

CSE-202 OPERATING SYSTEM

Module I: Introduction to Operating Systems

Operating system and function

Course Learning Objectives:

- To understand the concept of operating system and its function

Operating System and Function

Topics Covered

- Definition and Goal of Operating System
- Components Of Computer System
- Different View Of Operating System
- Functions of Operating System

What is an Operating System?

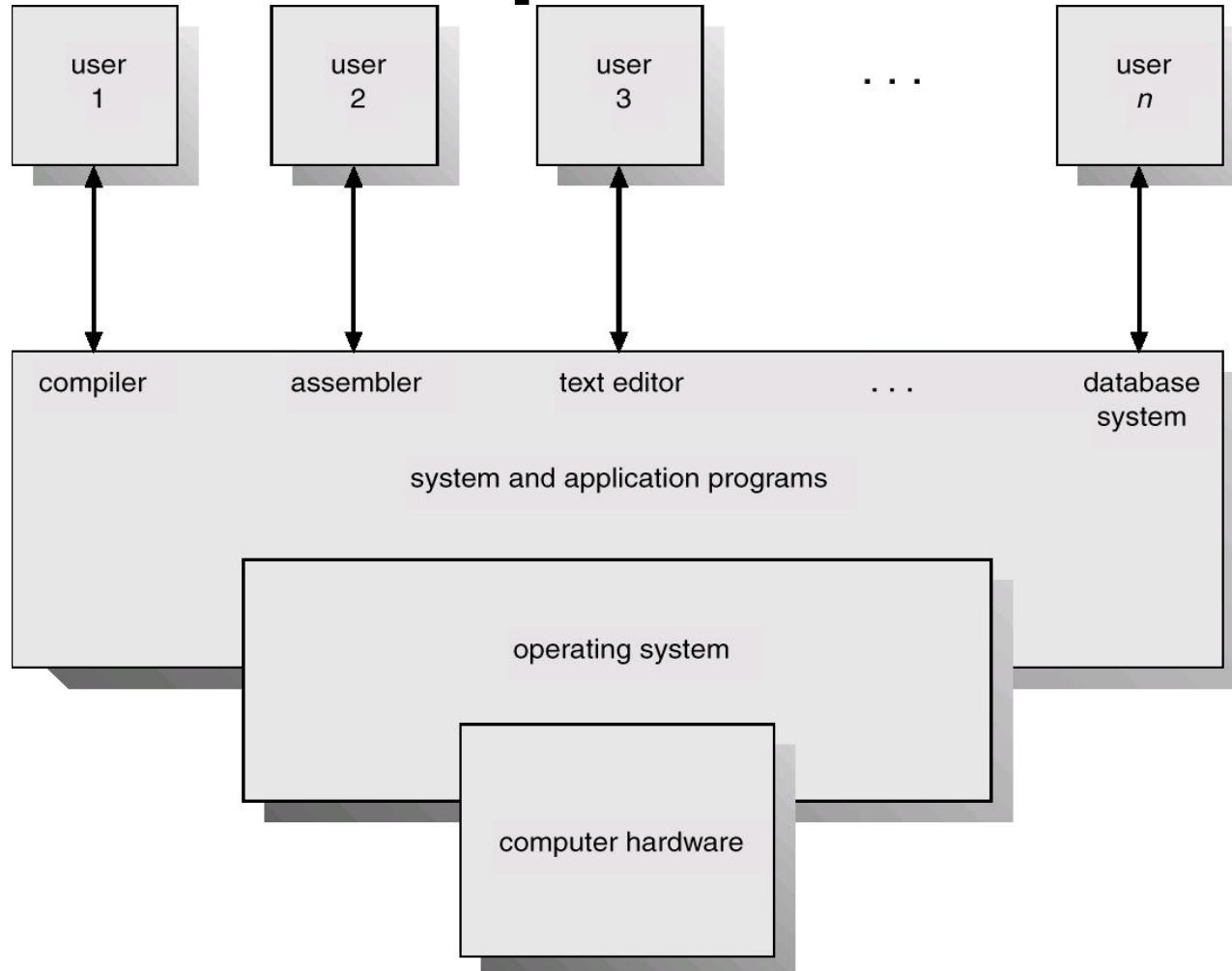
- A program that acts as an intermediary between a user of a computer and the computer hardware.
- Operating system goals:
 - Execute user programs and make solving user problems easier.
 - Make the computer system convenient to use.
 - Use the computer hardware in an efficient manner.



Computer System Components

1. Hardware – provides basic computing resources (CPU, memory, I/O devices).
2. Operating system – controls and coordinates the use of the hardware among the various application programs for the various users.
3. Applications programs – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
4. Users (people, machines, other computers).

Abstract View of System Components





AMITY UNIVERSITY

Operating System Definitions

- Resource allocator – manages and allocates resources.
- Control program – controls the execution of user programs and operations of I/O devices .
- Kernel – the one program always running (all else being application programs).

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

Course Learning Outcomes:

- learn basic concepts and responsibilities of operating system
- Analyze and evaluate various I/O component of computing system

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**

Reference

- Operating System Concepts :
Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher

- Definition and Goal of Operating System
- Components Of Computer System
- Different View Of Operating System
- Functions of Operating System

Thank You

OPERATING SYSTEMS

Module I: Introduction to Operating Systems

Evolution of Operating System

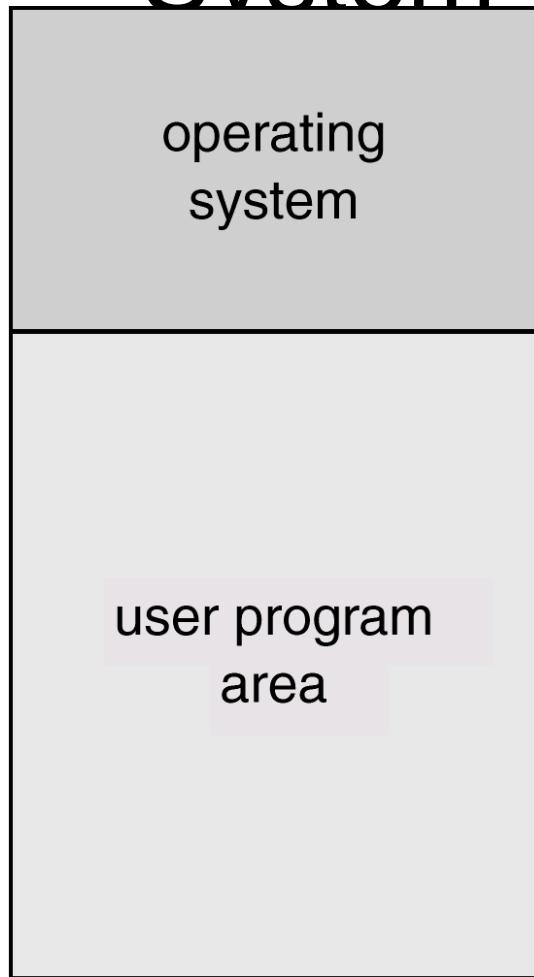
Course Learning Objectives:

- understand the evolution of operating system and difference between them

- Evolution of operating system
 - Batch
 - Interactive
 - Multiprogramming
 - Time Sharing
 - Real Time System
 - Multiprocessor system
 - Distributed system

- Hire an operator
- User ≠ operator
- Add a card reader
- Reduce setup time by batching similar jobs
- Automatic job sequencing – automatically transfers control from one job to another. First rudimentary operating system.
- Resident monitor
 - initial control in monitor
 - control transfers to job
 - when job completes control transfers back to monitor

Memory Layout for a Simple Batch System

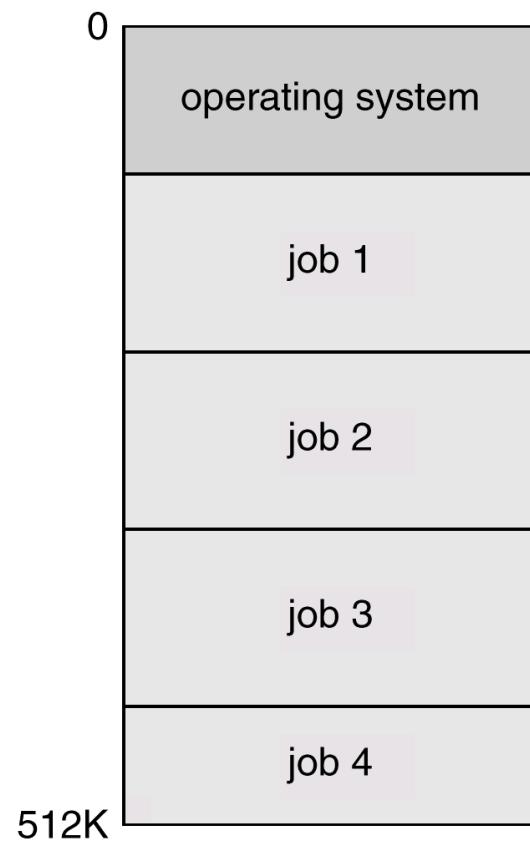


- Parts of resident monitor
 - Control card interpreter – responsible for reading and carrying out instructions on the cards.
 - Loader – loads systems programs and applications programs into memory.
 - Device drivers – know special characteristics and properties for each of the system's I/O devices.
- Problem: Slow Performance – I/O and CPU could not overlap ; card reader very slow.
- Solution: Off-line operation – speed up computation by loading jobs into memory from tapes and card reading and line printing done off-line.

Multiprogrammed Batch Systems

ASET

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



OS Features Needed for Multiprogramming

- I/O routine supplied by the system.
- Memory management – the system must allocate the memory to several jobs.
- CPU scheduling – the system must choose among several jobs ready to run.
- Allocation of devices.

- The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory).
- A job is swapped in and out of memory to the disk.
- On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next “control statement” not from a card reader, but rather from the user’s keyboard.
- On-line system must be available for users to access data and code.

Personal-Computer Systems

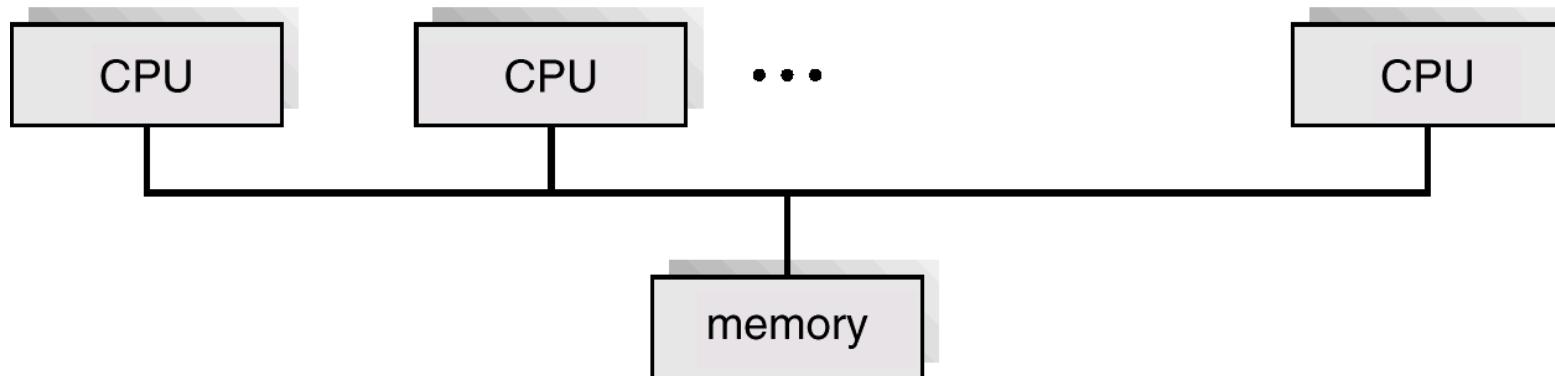
- *Personal computers* – computer system dedicated to a single user.
- I/O devices – keyboards, mice, display screens, small printers.
- User convenience and responsiveness.
- Can adopt technology developed for larger operating system' often individuals have sole use of computer and do not need advanced CPU utilization of protection features.

- Multiprocessor systems with more than one CPU in close communication.
- *Tightly coupled system* – processors share memory and a clock; communication usually takes place through the shared memory.
- Advantages of parallel system:
 - Increased *throughput*
 - Economical
 - Increased reliability
 - graceful degradation
 - fail-soft systems

Parallel Systems (Cont.)

- *Symmetric multiprocessing (SMP)*
 - No Master Slave Relationship
 - Many processes can run at once without performance deterioration.
 - Most modern operating systems support SMP
- *Asymmetric multiprocessing*
 - Each processor is assigned a specific task; master processor schedules and allocates work to slave processors.
 - More common in extremely large systems

Symmetric Multiprocessing Architecture



- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
- Well-defined fixed-time constraints.
- *Hard real-time system.*
 - Secondary storage limited or absent, data stored in short-term memory, or read-only memory (ROM)
 - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
- *Soft real-time system*
 - Limited utility in industrial control or robotics
 - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

- Distribute the computation among several physical processors.
- *Loosely coupled system* – each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.
- Advantages of distributed systems.
 - Resources Sharing
 - Computation speed up – load sharing
 - Reliability
 - Communications

Distributed Systems (Cont.)

- Network Operating System
 - provides file sharing
 - provides communication scheme
 - runs independently from other computers on the network
- Distributed Operating System
 - less autonomy between computers
 - gives the impression there is a single operating system controlling the network.

Course Learning Outcomes:

- learn basic concepts and responsibilities of operating system
- Analyze and evaluate various I/O component of computing system

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**

- Operating System Concepts :
Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher

OPERATING SYSTEMS

Module I: Introduction to Operating Systems

System Protection

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**

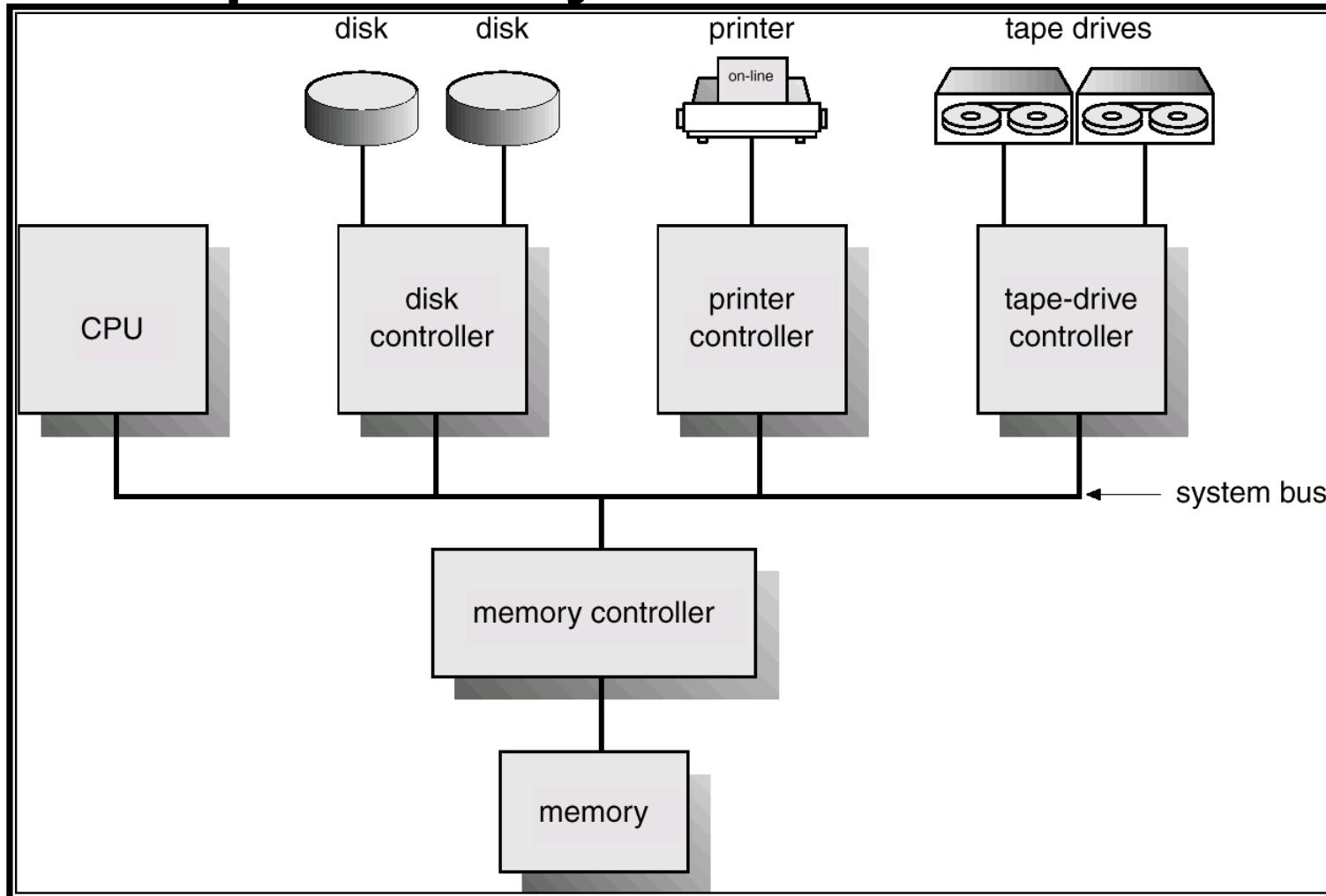
Course Learning Objectives:

