes...
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)
- Number of partitions limit the number of active processes





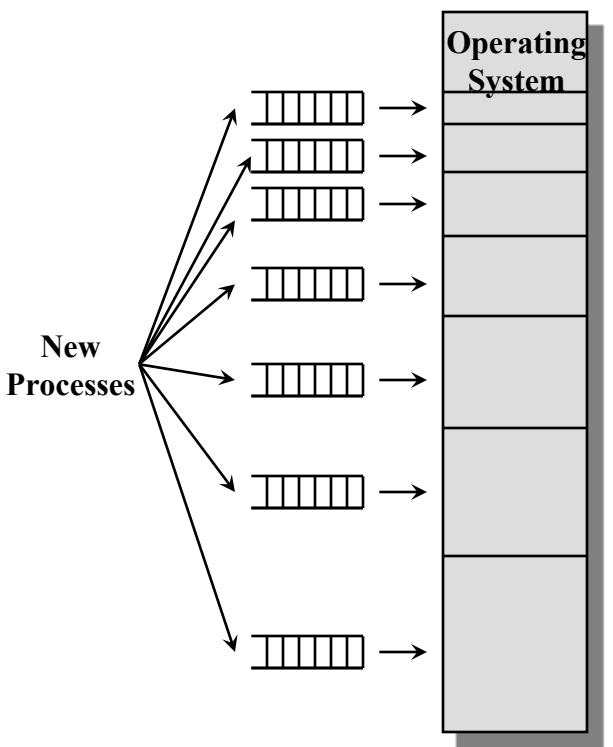
# Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.





# One Process Queue per Partition



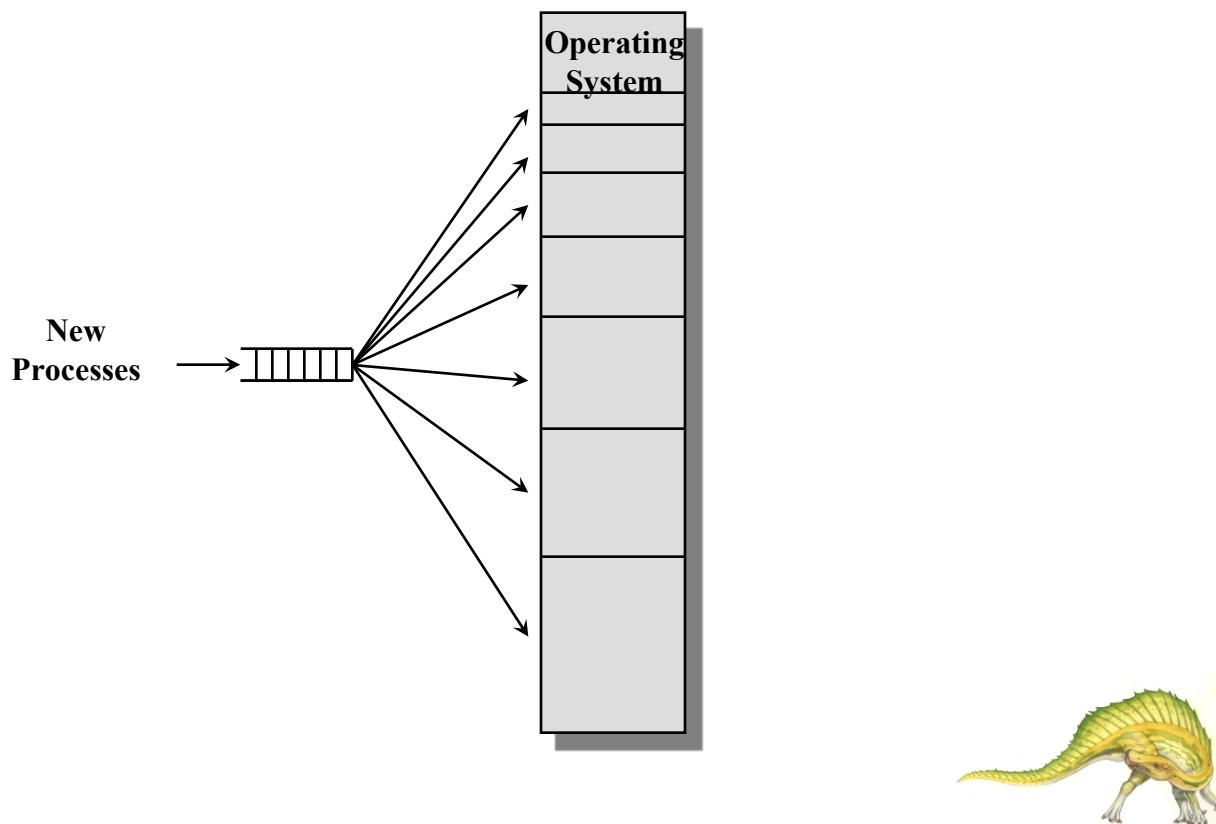
Problem with separate queue is if the bigger partition queue is empty and other queues process are waiting, the bigger partition remains unused.





# One Common Process Queue

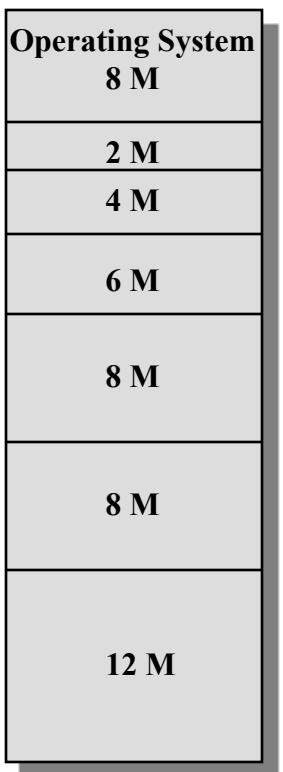
- When its time to load a process into main memory the smallest available partition that will hold the process is selected





# Fixed Partitioning

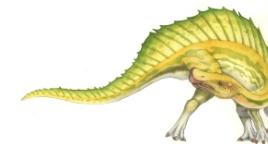
- Unequal-size partitions
  - lessens the problem with equal-size partitions





# Placement Algorithm with Partitions

- Equal-size partitions
  - because all partitions are of equal size, it does not matter which partition is used
- Unequal-size partitions
  - can assign each process to the smallest partition within which it will fit
  - queue for each partition
  - processes are assigned in such a way as to minimize wasted memory within a partition





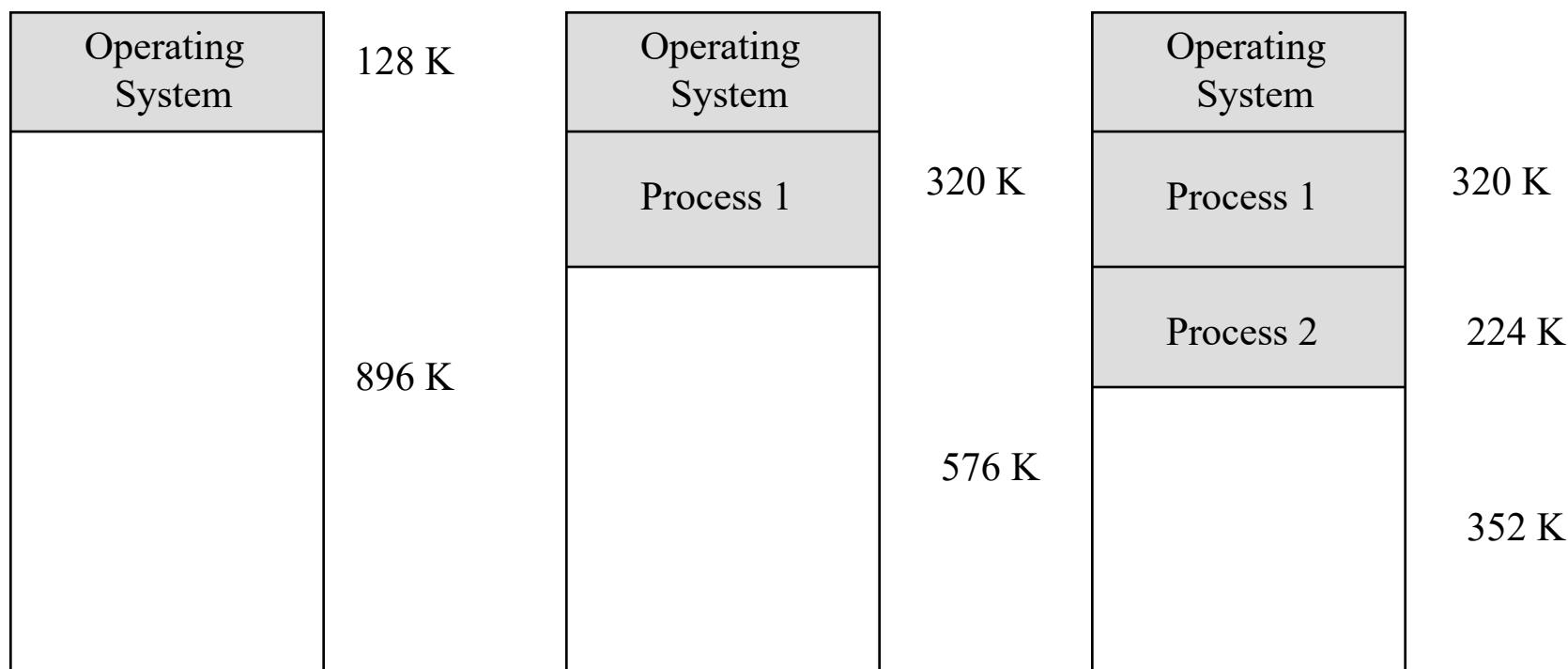
## Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called **external fragmentation**
- Must use **compaction** to shift processes so they are contiguous and all free memory is in one block



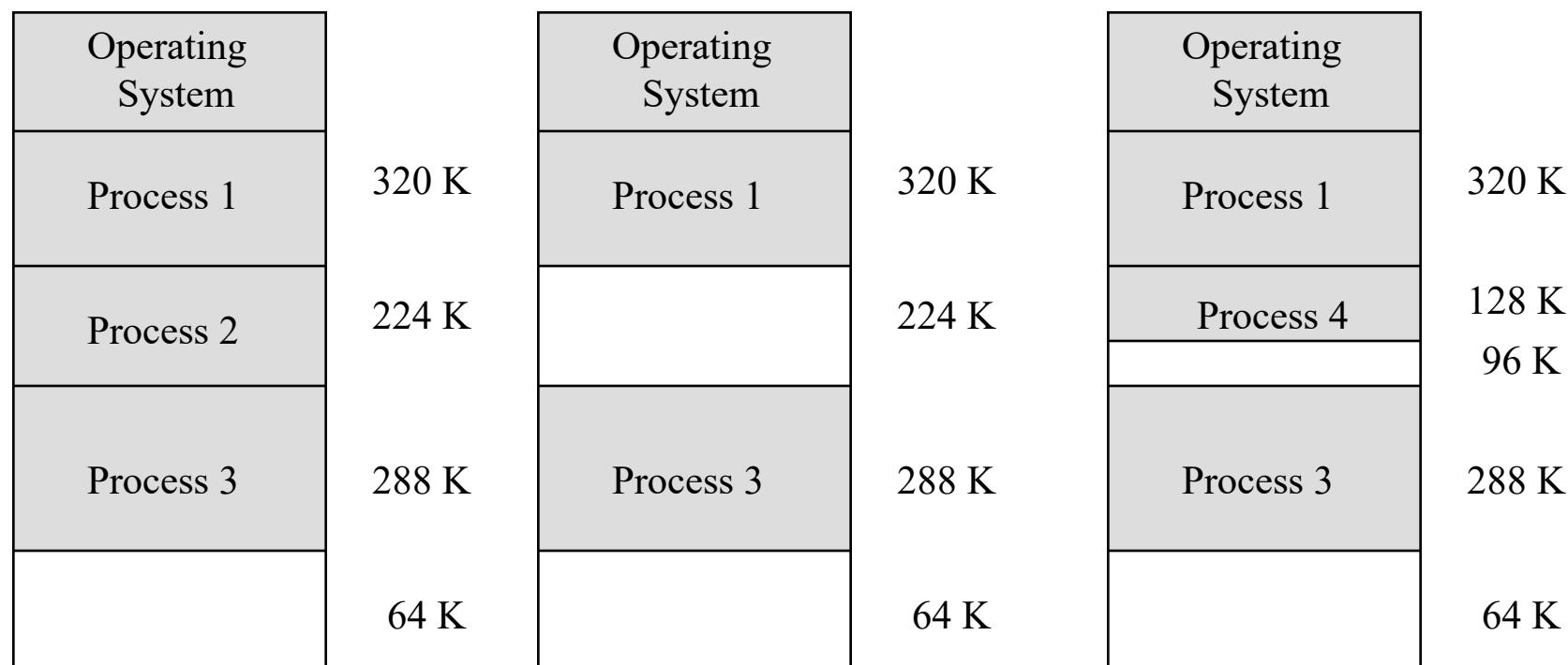


# Example Dynamic Partitioning



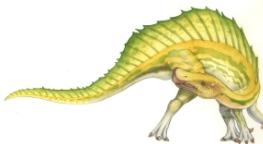
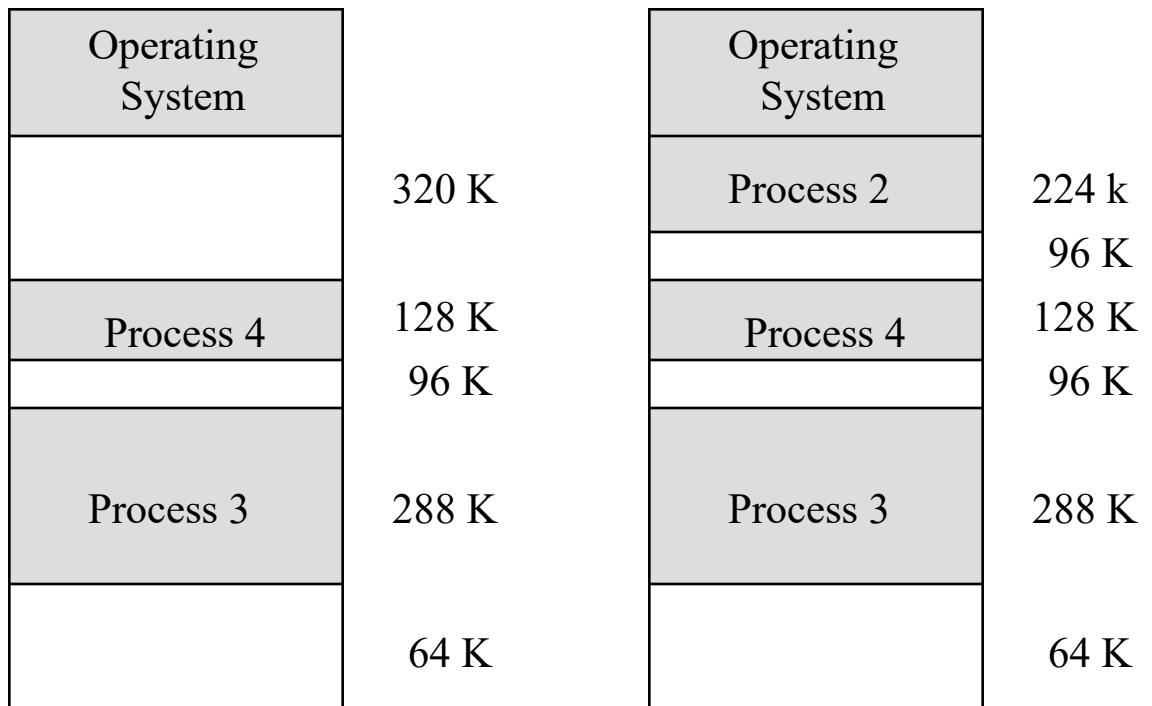


# Example Dynamic Partitioning





# Example Dynamic Partitioning





# Dynamic Partitioning Placement Algorithm

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - chooses the block that is closest in size to the request
  - since smallest block is found for process, the smallest amount of fragmentation is left - memory compaction must be done more often





# Dynamic Partitioning Placement Algorithm

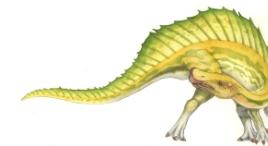
- First-fit algorithm
  - starts scanning memory from the beginning and chooses the first available block that is large enough.
- Next-fit
  - starts scanning memory from the location of the last placement and chooses the next available block that is large enough
- Worst-fit
  - Allocate biggest free partition that is available to the incoming process
  - Compaction is required to obtain a large block at the end of memory





# Dynamic Partitioning Placement Algorithm

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing both time and storage utilization.
- First fit generally faster





# Fragmentation

- **External Fragmentation in dynamic partitioning** – total memory space exists to satisfy a request, but it is not contiguous
  - Solution - compaction
- **Internal Fragmentation in fixed partitioning** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used





# Solution to Fragmentation problem

- 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





# Buddy Systems

- Both fixed and dynamic partitioning schemes have drawbacks
- A fixed partitioning scheme limits the number of active processes and may use space inefficiently, if there is a poor match between available partition sizes and process sizes
- A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction.
- An interesting compromise is buddy system.
- In buddy systems, memory blocks are available in sizes of powers of 2 words





# Buddy System

- Entire space available is treated as a single block of  $2^U$
- If a request of size  $s$  where  $2^{U-1} < s \leq 2^U$ 
  - entire block is allocated
- Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to  $s$  is generated





# Example of Buddy System

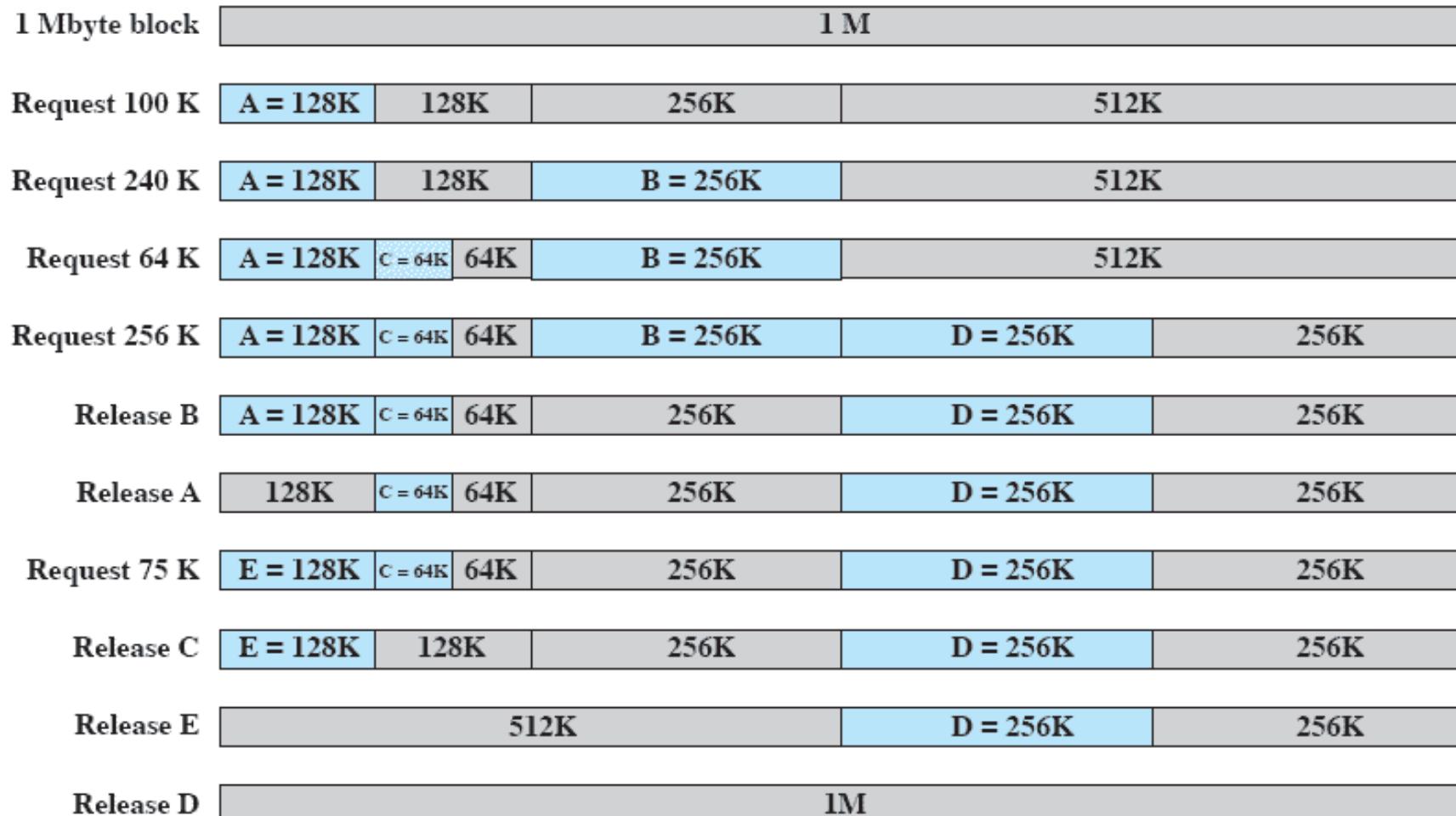
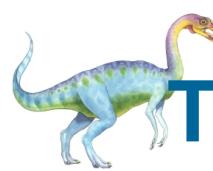


Figure 7.6 Example of Buddy System



# Tree Representation of Buddy System

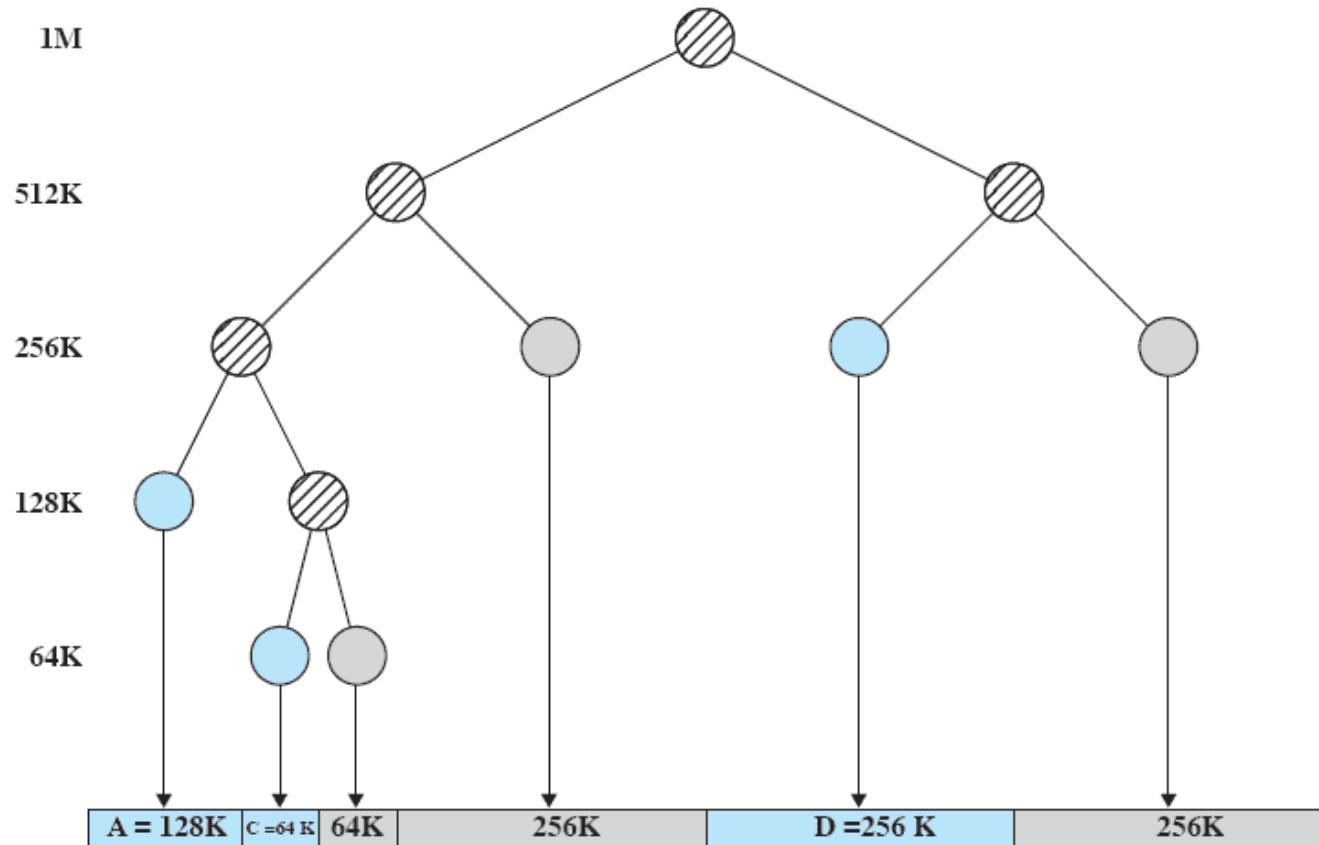


Figure 7.7 Tree Representation of Buddy System





# Swapping

A process needs to be in memory to be executed .

- \* however , it may be temporarily swapped out of memory into a backing store .
- \* the process can later be restarted by swapping it back into memory .
- \* Backing is generally a fast disk. Speed is important .

**Swapping example :-**

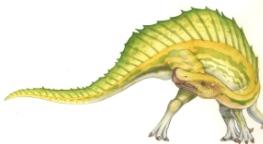
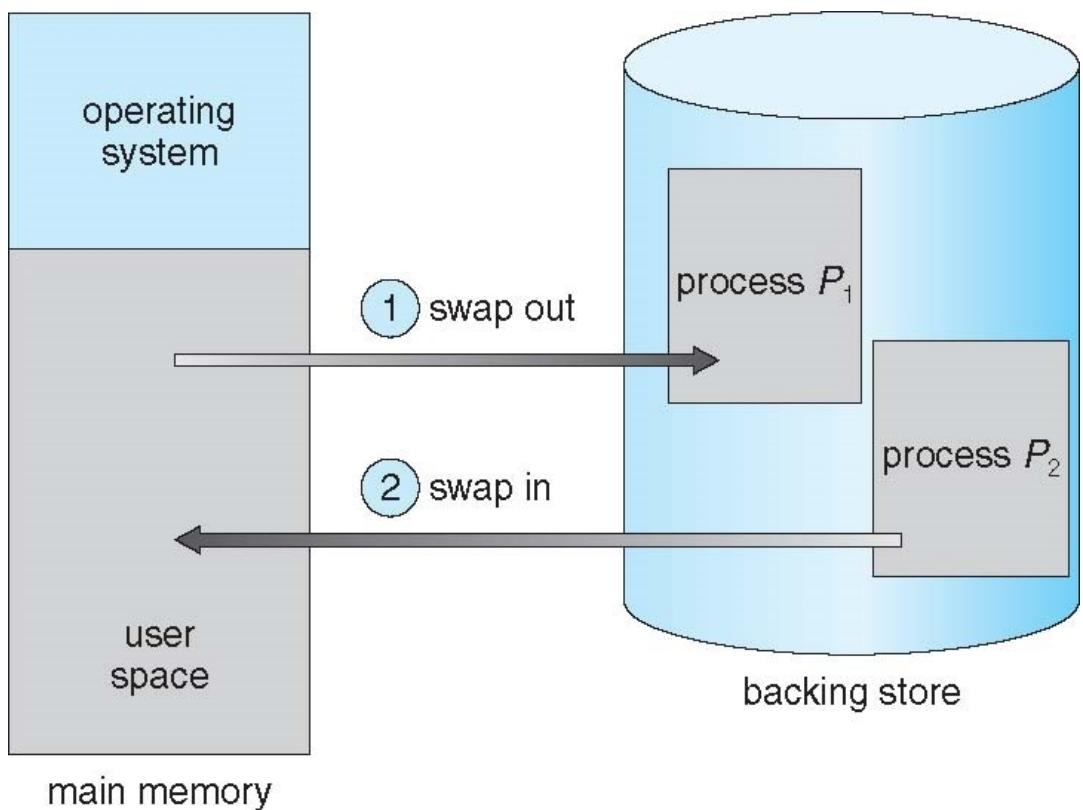
When a process block or timeout:-

- 1- swap it out to disk
- 2- swap in a new (ready ) process from disk to memory
- 3- dispatch the new process .





# Schematic View of Swapping





# Paging

- A memory management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store from which most of the previous memory management schemes suffered.
- When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.
- The fragmentation problems in main memory are prevalent with backing store, except that access is much slower, so compaction impossible.
- Traditionally paging has been handled by hardware, but recent designs have implemented paging by closely integrating hardware and OS.





# Paging

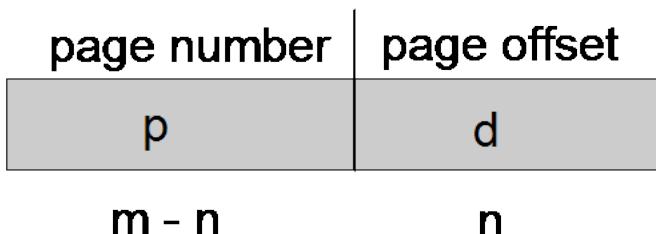
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- All free frames are also kept track of in frame table by OS.
- The backing store is also divided into fixed sized blocks of the same size as the memory 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
- **Still have Internal fragmentation**
- **But no external fragmentation**





# Address Translation Scheme

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

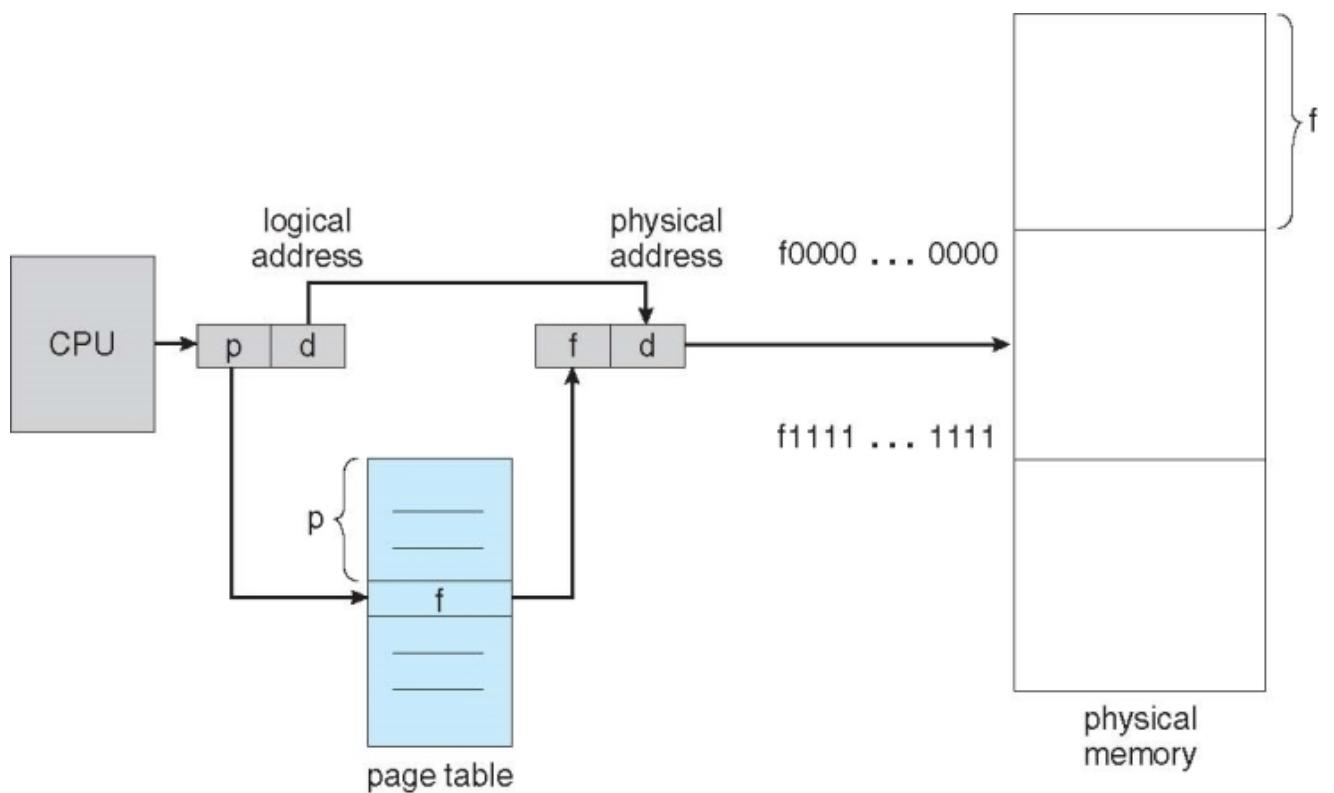


- For given logical address space  $2^m$  and page size  $2^n$



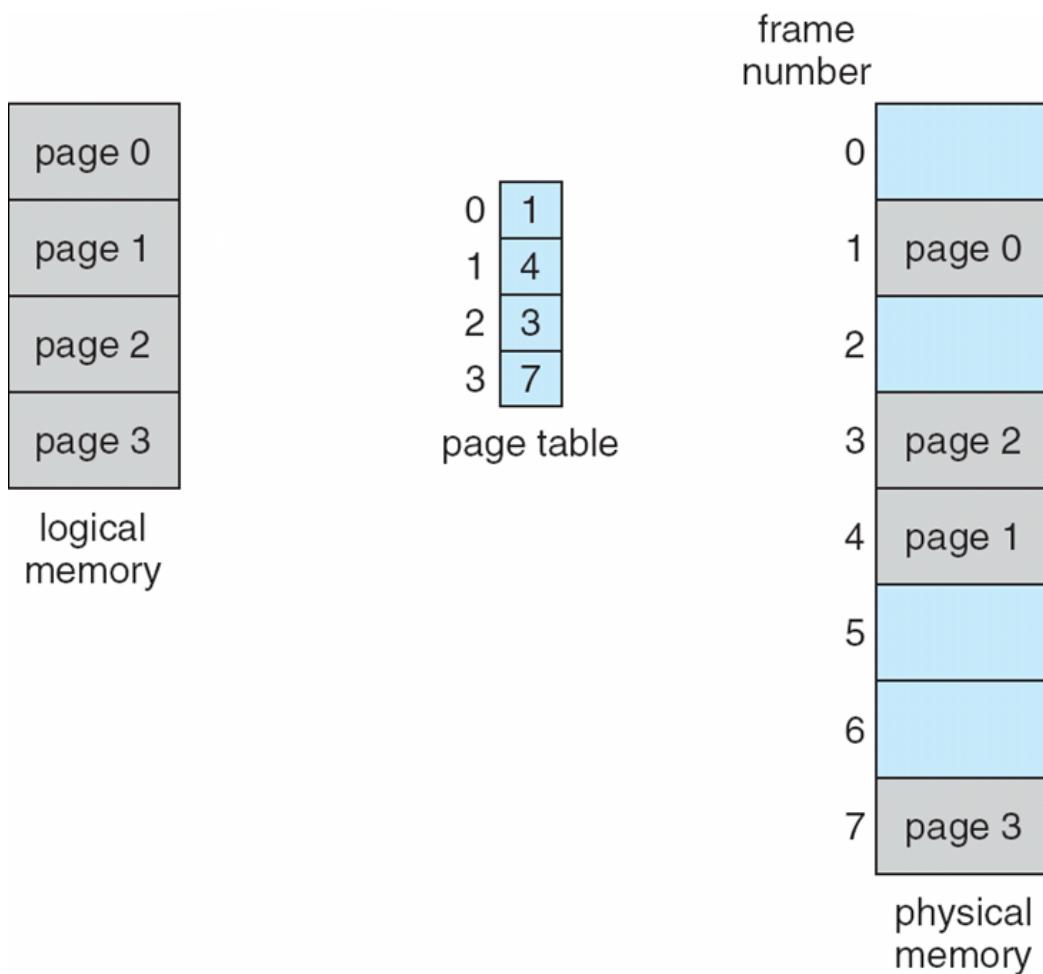


# Paging Hardware





# Paging Model of Logical and Physical Memory





# Paging Example

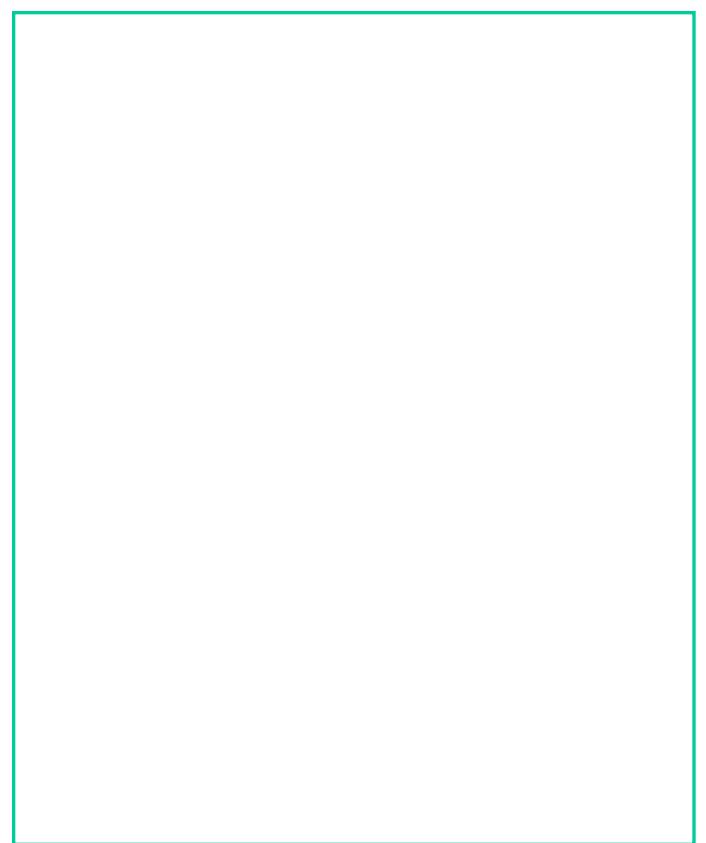
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	5
1	6
2	1
3	2
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	



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





# Paging

- Paging itself is a form of dynamic relocation
- Every logical address is bound by paging hardware to some physical address
- Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.
- When we are using paging scheme, we have no external fragmentation.
- Any free frame can be allocated to a process that needs it.
- But we may have internal fragmentation as shown in the next slide.





# Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes = frame size
  - 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
  - On average fragmentation =  $1 / 2$  frame size
  - So small frame sizes desirable?
    - But each page table entry takes memory to track
  - Page sizes growing over time
    - ▶ Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory





# Paging

- If we have **small page sizes**, to reduce internal fragmentation, overhead arises as we may have many entries in the page table for more pages.
- And also disk I/O is more efficient only if we have large pages and the data being transferred is larger.
- Each page table entry is usually 4 bytes long, but that size can vary as well.
- A 32 bit entry can point to  $2^{32}$  physical page frames and if frame is 4KB of size, then it can address  $2^{36}$  bytes or 64GB of physical memory.





# Paging

- When a process arrives in the system to be executed, its size expressed in pages is examined.
- Each page of process needs one frame.
- If process requires ‘n’ pages at least n frames must be available in memory.
- When each page loaded in a memory frame an entry is put into page table for that process.
- The address translation hardware takes care of converting CPU generated logical address to physical address since the pages are not contiguous.
- The OS maintains a **frame table** which has one entry for each physical page frame indicating whether the frame is free or allocated and, if it is allocated to which page of which process or processes.





## Paging: Example

- If the page size is equal to 1KB, in logical address 0000010111011110, how many bits are required for the page number?

- 1 KB = 10 bits for page offset
- 0000010111011110       16-10=6 bits for the page #  
16 bits





## Paging: Example

- There are 4 pages in a logical address space and each page has 2 words. If these pages are assigned to a physical address space with 8 frames, how many bits are required for physical and logical addresses?
  - Logical address: 4 pages = 2bits, 2 words=1 bit => 3 bits
  - Physical address: 8 frames=3 bits, 2 words=1 bit => 4 bit





# 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

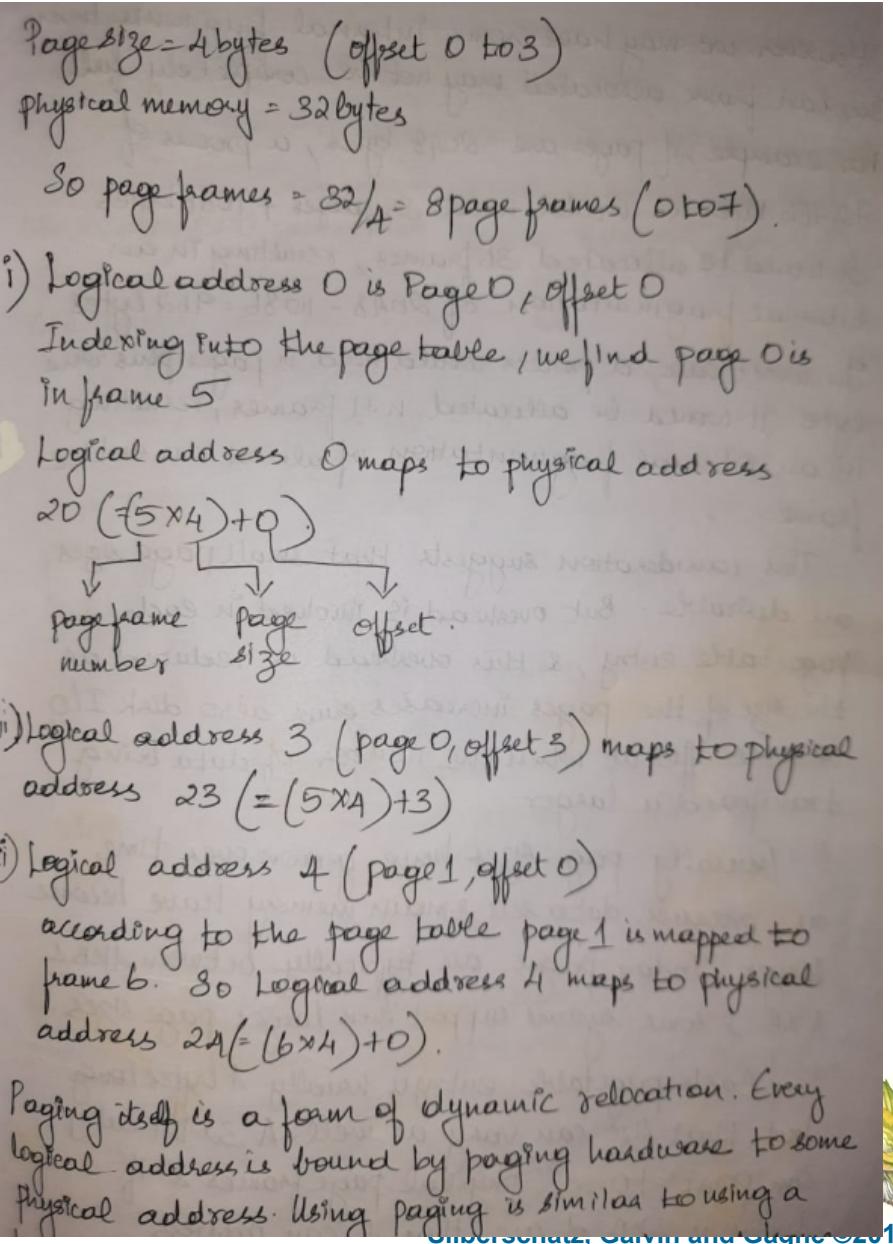
logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i
8	j
12	k
16	l
20	m
24	n
28	o

physical memory





# Hardware Support for Page Table

- Each OS has its own method of storing page tables.
- Most allocate a page table for each process
- A pointer to page table is stored with the other register values in PCB.
- When dispatcher is told to start a process, all register values are restored from PCB





# Implementation of Page Table

## 1) Method 1

- Can be done with a set of dedicated registers
- Very efficient in case of speed
- Dispatcher loads these register values also from PCB
- But it is suitable only if the page table size is reasonably small





# Implementation of Page Table

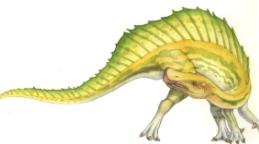
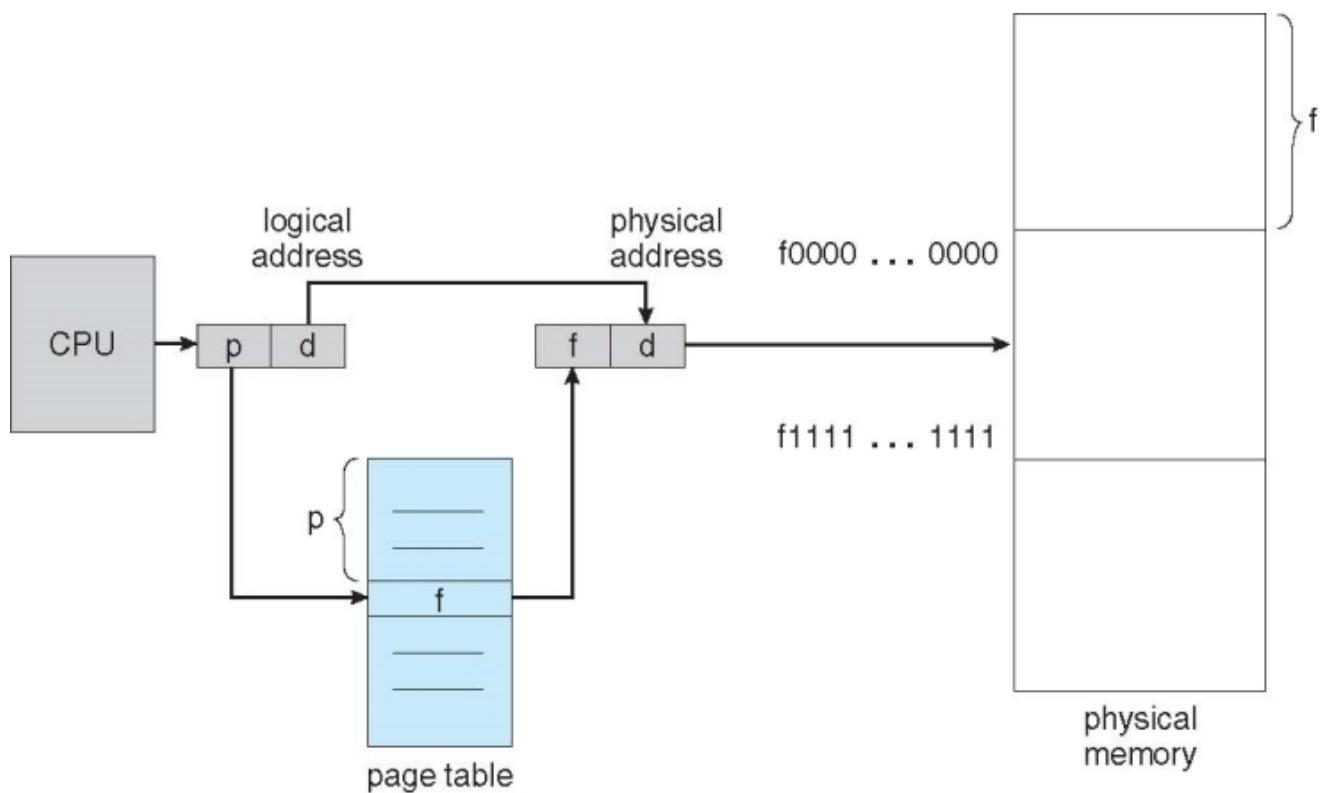
## 2) Method 2

- Page table is kept in main memory if its size is large say (1 million entries).
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table





# Paging Hardware





# Translation-lookaside Buffer (TLB)

- 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)**
- Each entry in the TLB consists of two parts: a key and a value
- When the associative memory is presented with an item(page no), the item is compared with all (entries)keys simultaneously and if a match is found, then the value(frame number) is returned.
- If no match found, then page table must be referenced for frame number.





# TLB Operation

- Given a virtual address,
  - processor examines the TLB
- If page table entry is present (TLB hit),
  - the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss),
  - the page number is used to index the process page table





# Looking into the Process Page Table

- First checks if page is already in main memory
  - If not in main memory a page fault is issued
  - Or else the TLB is updated to include the new page entry
- If the TLB is already full of entries, the OS must select one for replacement (LRU to random)
- Some TLBs store **address-space identifiers(ASID)** in each TLB entry
- An ASID uniquely identifies each process and is used to provide address space protection for that process.
- When TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.
- If ASIDs do not match, the attempt is treated as TLB miss.





- If no support for ASID then, every time a new page table is selected, the TLB must be flushed, to ensure that the next executing process does not use the wrong translation information.





# Translation Lookaside Buffer

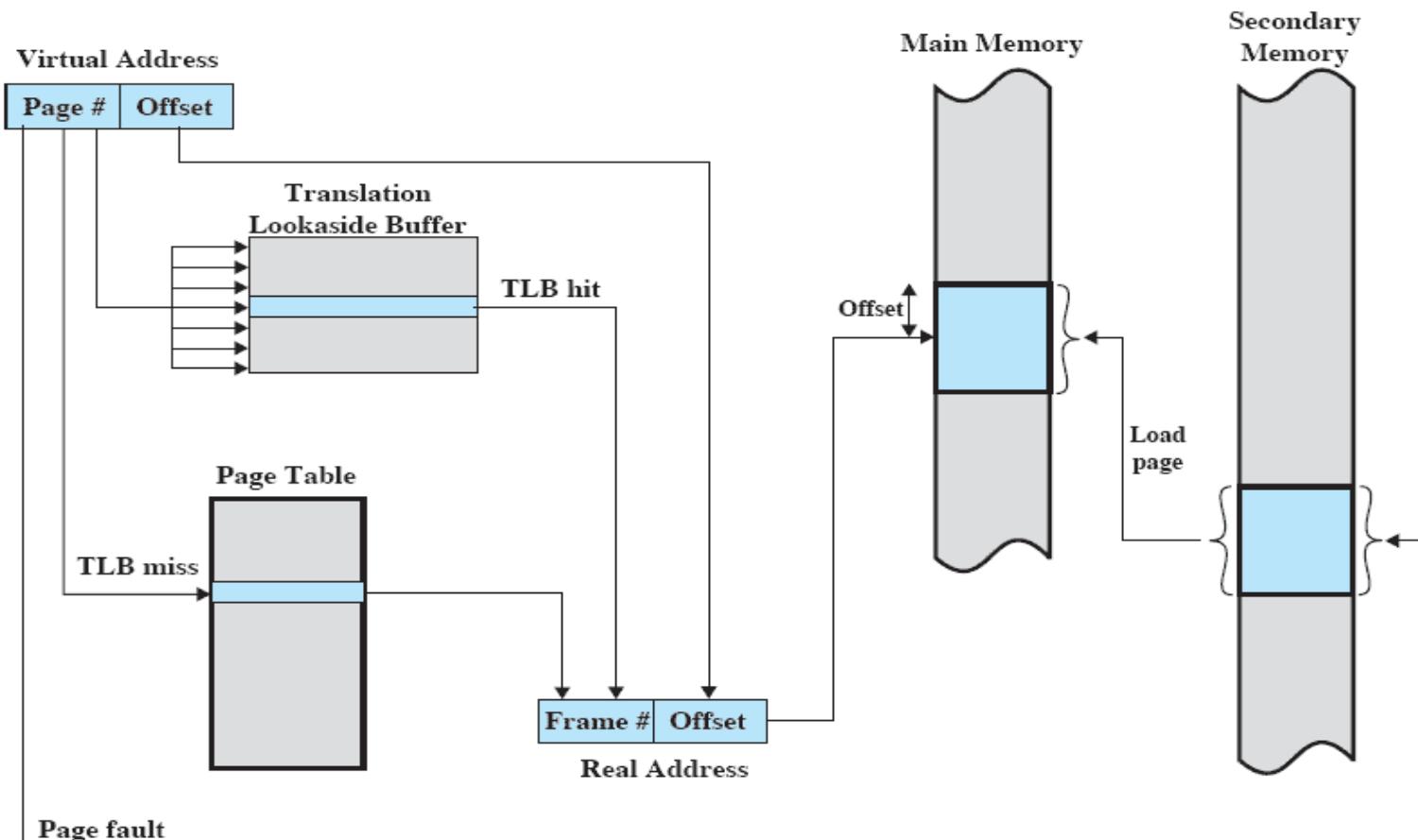


Figure 8.7 Use of a Translation Lookaside Buffer





# TLB operation

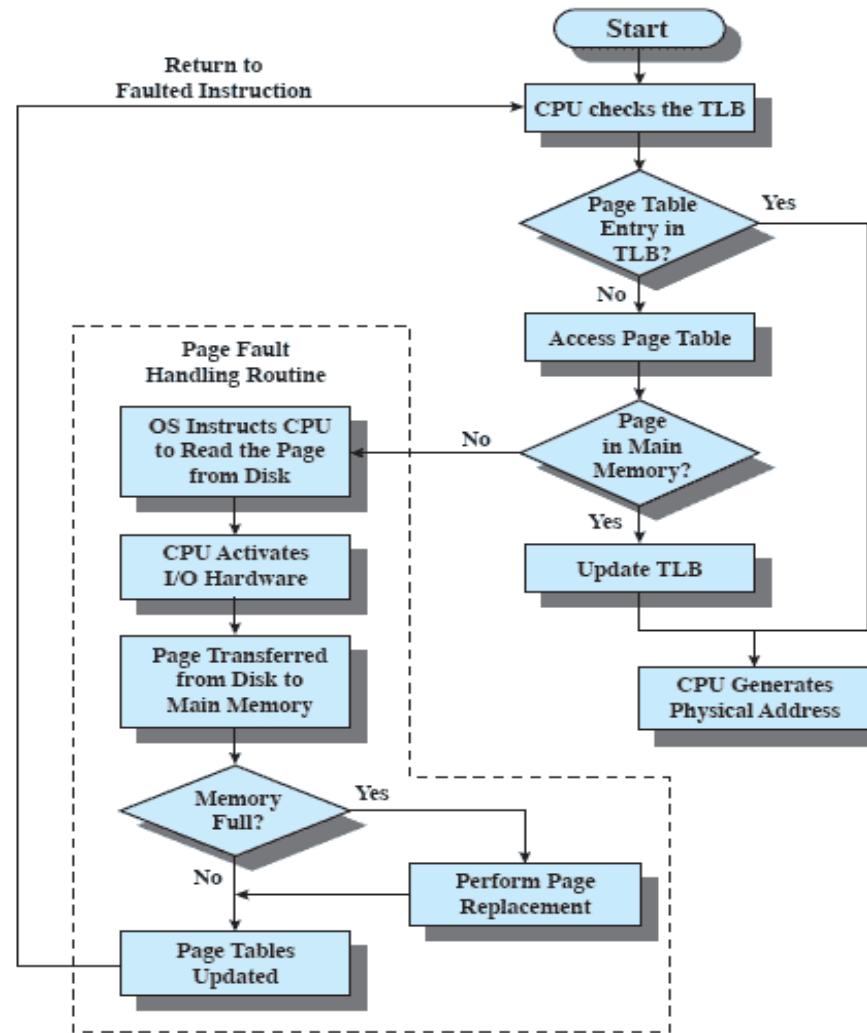


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]





## Hit Ratio

- The percentage of times that a particular page number is found in the TLB is called the hit ratio.
- **80% hit ratio means**, the desired page number in the TLB 80% of time.
- If it takes **20ns to search the TLB** and **100ns to access memory** then, a memory mapped access takes **120ns** when the page number is in the TLB.
- If we fail to find **the page number in TLB (20ns)** then, we must first access memory for the **page table and frame number (100ns)** and access the **desired byte in memory (100ns)**, for a total of **220ns**.





# Effective Access Time

Associative(TLB) Lookup =  $\varepsilon$  time unit

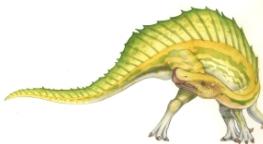
Assume memory cycle time is 1 microsecond

Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.

Hit ratio =  $\alpha$

## Effective Access Time (EAT)

$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha)$$





## Hit Ratio

- To find the effective memory access time =  
$$0.80 * 120 + (1 - 0.80) * 220 = 140\text{ns}$$
- In the above example we suffer a 40% slow down in memory access time (from 100 to 140ns)
- For a 98% hit ratio, we have  
Effective access time =  $0.98 * 120 + (1 - 0.98) * 220 = 122\text{ns}$ .
- The increased hit rate produces only a 22% slow down in access time.





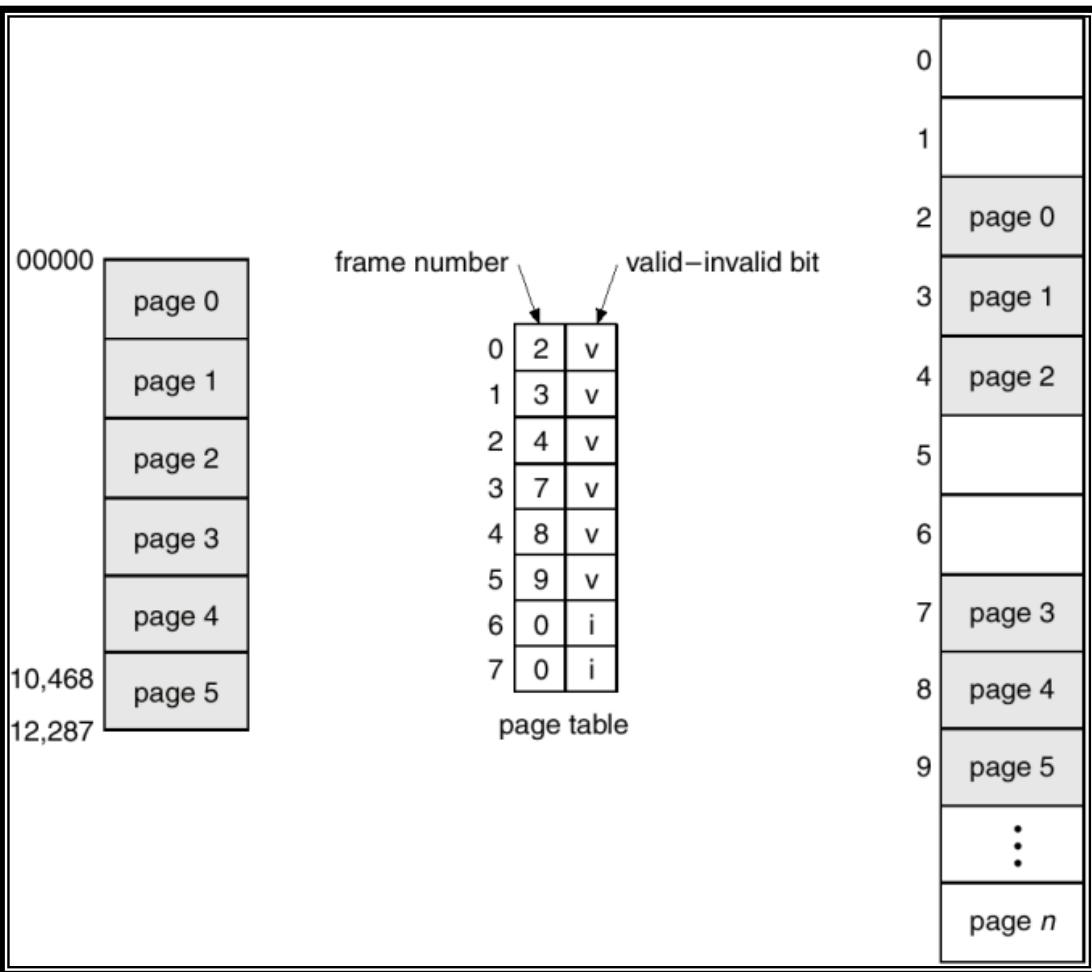
# Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- *Valid-invalid* bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.





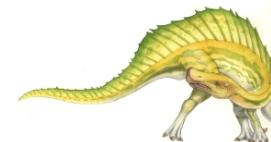
# Valid (v) or Invalid (i) Bit In A Page Table





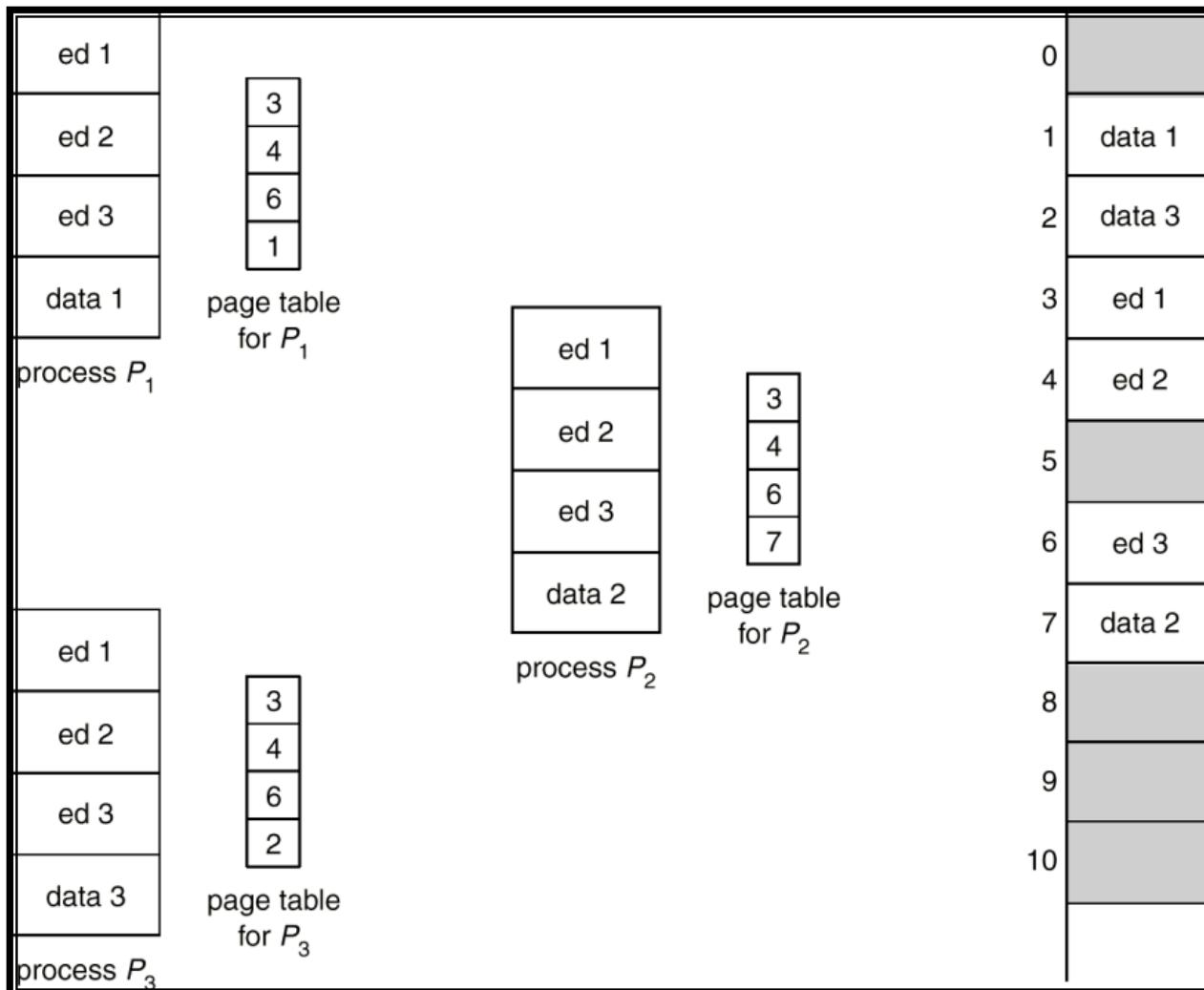
# Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- 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





# Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables





# Hierarchical Page Tables

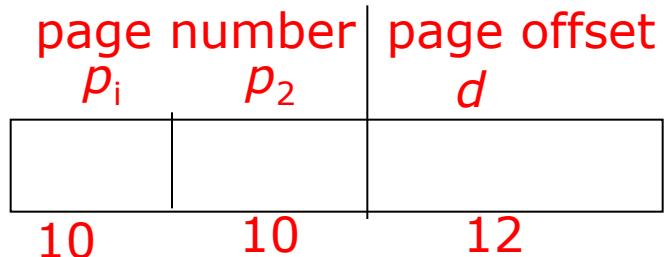
- Modern systems support a large logical address space ( $2^{32}$  to  $2^{64}$ ).
- In such systems the page table size is excessively large.
- Hence we need to break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.





## Two-Level Paging Example

- A logical address (on 32-bit machine with 4K( $2^{12}$ ) page size) is divided into:
  - a page number consisting of 20 bits  $= (2^{32}/2^{12}=2^{20})$ .
  - a page offset consisting of 12 bits.
- The page table may not be allocated continuously in main memory. The page table can be divided into smaller pieces.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

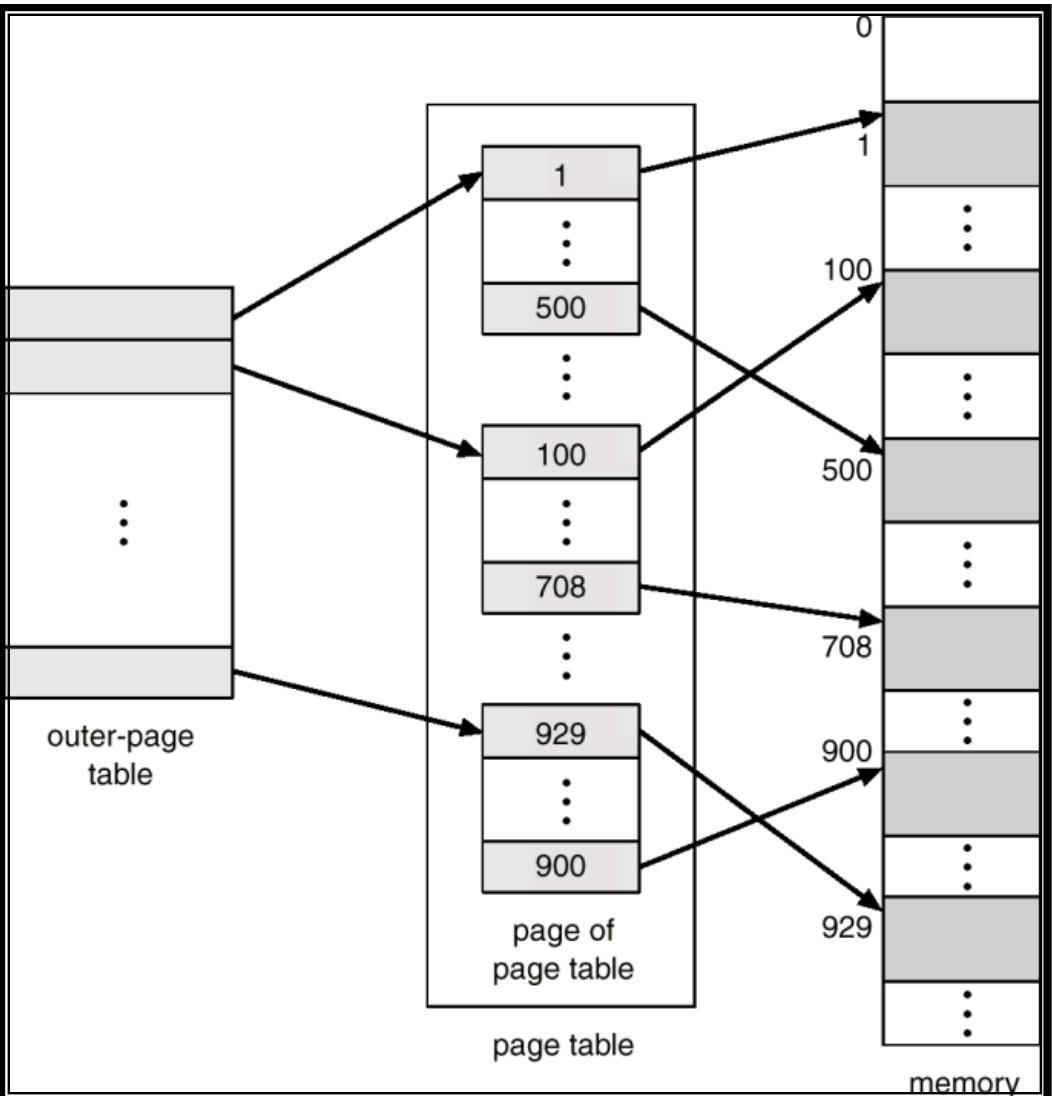


where  $p_i$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the outer page table.





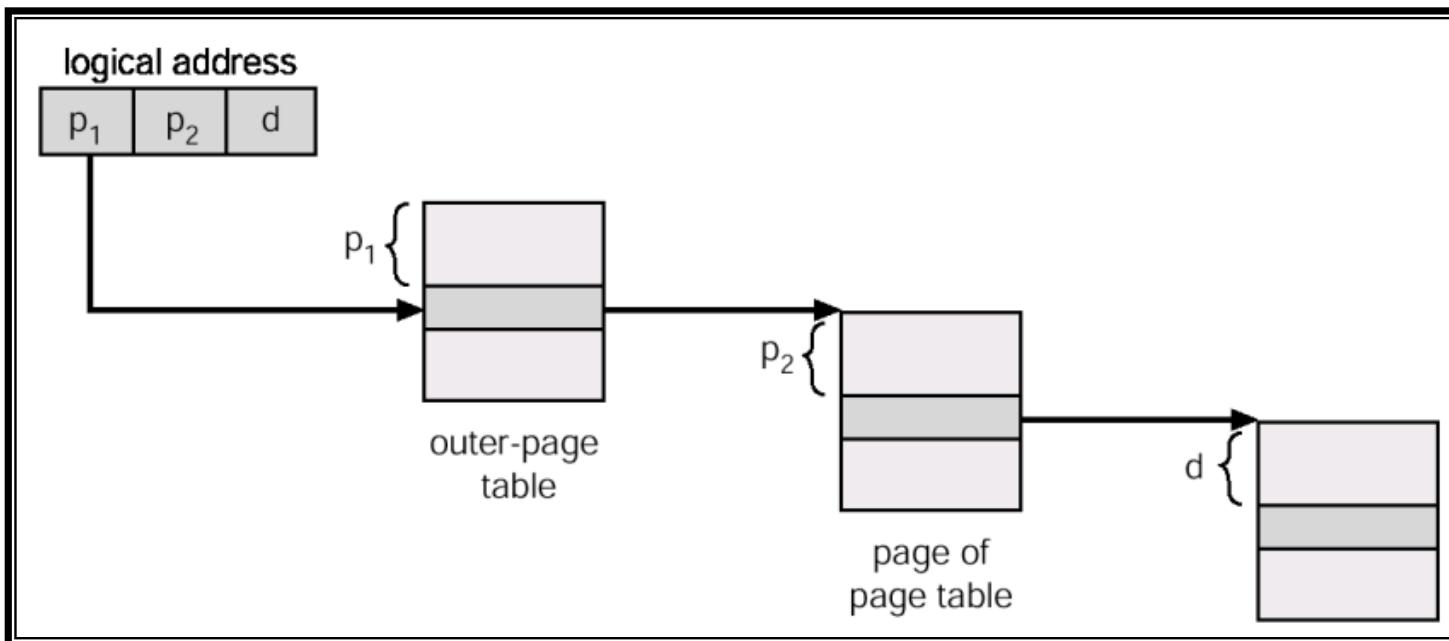
# Two-Level Page-Table Scheme





# Address-Translation Scheme

Address-translation scheme for a two-level 32-bit paging architecture





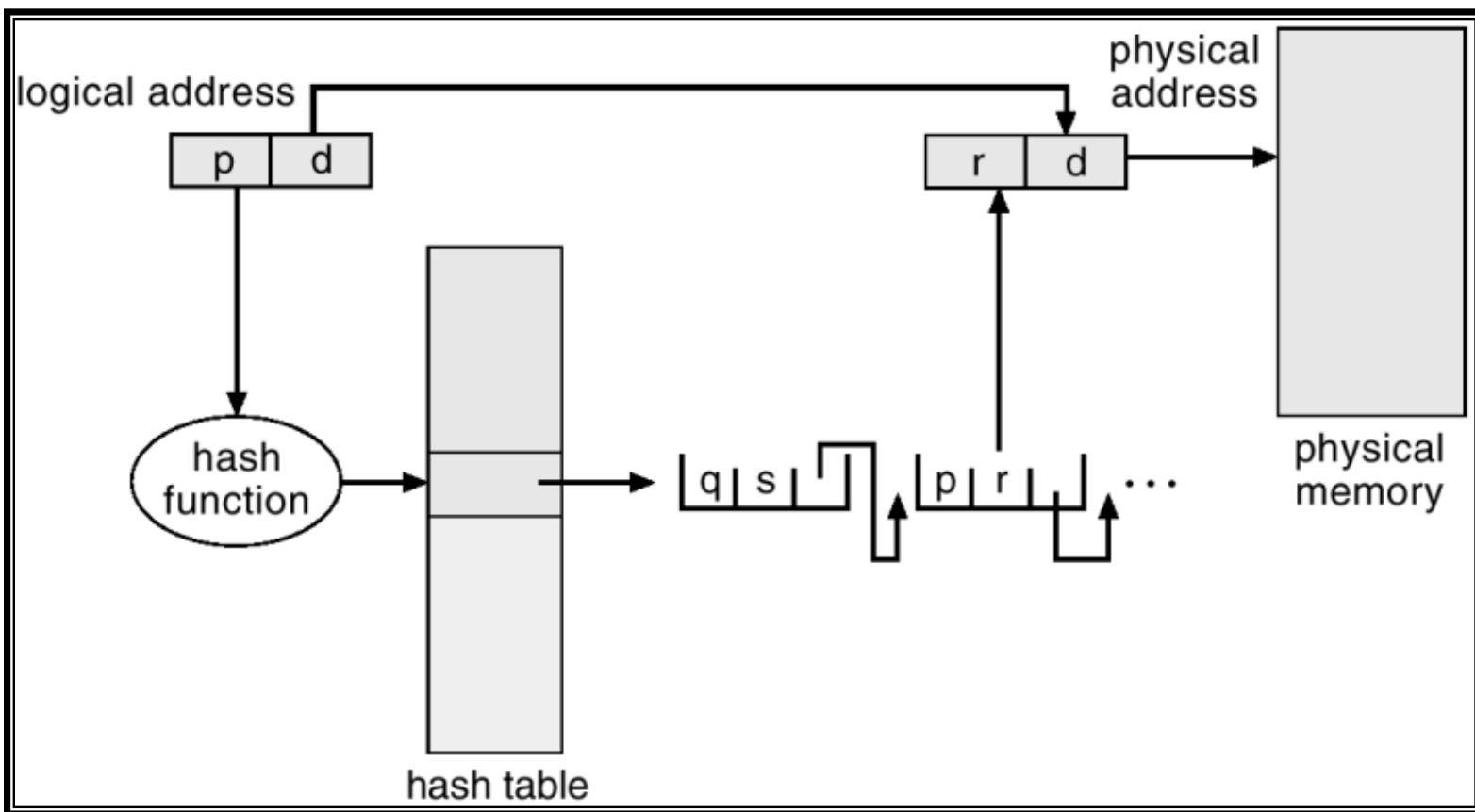
# Hashed Page Tables

- Common in handling logical 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 consists of 3 fields: virtual page number, the value of the mapped page frame and pointer to next element in the linked list.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.





# Hashed Page Table





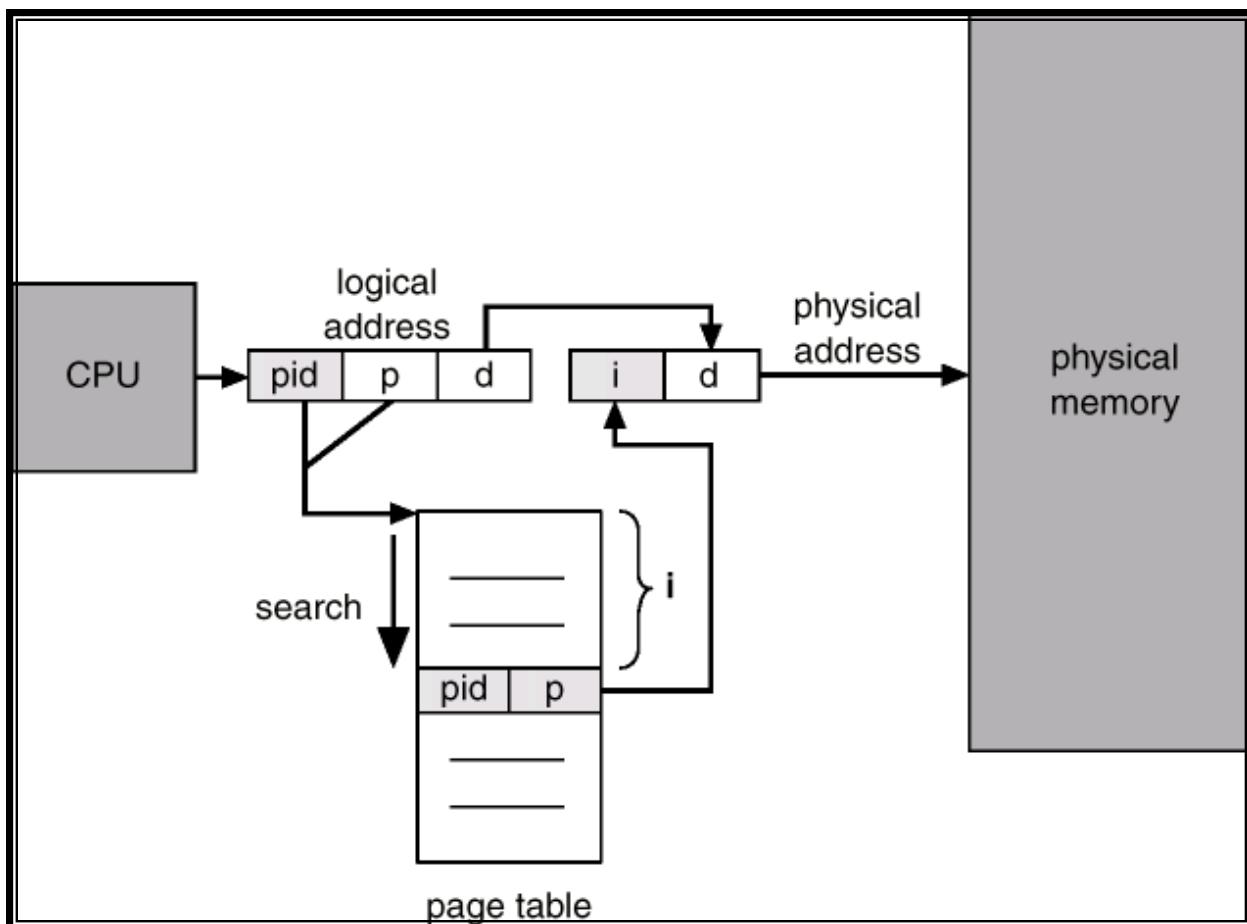
# Inverted Page Table

- 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.





# Inverted Page Table Architecture





# Segmentation

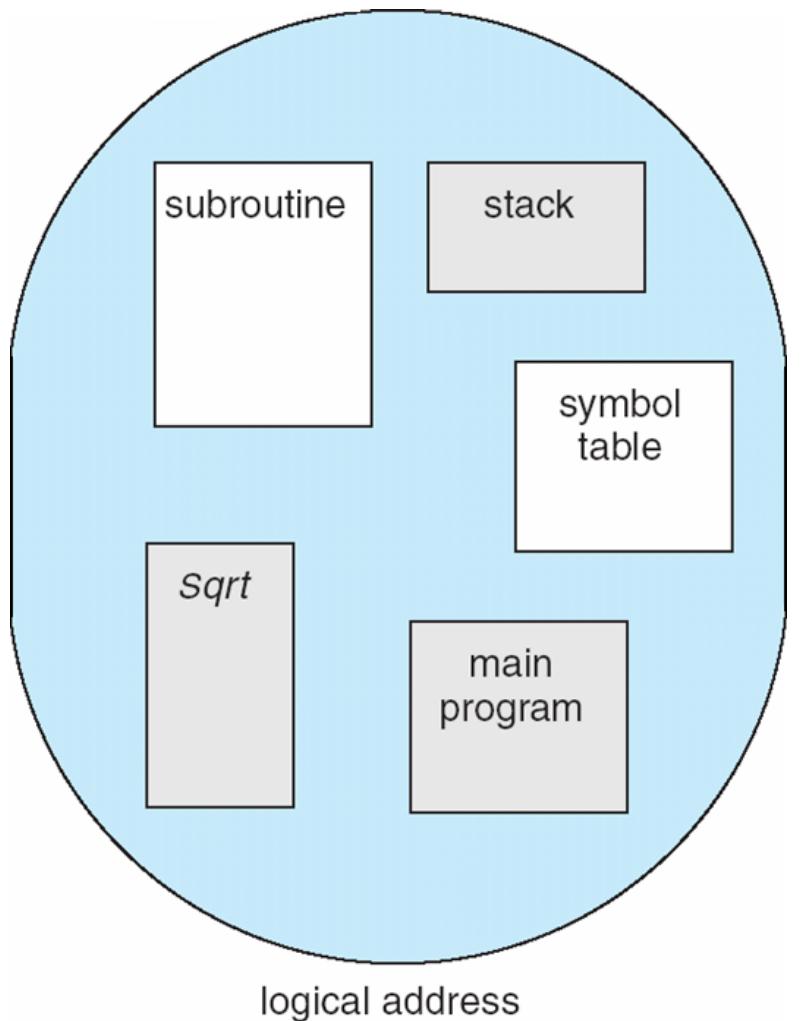
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
- A segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays

Each segment has a name and length.





# User's View of a Program



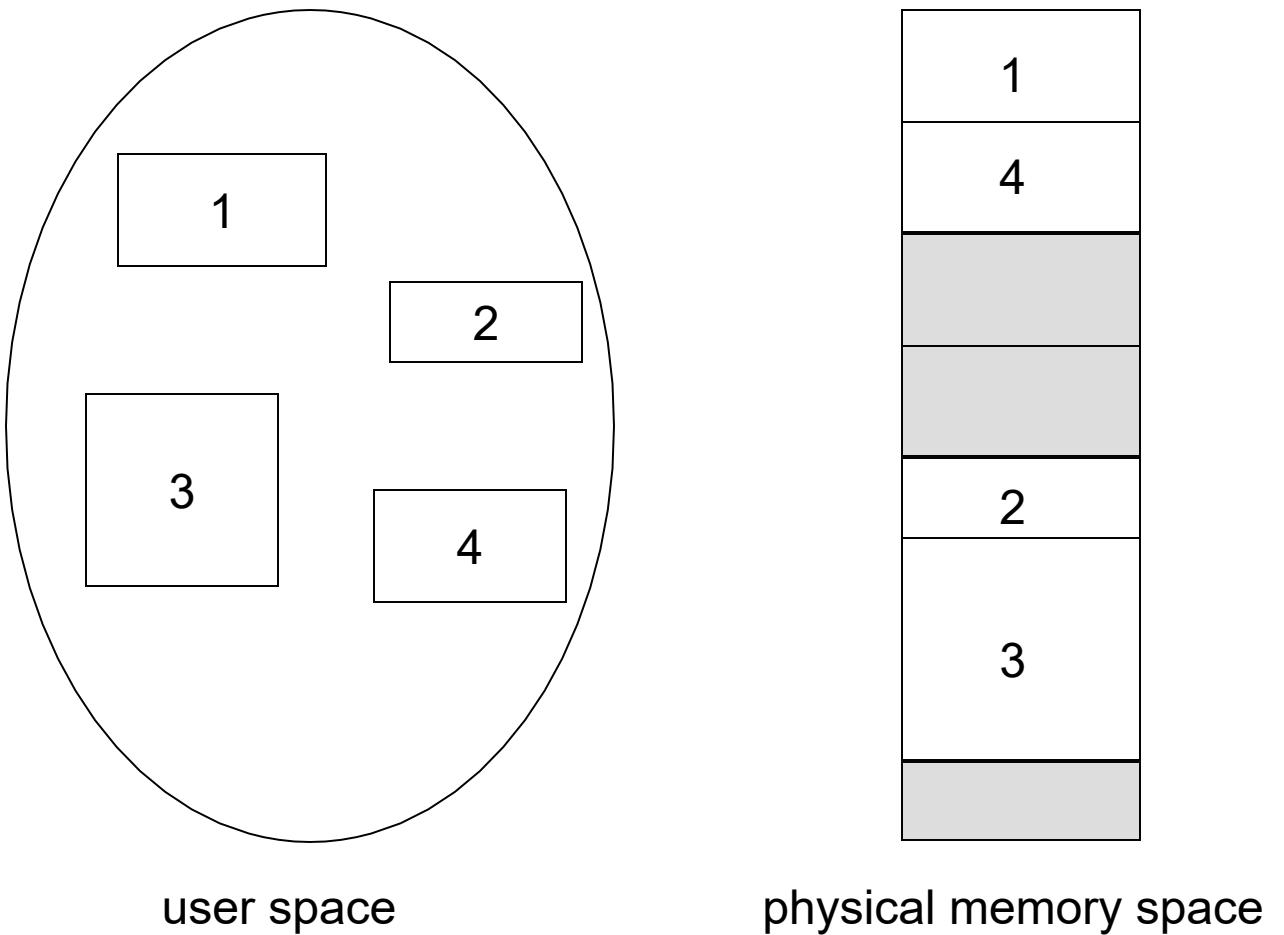
A C compiler might create separate segments for:  
The code  
Global variables  
The heap  
The stack  
The standard C library

The loader would take all these and assign them segment numbers





# Logical View of Segmentation





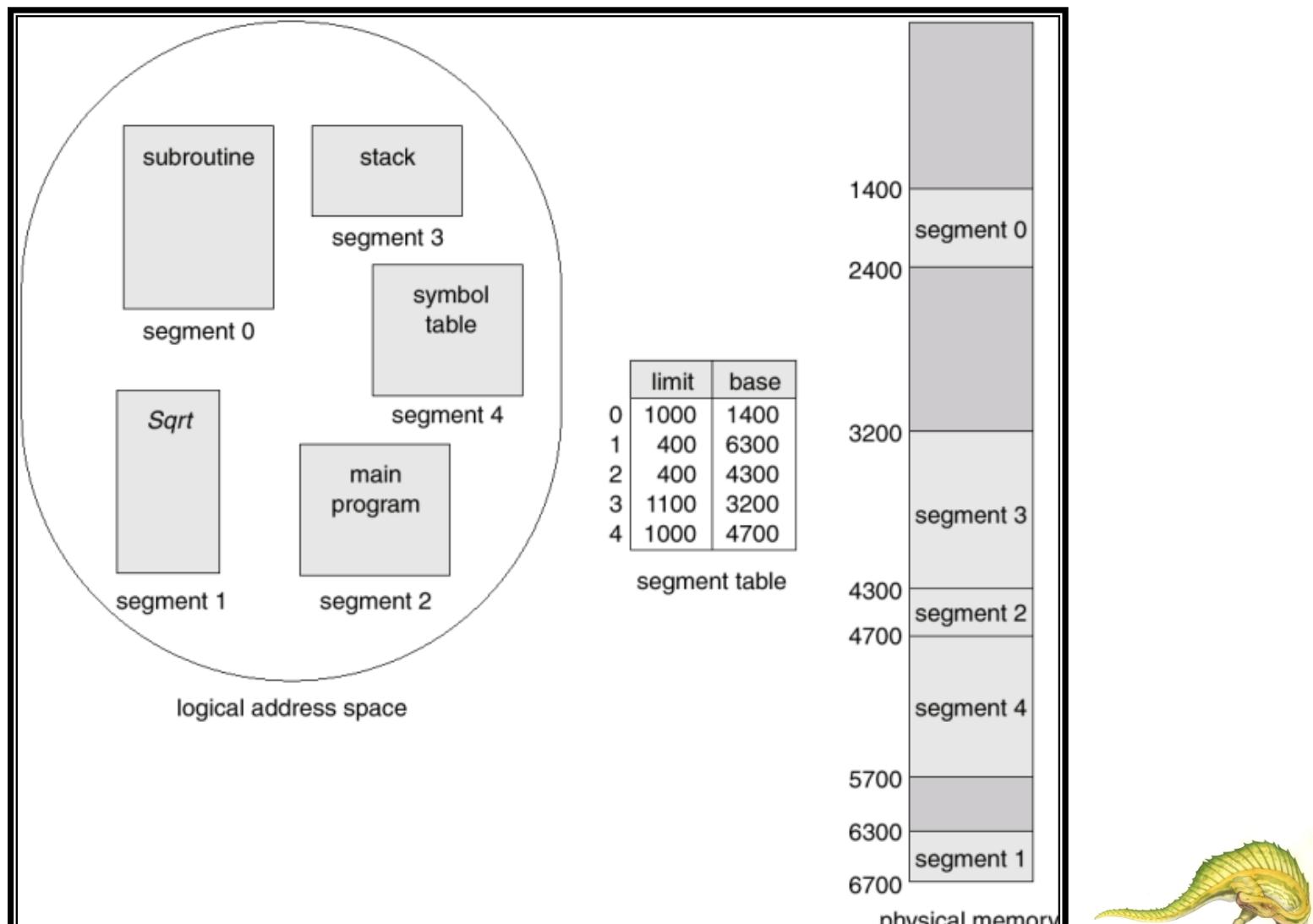
# Segmentation Architecture

- Segments are numbered and are referred to by segment number
- Hence, 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**





# Example of Segmentation





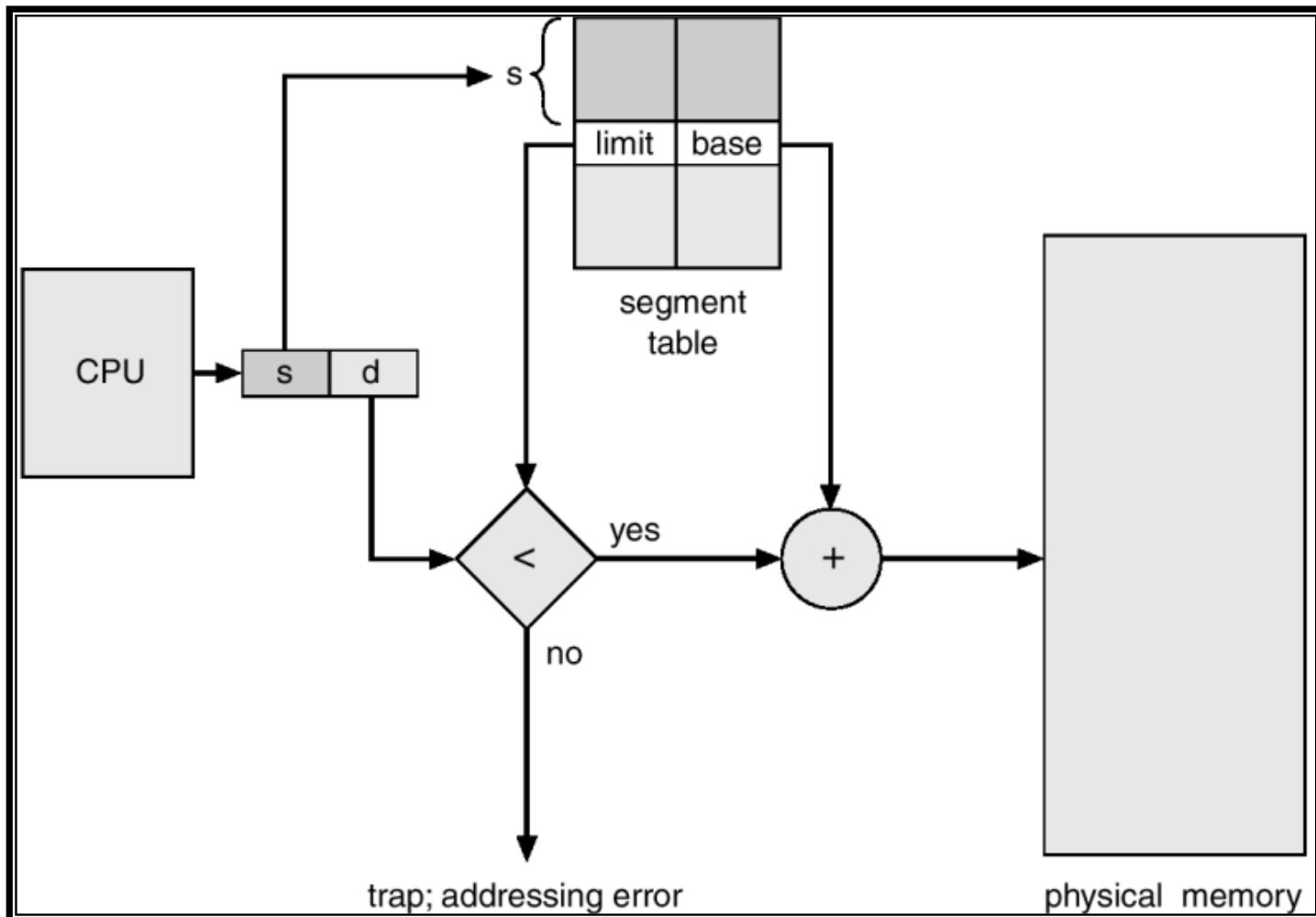
# Segmentation Architecture

- In the logical address <segment-number, offset>,
- The segment number is used as index to the segment table
- The offset must be between 0 and the segment limit, else trap to OS.
- If offset legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is essentially an array of base-limit register pairs.



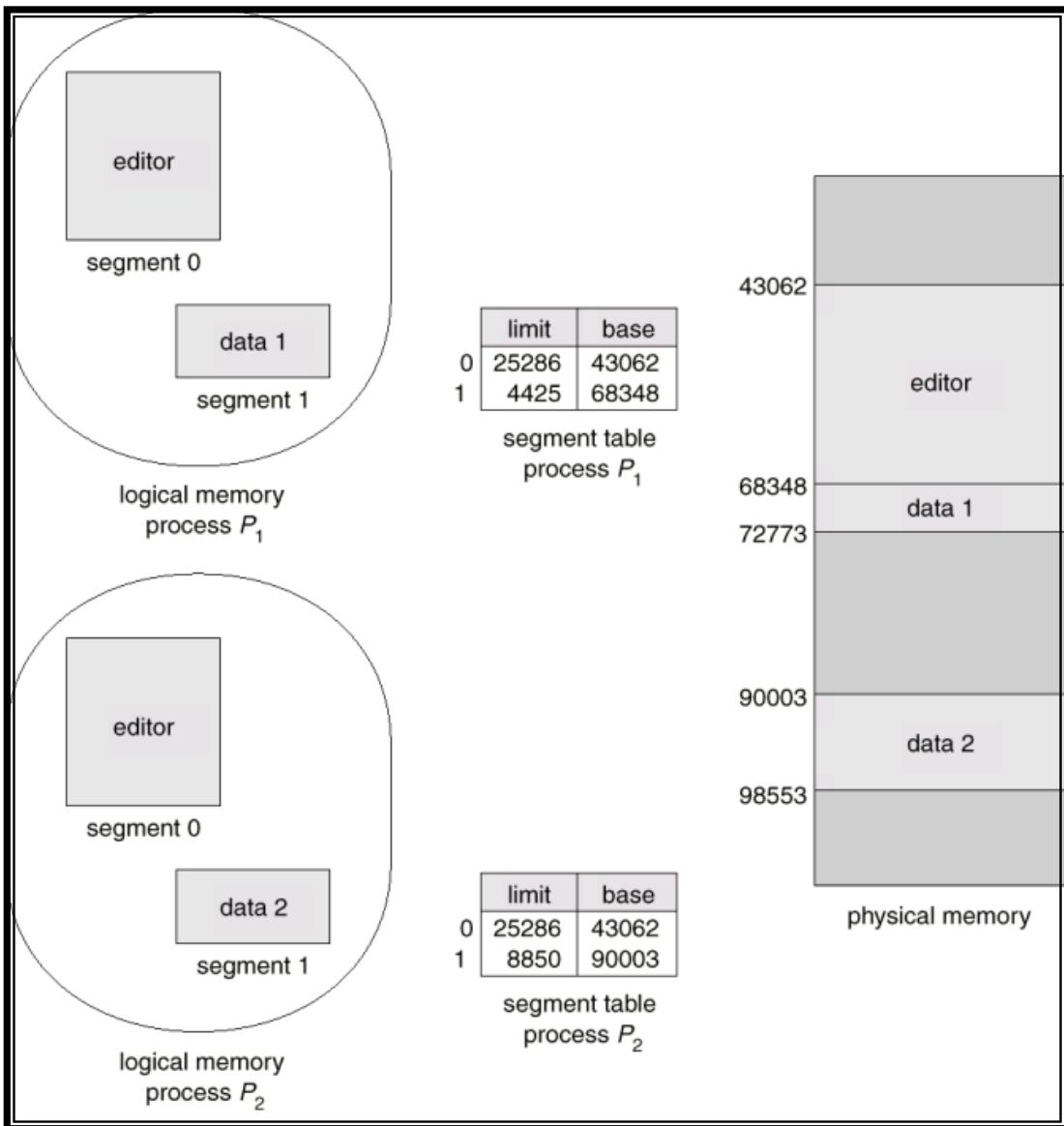


# Segmentation Hardware





# Sharing of Segments





## Segmentation: Example

- What is the physical addresses of below logical addresses(segment number, offset) according to the segment table?

- 2,700
  - 0,150
- 
- 2,700: Invalid
  - 0,150:  $500+150=650$  physical address

Base	Limit
500	200
700	1000
1800	600





# Paging and Segmentation

- Both paging and segmentation have their strengths
- Paging eliminates fragmentation(external) thus provides efficient use of main memory
- In addition because the pieces are of fixed equal size, it is possible to develop sophisticated memory management algorithms.
- Segmentation includes the ability to handle growing data structures, modularity and support for sharing and protection.
- To combine advantages of both, some systems are equipped with processor hardware and OS software to provide both





# Paging and segmentation Combination

- What if paging and segmentation used simultaneously?
  - What are the fields of a logical address?
  - How the physical address is achieved from a logical address?
  - What are the benefits?
  - In combined paging and segmentation, a user address space is broken up into number of segments, each segment in turn broken up into number of fixed size pages which are equal in length to main memory frame.





# Combined Paging and Segmentation

Virtual Address



Segment Table Entry



Page Table Entry



P = present bit  
M = Modified bit

(c) Combined segmentation and paging





# Address Translation

- Associated with each process is a segment table and a number of page tables, one per process segment.
- When a particular process is running, a register holds the starting address of the segment table for that process.
- Presented with a virtual address, the processor uses the segment number portion to index into the process segment table to find the page table for that segment.
- Then the page number portion of the virtual address is used to index the page table and look up the corresponding frame number.
- This is combined with offset portion of the virtual address to produce the desired real address.





# Address Translation

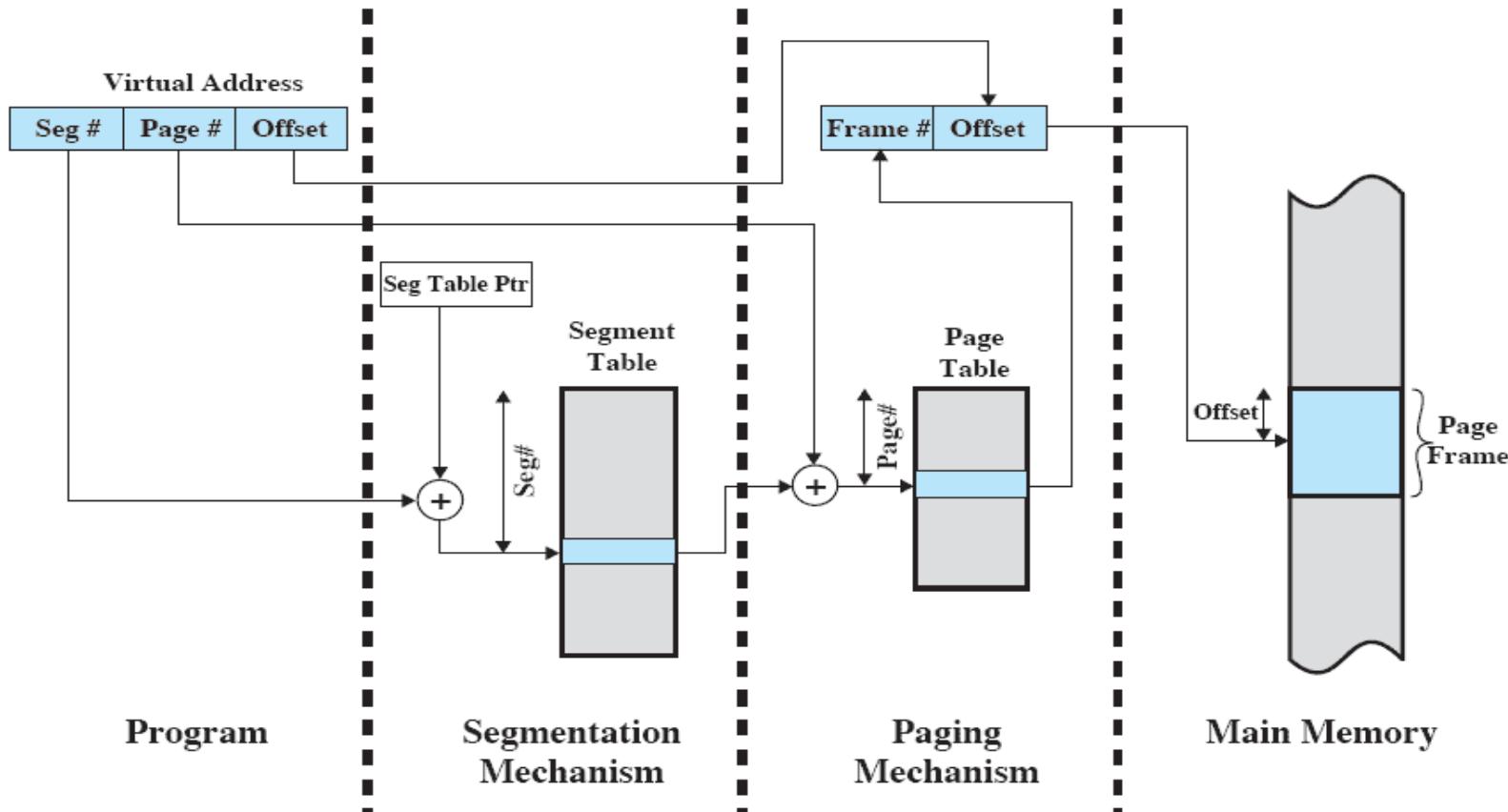
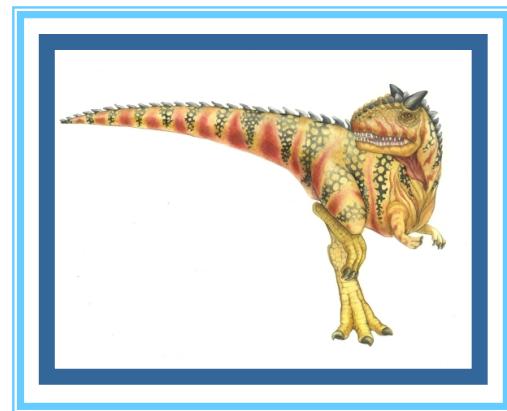


Figure 8.13 Address Translation in a Segmentation/Paging System



# Virtual Memory





# Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model





# Background

- We have discussed various memory management strategies that are used in computer systems.
- All these strategies have the same goal: to keep many processes in memory simultaneously to allow multi-programming.
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time – **virtual memory**
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and programs could be larger than physical memory





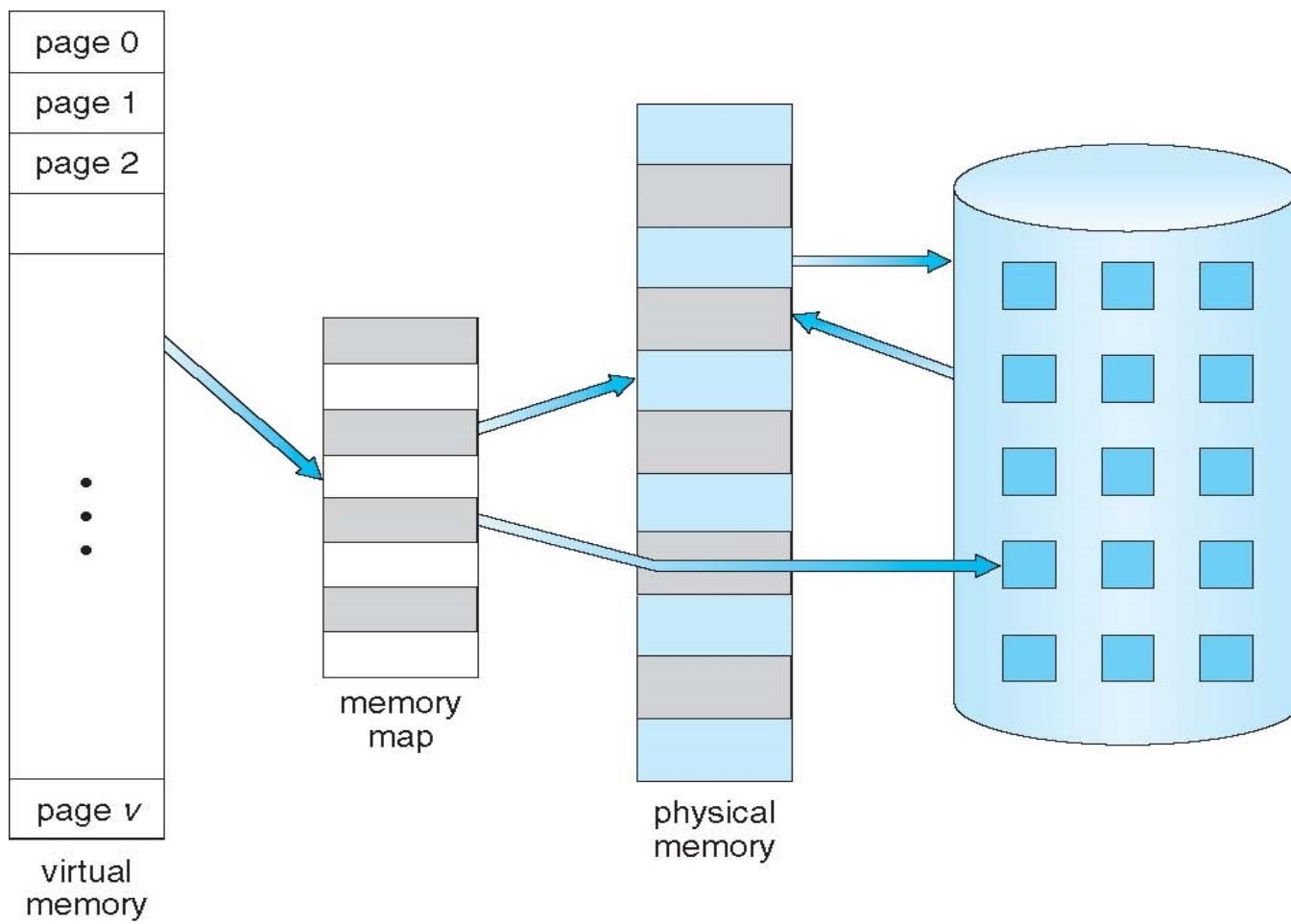
# Background

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory is not easy to implement, and may substantially decrease performance if it is used carelessly.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation - more complex to implement since segments size vary.





# Virtual Memory That is Larger Than Physical Memory





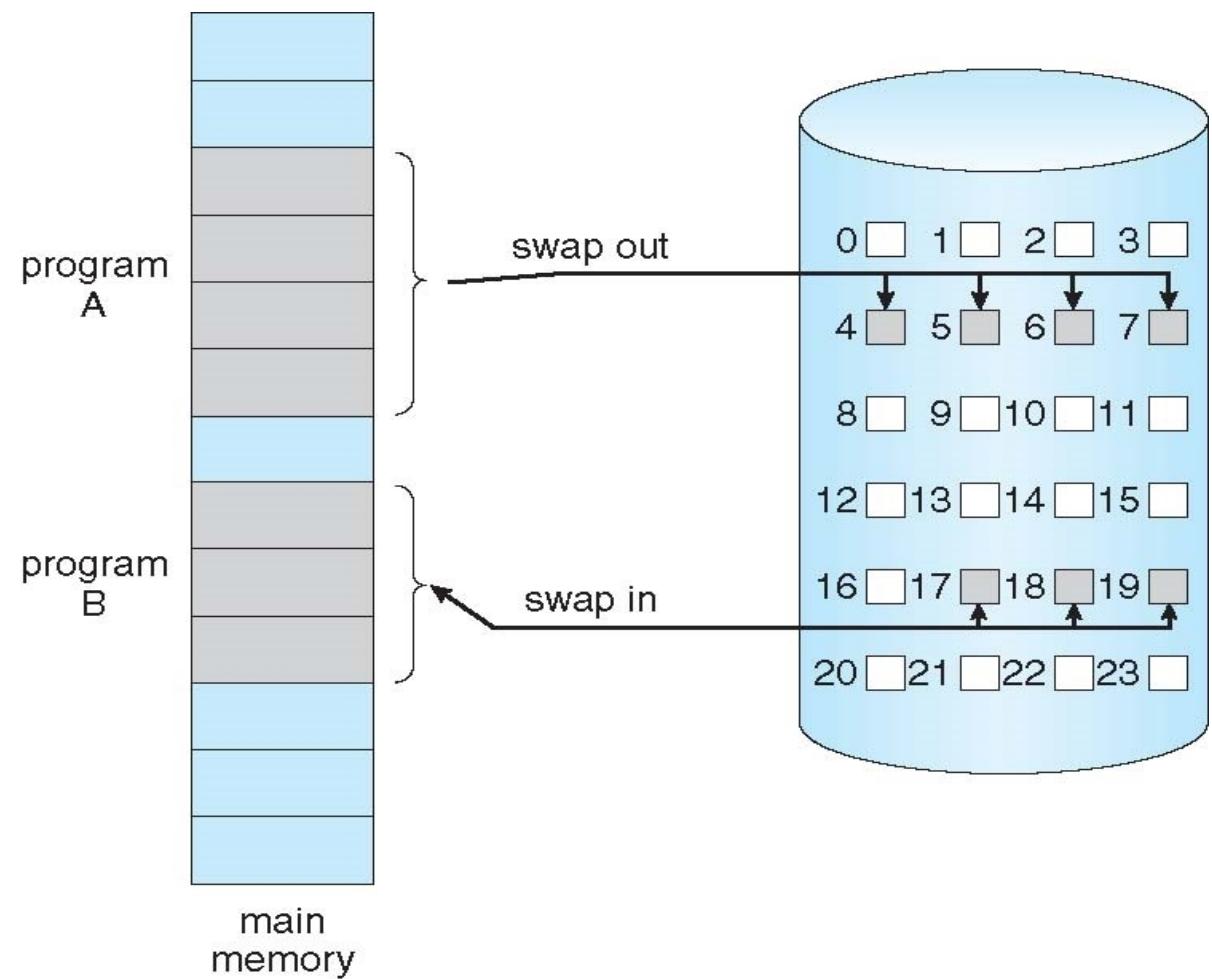
# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed – **demand paging** used in virtual memory systems
- Advantages
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
  - **swapper** or **sched** has **process ID 0** and is responsible for paging, and is actually part of the kernel rather than a normal user-mode process.





# Transfer of a Paged Memory to Contiguous Disk Space





## Valid-Invalid Bit

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again and brings only those pages into memory
- A form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid-invalid bit scheme can be used for this purpose.
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- If the bit is set to valid, the associated page is legal and is present in memory.
- If the bit is set to invalid, the page either is not valid or is valid but is currently on disk.
- Initially valid–invalid bit is set to **i** on all entries
  - Or may contain the address of the page on disk
- Example of a page table snapshot →
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

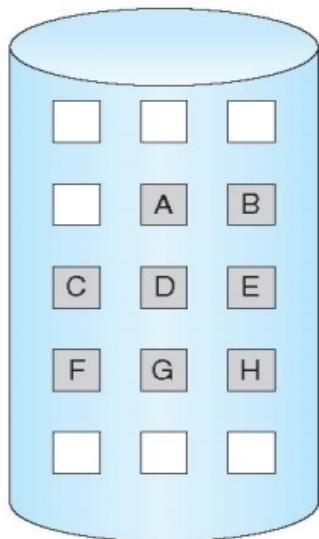
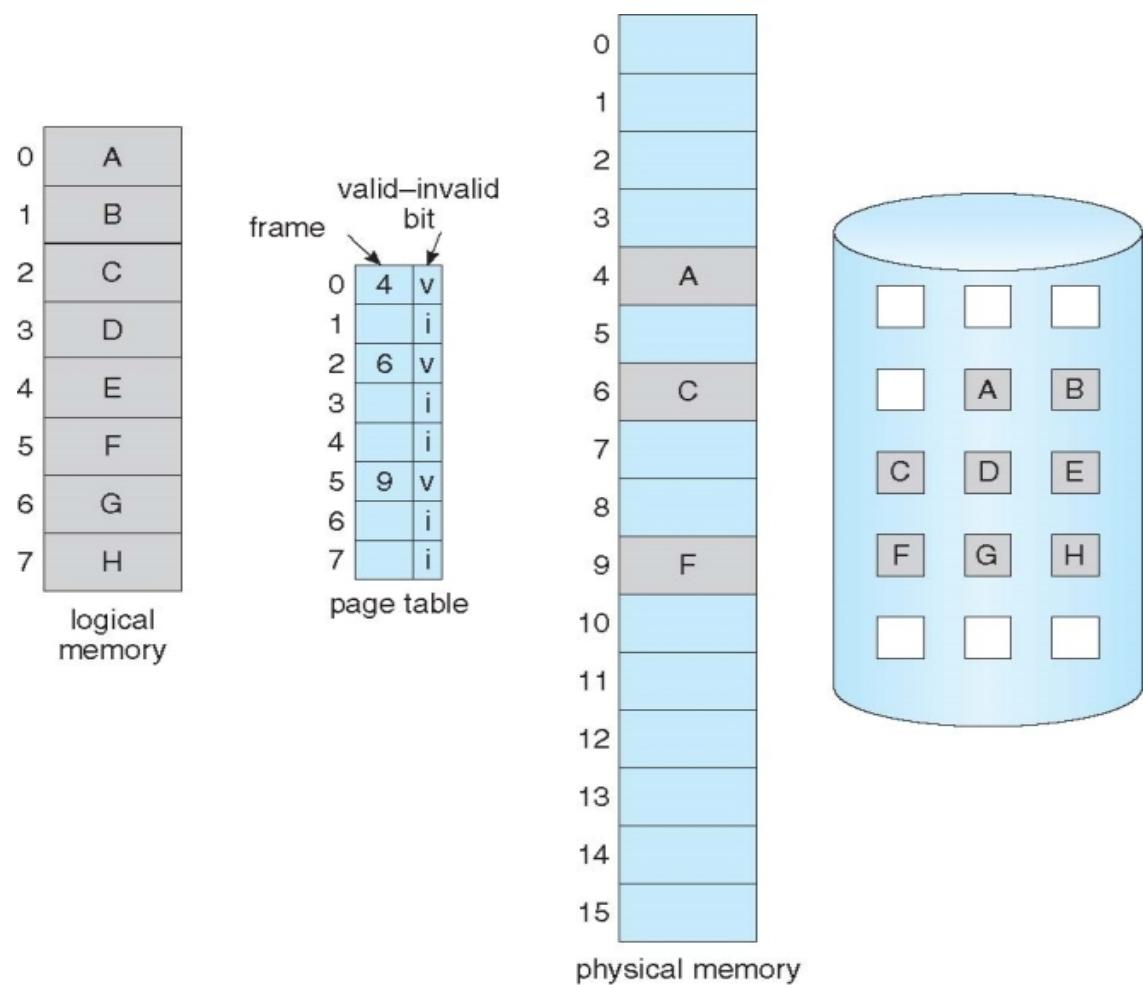
Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table





# Page Table When Some Pages Are Not in Main Memory





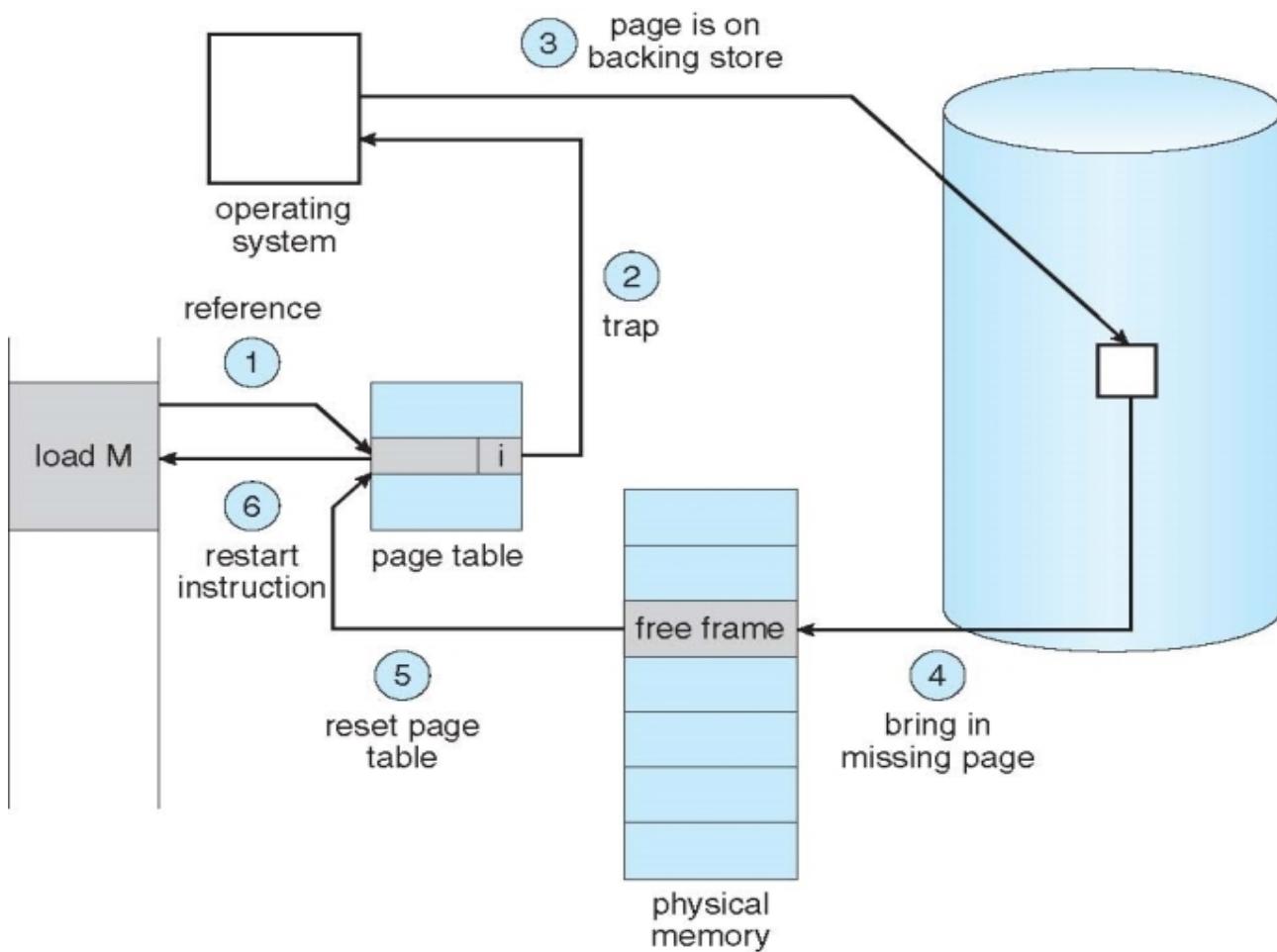
# Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
- **Procedure for handling page fault:**
  1. Operating system looks at an internal table (usually in PCB) to decide:
    - Invalid reference  $\Rightarrow$  abort
    - Just not in memory
  2. If the reference was invalid, terminate the process, or if valid but we have not yet brought that page in memory then, page it in
  3. Get empty frame from the free frame list
  4. Swap page into frame via scheduled disk operation
  5. When disk read is complete, reset the internal tables to indicate page now in memory: Set validation bit = **V**
  6. 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
  - 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





# 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.)

- 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)**

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





## Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} EAT &= (1 - p) \times 200 + p \text{ (8 milliseconds)} \\ &= (1 - p \times 200 + p \times 8,000,000) \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- 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 want performance degradation < 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 Considerations

- One additional aspect of demand paging is the handling and overall use of swap space
  - Can gain better paging throughput, by copying an entire file image into swap space at process start up and then performing demand paging from the swap space.  
or
  - Demand pages from file system initially, but to write the pages to swap space as they are replaced
    - ▶ This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space
    - ▶ Swap space is also used for pages including the stack and heap for a process





## Copy on write

- A process can start quickly by merely demanding in the page containing the first instruction.
- But process creation using the fork() system call may initially bypass the need for demand paging by a technique similar to page sharing.
- This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.





## Copy on write

- `fork()` creates a child process that is a duplicate of its parent.
- It creates a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
- Since the child processes usually call `exec()` immediately after creation, the copying of the parent's address space may be unnecessary.





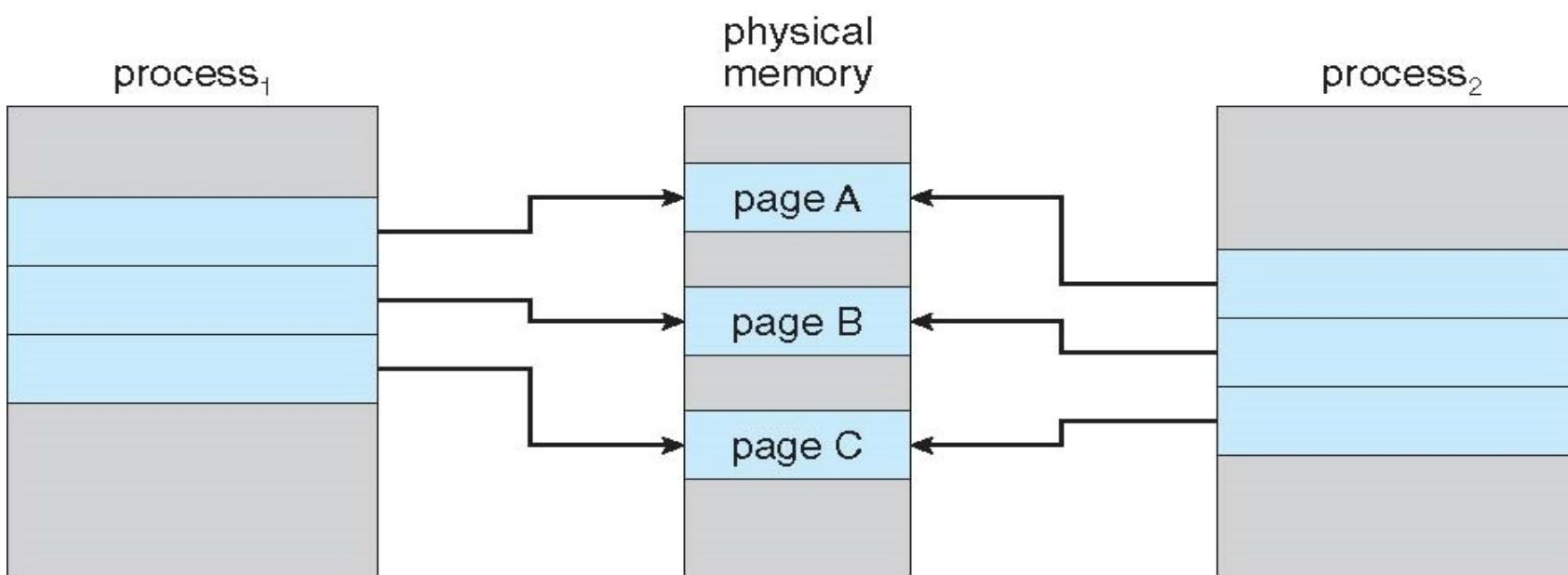
## Copy-on-Write

- **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
- Example:
  - Assume the child process attempts to modify a page containing portions of the stack; the OS recognizes this as a copy-on-write page.
  - The OS will then create a copy of this page, mapping it to the address space of the child process.
  - Therefore the child process will modify its copied page and not the page belonging to the parent process.



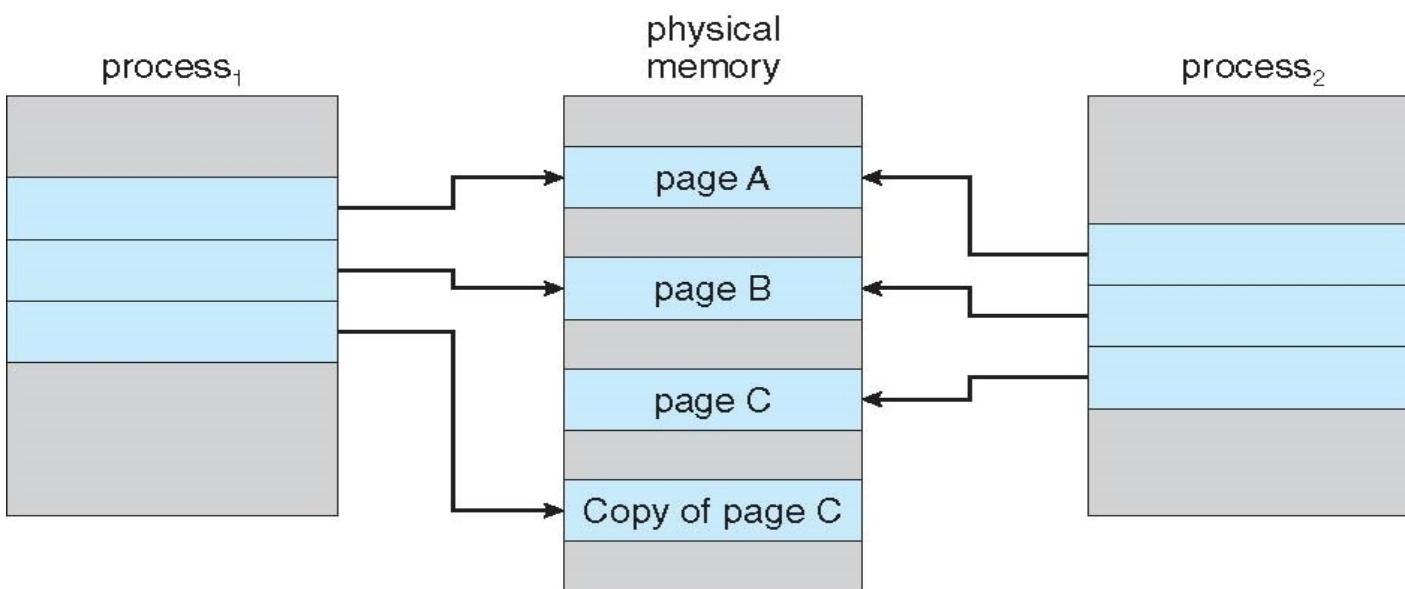


## Before Process 1 Modifies Page C





## After Process 1 Modifies Page C





## Copy-on-Write

- Using the copy-on-write technique, it is obvious that only the pages that are modified by either process are copied; all non-modified pages may be shared by the parent and child processes.
- Only those pages that may be modified need be marked as copy on write. Pages that cannot be modified (with executable code) may be shared by the parent and child.
- In general, free page frames are allocated from a **pool** of **zero-fill-on-demand** page frames.
  - These are allocated when the stack or heap for a process must expand or for managing copy-on-write pages.
  - These demand page frames have been zeroed-out before being allocated, thus erasing the previous contents of the page frame.
  - With copy-on-write, the page being copied to a zero-filled page frame.
  - Pages allocated for the stack or heap are similarly assigned zero filled page frames.





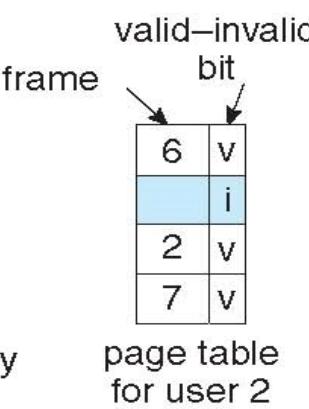
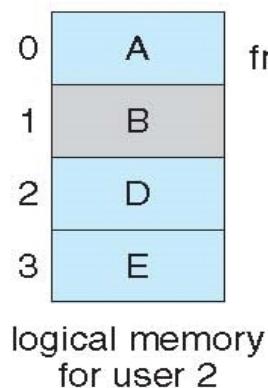
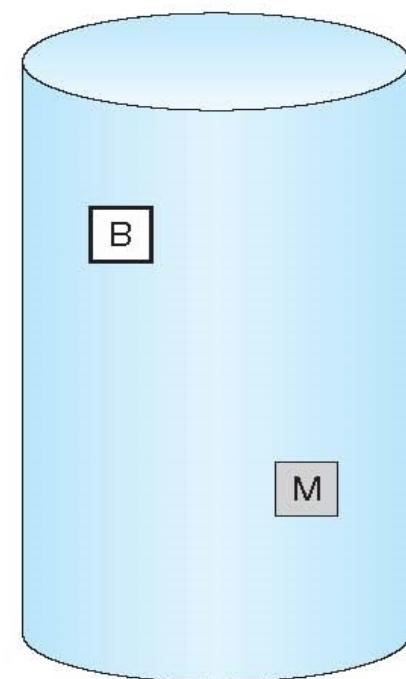
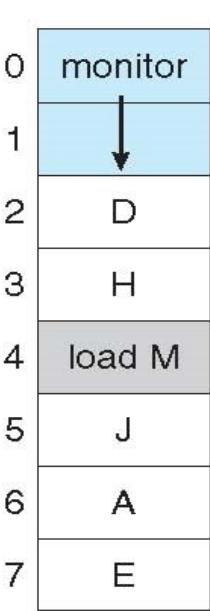
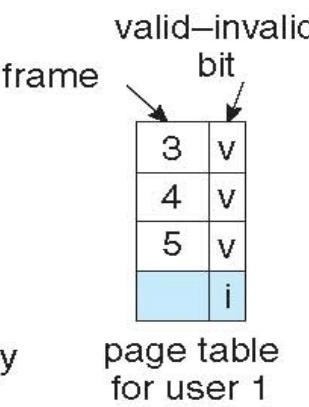
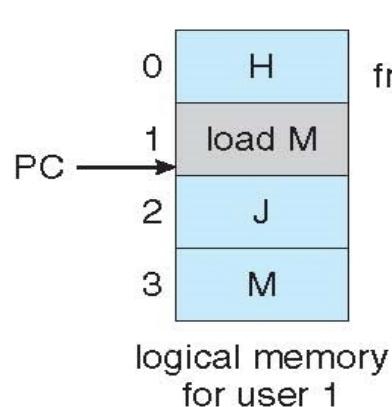
## What Happens if There is no Free Frame?

- While user process is executing page fault occurs.,
- The OS determines the desired page on disk to bring in, but no free frames in free frame list - Used up by process pages, also in demand from the kernel, I/O buffers, etc
- **Solution:**
- Page replacement – find some page in memory, but not really in use, page it out (writing its contents to swap space) and changing the page table to indicate that the page is no longer in main memory.
- We can now use the freed frame to hold the page for which the process faulted.





# 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

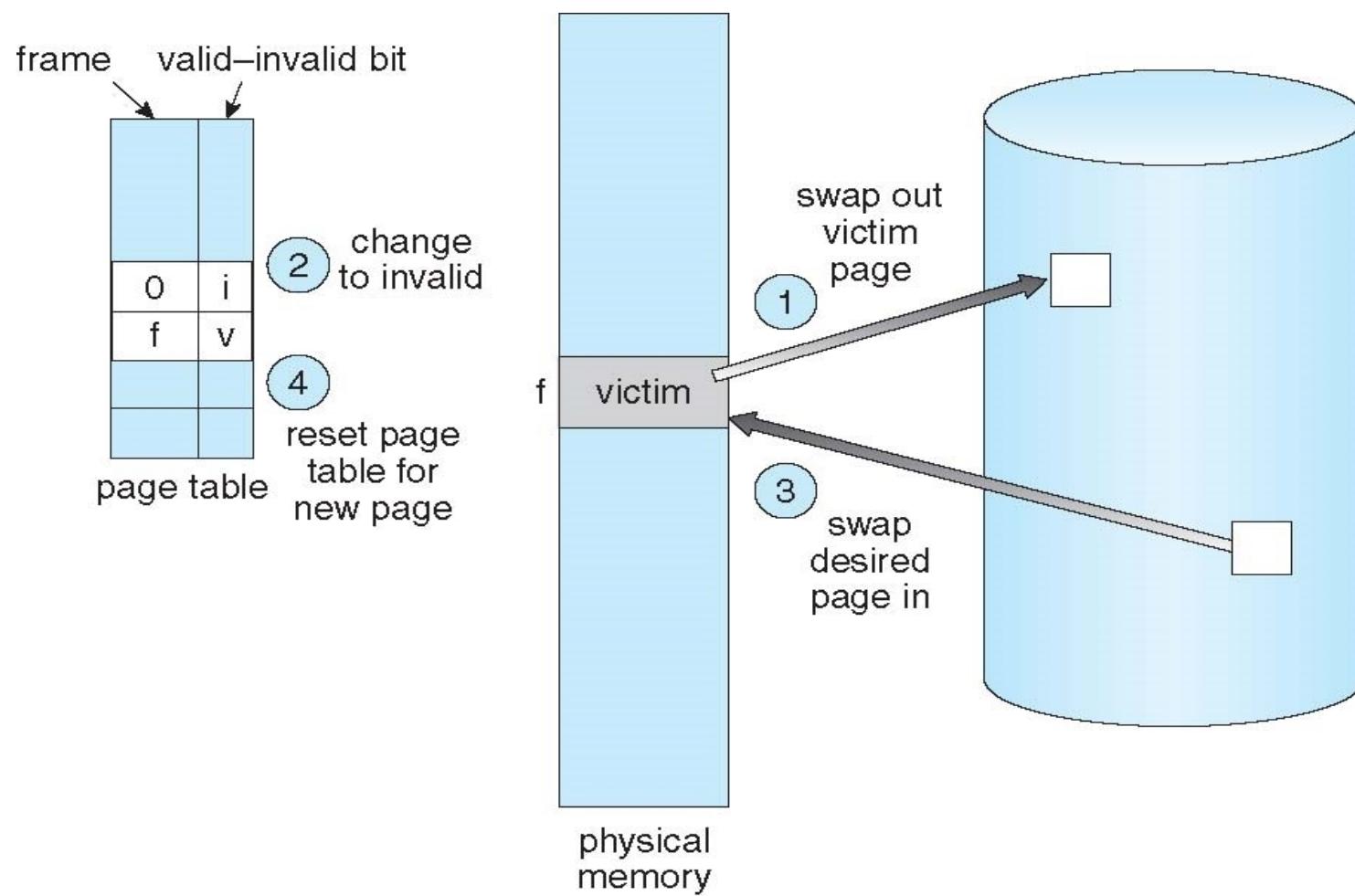
Note: now potentially 2 page transfers for page fault(one out and one in) – increasing EAT

Use **modify (dirty) bit for each frame** to reduce overhead of page transfers – only modified pages are written to disk, when it is selected for page replacement.





# Page Replacement





# Page and Frame Replacement Algorithms

- To implement demand paging we need to develop
  - a frame-allocation algorithm and
  - a page replacement algorithm
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace, when page replacement is required
- **Page-replacement algorithm**
  - Want lowest page-fault rate to be opted





# Page and Frame Replacement Algorithms

- 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

For example, if 100 bytes per page & if we trace a particular process, we might record the following address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0105, 0101, 0609, 0102, 0105.

This sequence can be reduced to the following reference string.

1, A, 1, 6, 1, b, 1, b, 1, 6,

- In all our examples, the reference string is

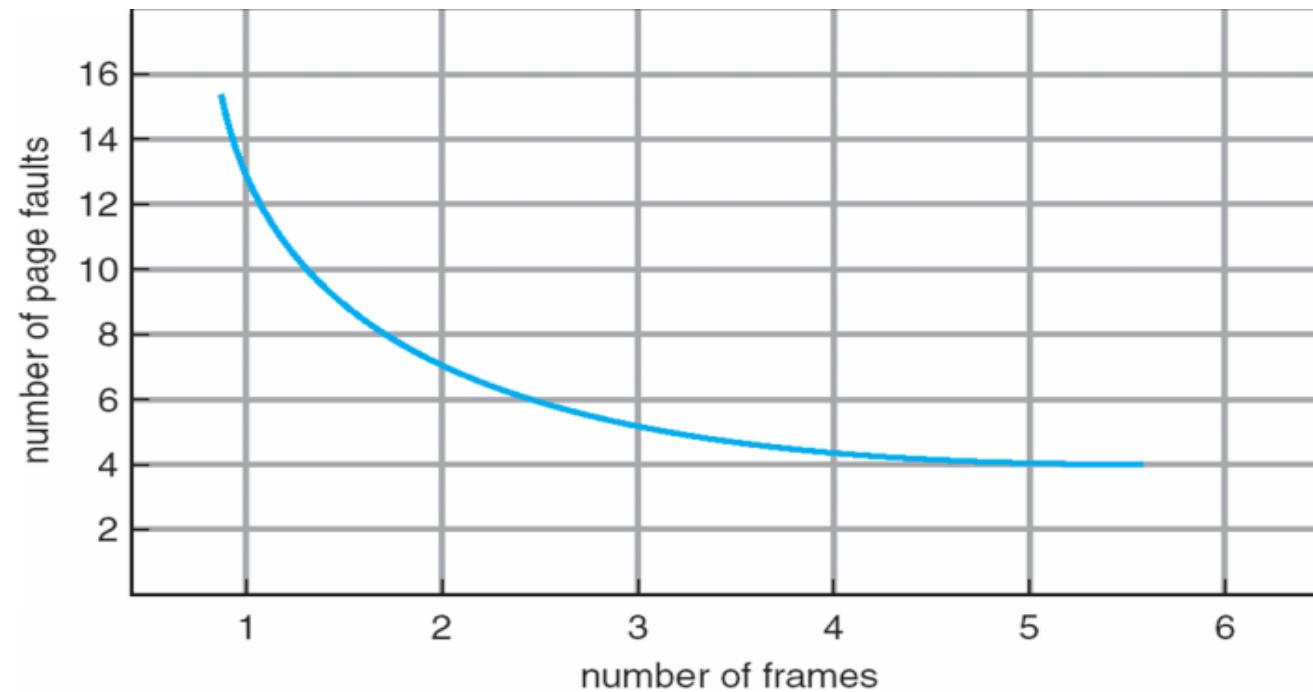
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1





## Graph of Page Faults Versus The Number of Frames

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As number of frames available increases, the number of page fault decreases.





## Reference String

- Reference string is the sequence of pages being referenced
- If user has the following sequence of addresses
  - 123, 215, 600, 1234, 76, 96
- If the page size is 100, then the reference string is
  - 1, 2, 6, 12, 0, 0





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

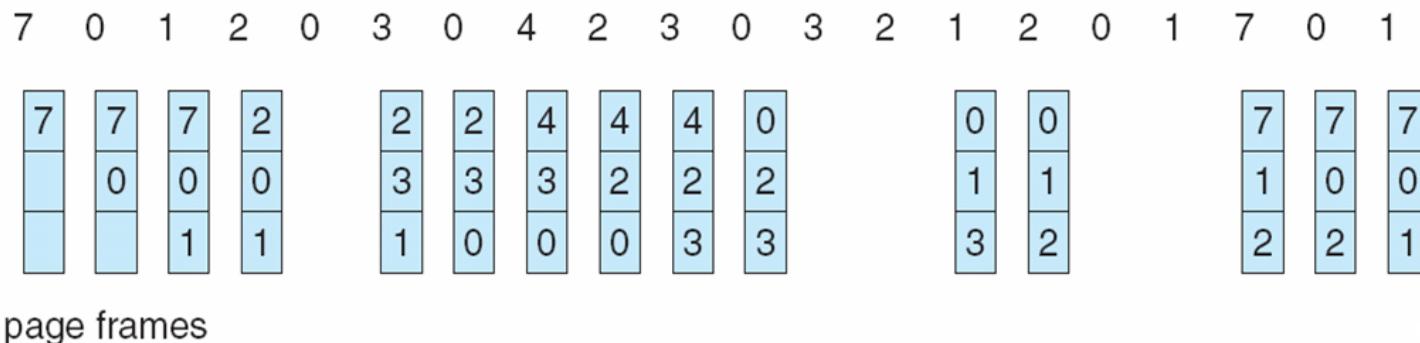
- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
  - keep a list (FIFO queue)
    - ▶ Replace the page at the head of the queue
    - ▶ Insert a new page brought into memory at the tail of the queue





# FIFO Page Replacement

## reference string



- Page Fault Rate = Number of Page Faults / Total Number of Page References  
 $= 15 / 20 = 0.75$

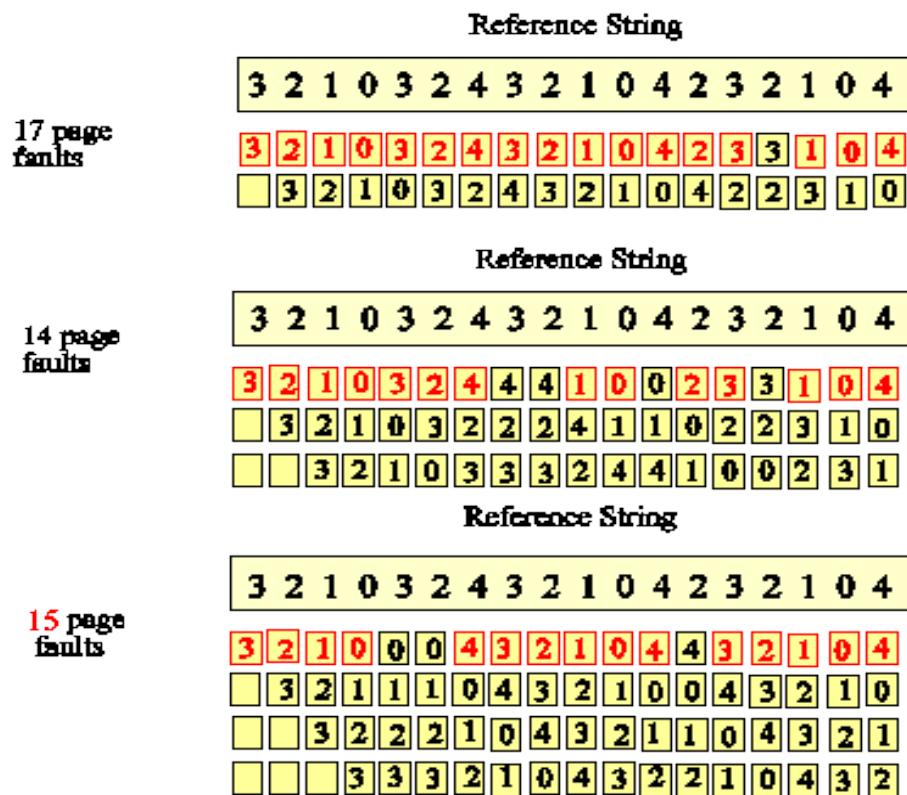




# Belady's Anomaly

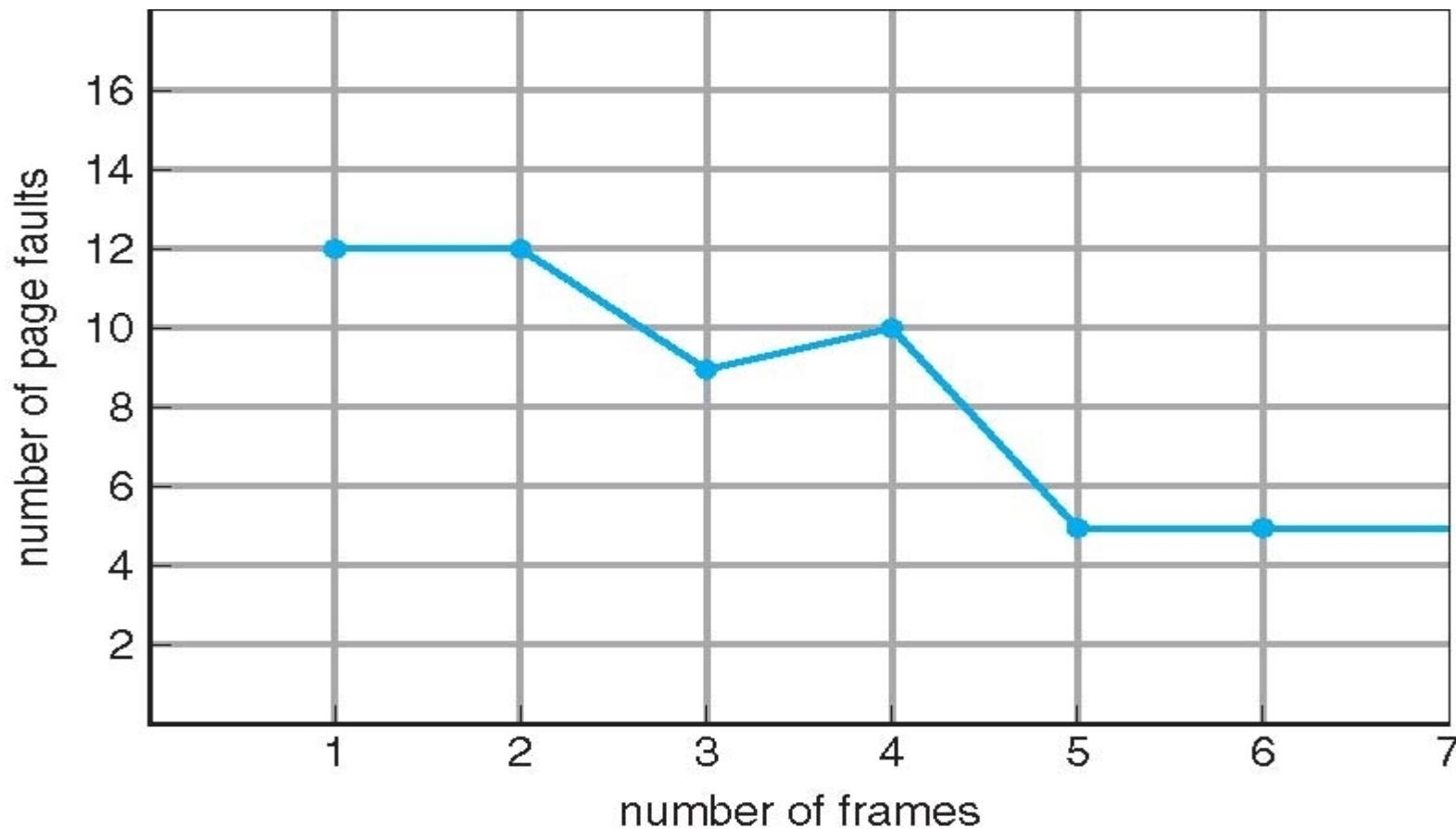
FIFO Replacement has Belady's Anomaly – increase in number of page frames increases page faults as in the below scenario

First In - First Out Page Replacement Algorithm





## FIFO Illustrating Belady's Anomaly





## FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage





# Optimal Algorithm

- Basic idea
  - replace the page that will not be referenced for the longest time
- This gives the lowest possible fault rate
- Impossible to implement
  - How do you know the future page references
  - Can't read the future
- Used for measuring how well your algorithm performs

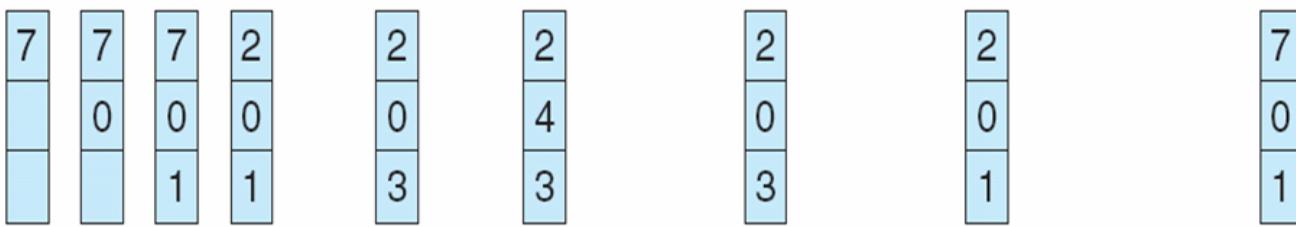




# Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

- Fault Rate =  $9 / 20 = 0.45$



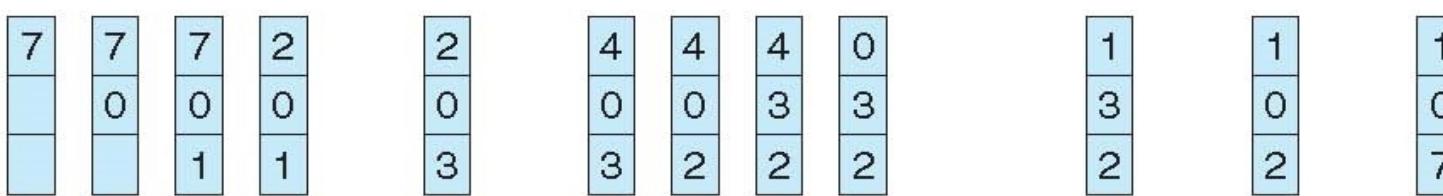


## Least Recently Used (LRU) Algorithm

- Replace the page in memory that has not been accessed for the longest time
- Use past knowledge rather than future
- 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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





## LRU Algorithm (Cont.)

- How to keep track of last page access?
  - requires special hardware support
  - 2 solutions – Counter Implementation and Stack Implementation
- 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 pointers to be changed
  - But each update more expensive
  - No search for replacement – bottom page of stack to be replaced
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

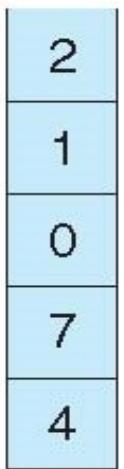




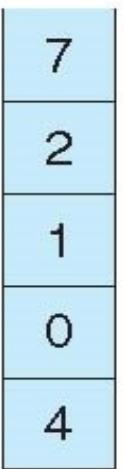
# Use Of A Stack to Record The Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2



stack  
before  
a



stack  
after  
b





## LRU Issues

- Both techniques just listed require additional hardware
  - remember, memory reference are very common
  - impractical to invoke software on every memory reference
- LRU is not used very often
- Instead, we will try to approximate LRU
- Few computers provide hardware support for true LRU Page replacement
- Many systems provide reference bits for LRU approximation.
- Initially all bits are cleared by the operating systems, and as user executes the bit associated with the reference bit is set to 1.
- Later by examining reference bits, we can determine which pages have been used and which have not been used.
- Some LRU approximation algorithms are:
  - Second Chance /CLOCK algorithm
  - Enhanced Second Chance algorithm
  - Additional Reference Bit algorithm





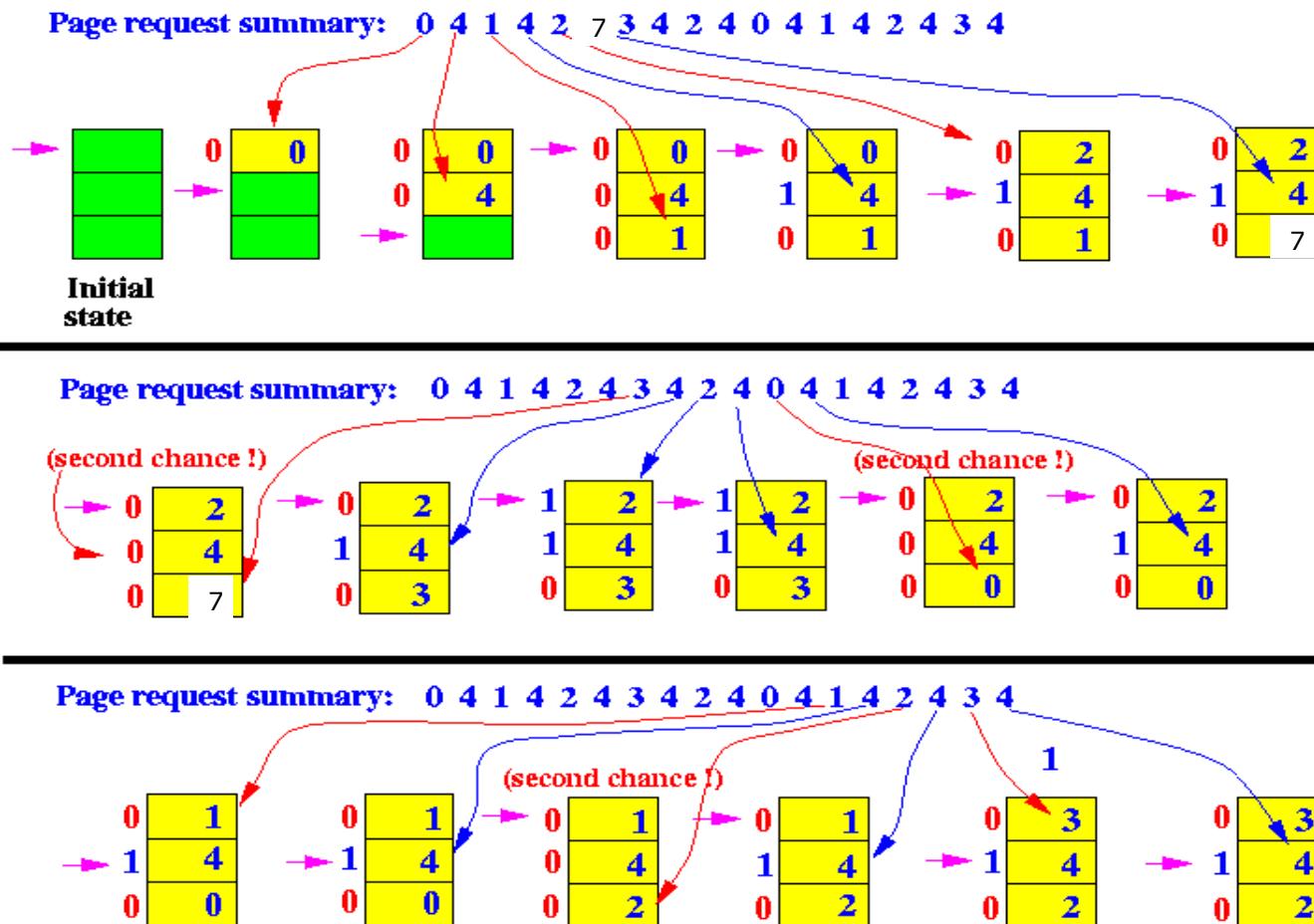
# CLOCK / Second-chance algorithm

- LRU needs special hardware and still slow
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - ▶ We do not know the order, however
- CLOCK / 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, whose reference bit is 0



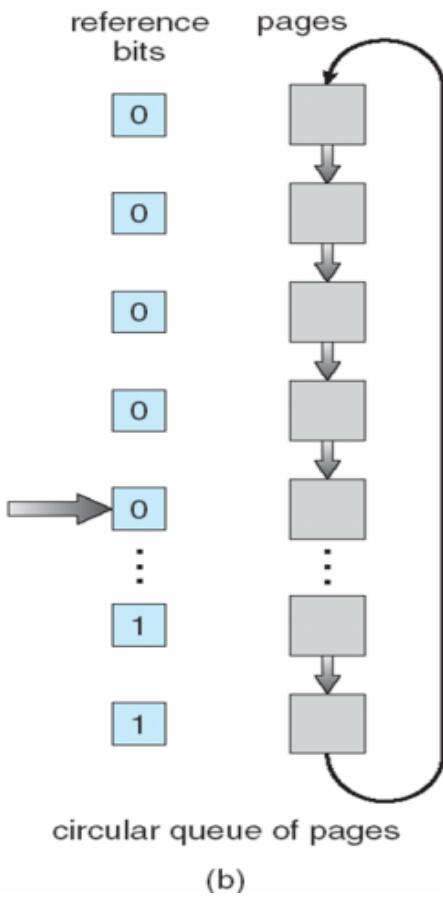
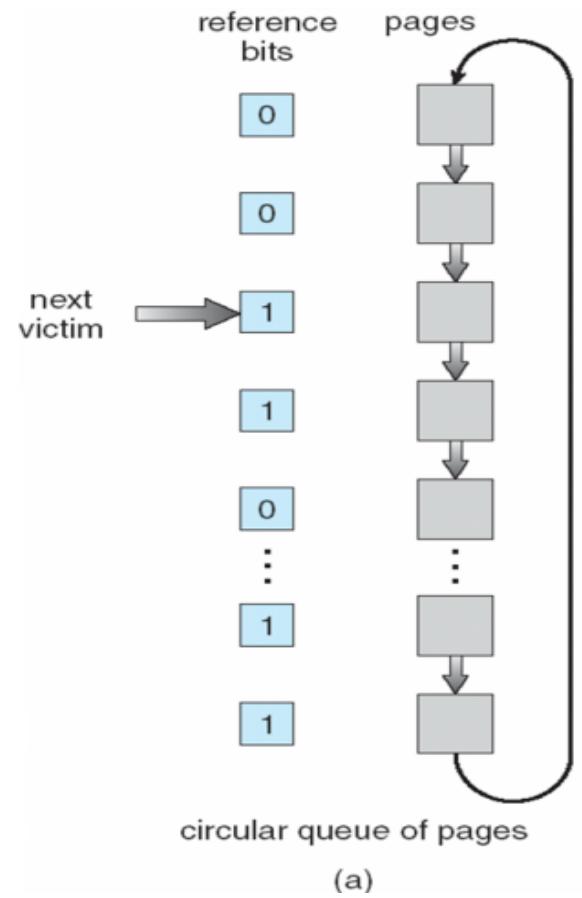


# Second Chance / CLOCK replacement





## Second-Chance (clock) Page-Replacement Algorithm





# Enhanced Second-Chance Algorithm

- Consider also the **reference bits** and the **modified bits** of pages
  - Reference (R) bit: page is referenced in the last interval
  - Dirty/Modified (M) bit: page is modified after being loaded into memory
- Four possible cases (R,M):
  - 0,0: neither recently used nor modified
  - 0,1: not recently used but modified
  - 1,0: recently used but clean
  - 1,1: recently used and modified
- We replace the first page encountered in the lowest non-empty class.
- Rest is the same with second-chance algorithm
- *We may need to scan the list several times until we find the page to replace*

139





## Additional-reference-bits algorithm

- Additional ordering information is gained by recording the reference bits at regular intervals
- At regular intervals, a timer interrupt transfers control to OS and OS shifts the reference bit for each page to high order bit of 8-bit byte shifting other bits to right by 1 bit
- Last 8 time period history of page is stored in this 8 bits.