- To understand protection and how operating system provides protection

Topics Covered

- Computer System Operation
- I/O Structure
- Storage Structure
- Storage Hierarchy
- Hardware Protection
- General System Architecture

Computer-System Architecture



Computer-System Operation

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

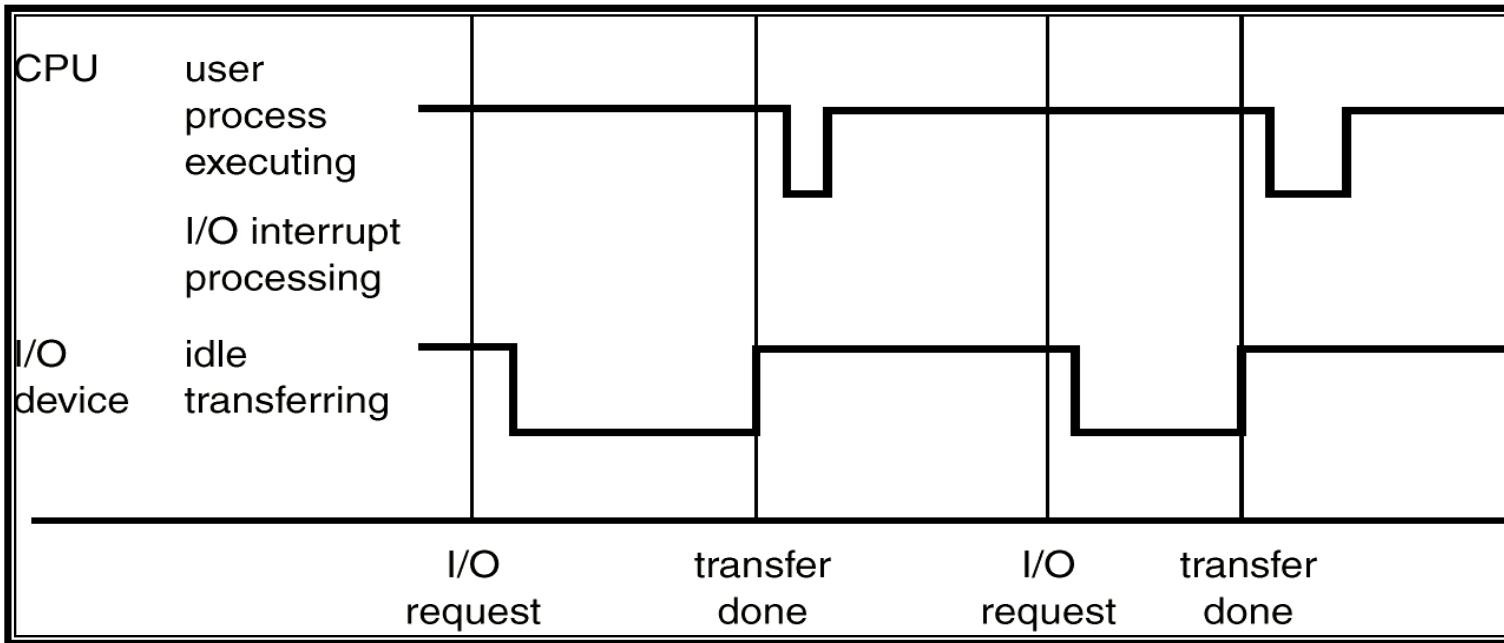
Common Functions of Interrupts

- ▶ Interrupt transfers control to the interrupt service routine generally, through the *interrupt vector*, which contains the addresses of all the service routines.
- ▶ Interrupt architecture must save the address of the interrupted instruction.
- ▶ Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.
- ▶ A *trap* is a software-generated interrupt caused either by an error or a user request.
- ▶ An operating system is *interrupt driven*.

Interrupt Handling

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

Interrupt Time Line For a Single Process Doing Output

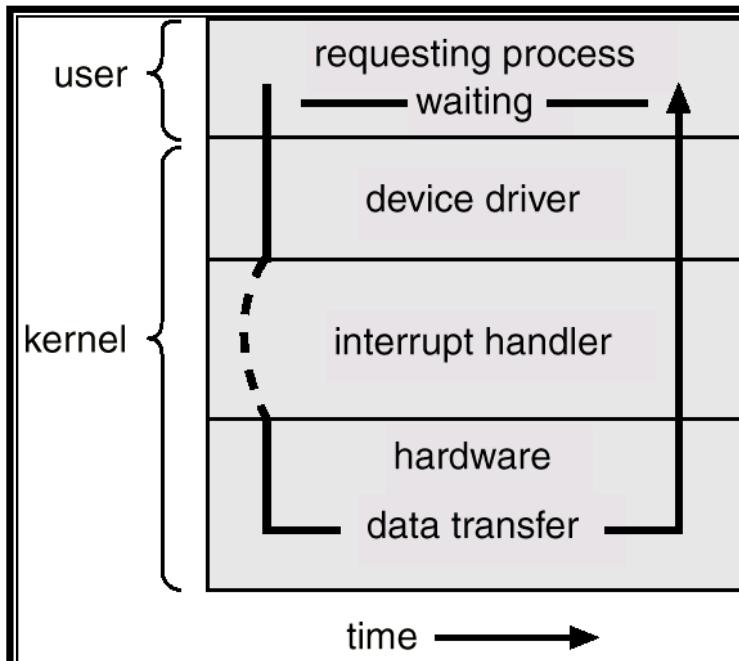


I/O Structure

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

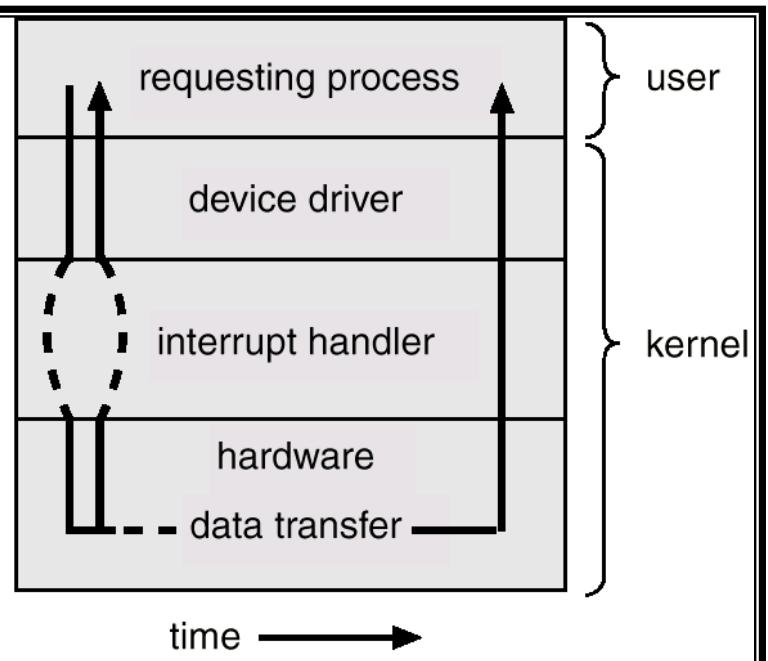
Two I/O Methods

Synchronous



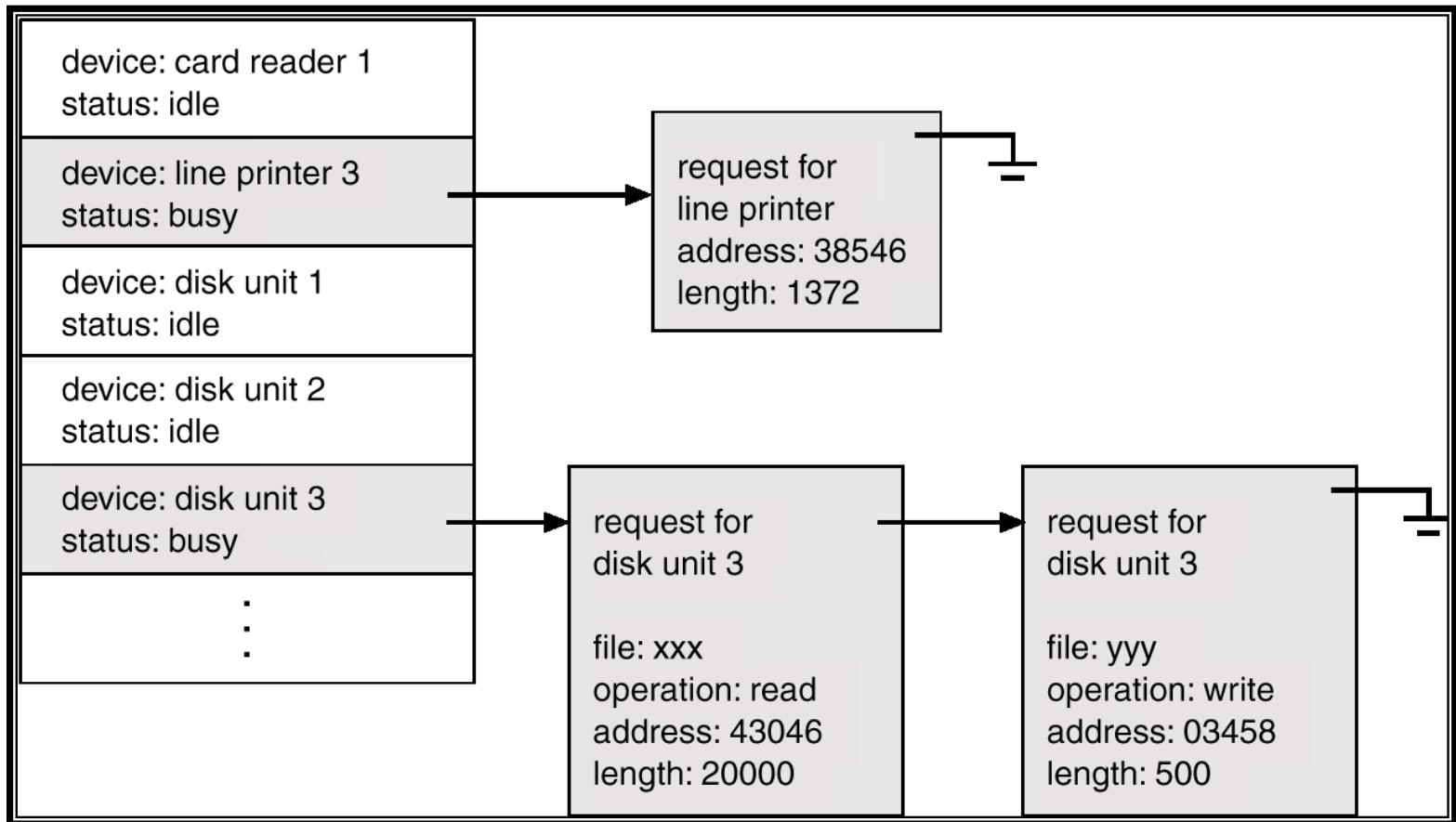
(a)

Asynchronous



(b)

Device-Status Table



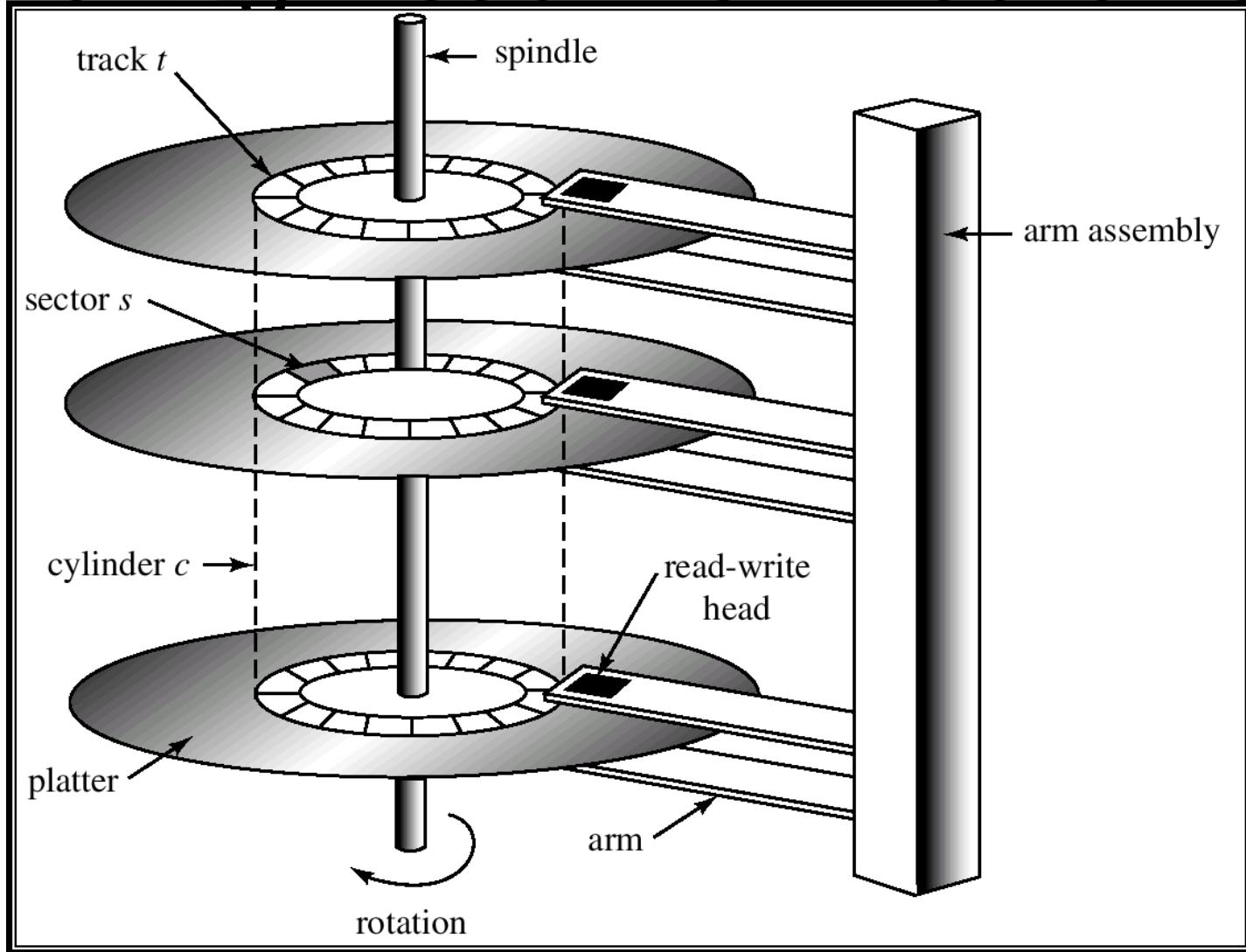
Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds.
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.
- Only one interrupt is generated per block, rather than the one interrupt per byte.