## Contd...

- 1000000-page not used for the last 8 time periods
- 11111111-page used once in each time period
- 11000100-used recently
- 10111011-not used recently
- The page with a lowest number(integer) is the LRU page.





# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **LFU Algorithm(least frequently used)**: replaces page with smallest count
- **MFU Algorithm(most frequently used)**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used





# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected





## Dirty Pages

- If a page has been written to, it is *dirty* – ***Dirty bit for page in page table set to 1***
- Before a dirty page can be replaced it must be written to disk
- A *clean* page does not need to be written to disk
  - the copy on disk is already up-to-date
- We would rather replace an old, clean page than an old, dirty page





# Cleaning Policy

- A cleaning policy is concerned with determining when a modified page should be written out to secondary memory.
- Demand cleaning
  - A page is written out only when it has been selected for replacement
- Precleaning
  - Pages are written out in batches





## Cleaning Policy

- Best approach uses page buffering
- Replaced pages are placed in two lists
  - Modified and unmodified
- Pages in the modified list are periodically written out in batches
- Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page





## Allocation of Frames

- Each process needs *minimum* number of frames
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations of schemes also available





## 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
  - Can also keep some as **free frame buffer pool**

- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} ? 64 ? 5$$

$$a_2 = \frac{127}{137} ? 64 ? 59$$





# Priority Allocation

- Use a **proportional allocation scheme using priorities rather than size**
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames
  - select for replacement 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 – **page stealing**
  - 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





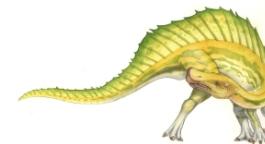
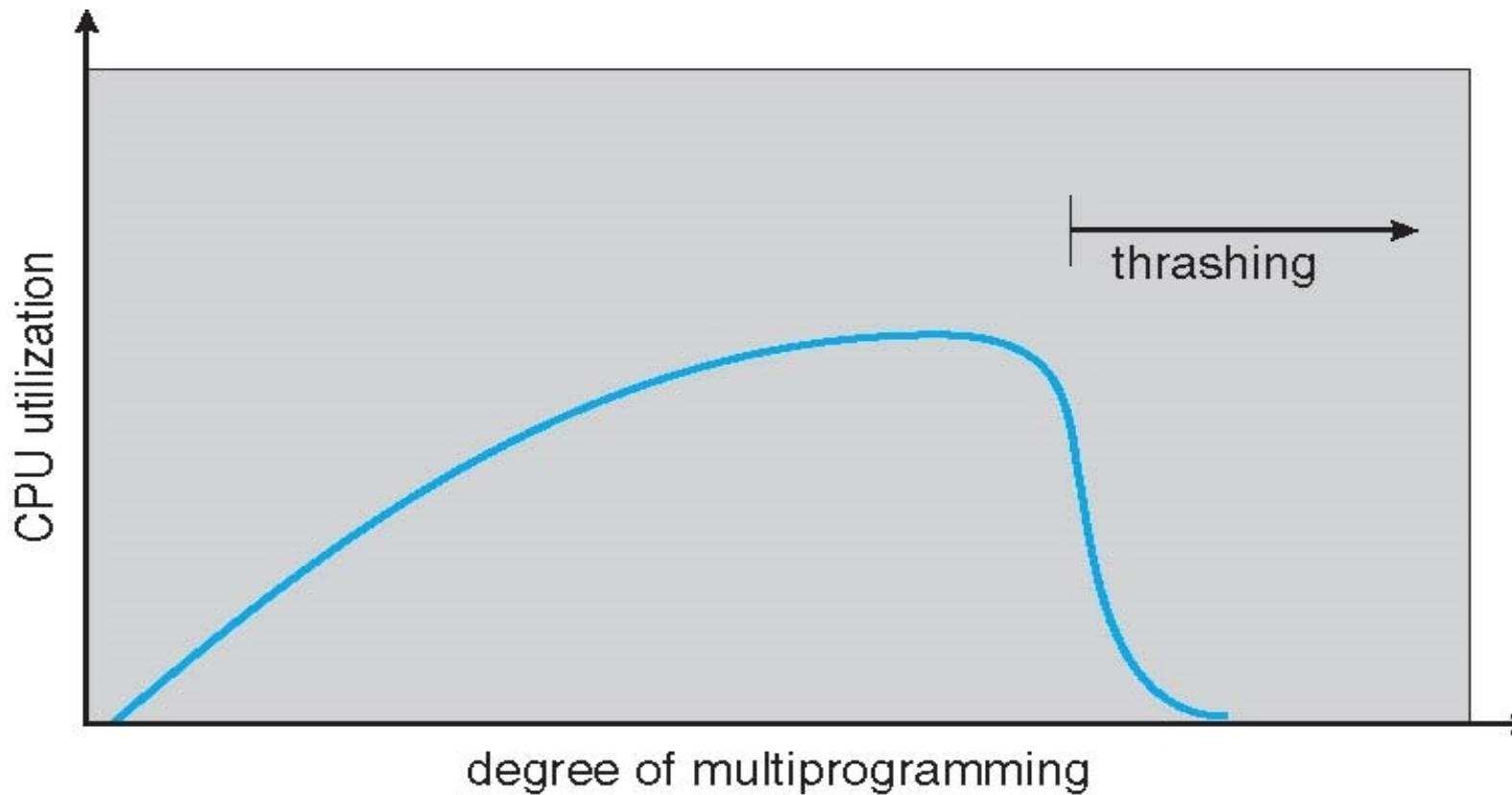
# Thrashing

- If a process does not have “enough” pages in main memory, the page-fault rate is very high
  - Page fault to get page
  - Replace existing page in a frame
  - But quickly need replaced page back
  - This leads to:
    - ▶ Low CPU utilization
    - ▶ Operating system thinking that it needs to increase the degree of multiprogramming- bringing more processes into memory
    - ▶ Another process added to the system
    - ▶ Degree of multiprogramming taken care of by **long term scheduler**
    - ▶ Global page replacement also leads to thrashing
- **Thrashing** ≡ a process is busy swapping pages in and out





## Thrashing (Cont.)





## Thrashing

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution.
- We should then page out its remaining pages, freeing all its allocated frames.
- This provision introduces a swap-in, swap-out level of intermediate CPU scheduling (**medium term scheduler**).
- And the process can be restarted later when required number of frames are available for its execution.
- This prevents thrashing for this process.





# Demand Paging and Thrashing

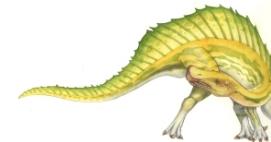
- How does demand paging work?  
**Locality model**
- *Resident Set* - *The set of pages of a process that are currently in main memory*
- *Locality of Reference* - *A locality is a set of pages that are actively used together*
  - Process migrates from one locality to another, during function calls
  - Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
- Why does thrashing occur? -  $\Sigma$  size of locality > total memory size
- Prevent Thrashing
  - By *providing a process with as many frames as it needs*. But how do we know how many frames it "needs"? There are several techniques. **The working-set strategy** starts by looking at how many frames a process is actually using.
  - By using local or priority page replacement, if one process starts thrashing, it cannot take frames from another process and cause the latter to thrash as well.





## Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
Example: 10
- $WSS_i$  (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
- Total demand frames  $D = \sum WSS_i$ , and Total Available frames is  $m$   
Then if  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

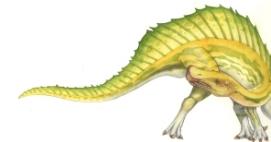




# Working-set model

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...





# Working-set model

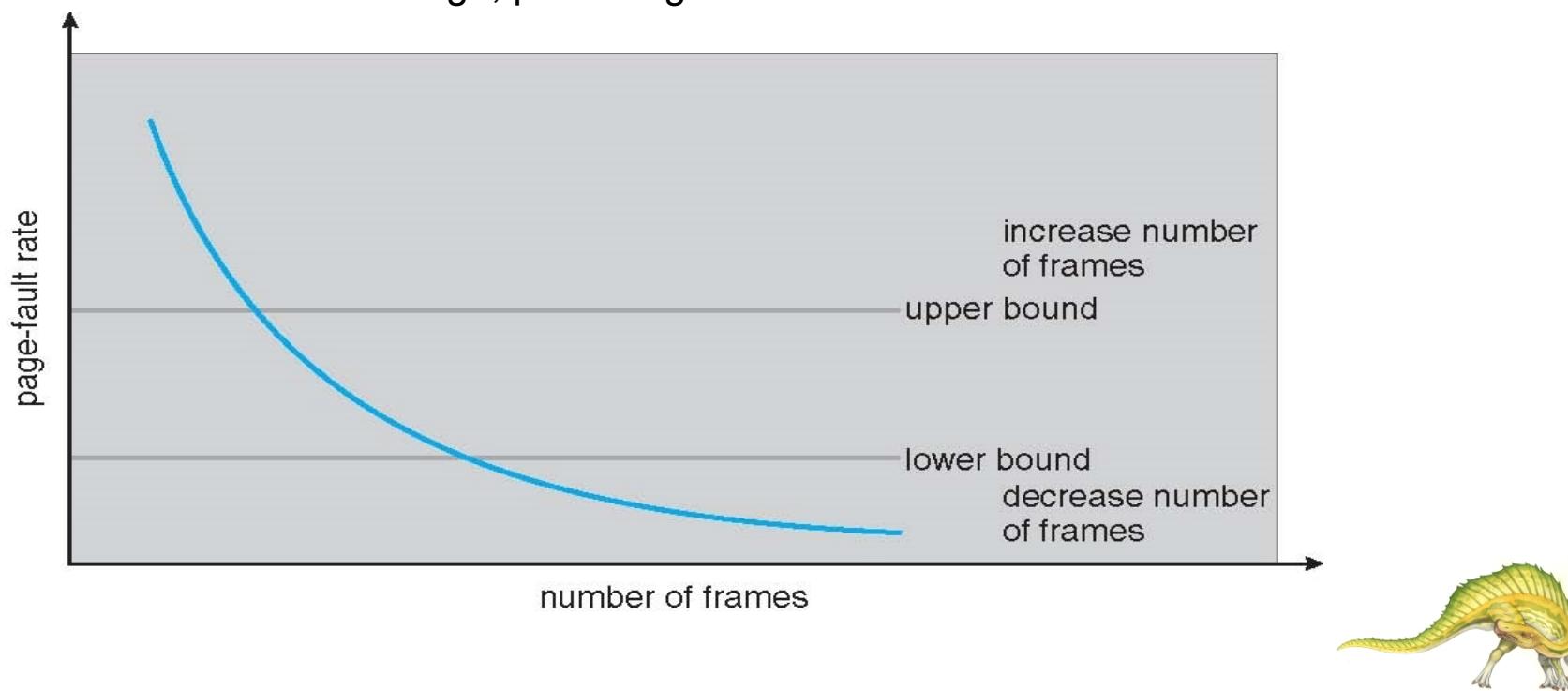
Sequence of Page References	Window Size, $\Delta$			
W	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18





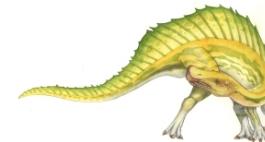
# Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** 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





# Program Structure

- Demand paging is designed to be transparent to the user program.
- In many cases, the user is completely unaware of the paged nature of memory.
- In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.
- Let's look at a contrived but informative example. Assume that pages are 128 words in size.
- Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array. The following code is typical:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```





## Program Structure

- Notice that the array is stored row major; that is, the array is stored  $\text{data}[0][0]$ ,  $\text{data}[0][1]$ , . . . ,  $\text{data}[0][127]$ ,  $\text{data}[1][0]$ ,  $\text{data}[1][1]$ , . . . ,  $\text{data}[127][127]$ .
- For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on.
- If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in  $128 \times 128 = 16,384$  page faults.
- In contrast, suppose we change the code to

```
int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

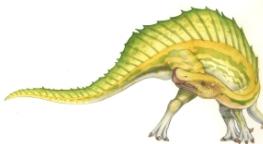
- This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.





# Program Structure

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. For example, a stack has good locality, since access is always made to the top. A hash table, in contrast, is designed to scatter references, producing bad locality. Of course, locality of reference is just one measure of the efficiency of the use of a data structure. Other heavily weighted factors include search speed, total number of memory references, and total number of pages touched.





# Program Structure

At a later stage, the compiler and loader can have a significant effect on paging. Separating code and data and generating reentrant code means that code pages can be read-only and hence will never be modified. Clean pages do not have to be paged out to be replaced. The loader can avoid placing routines across page boundaries, keeping each routine completely in one page. Routines that call each other many times can be packed into the same page. This packaging is a variant of the bin-packing problem of operations research: try to pack the variable-sized load segments into the fixed-sized pages so that interpage references are minimized. Such an approach is particularly useful for large page sizes.





# Demand Paging and Prepaging

## ■ Demand paging

- only brings pages into main memory when a reference is made to a location on the page
- Many page faults when process first started

## ■ Prepaging

- brings in more pages than needed
- More efficient to bring in pages that reside contiguously on the disk
- Don't confuse with "swapping"





## Page Size

- Page sizes are invariably powers of 2, generally ranging from 512 ( $2^9$ ) to 2048 Bytes.
- Why are page sizes always in powers of 2?
  - Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.





## How to select a page size?

- One concern is the size of the page table
- For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table.
- For a virtual memory of 4 MB ( $2^{22}$ ), for example, there would be 4,096 pages of 1,024 bytes but only 512 pages of 8,192 bytes.
- Because each active process must have its own copy of the page table, a large page size is desirable.





## How to select a page size?

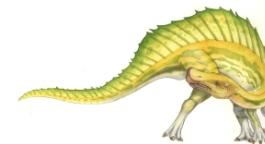
- Another concern is to minimize internal fragmentation in the last page.
- Assuming independence of process size and page size, we can expect that, on the average, half of the final page of each process will be wasted.
- This loss is only 256 bytes for a page of 512 bytes but is 4,096 bytes for a page of 8,192 bytes.
- To minimize internal fragmentation, then, we need a small page size.





## How to select a page size?

- Another problem is the time required to read or write a page.
- I/O time is composed of seek, latency, and transfer times.
- Transfer time is proportional to the amount transferred (that is, the page size)—a fact that would seem to argue for a small page size.
- If, It takes 28.4 milliseconds to read a single page of 1,024 bytes but 56.4 milliseconds to read the same amount as two pages of 512 bytes each.
- Thus, a desire to minimize I/O time argues for a larger page size.





## How to select a page size?

- Another concern is, with a smaller page size, though, total I/O should be reduced, since locality will be improved.
- A smaller page size allows each page to match program locality more accurately.
- For example, consider a process 200 KB in size, of which only half (100 KB) is actually used in an execution.
- If we have only one large page, we must bring in the entire page, a total of 200 KB transferred and allocated.
- If instead we had pages of only 1 byte, then we could bring in only the 100 KB that are actually used, resulting in only 100 KB transferred and allocated.
- With a smaller page size, then, we have better resolution, allowing us to isolate only the memory that is actually needed.
- With a larger page size, we must allocate and transfer not only what is needed but also anything else that happens to be in the page, whether it is needed or not.
- Thus, a smaller page size should result in less I/O and less total allocated memory.





## How to select a page size?

- The problem has no best answer.
- As we have seen, some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size.
- Nevertheless, the historical trend is toward larger page sizes.
- Modern systems may now use much larger page sizes.





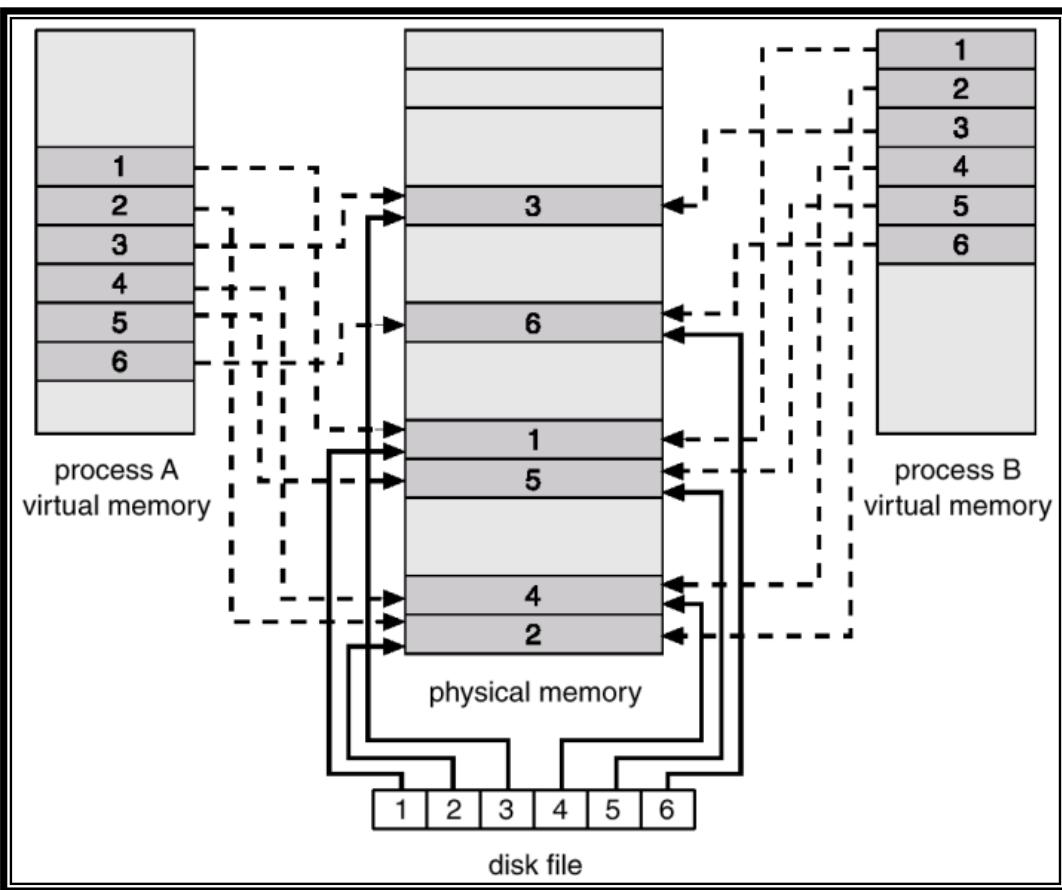
## Memory-Mapped Files

- 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 file access by treating file I/O through memory rather than **read()** **write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.





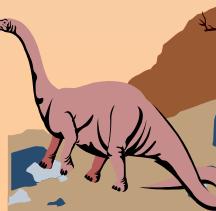
# Memory Mapped Files





# Chapter 11: File-System Interface

- File Concept
- Access Methods
- Directory Structure
- File System Mounting
- File Sharing
- Protection





# File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.

So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information.

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file.

Files are mapped by the operating system onto physical devices.

These storage devices are usually nonvolatile, so the contents are persistent between system reboots.





# File Concept

- A file is a named collection of related information that is recorded on secondary storage.
- From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.
- Representation of files as:
  - ☞ Data
    - ▀ numeric
    - ▀ character
    - ▀ binary
  - ☞ Program





# File Structure

- None - sequence of words, bytes
- Simple record structure
  - ☞ Lines
  - ☞ Fixed length
  - ☞ Variable length
- Complex Structures
  - ☞ Formatted document
  - ☞ Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters.
- Who decides:
  - ☞ Operating system
  - ☞ Program





# File Attributes

- **Name** – only information kept in human-readable form.
- **Type** – needed for systems that support different types.
- **Location** – pointer to file location on device.
- **Size** – current file size.
- **Protection** – controls who can do reading, writing, executing.
- **Time, date, and user identification** – data for protection, security, and usage monitoring.
- Information about files are kept in the directory structure, which is maintained on the disk.





# Directory Structure

- The information about all files is kept in the **directory structure**, which also resides on secondary storage.
- Typically, a directory entry consists of the file's name and its unique identifier.
- The identifier in turn locates the other file attributes.
- It may take more than a kilobyte to record this information for each file.
- In a system with many files, the size of the directory itself may be megabytes.
- Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.





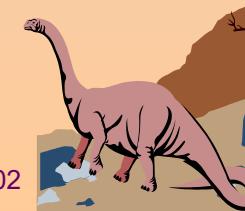
# File Operations

- Create
- Write
- Read
- Reposition within file – file seek
- Delete
- Truncate
- Open( $F_i$ ) – search the directory structure on disk for entry  $F_i$ , and move the content of entry to memory.
- Close ( $F_i$ ) – move the content of entry  $F_i$  in memory to directory structure on disk.



# Data Structures related to files

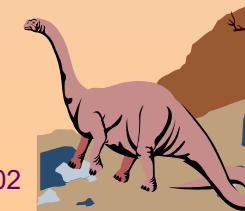
- The operating system keeps a table, called the **system-wide open-file table**, containing information about all open files.
- When a file operation is requested, the file is specified via an index into this table, so no searching is required.
- When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.





# Data Structures related to files

- The **per process file table** tracks all files that a process has open.
- Stored in this table is information regarding the process's use of the file.
- For instance, the current file pointer for each file is found here.
- Access rights to the file and accounting information can also be included.





# Data Structures related to files

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.





# Data Structures related to files

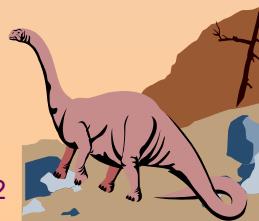
- Information to be stored for a open file
  - ☞ File pointer
  - ☞ Access rights
  - ☞ Disk location of the file
  - ☞ File open count





# File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information





# Access Methods

- Files store information.
- When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways.



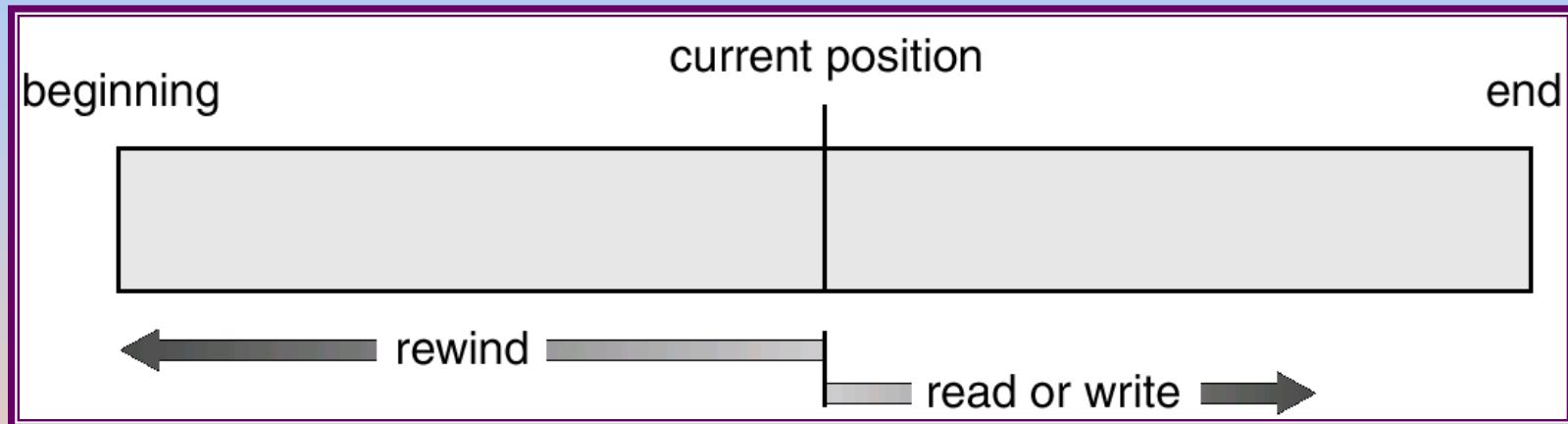
# Sequential Access

- The simplest access method is sequential access.
  - . Information in the file is processed in order, one record after the other.
- This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.





# Sequential-access File





# Direct Access

- Another method is direct access (or relative access).
  - . Here, a file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.





## Simulation of Sequential Access on a Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	$read cp;$ $cp = cp+1;$
<i>write next</i>	$write cp;$ $cp = cp+1;$





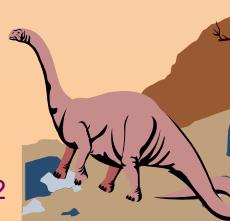
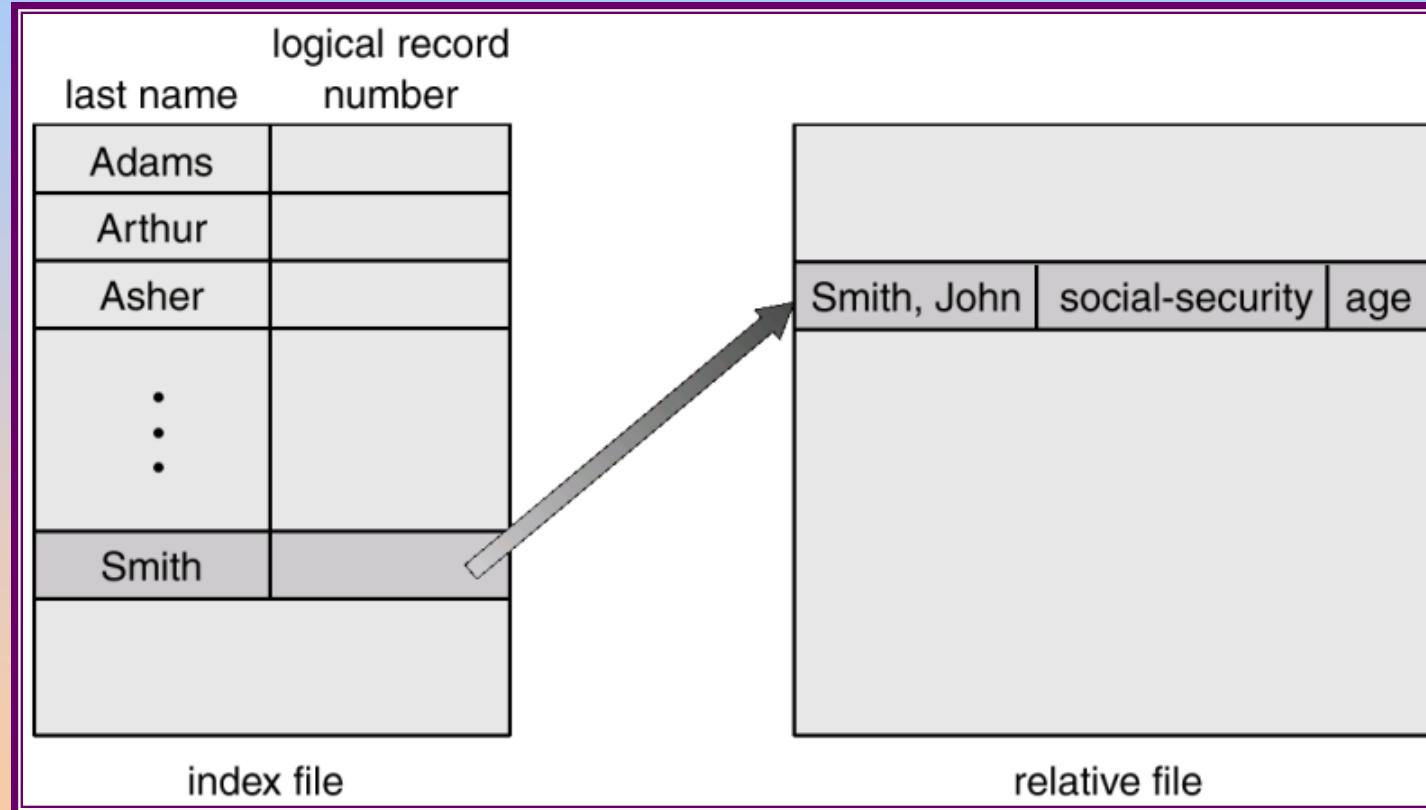
# Other Access Methods

- Other access methods can be built on top of a direct-access method.
- These methods generally involve the construction of an index for the file.
- The index, like an index in the back of a book, contains pointers to the various blocks.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.





# Example of Index and Relative Files





# Directory Structure

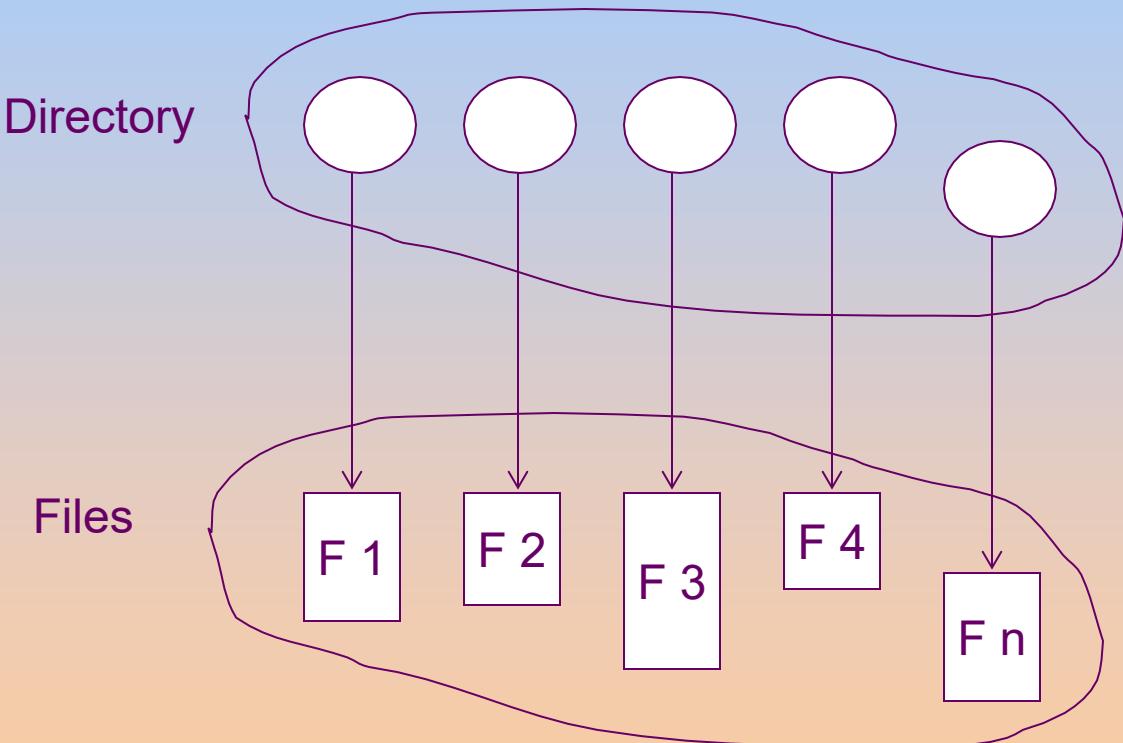
- Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks.
- A storage device can be used in its entirety for a file system.
- It can also be subdivided for finer-grained control. For example, a disk can **be partitioned** into quarters, and each quarter can hold a separate file system.
- Any **entity** containing a file system is generally known as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system.
- This information is kept in entries in a device directory or volume table of contents.
- The device directory (more commonly known simply as the directory) records information





# Directory Structure

- A collection of nodes containing information about all files.

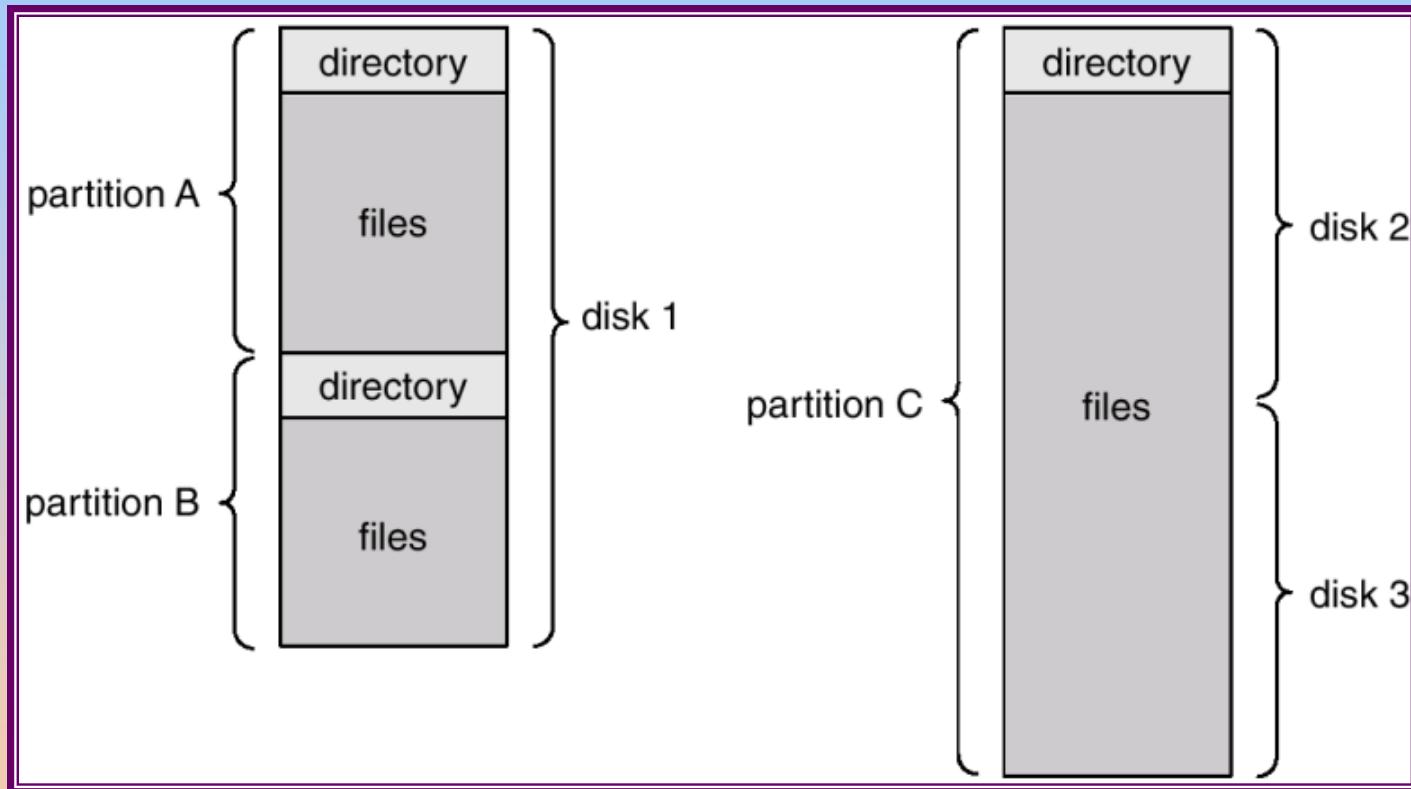


Both the directory structure and the files reside on disk.  
Backups of these two structures are kept on tapes.





# A Typical File-system Organization





# Information in a Device Directory (inode Block)

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information (discuss later)





# Operations Performed on Directory

- The directory can be viewed as a symbol table that translates file names into their directory entries.
- If we take such a view, we see that the directory itself can be organized in many ways.
- The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory as the operations shown below:
  - ☞ Search for a file
  - ☞ Create a file
  - ☞ Delete a file
  - ☞ List a directory
  - ☞ Rename a file
  - ☞ Traverse the file system





# Organize the Directory (Logically) to Obtain

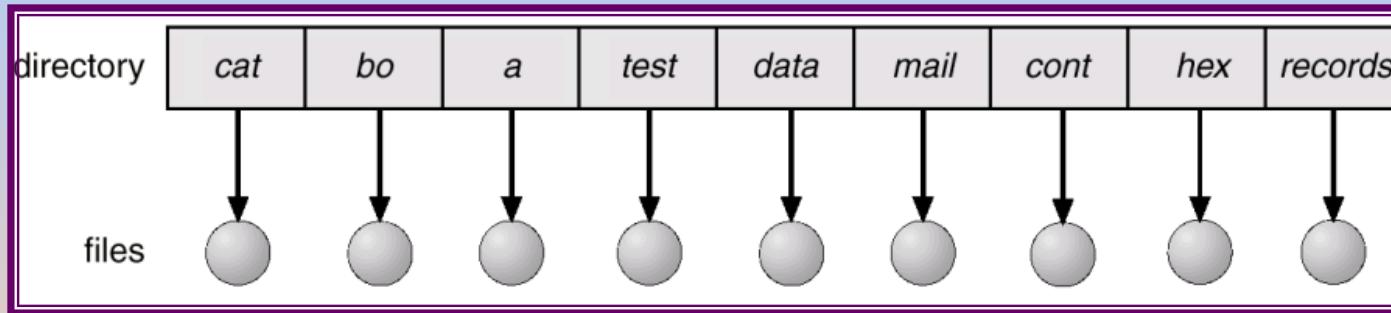
- **Efficiency** – locating a file quickly.
- **Naming** – convenient to users.
  - ☞ Two users can have same name for different files.
  - ☞ The same file can have several different names.
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





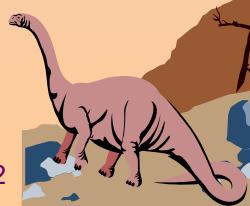
# Single-Level Directory

- A single directory for all users.



Naming problem

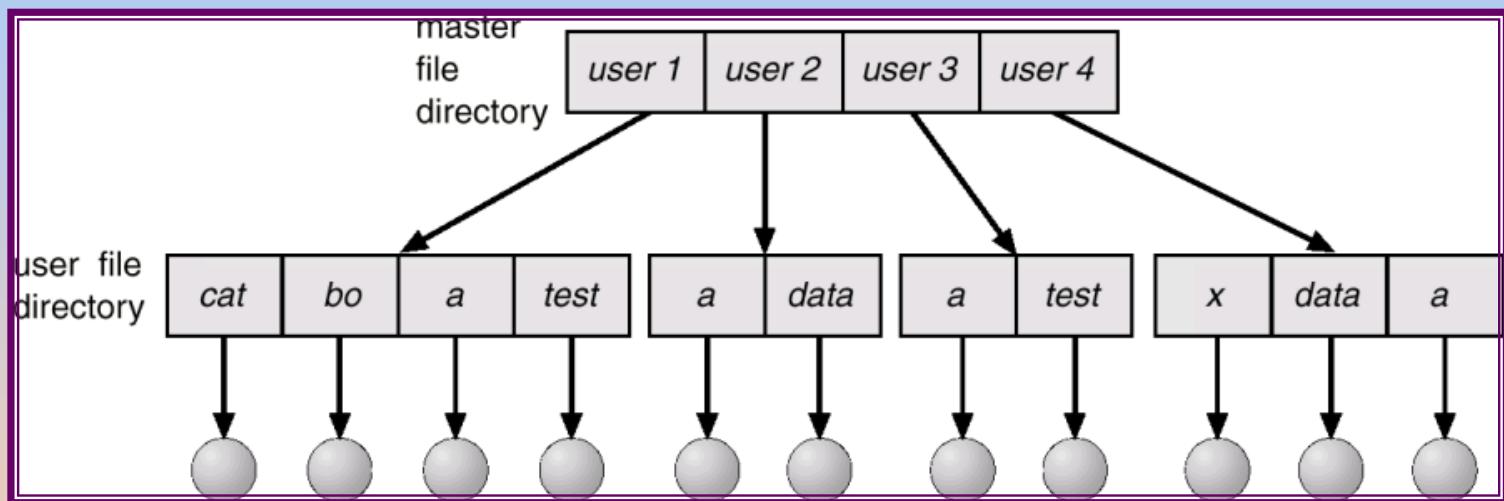
Grouping problem





# Two-Level Directory

- Separate directory for each user. User1/cat/f1.c



nPath name

nCan have the same file name for different user

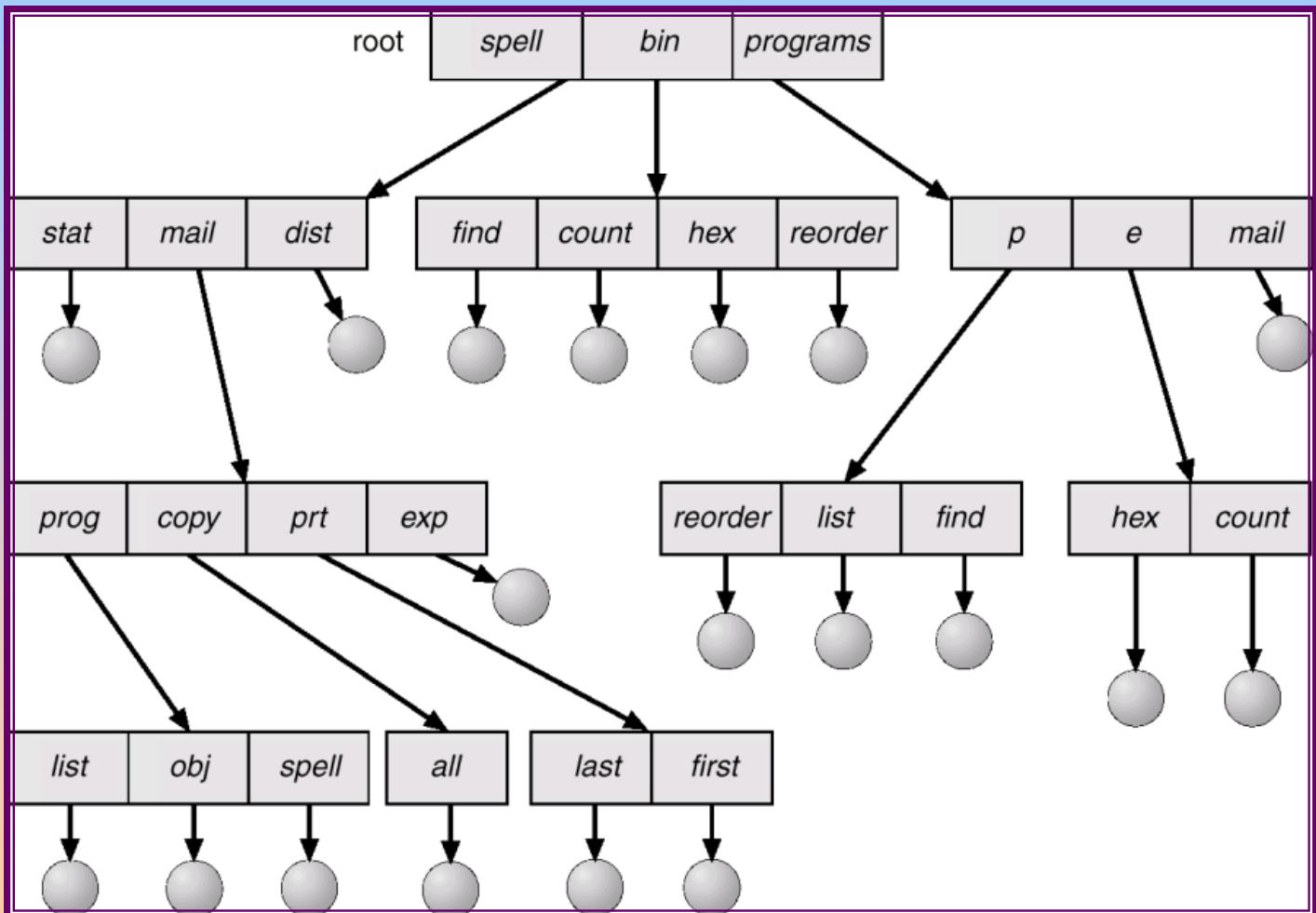
nEfficient searching

nNo grouping capability





# Tree-Structured Directories





# Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Used in many OS (like Linux)
  - Current directory (working directory)
    - ☞ `cd /spell/mail/prog`
    - ☞ `type list`





# Tree-Structured Directories (Cont.)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory.
- Delete a file

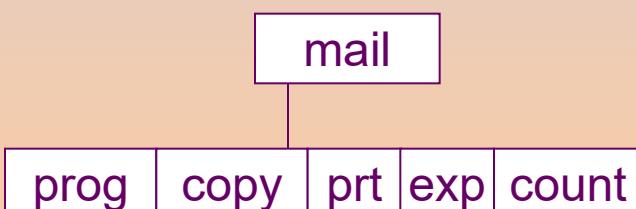
**rm <file-name>**

- Creating a new subdirectory is done in current directory.

**mkdir <dir-name>**

Example: if in current directory **/mail**

**mkdir count**



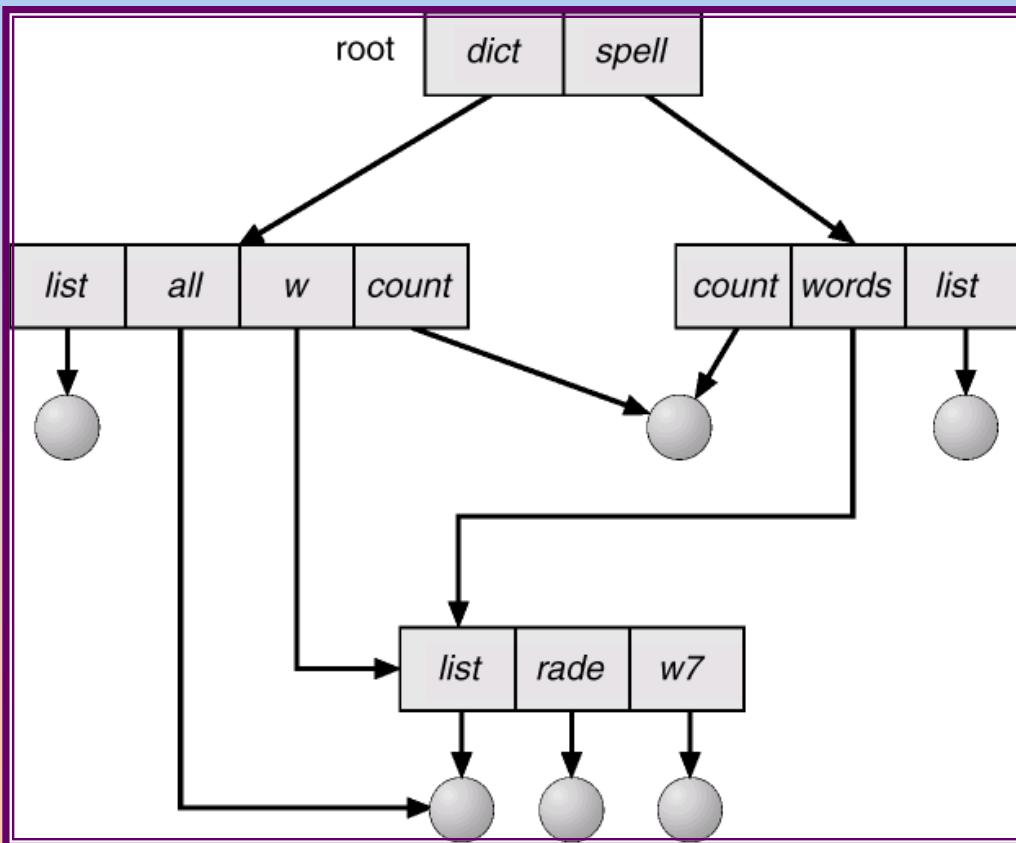
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”.





# Acyclic-Graph Directories

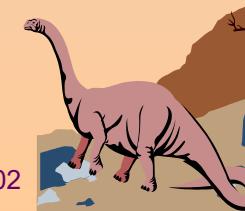
- Have shared subdirectories and files.





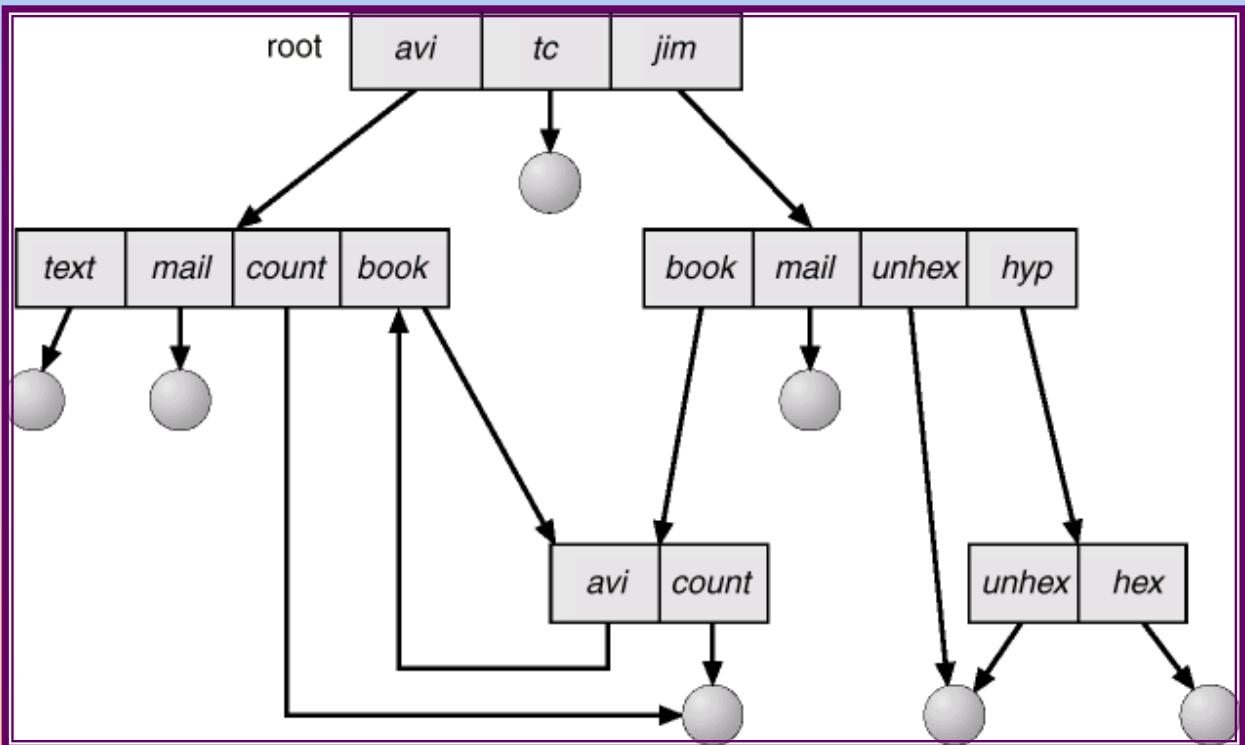
# Acyclic-Graph Directories (Cont.)

- Two different names (aliasing) - links
- If *dict* deletes *list*  $\Rightarrow$  dangling pointer.  
Solutions:
  - ☞ Backpointers, so we can delete all pointers.  
Variable size records a problem.
  - ☞ Backpointers using a daisy chain organization.
  - ☞ Entry-hold-count solution.





# General Graph Directory





# General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - ☞ Allow only links to file not subdirectories.
  - ☞ Garbage collection.
  - ☞ Every time a new link is added use a cycle detection algorithm to determine whether it is OK.





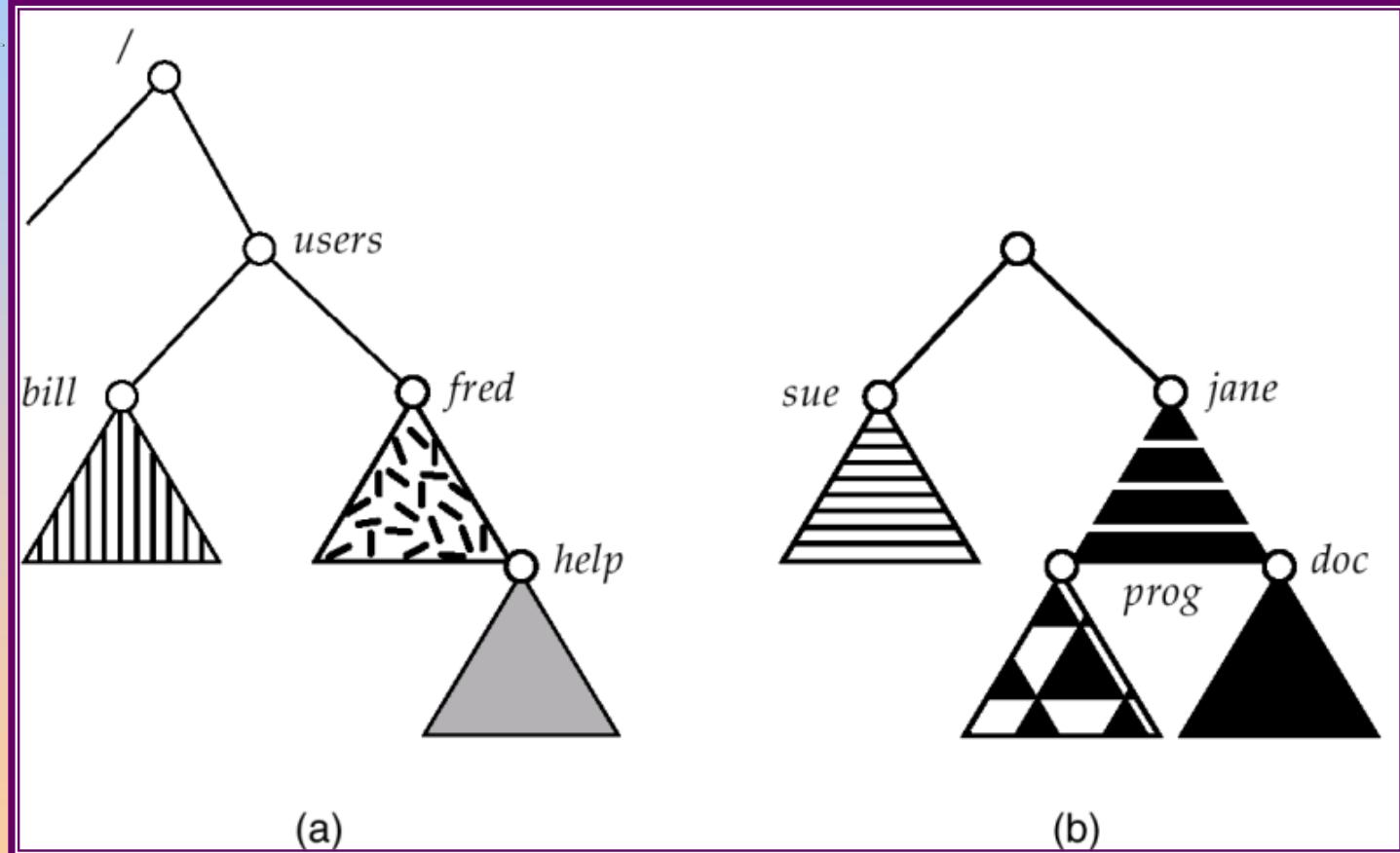
# File System Mounting

- A file system must be **mounted** before it can be accessed - **attaching portion of the file system into a directory structure.**
- A unmounted file system (I.e. Fig. 11-11(b)) is mounted at a **mount point**.

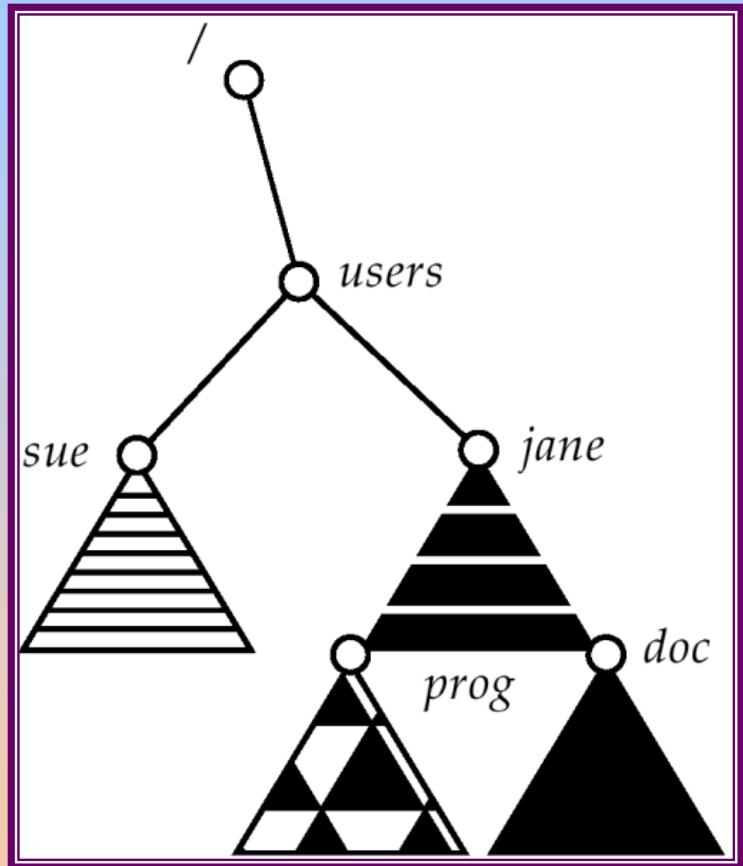




## (a) Existing. (b) Unmounted Partition



# Mount Point





# File Sharing

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a *protection* scheme.
- On distributed systems, files may be shared across a network.
- Network File System (NFS) is a common distributed file-sharing method.
- NTFS – New Technology File System



# Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
  
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List



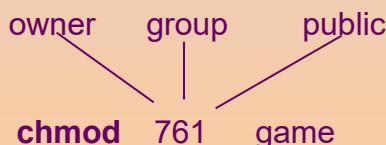


# Access Control Lists(ACL) and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) owner access	7	⇒	4 2 1
			RWX
b) group access	6	⇒	4 2 0
			RWX
c) public access	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