Storage Structure

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

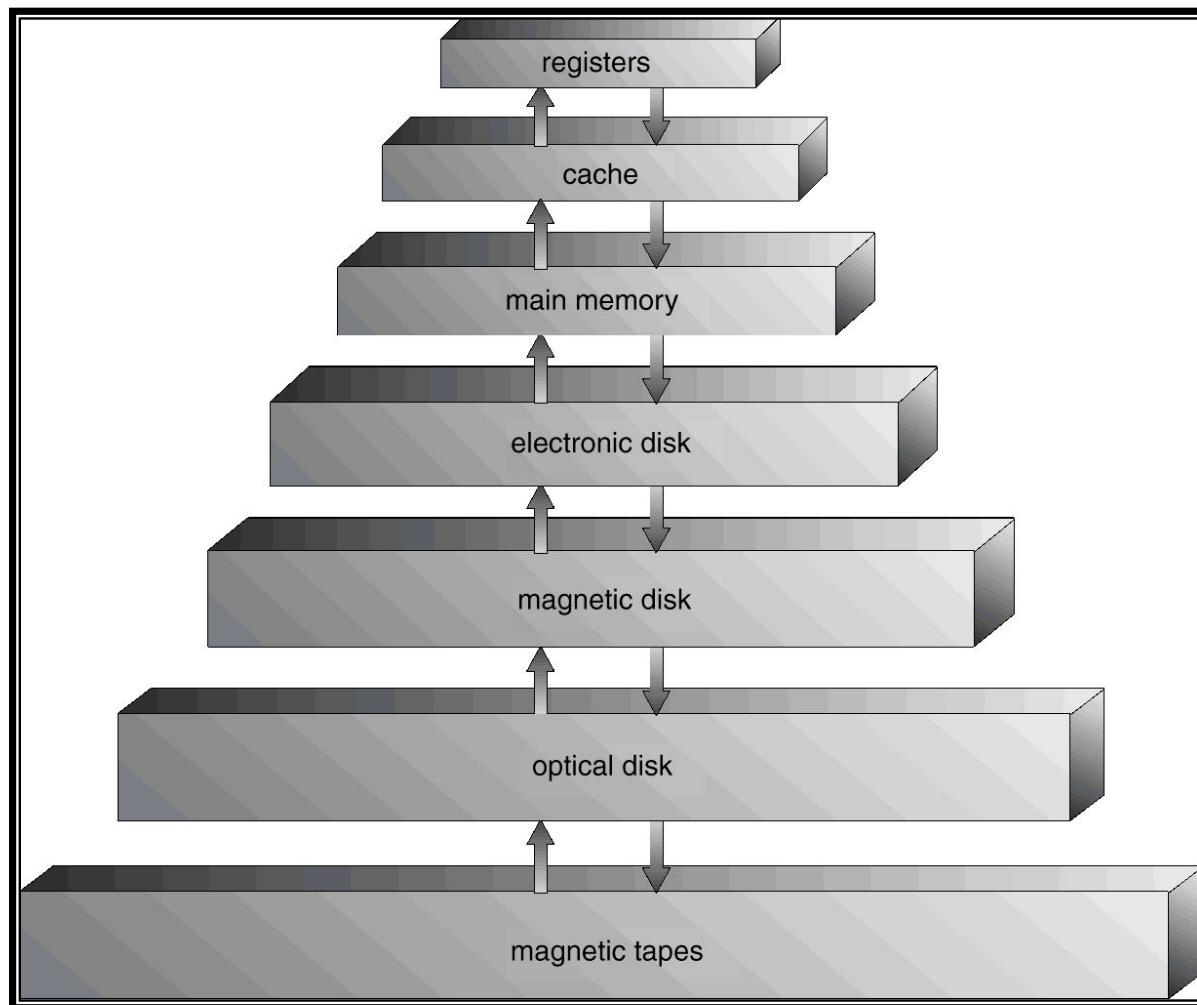
Moving-Head Disk Mechanism



Storage Hierarchy

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

Storage-Device Hierarchy

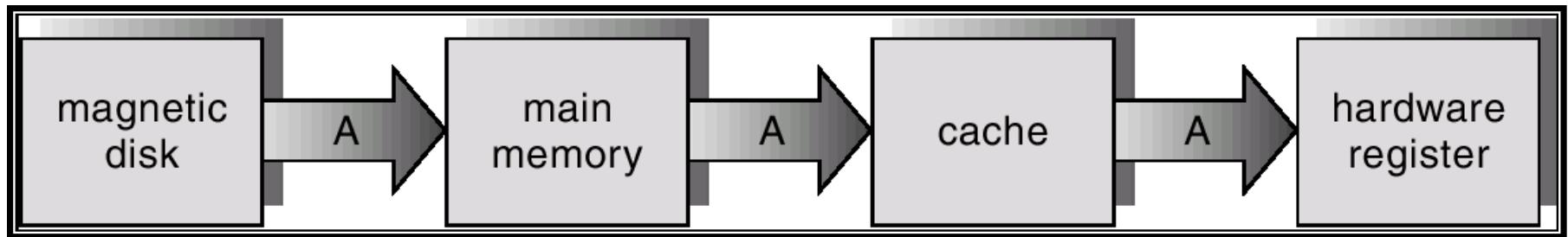


Caching

- Use of high-speed memory to hold recently-accessed data.
- Requires a *cache management* policy.
- Caching introduces another level in storage hierarchy. This requires data that is simultaneously stored in more than one level to be *consistent*.

AMITY UNIVERSITY

Migration of A From Disk to Register



Hardware Protection

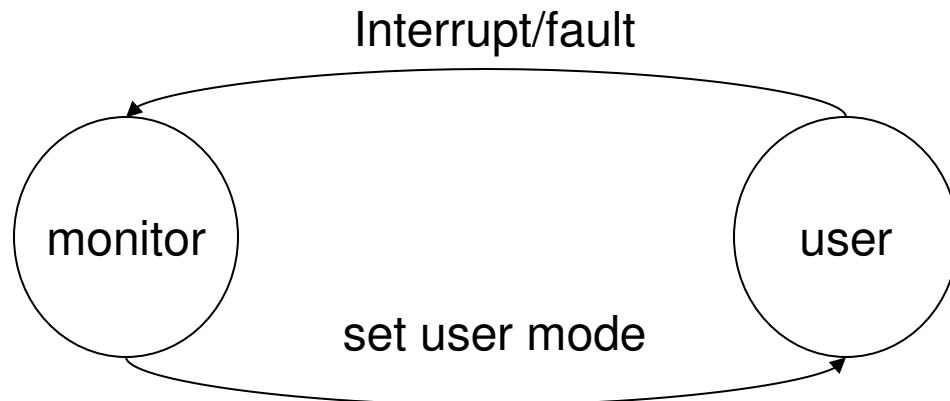
- Dual-Mode Operation
- I/O Protection
- Memory Protection
- CPU Protection

Dual-Mode Operation

- Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.
- Provide hardware support to differentiate between at least two modes of operations.
 1. *User mode* – execution done on behalf of a user.
 2. *Monitor mode* (also *kernel mode* or *system mode*) – execution done on behalf of operating system.

Dual-Mode Operation (Cont.)

- ▶ *Mode bit* added to computer hardware to indicate the current mode: monitor (0) or user (1).
- ▶ When an interrupt or fault occurs hardware switches to monitor mode.

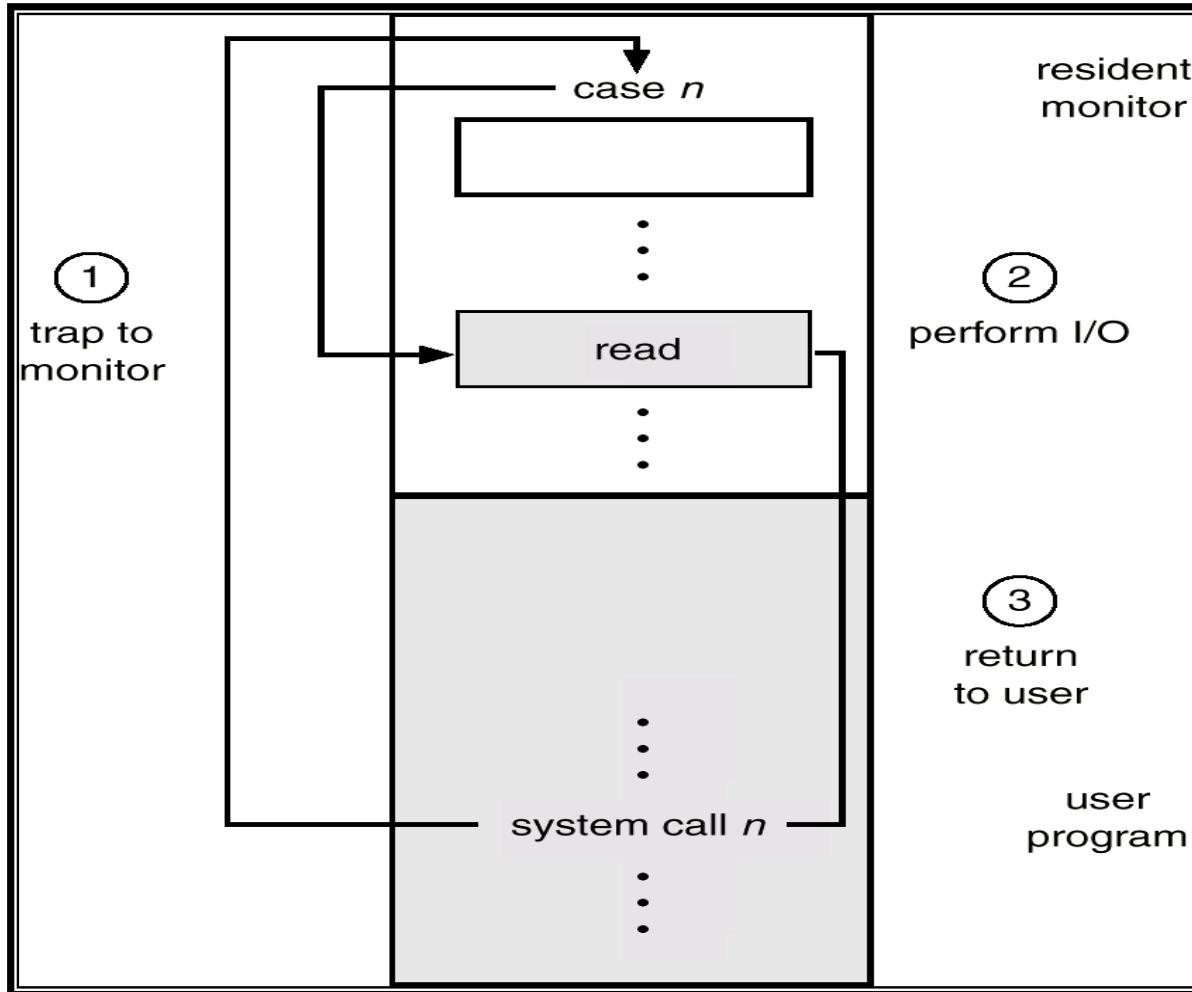


Privileged instructions can be issued only in monitor mode.

I/O Protection

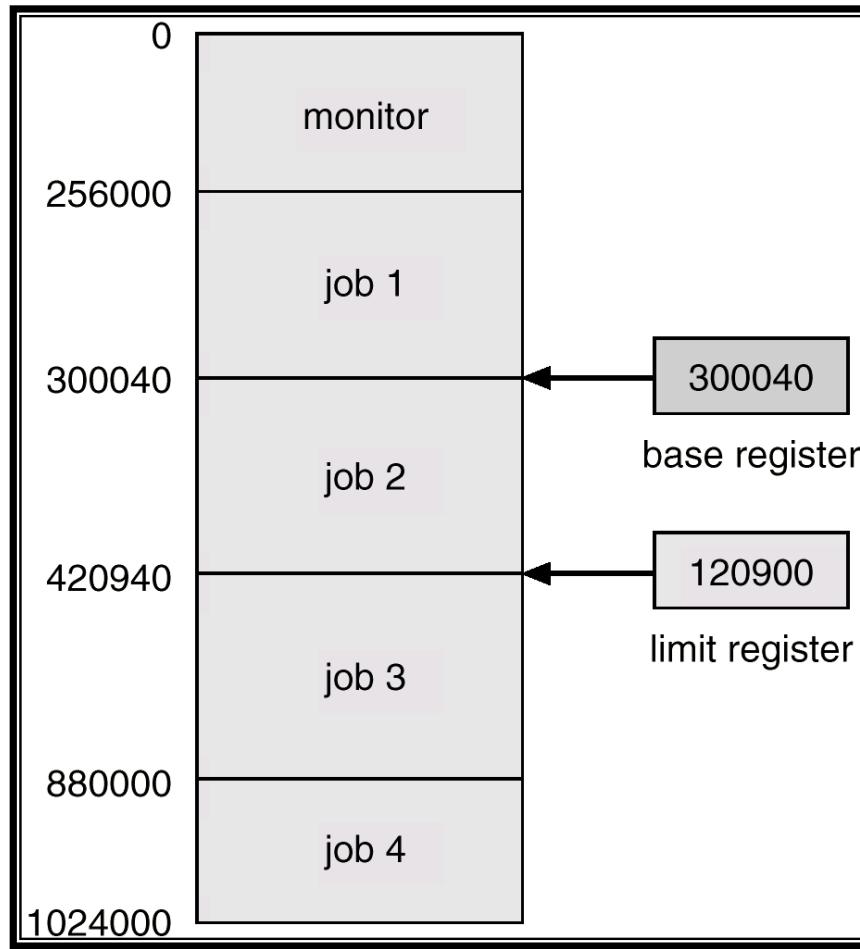
- All I/O instructions are privileged instructions.
- Must ensure that a user program could never gain control of the computer in monitor mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

Use of A System Call to Perform I/O

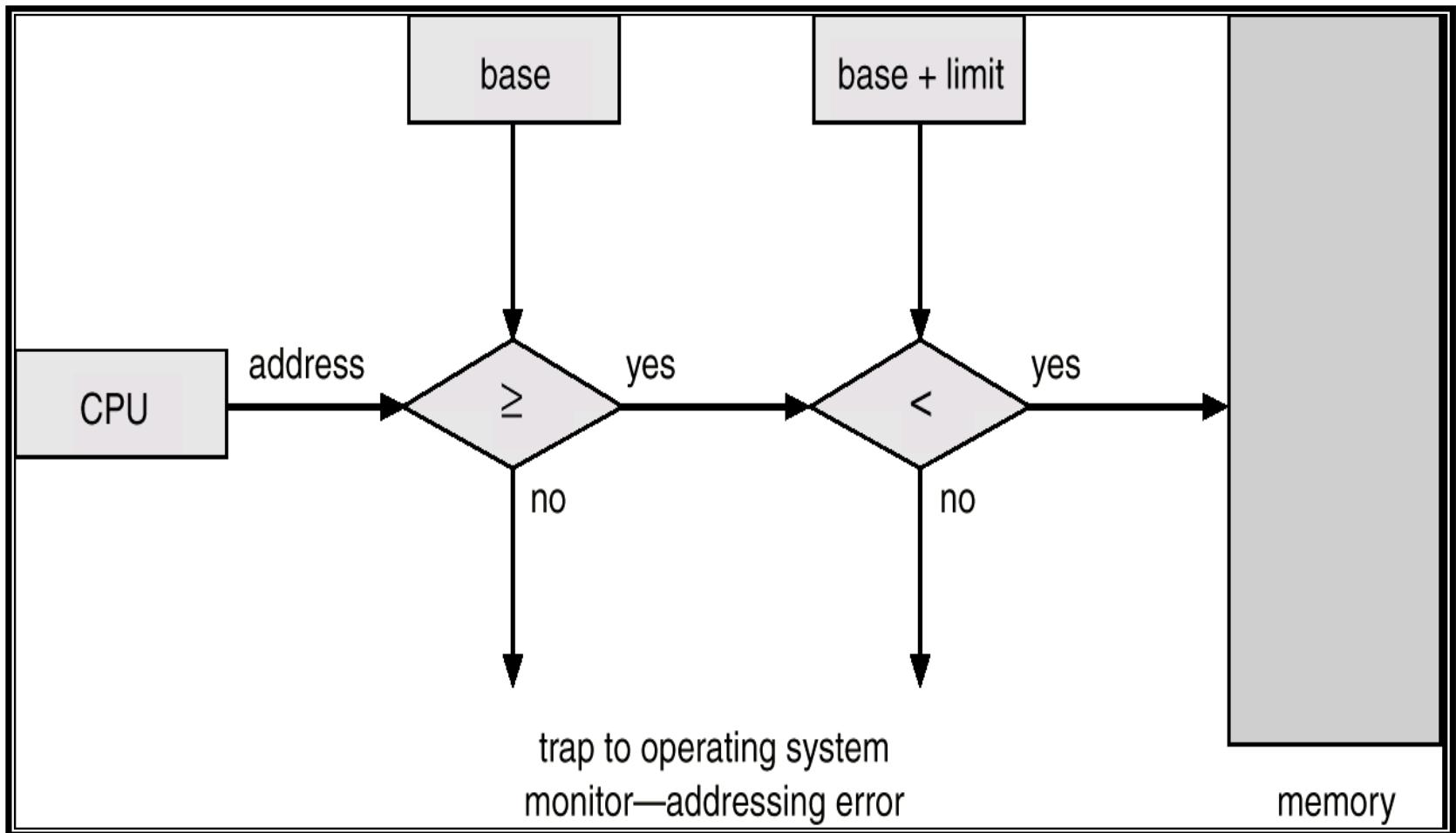


Memory Protection

- ▶ Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- ▶ In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
 - **Base register** – holds the smallest legal physical memory address.
 - **Limit register** – contains the size of the range
- ▶ Memory outside the defined range is protected.