```
chgrp G game
```



# OPERATING SYSTEMS

INTRODUCTION

# WHY STUDY OPERATING SYSTEMS?

- Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating system.
- Why, then, study operating systems and how they work?
  - Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming.
  - Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

# What is Operating System (OS)?

- A Computer consist of various hardwares such as



Processor



RAM



Keyboard & Mouse



Hard Disk



Monitor



Printer

**Who manages (controls) these hardwares??? Operating System**

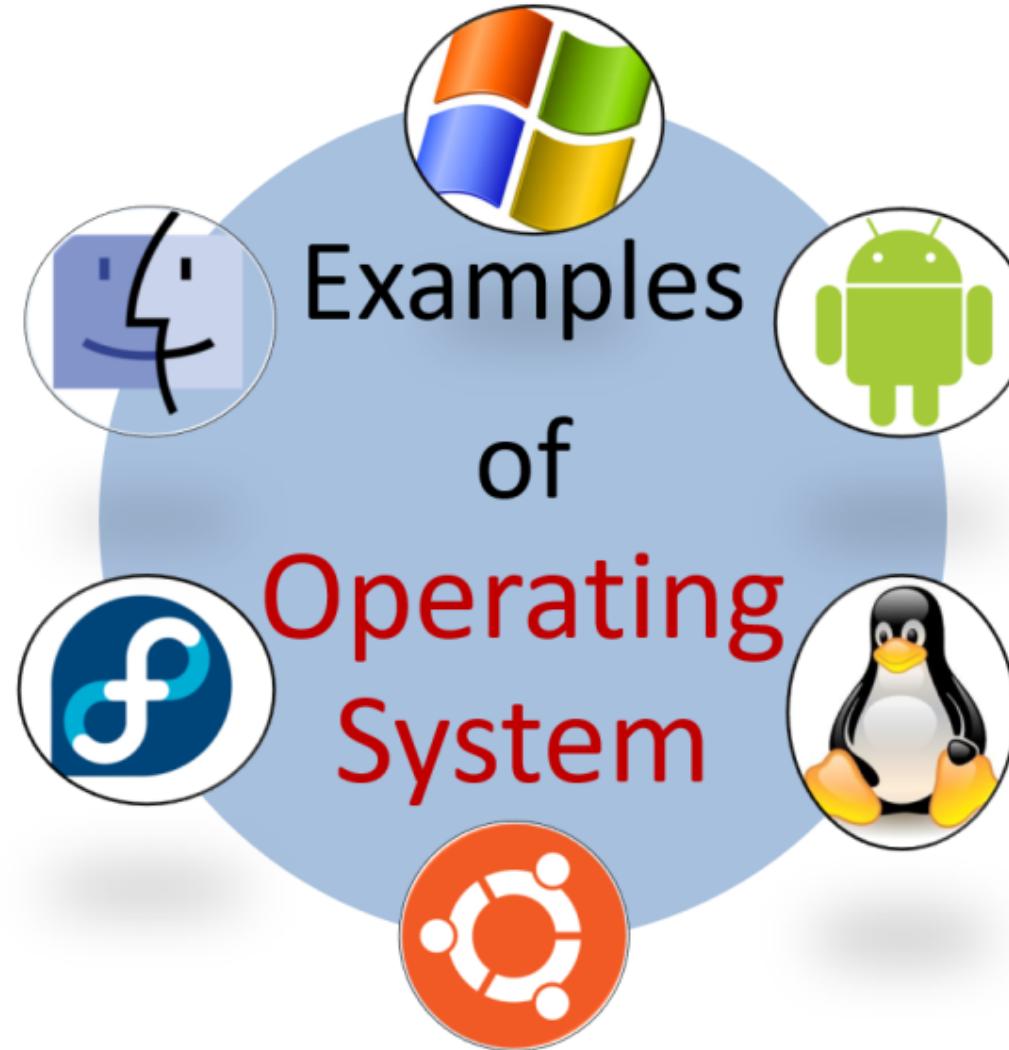
# Definition of Operating System

---

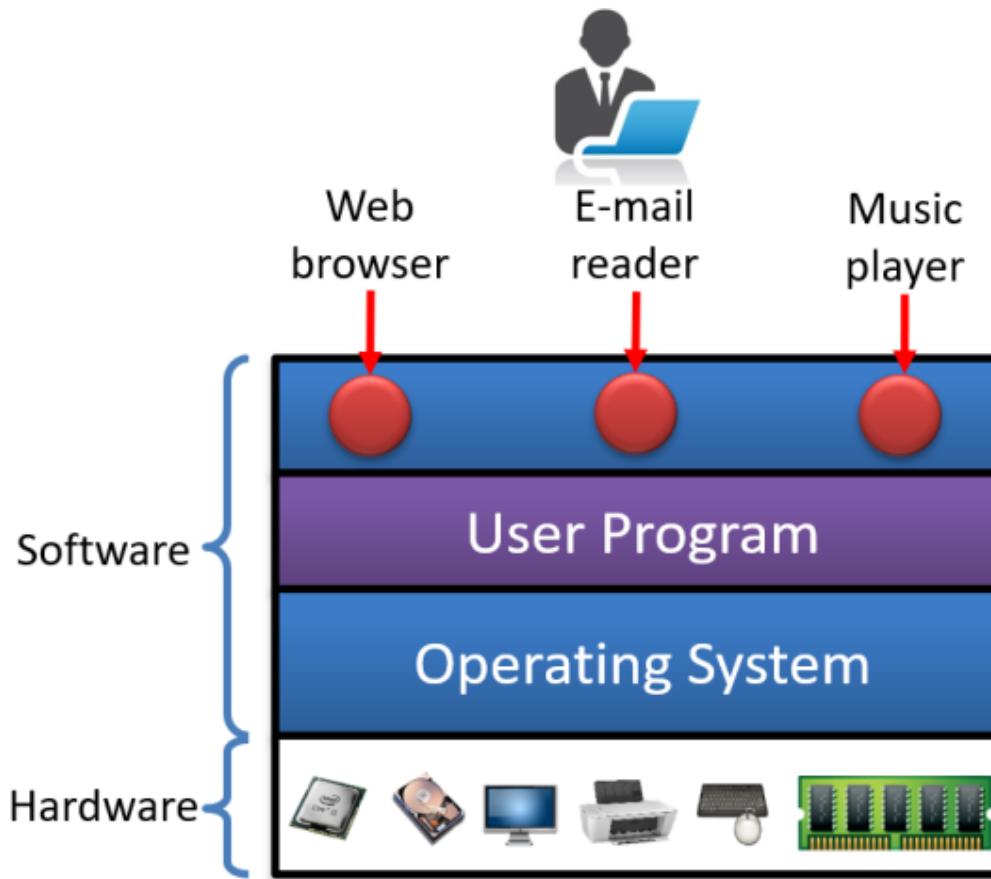
- An Operating System (OS) is a collection of software that
  - manages hardware resources
  - provides various service to the users

# Examples of Operating System

---



# Where OS lies?



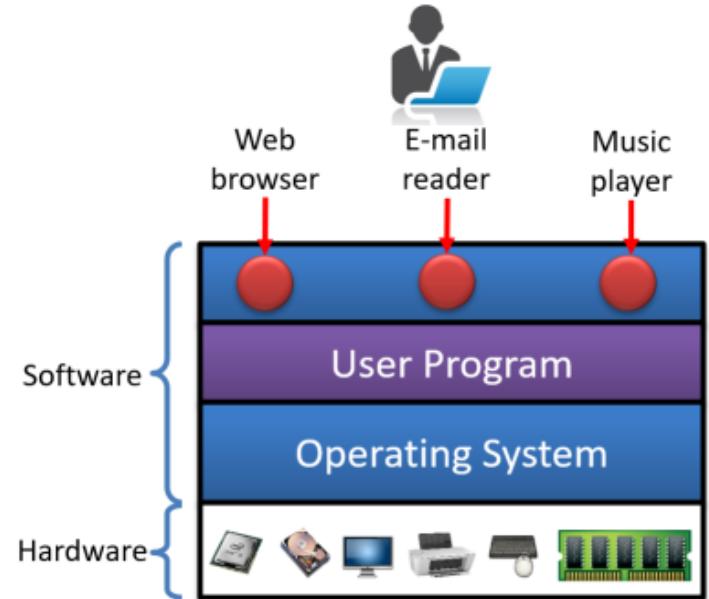
- OS **lies between hardware and user program**.
- It **acts as an intermediary** between the user and the hardware.

# Objectives of Operating Systems

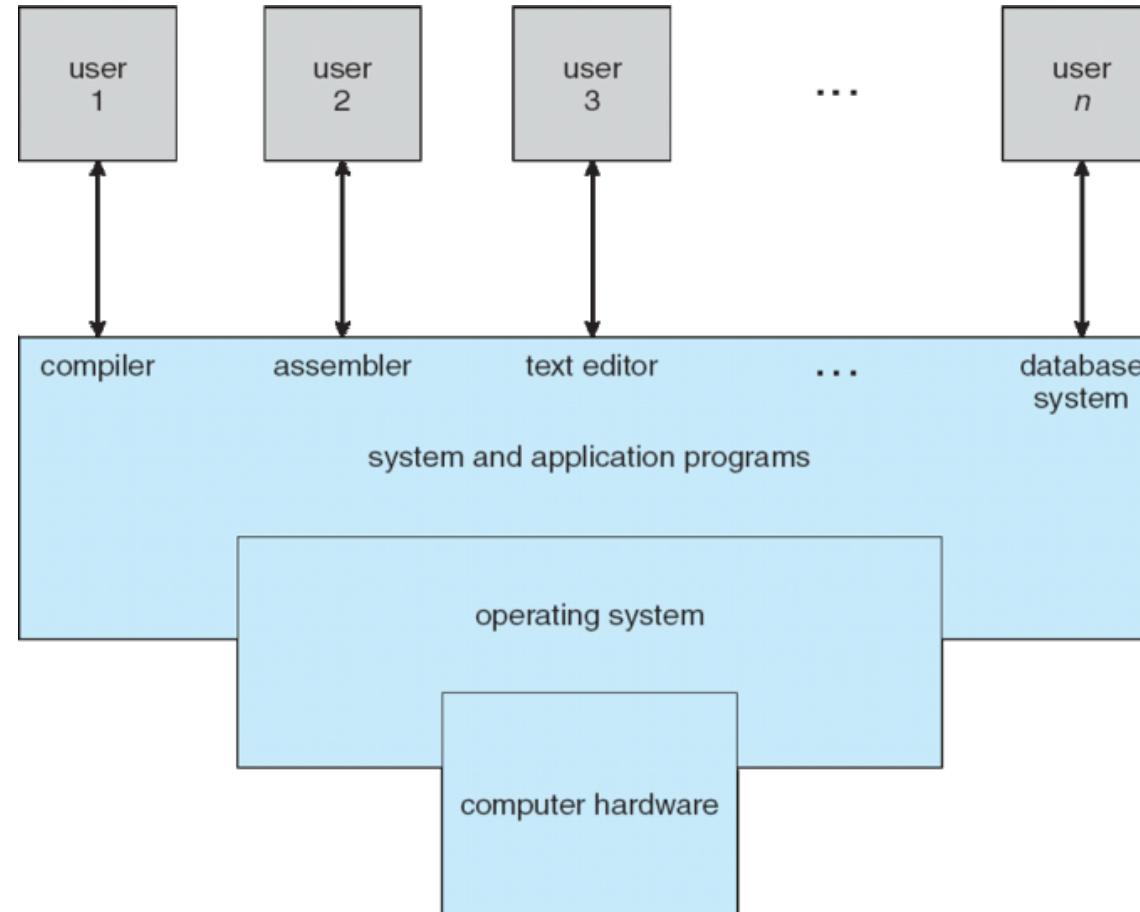
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

# 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
  - 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



# Operating System Definition

- The fundamental goal of computer systems is to execute programs and to make solving user problems easier.
- Computer hardware is constructed toward this goal.
- 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 then brought together into one piece of software: the operating system

# Operating System Definition

- A more common definition, is that the operating system is “**the one program running at all times on the computer**” - usually called the **kernel**.
- The kernel is the central module of an operating system (OS).
- It is the part of the operating system that loads first, and it remains in main memory.
- Typically, the kernel is responsible for memory management, process management, file management, and disk management.

# Operating System Definition

- The kernel connects the system hardware to the application software.
- Every operating system has a kernel.
  - For example the Linux kernel is used numerous operating systems including Linux, FreeBSD, Android and others.
- Along with the kernel, there are two other types of programs:
- **System Software:**
  - System programs(system software), which are associated with the operating system but are not necessarily part of the kernel.
  - System software is a collection of programs designed to operate, control and extend the processing capabilities of the computer. They interact with the hardware at basic level.

# Operating System Definition

- Even the operating system is considered as a system software.
- Examples for system software:
  - device drivers, compilers, assemblers, interpreters and so on.
- **Application Software:**
  - Application programs, which include all programs not associated with the operation of the system.
  - They are designed to satisfy a particular task of a particular environment.
  - Example :
    - database programs, word processors, web browsers and spreadsheets, payroll software, inventory management software etc.

# 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
- Resources managed by OS include CPU cycles, memory, file storage, peripheral devices and network connections.

# OS as Resource Manager

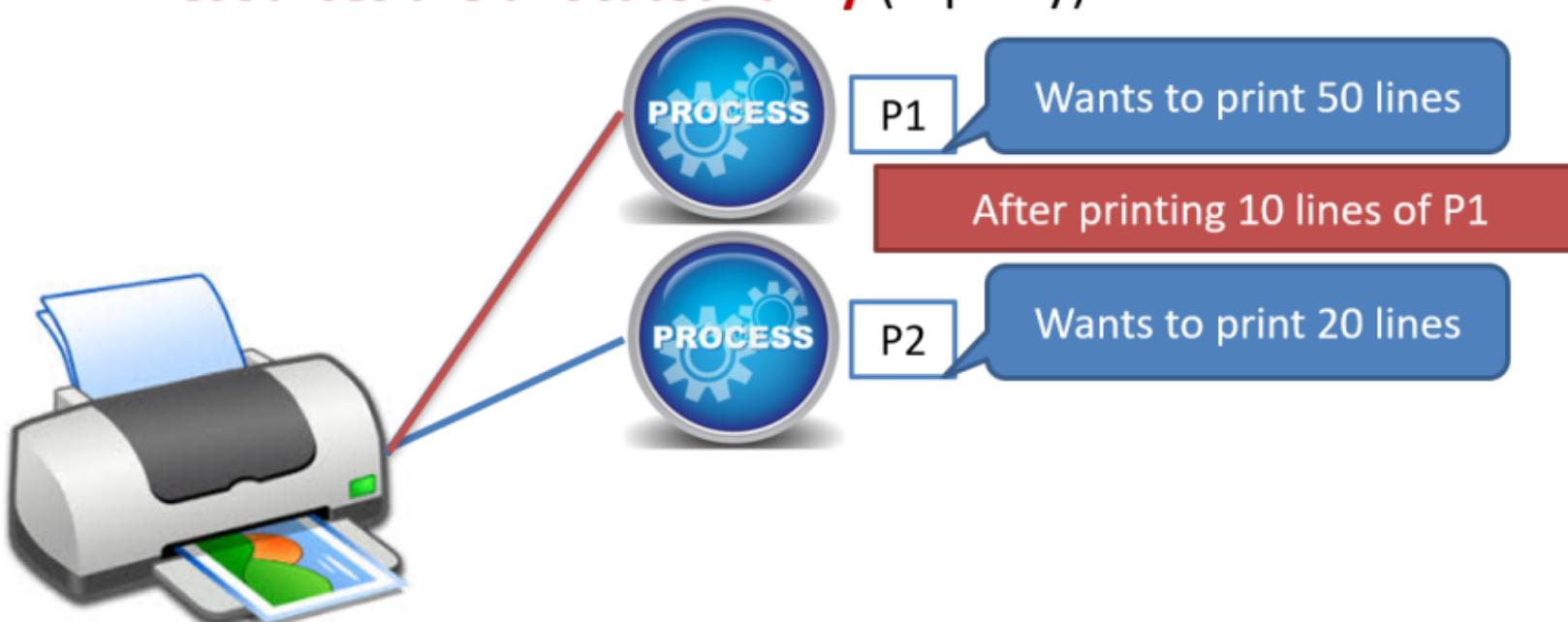
---

- There are lots of resources in computer system
  - CPU (Processor)
  - Memory
  - I/O devices such as hard disk, mouse, keyboard, printer, scanner etc.
- If a computer system is used by **multiple applications (or users)**, then they will **compete for these resources**.



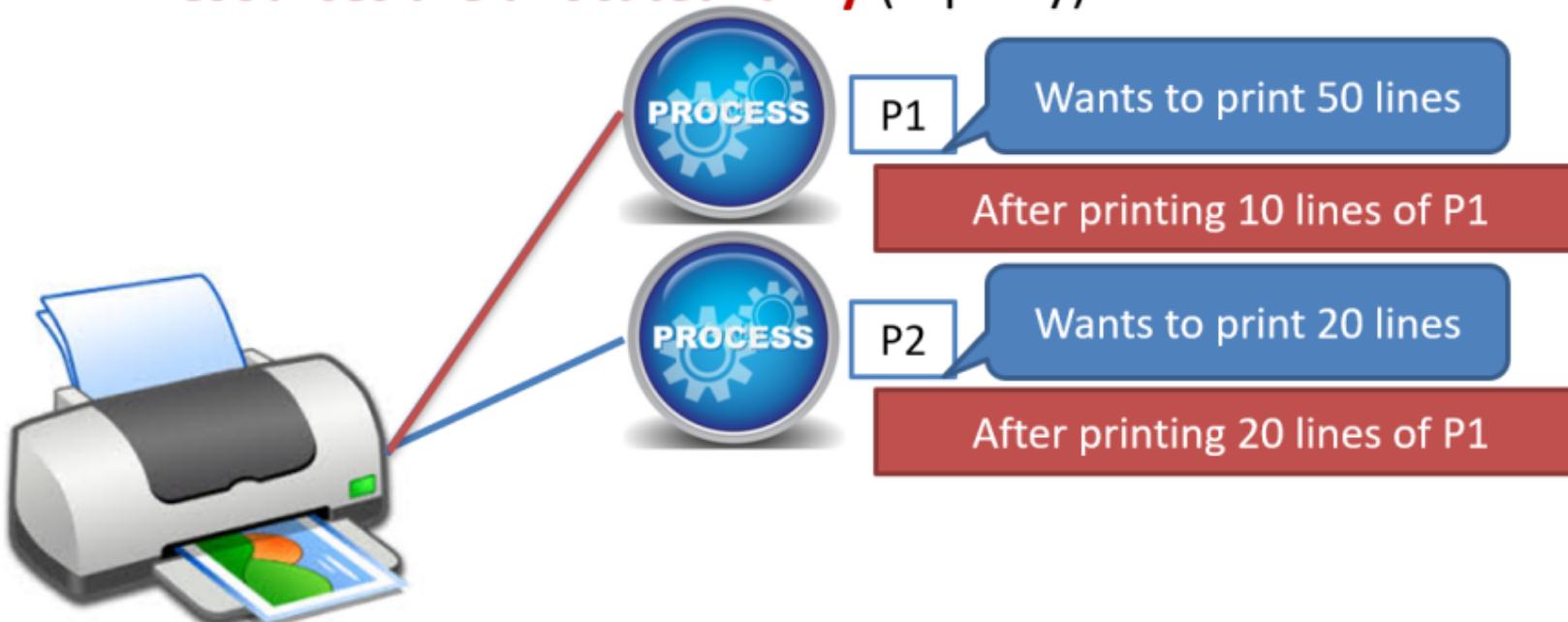
# OS as Resource Manager

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



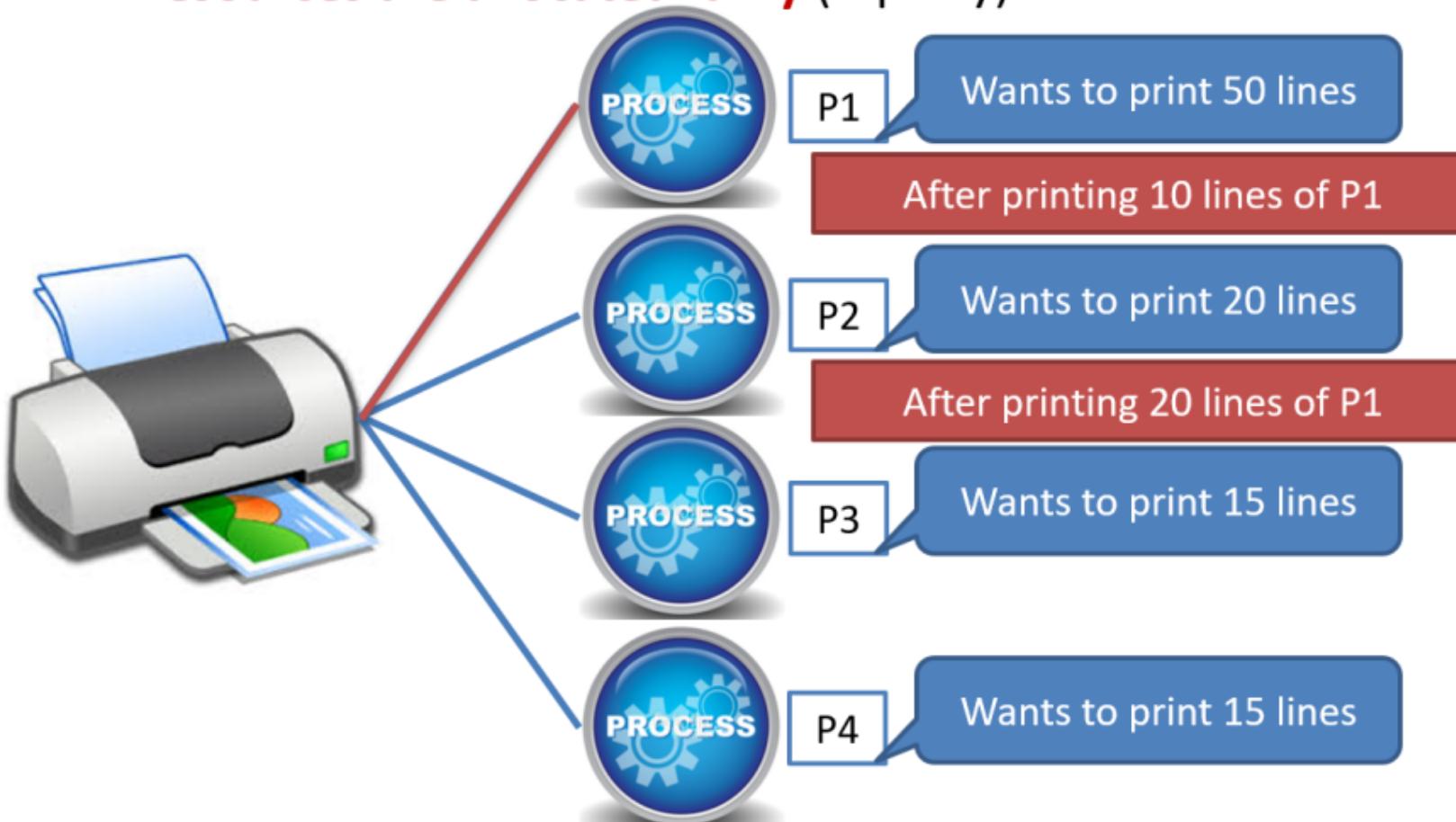
# OS as Resource Manager

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



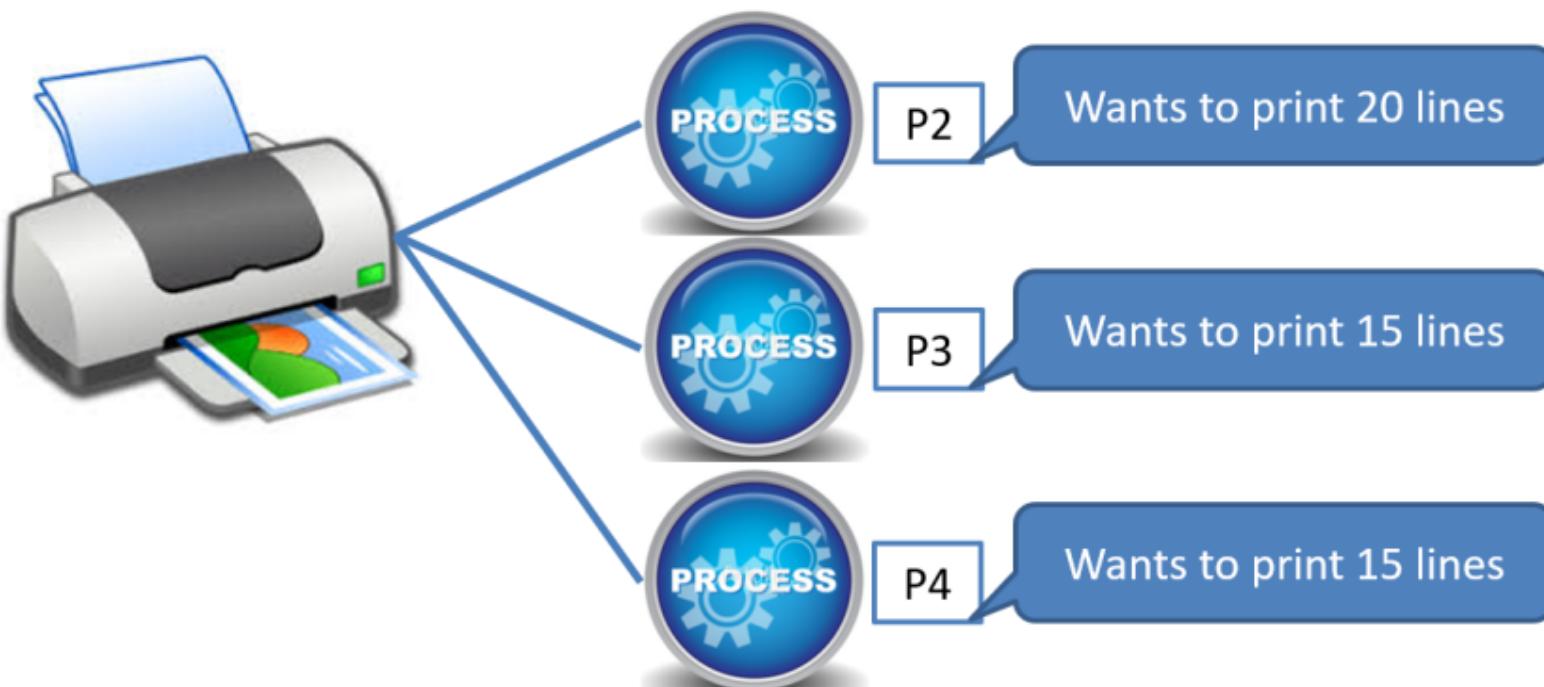
# OS as Resource Manager

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



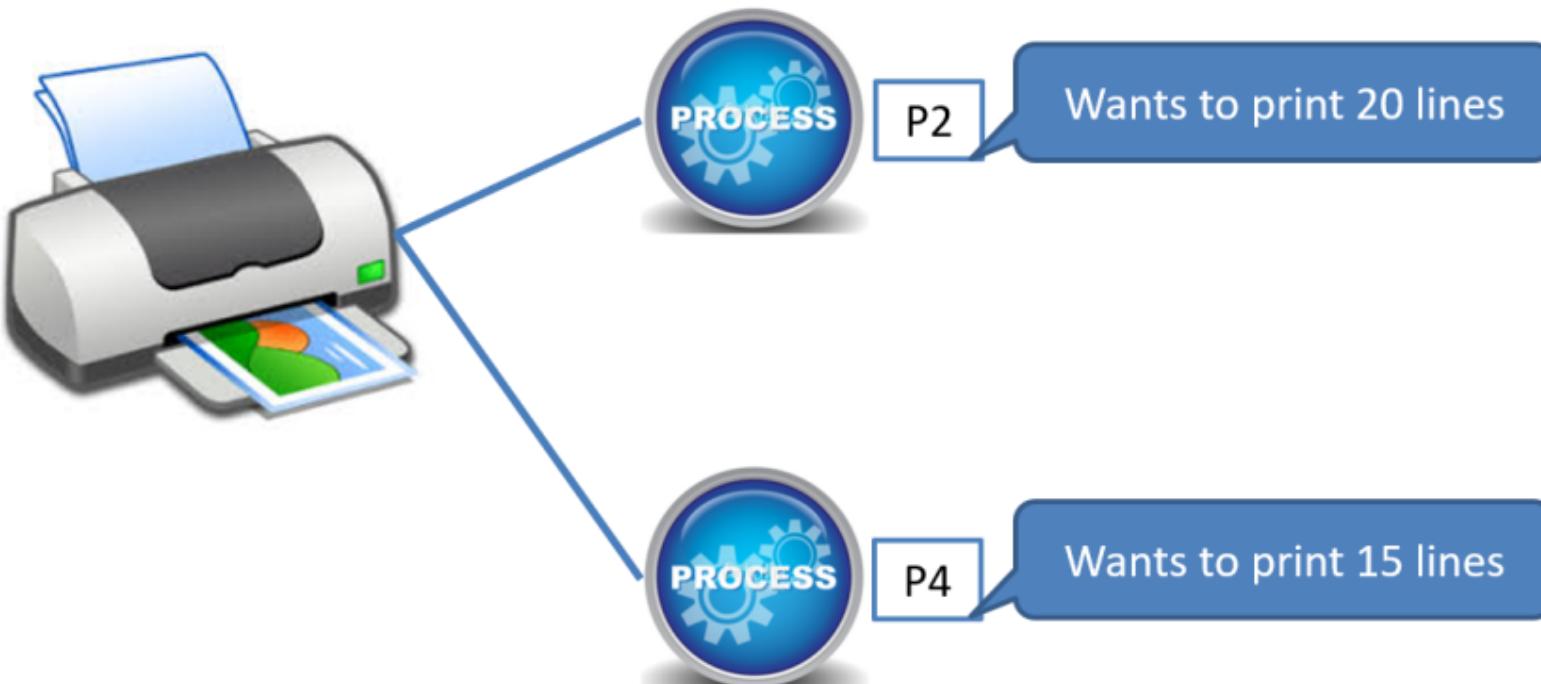
# OS as Resource Manager

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



# OS as Resource Manager

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



# OS as Resource Manager

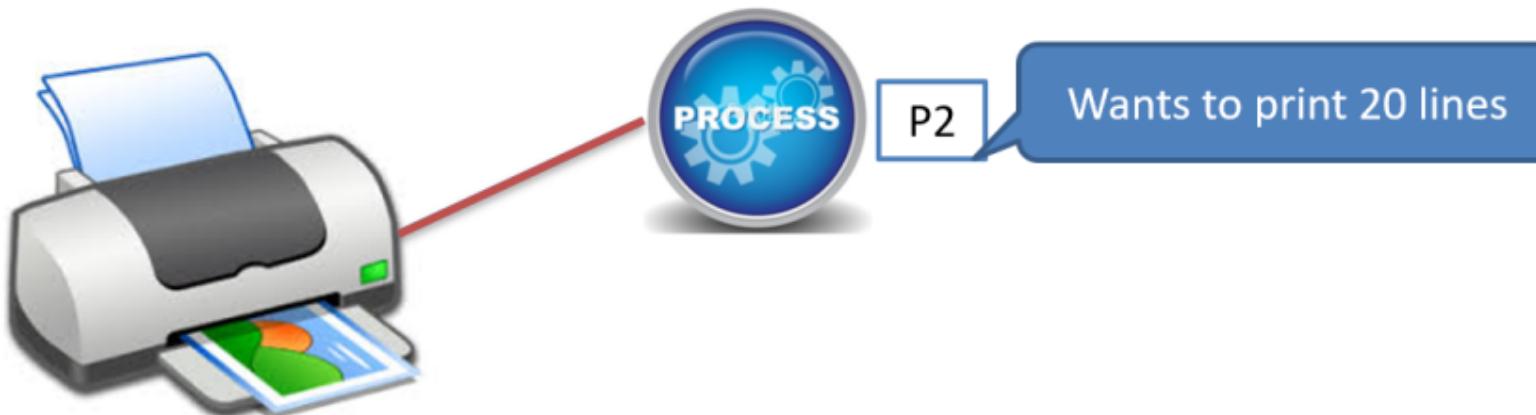
- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



# OS as Resource Manager

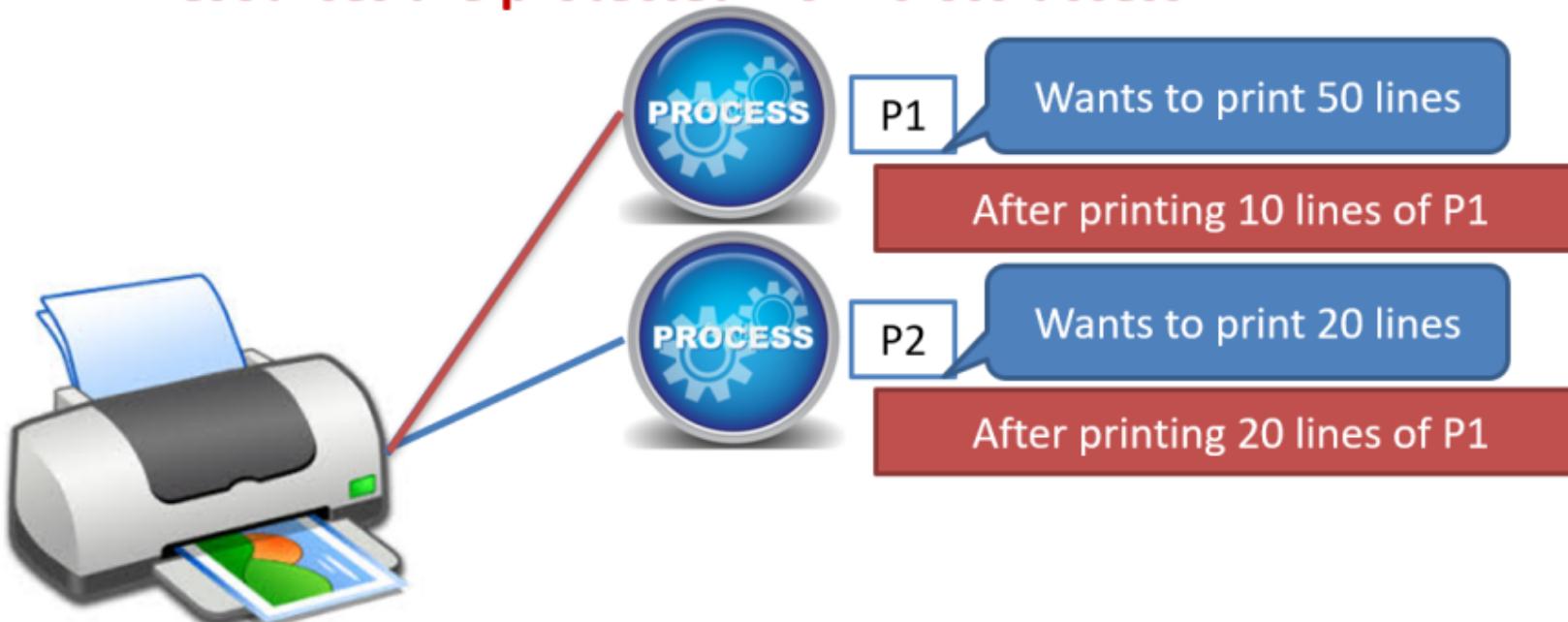
---

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are allocated fairly** (equally)



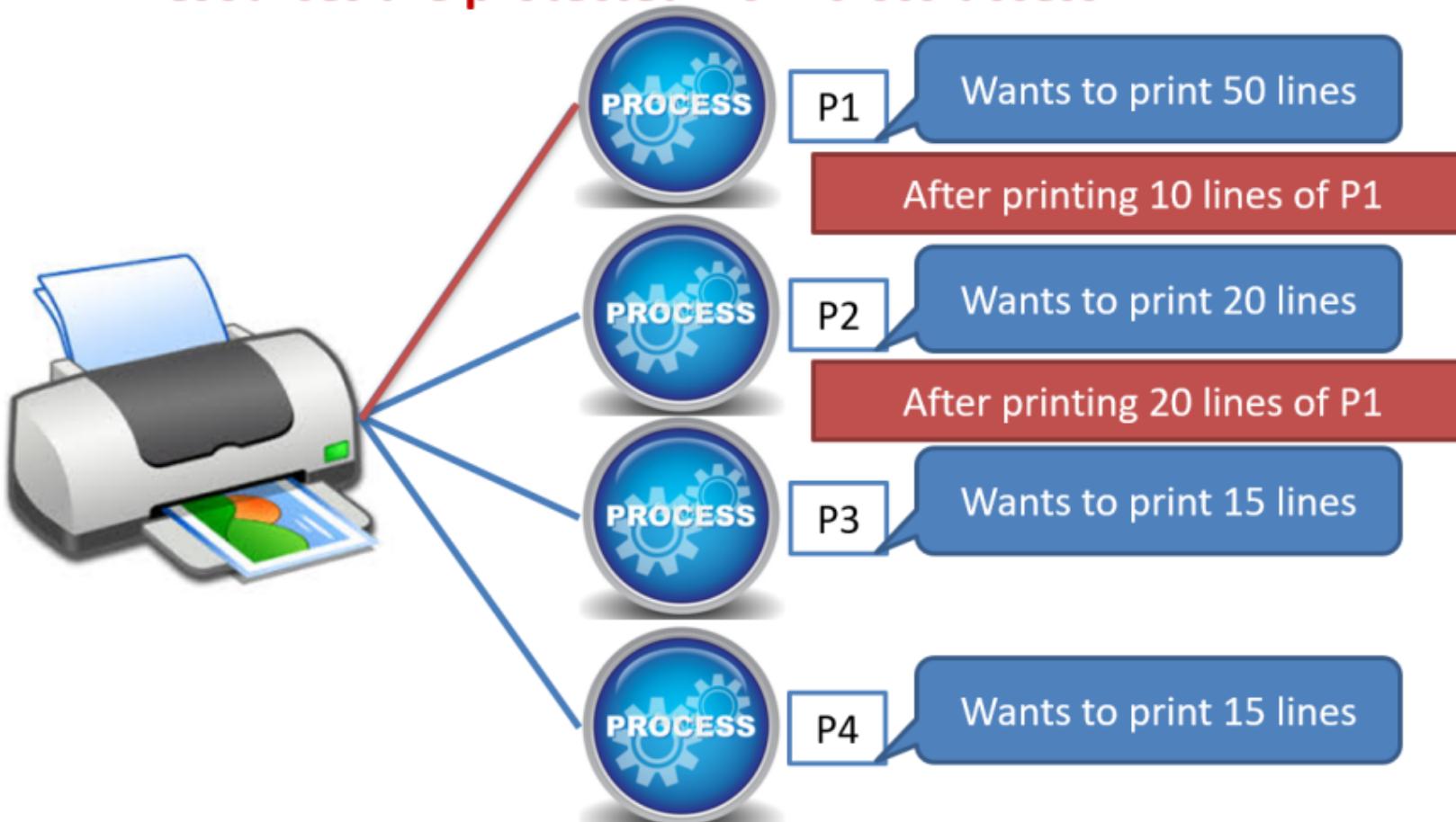
# OS as Control Program

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are protected from cross-access**



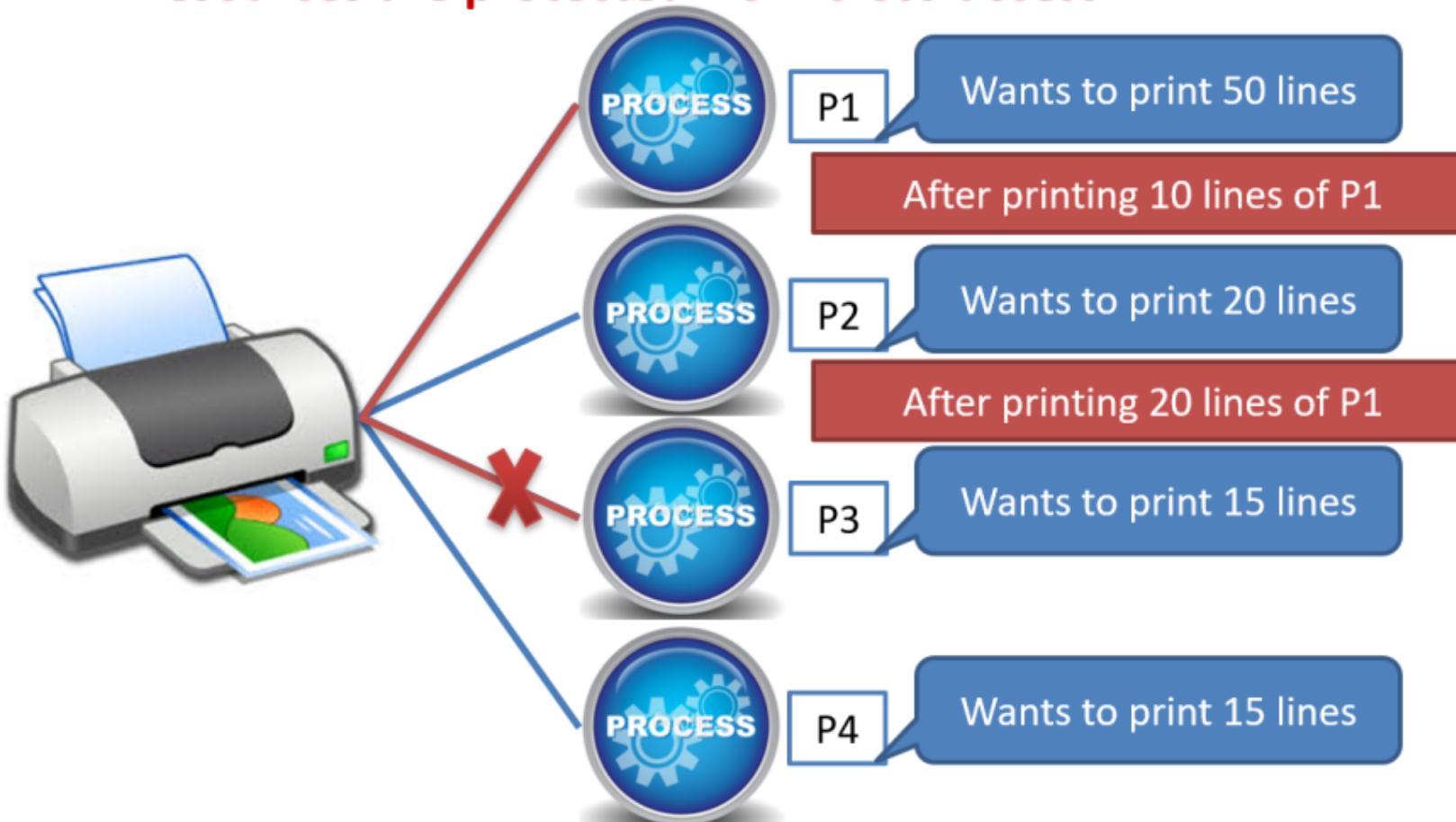
# OS as Control Program

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are protected from cross-access**



# OS as Control Program

- It is the job of OS to allocate these resources to the various applications so that:
  - The **resources are protected from cross-access**



# OS as Control Program

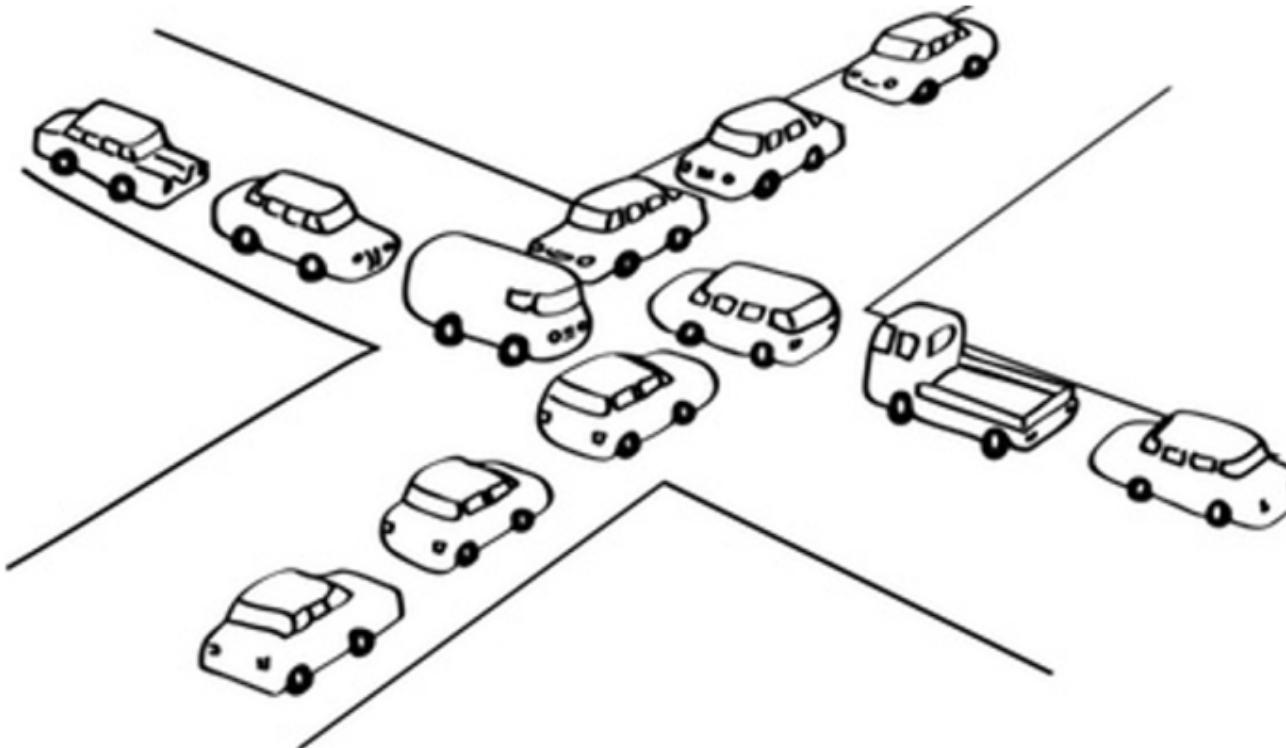
---

- It is the job of OS to allocate these resources to the various applications so that:
  - Access to the **resources is synchronized** so that operations are correct and consistent

# OS as Resource Manager

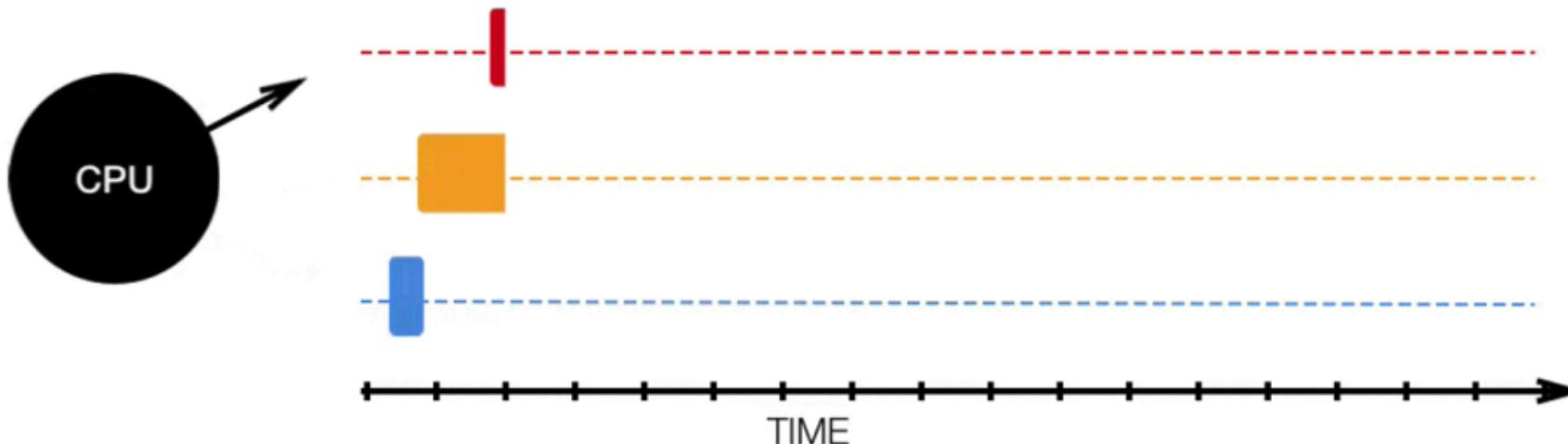
---

- It is the job of OS to allocate these resources to the various applications so that:
  - Deadlock are detected, resolved and avoided.



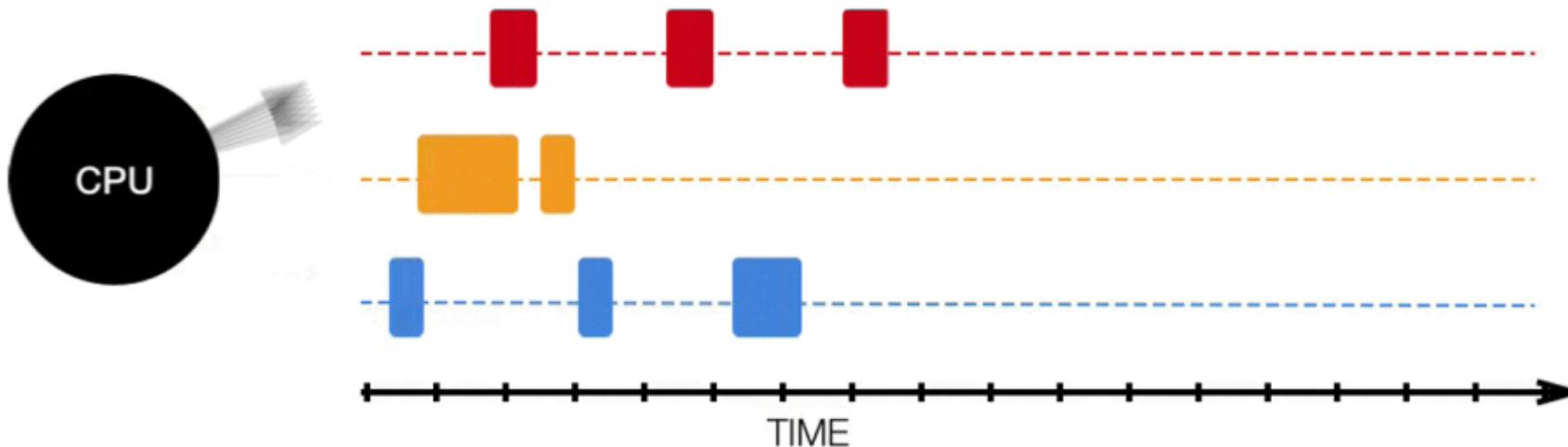
# OS as Resource Manager (Cont...)

- Resource manager – sharing resources in two different ways:
  1. In time sharing/multiplexing (**i.e CPU**)



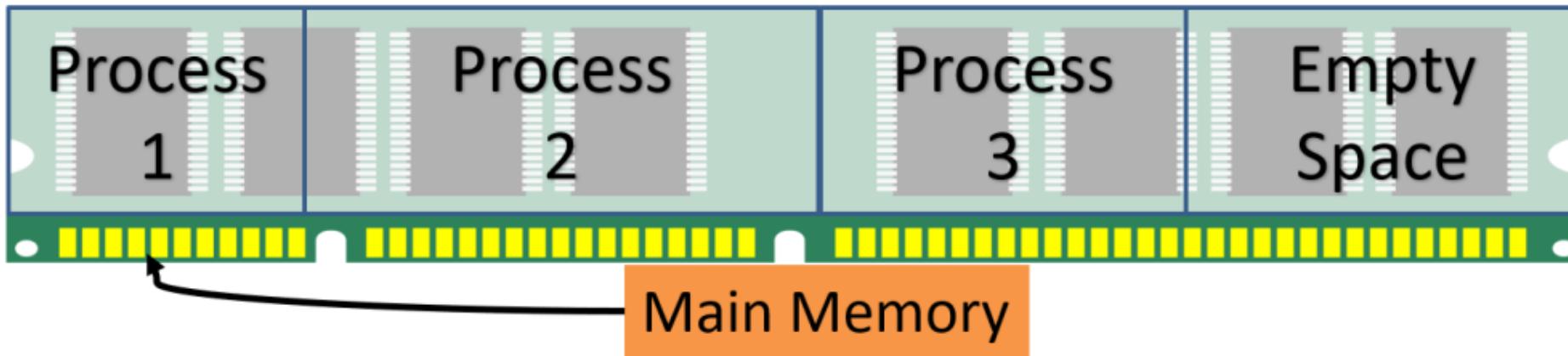
# OS as Resource Manager (Cont...)

- Resource manager – sharing resources in two different ways:
  1. In time sharing/multiplexing (**i.e CPU**)



# OS as Resource Manager (Cont...)

- Resource manager – sharing resources in two different ways
  2. In space sharing/multiplexing. **(i.e Memory)**



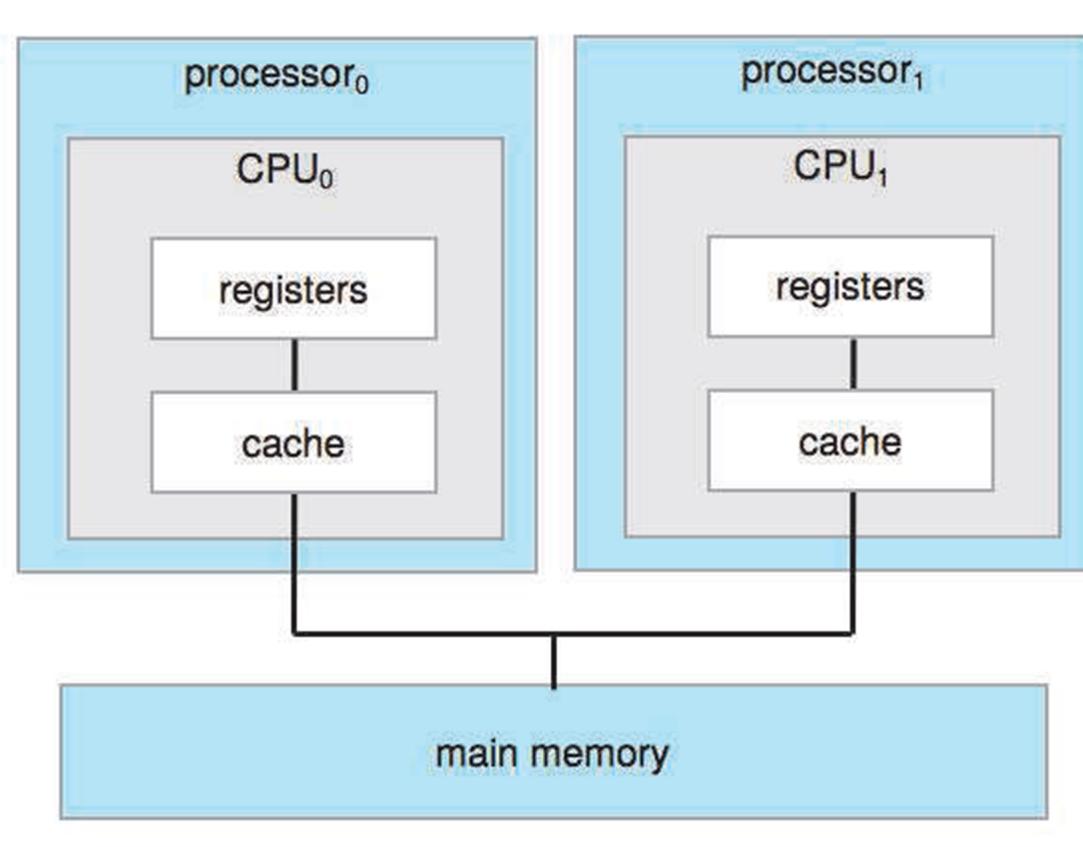
# DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- CPU—The hardware that executes instructions.
  - Many years ago, most computer systems used a single processor containing one CPU with a single processing core.
- Core—The component that executes instructions and registers for storing data locally.
- Multiprocessor—including multiple processors.
  - More work done in less time.

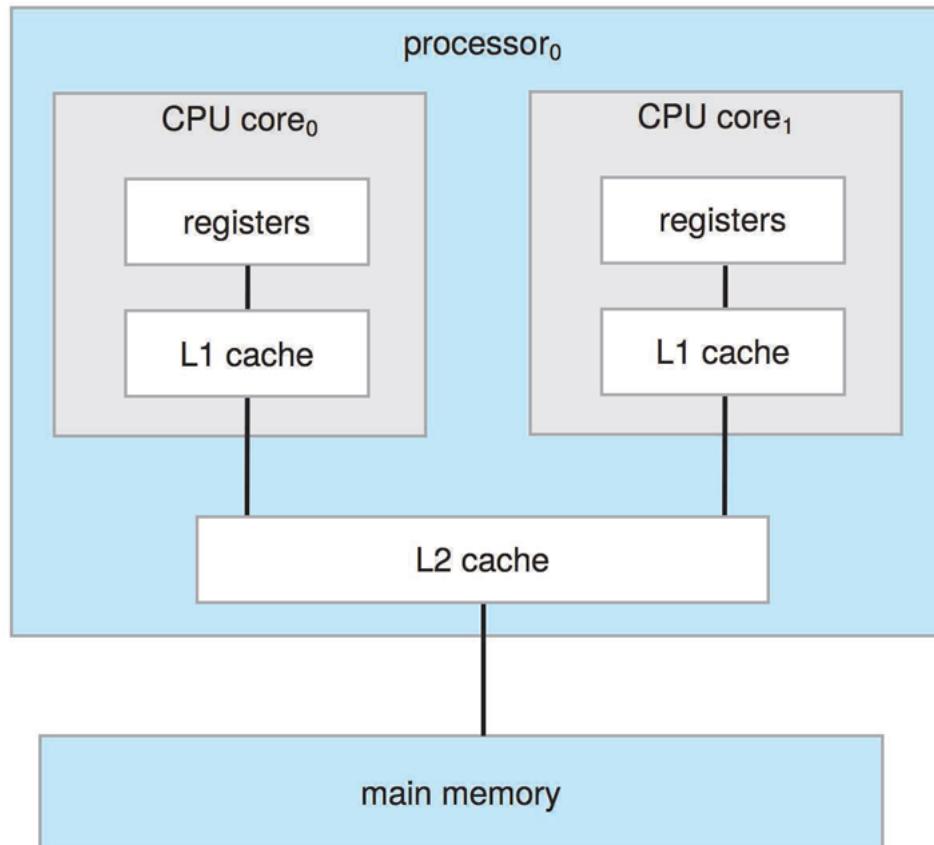
# DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- Multicore—Including multiple computing cores on the same CPU.
  - Each core consists of independent processor with components, such as registers, ALU, pipeline hardware and control unit, plus cache memory.
  - Uses less power than multiprocessor environment.
  - Communication between cores is faster than between multiple processors
- Although virtually all systems are now multicore, we use the general term CPU when referring to a single computational unit of a computer system and core as well as multicore when specifically referring to one or more cores on a CPU.

# Symmetric multiprocessing architecture



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



# OS Services / Functions

---

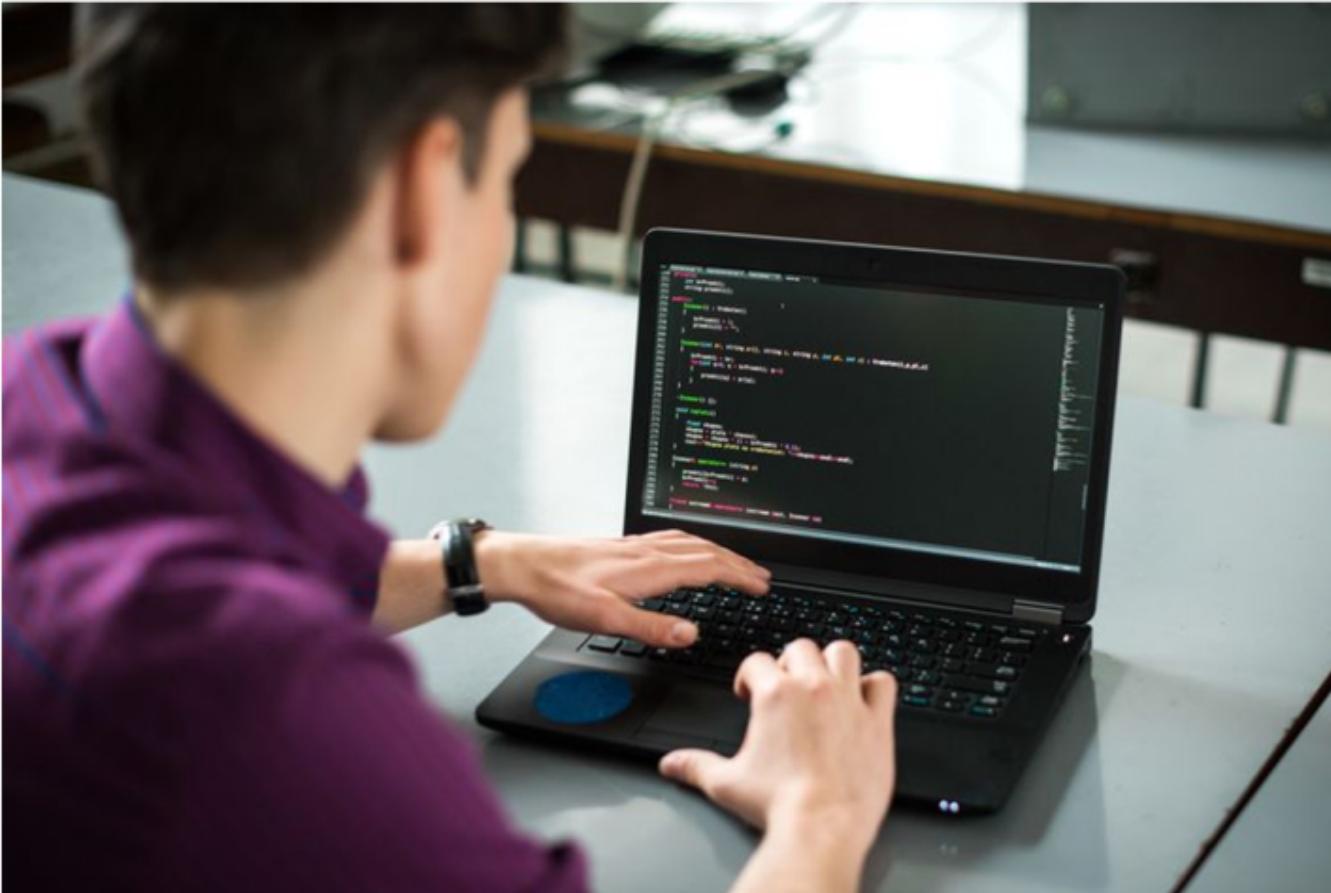
- An operating system provides an environment for the execution of programs.
- It makes certain services available to programs and to the users of those programs.
- One service provided by OS is **User interface**.
- This interface can take several forms.
  - a graphical user interface (GUI)
  - touch-screen interface
  - command-line interface (CLI),
- Some systems provide two or all three of these variations.
- The other functions of OS are:

# OS Services / Functions

---

## 1. Program development

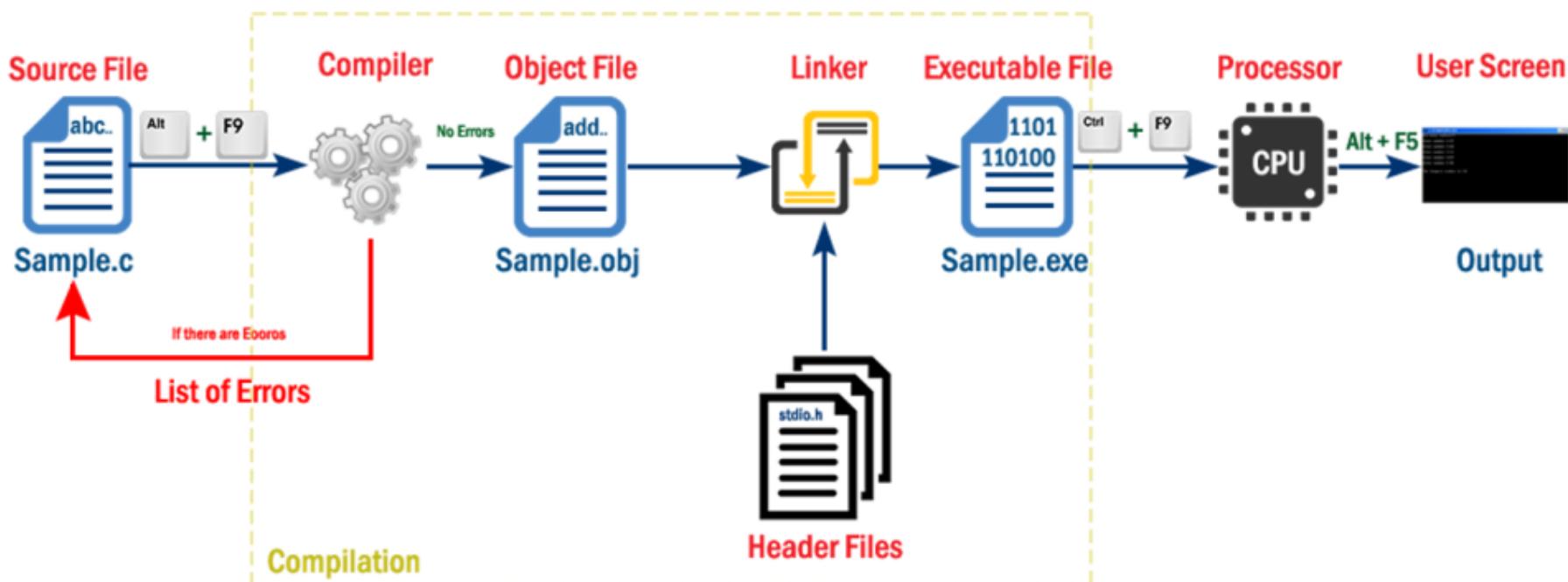
- It **provides editors** and debuggers to assist (help) the programmer in creating programs.



# OS Services / Functions

## 2. Program execution

- Following tasks need to be performed to execute a program:
  - ✓ Instructions and data must be **loaded into main memory**.
  - ✓ I/O **devices and files must be initialized**.
- The OS **handles all these duties** for the user.



# OS Services / Functions

---

## 3. Access to I/O devices (Resource allocation)

- A running program may require I/O, which may involve file or an I/O device.
- For efficiency and protection, users cannot control I/O devices directly.
- Therefore, the OS **controls these I/O devices** and **provides to program as per requirement**.

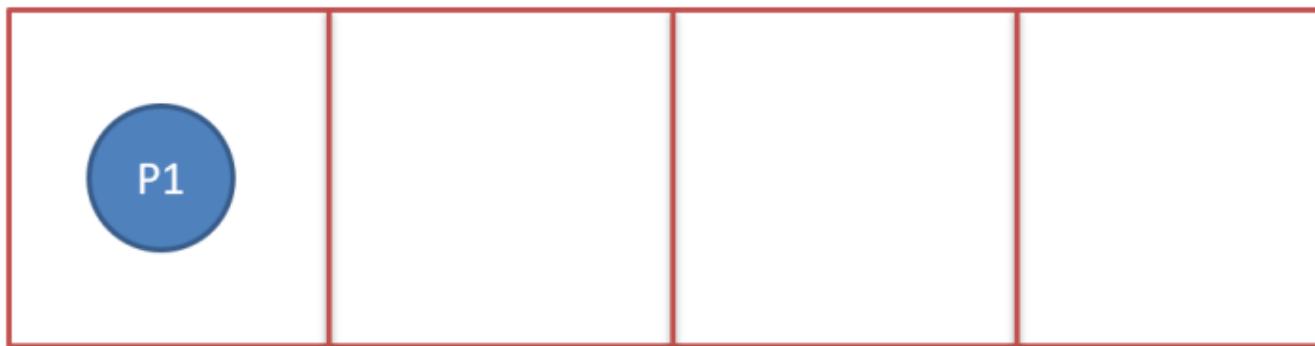


# OS Services / Functions

---

## 4. Memory management

- OS **manages memory hierarchy**.
- OS **keeps the track** of which part of memory area in use in use and free memory.

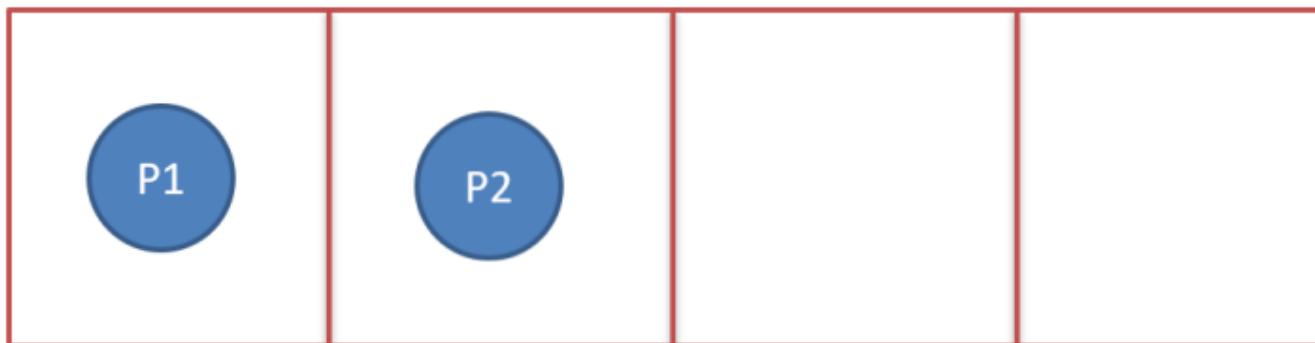


# OS Services / Functions

---

## 4. Memory management

- OS **manages memory hierarchy**.
- OS **keeps the track** of which part of memory area in use in use and free memory.
- It **allocates memory** to program when they need it.



# OS Services / Functions

---

## 4. Memory management

- OS **manages memory hierarchy**.
- OS **keeps the track** of which part of memory area in use in use and free memory.
- It **allocates memory** to program when they need it.
- It **de-allocate the memory** when the program finish execution.

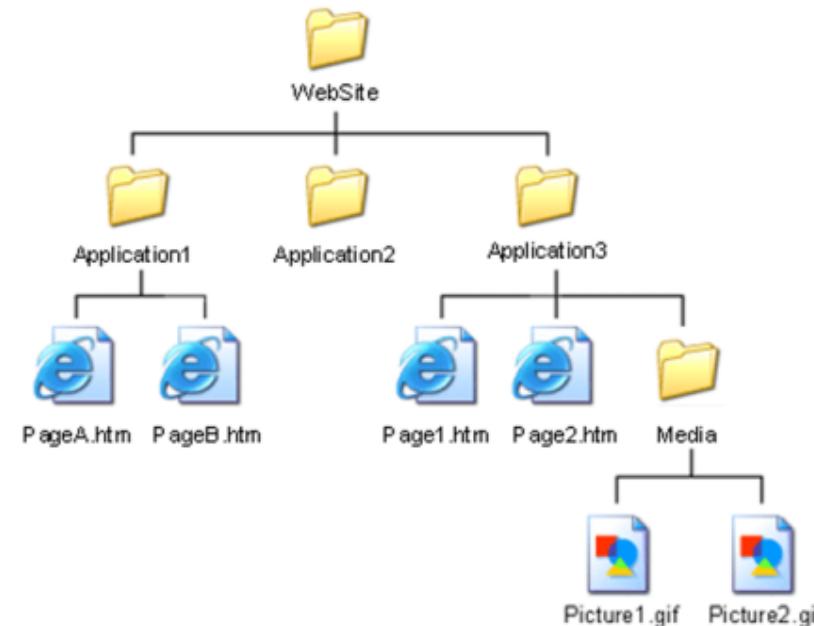


# OS Services / Functions

---

## 5. Controlled access to file

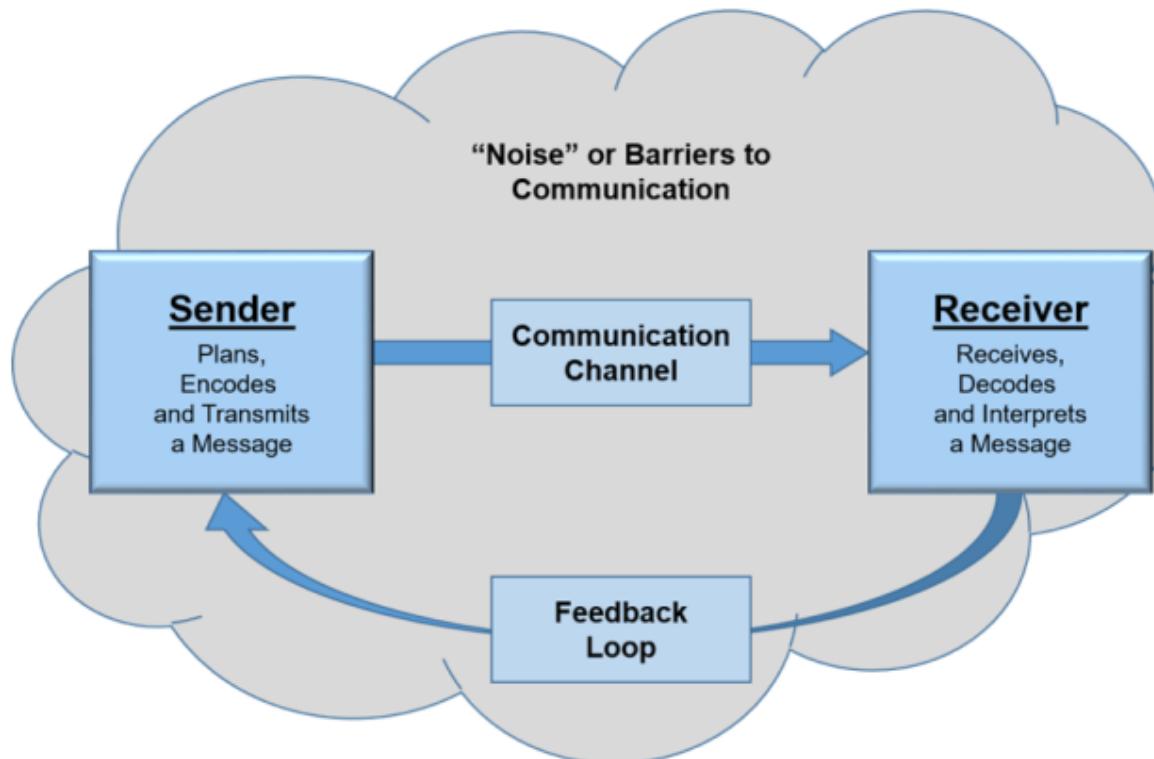
- In case of file access, OS **provides a directory hierarchy** for easy access and management of file.
- OS **provides various file handling commands** using which user can easily read, write and modify file.



# OS Services / Functions

## 6. Communication

- In multitasking environment, the processes need to communicate with each other and to exchange their information.
- Operating system **performs the communication among various types of processes** in the form of shared memory/message passing.



# OS Services / Functions

---

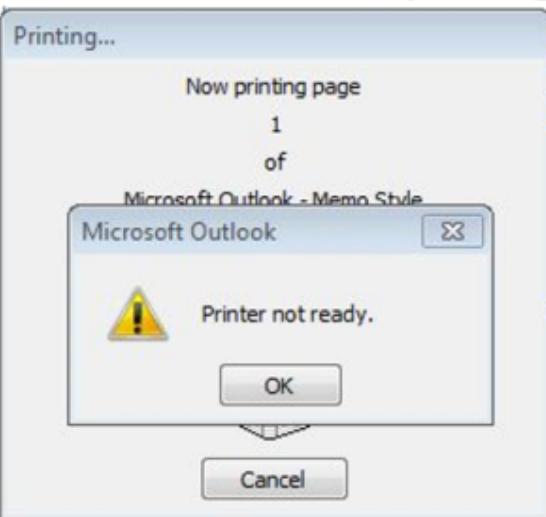
## 7. Error detection and response

- Errors that may occur in the system can be in
  - the CPU and memory hardware (such as a memory error or a power failure),
  - I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer),
  - the user program (such as an arithmetic overflow or an attempt to access an illegal memory location).

# OS Services / Functions

## 7. Error detection and response

- An error may occur in CPU, in I/O devices or in the memory hardware.
- Following are the major activities of an operating system with respect to error handling –
  - ✓ The OS **constantly checks for possible errors.**
  - ✓ The OS **takes an appropriate action** to ensure **correct and consistent** computing.



# OS Services / Functions

---

## 8. Accounting /Logging

- **Keeping a track of which users are using how much and what kinds of computer resources** can be used for accounting or simply for accumulating usage statistics.
- Usage statistics is used to reconfigure the system to improve computing services.



# OS Services / Functions

---

## 9. Protection & Security

- When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself.
- Protection must be ensured.

# OS Services / Functions

---

## 9. Protection & Security

- Protection involves **ensuring that all accesses to system resources is controlled.**
- To make a system secure, the user needs to authenticate himself or herself to the system.
- This is done usually by means of a password, to gain access to system resources.
- It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins



# OS Services / Functions (Revision)

---

1. Program development
2. Program execution
3. Access to I/O devices
4. Memory management
5. Controlled access to file
6. Communication
7. Error detection and response
8. Accounting
9. Protection & Security

# OPERATING SYSTEMS

INTRODUCTION

# Objectives of OS

- An operating system can be thought of as having three objectives or as performing three functions.
  - **Convenience** - An operating system makes a computer more convenient to use.
  - **Efficiency** - An operating system allows the computer system resources to be used in an efficient manner.
  - **Ability to evolve** - An operating system should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without interfering with current services provided.

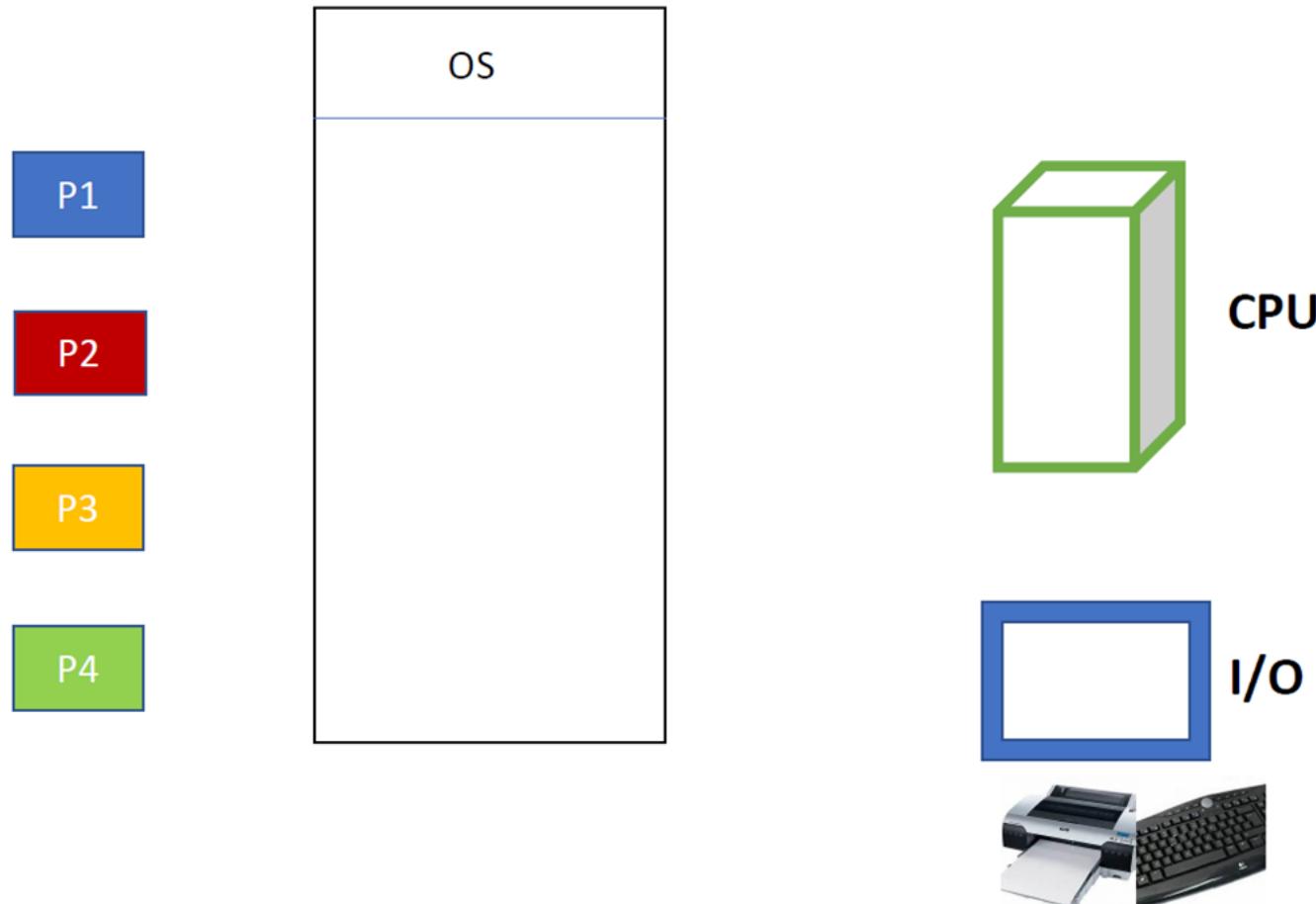
# Uni-programming

- Uni-programming is a term used for those computers which can have only a single task/job in the main memory at a time.
- Any job/task which has to be executed/run by the CPU must be present in the main memory and if a computer can have only a single task in the main memory at a time then it is known as Uni-Programming.

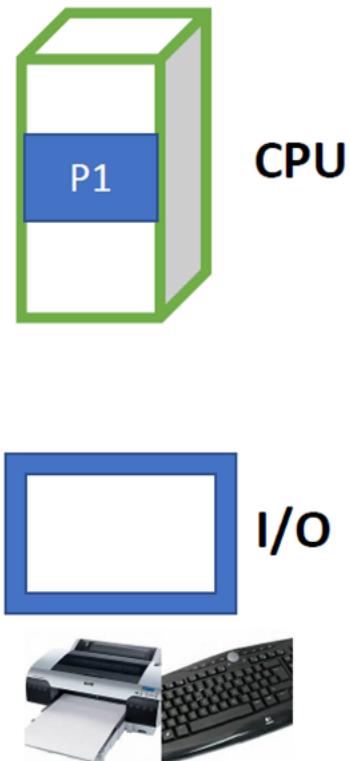
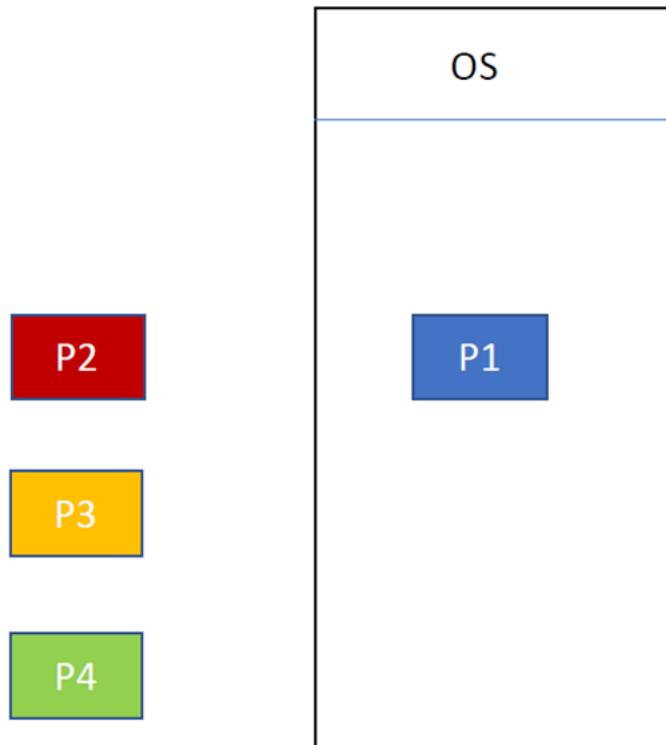
Main Memory



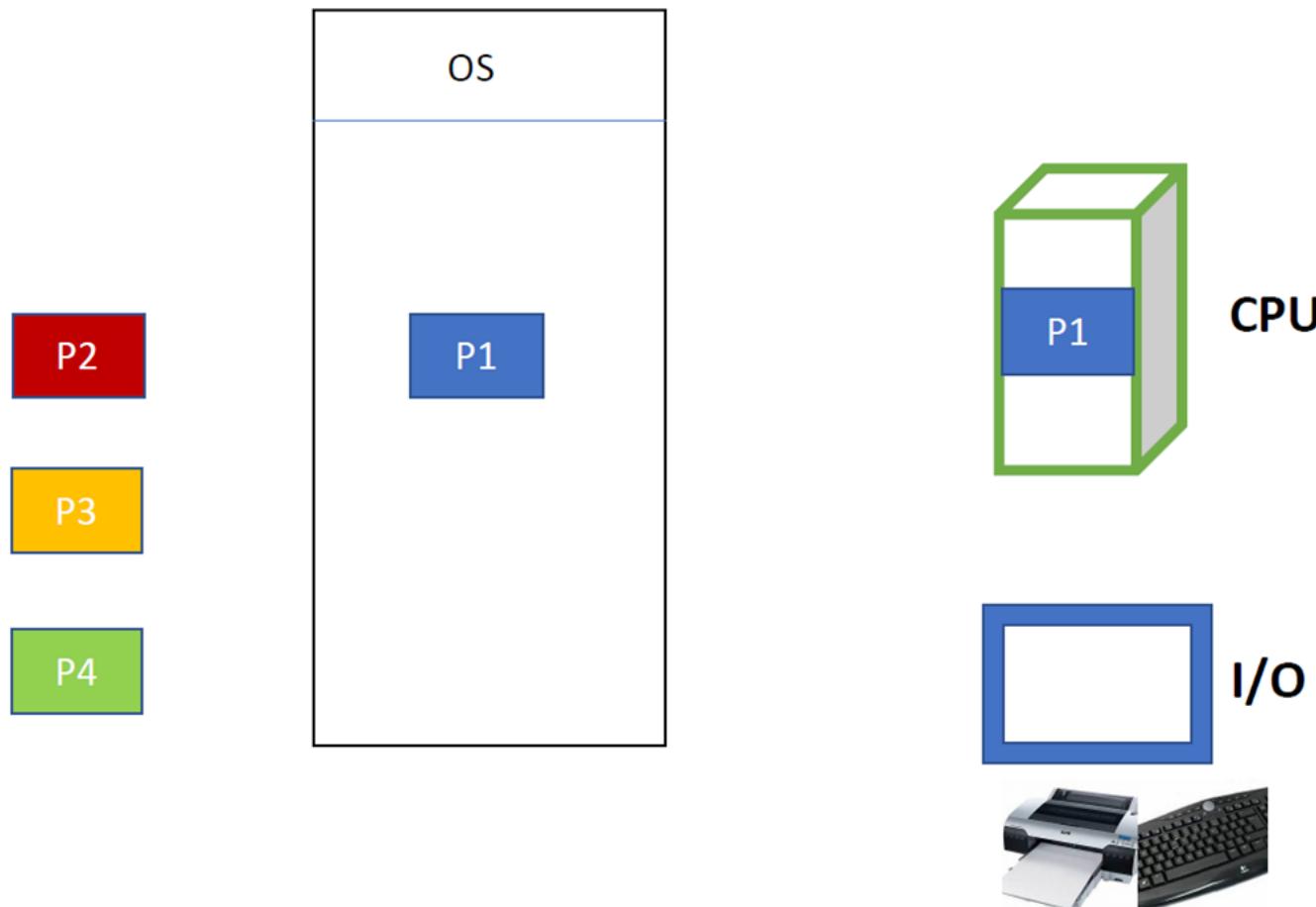
# Uni-Programming



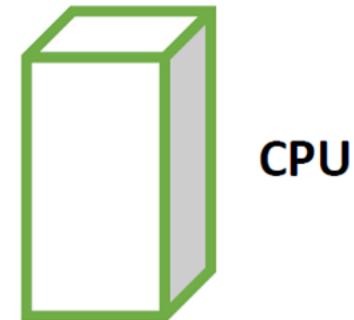
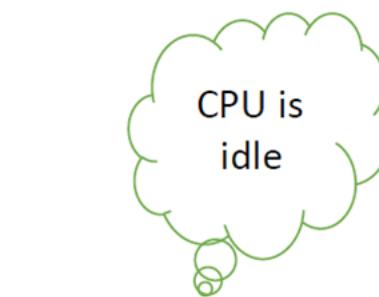
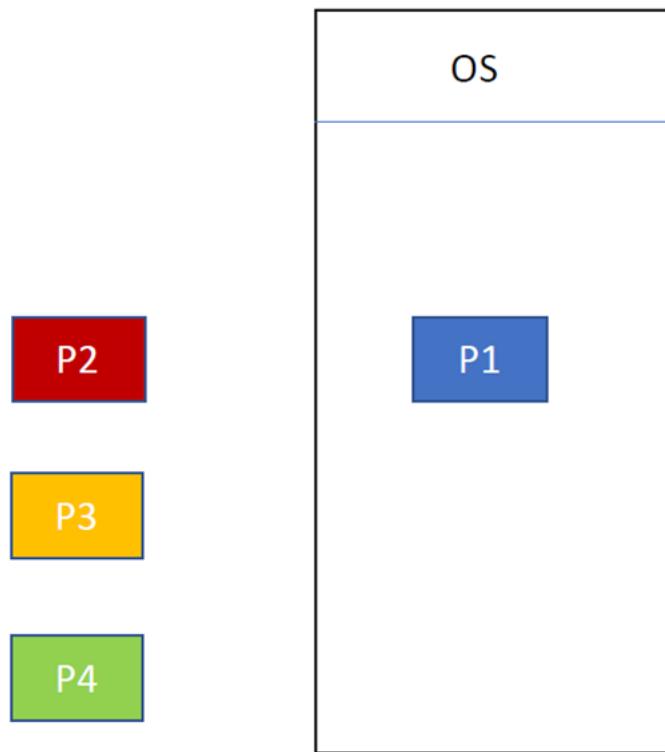
# Uni-Programming



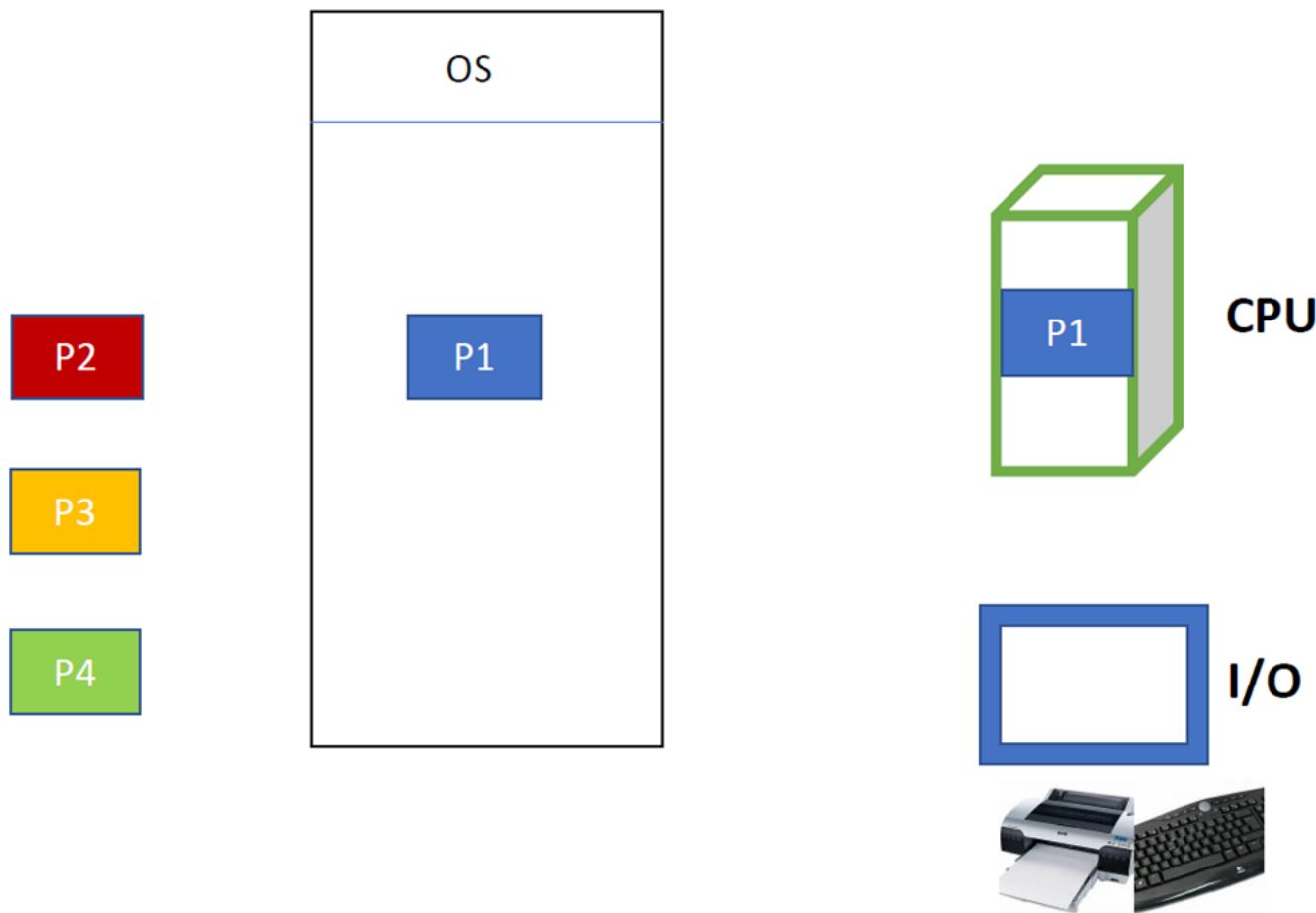
# Uni-Programming



# Uni-Programming

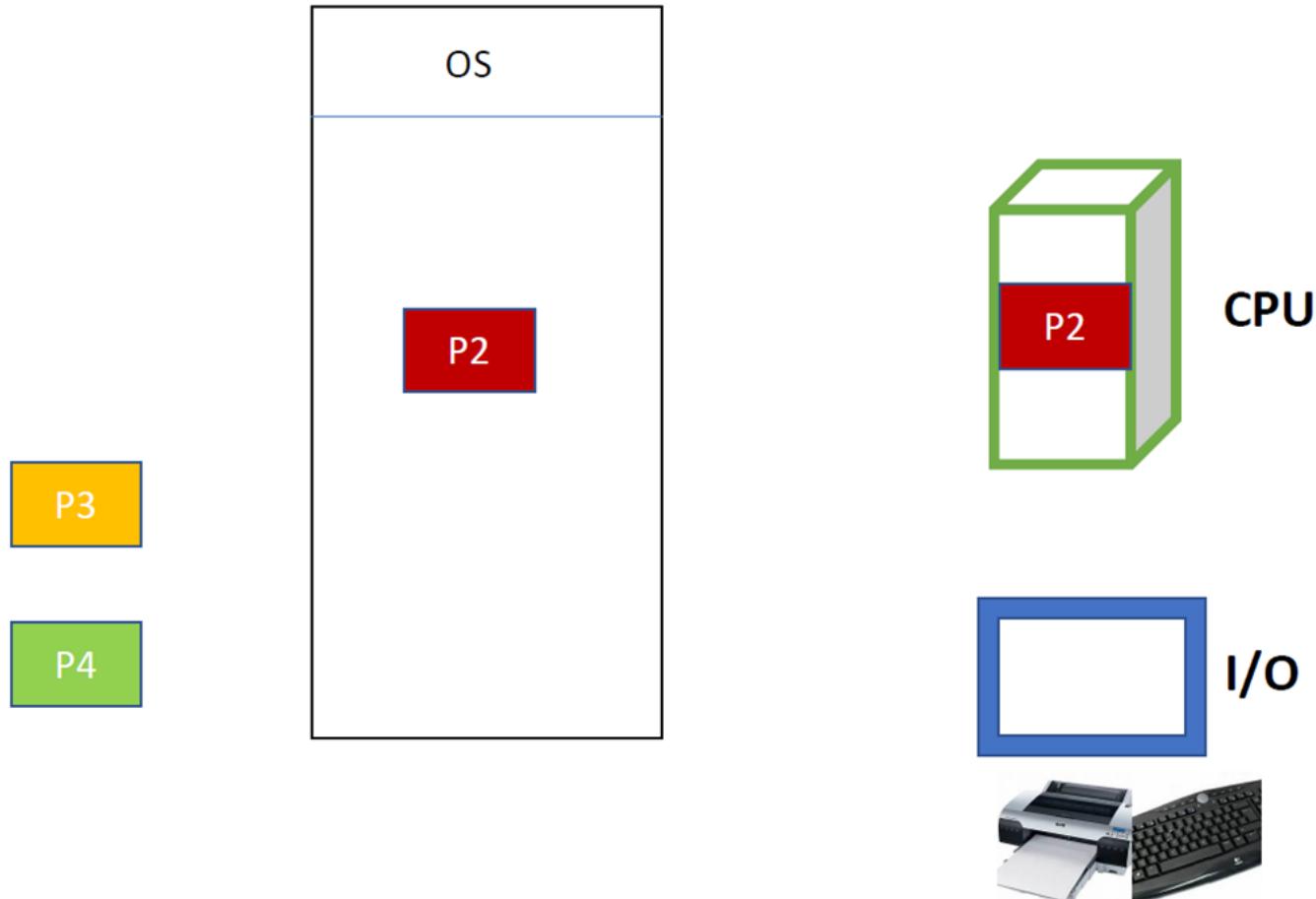


# Uni-Programming



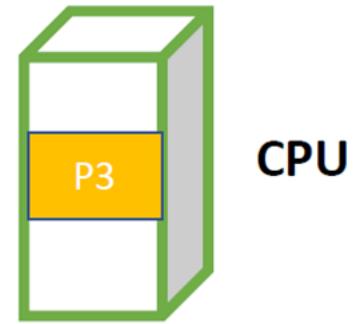
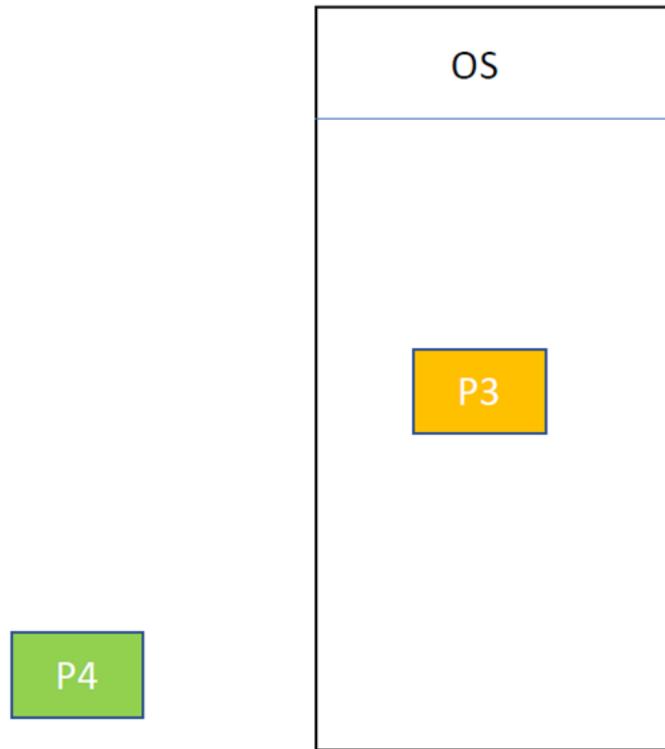
11  
Clock

# Uni-Programming



13  
Clock

# Uni-Programming



CPU



I/O

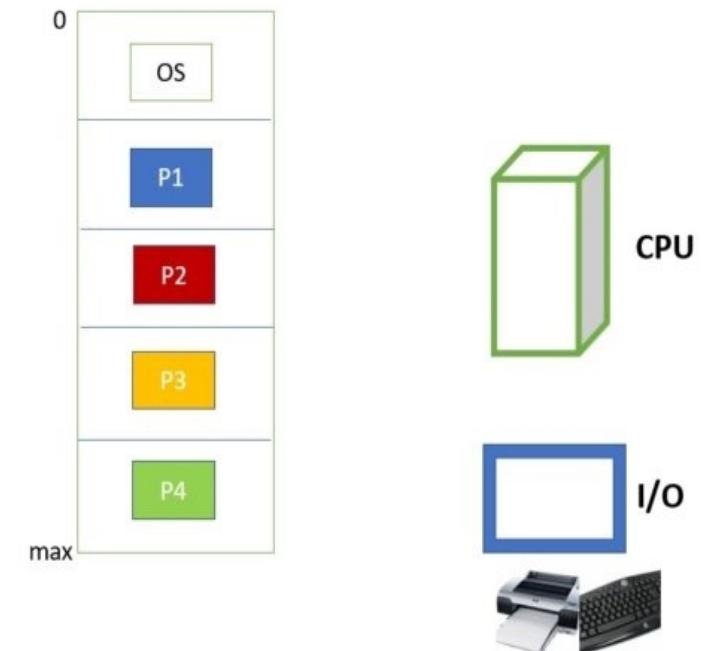


17  
Clock

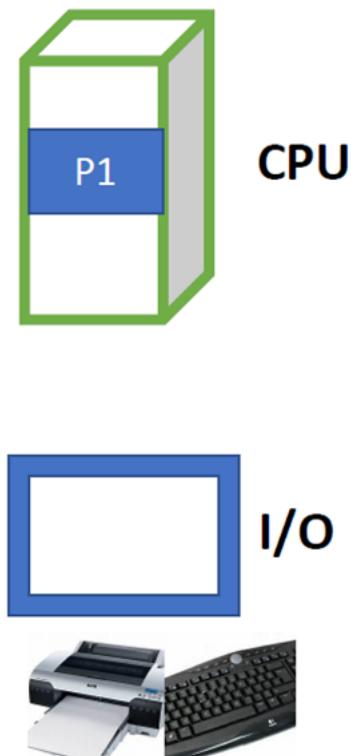
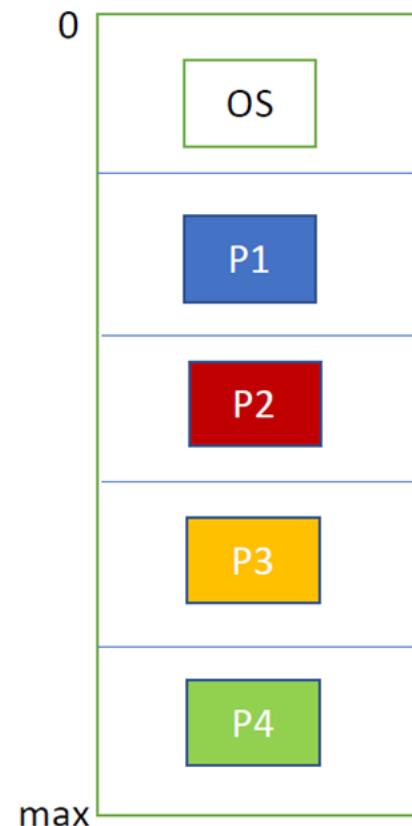
# Multi-Programming

- In multi-programming more than one process can reside in the main memory at a time.
- Thus, when process P1 goes for I/O operation the CPU is not kept waiting and is allocated to some another process (lets say P2).
- This keeps the CPU busy at all times.

Multi-Programming

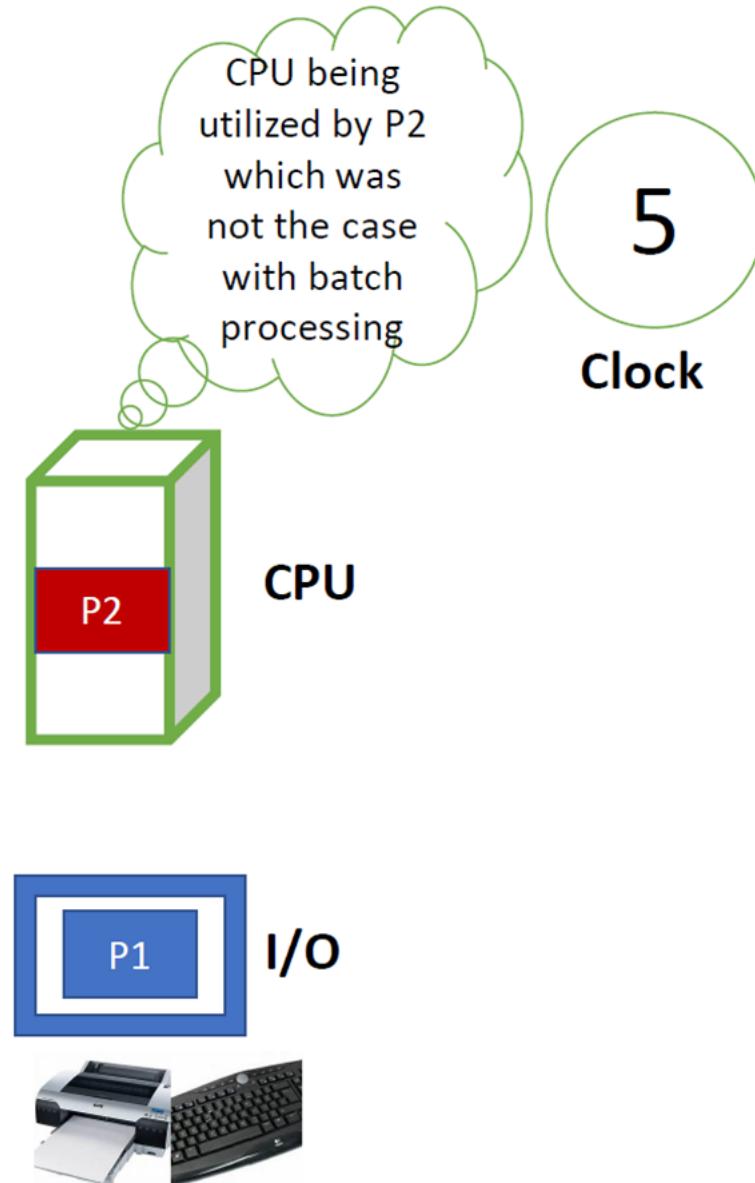
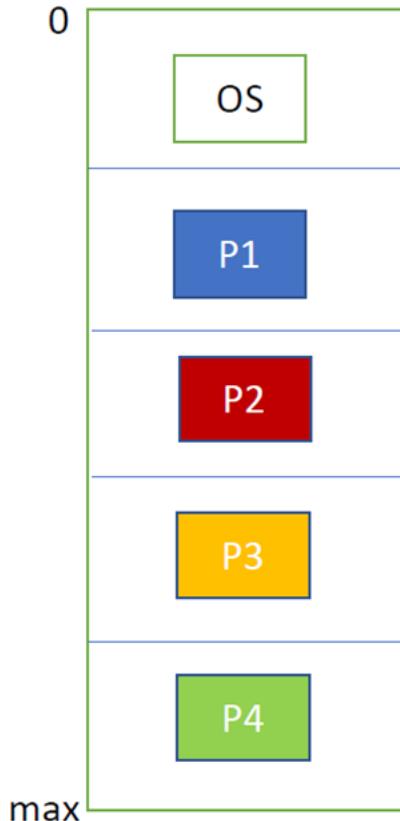


# Multi-Programming

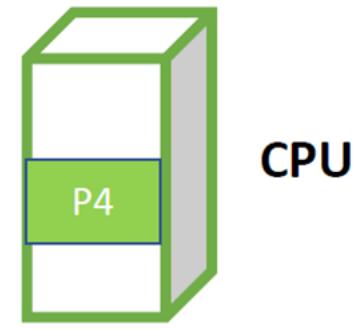
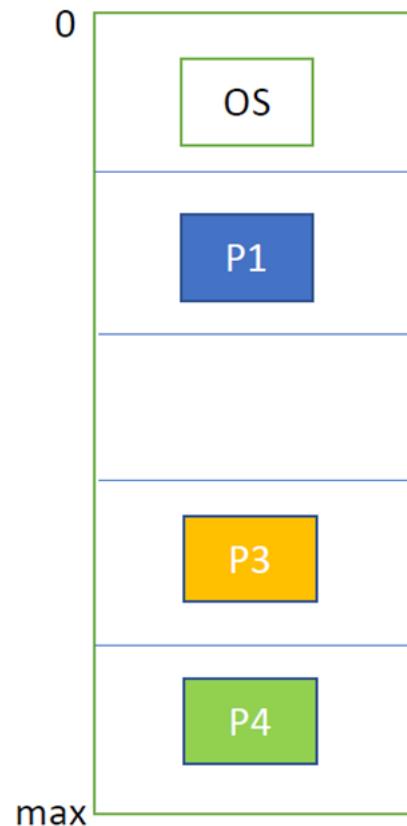


0  
Clock

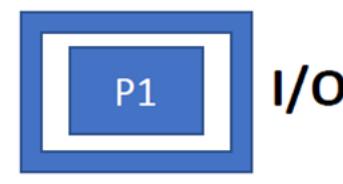
# Multi-Programming



# Multi-Programming



CPU

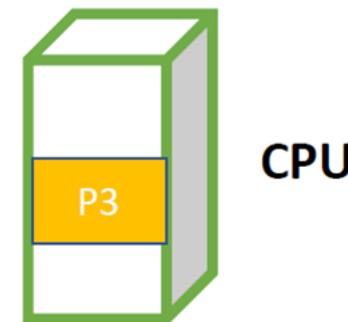
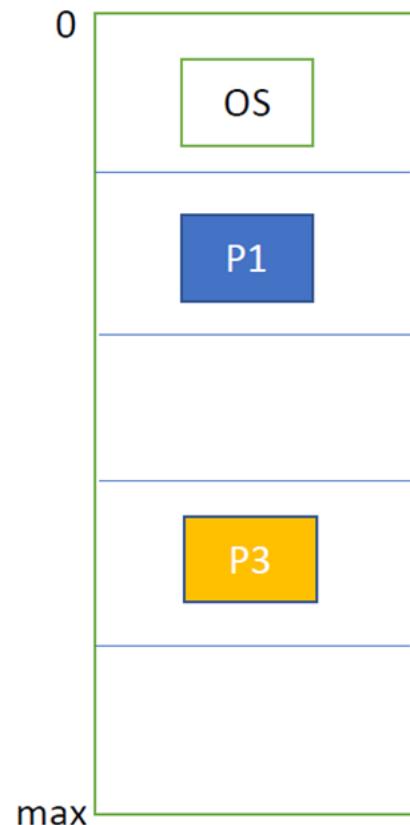


I/O



Clock

# Multi-Programming



CPU

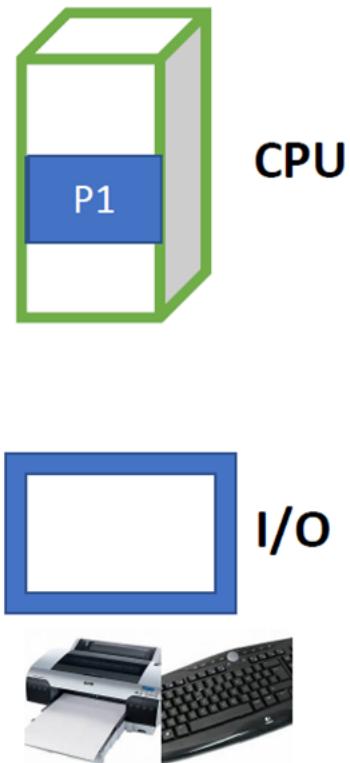
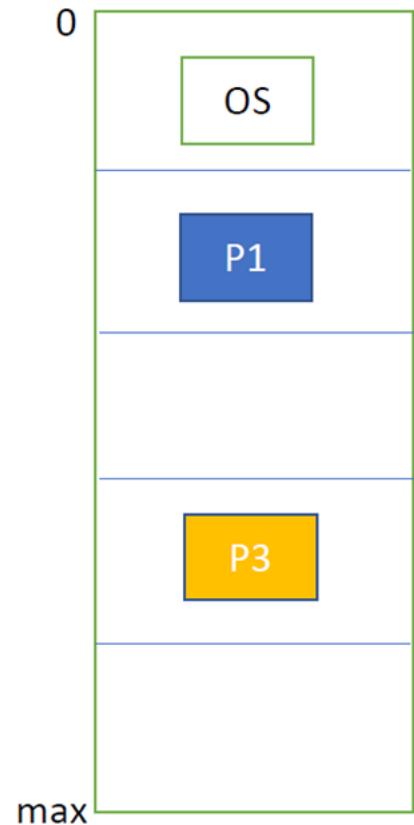


I/O



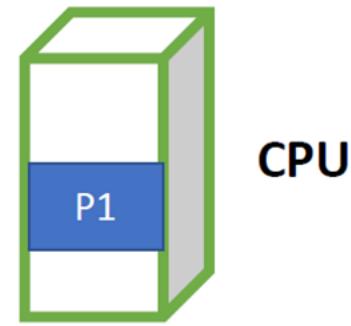
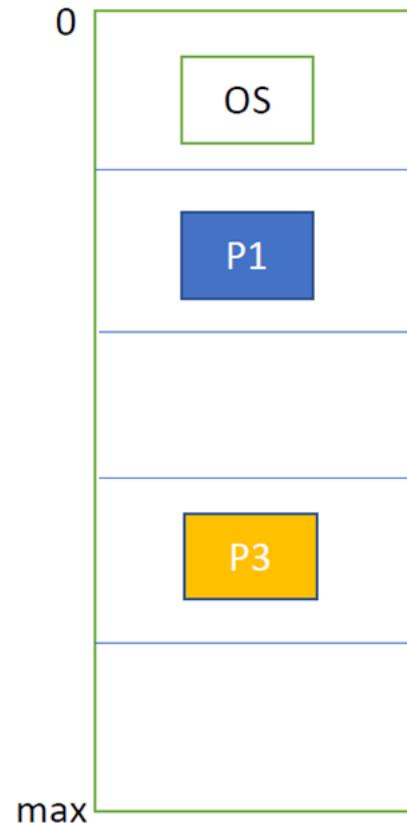
10  
Clock

# Multi-Programming



11  
Clock

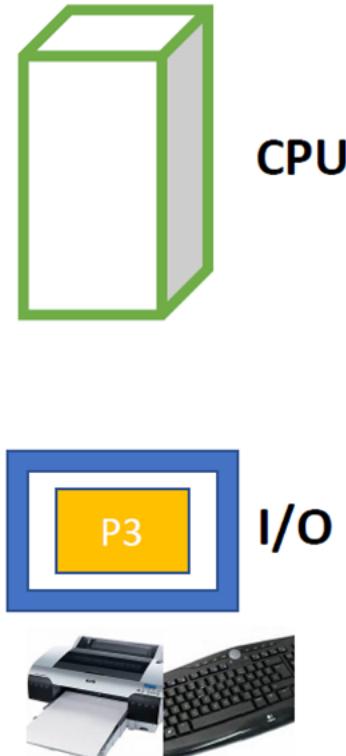
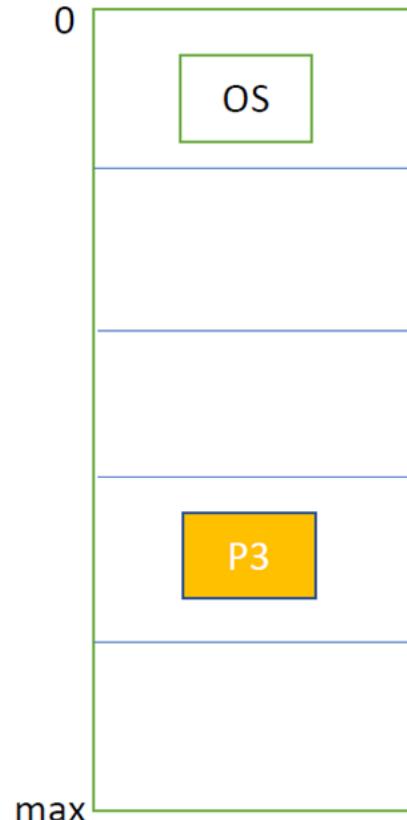
# Multi-Programming



12  
Clock

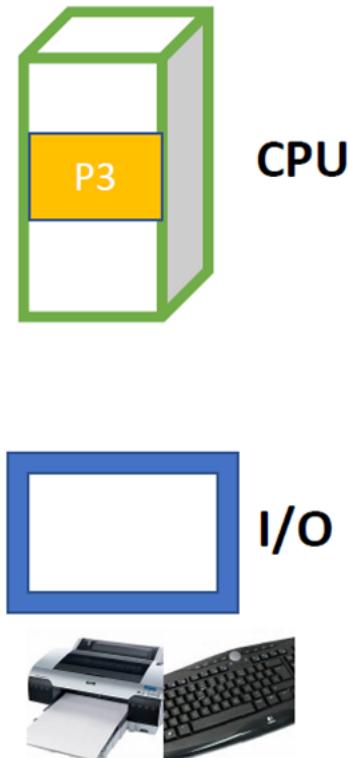
---

# Multi-Programming



14  
Clock

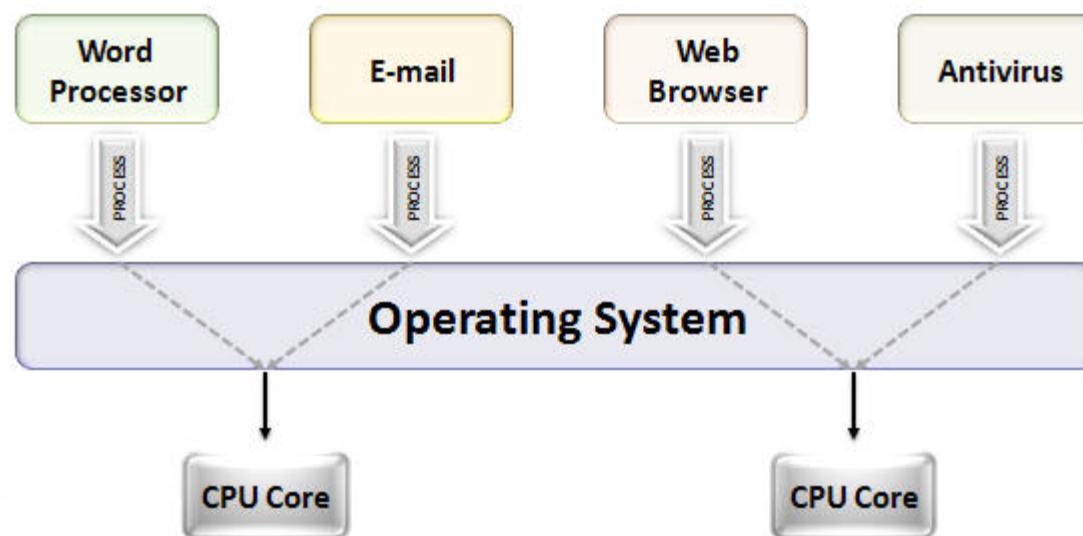
# Multi-Programming



17  
Clock

# Multi-tasking

- Multitasking is the process of performing multiple tasks at the same time.
- For example, multiple applications execute in a computer simultaneously.
- Browser, word application, calculator, etc. can execute at the same time.



# Multi-tasking

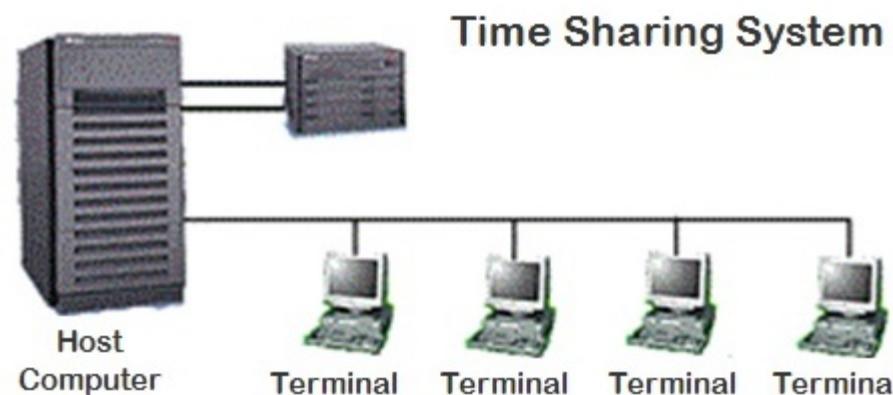
- Even though most modern computers support multitasking, the system can execute a specific number of tasks at a time.
- It is because when there are multiple tasks, each task requires more resources.
- Thus, this can cause a decrease in system performance.
- Therefore, an adequate number of tasks executes at a time.
- Overall, multitasking helps to improve the productivity of the system.
- Multi-tasking system uses multi-programming.

# Time-sharing

- Time sharing is a method that allows multiple users to share resources at the same time.
- Multiple users in various locations can use a specific computer system at a time.
- Several terminals are attached to a single dedicated server with its own process.
- Therefore, the processor executes multiple user programs simultaneously.
- In other words, the processor time is shared between multiple users at a time.
- The processor allows each user program to execute for a small time quantum.
- Moreover, time sharing systems use multiprogramming and multitasking.

# Time-sharing vs Multi-tasking

- Time sharing is the sharing of a computing resource among many users by means of multiprogramming and multitasking at the same time whereas **multitasking** is **the concurrent execution of multiple tasks or processes over a certain period of time.**

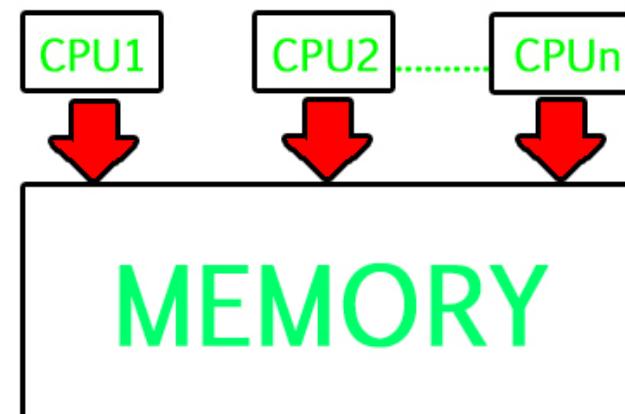


# Multi-processing

- In a uni-processor system, only one process executes at a time.
- Multiprocessing is the use of two or more CPUs (processors) within a single Computer system.
- The term also refers to the ability of a system to support more than one processor within a single computer system.
- Now since there are multiple processors available, multiple processes can be executed at a time.
- These multi processors share the computer bus, sometimes the clock, memory and peripheral devices also.

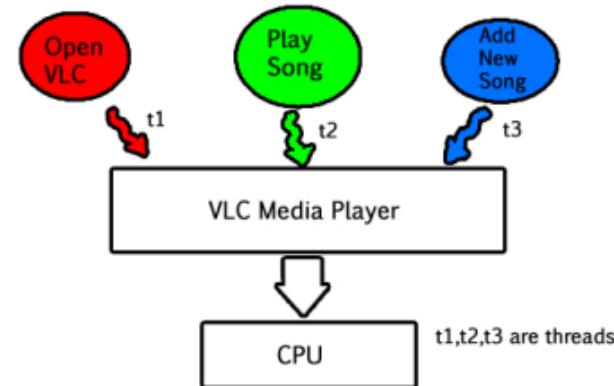
# Multi-processing

- Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes).
- If the underlying hardware provides more than one processor then that is multiprocessing.
- It is the ability of the system to leverage multiple processors' computing power.



# Multi-threading

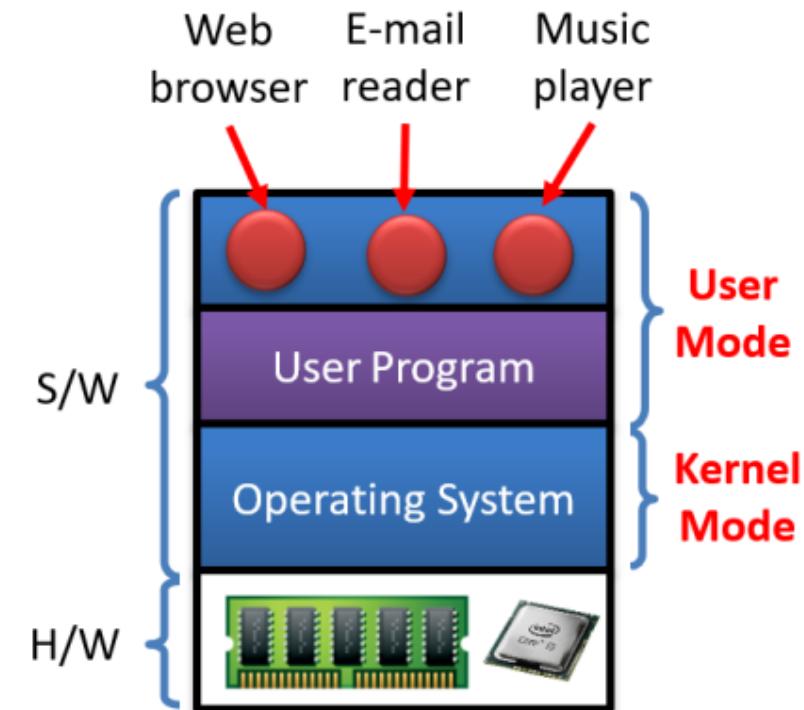
- A thread is a basic unit of CPU utilization.
- Multi threading is an execution model that allows a single process to have multiple code segments (i.e., threads) running concurrently within the “context” of that process.
- e.g. VLC media player, where one thread is used for opening the VLC media player, one thread for playing a particular song and another thread for adding new songs to the playlist.



# Modes of operation of computer

---

# Modes of operation of computer

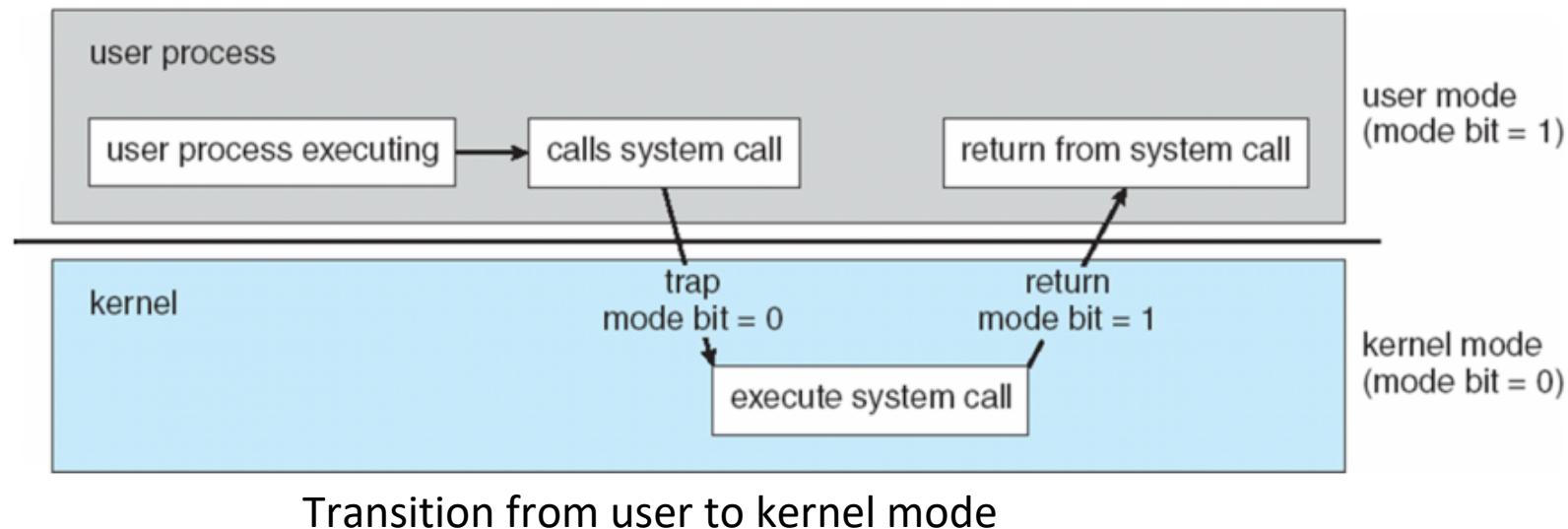


# Dual mode operation

- In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code.
- At the very least, we need two separate modes of operation:
  - **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**).
- A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: **kernel (0)** or **user (1)**.
- Increasingly CPUs support **multi-mode** operations
  - i.e. virtual machine manager (VMM) mode for guest VMs

# Dual mode operation

- When the computer system is executing on behalf of a user application, the system is in **user mode**.
- However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.



# Dual mode operation

- At system boot time, the hardware starts in **kernel mode**.
- The operating system is then loaded and starts user applications in **user mode**.
- Whenever a trap or interrupt occurs, the hardware switches from **user mode to kernel mode** (that is, changes the state of the mode bit to 0).
- Thus, whenever the operating system gains control of the computer, it is in **kernel mode**.
- The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

# Dual mode operation

- The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.
- We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.
- The hardware allows privileged instructions to be executed only **in kernel mode**.
- If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and **traps** it to the operating system.

A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

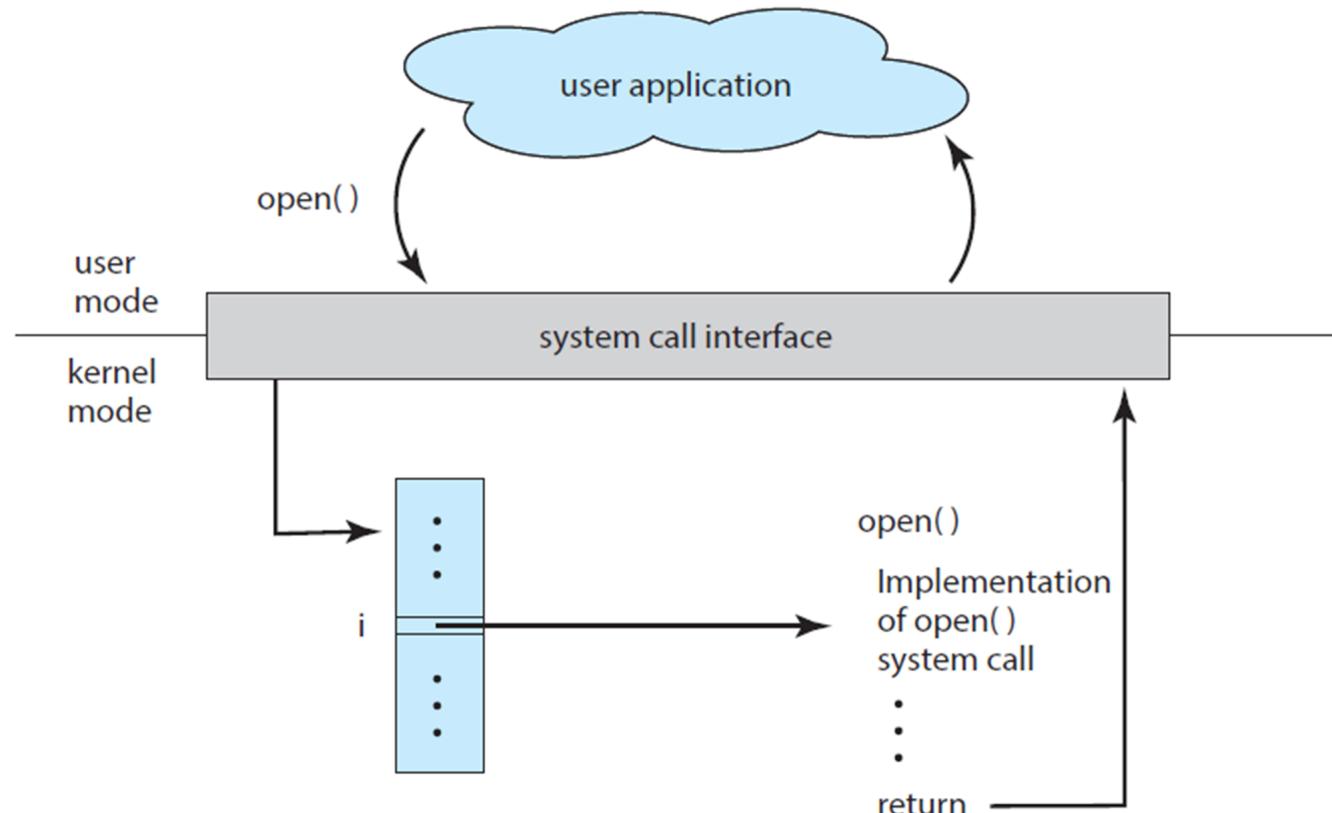
# System Calls

- **System calls** provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
- A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor, like a trap to a specific location in the interrupt vector.
- In all forms, it is the method used by a process to request action by the operating system.
- The system-call service routine is a part of the operating system.
- These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

# System Calls

- When a system call is executed, it is typically treated by the hardware as a software interrupt.
- Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode.
- The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting.
- Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers).
- The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

# System Calls



The handling of a user application invoking the `open()` system call.

# Types of System Calls

- System calls can be grouped roughly into six major categories:
  - process control
  - file management
  - device management
  - information maintenance
  - communications
  - protection

# Types of System Calls

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

# Some examples of System Calls

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

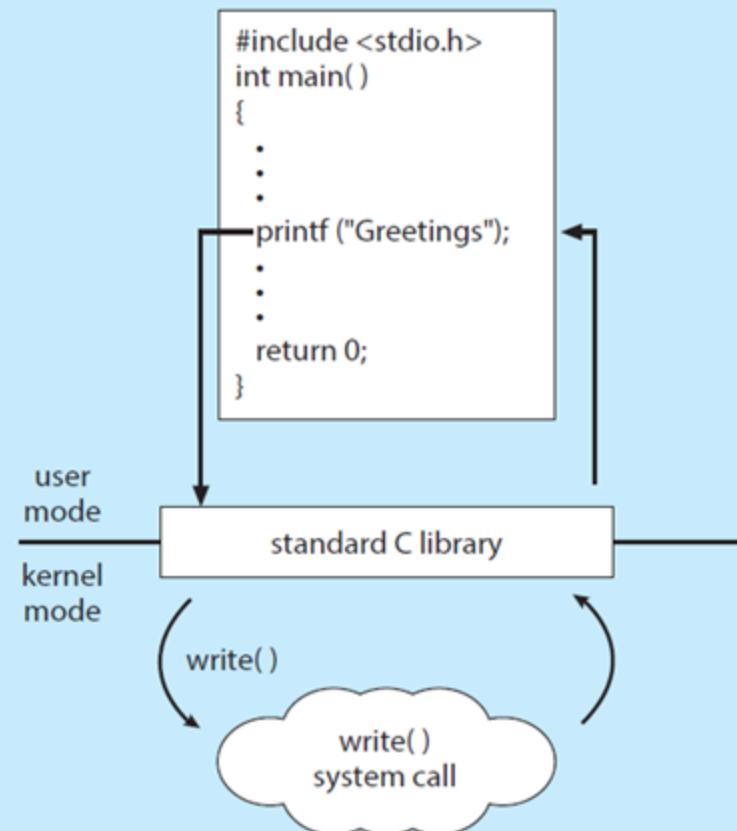
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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

# C Library

## THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



# API for System Calls

- Typically, application developers design programs according to an application programming interface (API).
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- A programmer accesses an API via a library of code provided by the operating system and is called **libc** (in case of Linux and Unix).
- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?
  - One benefit concerns program portability.
  - An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API

# System Boot

- The procedure of starting a computer by loading the kernel is known as booting the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution.
- When a CPU is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there.
- At that location is the initial bootstrap program.
- This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup.
- ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus

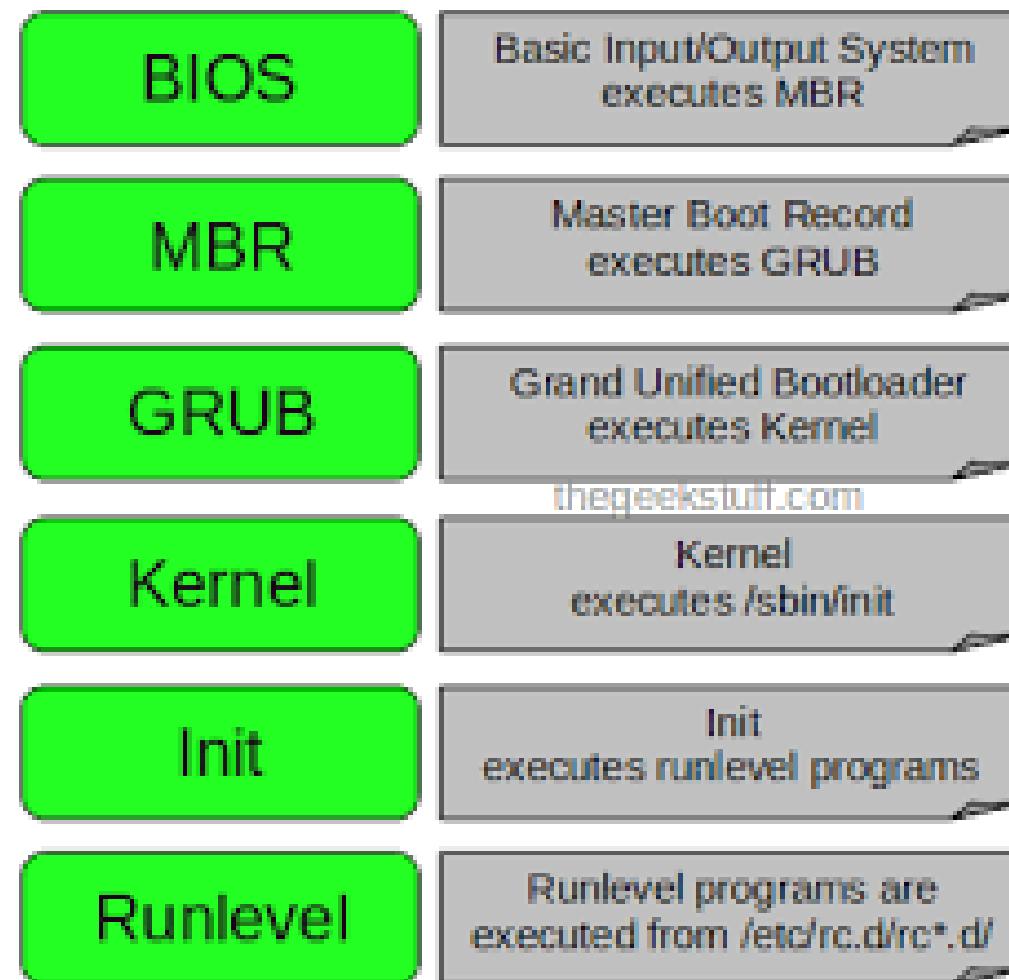
# System Boot

- The bootstrap program can perform a variety of tasks.
- Usually, one task is to run diagnostics to determine the state of the machine.
- If the diagnostics pass, the program can continue with the booting steps.
- It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory.
- Sooner or later, it starts the operating system.
- A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips.
- Some systems resolve this problem by using erasable programmable read-only memory (EPROM), which is read-only except when explicitly given a command to become writable.

# System Boot

- For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk.
- In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that boot block.
- The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.
- **GRUB (Grand Unified Boot Loader)** is an example of an open-source bootstrap program for Linux systems. It also allows user to chose the OS to be booted, in case more than one OS available in the system.

# Linux Boot Process



# Evolution of Operating Systems

# Need for Evolution of an Operating Systems

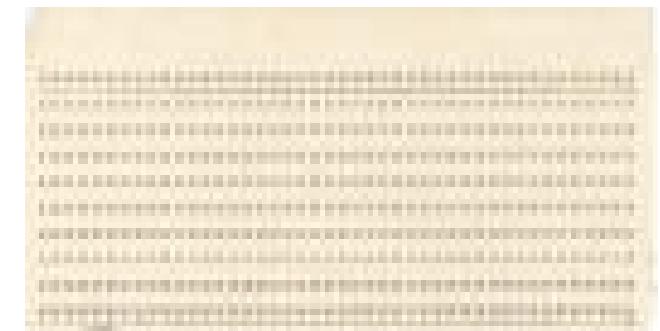
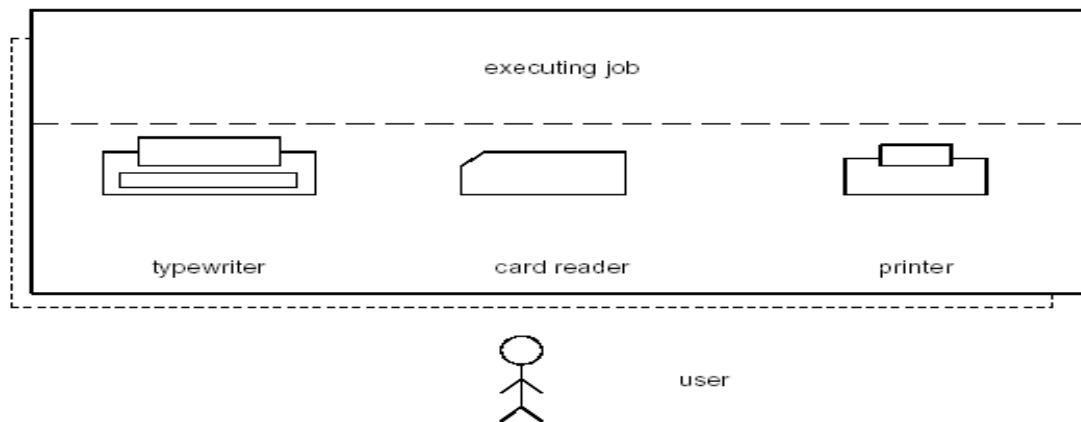
- Must adapt to hardware upgrades and new types of hardware
- Must offer new services
- Fixes – Any OS has faults
  - The need to change the OS on regular basis places certain requirements on it's design:
    - modular construction with clean interfaces.
    - much more to be done than simple partitioning a program into modules - object oriented methodology.

# Evolution of an Operating Systems

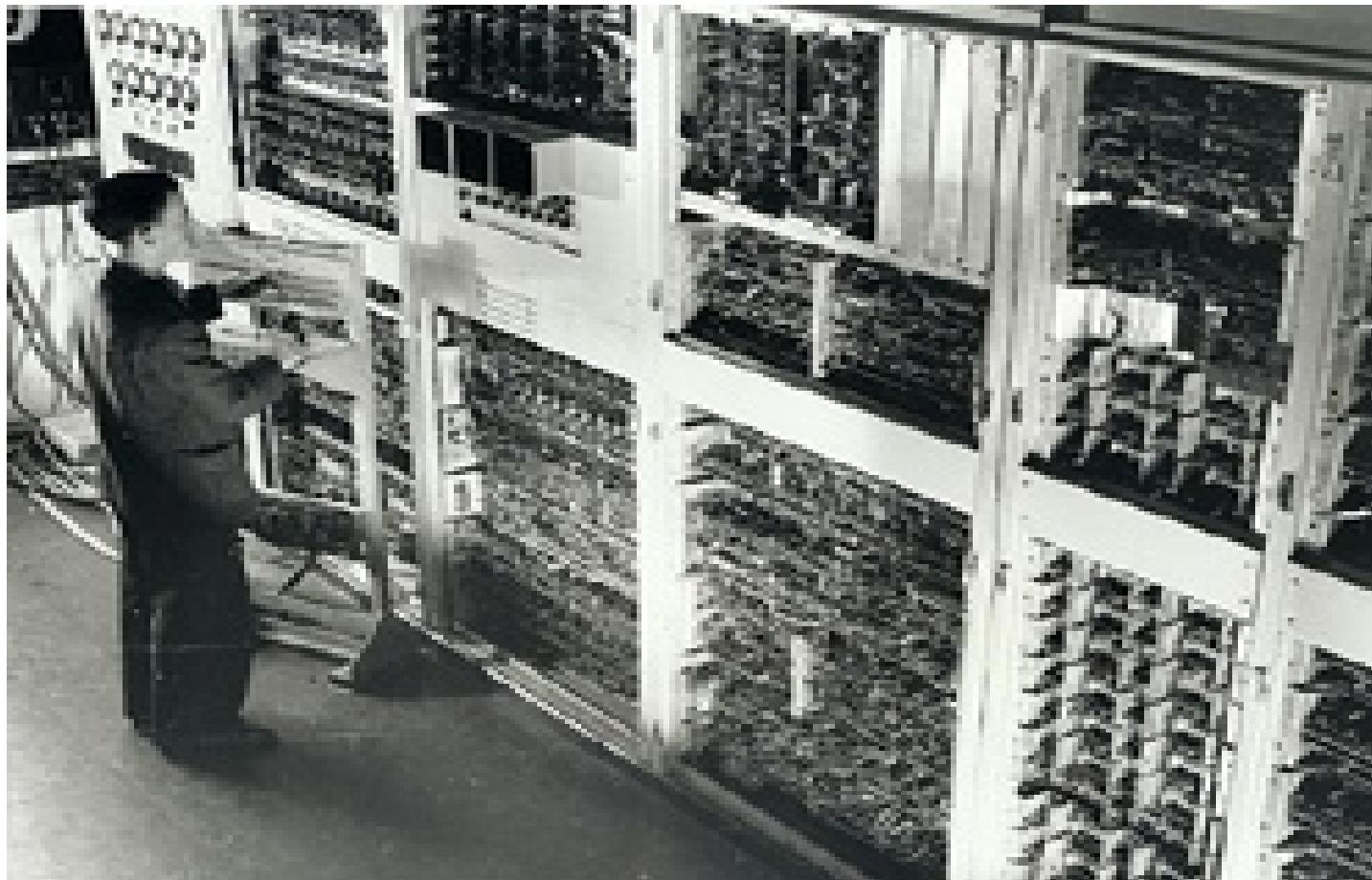
- Early Systems/Serial Processing (mid 1940s to mid 1950s)
- Simple Batch Systems (1960)
- Multi-programmed Batch Systems (1970)
- Time-Sharing and Real-Time Systems (1970)
- Multiprocessor Systems (1980)
- Networked/Distributed Systems (1980)

# Early Systems/Serial Processing

- A Programmer/User as operator interacted directly with the hardware
- Large machines run from console, since **no OS**
- Programs in machine code are read through card reader



# Example of an early computer system



# Early Systems /Serial Processing

- Machines run from a console with display lights, toggle switches, input device, and printer
- If no error, program output in printer, or else error indicated by lights
- Extremely slow I/O devices
- Very low CPU utilization



# Early Systems /Serial Processing

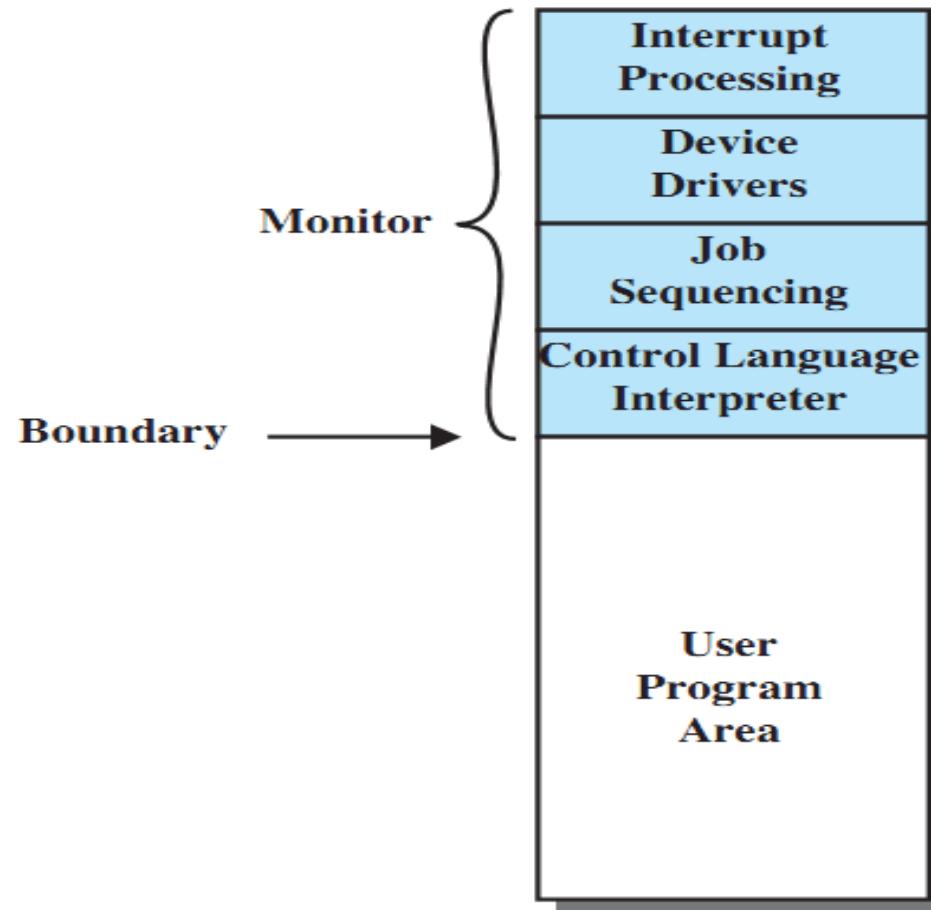
- Serial processing
  - Scheduling:
    - User should reserve computer time in hard copy sheet, in multiples of 30 minutes or so.
    - Overhead in scheduling time.
  - Setup included loading the compiler, source program, saving compiled program, and loading and linking
    - Need significant amount of setup time in mounting and dismounting card decks
  - Various system software tools were developed to make serial processing easier – linkers, loaders, debuggers and I/O driver routines



# Simple Batch Systems

- Early computers were expensive - important to maximize CPU utilization
  - Scheduling and setup time was unacceptable
- Jobs of users are batched together by an operator - submits the entire batch to the input device for use by the **monitor**.
- After each job, the control branches back to the monitor, and the monitor automatically begins loading the next job for execution
- A special program, the monitor, manages the execution of each program in the batch.
- Monitor utilities are loaded when needed – compilers, assemblers, drivers...
- “Resident monitor” is always in main memory and available for execution that controls the sequence of events

# Resident Monitor Layout



# Control Cards

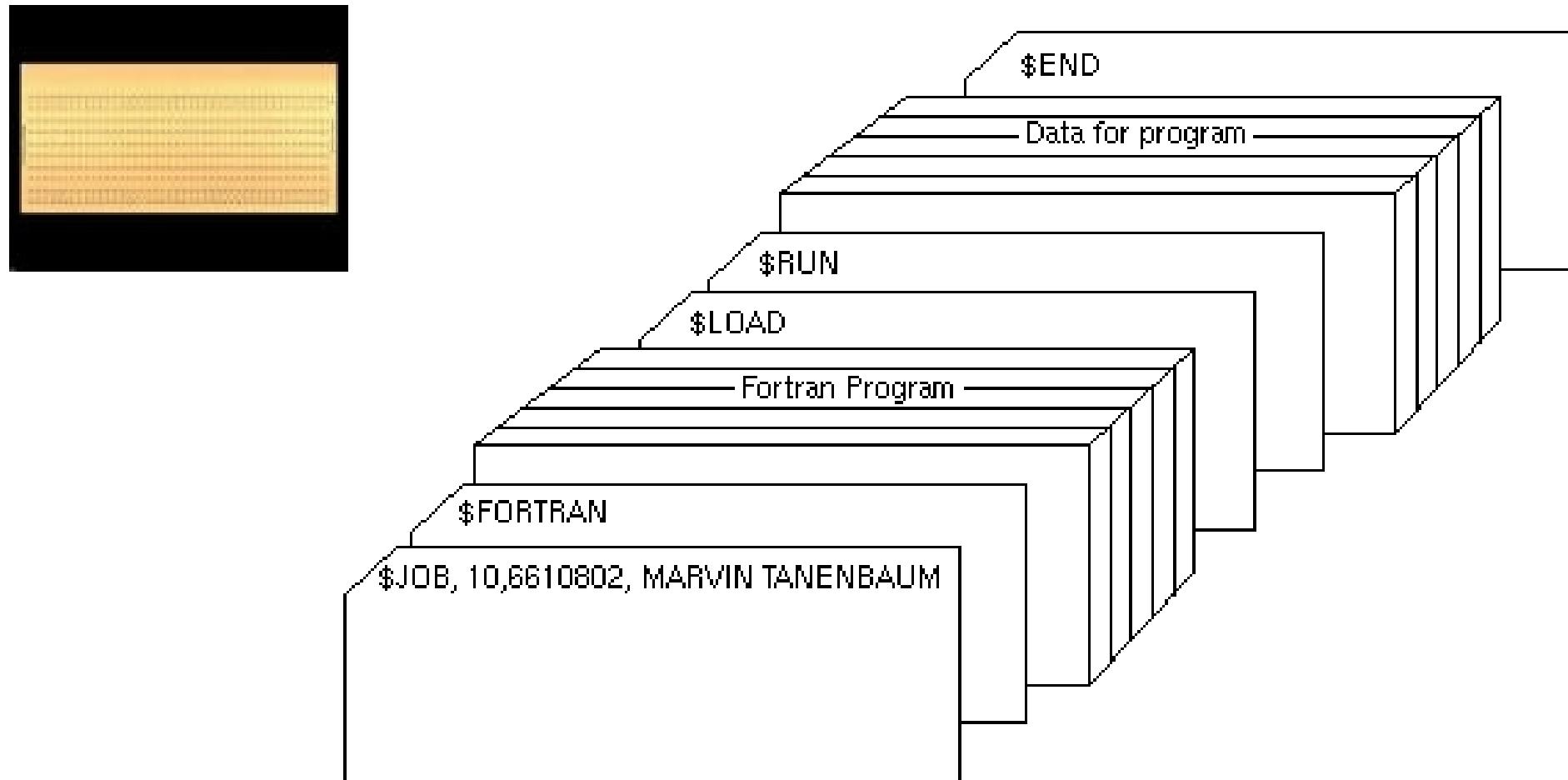
- Problems:
  - How does the monitor know about the nature of the job (e.g., Fortran versus Assembly) or which program to execute?
  - How does the monitor distinguish:
    - (a) job from job?
    - (b) data from program?
- Solution: Job Control Language (JCL) and control cards.

# Job Control Language (JCL)

- JCL is the language that provides instructions to the monitor:
  - what compiler to use
  - what data to use
- Example of job format: ----->>
  - \$FTN loads the FORTRAN compiler and transfers control to it.
  - \$LOAD loads the object code (along in place of compiler).
  - \$RUN transfers control to user program.

```
$JOB  
$FTN  
...  
FORTRAN  
program  
...  
$LOAD  
$RUN  
...  
Data  
...  
$END
```

# Example card deck of a Job



# Idea of Simple Batch Systems

- Reduce setup time by batching similar jobs.
- Alternate execution between user program (user mode) and the monitor program (kernel mode).
- Sacrifices:
  - some main memory is now given over to the monitor
  - some processor time is consumed by the monitor

# Desirable Hardware Features

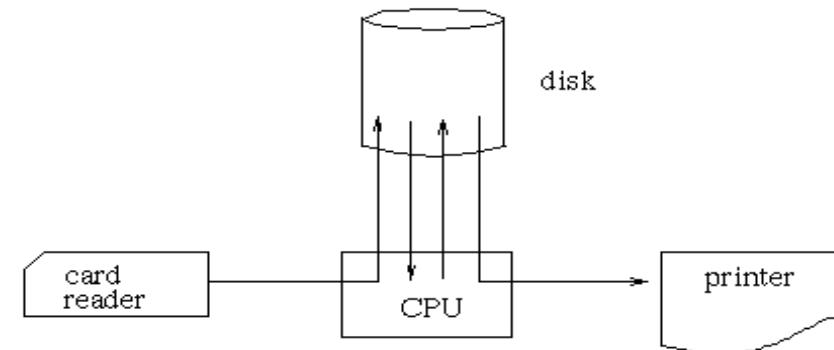
- Memory protection
  - do not allow the memory area containing the monitor to be altered by a user program.
- Privileged instructions
  - can be executed only by the resident monitor.
  - A trap occurs if a program tries these instructions.
- Interrupts
  - provide flexibility for relinquishing control to and regaining control from user programs.
  - Timer interrupts prevent a job from monopolizing the system.

# Spooling

- Problem:
  - Card reader, Line printer and Tape drives slow (compared to Disk).
  - I/O and CPU could not overlap.
- Solution: Spooling -
  - Overlap I/O of one job with the computation of another job (using double buffering, DMA, etc).
  - Technique is called SPOOLing: Simultaneous Peripheral Operation On Line.

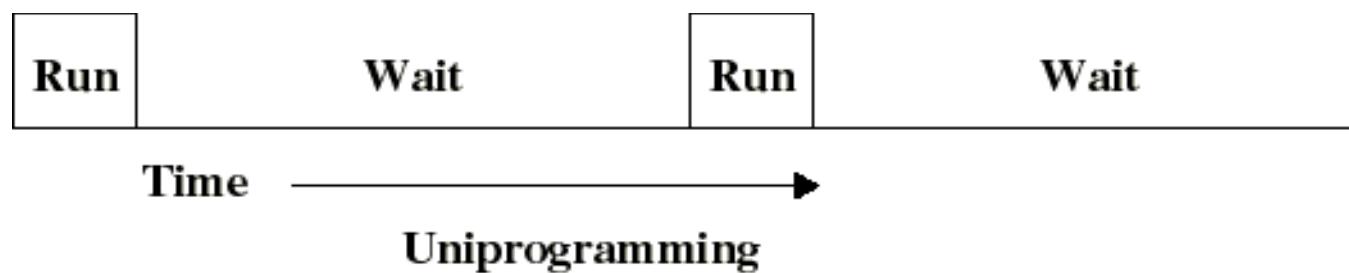
# Spooling

- While executing one job, the OS:
  - Reads next job from card reader into a storage area on the disk (Job pool).
  - Outputs printout of previous job from disk to printer.
- Job pool – data structure that allows the OS to select which job to run next in order to increase CPU utilization.

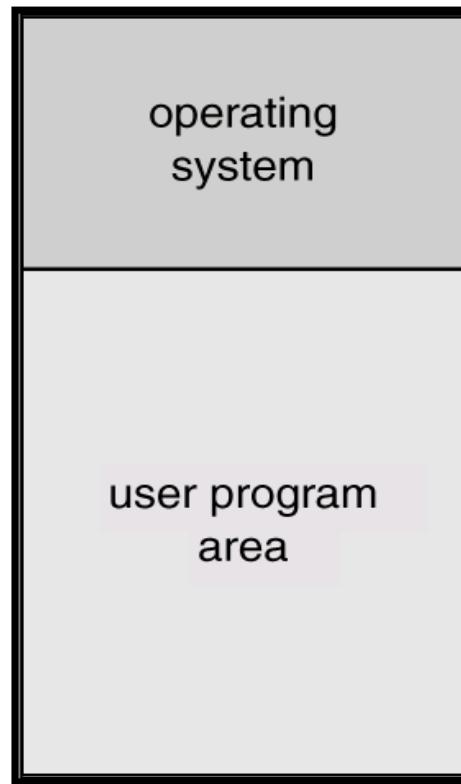


# Uni-programming until now

- I/O operations are exceedingly slow (compared to instruction execution).
- A program containing even a very small number of I/O operations, will spend most of its time waiting for them.
- Hence: poor CPU usage when only one program is present in memory.

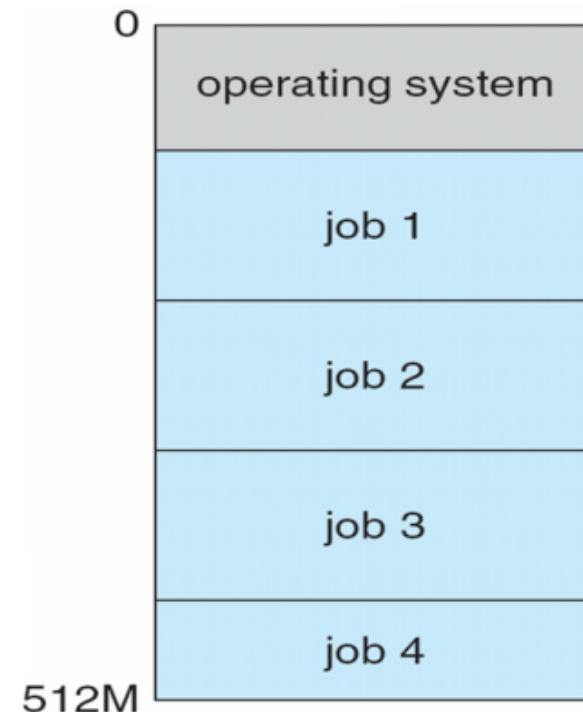


# Memory Layout for Uni-programming



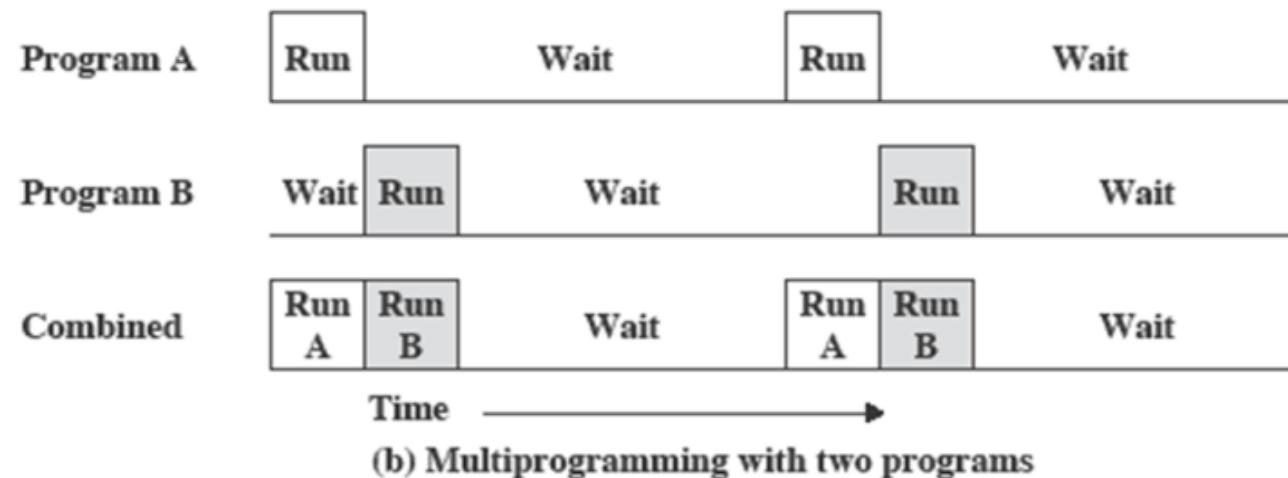
# Memory Layout for Batch Multiprogramming

- Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them.



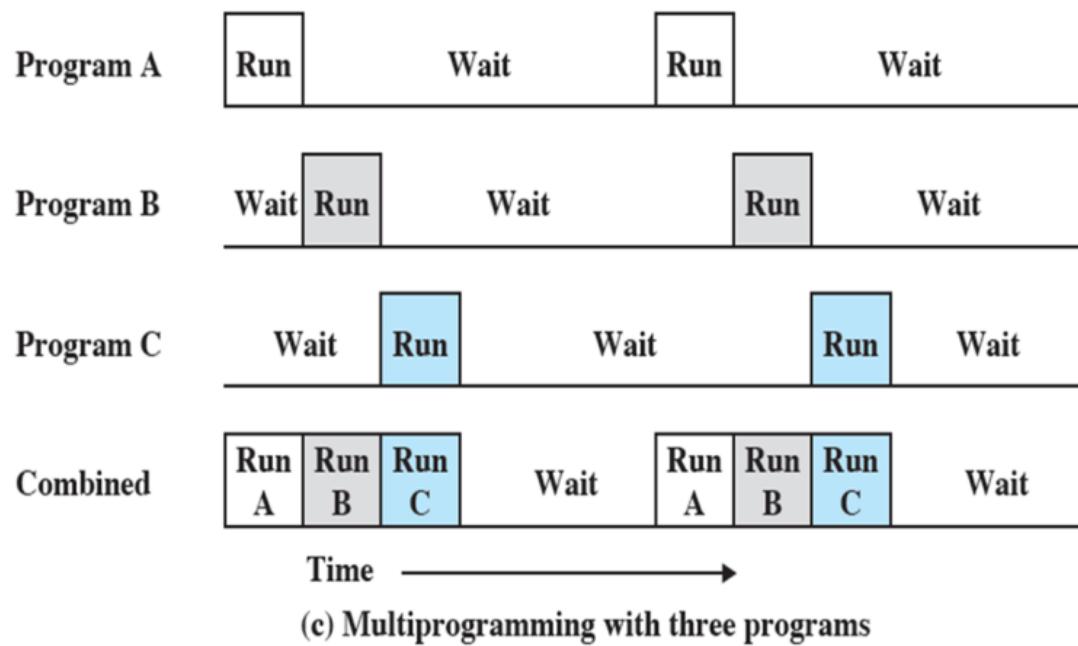
# Multiprogramming

- There must be enough memory to hold the OS (resident monitor) and one user program
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O



# Multiprogramming

- Multiprogramming
  - also known as multitasking
  - memory is expanded to hold three, four, or more programs and switch among all of them



# Why Multiprogramming?

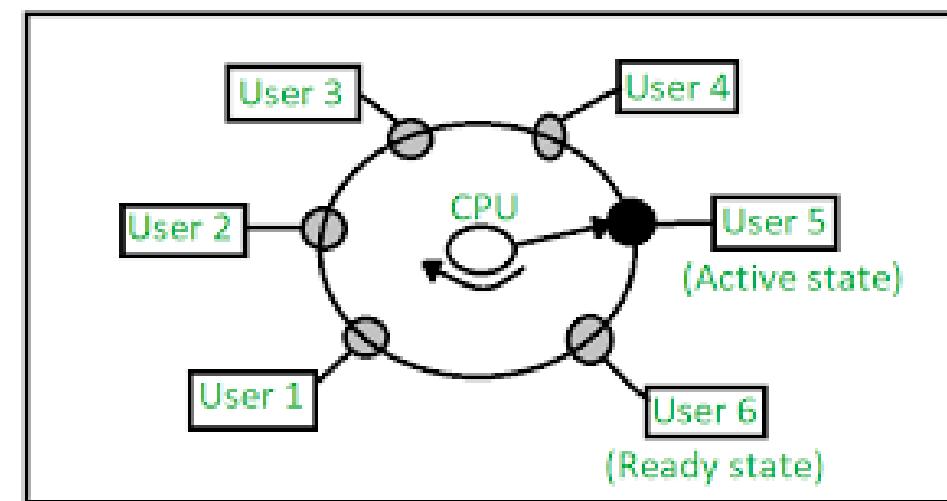
- Multiprogramming needed for efficiency:
  - Single user cannot keep CPU and I/O devices busy at all times.
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute.
  - A subset of total jobs in system is kept in memory.
  - One job selected and run via job scheduling.
  - When it has to wait (for I/O for example), OS switches to another job.

# Requirements for Multiprogramming

- Hardware support:
  - I/O interrupts and DMA controllers
    - in order to execute instructions while I/O device is busy.
  - Timer interrupts for CPU to gain control.
  - Memory management
    - several ready-to-run jobs must be kept in memory.
  - Memory protection (data and programs).
- Software support from the OS:
  - For scheduling (which program is to be run next).
  - To manage resource contention.

# Time-Sharing Systems

- Can be used to handle multiple interactive jobs
- Processor time is shared among multiple users
- Multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short CPU burst or time quantum of computation



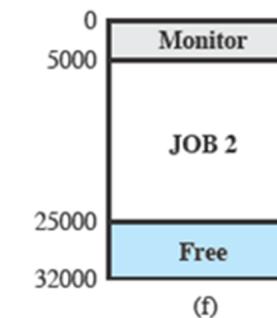
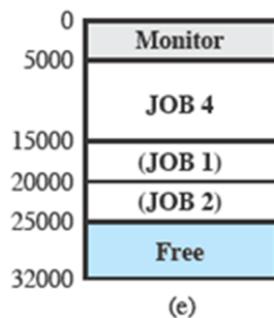
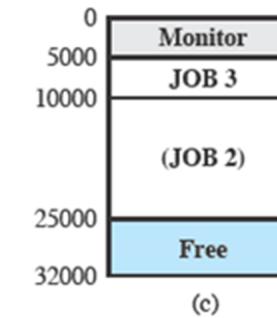
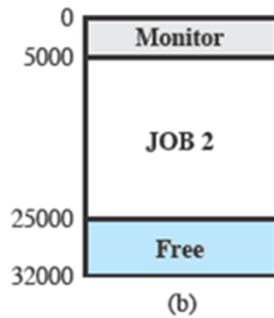
# Compatible Time-Sharing Systems CTSS

- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC
- Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that
- To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word

# Time Slicing

- System clock generates interrupts at a rate of approximately once every 0.2 seconds
- At each interrupt OS regained control and could assign processor to another user
- At regular time intervals the current user would be preempted and another user loaded in
- Old user programs and data were written out to disk
- Old user program code and data were restored in main memory when that program was next given a turn

# CTSS Operation



# Real-time Operating Systems

- Real-time operating system (RTOS) is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay.
- It is time-bound system that can be defined as fixed time constraints.
- In this type of system, processing must be done inside the specified constraints. Otherwise, the system will fail.

# Real-time Systems

- Three types of RTOS systems are:
- Hard Real Time :
  - In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.
  - Example: Medical critical care system, Aircraft systems, etc.

# Real-time Systems

- Firm Real time:
  - These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired affects, like a huge reduction in quality of a product.
  - Example: Various types of Multimedia applications.
- Soft Real Time:
  - Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.
  - Example: Online Transaction system and Livestock price quotation System.

# Real-time Systems

- Real-time systems are used in:
  - Airlines reservation system.
  - Air traffic control system.
  - Systems that need immediate updating.
  - Used in any system that provides up to date and minute information on stock prices.
  - Defense application systems like RADAR.
  - Networked Multimedia Systems
  - Command Control Systems

# Multi-processor Systems

- Multiprocessor Operating System refers to the use of two or more central processing units (CPU) within a single computer system.
- These multiple CPUs are in a close communication sharing the computer bus, memory and other peripheral devices.
- These systems are referred as tightly coupled systems.
- These types of systems are used when very high speed is required to process a large volume of data.
- These systems are generally used in environment like satellite control, weather forecasting etc.

# Multi-processor Systems

- Symmetric Multiprocessors
  - In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer to peer relationship i.e. no master - slave relationship exists between them.
  - An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer.

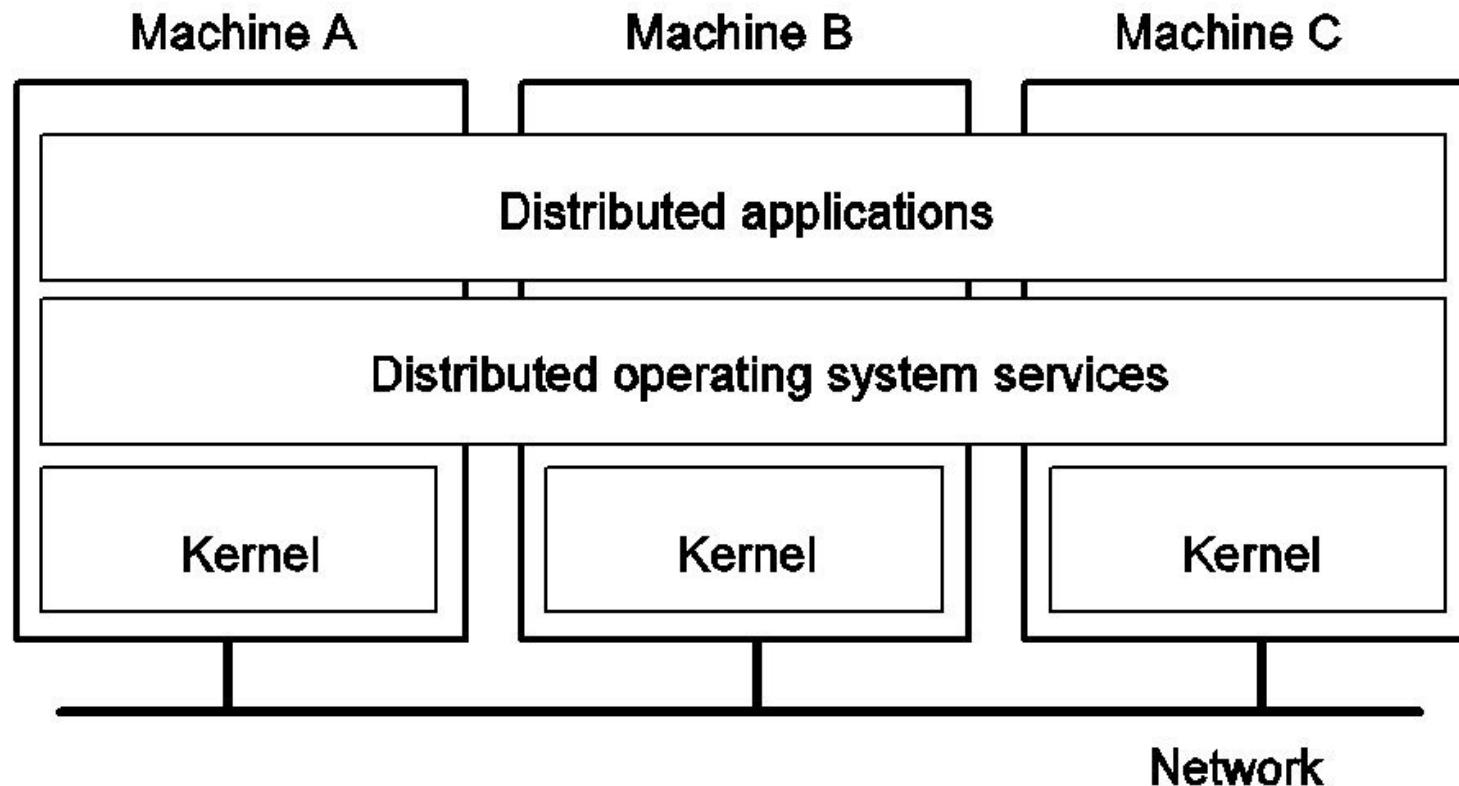
# Multi-processor Systems

- Asymmetric Multiprocessors
  - In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors. Asymmetric multiprocessor system contains a master slave relationship.
  - Asymmetric multiprocessor was the only type of multiprocessor available before symmetric multiprocessors were created.

# Distributed Operating Systems

- A DOS is a system which contains multiple components located on different machines, which coordinate and communicate actions in order to appear as a single coherent working system to the user.
- A distributed operating system (DOS), are systems which model where distributed applications are running on multiple computers, linked by communications.
- All software and hardware compounds are located remotely. In order for them to communicate with each other, they pass messages.

# Distributed Operating Systems (DOS)



# Network Operating System

- Network Operating System is an operating system that includes special functions for connecting computers and devices into a local-area network (LAN) or Inter-network.
- Some popular network operating systems are Novell Netware, Windows NT/2000, Linux, Sun Solaris, UNIX, and IBM OS/2.
- An operating system that provides the connectivity among a number of autonomous computers is called a network operating system.
- A typical configuration for a network operating system is a collection of personal computers along with a common printer, server and file server for archival storage, all tied together by a local network.

# Network Operating System

- Provide file, print, web services and back-up services.
- Support Internet working such as routing and WAN ports.
- User management and support for logon and logoff, remote access; system management, administration and auditing tools with graphical interfaces.
- In this, the users can remotely access each other.
- It also includes security features.
  - Example: authentication of data, restrictions on required data, authorizations of users etc.

# Homework

- Clustered Systems
- Peer-Peer Systems
- Mobile Computing Systems
- Web/Cloud-based Systems

# Operating Systems Structures

# Operating Systems Structures

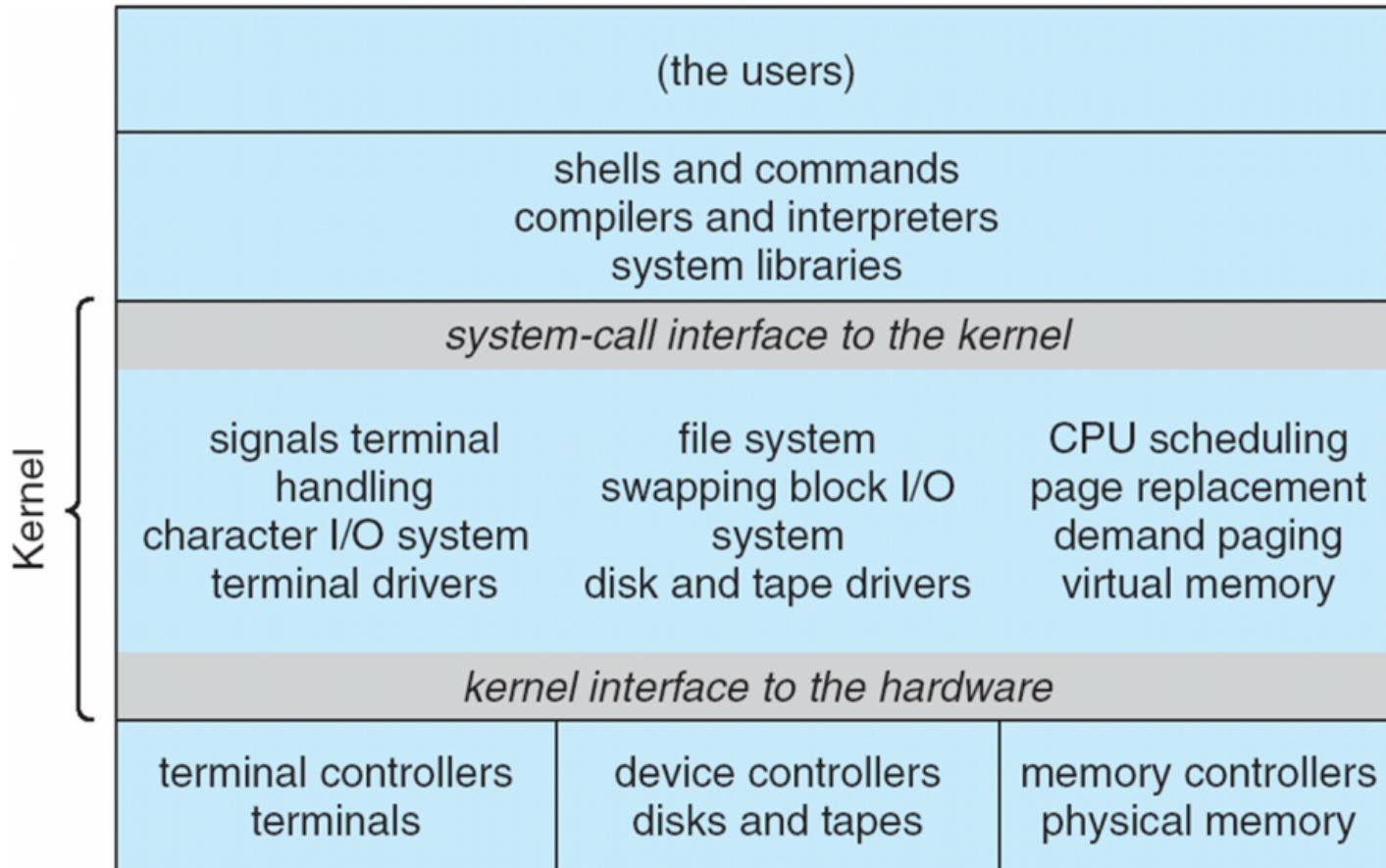
- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily
- Structure/Organization/Layout of OSs:
  - Monolithic Structure(one unstructured program)
  - Layered Approach
  - Microkernels
  - Hybrid systems



# Monolithic Structure

- The simplest structure for organizing an operating system is no structure at all.
- That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space.
  - Eg: Traditional UNIX System Structure
- Everything below the system-call interface and above the physical hardware is the kernel.
- The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls.
- Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

# Traditional UNIX System Structure



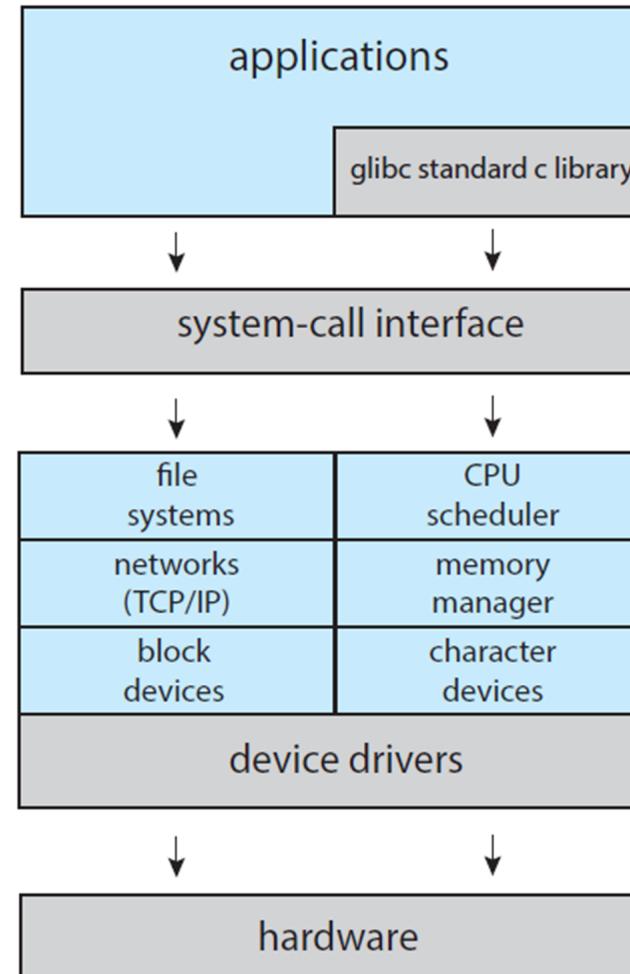
# Traditional UNIX System Structure

- UNIX – limited by hardware functionality, the original UNIX OS 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.

# Linux system structure

- The Linux operating system is based on UNIX and is structured similarly, as shown in below Figure.
- Applications typically use the glibc standard C library when communicating with the system call interface to the kernel.
- The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but as we shall see later that, it does have a modular design that allows the kernel to be modified during run time.

# Linux system structure



# Monolithic Structure

- Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend.
- Despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

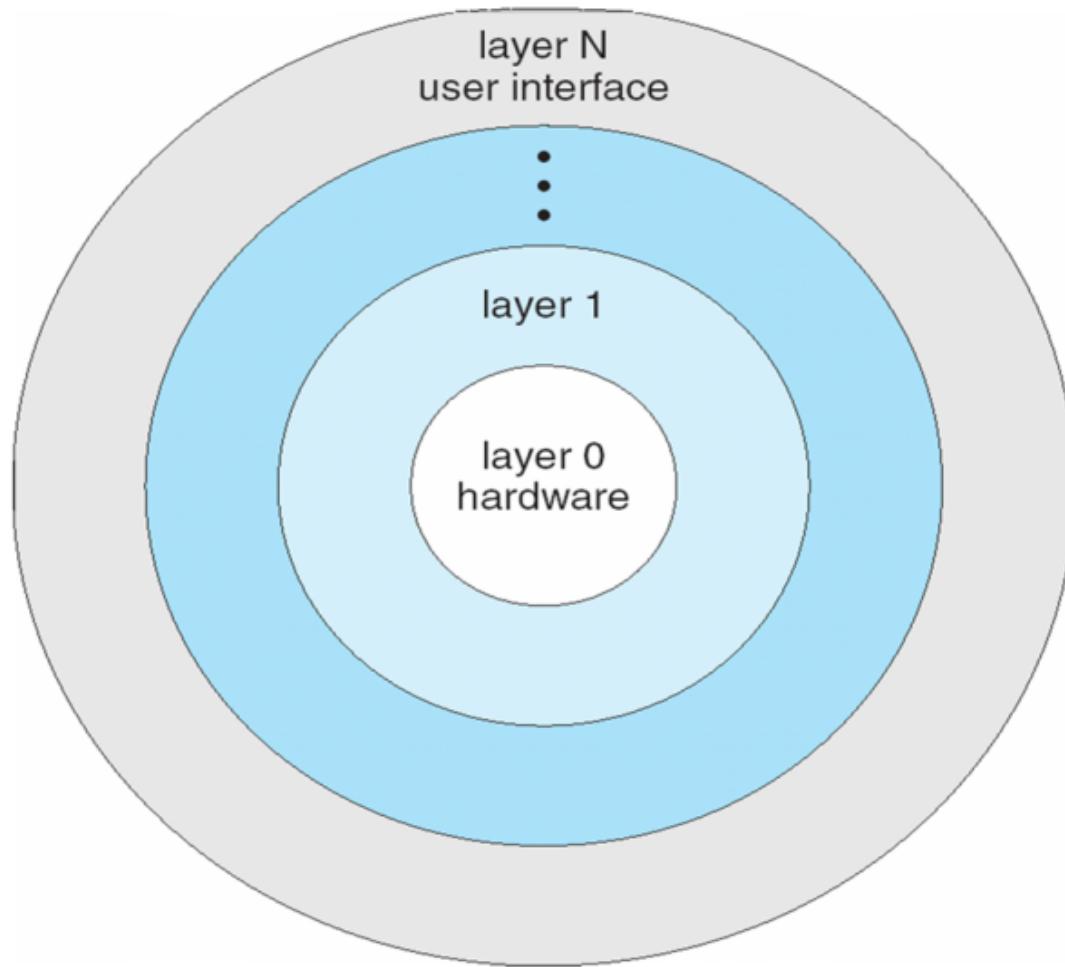
# Layered Approach

- The monolithic approach is often known as a tightly coupled system because changes to one part of the system can have wide-ranging effects on other parts.
- Alternatively, we could design a loosely coupled system.
- Such a system is divided into separate, smaller components that have specific and limited functionality.
- All these components together comprise the kernel.
- The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

# 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.
- Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

# Layered Operating System



# Layered Approach

- Relatively few operating systems use a pure layered approach.
- One reason involves the challenges of appropriately defining the functionality of each layer.
- In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.
- Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

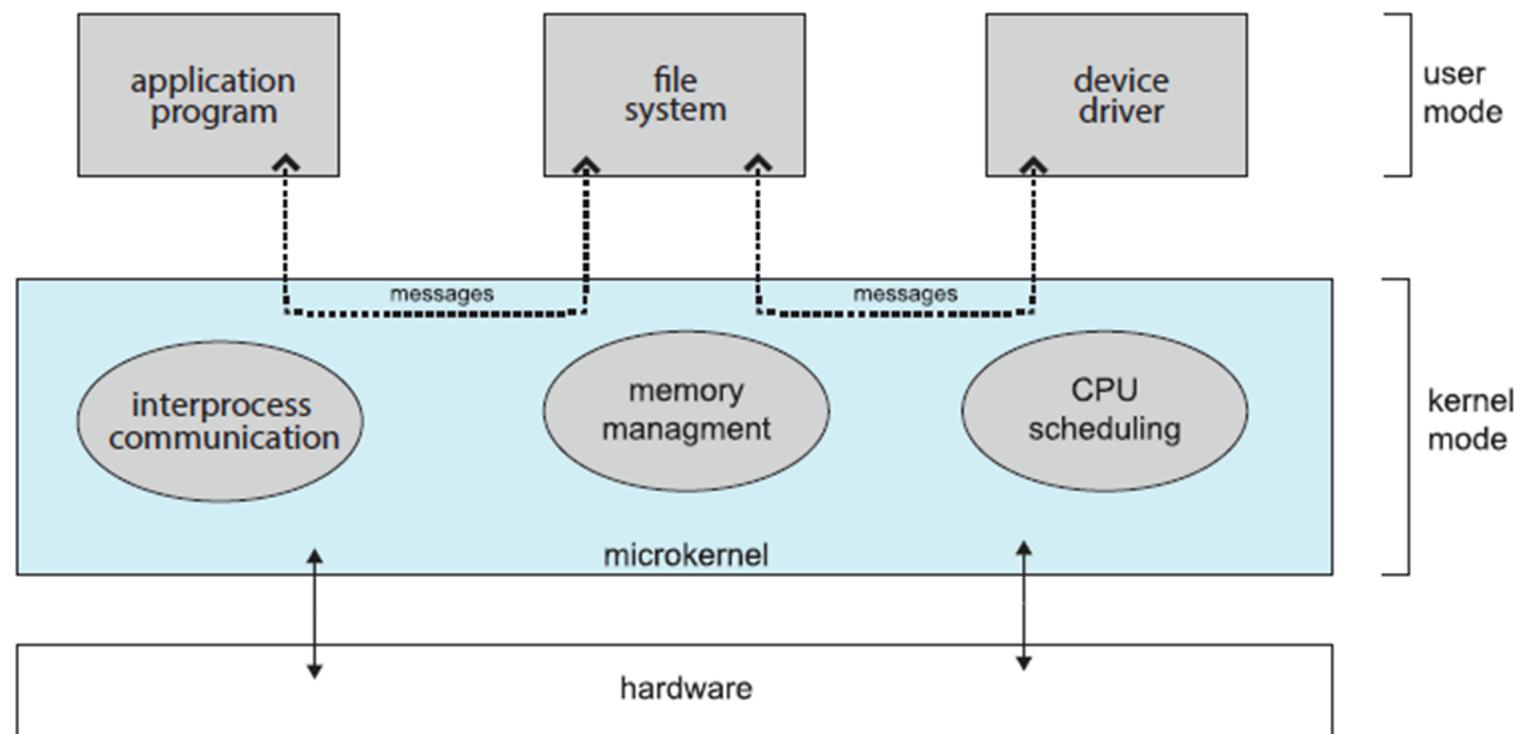
# Microkernel System Structure

- We have already seen that the original UNIX system had a monolithic structure.
- As UNIX expanded, the kernel became large and difficult to manage.
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces.

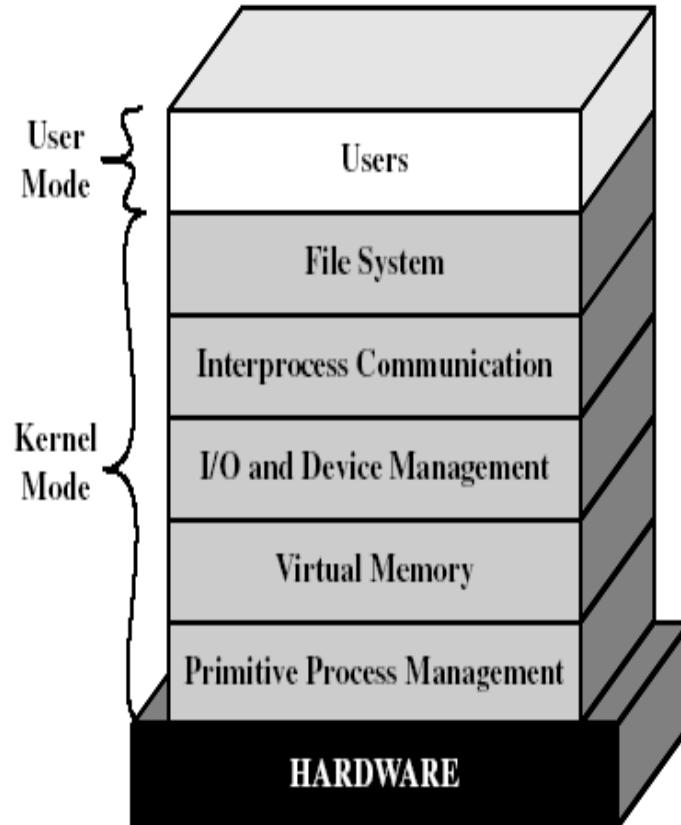
# Microkernel System Structure

- Move as much functionality as possible from the kernel into “user” space.
- Only a few essential functions in the kernel:
  - primitive memory management (address space)
  - I/O and interrupt management
  - Inter-Process Communication (IPC)
  - basic scheduling
- Other OS services are provided by processes running in user mode (vertical servers):
  - device drivers, file system, virtual memory...

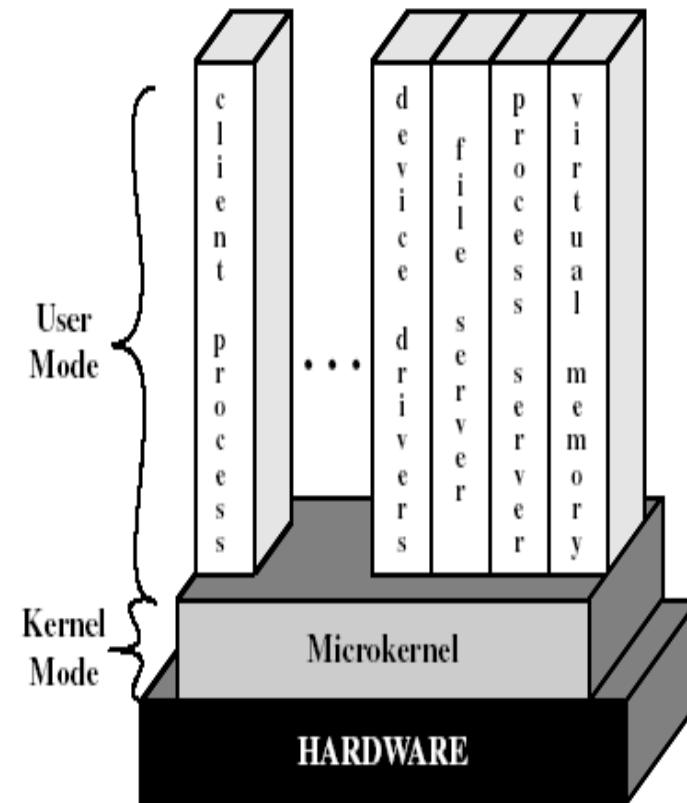
# Architecture of a typical microkernel



# Layered vs. Microkernel Architecture



(a) Layered kernel



(b) Microkernel

# Microkernel System Structure

- Communication takes place between user modules using message passing.
- More flexibility, extensibility, portability and reliability.
- But performance overhead caused by replacing service calls with message exchanges between processes.



# Benefits of a Microkernel Organization

- Extensibility/Reliability
  - 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
  - small microkernel can be rigorously tested.
- Portability
  - changes needed to port the system to a new processor is done in the microkernel, not in the other services.

# Benefits of Microkernel Organization

- Distributed system support
  - messages are sent without knowing what the target machine is.
- Object-oriented operating system
  - components are objects with clearly defined interfaces that can be interconnected to form software.

# Disadvantages of Microkernel Structure

- Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.
- When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces.
- In addition, the operating system may have to switch from one process to the next to exchange the messages.
- The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems.

# Modules

- The best current methodology for operating-system design involves using loadable kernel modules (LKMs).
- Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time.
- This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.
- The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running.

# Modules

- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module.

# Modules

- The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.
- Linux uses loadable kernel modules, primarily for supporting device drivers and file systems.
- LKMs can be “inserted” into the kernel as the system is started (or booted) or during run time, such as when a USB device is plugged into a running machine.
- If the Linux kernel does not have the necessary driver, it can be dynamically loaded.
- LKMs can be removed from the kernel during run time as well.
- For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system.

# Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure.
- Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.
- For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance.
- However, it also modular, so that new functionality can be dynamically added to the kernel.

# Hybrid Systems

- Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes.
- Windows systems also provide support for dynamically loadable kernel modules.

# Process

# Introduction

- Early computers allowed only one program to be executed at a time.
- This program had complete control of the system and had access to system resources.
- In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed simultaneously.
- This resulted in the notion of a process, which is a program in execution.

# Process Concept

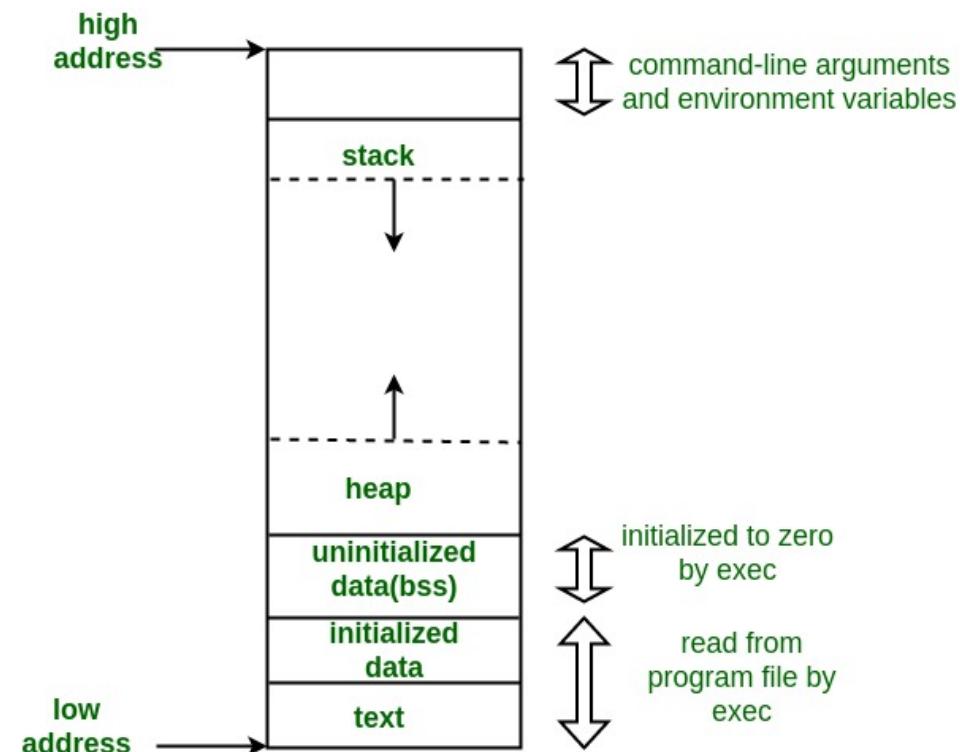
- A time shared system has user programs or tasks.
- Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser and an email package.
- The above all are processes.

# The Process

- A process is a program in execution.
- But it is more than program code (text section).
- It also includes current activity as represented by the value of the program counter, process-id and the contents of the CPU registers.
- A process generally includes,
  - Process stack (contains temporary data - function parameters, return addresses and local variables)
  - Data section (contains global variables)
  - Heap (for dynamically allocated memory)

# Memory Layout of C Programs

- A typical memory representation of C program consists of following sections
  - Text segment
  - Initialized data segment
  - Uninitialized data segment
  - Stack
  - Heap



# Memory Layout of C Programs

- The GNU **size** command can be used to determine the size (in bytes) of some of these sections.
- Assuming the name of the executable file of the above C program is **memory**, the following is the output generated by entering the command **size memory**:

<b>text</b>	<b>data</b>	<b>bss</b>	<b>dec</b>	<b>hex</b>	<b>filename</b>
1158	284	8	1450	5aa	memory

- The data field refers to uninitialized data, and bss refers to initialized data. (bss is a historical term referring to block started by symbol.)
- The dec and hex values are the sum of the three sections represented in decimal and hexadecimal, respectively.

# The Process

- A program by itself is not a process.
- A program is a passive entity, such as a file containing a list of instructions stored on a disk (often an exe file).
- A process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an exe file is loaded into memory.
  - Eg: double clicking pgm.exe (or) typing ./a.out in command prompt

# The Process

- Although two processes may be associated with same program, they are nevertheless two separate execution sequences.
- Eg:
  - Several users running same copy of the mail program.  
(or)
  - The same user may invoke many copies of the web browser program.
  - Each of these is a separate process and although the text sections are equivalent, the data, heap and stack segments vary.

# Process Control Block (PCB)

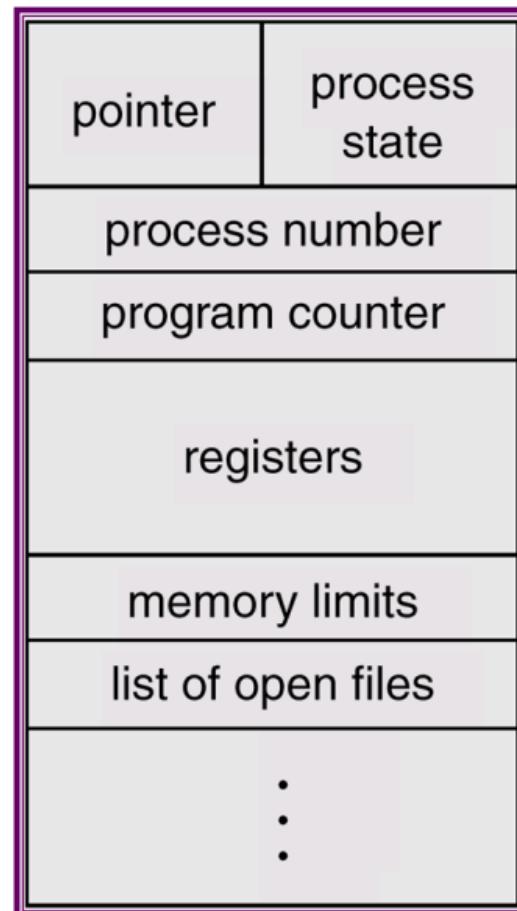
- Each process is represented in the OS by a PCB.
- It contains many pieces of information associated with a specific process.
- PCB contains sufficient information so that it is possible to interrupt a running process and later resume execution.
- It enables OS to support multiple processes.

# Process Control Block (PCB)

Information associated with each process.

- Process identifier
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

# Process Control Block (PCB)

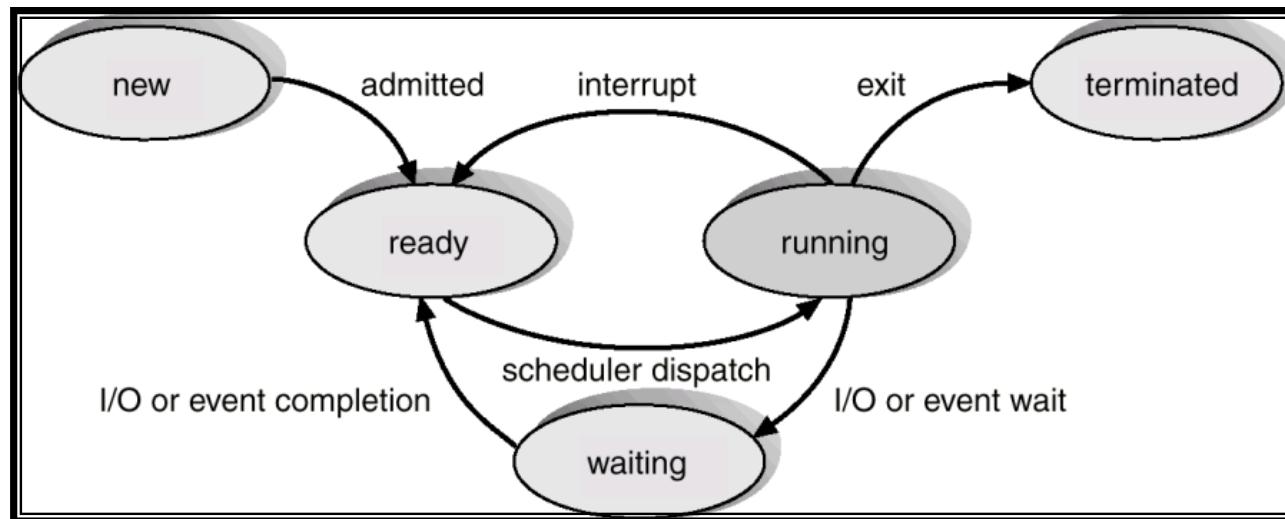


# Process State

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- A process may be in any one of the following states.
  - new: The process is being created.
  - running: Instructions are being executed.
  - waiting/blocked: 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.

# Five State Process Model

- Only one process can be running an any processor at any instant.
- Many processes can be ready and waiting



# State transitions

- Null->new
  - A new process is created to execute a program
- New->ready
  - The OS will move process from new to ready when it is prepared to take on additional process. There is a limit on number of processes in main memory
- Ready->running
  - When it is time to select a process to run, the OS chooses one of the processes in ready state. Done by scheduler or dispatcher.
- Running->exit
  - Currently running process terminated by OS, if the process indicates that it has completed or if it aborts.

# State transitions

- Running -> ready
  - The most common reason for this transition is that the running processes has reached the max allowable time or some other high priority process has come in for execution
- Running->Blocked
  - A process is put to blocked state if it requests for OS services like an I/O operation, or resource.
- Blocked->ready
  - When the event for which the process has been waiting occurs this transition happens
- Ready->exit/Blocked->exit
  - A parent process may terminate a child process at any time. Also if parent terminates all child processes associated with parent may also terminate.

# Suspended Processes

- Processor is faster than I/O devices, so many processes could be waiting for I/O
  - Swap these processes to disk to free up more memory and use processor on more processes
- Blocked/waiting state becomes ***suspend*** state when swapped to disk

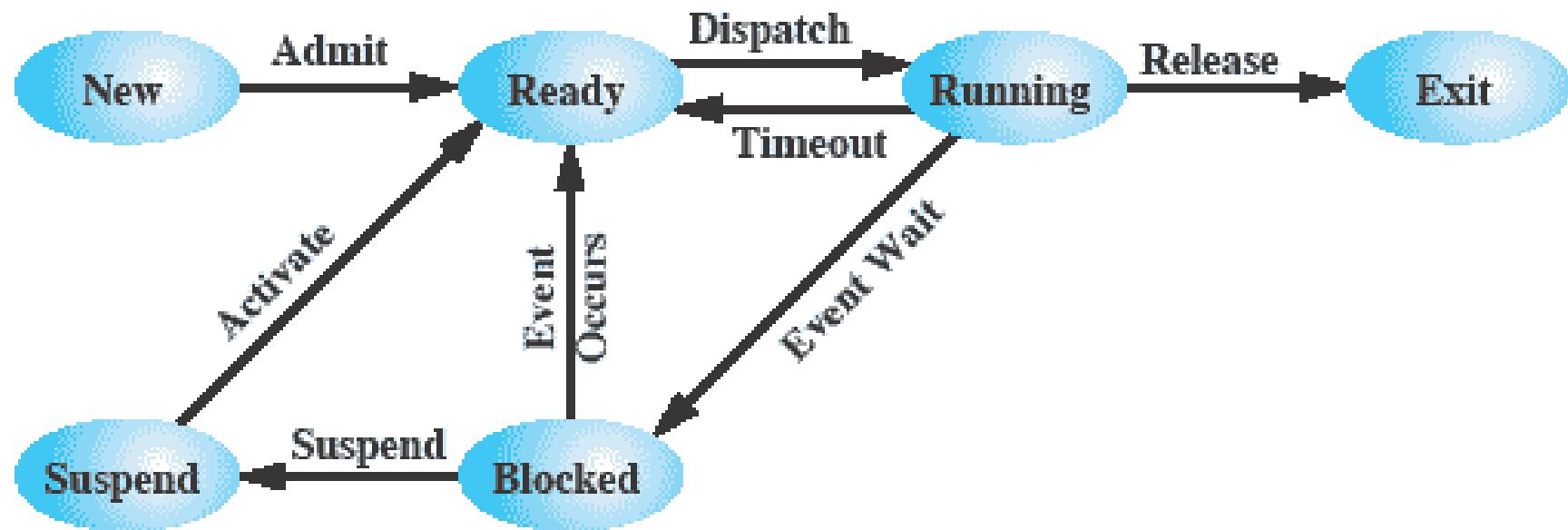
# Reasons for Process Suspension

Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

# Process States during Suspension

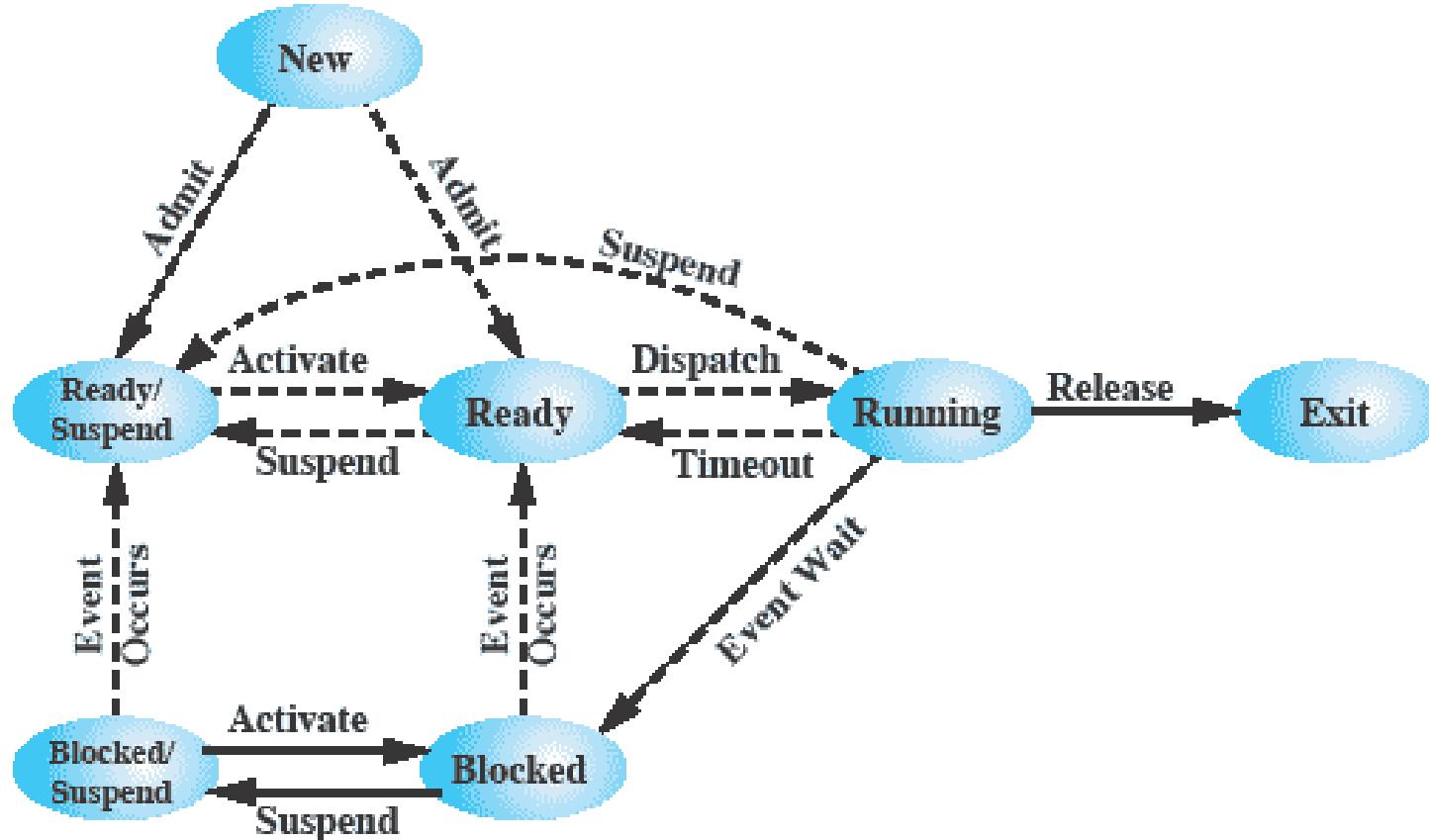
- One new state
  - Blocked
- or
- Two new states
  - Blocked/Suspend
  - Ready/Suspend

# One Suspend State



(a) With One Suspend State

# Two Suspend States - Seven State Process Model



(b) With Two Suspend States

# Additional process states

- Blocked/Suspend : the process in secondary memory and awaiting an event
- Ready/ Suspend: the process in secondary storage but is available for execution as soon as it is loaded in memory

# New State transitions

- Blocked ->blocked/suspend
  - If there are no ready processes, then at least one blocked process is swapped out to secondary storage to make space for another process which is not blocked.
- Blocked/suspend ->ready/suspend
  - This transition happens when the event for which the process has been waiting for occurs
- Ready/suspend->ready
  - When no ready processes in memory, the OS will need to bring one in to continue execution or process in ready/suspend state has high priority than ready state processes.
- Ready->ready suspend
  - Usually OS will suspend a blocked process to suspend state. But it may be necessary to suspend a ready process if that is the only way to free large block of main memory or blocked process may be high priority process than ready process

# Other transitions to consider

- New->ready/suspend and new ->ready
  - When a new process is created it can be added either to ready state or ready suspend state based on the availability of main memory.
- Blocked/Suspend->Blocked
  - It would be reasonable to bring a blocked process into main memory if it is a high priority process than any other process in ready suspend state and the OS has reason to believe that the blocking event for the process will occur soon.
- Running ->ready/suspend
  - OS can pre-empt a running process and move it to ready suspend state directly because a blocked suspended high priority process has just become unblocked.
- Any state ->exit
  - A process get terminated while running, due to its completion or fatal error conditions. Sometimes if a parent process terminates, the child processes associated with the parent process can also terminate, whatever may be the state the child process are in.

# New State of a Process

- A new state corresponds to a process that has just been defined.
- An identifier is associated with the process.
- Any tables needed to manage the process are allocated and built.
- At this stage the process is in new state.
- While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory.
- When the process is in new state, the code of the program is not in main memory and no space has been allocated for the data associated with the program.
- The program remains in secondary storage.

# Reasons for Process Creation

- When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.

New batch job

The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.

Interactive logon

A user at a terminal logs on to the system.

Created by OS to provide a service

The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).

Spawned by existing process

For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

# Process Creation

## *Process spawning*

- when the OS creates a process at the explicit request of another process

## *Parent process*

- is the original, creating, process

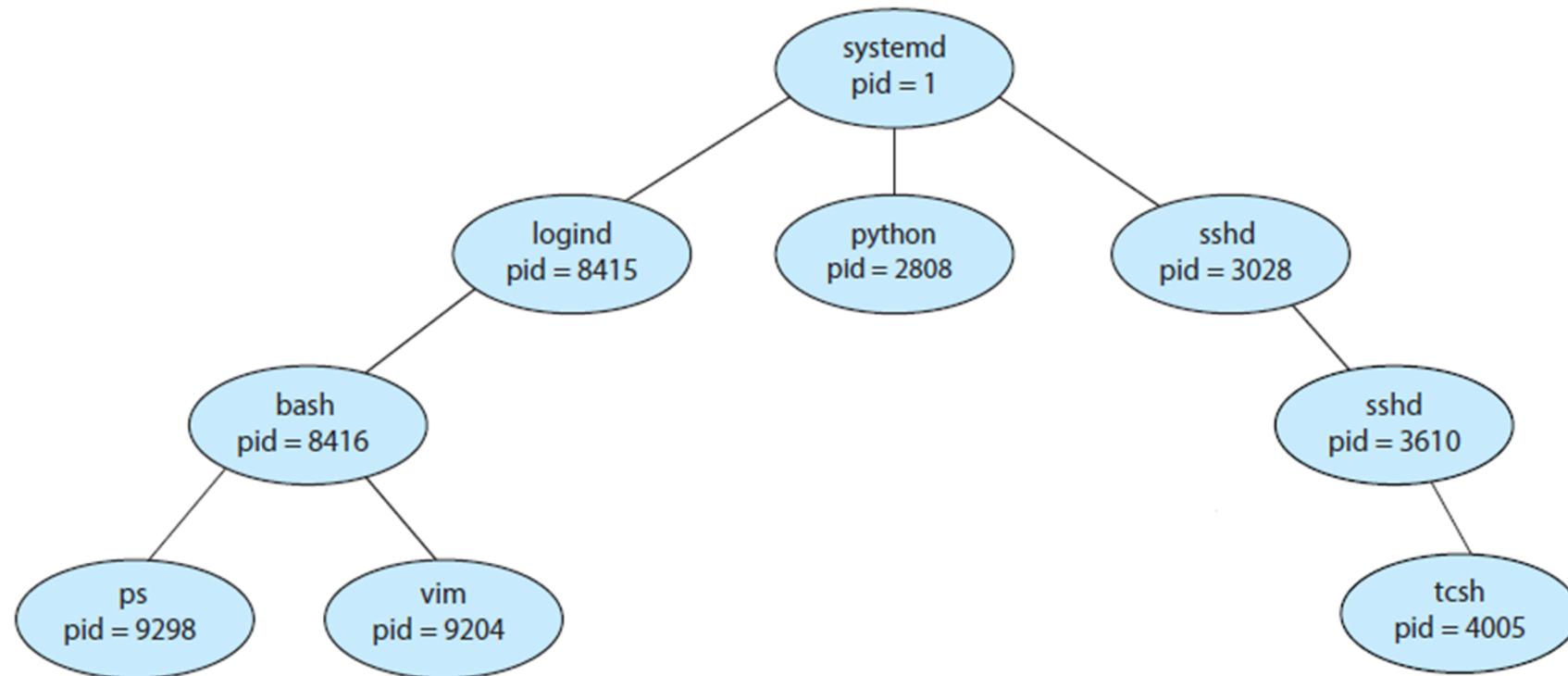
## *Child process*

- is the new process

# Process Creation

- During the course of execution, a process may create several new processes.
- The creating process is called a parent process, and the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a tree of processes.
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or pid), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

# Process Creation



The above figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.

# Process Creation

- The systemd process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots.
  - Once the system has booted, the systemd process creates processes which provide additional services such as a web or print server, an ssh server, and the like.
- Two children of systemd—logind and sshd.
  - The logind process is responsible for managing clients that directly log onto the system.
  - In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
  - Using the bash command-line interface, this user has created the process ps as well as the vim editor.
  - The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell).

# Process Creation

- Traditional UNIX systems identify the process init as the root of all child processes.
- init (also known as System V init) is assigned a pid of 1, and is the first process created when the system is booted.
- On a process tree similar to what is shown in previous figure, init is at the root.
- Linux systems initially adopted the System V init approach, but recent distributions have replaced it with systemd.
- systemd serves as the system's initial process, much the same as System V init; however it is much more flexible, and can provide more services, than init.

# Process Creation

- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.
- For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system.

- When a process creates a new process, two possibilities for execution exist:
  - The parent continues to execute concurrently with its children.
  - The parent waits until some or all of its children have terminated.
- There are also two address-space possibilities for the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent - text segment is common for both process, data segment of parent duplicated(copied) for child process).
  - The child process has a new program loaded into it.

# C Program - Process Creation & Termination

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
    int forkresult;

    printf("Parent now going to fork a child... \n");

    forkresult=fork();

    if(forkresult== -1)
    {
        printf("Fork failed\n");
        exit(0);
    }
```