Hardware Address Protection



Hardware Protection

- When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory.
- The load instructions for the *base* and *limit* registers are privileged instructions.

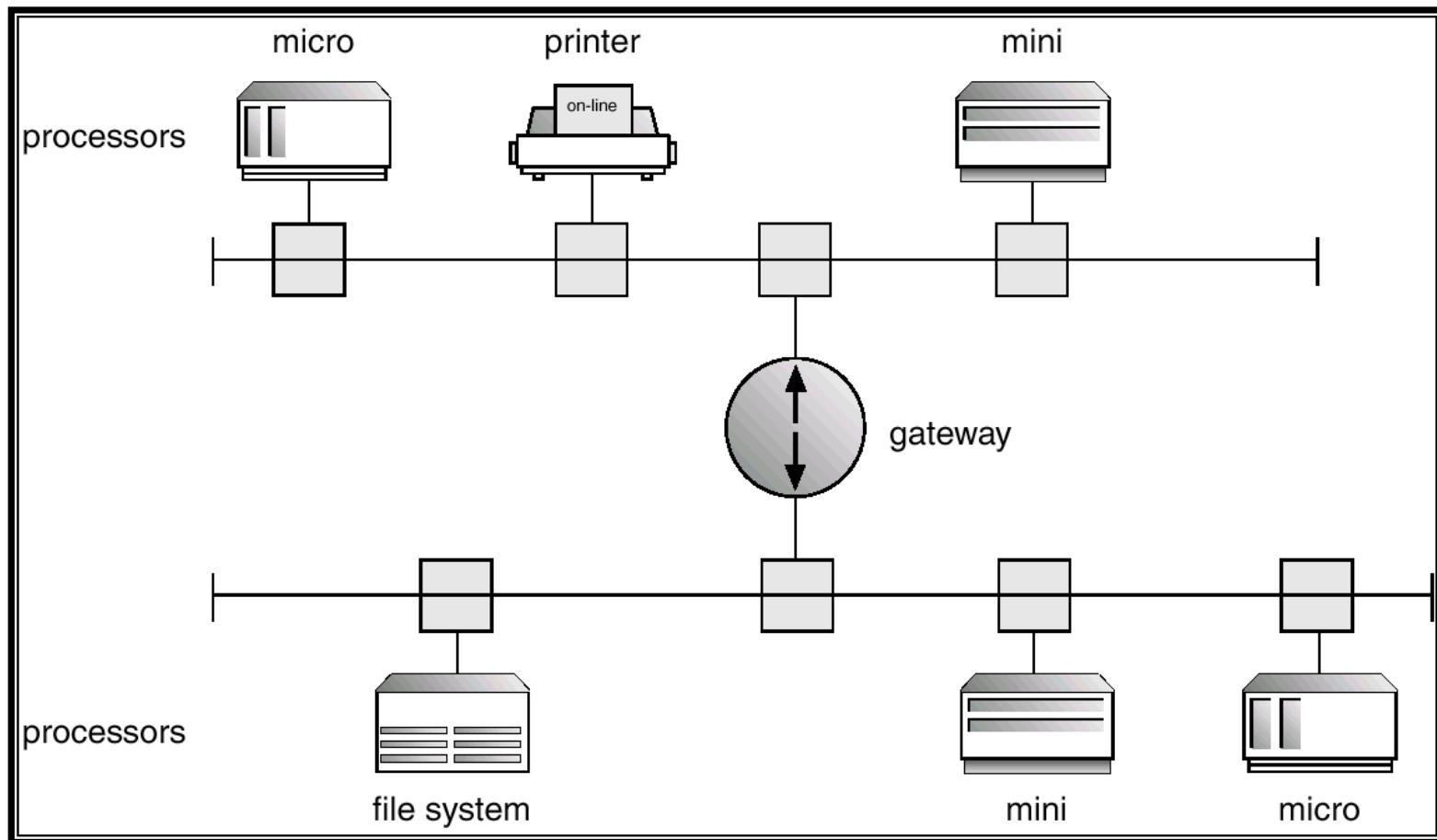
CPU Protection

- *Timer* – interrupts computer after specified period to ensure operating system maintains control.
 - Timer is decremented every clock tick.
 - When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing.
- Time also used to compute the current time.
- Load-timer is a privileged instruction.

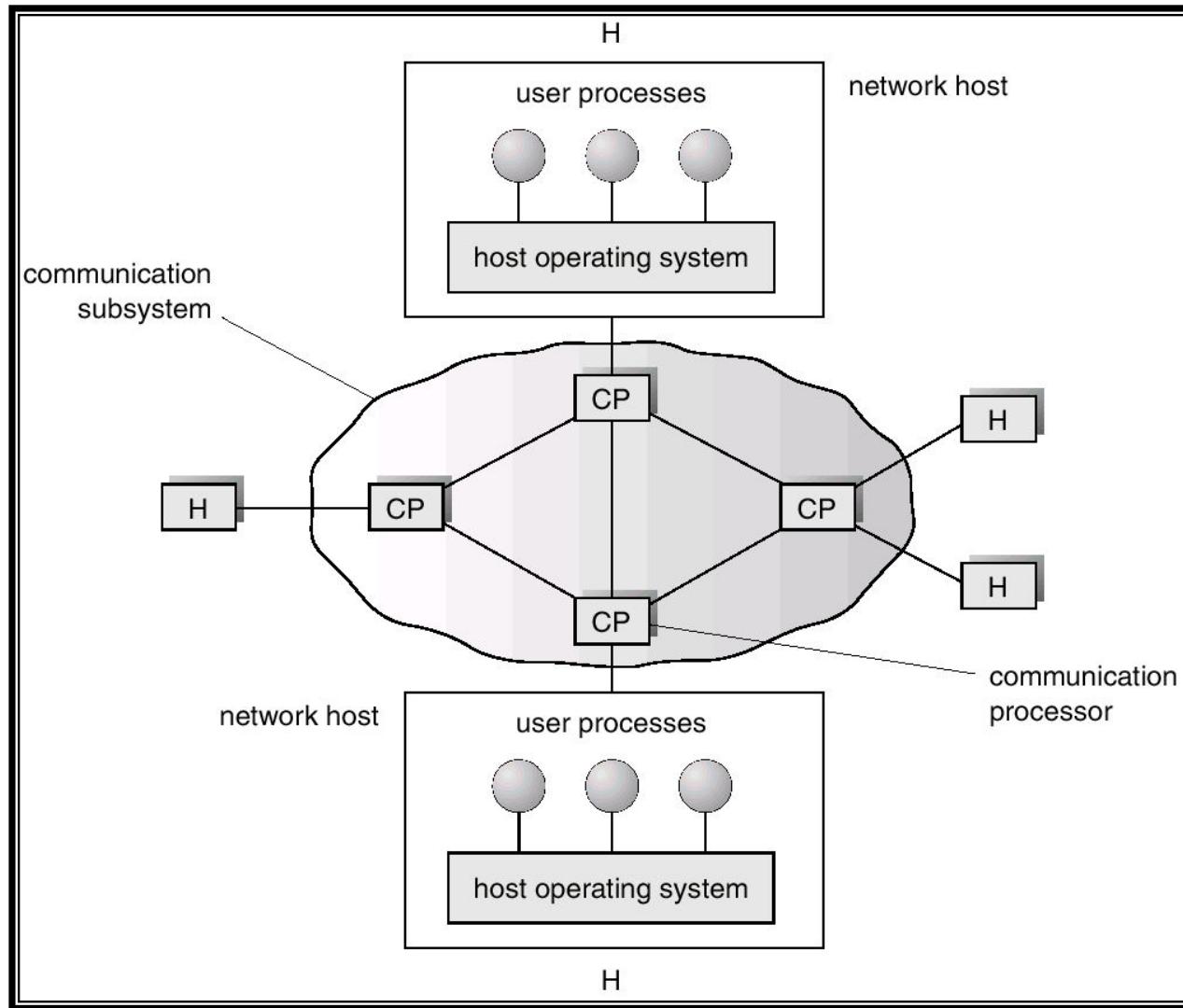
Network Structure

- Local Area Networks (LAN)
- Wide Area Networks (WAN)

Local Area Network Structure



Wide Area Network Structure



References

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9th Edition*

Thank You

OPERATING SYSTEMS

Module I: Introduction to Operating Systems

Operating-System Structures

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**

Course Learning Outcomes:

- learn basic concepts and responsibilities of operating system
- Analyze and evaluate various I/O component of computing system

Topics Covered

- System Components
- Operating System Services
- System Calls
- System Programs
- System Structure
- System Design and Implementation
- System Generation

Common System Components

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

Process Management

- A process is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.
- The operating system is responsible for the following activities in connection with process management.
 - Process creation and deletion.
 - process suspension and resumption.
 - Provision of mechanisms for:
 - process synchronization
 - process communication and
 - deadlock and handling

Main-Memory Management

- ▶ Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.
- ▶ Main memory is a volatile storage device. It loses its contents in the case of system failure.
- ▶ The operating system is responsible for the following activities in connections with memory management:
 - Keep track of which parts of memory are currently being used and by whom.
 - Decide which processes to load when memory space becomes available.
 - Allocate and deallocate memory space as needed.

File Management

- ▶ A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
- ▶ The operating system is responsible for the following activities in connections with file management:
 - File creation and deletion.
 - Directory creation and deletion.
 - Support of primitives for manipulating files and directories.
 - Mapping files onto secondary storage.
 - File backup on stable (nonvolatile) storage media.

I/O System Management

- The I/O system consists of:
 - A buffer-caching system
 - A general device-driver interface
 - Drivers for specific hardware devices

Secondary-Storage Management

- ▶ Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.
- ▶ Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
- ▶ The operating system is responsible for the following activities in connection with disk management:
 - Free space management
 - Storage allocation
 - Disk scheduling

- ▶ A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.
- ▶ The processors in the system are connected through a communication network.
- ▶ Communication takes place using a *protocol*.
- ▶ A distributed system provides user access to various system resources.
- ▶ Access to a shared resource allows:
 - Computation speed-up
 - Increased data availability
 - Enhanced reliability

Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
 - distinguish between authorized and unauthorized usage.
 - specify the controls to be imposed.
 - provide a means of enforcement.

Command-Interpreter System

- Many commands are given to the operating system by control statements which deal with:
 - process creation and management
 - I/O handling
 - secondary-storage management
 - main-memory management
 - file-system access
 - protection
 - networking

Command-Interpreter System (Cont.)



- The program that reads and interprets control statements is called variously:
 - command-line interpreter
 - shell (in UNIX)

Its function is to get and execute the next command statement.

Operating System Services

- Program execution – system capability to load a program into memory and to run it.
- I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
- File-system manipulation – program capability to read, write, create, and delete files.
- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.
- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

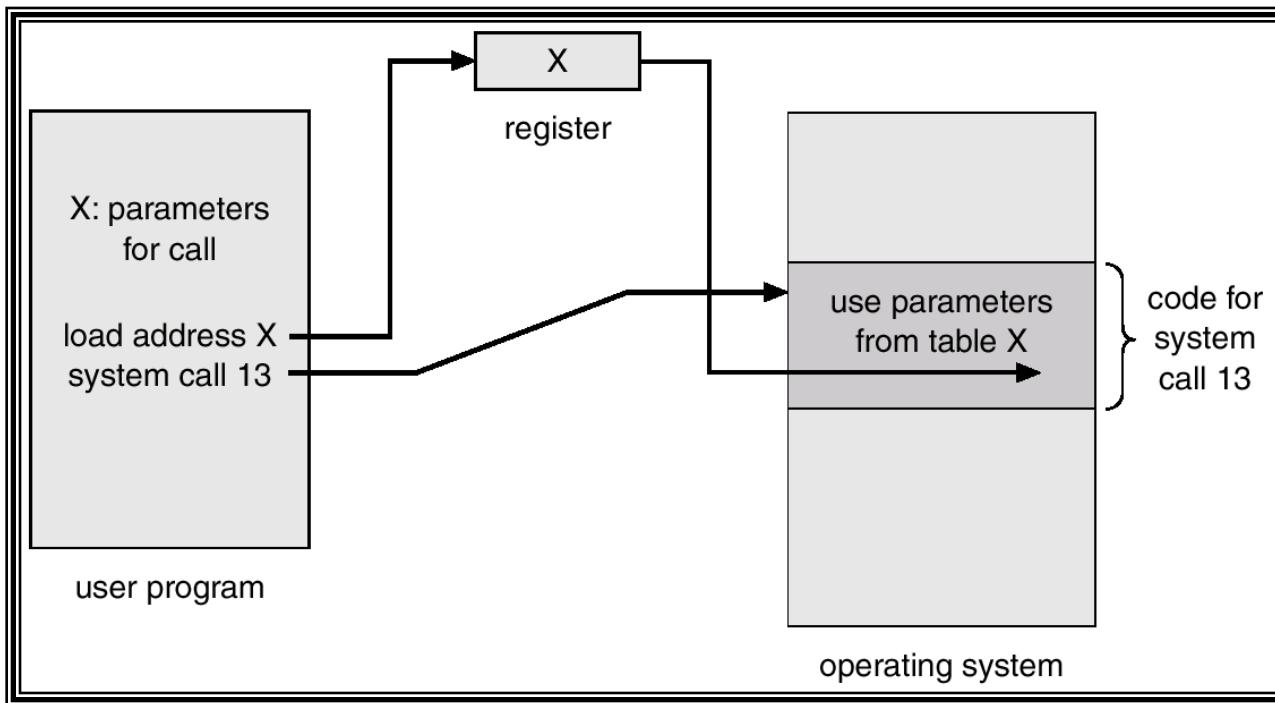
- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.
- Accounting – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- Protection – ensuring that all access to system resources is controlled.

System Calls

- ▶ System calls provide the interface between a running program and the operating system.
 - Generally available as assembly-language instructions.
 - Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)

- ▶ Three general methods are used to pass parameters between a running program and the operating system.
 - Pass parameters in *registers*.
 - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
 - *Push* (store) the parameters onto the *stack* by the program and *pop* off the stack by operating system.

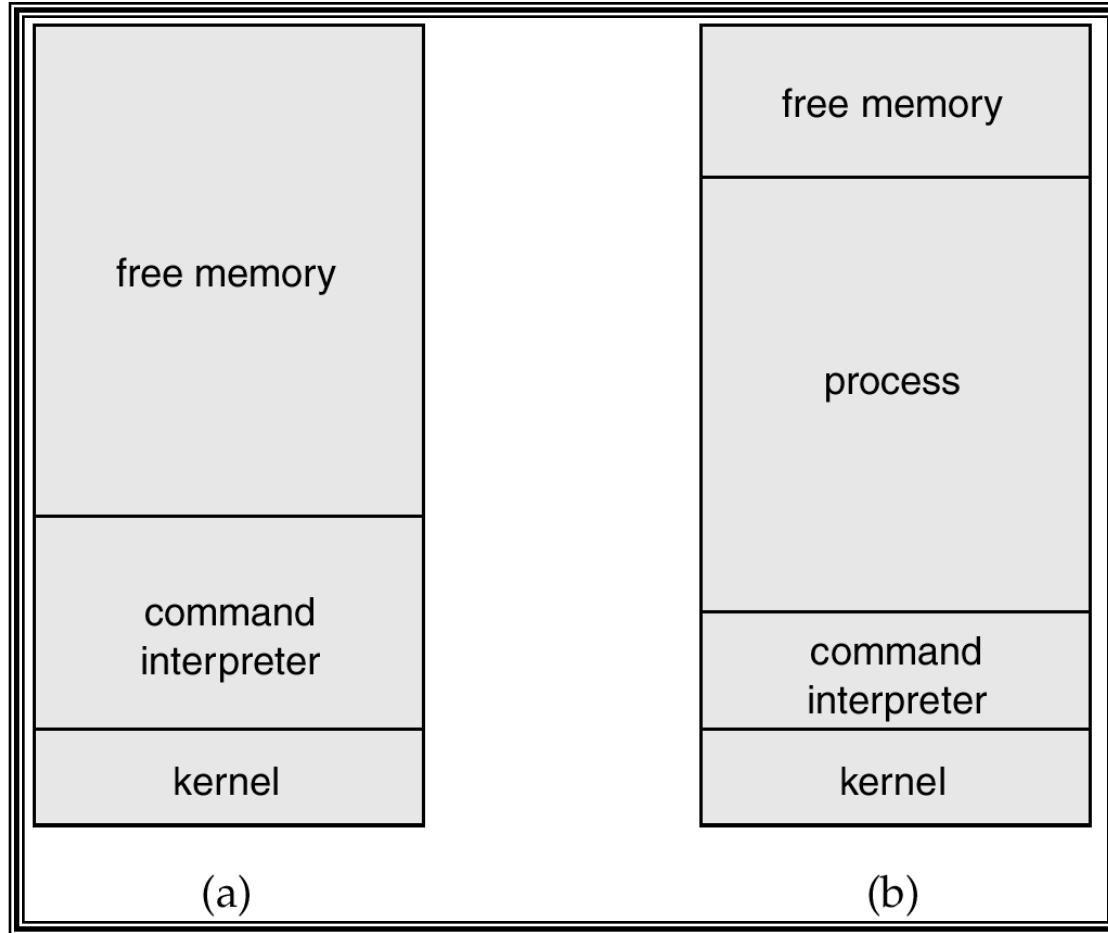
Passing of Parameters As A Table



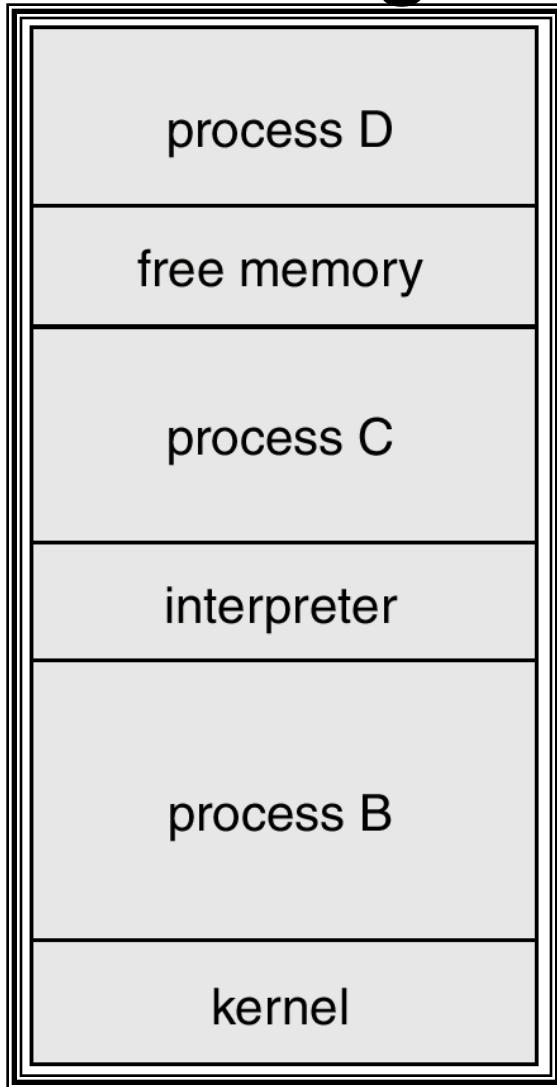
Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

MS-DOS Execution

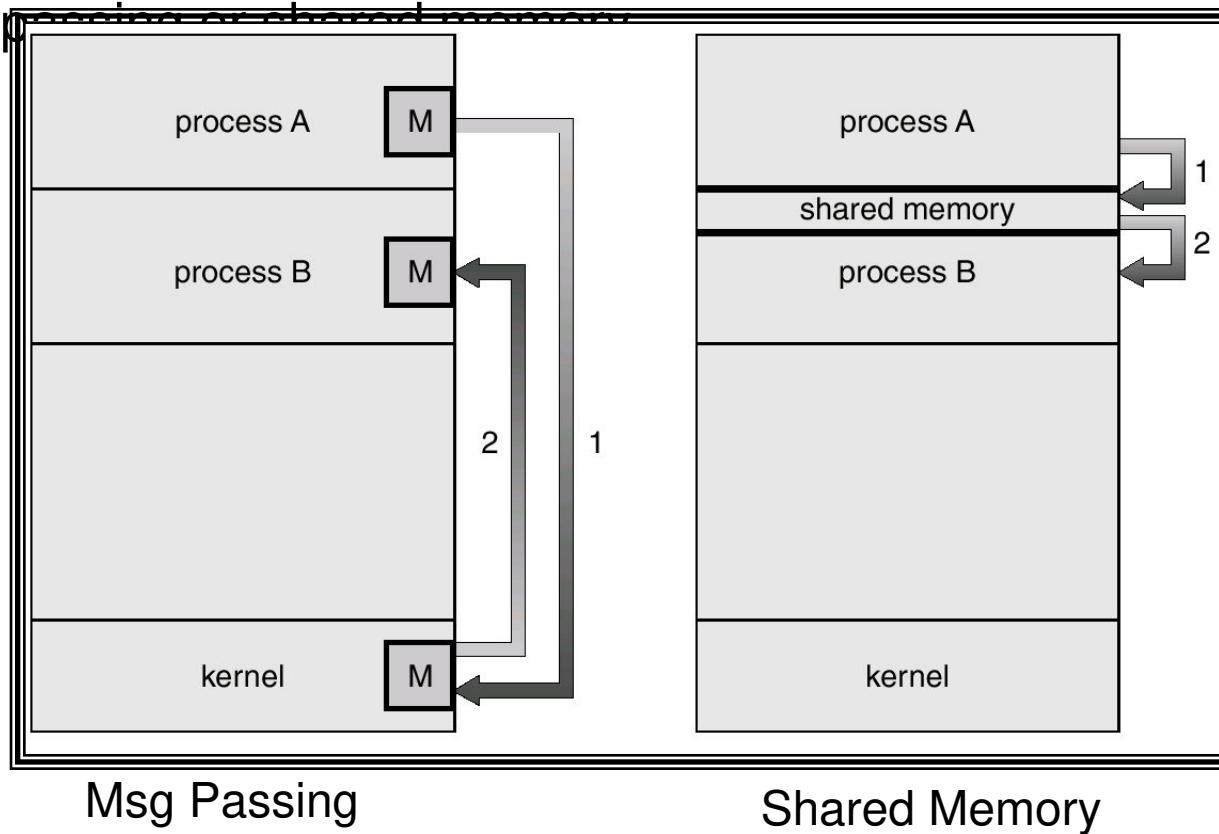


UNIX Running Multiple Programs



Communication Models

- Communication may take place using either message



System Programs

- ▶ System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- ▶ Most users' view of the operation system is defined by system programs, not the actual system calls.

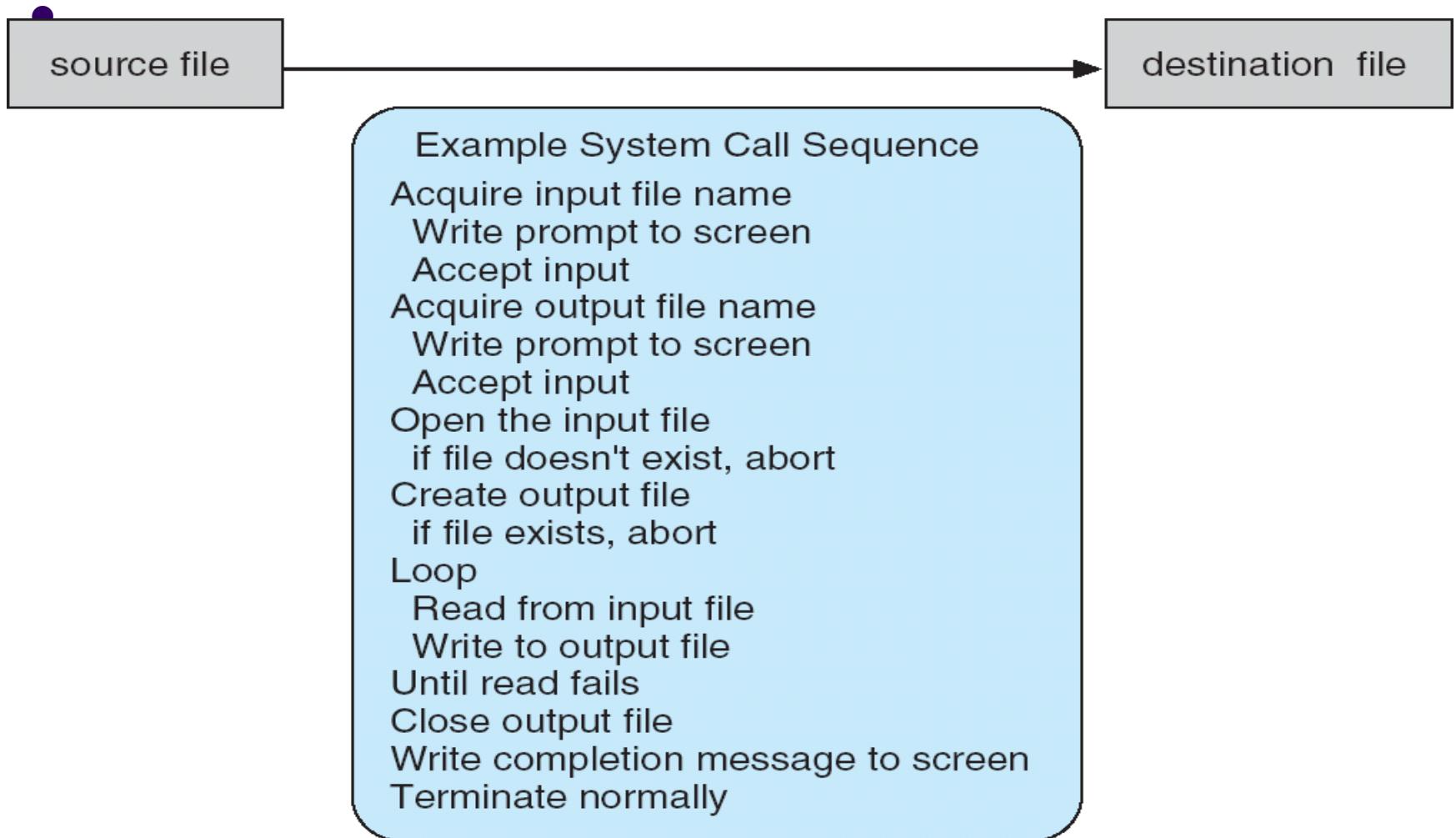
System Call



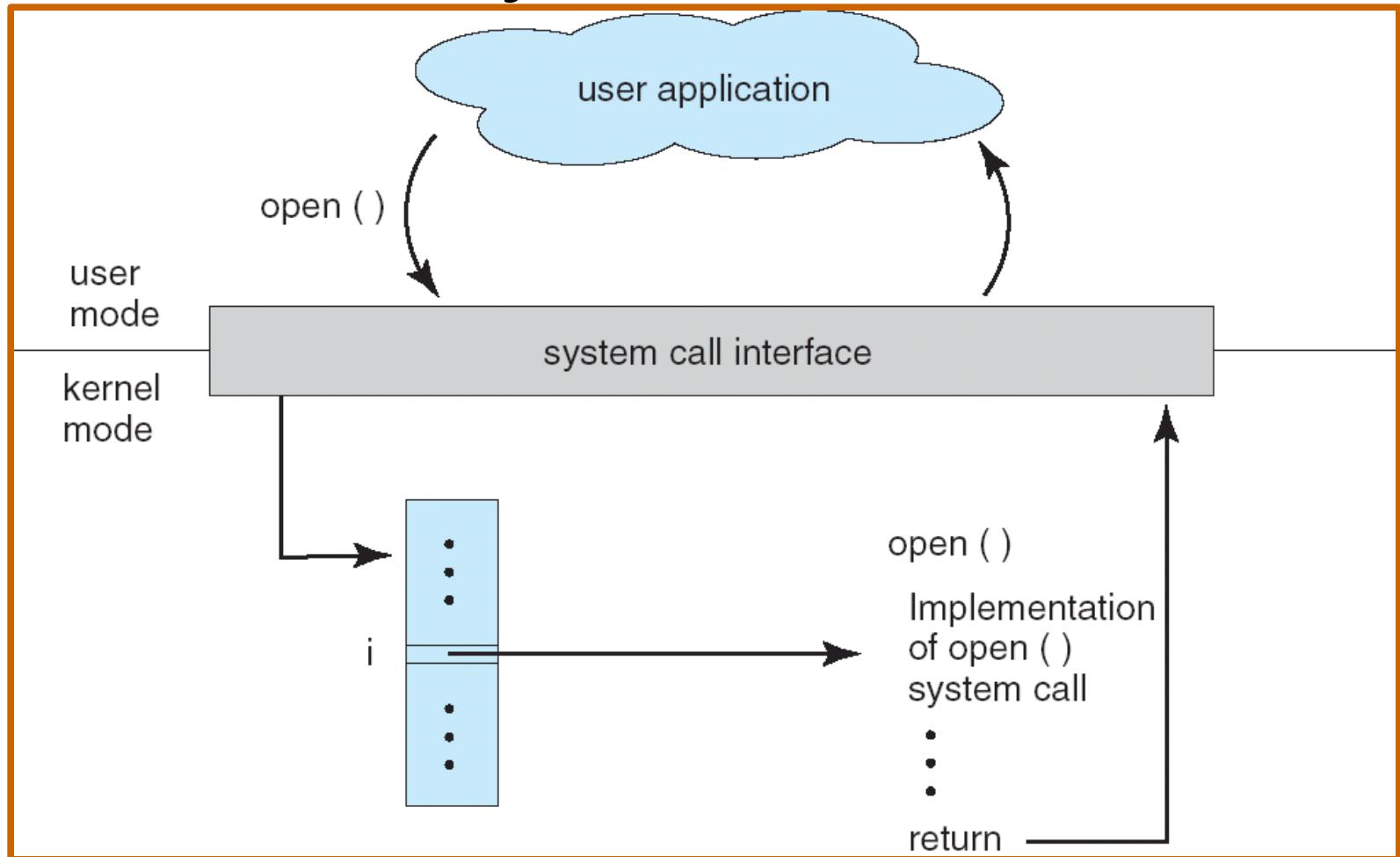
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