# C Program - Process Creation & Termination

```
else if(forkresult!=0)
{
    printf("Parent waits for child to execute ls then parent executes pwd.\n");
    wait(NULL);
    execlp("pwd","pwd",NULL);
    printf("This line gets printed when the above exec statement pwd fails\n");
}
else
{
    printf("Child : I'm now going to execute ls!\n\n");
    execlp ("ls", "ls", NULL);
    printf ("Child prints %d: AAAAH !! My EXEC failed !!! !\n", getpid());
    exit(1);
}
```

# Terminate State of a Process

- A process exits a system in following stages.
  - When it reaches a natural completion point
  - When it aborts due to an unrecoverable error
  - When another process with the appropriate authority causes the process to abort.
- Termination moves the process to the exit state, and the process is no longer eligible for execution.
- The tables and other information associated with the process are temporarily preserved by the OS, for the auxiliary programs, to extract any needed information.
- Once these processes has extracted the needed information, the OS no longer needs to maintain any data relating to the process, as it is deleted from the system.

# Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

# Process Termination

- When a process terminates, its resources are deallocated by the operating system.
- However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie process**.
- All processes transition to this state when they terminate, but generally they exist as zombies only briefly.
- Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

# Process Termination

- Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphan processes**.
- Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to orphan processes.
- The init process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.
- Although most Linux systems have replaced init with systemd, the latter process can still serve the same role, although Linux also allows processes other than systemd to inherit orphan processes and manage their termination.

# Process Description

# Process Description

- The OS controls events within the computer system. It schedules and dispatches processes for execution by the processor, allocates resources to processes, and responds to requests by user processes for basic services.
- Fundamentally, we can think of the OS as that entity that manages the use of system resources by processes.

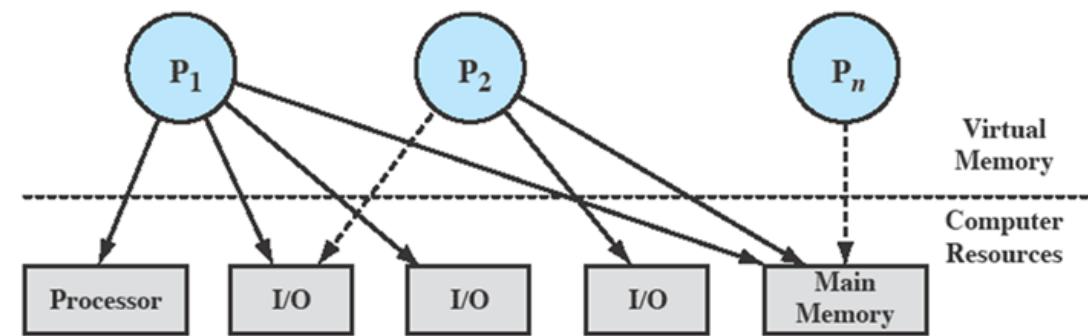


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

# Process Description

- In above figure
  - There are a number of processes ( $P_1, \dots, P_n$ ) that have been created and exist in virtual memory.
  - Each process, during the course of its execution, needs access to certain system resources, including the processor, I/O devices, and main memory.
  - In the figure, process  $P_1$  is running; at least part of the process is in main memory, and it has control of two I/O devices.
  - Process  $P_2$  is also in main memory but is blocked waiting for an I/O device allocated to  $P_1$ .
  - Process  $P_n$  has been swapped out and is therefore suspended.

# Operating System Control Structures

- For the OS to manage processes and resources, it must have information about the current status of each process and resource.
- Tables are constructed for each entity the operating system manages

# OS Control Tables

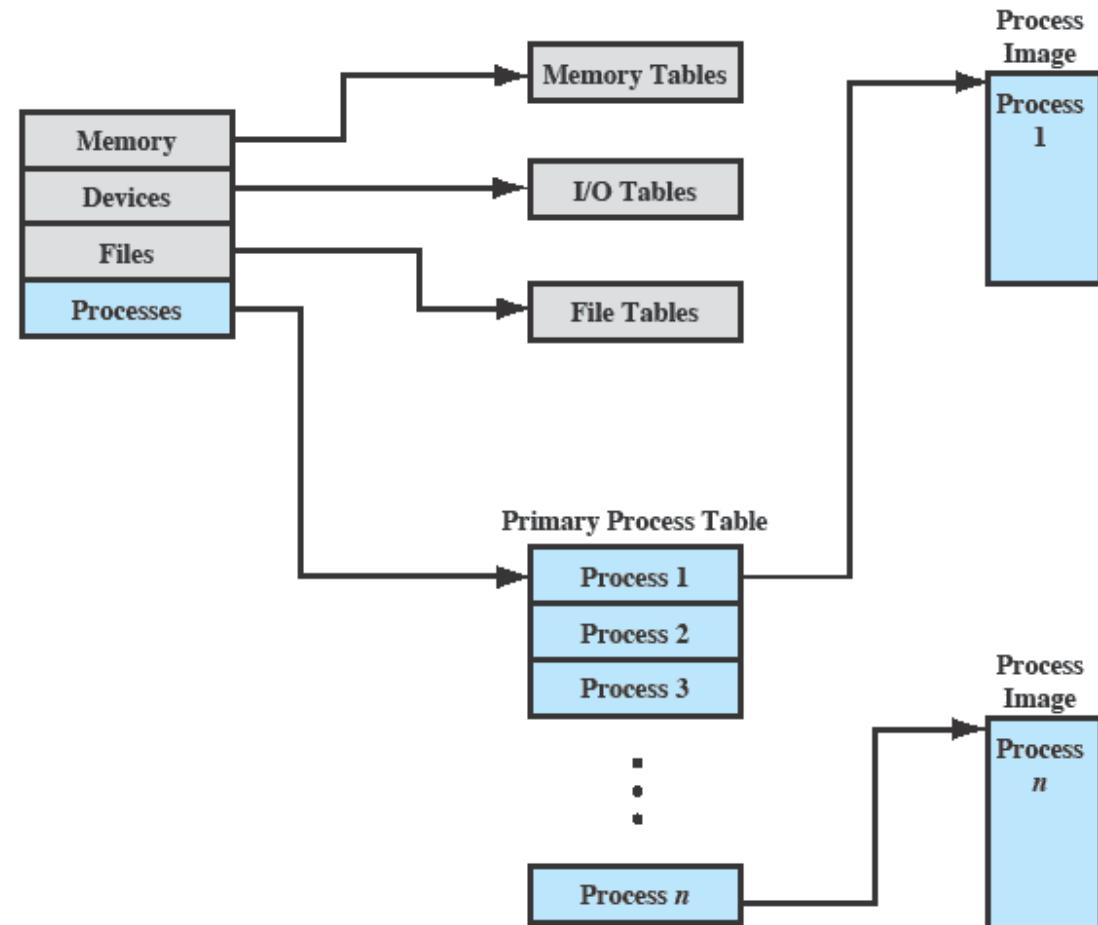


Figure 3.11 General Structure of Operating System Control Tables

# OS Control Tables

- A general idea of the scope of the tables is in Figure, which shows four different types of tables maintained by the OS:
  - memory,
  - I/O,
  - file,
  - process.
- Although the details will differ from one OS to another, fundamentally, all operating systems maintain information in these four categories.

# Memory Tables

- Memory tables are used to keep track of both main and secondary memory.
- Must include this information:
  - Allocation of main memory to processes
  - Allocation of secondary memory to processes
  - Protection attributes for access to shared memory regions
  - Information needed to manage virtual memory
- Memory tables are used to keep track of allocation of both main (real) and secondary (virtual) memory to processes.
- Some of main memory is reserved for use by the OS; the remainder is available for use by processes.
- Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism.

# I/O Tables

- Used by the OS to manage the I/O devices of the computer.
- The OS needs to know
  - Whether the I/O device is available or assigned
  - The status of I/O operation
  - The location in main memory being used as the source or destination of the I/O transfer

# File Tables

- The OS may also maintain file tables.
- These tables provide information about:
  - Existence of files
  - Location on secondary memory
  - Current Status
  - other attributes.
- Sometimes this information is maintained by a file management system, in which case the OS has little or no knowledge of files.
- In other operating systems, much of the detail of file management is managed by the OS itself.

# Process Tables

- To manage processes the OS needs to know details of the processes
  - Current state
  - Process ID
  - Location in memory
  - etc
- Process control block
  - Process image is the collection of program, Data, stack, and attributes(PCB)
- Memory, I/O, and files are managed on behalf of processes, so there must be some reference to these resources, directly or indirectly, in the process tables.
- The files referred to in the file tables are accessible via an I/O device and will, at some times, be in main or virtual memory.
- The tables themselves must be accessible by the OS and therefore are subject to memory management.

# Process Control Structures

- For the OS to manage and control a process, it must know:
  - Where the process is located
  - The attributes of the process like process id, process state...

# Process Location

- Each process has a number of attributes used by OS for process control.
- These collection of attributes referred to a Process Control Block
- The collection of program, data, stack and attributes(PCB) can be referred to as Process Image.

## Table 3.4

# Typical Elements of a Process Image

### **User Data**

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

### **User Program**

The program to be executed.

### **Stack**

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

### **Process Control Block**

Data needed by the OS to control the process (see Table 3.5).

# Process Location

- For the OS to manage process, at least a small portion of its image must be loaded into main memory or at least **virtual memory** and OS must know the location of each process.
- Therefore process tables maintained by the OS must show the location of the entire process image.
- Each entry contains at least a pointer to a process image.

# Process Attributes

- A sophisticated multiprogramming system requires a great deal of information about each process.
- Different systems will organize this information in different ways.
- For now, let us simply explore the type of information that might be of use to an OS without considering in any detail how that information is organized.
- We can group **the process control block information** into three general categories:
  - Process identification
  - Processor state information
  - Process control information

# Typical Elements of Process Control Block

## Process Identification

### Identifiers

Numeric identifiers that may be stored with the process control block include

- Identifier of this process.
- Identifier of the process that created this process (parent process).
- User identifier.

## Processor State Information

### User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

### Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include

- **Program counter:** Contains the address of the next instruction to be fetched.
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow).
- **Status information:** Includes interrupt enabled/disabled flags, execution mode.

### Stack Pointers

Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

## Process Control Information

### Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

### Data Structuring

A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

### Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

### Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

### Memory Management

This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

### Resource Ownership and Utilization

Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

# Process Identification

- Each process is assigned a unique numeric identifier.
- Many of the other tables controlled by the OS may use process identifiers to cross-referencce process tables
  - In virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table; otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier.

# Process Identification

- For example,
  - The memory tables may be organized so as to provide a map of main memory with an indication of which process is assigned to each region.
  - Similar references will appear in I/O and file tables.
  - When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication.
  - When processes are allowed to create other processes, identifiers indicate the parent and descendants of each process.
  - In addition to these process identifiers, a process may be assigned a user identifier that indicates the user responsible for the job.

# Processor State Information

- This consists of the contents of processor registers.
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program status word (PSW)
  - contains status information
  - Example: the EFLAGS register on Pentium processors

# Processor State Information

- Processor state information consists of the contents of processor registers.
- While a process is running, the information is in the registers.
- When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution. Nature and number of registers vary by processor design
- Typically, the register set will include user-visible registers, control and status registers, and stack pointers.

# Process Control Information

- This is the additional information needed by the OS to control and coordinate the various active processes.

# Structure of Process Images in Virtual Memory

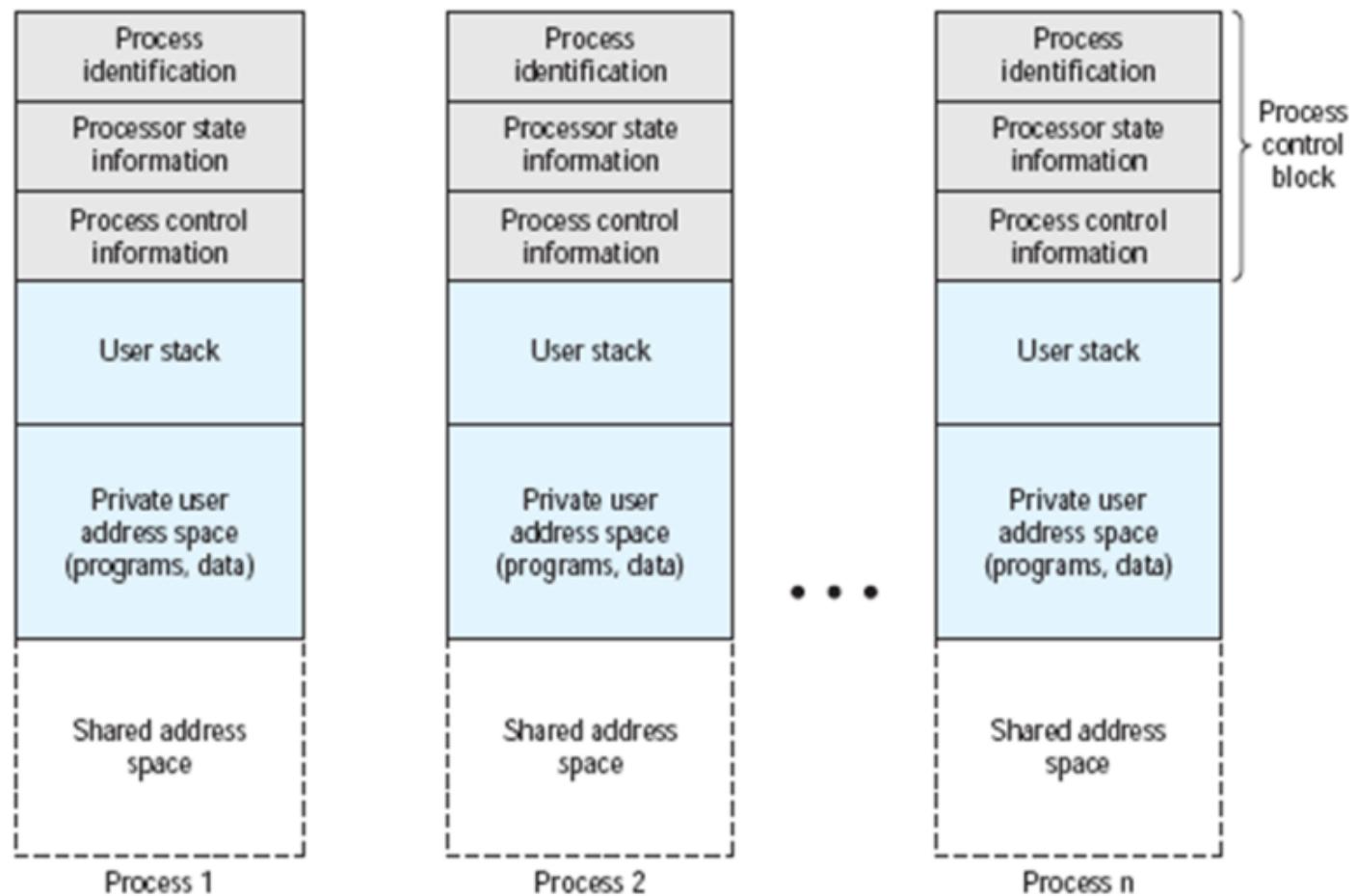


Figure 3.13 User Processes in Virtual Memory

# Structure of Process Images in Virtual Memory

- The above figure suggests the structure of process images in virtual memory.
- Each process image consists of a process control block, a user stack, the private address space of the process, and any other address space that the process shares with other processes.
- In the figure, each process image appears as a continuous range of addresses.
- In actual implementation, this may not be the case, it may depend on the memory management scheme and the way in which control structures are organized by the OS.

# Role of the Process Control Block

- The process control block is the most important data structure in an OS.
  - Each process control block contains all of the information about a process that is needed by the OS.
  - The blocks are read and/or modified by virtually every module in the OS, including those involved with scheduling, resource allocation, interrupt processing, and performance monitoring and analysis.
- Hence Process Control Block requires protection due to the below two problems:
  - A faulty routine could cause damage to the PCB destroying the OS's ability to manage the process
  - Any design change to the PCB could affect many modules of the OS

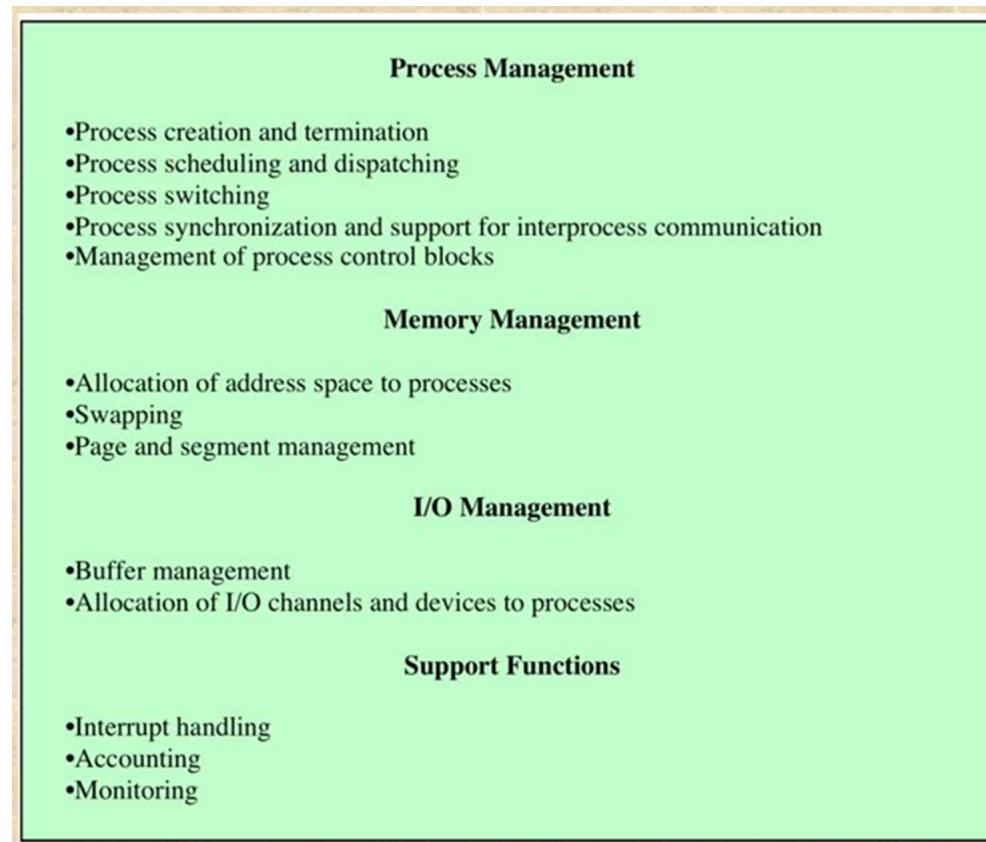
# Process Control

# Process Control – Modes of Execution

- The less-privileged mode is often referred to as the user mode, because user programs typically would execute in this mode.
- More-privileged mode is System Mode also known as
  - Control Mode
  - Kernel Mode
  - Protected Mode
- Certain instructions can only be executed in the more-privileged mode.
  - Including reading or altering a control register, such as the program status word;
  - Primitive I/O instructions;
  - Instructions that relate to memory management.
- In addition, certain regions of memory can only be accessed in the more-privileged mode.

# Process Control – Modes of Execution

- Functions typically found in the kernel of an OS



# Process Control – Modes of Execution

- How does the processor know in which mode it is to be executing? And how does it change?
  - Typically a flag (single bit) in the program status word (PSW). This bit is changed in response to certain events.
  - Typically, when a user makes a call to an operating system service or when an interrupt triggers execution of an operating system routine, the mode is set to the kernel mode and, upon return from the service to the user process, the mode is set to user mode.

# Process Control – Process Creation

- Once the OS decides to create a new process it can proceed as follows:
  - Assigns a unique process identifier
    - At this time, a new entry is added to the primary process table, which contains one entry per process
  - Allocates space for the process
    - This includes all elements of process image. The size of memory to be allocated can be assigned as default based on type of process, as per user request or the parent process can pass the needed values to the OS.
  - Initializes process control block
    - Process Identification portion
    - Processor State Information portion
    - Process Control Information portion

# Process Control – Process Creation

- Sets up appropriate linkages
  - If OS maintains each scheduling queue as linked list, then the new process must be put in the Ready or Ready/Suspend list
- Creates or expand other data structures
  - The OS may maintain an accounting file on each process to be used subsequently for billing or performance assessment purposes.

# Process/Context Switching and Mode Switching

# Process Control – Process Switching

- At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process.
- Several design issues are raised regarding process switching
  - What events trigger a process switch?
  - We must distinguish between mode switching and process switching.
  - What must the OS do to the various data structures under its control to achieve a process switch?

# When to switch processes?

- A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

# Process Control – Process Switching

- Two kinds of system interrupts,
  - one is simply called an interrupt,
  - and the other called a trap.
- “Interrupts” are due to some sort of event that is external to and independent of the currently running process, such as the completion of an I/O operation.
- With an ordinary interrupt, control is first transferred to an interrupt handler, which does some basic housekeeping and then branches to an OS routine that is concerned with the particular type of interrupt that has occurred.
- “Traps” relate to an error or exception condition generated within the currently running process, such as an illegal file access attempt.

# Process Control – Process Switching

- With traps, the OS determines if the error or exception condition is fatal.
  - If so, then the currently running process is moved to the Exit state and a process switch occurs.
  - If not, then the action of the OS will depend on the nature of the error and the design of the OS.
    - It may attempt some recovery procedure or simply notify the user.
    - It may do a process switch or resume the currently running process.
- Finally, the OS may be activated by a supervisor call from the program being executed.
  - For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open.
  - This call results in a transfer to a routine that is part of the operating system code. The use of a system call may place the user process in the Blocked state.

# Different Kinds of Interrupts

- Clock Interrupt
  - The OS determines whether the currently running process has been executing for maximum allowable unit of time referred to as **time slice**. If so, this process must be switched to a ready state and another process dispatched.
- I/O Interrupt
  - If the I/O operation completes for which one or more processes are waiting, then OS moves all those processes from blocked state to ready state. Then OS may continue with currently running process or schedule another high priority process that has been unblocked.

# Different Kinds of Interrupts

- Memory Fault
  - The processor encounters a virtual memory address reference for a word that is not in main memory. The OS must bring in the block of memory containing the reference from secondary memory to main memory.
  - After the I/O request is issued to bring in the block of memory, the process with the memory fault is placed in a blocked state; the OS then performs a process switch to resume execution of another process.
  - After the desired block is brought into memory, that process is placed in the Ready state.

# Process Control – Mode Switching

- In the interrupt stage, the processor checks for interrupt. If an interrupt is pending, the processor does the following:
  - It sets the program counter to the starting address of an interrupt handler program.
  - It switches from user to kernel mode so that the interrupt processing code may include privileged instructions.
- The processor now proceeds to fetch stage and fetches the first instruction of the interrupt handler program, which will service the interrupt.
- Now the context of the process that has been interrupted is saved into that PCB of the interrupted program.

# Process Control – Mode Switching

- What is the context of the process that gets saved when that interrupt occurs?
- It must include any information that may be altered by the execution of the interrupt handler and that will be needed to resume the program that was interrupted.
- Thus the portion of the PCB that was referred to as processor state information must be saved.
- This includes the program counter, other processor registers and stack information.

# Process Control – Mode Switching

- In most operating systems, the occurrence of an interrupt does not necessarily mean a process switch.
- It is possible that, after the interrupt handler has executed, the currently running process must resume execution.
- In that case, all that is necessary is to save the processor state information when the interrupt occurs and restore the information when control is returned to the program that was running.
- If the interrupt is to be followed by a switch to another process, then some work will need to be done.

# Process Control – Change of Process State

- A mode switch may occur without changing the state of the process that is currently in the running state.
- In that case, the context saving and subsequent restoral involves little overhead.
- If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment.