Example how System Calls are used

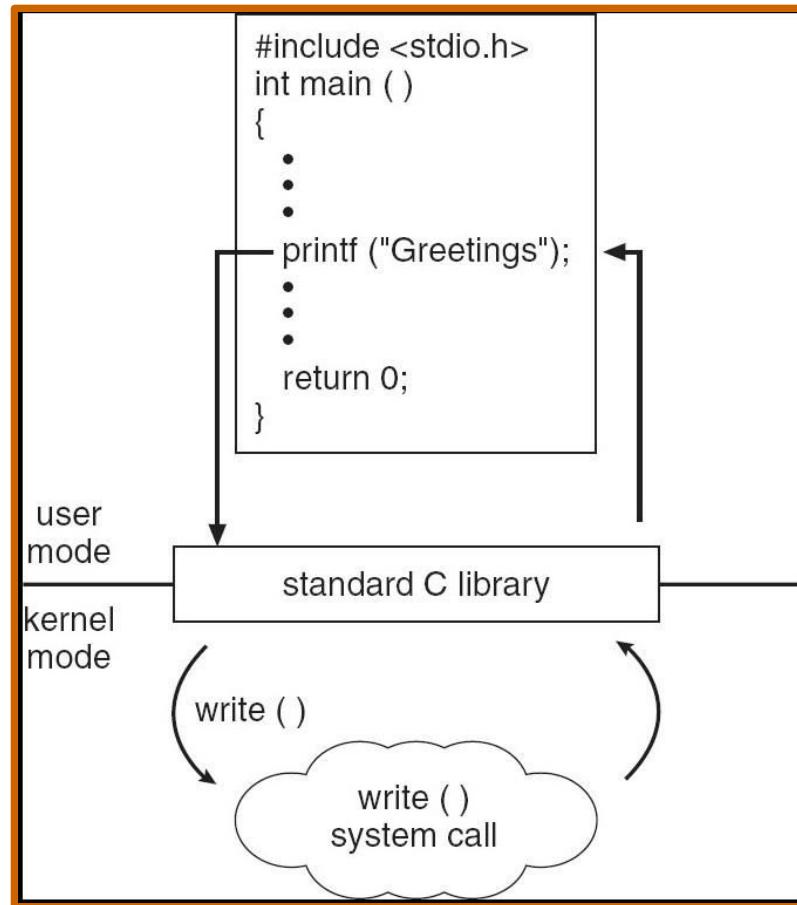


API – System Call – OS



Standard C Library Example

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

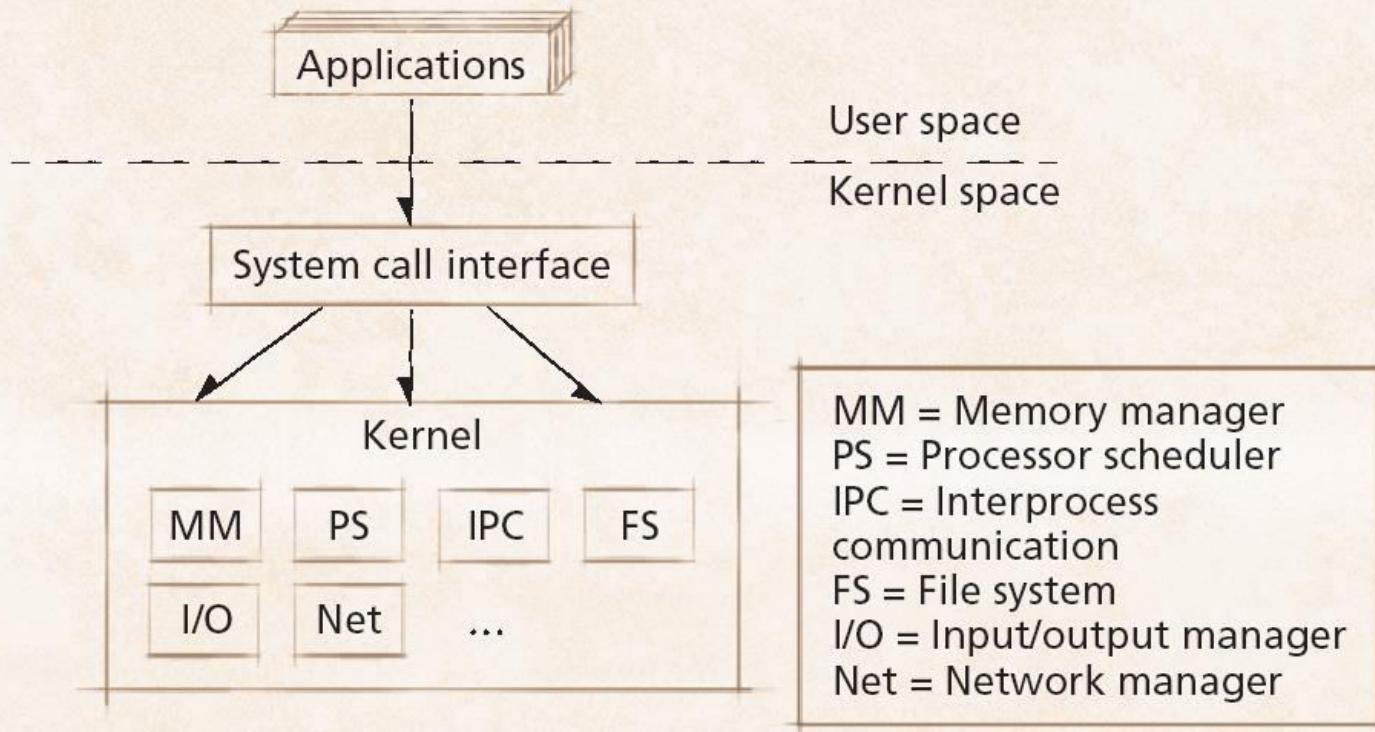


Operating System Structure

- monolithic
- layered
- micro-kernel
- Modular

- Monolithic operating system
 - Every component contained in kernel
 - direct communication among all elements
 - highly efficient
 - Problems:
 - complexity
 - new devices, emerging technologies
 - enabling, protection

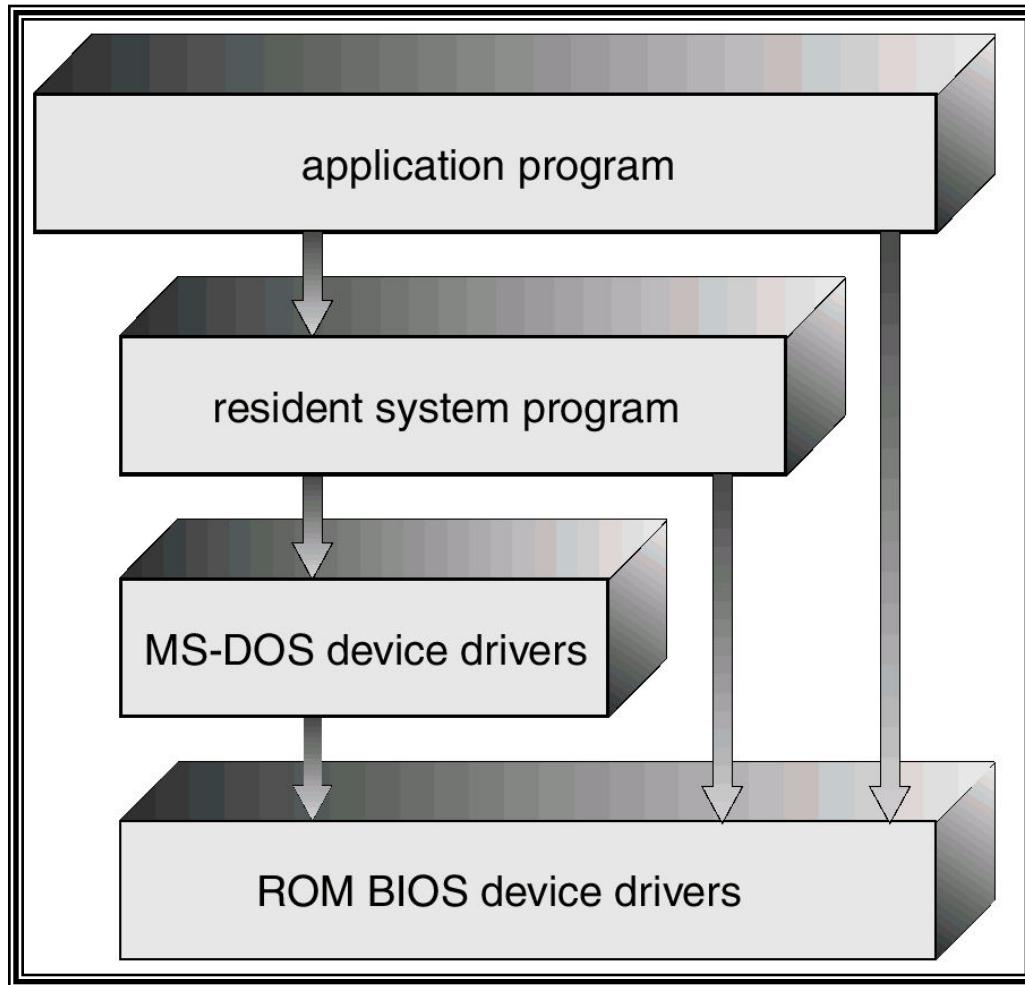
Monolithic Architecture



MS-DOS System Structure

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

MS-DOS Layer Structure



UNIX System Structure

- ▶ UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.
 - Systems programs
 - The kernel
 - Consists of everything below the system-call interface and above the physical hardware
 - Provides the file system, CPU scheduling, memory management, and other operating-system functions; many functions for one level.

UNIX System Structure

(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal
handling
character I/O system
terminal drivers

file system
swapping block I/O
system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

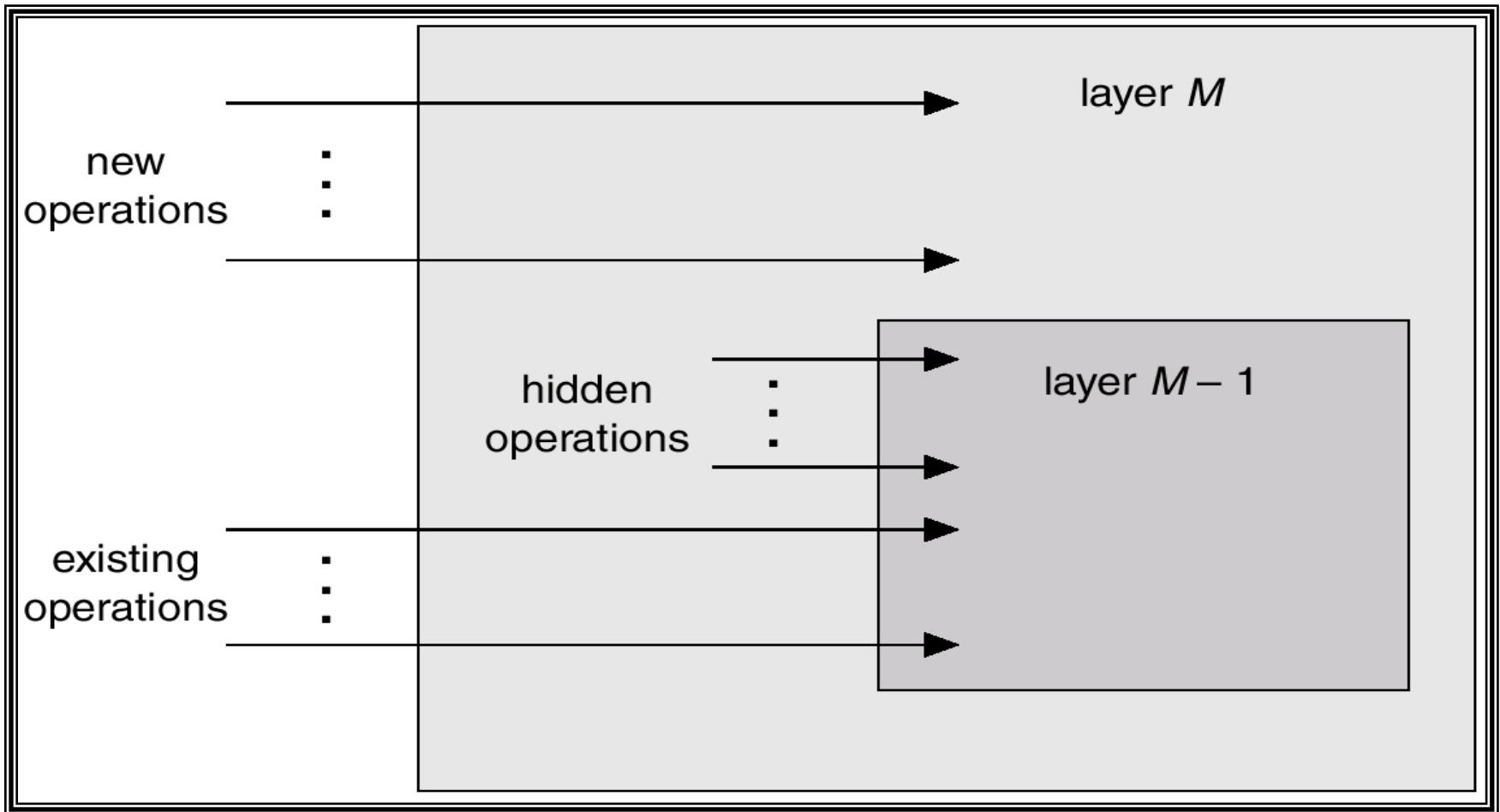
device controllers
disks and tapes

memory controllers
physical memory

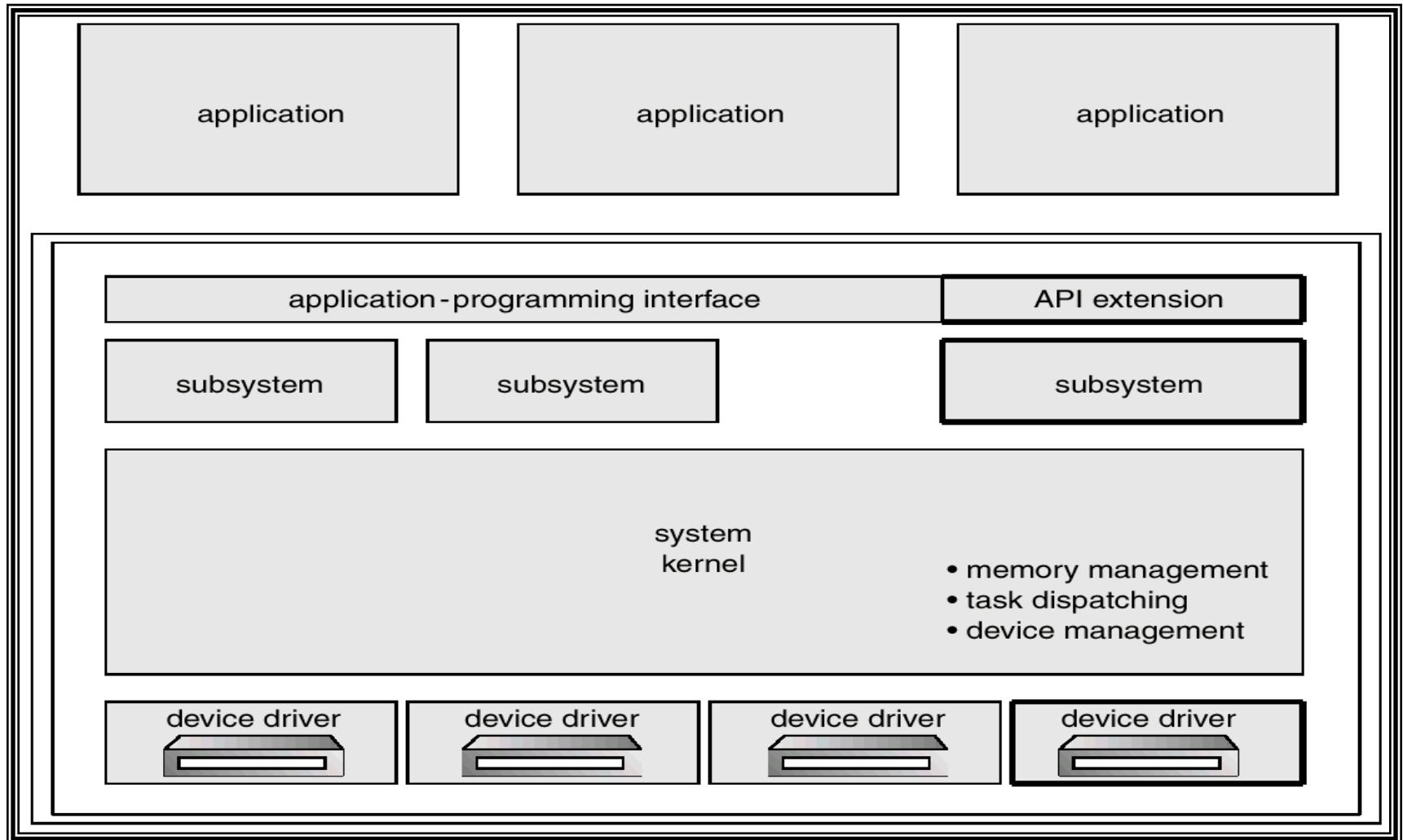
Layered Approach

- The operating system is divided into several 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.

An Operating System Layer

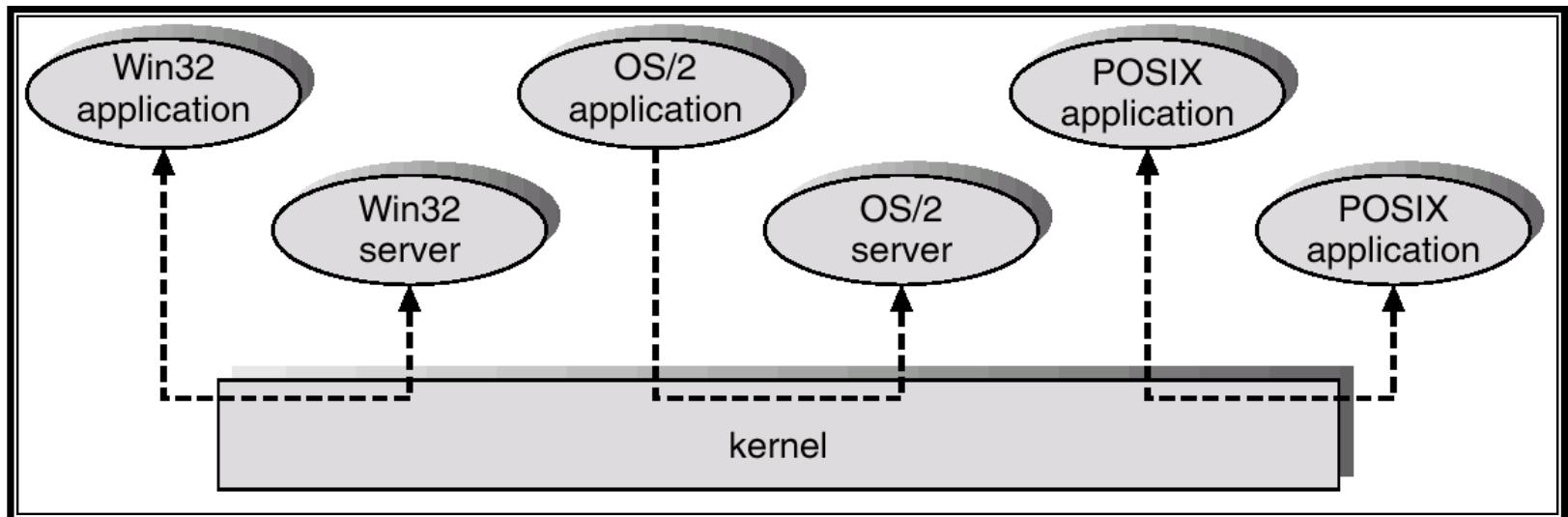


OS/2 Layer Structure



- ▶ Moves as much from the kernel into “*user*” space.
- ▶ Communication takes place between user modules using message passing.
- ▶ Benefits:
 - easier to extend a microkernel
 - easier to port the operating system to new architectures
 - more reliable (less code is running in kernel mode)
 - more secure

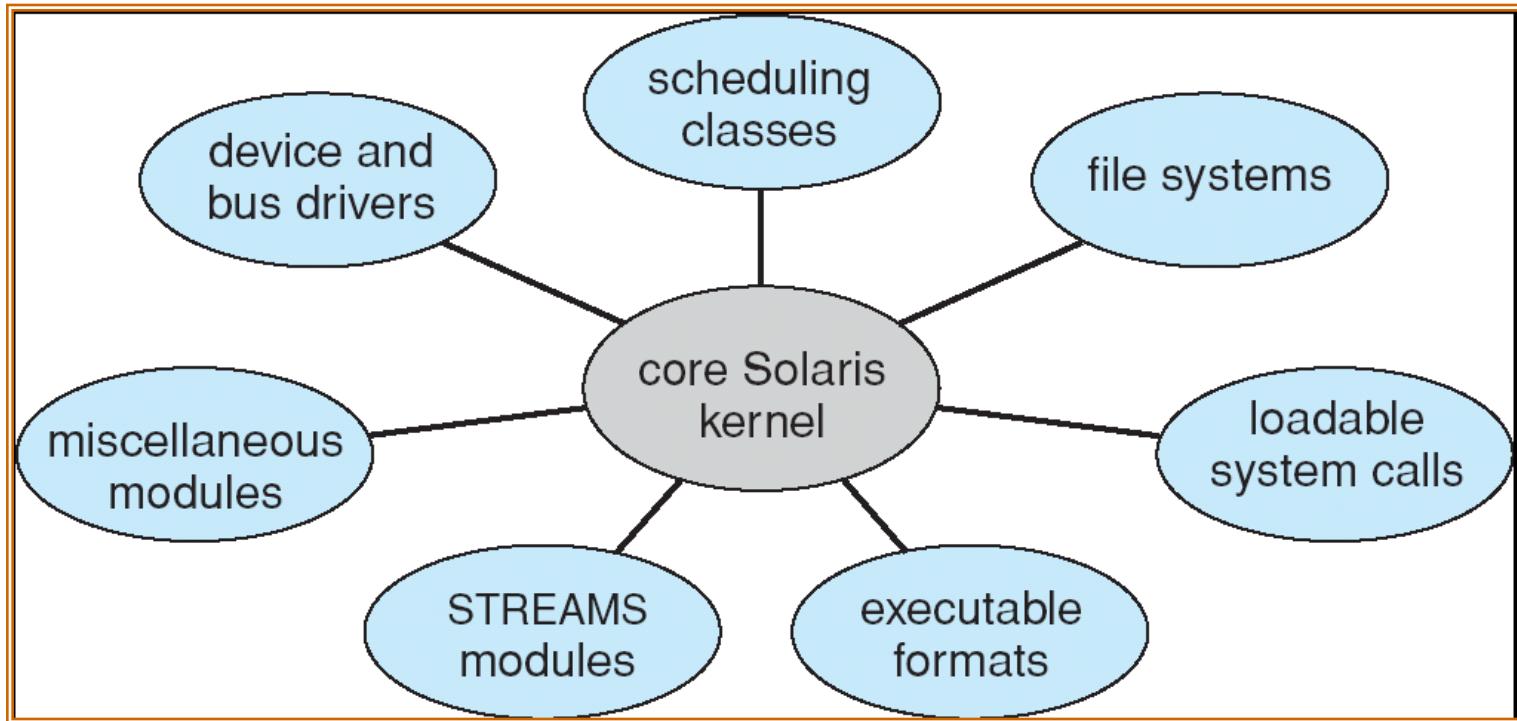
Windows NT Client-Server Structure



Modules

- Most modern operating systems implement kernel modules
 - Uses object-oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, like layers but with more flexible

Solaris Modular Approach



System Design Goals

- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

Mechanisms and Policies

- Mechanisms determine how to do something; policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.

System Implementation

- Traditionally written in assembly language, operating systems can now be written in higher-level languages.
- Code written in a high-level language:
 - can be written faster.
 - is more compact.
 - is easier to understand and debug.
- An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

System Generation (SYSGEN)

- ▶ Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- ▶ SYSGEN program obtains information concerning the specific configuration of the hardware system.
 - ▶ What CPU is to be used?
 - ▶ How much memory is available?
 - ▶ What devices are available?
 - ▶ What operating system options are desired or what parameter values are to be used?
- ▶ *Booting* – starting a computer by loading the kernel.
- ▶ *Bootstrap program* – code stored in ROM that can locate the kernel, load it into memory, and start its execution.

References

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9th Edition*

Thank You

CSE-202 OPERATING SYSTEM

Module II: Process Management

Process Concept

Course Learning Objectives:

- To understand the notion of a process -- a program in execution, which forms the basis of all computation

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**



Module II: Process Management

Process concept, State model, process scheduling, job and process synchronization, structure of process management, Threads.

Interprocess Communication and Synchronization: Principle of Concurrency, Producer Consumer Problem, Critical Section problem, Semaphores, Hardware Synchronization, Critical Regions, Conditional critical region, Monitor, Inter Process Communication.

CPU Scheduling: Job scheduling functions, Process scheduling, Scheduling Algorithms, Non Preemptive and preemptive Strategies, Algorithm Evaluation, Multiprocessor Scheduling.

Deadlock: System Deadlock Model, Deadlock Characterization, Methods for handling deadlock, Prevention strategies, Avoidance and Detection, Recovery from deadlock combined approach.

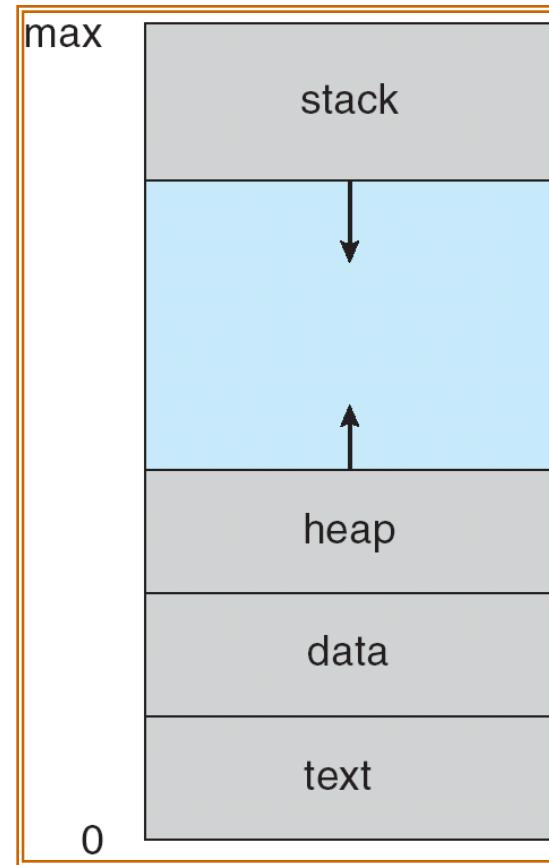
Topics to be Covered

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes

Process Concept

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
 - program counter
 - stack
 - data section

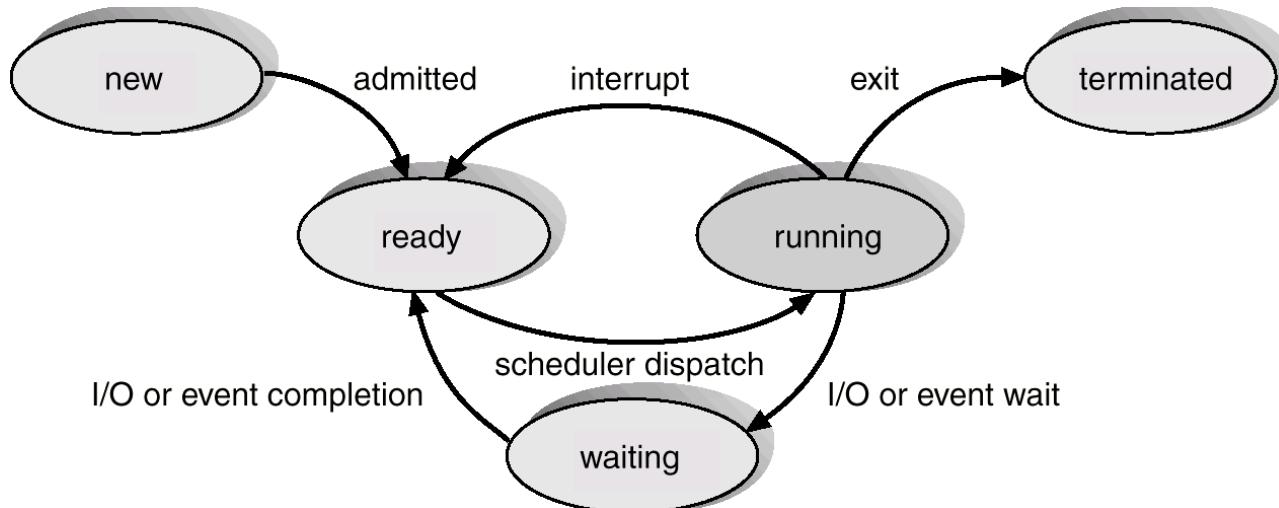
Process in Memory



Process State

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

Diagram of Process State

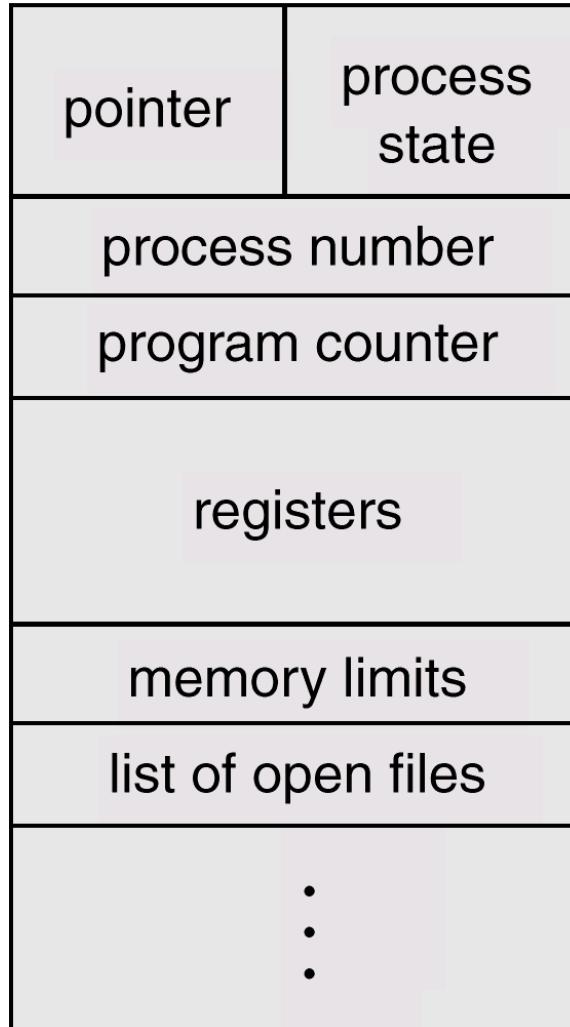


Process Control Block (PCB)

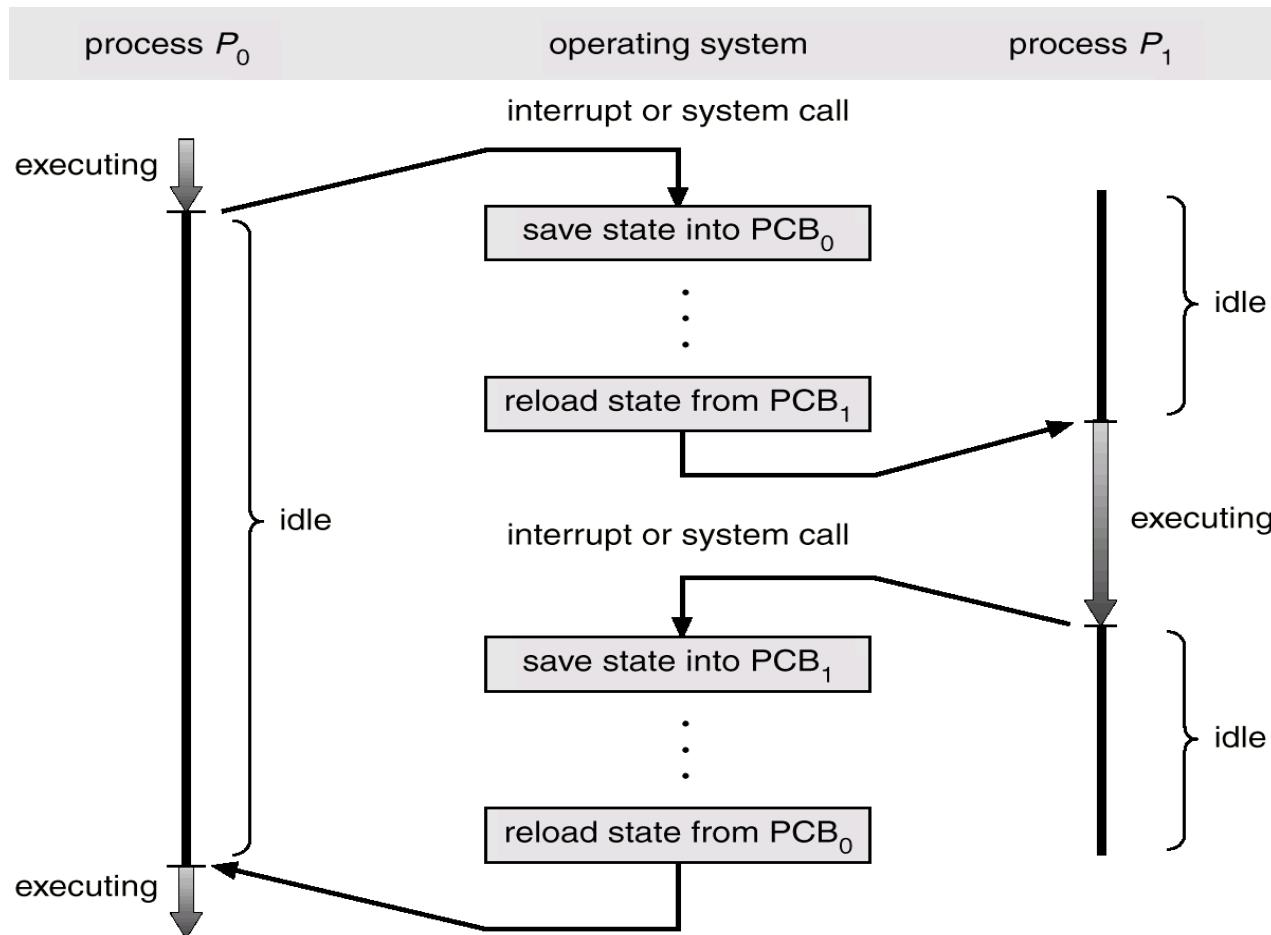
Information associated with each process.