# Process Control – Change of Process State

- The steps in a process switch are:
  - Save context of processor including program counter and other registers
  - Update the process control block of the process that is currently in the Running state
  - Move process control block to appropriate queue – ready; blocked; ready/suspend
  - Select another process for execution
  - Update the process control block of the process selected
  - Update memory-management data structures
  - Restore context of the selected process
- Thus the process switch, which involves a state change, requires more effort, than a mode switch

# Process Scheduling

# Basic Concepts

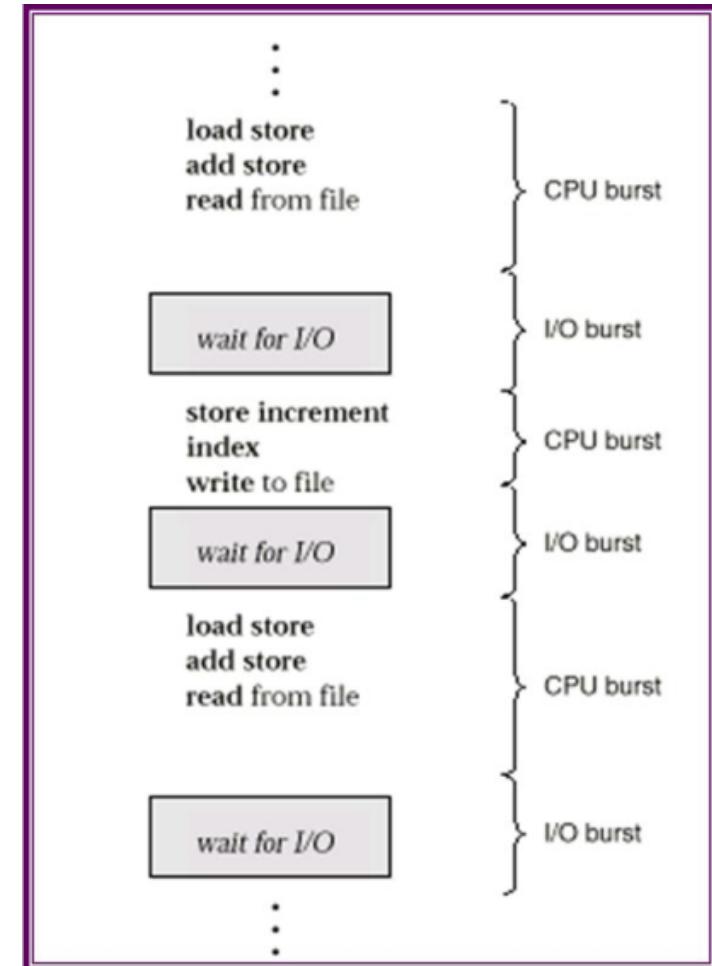
- CPU scheduling is the basis of multi-programmed operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive.
- In a system with a single CPU core, only one process can run at a time.
- Others must wait until the CPU's core is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
  - Several processes are kept in memory at one time.
  - The number of processes currently in memory is known as the **degree of multiprogramming**.

# Basic Concepts

- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU.
- On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.
- Scheduling of this kind is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.
- The CPU is, of course, one of the primary computer resources.
- Thus, its scheduling is central to operating-system design.

# CPU–I/O Burst Cycle

- Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account.
- In general, most processes can be described as either I/O bound or CPU bound.
- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.



# CPU–I/O Burst Cycle

- Processes alternate between these two states.
- Process execution begins with a CPU burst.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution
- An I/O-bound program typically has **many short CPU bursts**.
- A CPU-bound program might have a **few long CPU bursts**.
- This distribution can be important when implementing a CPU-scheduling algorithm.

# Scheduling Queues

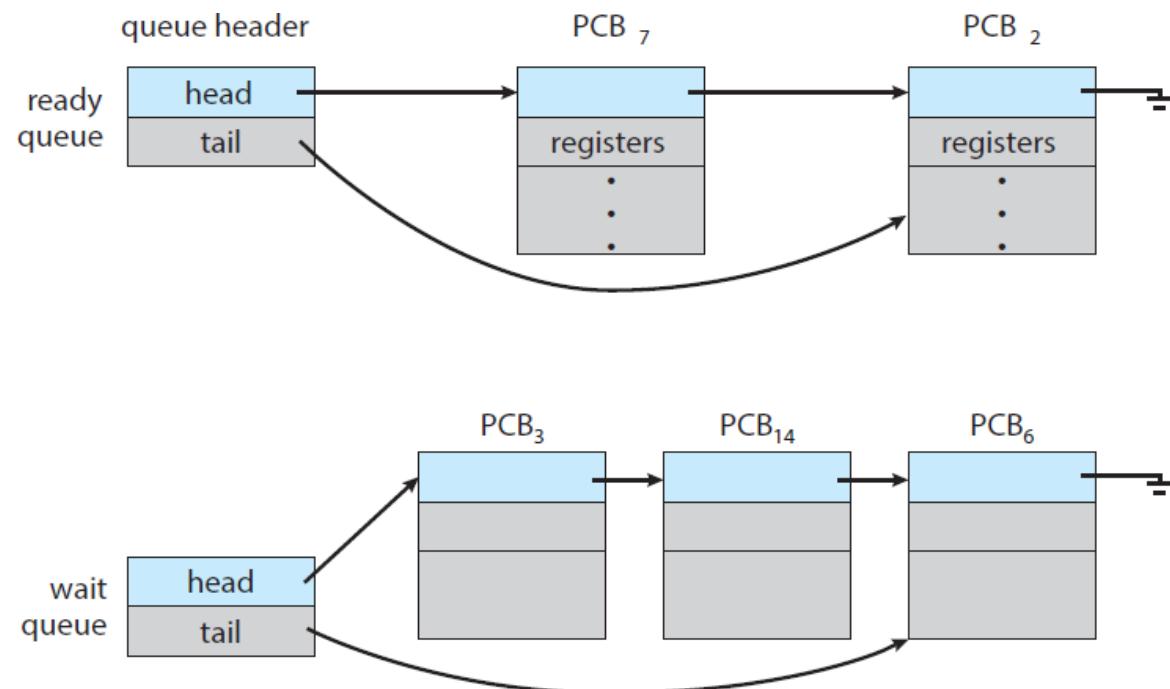
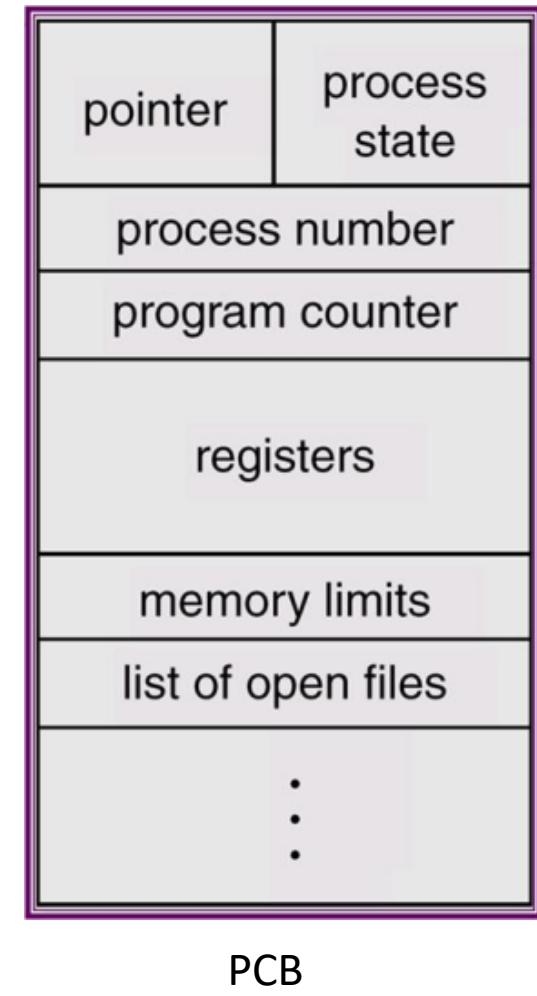


Figure 3.4 The ready queue and wait queues.



# Scheduling Queues

- As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core.
- This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.
- The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

# Scheduling Queues

- Suppose the process makes an I/O request to a device such as a disk.
- Since devices run significantly slower than processors, the process will have to wait for the I/O to become available.
- Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a wait queue

# Scheduling Queues

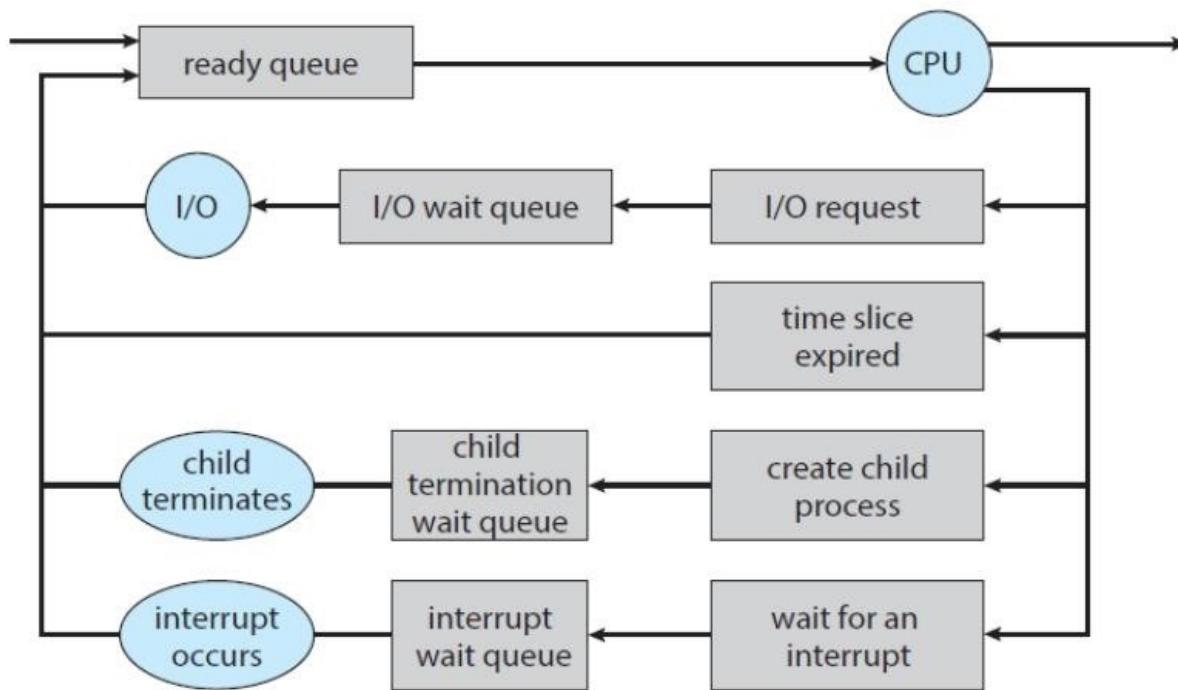


Figure 3.5 Queueing-diagram representation of process scheduling.

# Scheduling Queues

- A common representation of process scheduling is a queueing diagram, such as that in Figure 3.5.
- Two types of queues are present:
  - the ready queue and
  - a set of wait queues.
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or dispatched.

# Scheduling Queues

- Once the process is allocated a CPU core and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O wait queue.
  - The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
  - The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

# Types of Scheduling

**Table 9.1 Types of Scheduling**

<b>Long-term scheduling</b>	The decision to add to the pool of processes to be executed
<b>Medium-term scheduling</b>	The decision to add to the number of processes that are partially or fully in main memory
<b>Short-term scheduling</b>	The decision as to which available process will be executed by the processor
<b>I/O scheduling</b>	The decision as to which process's pending I/O request shall be handled by an available I/O device

# Types of Scheduling – Short-term scheduling

- Also known as the **CPU Scheduler**, decides which process to run after **clock or I/O interrupt or system calls**.
- Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- This selection process is done by the **CPU/Short-term scheduler**.
- This scheduler selects a process from the processes in memory that are ready to execute and allocates CPU to that process.
- Ready queue need not follow FIFO order(priority or random order also allowed)
- The records in the queues are generally PCBs of the processes.

# Types of Scheduling – Long-term scheduling

- Determines which programs are admitted to the system for processing
  - May be first-come-first-served
  - Or according to criteria such as priority, I/O requirements or expected execution time
- Controls the degree of multiprogramming
- More processes, smaller percentage of time each process is executed

# Types of Scheduling – Medium-term scheduling

- The **medium-term scheduler** is executed somewhat more frequently.
- Medium-term scheduling is part of the swapping function.
- Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming.
- On a system that does not use virtual memory, memory management is also an issue.
- Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

# Dispatcher

- Another component involved in the CPU-scheduling function is the dispatcher.
- The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler.
- This function involves the following:
  - Switching context from one process to another
  - Switching to user mode
  - Jumping to the proper location in the user program to resume that program
- The dispatcher should be as fast as possible, since it is invoked during every context switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**

# Preemptive and Non-preemptive Scheduling

- CPU-scheduling decisions may take place under the following four circumstances:
  1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process) – **non-preemptive**
  2. When a process switches from the running state to the ready state (for example, when an interrupt occurs) - **preemptive**
  3. When **a process switches from the waiting state to the ready state** (for example, at completion of I/O) - **preemptive**
  4. When a process terminates – **non-preemptive**

# Preemptive and Non-preemptive Scheduling

- For situations 1 and 4, there is no choice in terms of scheduling.
- A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3.
- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive or cooperative**.
- Otherwise, it is **preemptive**.
- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

# Preemptive and Non-preemptive Scheduling

- Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.
  - Consider the case of two processes that share data.
  - While one process is updating the data, it is preempted so that the second process can run.
  - The second process then tries to read the data, which are in an inconsistent state.

# Preemptive and Non-preemptive Scheduling

- Preemption also affects the design of the operating-system kernel.
  - During the processing of a system call, the kernel may be busy with an activity on behalf of a process.
  - Such activities may involve changing important kernel data (for instance, I/O queues).
  - What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure?
  - Chaos ensues.
- Solution : Process Synchronization using locks

# CPU Scheduling Algorithms

- Deals with the problem of deciding which of the process in ready queue is to be allocated the CPU
- Different scheduling algorithms
  - First Come, First Served
  - Shortest Job First
  - Priority
  - Round Robin

# Scheduling Criteria

- Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.

# Scheduling Criteria

- The criteria include the following:
  - **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
  - **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

# Scheduling Criteria

- **Turnaround time.** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. **Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.**
- **Waiting time.** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

# Scheduling Criteria

- **Response time.** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. **This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.**
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Q Consider the following set of processes with the indicated arrival time. The length of the CPU-burst time is given in ms.

Process	Arrival Time	CPU-Burst Time	Priority
P1	0	10	7
P2	1	1	2
P3	3	4	4
P4	5	2	3
P5	11	3	5

For each of the following scheduling algorithms, draw the Gantt chart and compute the avg. waiting time, avg. Response time, avg. turnaround time.

# First Come First Served (FCFS)

- The process that requests CPU first is allocated the CPU first (done using FIFO queue).
- When a process enters the ready queue, its PCB is linked to the end of the queue.
- When CPU is free, it is allocated to the process at the head of the queue.
- Then the running process is removed from the queue.

# Scheduling Criteria

- Turn Around Time = Completion time – Arrival Time
- Waiting Time = Turn Around Time – CPU Burst
- Response Time = First response – Arrival Time

# FCFS

## ❖ Gantt Chart



## ❖ Calculation

Process	Waiting Time	Response Time	Turnaround Time
P1	10 -10=0	0	10-0=10
P2	10 -1=9	10-1=9	11-1=10
P3	12-4=8	11-3=8	15-3=12
P4	12-2=10	15-5=10	17-5=12
P5	9-3=6	17-11=6	20-11=9

$$\text{Avg. W. T.} = (0+9+8+10+6)/5 = \mathbf{6.6 \text{ ms.}}$$

$$\text{Avg. R. T.} = (0+9+8+10+6)/5 = \mathbf{6.6 \text{ ms.}}$$

$$\text{Avg. T. T.} = (10+10+12+12+9)/5 = \mathbf{10.6 \text{ ms.}}$$

# First Come First Served (FCFS)

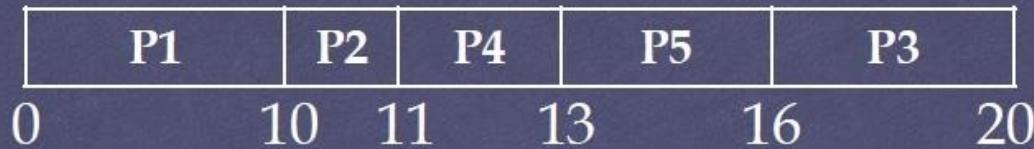
- If processes arrive in different order, like sorted based on CPU time, then average waiting time is reduced.
- FCFS is **non-preemptive** and so not used in Time Sharing Systems.
- **Convoys effect**
  - All the other processes wait for one big process to get off the CPU, results in lower CPU utilization

# Shortest Job First (SJF)

- When the CPU is available, it is assigned to the process that has smallest next CPU burst
- If next CPU burst of 2 processes same, then FCFS is followed.
- Two kinds of SJF
  - Non-preemptive SJF
  - Preemptive SJF (Shortest Remaining Time First/Next)

# Non-Preemptive SJF

## ❖ Gantt Chart



## ❖ Calculation

P	W.T.	R.T.	T.T.
P1	10 -10=0	0	10
P2	10 -1=9	10-1=9	11-1=10
P3	17-4=13	16-3=13	20-3=17
P4	8-2=6	11-5=6	13-5=8
P5	5-3=2	13-11=2	16-11=5

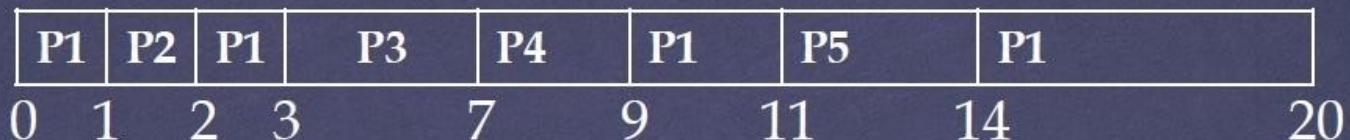
$$\text{Avg. W. T.} = (0+9+13+6+2)/5 = \mathbf{6} \text{ ms.}$$

$$\text{Avg. R. T.} = (0+9+13+6+2)/5 = \mathbf{6} \text{ ms.}$$

$$\text{Avg. T. T.} = (10+10+17+8+5)/5 = \mathbf{10} \text{ ms.}$$

# Preemptive SJF

❖ Gantt Chart



❖ Calculation

P	W.T.	R.T.	T.T.
P1	20 -10=10	0	20-0=20
P2	1-1=0	1-1=0	2-1=1
P3	4-4=0	3-3=0	7-3=4
P4	4-2=2	7-5=2	9-5=4
P5	3-3=0	11-11=0	14-11=3

$$\text{Avg. W. T.} = (10+0+0+2+0)/5 = \mathbf{2.4} \text{ ms.}$$

$$\text{Avg. R. T.} = (0+0+0+2+0)/5 = \mathbf{0.4} \text{ ms.}$$

$$\text{Avg. T. T.} = (20+1+4+4+3)/5 = \mathbf{6.4} \text{ ms.}$$

# SJF

- SJF is optimal, minimum average waiting time and decreases waiting time of short process
- But difficult to find the length of next CPU request time unless specified by user
- SJF used for long-term scheduling but not used for short-term scheduling, since cannot find next CPU burst time, only can predict.

# Priority Scheduling

- Priority associated with each process and CPU allocated to process with the highest priority
- Process with equal priority can follow FCFS order
- The larger the CPU burst, the lower the priority, usually.
- Sometimes, priority values can be modified explicitly.
- Two kinds of priority scheduling
  - Non-preemptive priority
  - Preemptive priority

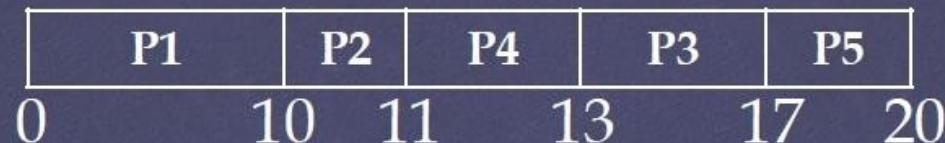
# Non-preemptive priority

- A non-preemptive priority scheduling will simply put the new process at the head of the ready queue

# Non-Preemptive Priority

☞ **Note:** Smaller value means a higher priority.

☞ Gantt Chart



☞ Calculation

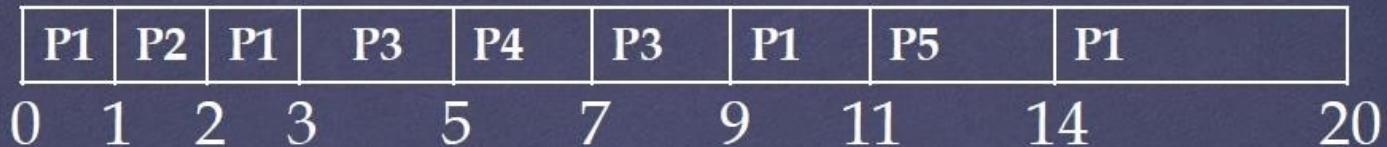
P	T.T.	W.T.	R.T.
P1	$10-0=10$	$10-10=0$	0
P2	$11-1=10$	$10-1=9$	$10-1=9$
P3	$17-3=14$	$14-4=10$	$13-3=10$
P4	$13-5=8$	$8-2=6$	$11-5=6$
P5	$20-11=9$	$9-3=6$	$17-11=6$
Avg.	10.2	6.2	6.2

# Preemptive priority

- Preemptive priority scheduling algorithm will pre-empt the CPU if the priority of the newly arrived process is higher than the priority of currently running processs

# Preemptive Priority

## ❖ Gantt Chart



## ❖ Calculation

P	T.T.	W.T.	R.T.
P1	$20-0=20$	$20-10=10$	0
P2	$2-1=1$	$1-1=0$	$1-1=0$
P3	$9-3=6$	$6-4=2$	$3-3=0$
P4	$7-5=2$	$2-2=0$	$5-5=0$
P5	$14-11=3$	$3-3=0$	$11-11=0$
Avg.	6.4	2.4	0

# Priority Scheduling Issue - Solution

- Issue – Starvation
  - Indefinite blocking/starvation of low priority processes
- Solution
  - Aging – involves gradually increasing the priority of processes that wait in the system for a long time

# Round Robin Scheduling

- Designed essentially for time sharing systems
- Similar to FCFS, but pre-emption is added to enable the system to switch between processes
- A small unit of time called a time quantum or time slice is defined (10-100ms in length)
- Ready queue treated as circular queue
- The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to 1 time quantum
- Here ready queue maintained as FIFO queue – new processes added to the tail of the ready queue

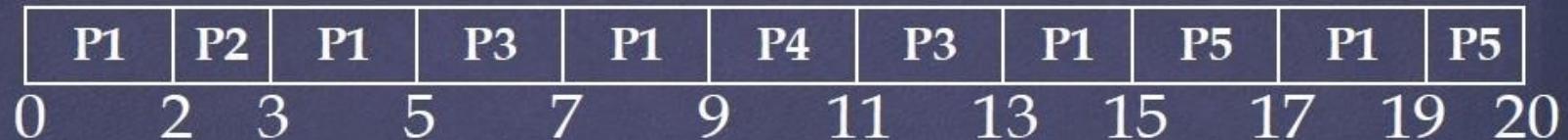
# Round Robin Scheduling

- The CPU Scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process
- The process may have a CPU burst of less than 1 time quantum
- Then the process itself will release the CPU else interrupted and context switch occurs

# RR (q=2 ms)

☞ Note: If two process need to enter the ready queue at the same time, use the FCFS to choose between them.

## ☞ Gantt Chart



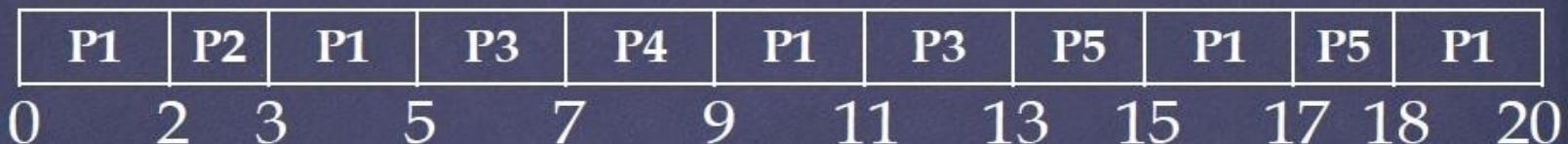
## ☞ Calculation

P	T.T.	W.T.	R.T.
P1	19-0=19	19-10=9	0
P2	3-1=2	2-1=1	2-1=1
P3	13-3=10	10-4=6	5-3=2
P4	11-5=6	6-2=4	9-5=4
P5	20-11=9	9-3=6	15-11=4
Avg.	9.2	5.2	2.2

# RR (q=2 ms)

☞ Note: If two process need to enter the ready queue at the same time, the new process will be served first.

☞ Gantt Chart



☞ Calculation

P	T.T.	W.T.	R.T.
P1	20-0=20	20-10=10	0
P2	3-1=2	2-1=1	2-1=1
P3	13-3=10	10-4=6	5 - 3=2
P4	9-5=4	4-2=2	7 - 5=2
P5	18-11=7	7-3=4	13 -11=2
Avg.	8.6	4.6	1.4

# Round Robin

- If time quantum so large, then Round robin becomes same as FCFS
- If time quantum so small, then it leads to increase in context switching
- So time quantum should not be large or small, but must be medium

# Multi-Level Queue Scheduling

- Processes are classified into different groups
  - System processes -> usually ordered by priority
  - Interactive processes or foreground processes
  - Interactive editing processes
  - Background/batch processes
  - Student processes
- They have different response time requirement, so may require different scheduling needs and foreground processes can have priority over background processes

# Multi-Level Queue Scheduling

- A multi-level queue scheduling algorithm partitions ready queue into several separate queues
- Processes are permanently assigned to one queue based on some property of the process such as memory size, priority and process time.
- Each queue has its own scheduling algorithm
  - Eg:
    - Foreground process by Round Robin
    - Background process by FCFS
- Additionally scheduling among queues based on priority.
  - Eg:
    - Foreground queue -> high priority
    - Background queue -> low priority

# Multi-Level Queue Scheduling

- Similarly below is the five scheduling queues in order of priority
  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
  - Student processes
- If interactive editing processes enters ready queue,a batch process which is running can be pre-empted
- Time slice is also focused among the queues
- 80% for foreground process with RR
- 20% for background process with FCFS

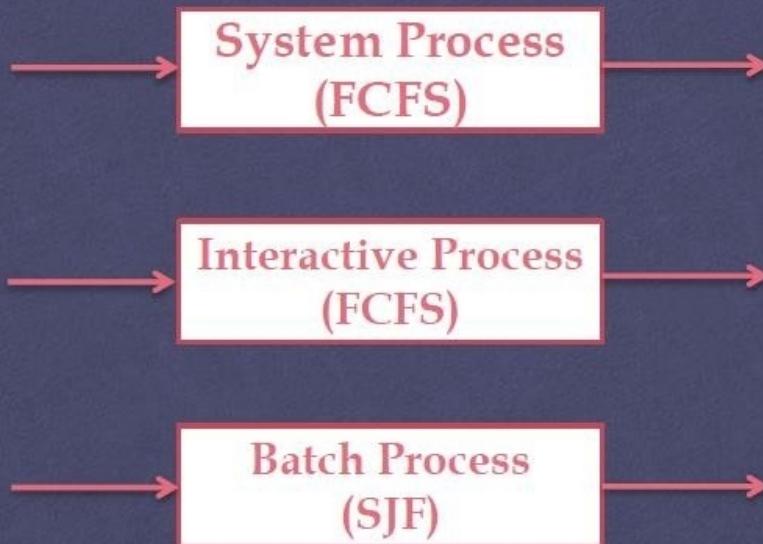
Q Consider the following set of processes with the indicated arrival time. The length of the CPU-burst time is given in ms.

Process	Arrival Time	CPU-Burst Time	Priority
P1	0	10	7
P2	1	1	2
P3	3	4	4
P4	5	2	3
P5	11	3	5

For each of the following scheduling algorithms, draw the Gantt chart and compute the avg. waiting time, avg. Response time, avg. turnaround time.

# Multi-level Queue

- ❖ Assume both of P1 and P3 are batch processes, P2 is interactive processes, and both of P4 and P5 are system processes.
- ❖ Assume also that the batch processes queue is scheduled using the preemptive SJF and the other queues use FCFS.



processes  
FCFS

Gantt chart

	P1	P2	P1	P3	P4	P3	P1	P5	P1	
	0	1	2	3	5	7	9	11	14	20
P1		TAT		WT		RT				
		$20-0=20$		$20-10=10$		$0$				
P2		$2-1=1$		$1-1=0$		$1-1=0$				
P3		$9-3=6$		$6-4=2$		$3-3=0$				
P4		$7-5=2$		$2-2=0$		$5-5=0$				
P5		$14-11=3$		$3-3=0$		$11-11=0$				
		<u><math>6-4</math></u>		<u><math>24</math></u>		<u><math>0</math></u>				

# CPU Utilization

- CPU utilization =  $(\text{CPU time} / \text{Total time}) * 100 \%$
- In the above example, CPU time = 20 (to complete process p1 to p5)
- Assume context switch time = 0.5ms
- Total time =  $(20 + (0.5 * 10)) = 25 \text{ ms}$
- CPU Utilization =  $(20/25) * 100 = 80\%$

# Multi-Level Feedback Queue Scheduling

- In multi-level queue scheduling, processes remain in same queue, not move from one queue to another.
- But in multi-level feedback queue scheduling, it allows processes to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If processes use too much CPU time, they will move to lower priority queue
- This leave I/O bound and interactive process in high priority queues.
- If process waits too long in a lower priority queue, it may be moved to a higher priority queue to prevent starvation(Aging technique).

# Multi-Level Feedback Queue Scheduling

- Multi-Level Feedback Queue Scheduling is defined by following parameters
  - Number of queues
  - Scheduling algorithms
  - Method used to determine when to upgrade a process to a higher priority queue
  - Method used to determine when to demote a process to a lower priority queue
  - Method used to determine which queue a process will enter when that process needs service.

## Multilevel Feedback Queue Scheduling.

Let us consider an example of MLFS where ready queue is divided into 3 queues  $Q_0$ ,  $Q_1$  &  $Q_2$ .

$Q_0$  has high priority than  $Q_1$  and then  $Q_2$ .

$Q_0$  and  $Q_1$  follows RoundRobin(RR) and  $Q_2$  follows FCFS. If the following shows the processes, their order of arrival and burst time, draw the Gantt chart.

	AT	BT
T1	0	40
T2	0	30
T3	0	50
T4	2	70
T5	4	25
T6	6	60
T7	7	45

$T_Q = 10$  for  $Q_0$  (RR)

$T_Q = 20$  for  $Q_1$  (RR)

$Q_2 = \text{FCFS}$

When a task enters the system, firstly it is added at the tail end of  $Q_0$  and then system allocates a fixed single Time Quantum. This scheduling algorithm provides the facility to move the tasks from one queue to another queue. If the task consumes more CPU time, the task is moved to the lower priority queue  $Q_1$  & allotted double Time Quantum.

$Q_0$	T1	T2	T3	T4	T5	T6	T7
	0	10	20	30	40	50	60

$Q_1$	T1	T2	T3	T4	T5	T6	T7
	70	90	110	130	150	165	185

$Q_2$	T1	T3	T4	T6	T7	
	205	215	235	245	305	320

# Predicting Next CPU Burst Length in SJF

- **Exponential Averaging or Aging**
- Let,  $T_n$  be the actual CPU burst time of nth process.  $T(n)$  be the predicted CPU burst time for nth process then the CPU burst time for the next process ( $n+1$ ) will be calculated as,

$$T(n+1) = \alpha * T_n + (1-\alpha) * T(n)$$

- Where,  $\alpha$  is the smoothing. Its value lies between 0 and 1.

# Next CPU Burst Length

**Q:** If the length of the next CPU burst was predicted to be 8ms, the actual length used in this prediction was 5ms. According to the average exponential formula, what was the previous predicted value of CPU burst if  $\alpha=0.4$ .

## Solution

$$t_{n+1} = 8 \text{ ms.}$$

$$t_n = 5 \text{ ms.}$$

$$t_n = ?$$

$$\alpha = 0.4$$

$$t_{n+1} = \alpha * t_n + (1 - \alpha) * t_n$$

$$8 = 0.4 * 5 + 0.6 * t_n$$

$$t_n = (8 - 2) / 0.6 = 10 \text{ ms.}$$

# Algorithm Evaluation

- How do we select a CPU scheduling algorithm for a particular system?
- There are many scheduling algorithms, each with its own parameters.
- As a result, selecting an algorithm can be difficult.
- The first problem is defining the criteria to be used in selecting an algorithm.
- Criteria are often defined in terms of CPU utilization, response time, or throughput.
- To select an algorithm, we must first define the relative importance of these measures.
- Our criteria may include several measures, such as:
  - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time.

# Algorithm Evaluation

- Once the selection criteria have been defined, we want to evaluate the algorithms under consideration.
- Various evaluation methods can be used like:
  - Deterministic modeling
  - Queuing Models
  - Simulations
  - Implementation

# Deterministic Modeling

- This evaluation method takes a predetermined workload and evaluates each algorithm using that workload.
- The advantages of deterministic modeling is that it is exact and fast to compute.
- The disadvantage is that it is only applicable to the workload that we use to test.
- Of course, the workload might be typical and scale up but generally deterministic modeling is too specific and requires too much knowledge about the workload.

# Queuing Models

- Another method of evaluating scheduling algorithms is to use queuing theory.
- Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process.
- We can now generate these times with a certain distribution.
- We can also generate arrival times for processes (arrival time distribution).
- If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.
- Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.

# Queuing Models

- One useful formula is Little's Formula.
  - $n = \lambda w$
- Where
  - $n$  is the average queue length
  - $\lambda$  is the average arrival rate for new processes (e.g. five a second)
  - $w$  is the average waiting time in the queue
- Knowing two of these values we can, obviously, calculate the third.
  - For example, if we know that eight processes arrive every second and there are normally sixteen processes in the queue we can compute that the average waiting time per process is two seconds.
  - The main disadvantage of using queuing models is that it is not always easy to define realistic distribution times and we have to make assumptions.
  - This results in the model only being an approximation of what actually happens.

# Simulations

- Rather than using queuing models we simulate a computer.
- A Variable, representing a clock is incremented. At each increment the state of the simulation is updated.
- Statistics are gathered at each clock tick so that the system performance can be analyzed.
- The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.
- Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation.
- This can often provide good results and good comparisons over a range of scheduling algorithms.
- However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

# Implementation

- The best way to compare algorithms is to implement them on real machines.
- This will give the best results but does have a number of disadvantages.
  - It is expensive as the algorithm has to be written and then implemented on real hardware.
  - If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.
  - If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

# Threads

# Overview

- A thread is a **basic unit of CPU utilization**; it comprises a thread ID, a program counter, a register set, and a stack in Thread Control Block (TCB).
- It **shares** with other threads belonging to the same process its **code section, data section**, and other operating-system resources, such as open files and signals.
- A traditional (or heavyweight) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.
- The figure below illustrates the difference between a traditional single-threaded process and a multithreaded process.

# Single and Multithreaded Processes

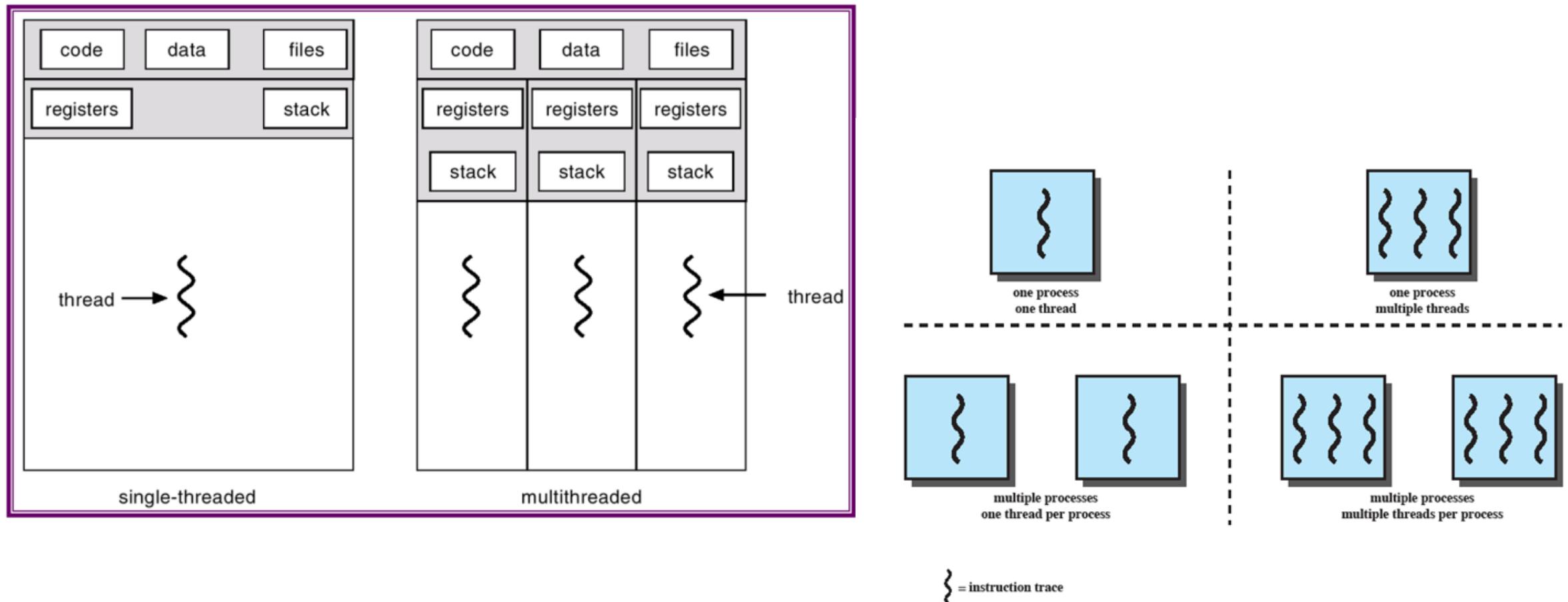


Figure 4.1 Threads and Processes [ANDE97]

# Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process - **multithreading**.
- MS-DOS supports a single user process and a single thread.
- Some UNIX, support multiple user processes but only support one thread per process
- Java run-time environment is a single process with multiple threads
- Multiple processes and threads are found in Windows, Solaris, and many modern versions of UNIX

# Motivation

- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separate process with several threads of control. For example:
  - A **web browser** might have one thread display images or text while another thread retrieves data from the network.
  - A **word processor** may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- Applications can also be designed to leverage processing capabilities on multicore systems.
- Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

# Motivation

- In certain situations, a single application may be required to perform several similar tasks.
  - For example, a **web server** accepts client requests for web pages, images, sound, and so forth.
  - A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
  - If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.
  - One solution is to have the server run as a single process that accepts requests.
  - When the server receives a request, it creates a separate process to service that request.

# Motivation

- In fact, this process-creation method was in common use before threads became popular.
- **Process creation is time consuming and resource intensive**, however.
- If the new process will perform the same tasks as the existing process, why incur all that overhead?
- It is generally more efficient to use one process that contains multiple threads.
  - If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
  - When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

# Motivation

- Finally, most operating-system kernels are now multithreaded.
- Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.
  - For example, Solaris has a set of threads in the kernel specifically for interrupt handling;
  - Linux uses a kernel thread for managing the amount of free memory in the system.

# Process vs Threads

- Process
  - A virtual address space which holds the process image
  - Protected access to
    - Processors, Other processes, Files, I/O resources
- One or More Threads in Process
  - Each thread has
    - An execution state (running, ready, etc.)
    - Saved thread context when not running
    - An execution stack
    - Some per-thread static storage for local variables
    - Access to the memory and resources of its process (all threads of a process share this)

# Single Threaded and Multithreaded process model

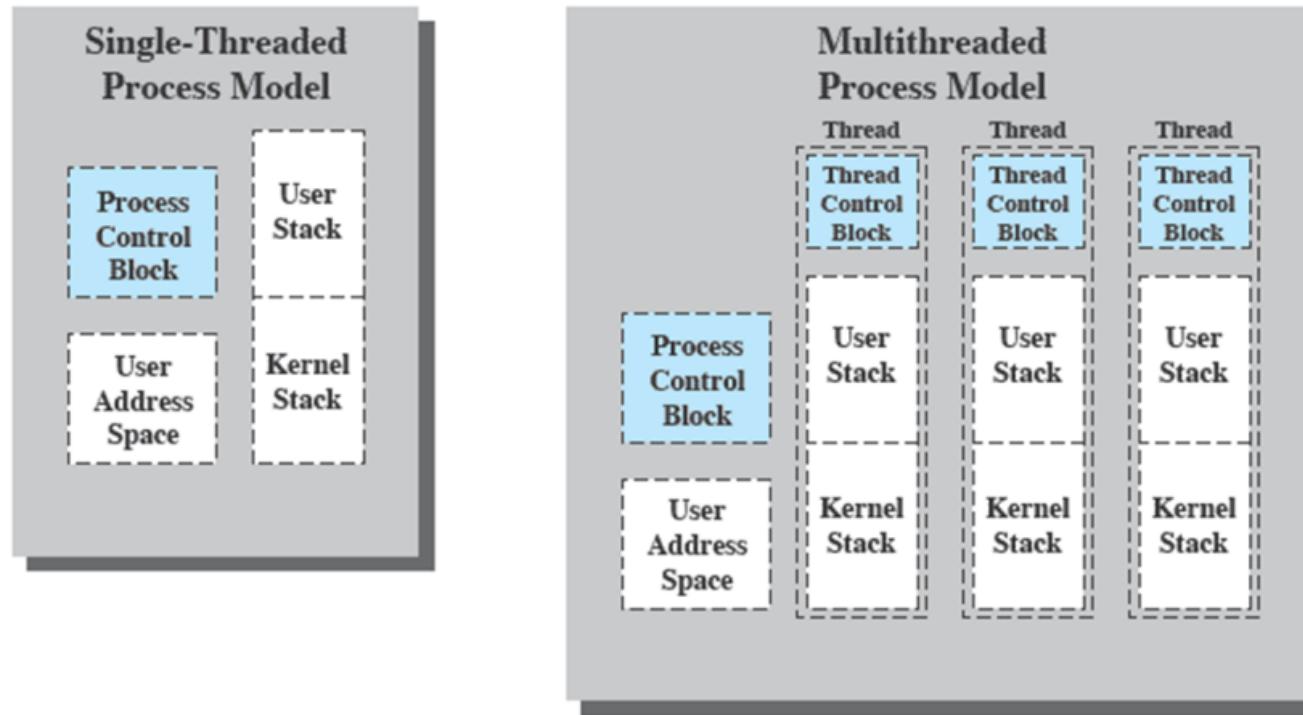


Figure 4.2 Single Threaded and Multithreaded Process Models

# Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching processes
- Threads can communicate with each other
  - without invoking the kernel
- Responsiveness.
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

# Benefits of Threads

- Resource sharing.
  - Processes can only share resources through techniques such as shared memory and message passing.
  - Such techniques must be explicitly arranged by the programmer.
  - However, threads share the memory and the resources of the process to which they belong by default.
  - The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

# Benefits of Threads

- Economy.
  - Allocating memory and resources for process creation is costly.
  - Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
  - In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
- Scalability.
  - The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.
  - A single-threaded process can run on only one processor, regardless how many are available.

# Threads

- Several actions that affect all of the threads in a process
- The OS must manage these at the process level.
  - Examples:
    - Suspending a process involves suspending all threads of the process(ULT)
    - Termination of a process, terminates all threads within the process
- Threads have execution states and may synchronize with one another.
  - Similar to processes
- We look at these two aspects of thread functionality in turn.
  - States
  - Synchronization

# Thread Execution States

- States associated with a change in thread state
  - Spawn (another thread)
  - Block
    - Issue: will blocking a thread block other, or all, threads
  - Unblock
  - Ready
  - Running
  - Finish (thread)
    - Deallocate register context and stacks

# Thread Synchronization

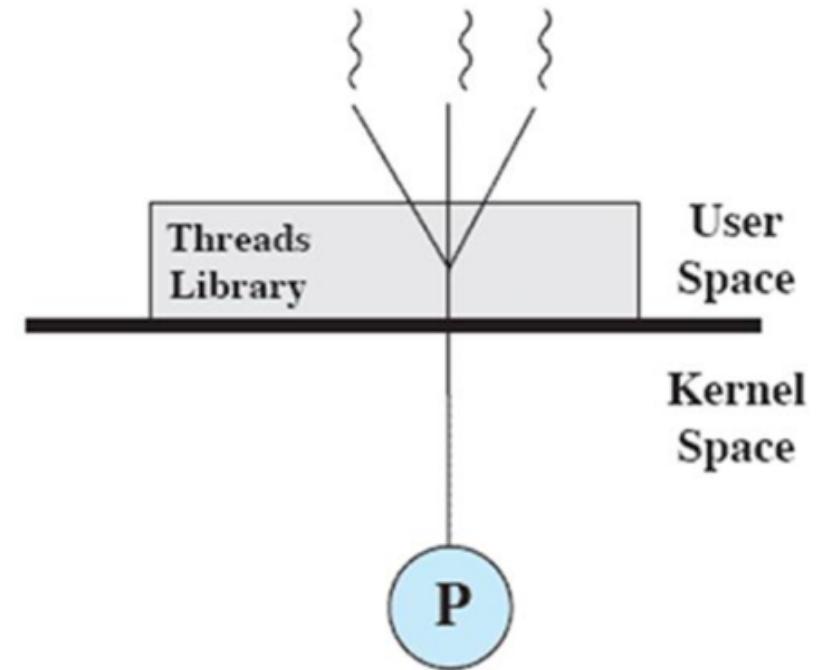
- All the threads of a process share the same address space and other resources such as open files.
- Any alteration of a resource by one thread affects the environment of the other threads in the same process
- It is therefore necessary to synchronize the activities of various threads so that they do not interfere with each other or corrupt data structures.
- Ex:
  - If two threads each try to add an element to a doubly linked list at same time, one element may be lost or the list may end up malformed.

# Types of threads

- User Level Thread (ULT)
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes

# User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Any application can be programmed to be multithreaded by using threads library
- By default, application begins with single thread of execution
- At run time application may spawn a new thread within the process using thread library



(a) Pure user-level

# User-Level Threads

- **The thread library creates a data structure for new thread** and passes control to one of the threads in ready state within the process using a scheduling algorithm.
- Whenever control passed to thread library, context of the current thread saved and when control is passed from library to a thread, the context of that thread (user registers, program counter and stack pointer) restored.
- The kernel unaware of this thread activity continues to schedule the process as a unit and assigns a single execution state to that process.

## Relationships between ULT Thread and Process States

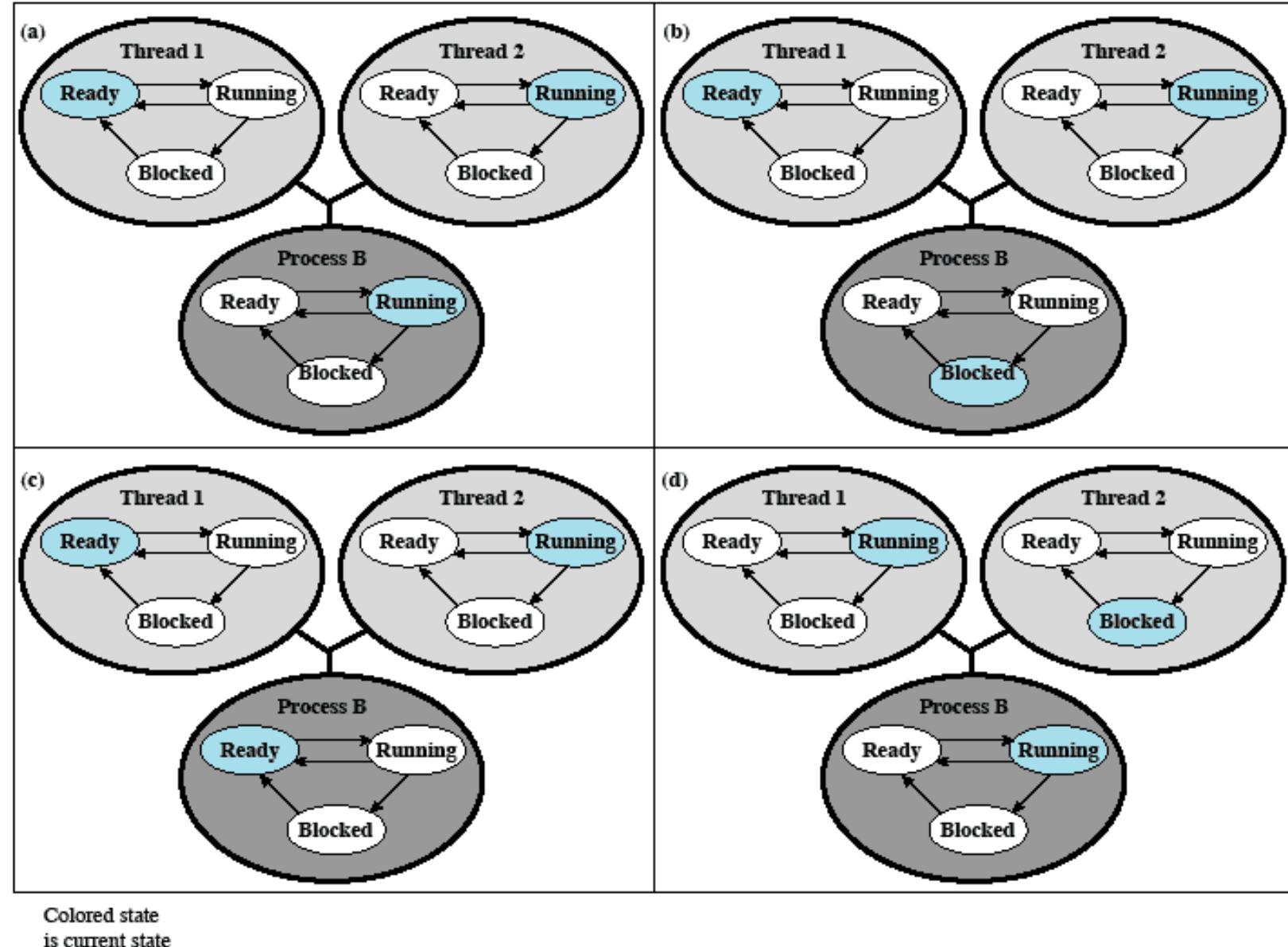


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

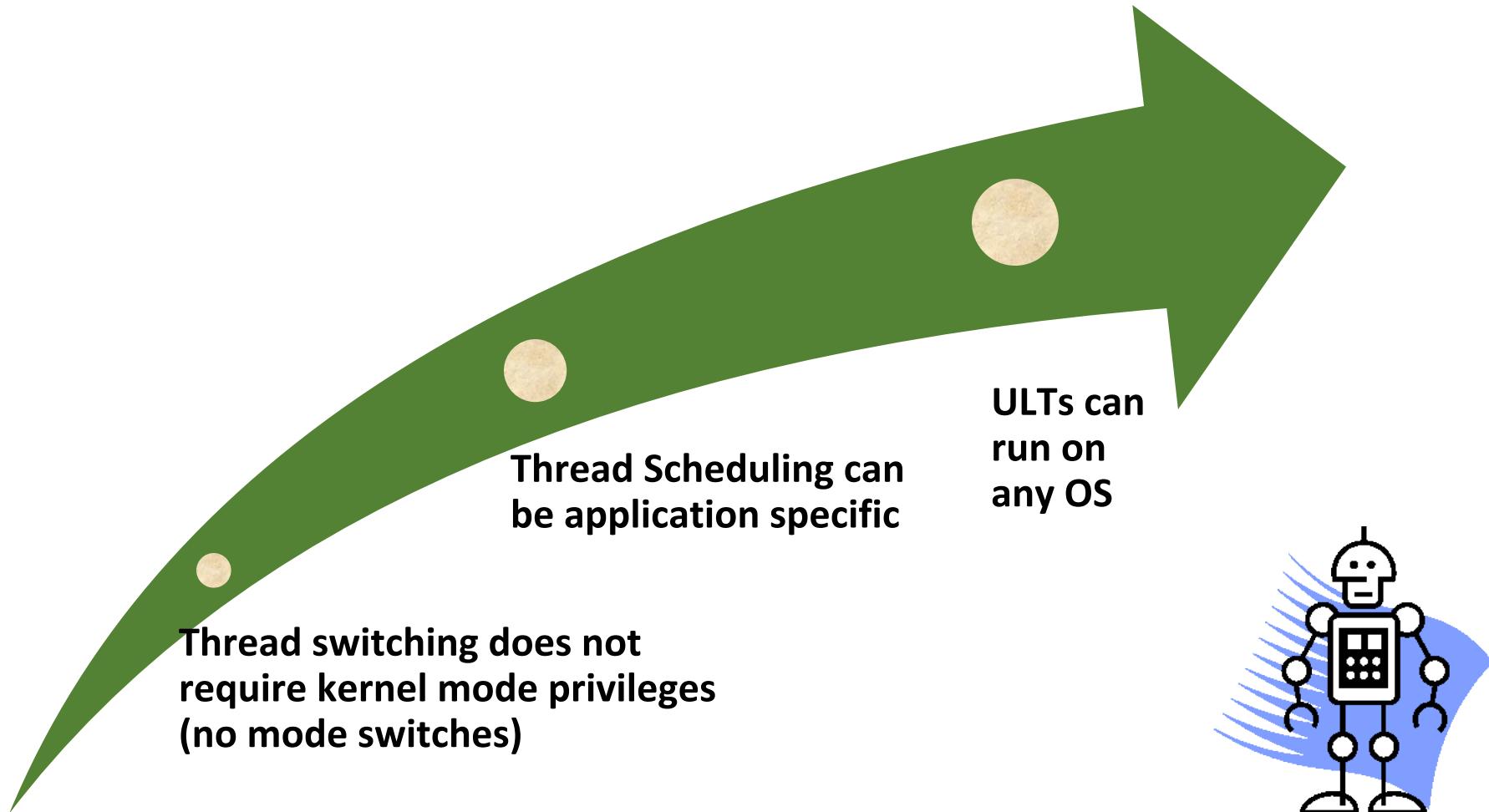
# Relationships between ULT Thread and Process States

- a) Process B executing in its thread 2; and the state of process and its two ULTs are shown in Figure 4.7(a)
- b) The application executing in thread 2 makes a system call that blocks B.
  - For eg: I/O call is made.
  - Process B now placed in blocked state.
  - So process switching happens.
  - According to data structure maintained by thread library, thread 2 of process B still in the Running state.

# Relationships between ULT Thread and Process States

- c) A clock interrupt passes control to the kernel, and the kernel determines that the currently running process has exhausted its time slice.
  - The kernel places process B in ready state and switches to another process.
  - But according to data structure maintained by thread library, thread 2 of process B is still in running state.
- d) Thread 2 has reached a point where it needs some action performed by thread 1 of process B.
  - Thread 2 enters a blocked state and thread 1 transitions from ready to running.
  - The process itself remains in the Running state.

# Advantages of ULTs



# Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing



# Overcoming ULT Disadvantages

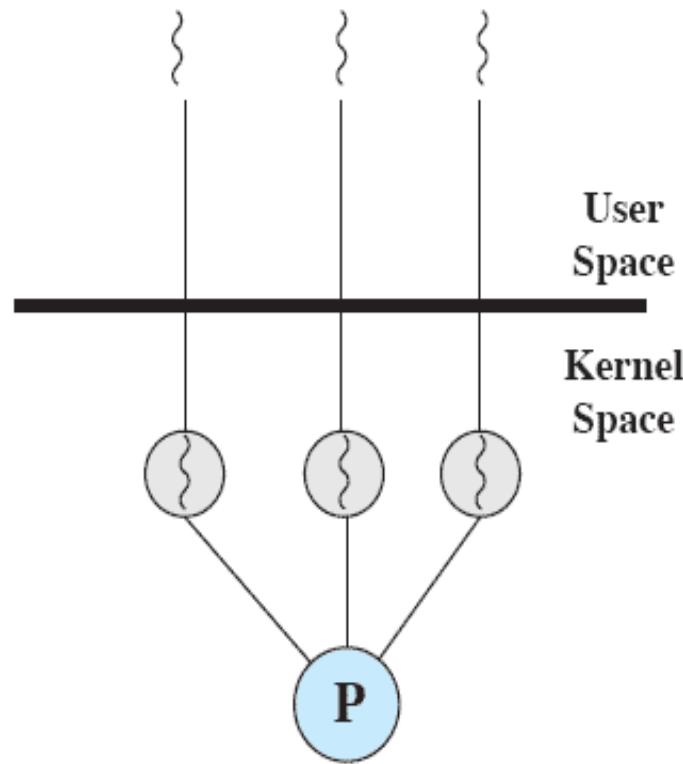
Jacketing → Application-level routine

- converts a blocking system call into a non-blocking system call (like checking for I/O device status before initiating an I/O request)



Writing an application  
as multiple processes  
rather than multiple  
threads

# Kernel-Level Threads (KLTs)

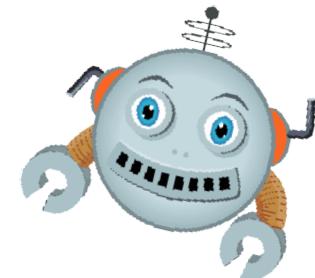


(b) Pure kernel-level

- ◆ Thread management is done by the kernel
- ◆ The kernel maintains the context information of a process in whole, as well as thread context within the process
  - ◆ no thread management is done by the application
  - ◆ Windows is an example of this approach

# Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

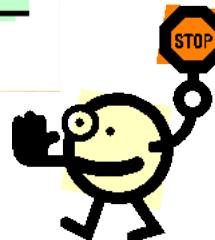


# Disadvantage of KLTs

The transfer of control from one thread to another within the same process requires a mode switch to the kernel

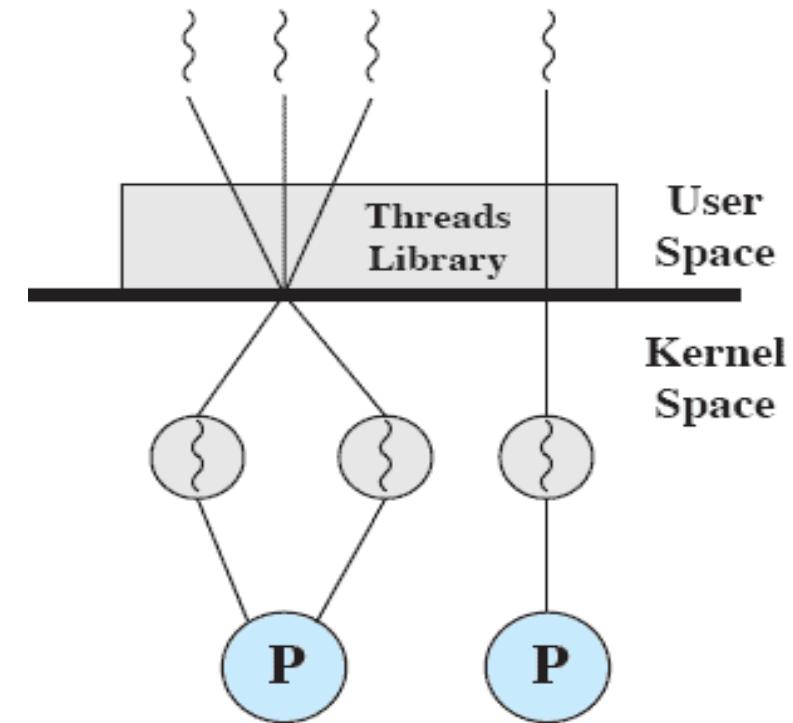
Operation	Kernel-Level		
	User-Level Threads	Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1 Thread and Process Operation Latencies ( $\mu\text{s}$ )



# Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Multiple ULTs from a single application are mapped on to some number of KLTs
- Multiple threads of same application can run in parallel on multiple processors
- Advantages of both ULT and KLT.
- Solaris is an example



(c) Combined

# Relationship Between Threads and Processes

Threads:Processes	Description	Example Systems
<b>1:1</b>	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
<b>M:1</b>	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
<b>1:M</b>	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
<b>M:N</b>	Combines attributes of M:1 and 1:M cases.	TRIX

Table 4.2 Relationship between Threads and Processes

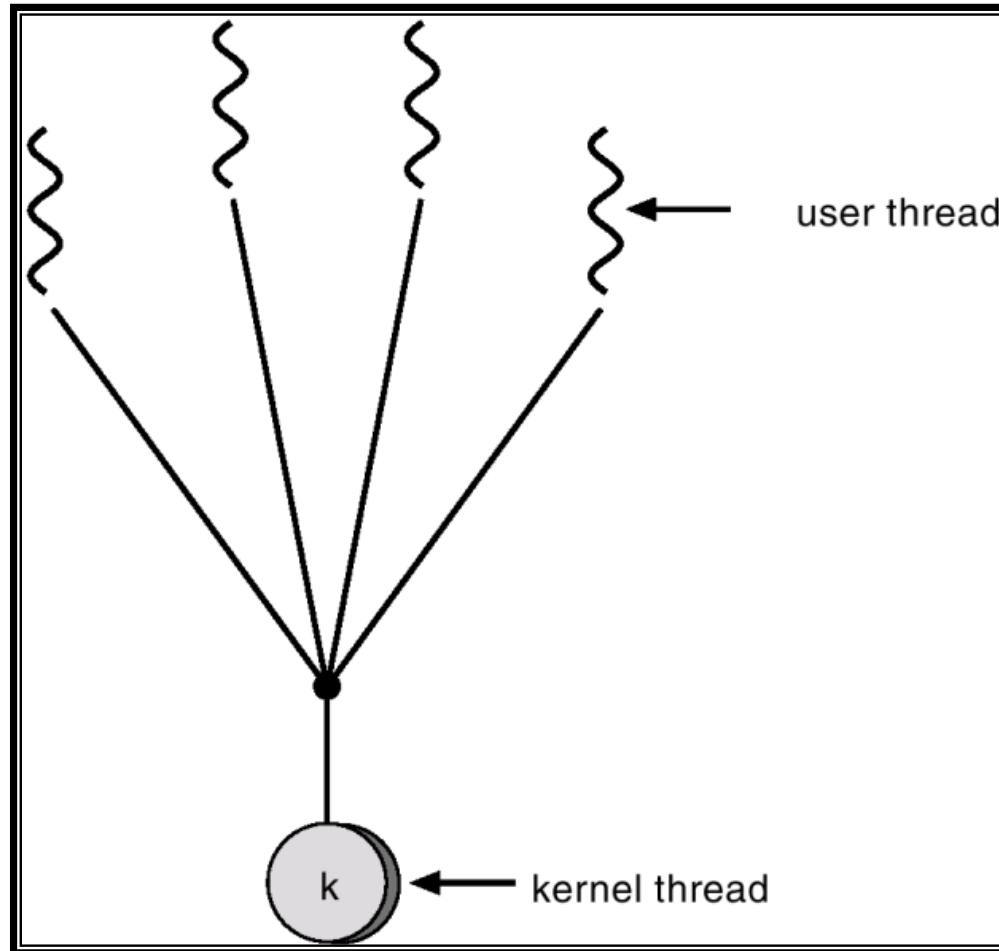
# Multithreading Models

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

# Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.
- Thread management is done by the thread library in user space, so it is efficient.
- However, the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

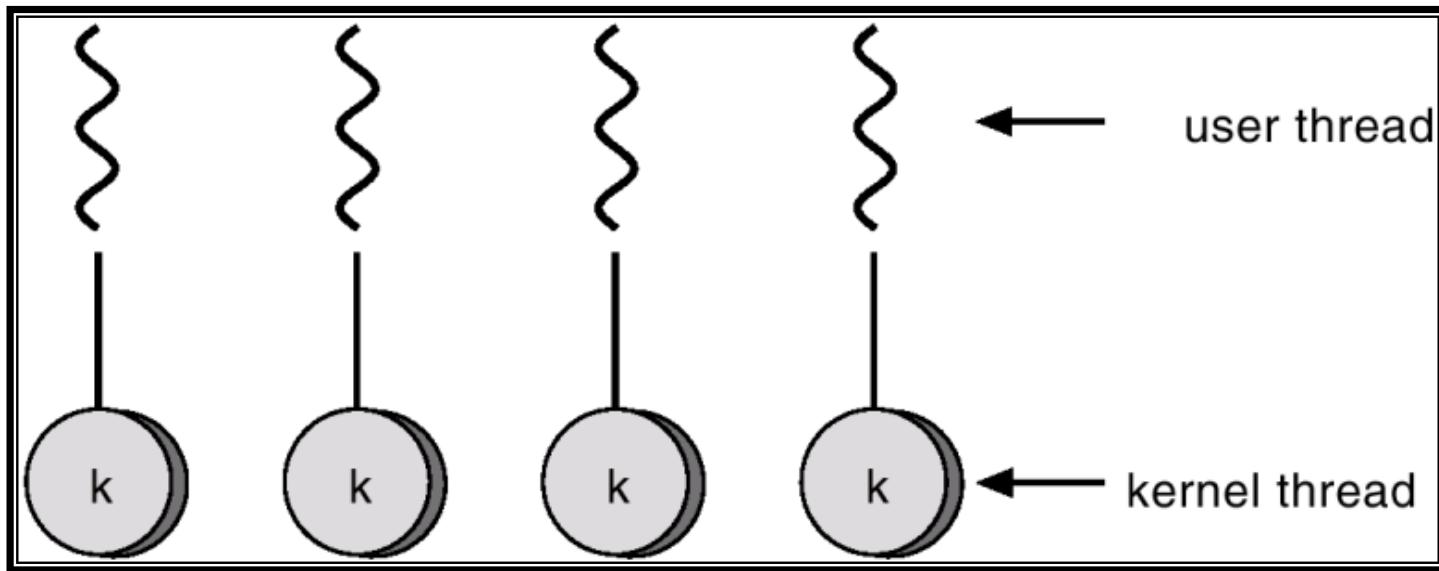
# Many-to-One Model



# One-to-One

- Each user-level thread maps to kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Examples
  - Windows 95/98/NT/2000 , OS/2

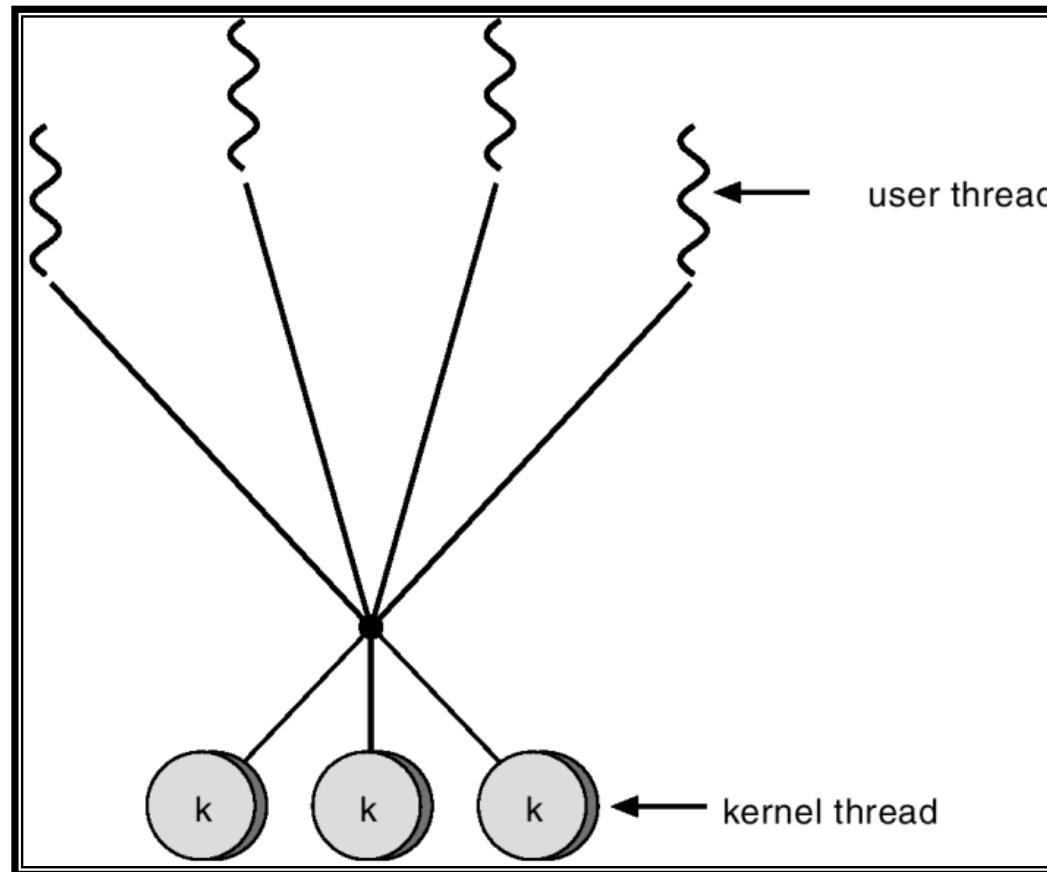
# One-to-one Model



# 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.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).
- Solaris 2
- Windows NT/2000 with the *ThreadFiber* package

# Many-to-Many Model



# Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads.
- There are two primary ways of implementing a thread library.
- The first approach is to provide a library entirely in user space with no kernel support.
- All code and data structures for the library exist in user space.
- This means that invoking a function in the library results in a local function call in user space and not a system call.

# Thread Libraries

- The second approach is to implement a kernel-level library supported directly by the operating system.
- In this case, code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel.
- Three main thread libraries are in use today:
  - POSIX Pthreads, Windows, and Java.

# Thread Libraries

- Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
- The Windows thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.
- However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
- This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX and Linux systems often use Pthreads.

## Thread Libraries – Pthreads in Linux and Unix

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

# Questions

- How many child process are created here?
- Total number of processes?

```
void main()
{
    fork();
    fork();
    fork();
}
```

# Thread Scheduling

- One distinction between user-level and kernel-level threads lies in how they are scheduled.
- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available Light Weight Process(LWP).
- This scheme is known as **process contention scope** (PCS), since competition for the CPU takes place among threads belonging to the same process.
- (When we say the thread library schedules user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.)

# Thread Scheduling

- To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (SCS).
- Competition for the CPU with SCS scheduling takes place among all threads in the system.
- Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.
- Typically, PCS is done according to priority—the thread library scheduler selects the runnable thread with the highest priority to run.
- User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.
- It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

# Pthread Scheduling

- Pthreads identifies the following contention scope values:
  - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling.
  - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling.
- On systems implementing the many-to-many model, the `PTHREAD_SCOPE_PROCESS` policy schedules user-level threads onto available LWPs.
- The number of LWPs is maintained by the thread library, perhaps using scheduler activations.
- The `PTHREAD_SCOPE_SYSTEM` scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

# Pthread Scheduling

- The Pthread IPC provides two functions for getting—and setting—the contention scope policy:
  - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
  - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- The first parameter for both functions contains a pointer to the attribute set for the thread.
- The second parameter for the `pthread_attr_setscope()` function is passed either the `PTHREAD_SCOPE_SYSTEM` or the `PTHREAD_SCOPE_PROCESS` value, indicating how the contention scope is to be set.
- In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an int value that is set to the current value of the contention scope.
- If an error occurs, each of these functions returns a nonzero value.

# Sample pthread pgm using contention scope

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }

    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

# Threading Issues

- The fork() and exec() System Calls
- Signal Handling
- Thread Cancellation
- Thread-Local Storage
- Scheduler Activations
  - One scheme for communication between the user-thread library and the kernel is known as scheduler activation.
  - It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
  - Furthermore, the kernel must inform an application about certain events.

# Inter Process Communication

# Interprocess Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
  - Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system.
  - Clearly, any process that shares data with other processes is a cooperating process.

# Reasons for Process Co-operation

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
- **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

# IPC Mechanisms

- Shared Memory
- Pipes
  - Unnamed pipes – pipe() for related process like parent, child, grand child - unidirectional
  - Named pipes – mkfifo() or mknod() for unrelated process – bidirectional communication
- Message Passing
- Signals
- Sockets

# Communication Models

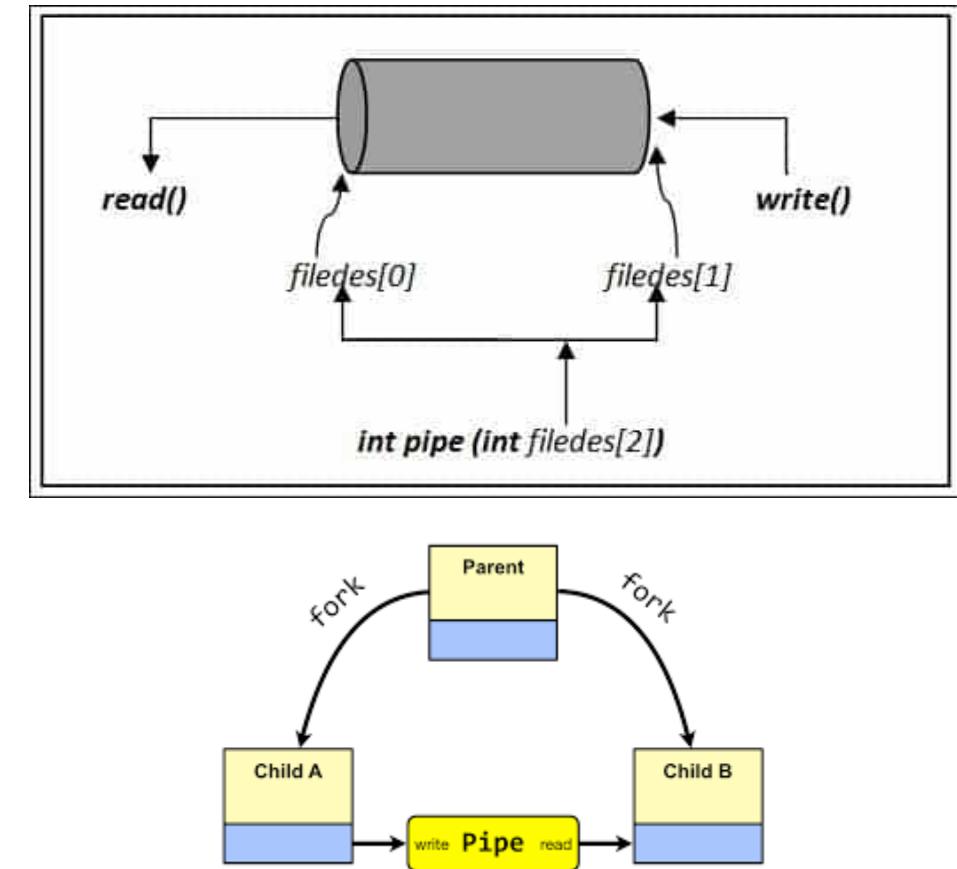
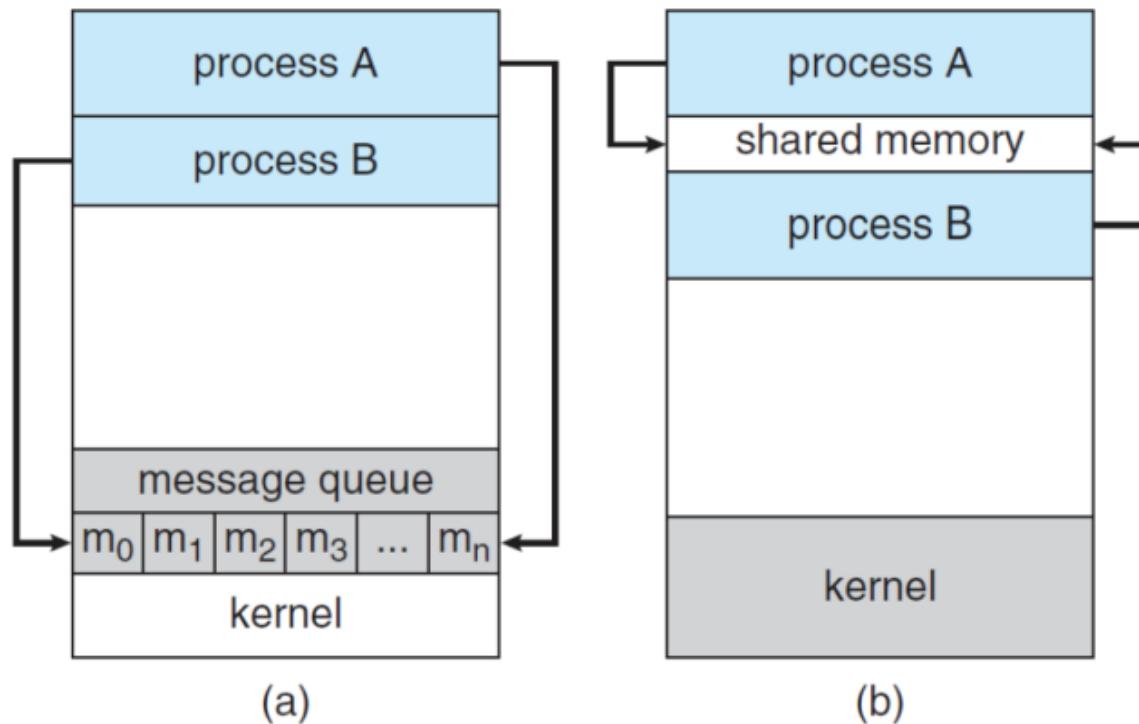


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

# Interprocess Communication

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
- Let us consider the two fundamental models of interprocess communication:
  - shared memory
  - message passing
- In the shared-memory model, a region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

# Interprocess Communication

- Message passing is useful for exchanging smaller amounts of data, because no conflicts need to be avoided.
- Message passing is also easier to implement than shared memory for inter-computer communication.
- Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.
- Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

# Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach the shared memory to their address space.

# Shared-Memory Systems

- Processes can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- To illustrate the concept of co-operating processes, let us consider the **producer-consumer problem**.

# Producer Consumer Problem

- A producer process produces information that is consumed by consumer process.
  - For example, compiler code makes assembly code consumed by an assembler. A Web server produces HTML, images, etc., and client computers consume the information through the browser.
- The producer-consumer problem can be solved by a shared memory region.
- We must have a buffer of items that is filled by a producer and the buffer emptied by the consumer.
- The buffer resides in the shared memory region.
- The producer and the consumer must be synchronized.
- While producer creates an item, the consumer consumes another item and not try to consume something not yet created.

# Producer Consumer Problem - The Buffer

- The buffer can be of two types:
  - Unbounded buffer
  - Bounded buffer
- Unbounded buffer
  - There is no limit on the size of the unbounded buffer.
  - The consumer waits for a new item, however, there is no restriction on the producer to produce items.
- Bounded buffer
  - If the buffer is empty, the consumer must wait for a new item.
  - When the buffer is full, the producer waits until it can produce new items.

# Producer Consumer Problem - The Buffer

- Let us see how bounded buffer is implemented. A bounded buffer is implemented using a circular queue of an array.

```
#define BUFFER_SIZE 10
typedef struct
{
    .....
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

# Producer Consumer Problem - The Producer

- Consider the producer and consumer code below. The buffer is empty when  $\text{in} == \text{out}$ . The buffer is full when  $((\text{in} + 1) \% \text{BUFFER\_SIZE}) == \text{out}$ ;
- **Producer**

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

# Producer Consumer Problem - The Consumer

- **Consumer**

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

- The buffer contains BUFFER\_SIZE - 1 items in the above implementation and no more. And no synchronized access to the shared buffer here.

# Message Passing

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
  - For example. An Internet chat program
- A message passing facility provides at least two operations
  - send(message)
  - receive(message)

# Message Passing

- Messages sent by processes are of fixed size or variable size.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other.
- The communication can be
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

# Naming – Message Passing

- In **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication
- Eg:
  - `send(P,message)` – send a message to process P
  - `receive(Q,message)` – receive a message from process Q
- Here
  - The link is established automatically, the processes need to know each others identity to communicate
  - A link is associated with exactly two processes
  - Between each pair of processes, there exists exactly one link

# Naming – Message Passing

- The above scheme **exhibits symmetry** in addressing; both sender and receiver must name each other to communicate.
- A variant of the above scheme is only the sender has to name the recipient.
  - `send(P, message)`
  - `receive(id, message)` – the variable `id` is set to the name of the process with which communication has taken place.

# Message Passing

- In **indirect communication**, the messages are sent to and received from mailboxes or ports.
- Each mailbox has an unique id.
  - `send(A, message)` - send a message to mailbox A
  - `receive(A, message)`- receive a message from mailbox A
- Here
  - A link is established between pair of processes only if both the members have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox

# Message Passing

- Suppose P1,P2 and P3 all share mailbox. P1 sends message to A, while both P2 and P3 execute a receive() from A.
- Which process will receive the message from P1?
- The answer depends on:
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time to execute a receive() operation
  - The system may have an algorithm to select arbitrarily which process will receive the message.

# Message Passing

- A mailbox can be owned by a process or OS.
- If a process which owns the mailbox terminates, the mailbox disappears
- Any process subsequently sends a message to that mailbox must be notified that the mailbox no longer exists.
- If mailbox owned by OS, then it is not attached to any process.
- The OS must provide mechanism that allows a process to do the following
  - Create a new mailbox
  - Send and receive messages through mailbox
  - Delete a mailbox

# Message Passing

- The process which has created the mailbox is the owner and it can receive messages through the mailbox
- But the ownership and receiving privileges can be passed on to other processes that could result in multiple receiver for each mailbox.

# Synchronization

- Communication between processes takes place through calls to send() and receive() primitives.
- Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.
  - **Blocking send:** the sending process is blocked until the message is received by the receiving process or by the mailbox.
  - **Nonblocking send:** the sending process sends the message and resumes operation.
  - **Blocking receive:** the receiver blocks until a message is available.
  - **Nonblocking receive:** the receiver retrieves either a valid message or a null.

# Synchronization

- The solution to the producer-consumer problem becomes trivial when we use blocking send() and receive() statements.
- The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox.
- Likewise, when the consumer invokes receive(), it blocks until a message is available.

# Producer Consumer - Blocking send and receive

```
message next_produced;  
  
while (true) {  
    /* produce an item in next_produced */  
  
    send(next_produced);  
}
```

Figure 3.15 The producer process using message passing.

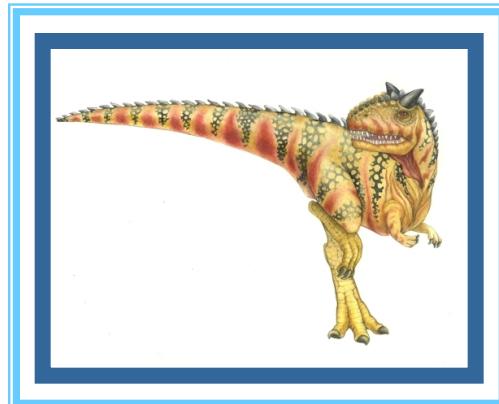
```
message next_consumed;  
  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next_consumed */  
}
```

Figure 3.16 The consumer process using message passing.

# Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in temporary queue.
- Such queues can be implemented in three ways:
- **Zero capacity**: the queue has a maximum length of zero; thus the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**: the queue has finite length  $n$ ; thus at most ' $n$ ' messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link capacity is finite. However, if the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**: the queue length is potentially infinite; thus any number of messages can wait in it. The sender never blocks.

# Chapter 6: Process Synchronization





# Chapter 6: Process Synchronization

---

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





# Objectives

---

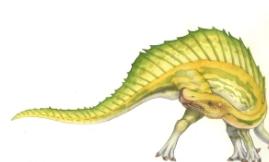
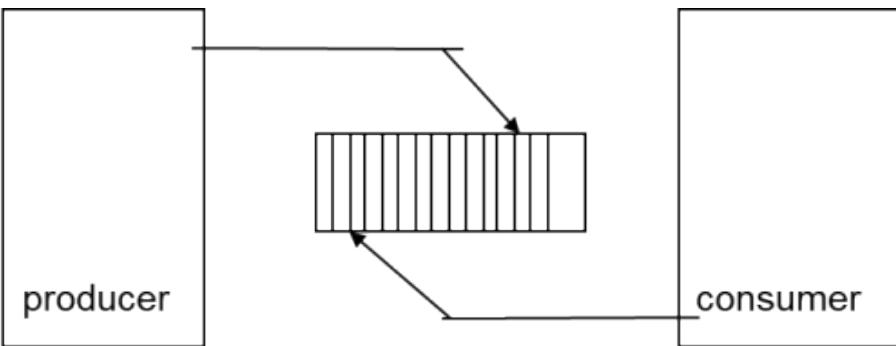
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios





# Background

- 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





# What's the problem?

---

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
- We can do so by having a shared **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 item into the buffer
  - It is decremented by the consumer after it consumes an item from the buffer
- If this is written in C++, what is the code to increment/decrement the counter?





# 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++;  
}
```





# 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

- `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 “count = 5” initially:

S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}

- When several processes/threads 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 **race condition**.





# Critical Section

---

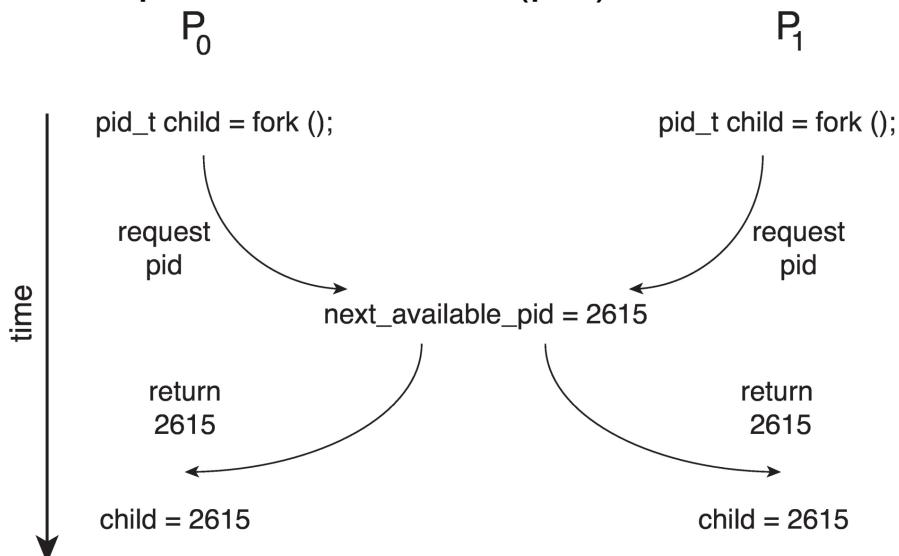
- n Any code segment that manipulates a shared variable should be part of a critical section.
- n In the producer-consumer example, counter should be incremented/decremented in a critical section.
- n A critical section is a section that must be synchronized for multiple processes or threads because lack of synchronization can result in errors.
- n The simplest case is shared variables, but other shared resources need to be synchronized as well. Often the code to allocate / return the resource is part of a critical section.





# Race Condition

- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!
- These issues led us to concentrate on process synchronization and cooperation.





# 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 table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- 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

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 only those processes that are not in their remainder sections can participate in deciding which will enter the critical section next, and this selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - ❖ Assume that each process executes at a nonzero speed
  - ❖ No assumption concerning **relative speed** of the  $n$  processes





# Solution Synonyms

---

## ■ Mutual Exclusion

- Mutual Exclusion guarantees that results are correct and not corrupted. Each process has access to the critical region and is the only process that has access at that time

## ■ Deadlock happens when one or more processes can't make progress.

- Deadlock generally involves two or more processes competing for the same resources where  $P_i$  has  $R_i$  but needs  $R_j$  and  $P_j$  has  $R_j$  but needs  $R_i$ .
- Deadlock avoidance guarantees that every process makes progress and there is no process stuck waiting on a resource held by another process that is also waiting on a resource it can't get.

## ■ Starvation is when a process never gets a chance to run. Waiting is unbounded.

- Usually, starvation is the result of policy such as priority scheduling.
- As long as every process gets a chance to run, starvation is avoided.





# Kernel-mode processes

---

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.





# Preemptive and non-preemptive kernels

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.





# 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. ↪ Hard one!!
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - ▶ Essentially free of race conditions in kernel mode





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

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

Reasons for failure:

Ensures Mutual Exclusion and bounded waiting (P0 and P1 alternate access).

If P0 takes an unduly long time in its remainder section, P1 never gets into the critical section there Progress is not made for P1.





# Algorithm for Process $P_i$

```
do {  
    flag[j] = TRUE;  
    while (flag[i]);  
        critical section  
    flag[j] = FALSE;  
        remainder section  
} while (true);
```

Reasons for failure:

Satisfies Mutual Exclusion but not bounded waiting or progress.





# Algorithm for Process $P_i$

```
do {  
    while (flag[i]);  
        flag[j] = TRUE;  
            critical section  
        flag[j] = FALSE;  
            remainder section  
    } while (true);
```

Reasons for failure:

Satisfies progress but not bounded waiting. Can deadlock.





# Peterson's Solution for CS Problem

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process  $P_i$  is ready!
- NEED BOTH THE **turn** and **flag[2]** to guarantee Mutual Exclusion, Bounded waiting, Progress.
- Initially **flag[0]** and **flag[1]=false**





# Peterson's Algorithm

## Process $P_i$

```
do {  
    flag[i] = true; // intent  
    turn = j; // giving away  
    while (flag[j] && turn==j)  
        ; // wait for either  
        critical section  
    flag[i] = false;  
    remainder section  
} while (true);
```

## Process $P_j$

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





# Algorithm for Process $P_i$

```
do {
    flag[i] = true; // i is ready
    turn = j; // Give  $P_j$  a chance first
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
    remainder section
} while (true);
```

Peterson's says,  $P_i$  is ready to enter critical section. Let  $P_j$  go first.

If  $P_j$  is waiting, then  $\text{flag}[j] == \text{true}$  but  $\text{turn} != i$  because  $\text{turn}$  was just set to  $j$ .. therefore,  $P_j$  proceeds out of loop to critical section and  $P_i$  is in the loop.

if  $P_j$  is not waiting, then  $\text{flag}[j] == \text{false}$ , so  $P_i$  can proceed into the critical section.

if  $P_j$  is in the critical section, then  $P_i$  waits because  $\text{flag}[j] == \text{true}$  ( $P_j$  set it) and  $\text{turn} == j$  ( $P_i$  set it). When  $P_j$  is done,  $\text{flag}[j] == \text{false}$ , so  $P_i$  can proceed.





# Peterson's Solution

- Here, mutual exclusion, progress and bounded waiting are all satisfied.
- But this solution works for only two processes.
- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures with multiple cores.





# Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions mostly based on idea of **locking**
  - Protecting critical regions via locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - ▶ Time consuming, since message must be passed to all processors that delays entry to critical section.





# Hardware Instructions

- Modern machines provide special atomic hardware instructions
  - ▶ **Atomic** = non-interruptible
- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptedly.)
- **Test-and-Set** instruction
- **Swap** instruction





# 1. 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”.
4. When target == FALSE, target is set to TRUE but FALSE is returned, so loop is exited.
5. When target == TRUE, target is set to TRUE, but TRUE is returned, so continue to loop.





# Solution using test\_and\_set()

---

- n Shared Boolean variable lock, initialized to FALSE
- n Solution:

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

Think of `test_and_set` as an atomic version of `i++` except it can only ever be 0 or 1. When leaving the critical section, lock is reset.





# Test and Set Instruction

---

- Mutual exclusion is preserved:
- if  $P_i$  enters the CS, the other  $P_j$  are busy waiting
- • Problem: still using busy waiting
- • When  $P_i$  exits CS, the selection of the  $P_j$  who will enter CS is arbitrary: no bounded waiting. Hence starvation is possible





## 2. Swap Instruction

---

Definition:

```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

Executed atomically





# Solution using Swap

---

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    key = true;  
    while (key == true)  
        Swap(lock,key);  
    critical section  
    lock = false;  
    remainder section  
}
```





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

```
boolean waiting[n];  
  
boolean lock; //INITIALLY FALSE  
  
do {  
    waiting[i] = true; // i wants to enter CS  
    key = true; // assume we are locked out  
    while (waiting[i] && key) // busy wait for lock  
        key = test_and_set(&lock);  
    waiting[i] = false; // i no longer waiting, in CS  
  
    /* critical section */  
  
    j = (i + 1) % n; // select next process j  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n; // does this j want to be next  
    if (j == i) // no other process waiting  
        lock = false; // unlock  
    else // otherwise, give j the lock  
        waiting[j] = false; // let j in CS  
  
    /* remainder section */  
} while (true);
```

Entry

Exit





# Machine Instruction Solution

## ■ Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

## ■ Disadvantages

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible if a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region





# Mutex Locks

---

- n Previous solutions are complicated and generally inaccessible to application programmers
- n OS designers build software tools to solve critical section problem – OS ensures it is atomic.
- n Simplest is **mutex** lock – MUTUAL EXCLUSION LOCK
  - n **Boolean variable** indicating if critical section is available or not
- n Protect a critical section by first **acquire()** a lock then **release()** the lock
- n Calls to **acquire()** and **release()** must be atomic
  - | Usually implemented via hardware atomic instructions
- n But this solution requires **busy waiting**
  - n 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**
- Apart from initialization the semaphore can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()** or **sem\_wait()** and **sem\_post()**
    - ▶ Originally called **P()** and **V()**

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

```
wait(S) {  
    while (S <= 0) // 0 means first waiting, <0 means queue  
        ; // busy wait  
    S--;  
}
```

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

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





# Semaphore Usage – Binary Semaphore

- **Binary semaphore** – integer value can range only between 0 and 1
- Same as a **mutex lock**, provide mutual exclusion and can be used to deal with critical section problem for multiple processes for a single resource.
- The ‘n’ processes share a semaphore **mutex** initialized to 1.
- Each process  $P_i$  organized as shown.

```
do
{
    wait(mutex);
        //critical section
    signal(mutex);
        //remainder section
}while(TRUE);
```





# Semaphore Usage – Counting semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- This semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore thereby decrementing the count.
- When the count for semaphore goes to 0, all resources are being used.
- After that processes that wish to use a resource will block until the count becomes greater than 0.





# Semaphore Usage

- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “synch” initialized to 0

P1 :

`s1;` If P1 gets there first, no problem.  
`signal(synch);` If P2 gets there first, wait for synch.

P2 :

`wait(synch);` Process P2 will busy wait for P1.  
`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
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution





# Semaphore Implementation with no Busy waiting

---

- With each semaphore there is an associated waiting queue
- 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) { // Process has to wait  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
  
    if (S->value <= 0) { // Processes are waiting  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```





# 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

$$\begin{array}{l} P_0 \\ \text{wait}(S); \\ \text{wait}(Q); \\ \dots \\ \text{signal}(S); \\ \text{signal}(Q); \end{array}$$
$$\begin{array}{l} P_1 \\ \text{wait}(Q); \\ \text{wait}(S); \\ \dots \\ \text{signal}(Q); \\ \text{signal}(S); \end{array}$$

- **Starvation – indefinite blocking**

- A process may never be removed from the semaphore queue in which it is suspended





# Priority Inversion

- Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Assume that 3 processes L, M and H with priority order  $L < M < H$  arrives for execution in order L, M and H with some time interval between them.
  - L uses Resource R1 - So R1 locked
  - H arrives, L is preempted and H executes
  - Now, H needs Resource R1 - But R1 locked so H goes for waiting and L continues.
  - Now M comes for execution. L preempted since H has high priority than L, so waiting time of H increased. **This is priority inversion problem.**
- Solved via **priority-inheritance protocol**
  - All processes that are accessing resources needed by higher priority process inherit higher priority until they are finished using the resource, then priority restored.





# 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  $n$



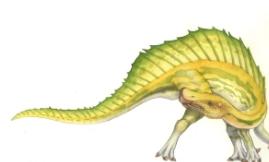


# Bounded Buffer Problem (Cont.)

---

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced, see slide 6 */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```





# Bounded Buffer Problem (Cont.)

- n The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    /* see slide 7 */  
    ...  
    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 single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1 controls access to write.
  - Semaphore `mutex` initialized to 1 controls access to `read_count`
  - Integer `read_count` initialized to 0, keeps track of readers.





# 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.)

- n The structure of a reader process

```
do {
    wait(mutex);           // Only 1 can access read_count
    read_count++;
    if (read_count == 1)
        wait(rw_mutex); // First reader blocks writer.
    signal(mutex);

    ...
    /* reading is performed */

    ...

    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex); // No reader, unblock writer
    signal(mutex);
} while (true);
```





# Readers-Writers Problem Variations

---

- **First** variation – no reader 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





## 2nd Reader-Writers, Writers Priority

```
do { // Reader process, extra mutex
    wait(read_mutex);
    wait(mutex);           // Only 1 can access read_count
    read_count++;
    if (read_count == 1)
        wait(rw_mutex); // First reader blocks writer.
    signal(mutex);
    signal(read_mutex);

    ...
    /* reading is performed */

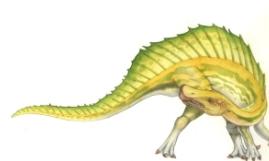
    ...

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex); // No reader, unblock writer
    signal(mutex);
} while (true);
```





```
do { // Writer process similar to reader, also uses extra mutex
    wait(mutex);           // Only 1 can access write_count
    write_count++;
    if (write_count == 1)
        wait(read_mutex); // First writer blocks readers.
    signal(mutex);
    wait(rw_mutex);      // A writer has access
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
    wait(mutex);
    write_count--;
    if (write_count == 0)
        signal(read_mutex); // No writer, unblock writer
    signal(mutex);
} while (true);
```





# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- 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? Deadlock





## 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

---

- n Semaphores are complicated and hard to get right.
  - | Extremely error prone!
  
- n Incorrect use of semaphore operations:
  - | signal (mutex) .... wait (mutex)
  
  - | wait (mutex) ... wait (mutex)
  
  - | Omitting of wait (mutex) or signal (mutex) (or both)
  
- n Deadlock and starvation are possible.





# Monitors

---

- A programming language construct that controls access to shared objects
  - Synchronization code added by compiler, enforced at runtime.
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures
    - ▶ That operate on shared data structures
  - Synchronization
    - ▶ Used for concurrent procedure invocations
- A monitors protects its data from unstructured access
- It guarantees that threads access its data through its procedures





# Monitors

---

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization using mutual exclusion.
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time.
  - Great for ensuring mutual exclusion.
  - Monitor keeps track of waiting threads/processes using wait queue.
  - If a thread within a monitor blocks, another one can enter.
- But not powerful enough to model some synchronization schemes

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

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

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





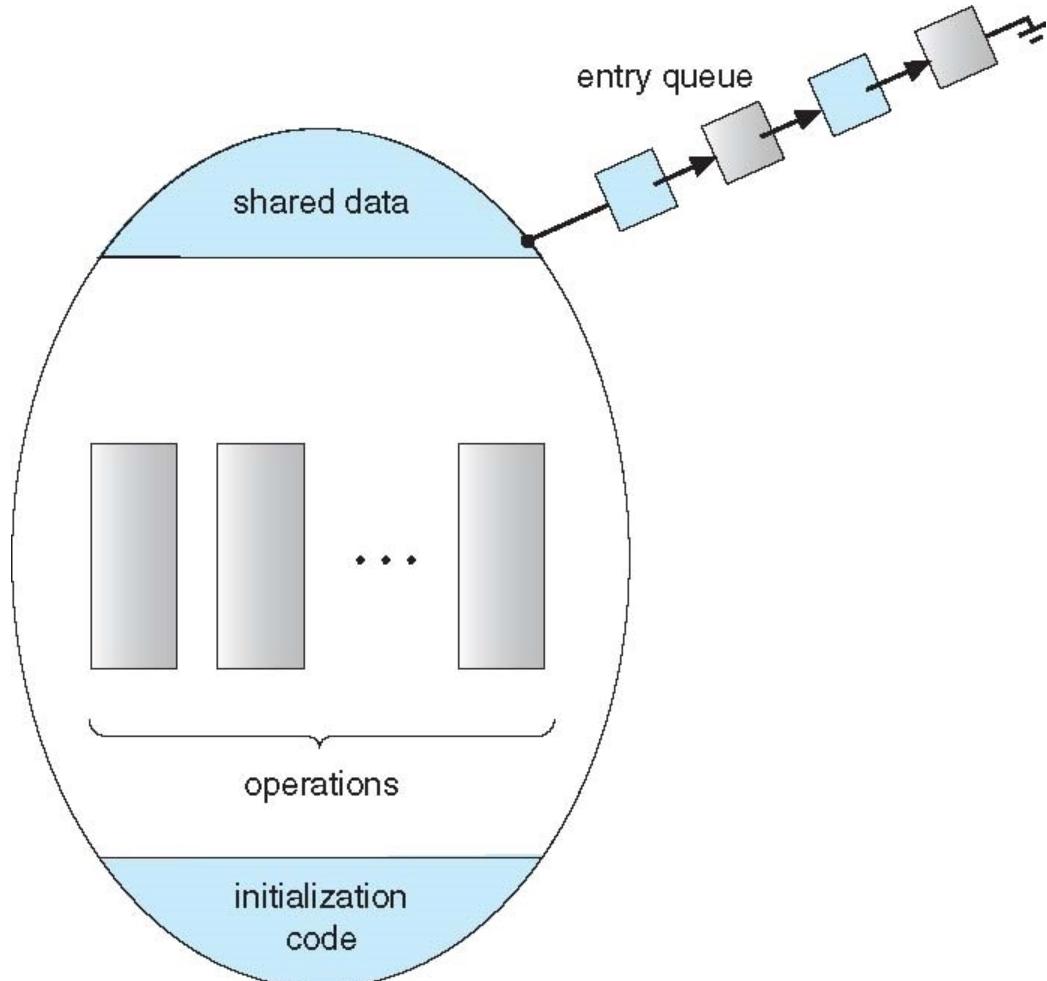
# Monitors – Abstract Data Type

- n Has an initialization section which runs once.
- n Shared data that is often the reason for synchronization.
- n One or more operations that access the shared data.
- n A queue of waiting tasks (threads/processes).
- n A simple monitor cannot control inter-task dependencies such as
  - | T1 must perform some operation which is required before T2 can proceed.





# Schematic view of a Monitor





# Condition Variables

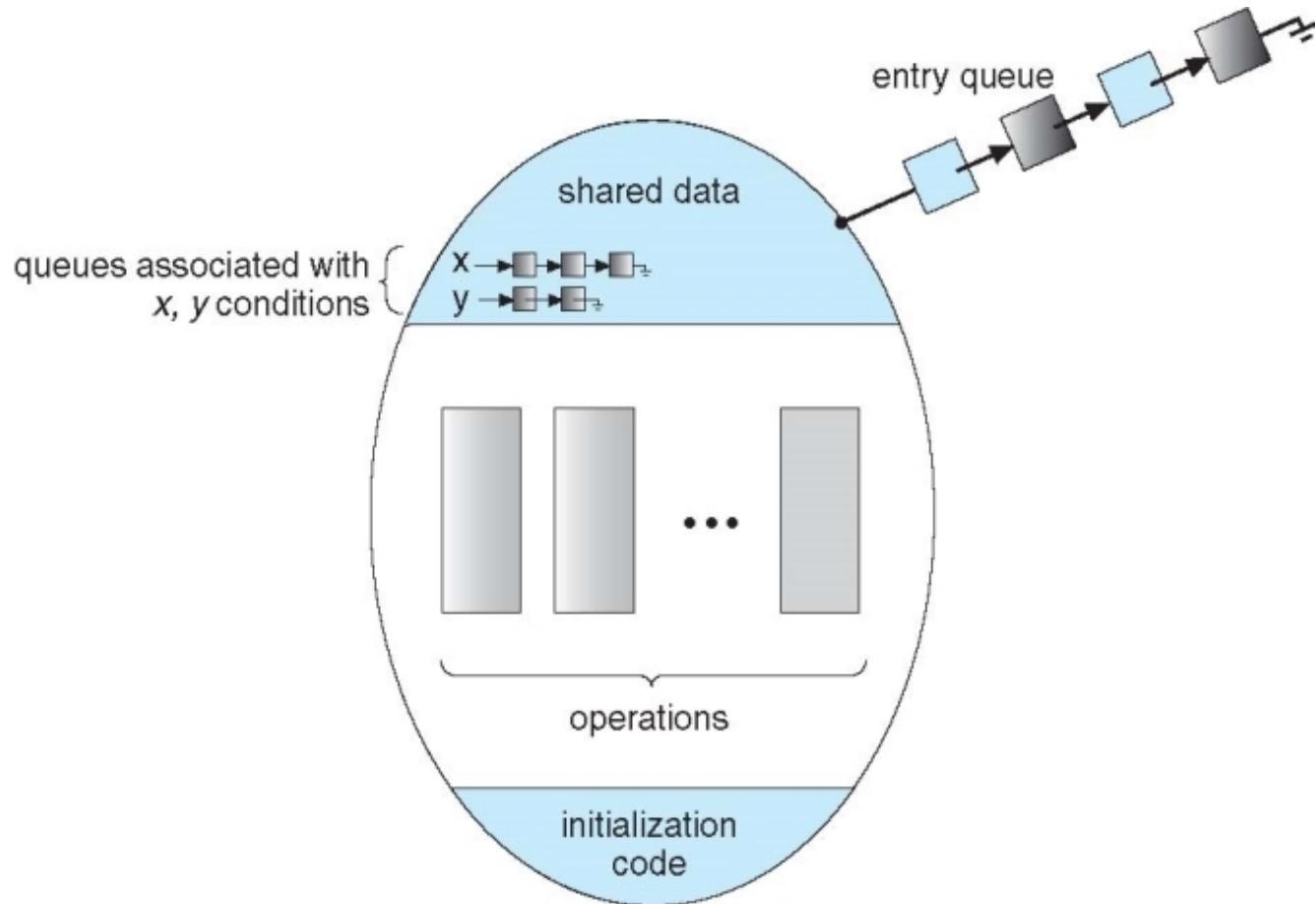
---

- n **condition x, y;**
- n 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
- n If a thread issues **x.wait()** on condition **x**, if another thread has the resource, then the thread blocks and is put on the queue for **x**.
  - | Another thread now has access to the monitor.





# 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. One must continue, but which one?
- Options include
  - **Signal and wait** – P waits Q continues until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – P continues until it either leaves the monitor or it waits for another condition, Q waits for P to relinquish monitor
  - 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

- Problem: Philosophers all sitting around a table with one chopstick between each pair.
- Solution is for a Philosopher to only pick up chopsticks when the two closest chopsticks are available.
- Means that the Philosophers on either side do not have the chopsticks.
- A Philosopher can either be: **HUNGRY**, **THINKING** or **EATING**.
  
- If a Philosopher is **HUNGRY**, request access to the chopsticks on either side:
  - Must test if Philosophers on either side are **NOT EATING**.
  - If a Philosopher is running a function of the Monitor, that Philosopher has exclusive access to the Monitor.





# 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); // Tests if chopsticks are available
        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);
    }
}
```





# Solution to Dining Philosophers (Cont.)

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





# Solution to Dining Philosophers (Cont.)

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

**DiningPhilosophers.pickup(i) ;**

**EAT**

**DiningPhilosophers.putdown(i) ;**

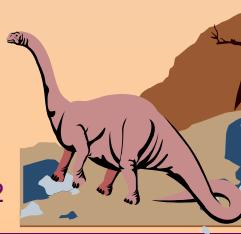
- No deadlock, but starvation is possible





# Chapter 8: Deadlocks

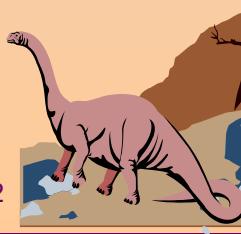
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling





# What is deadlock?

- Several processes may compete for a finite number of resources.
- A process requests a resource; if resource not available at that time, then the process enters a waiting state.
- Sometimes the waiting process is never again able to change state, because the resource(s) it has requested are held by other waiting process.
- This is deadlock.

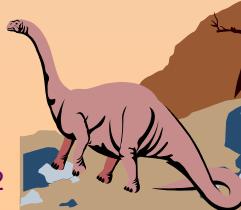




# The Deadlock Problem

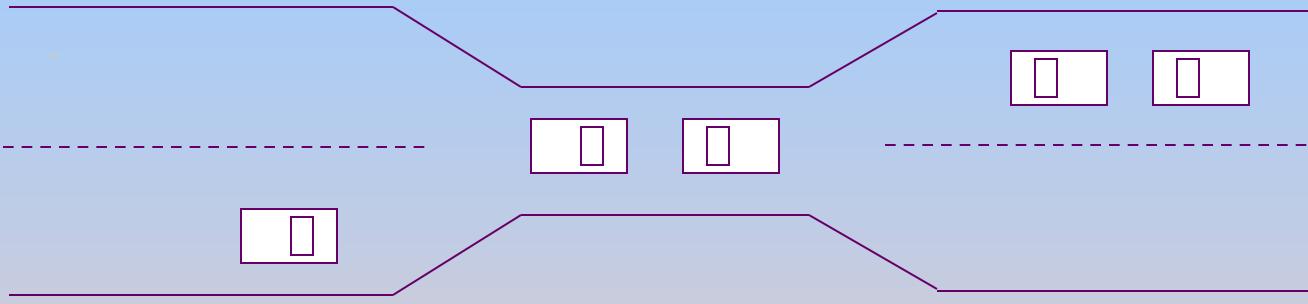
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - ☞ System has 2 tape drives.
  - ☞  $P_1$  and  $P_2$  each hold one tape drive and each needs another one.
- Example
  - ☞ semaphores  $A$  and  $B$ , initialized to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

|                  |                |
|------------------|----------------|
| $P_0$            | $P_1$          |
| <i>wait (A);</i> | <i>wait(B)</i> |
| <i>wait (B);</i> | <i>wait(A)</i> |

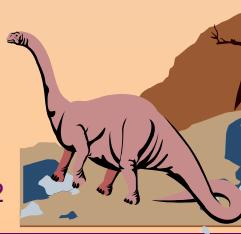




# Bridge Crossing Example



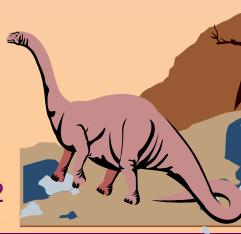
- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.





# System Model

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

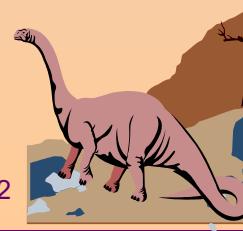




# Deadlock Characterization

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  $P_2, \dots, P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

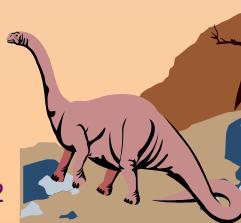




# Resource-Allocation Graph (RAG)

Deadlocks described more precisely in terms of a directed graph called system resource allocation graph.

- The graph consists of 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.)

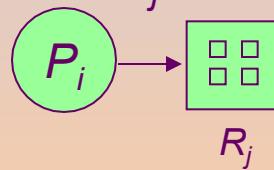
- Process



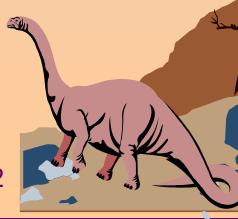
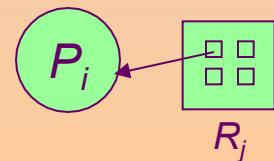
- Resource Type with 4 instances



- $P_i$  requests instance of  $R_j$



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





# Example of a Resource Allocation Graph

V- Set of vertices P and R

$P = \{P_1, P_2, P_3\}$ , the set consisting of all the processes in the system.

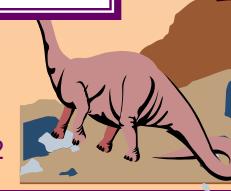
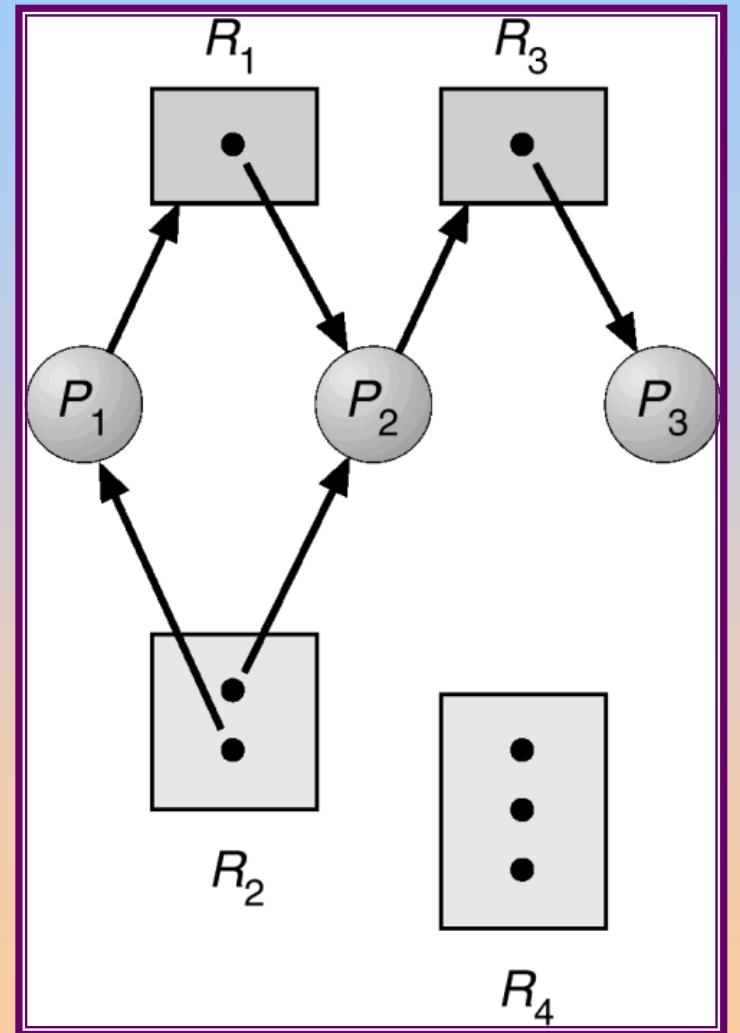
$R = \{R_1, R_2, R_3, R_4\}$ , the set consisting of all resource types in the system.

2 instances of resource R2

1 instance of resource R1, R3

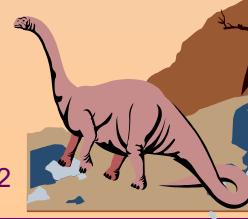
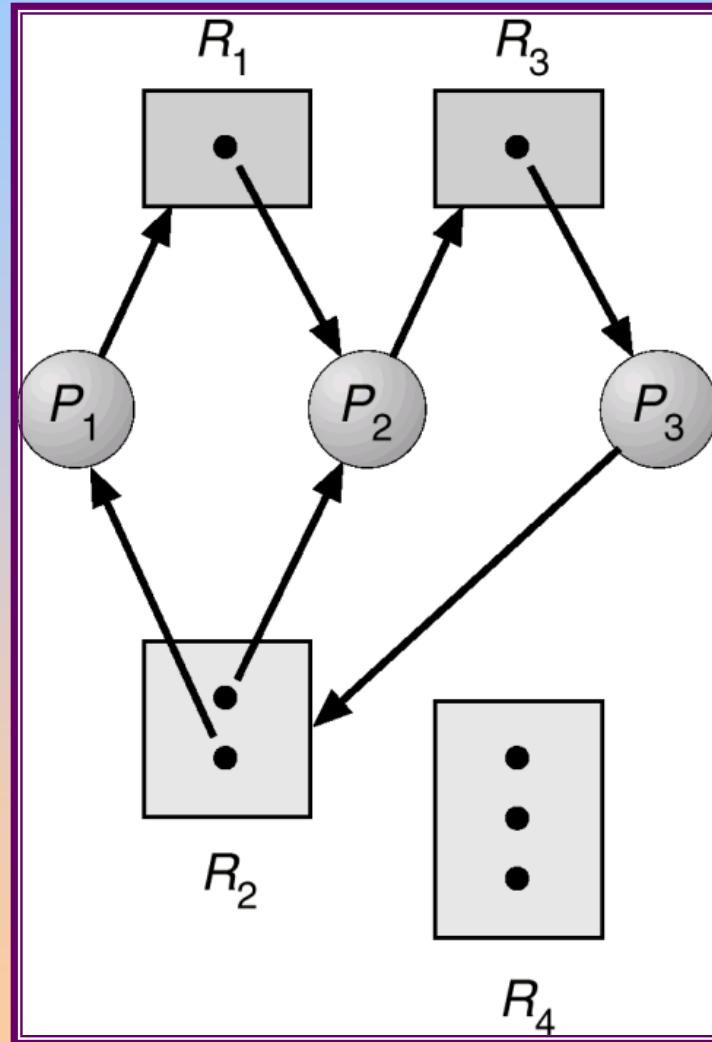
3 instances of resource R4

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$



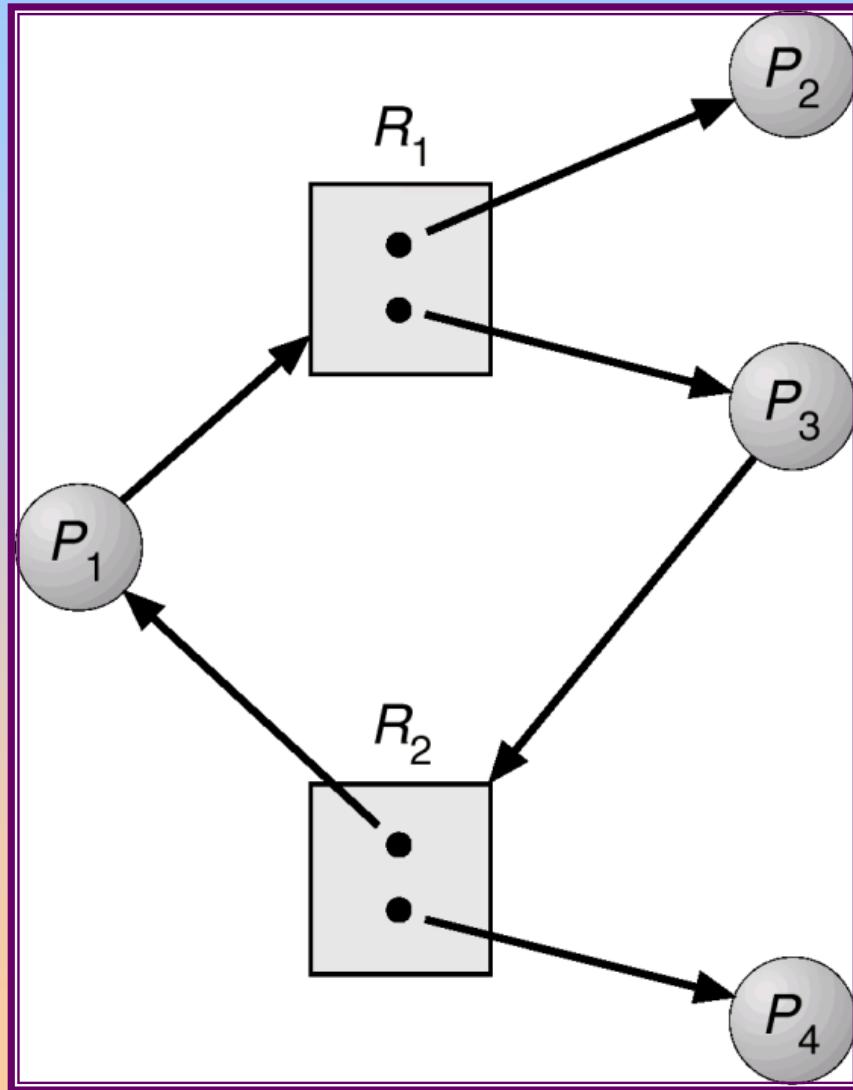


# Resource Allocation Graph With A Deadlock





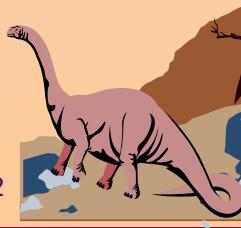
## Resource Allocation Graph With A Cycle But No Deadlock





# Basic Facts

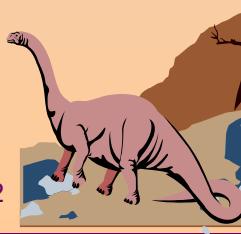
- If graph contains no cycles  $\Rightarrow$  no deadlock.
- If graph contains a cycle  $\Rightarrow$ 
  - ☞ if only one instance per resource type, then deadlock.
  - ☞ if several instances per resource type, possibility of deadlock.





# Methods for Handling Deadlocks

- Using protocols prevent/avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then detect it and recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.



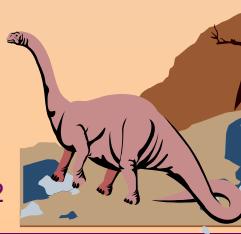


# Deadlock Prevention

Restrain the ways request can be made.

Ensures at least one of the necessary conditions for deadlock cannot hold.

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



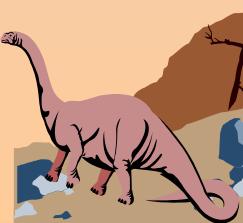


# Deadlock Prevention (Cont.)

## ■ No Preemption –

- ☞ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- ☞ Preempted resources are added to the list of resources for which the process is waiting.
- ☞ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

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



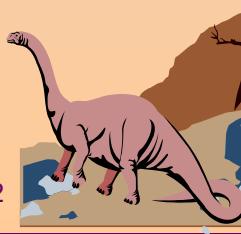


# Deadlock Avoidance

Requires that the system has some additional *apriori* information available.

If the system knows before, the resources the processes are going to use during their life time, then the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

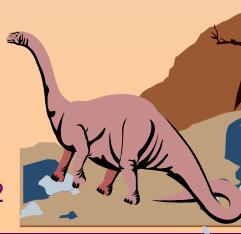
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.





# Deadlock Avoidance

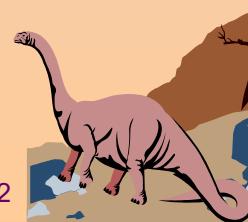
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes(future requests and release of resources of each process).
- **The system uses the above information and constructs an algorithm that ensures the system will never enter a deadlock state.**





# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- A system is safe if the system can allocate resources to each process in some order and still avoid deadlock.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence  $\langle P_1, P_2, \dots, P_n \rangle$  is safe if for each  $P_i$ , the resources that  $P_i$  can still request can 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  $P_i$  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.





# Safe State

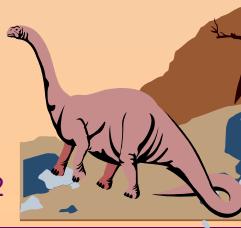
- Consider a system with 12 magnetic tape drives and with 3 processes P0, P1 and P2.

|      | Max needs | current allocation |
|------|-----------|--------------------|
| ■ P0 | 10        | 5                  |
| ■ P1 | 4         | 2                  |
| ■ P2 | 9         | 2                  |

Remaining tape drives 3

The system is in safe state if executed in order <P1,P0,P2>

Suppose if P2 requests for 1 more resource then





# Safe State

|      | Max needs | current allocation |
|------|-----------|--------------------|
| ■ P0 | 10        | 5                  |
| ■ P1 | 4         | 2                  |
| ■ P2 | 9         | 3                  |

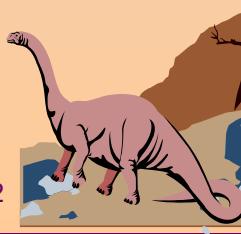
Remaining tape drives 2

Now, the system is not in safe state.

Only P1 can complete.

So it is a mistake to give this resource to P2

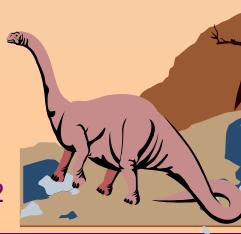
- Hence, the deadlock avoidance algorithm will allocate resources to process requesting for resources only if the system will be in safe state after allocation

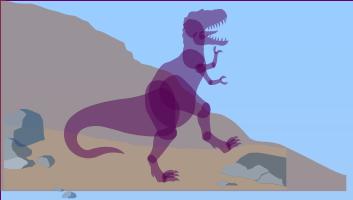




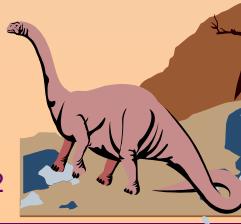
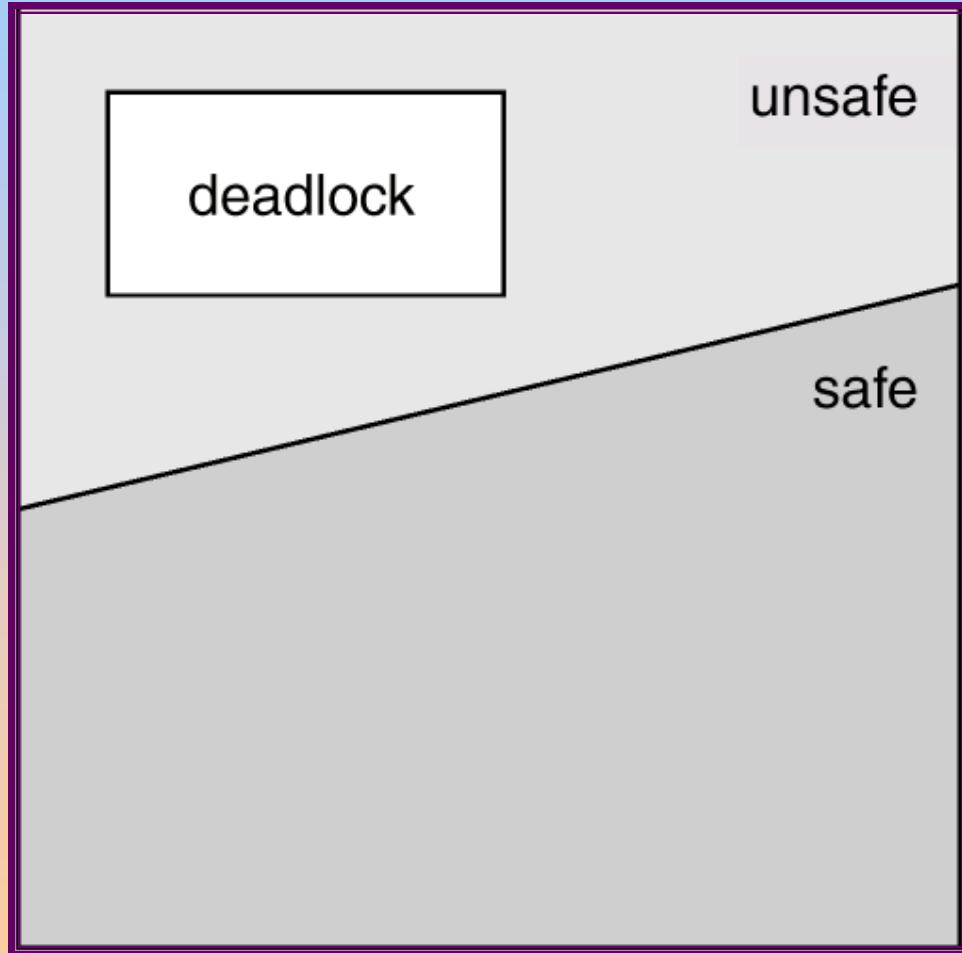
# Basic Facts

- 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.





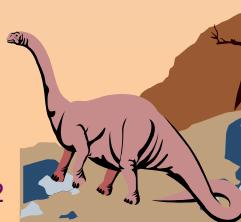
# Safe, Unsafe , Deadlock State





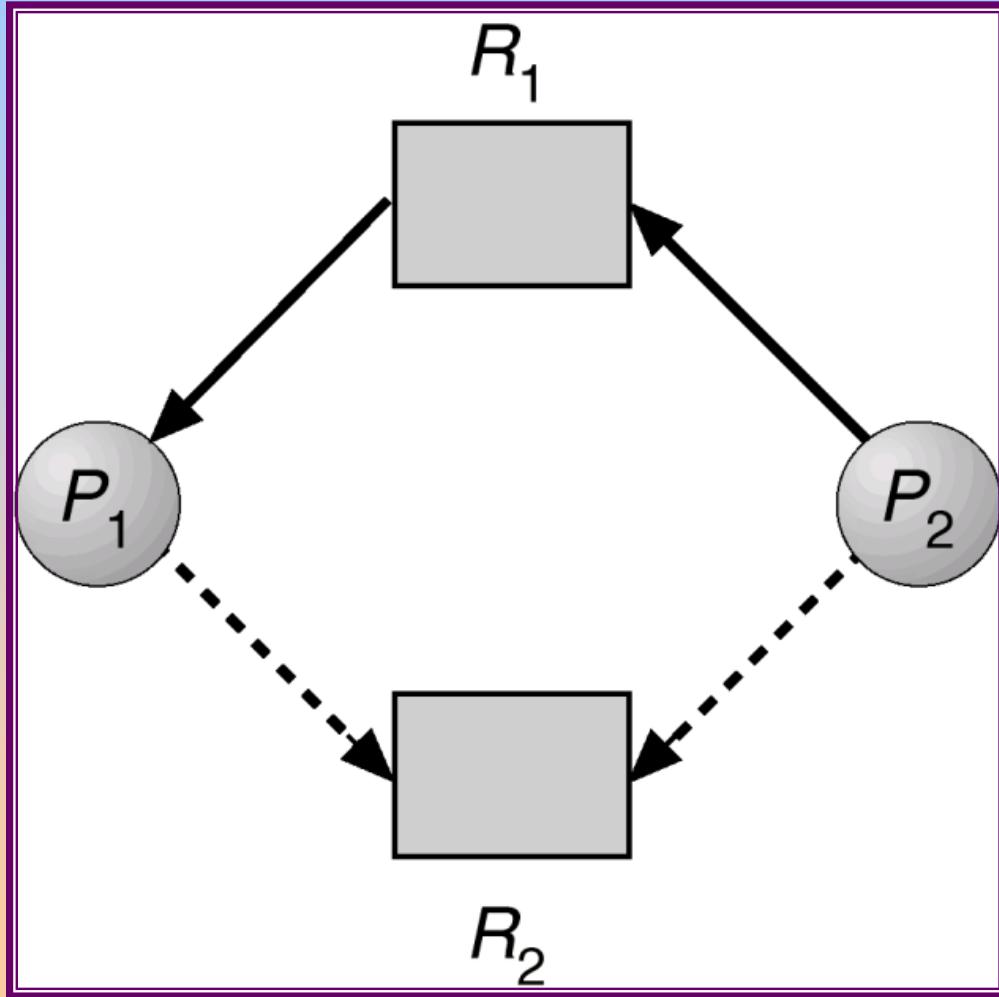
# Resource-Allocation Graph Algorithm

- *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.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.



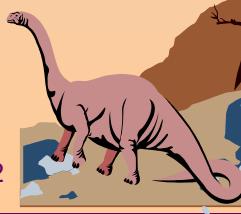
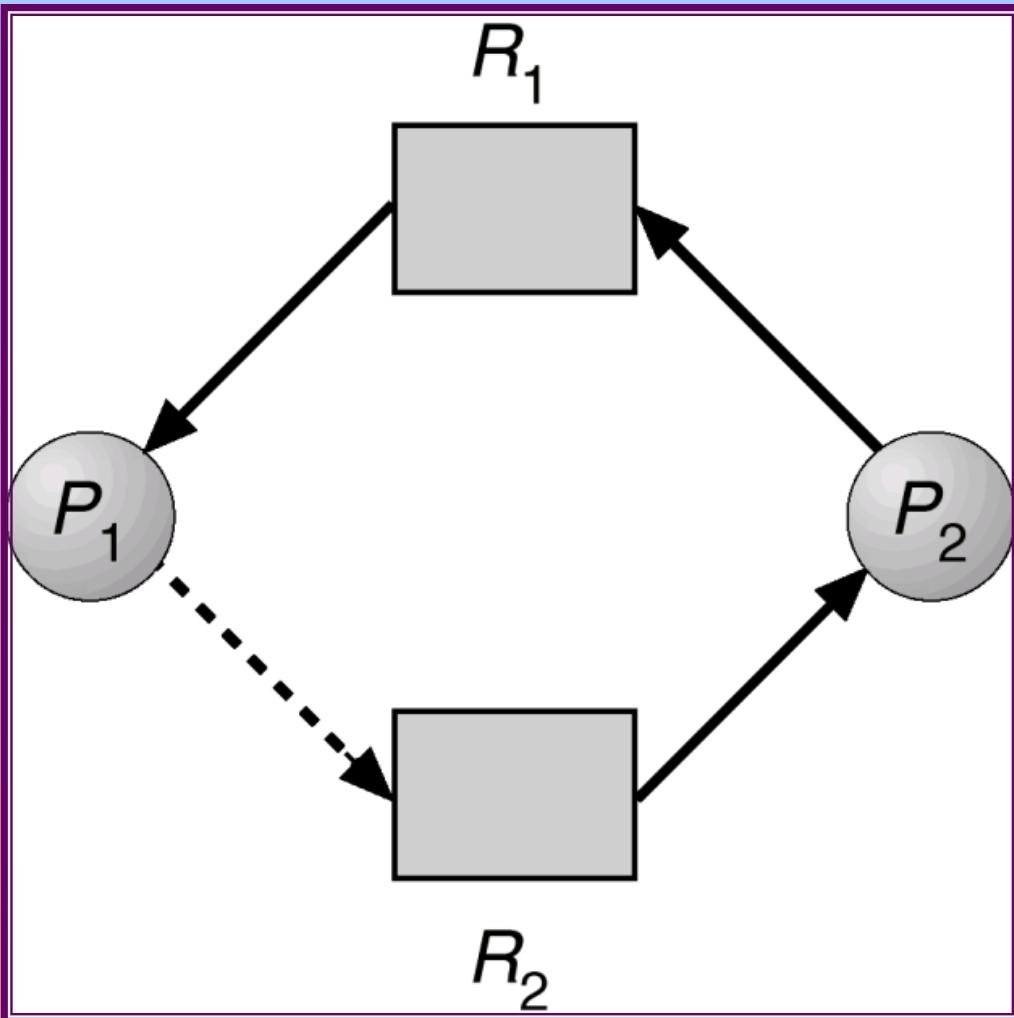


# Resource-Allocation Graph For Deadlock Avoidance





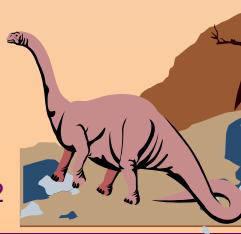
# Unsafe State In Resource-Allocation Graph





# Banker's Algorithm

- If multiple instances of the same resource type is available we go for this algorithm than RAG
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.



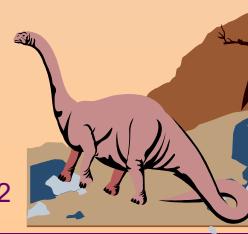


# Data Structures for the Banker's Algorithm

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

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

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





# Safety Algorithm

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

$Work = Available$

$Finish[i] = false$  for  $i = 1, 2, \dots, n$ .

2. Find an index  $i$  such that both:

(a)  $Finish[i] = false$

(b)  $Need_i \leq Work$

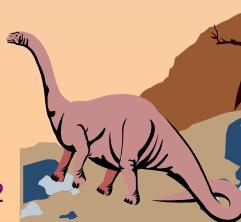
If no such  $i$  exists, go to step 4.

3.  $Work = Work + Allocation_i$

$Finish[i] = true$

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 $P_i$

$\text{Request}$  = 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

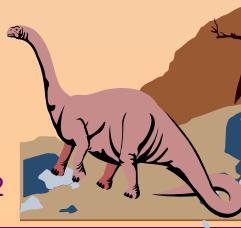




# Example of Banker's Algorithm

- 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> |   |   |
|-------|-------------------|---|---|------------|---|---|------------------|---|---|
|       | A                 | B | C | A          | B | C | A                | B | C |
| $P_0$ | 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 |                  |   |   |



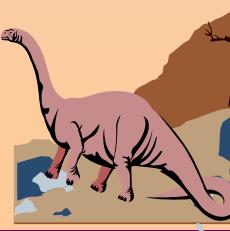


## Example (Cont.)

- 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  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria.



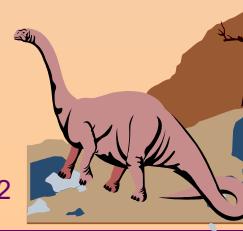


# Example $P_1$ Request (1,0,2) (Cont.)

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

|       | <u>Allocation</u> |          |          | <u>Need</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           | 4        | 3        | 2                | 3        | 0        |
| $P_1$ | 3                 | 0        | 2        | 0           | 2        | 0        |                  |          |          |
| $P_2$ | 3                 | 0        | 1        | 6           | 0        | 0        |                  |          |          |
| $P_3$ | 2                 | 1        | 1        | 0           | 1        | 1        |                  |          |          |
| $P_4$ | 0                 | 0        | 2        | 4           | 3        | 1        |                  |          |          |

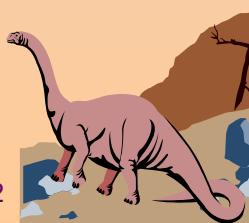
- Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.
- Can request for  $(3,3,0)$  by  $P_4$  be granted?
- Can request for  $(0,2,0)$  by  $P_0$  be granted?





# Deadlock Detection

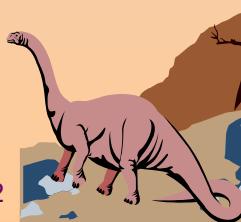
- If no deadlock prevention/deadlock avoidance mechanism is followed then deadlock situation may occur. Hence the system may provide
  - ☞ Algorithm that examines the state of the system to determine whether deadlock has occurred.
  - ☞ Algorithm to recover from deadlock.
- We can use for deadlock detection
  - ☞ Wait-for-graph derived from RAG
  - ☞ An algorithm similar to safety algorithm and resource request algorithm





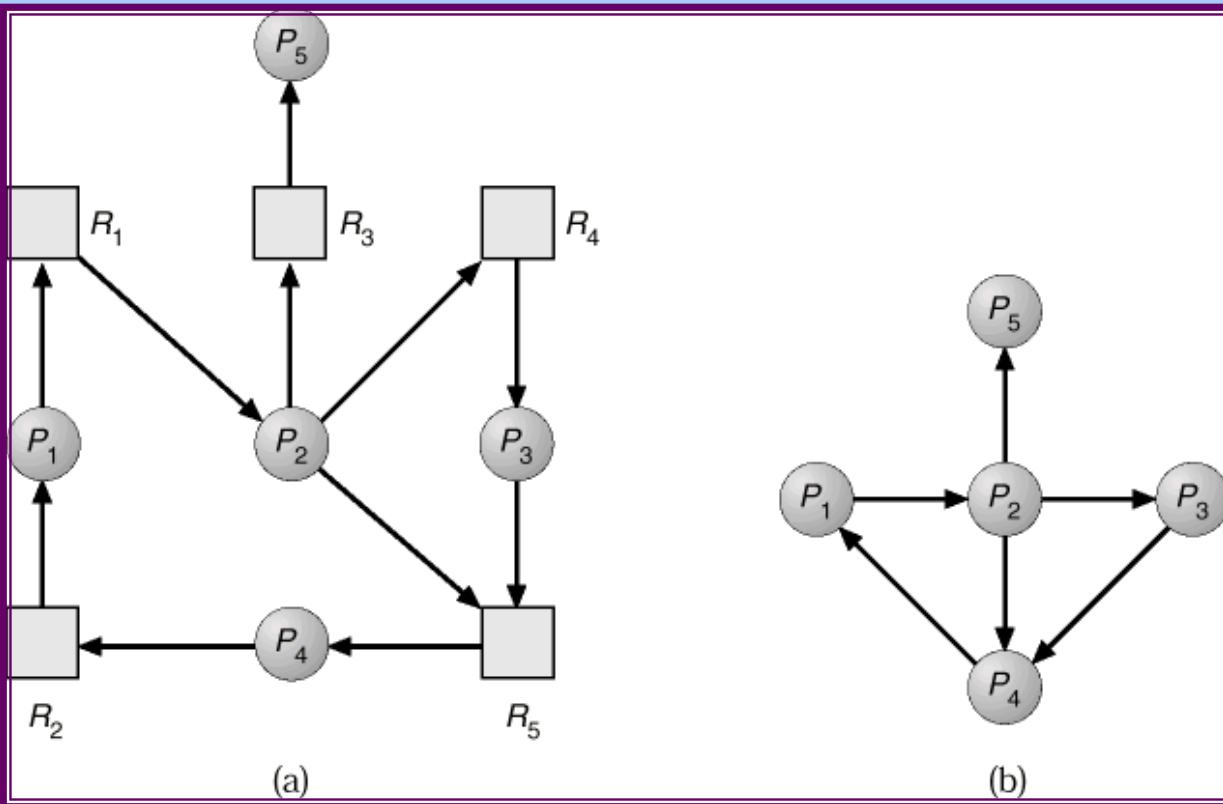
# Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - ☞ Nodes are processes.
  - ☞  $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$ .
- Periodically invoke an algorithm that searches for a cycle in the graph.
- 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.



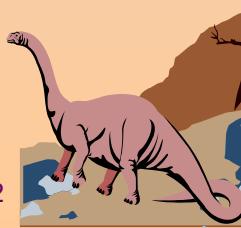


# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

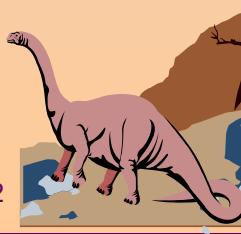
Corresponding wait-for graph





# 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.  $R_j$ .

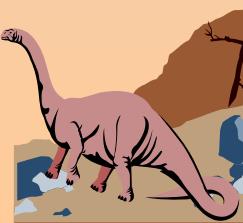




# Detection Algorithm

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively Initialize:
  - (a)  $Work = Available$
  - (b) For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then  $Finish[i] = \text{false}$ ; otherwise,  $Finish[i] = \text{true}$ .
2. Find an index  $i$  such that both:
  - (a)  $Finish[i] == \text{false}$
  - (b)  $Request_i \leq Work$

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

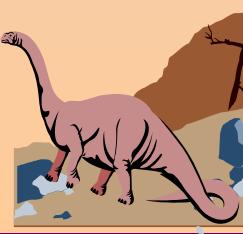




# Detection Algorithm (Cont.)

3.  $Work = Work + Allocation_i$ ,  
 $Finish[i] = true$   
go to step 2.
4. If  $Finish[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == \text{false}$ , 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.



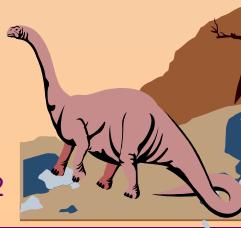


# Example of Detection Algorithm

- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time  $T_0$ :

|       | <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  $Finish[i] = \text{true}$  for all  $i$ .





# Example (Cont.)

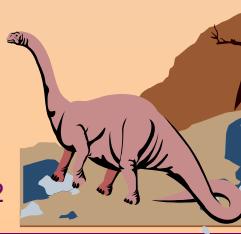
- $P_2$  requests an additional instance of type C.

Request

|       | A | B | C |
|-------|---|---|---|
| $P_0$ | 0 | 0 | 0 |
| $P_1$ | 2 | 0 | 1 |
| $P_2$ | 0 | 0 | 1 |
| $P_3$ | 1 | 0 | 0 |
| $P_4$ | 0 | 0 | 2 |

- State of system?

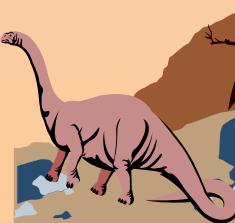
- ☞ Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes' requests.
- ☞ Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$ .





# Detection-Algorithm Usage

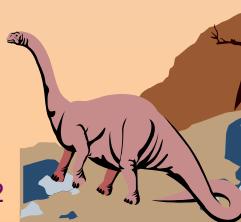
- When, and how often, to invoke depends on:
  - ☞ How often a deadlock is likely to occur?
  - ☞ How many processes will need to be rolled back?
    - ☞ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.





# Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - ☞ Priority of the process.
  - ☞ How long process has computed, and how much longer to completion.
  - ☞ Resources the process has used.
  - ☞ Resources process needs to complete.
  - ☞ How many processes will need to be terminated.
  - ☞ Is process interactive or batch?

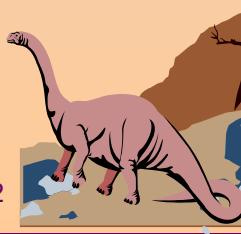




# Recovery from Deadlock: Resource Preemption

Preempting resources from processes and giving these resources to other processes until deadlock cycle is broken.

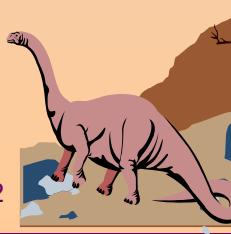
- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, so include number of rollback in cost factor.





# Combined Approach to Deadlock Handling

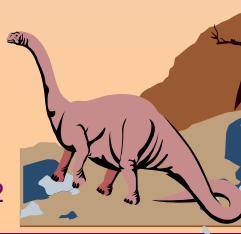
- Combine the three basic approaches
  - ☞ prevention
  - ☞ avoidance
  - ☞ detectionallowing the use of the optimal approach for each of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.





# Resource Trajectories

- One way of describing resource allocation is through resource trajectories, sometimes called joint progress diagrams.
- Example, there are two processes, P and Q. Each uses shared resources A and B, but in slightly different order. Process P proceeds along the X axis, process Q along the Y axis.

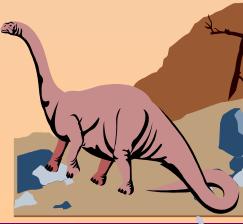




# Resource Trajectories

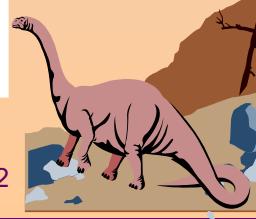
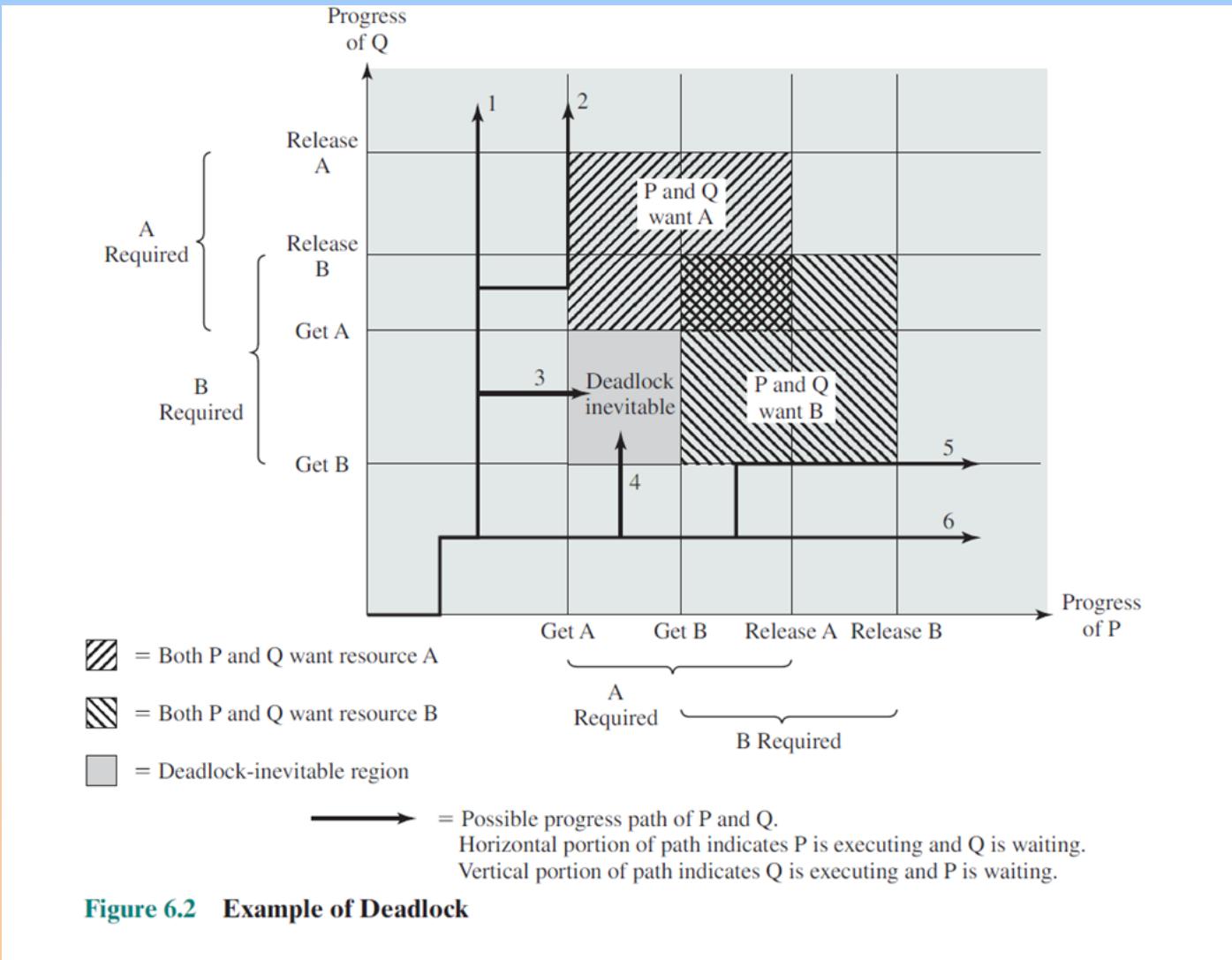
```
void P (void) {  
    Get_A();  
    Get_B();  
    /* Need both resources here */  
    Release_A();  
    Release_B();  
}
```

```
void Q (void) {  
    Get_B();  
    Get_A();  
    /* Need both resources here */  
    Release_B();  
    Release_A();  
}
```





# Resource Trajectories

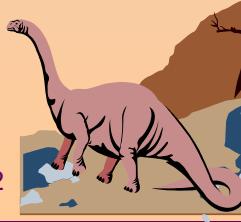




# Resource Trajectories

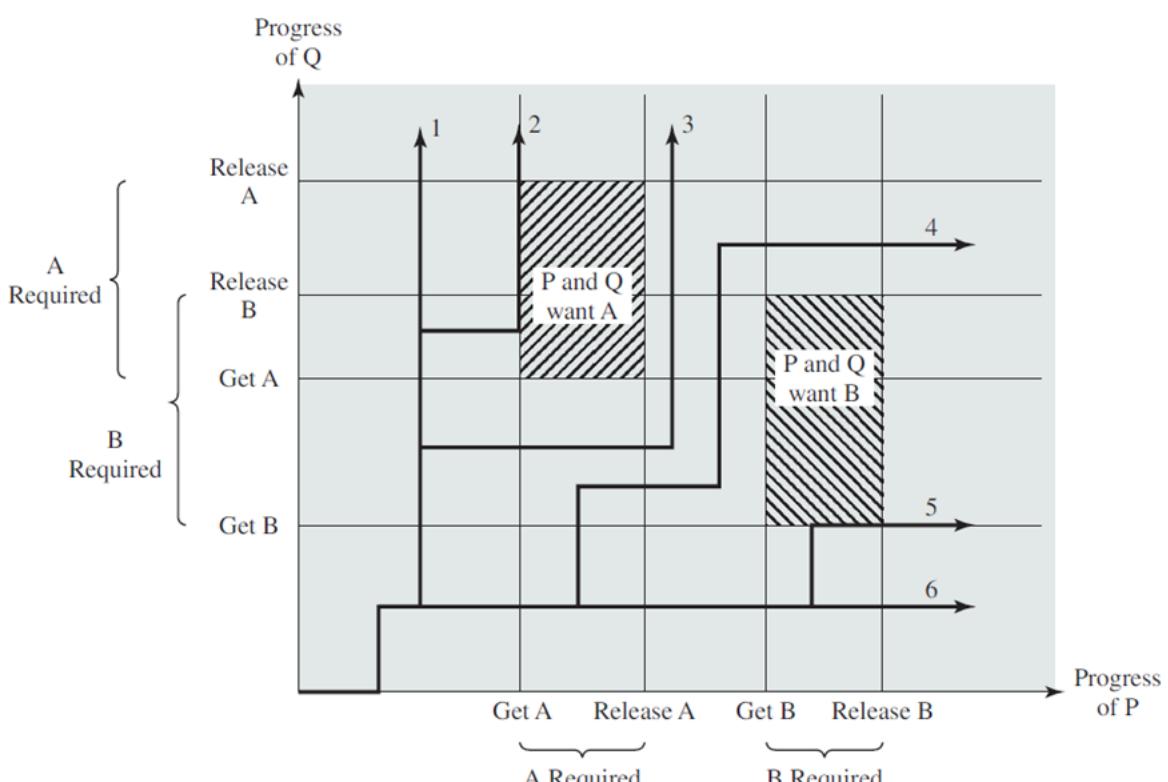
Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application. For example, suppose that P does not need both resources at the same time so that the two processes have the following form:

| Process P | Process Q |
|-----------|-----------|
| • • •     | • • •     |
| Get A     | Get B     |
| • • •     | • • •     |
| Release A | Get A     |
| • • •     | • • •     |
| Get B     | Release B |
| • • •     | • • •     |
| Release B | Release A |
| • • •     | • • •     |





# Resource Trajectories



**Figure 6.3** Example of No Deadlock [BACO03]