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

Process Control Block (PCB)



CPU Switch From Process to Process

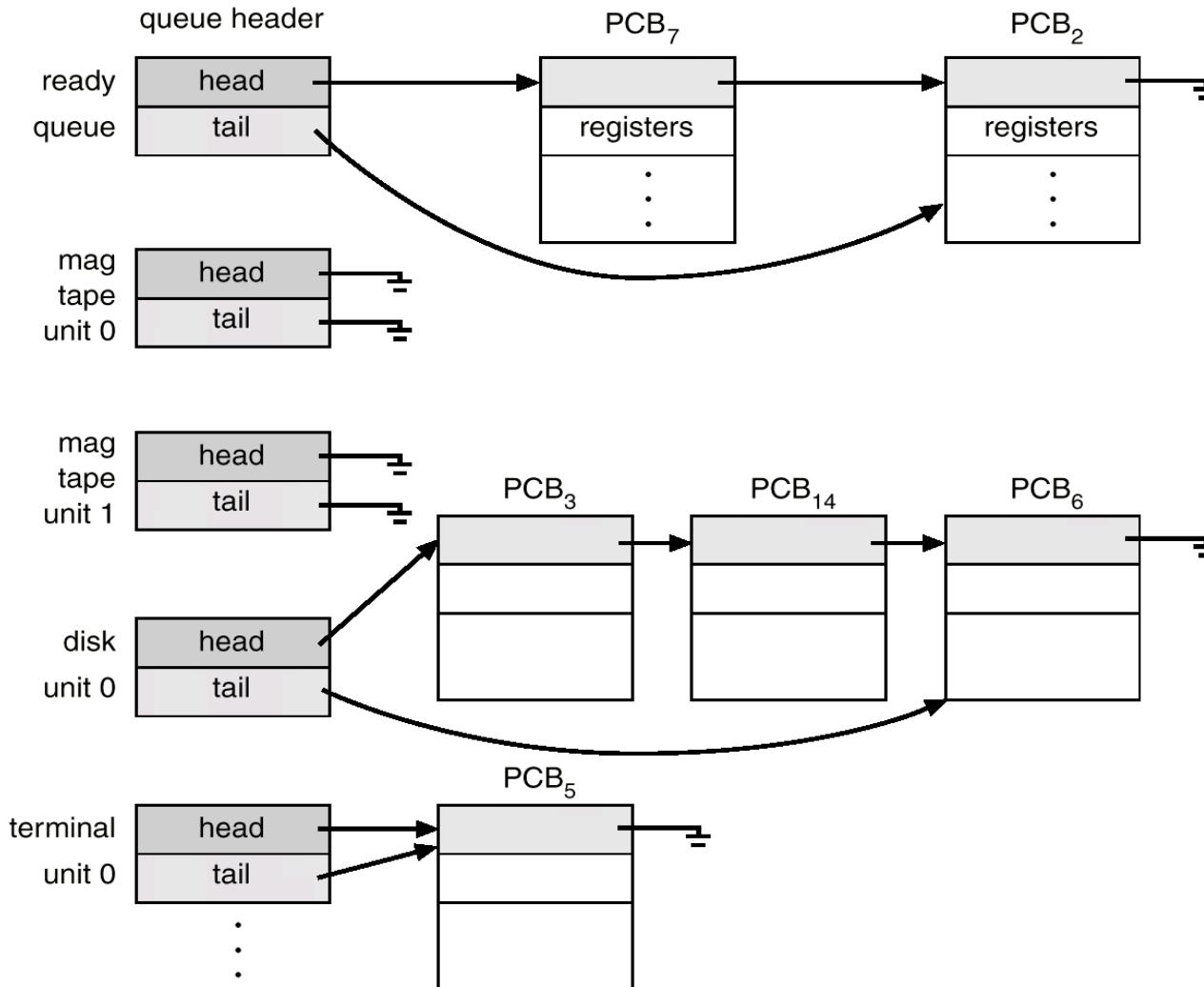


Process Scheduling Queues

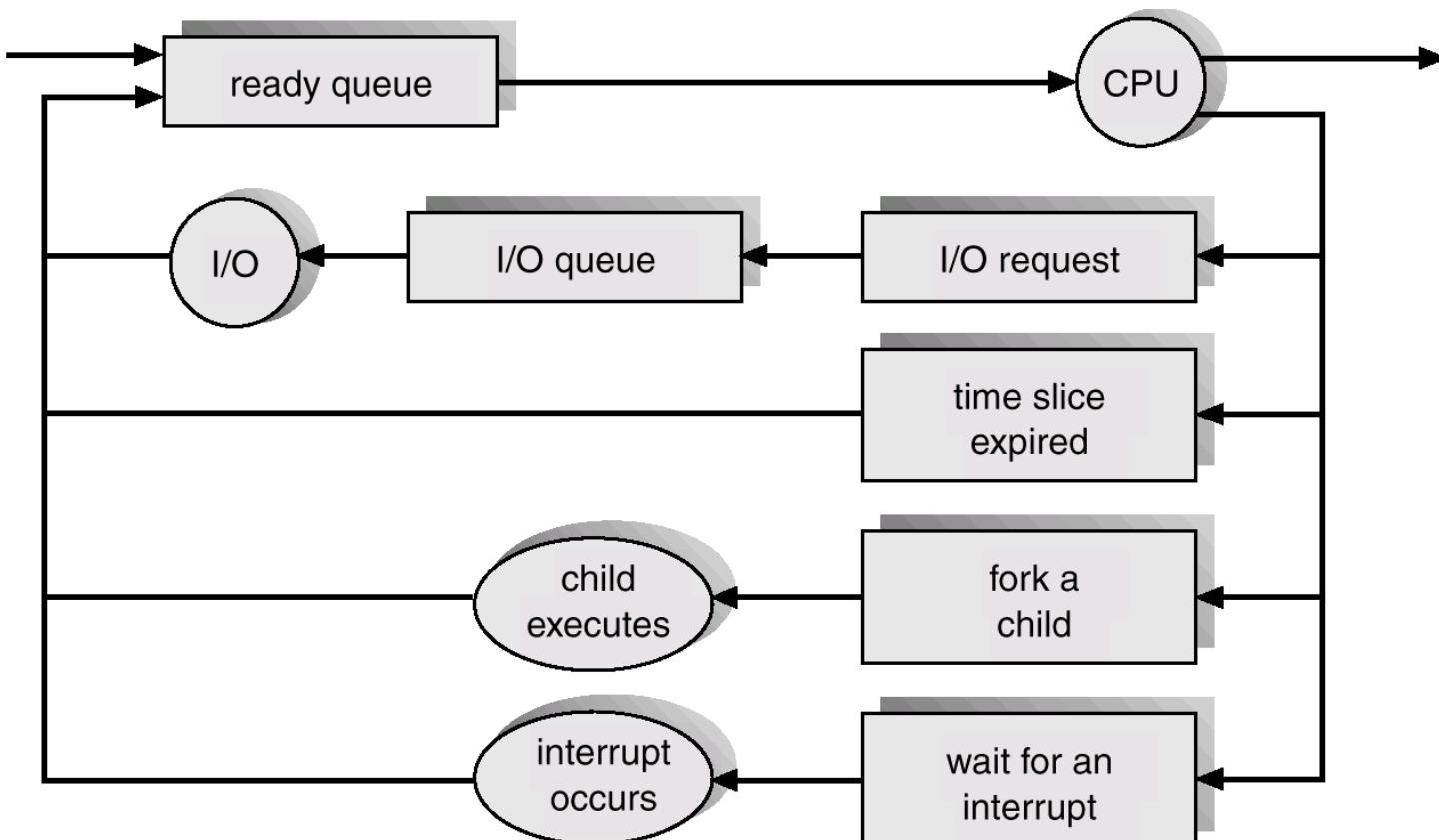
- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.

Ready Queue And Various I/O

Device Queues



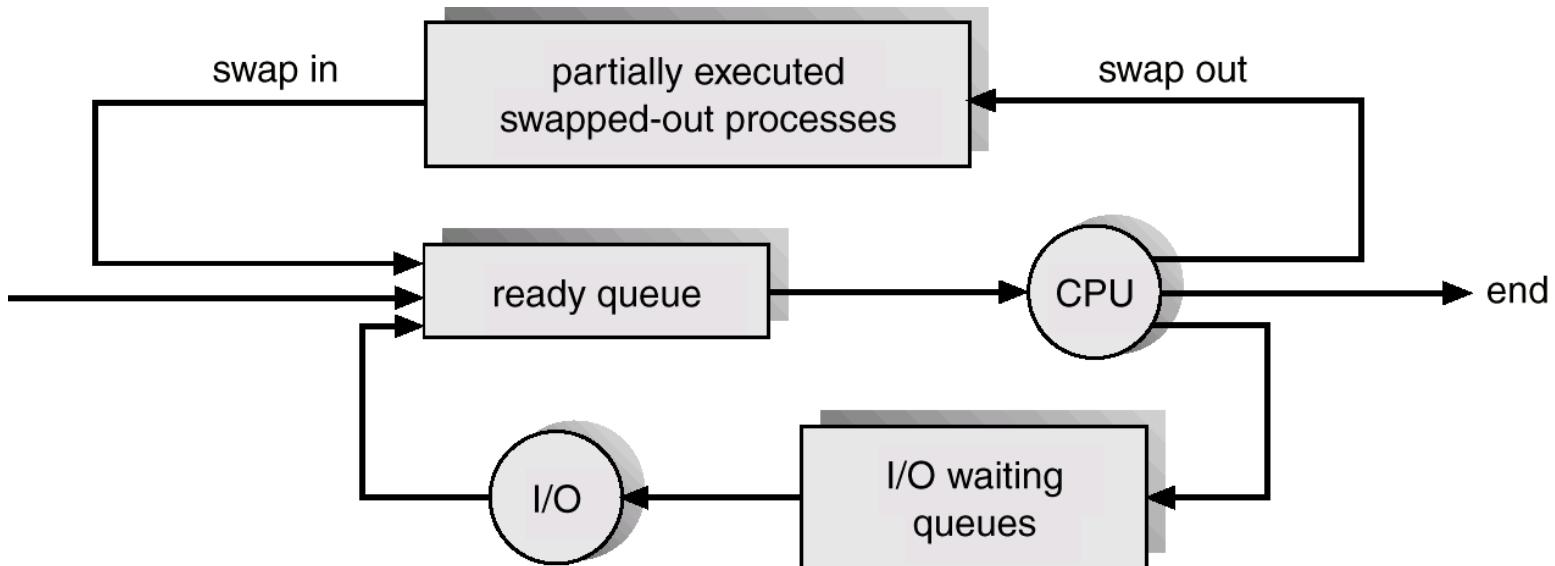
Representation of Process Scheduling



Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

Addition of Medium Term Scheduling



Schedulers (Cont.)

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

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
 - Parent and children share all resources.
 - Children share subset of parent's resources.
 - Parent and child share no resources.
- Execution
 - Parent and children execute concurrently.
 - Parent waits until children terminate.

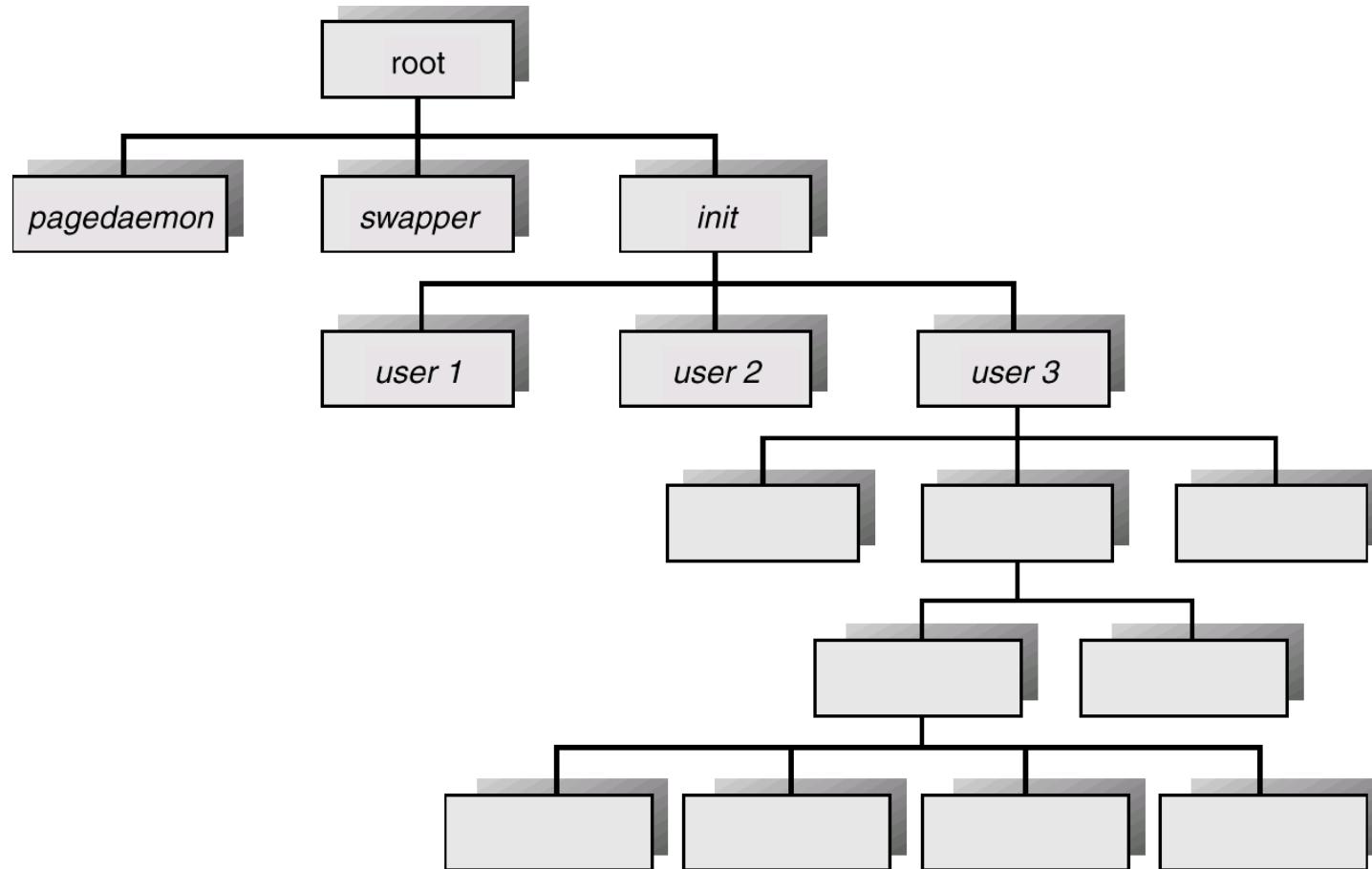
Process Creation (Cont.)

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

C Program Forking Separate Process

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

A Tree of Processes On A Typical UNIX System



Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
 - Output data from child to parent (via **wait**).
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting.
 - Operating system does not allow child to continue if its parent terminates.
 - Cascading termination.

Cooperating Processes

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

References

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9th Edition*

Thank You

CSE-202 OPERATING SYSTEM

Module II: Process Management

Process Synchronization

Course Learning Objectives:

- To understand the notion of a process -- a program in execution, which forms the basis of all computation

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**

Topics to be Covered

- Cooperating Processes
- Threads
- RPC



Module II: Process Management

Process concept, State model, process scheduling, job and process synchronization, structure of process management, Threads.

Interprocess Communication and Synchronization: Principle of Concurrency, Producer Consumer Problem, Critical Section problem, Semaphores, Hardware Synchronization, Critical Regions, Conditional critical region, Monitor, Inter Process Communication.

CPU Scheduling: Job scheduling functions, Process scheduling, Scheduling Algorithms, Non Preemptive and preemptive Strategies, Algorithm Evaluation, Multiprocessor Scheduling.

Deadlock: System Deadlock Model, Deadlock Characterization, Methods for handling deadlock, Prevention strategies, Avoidance and Detection, Recovery from deadlock combined approach.

Process Creation

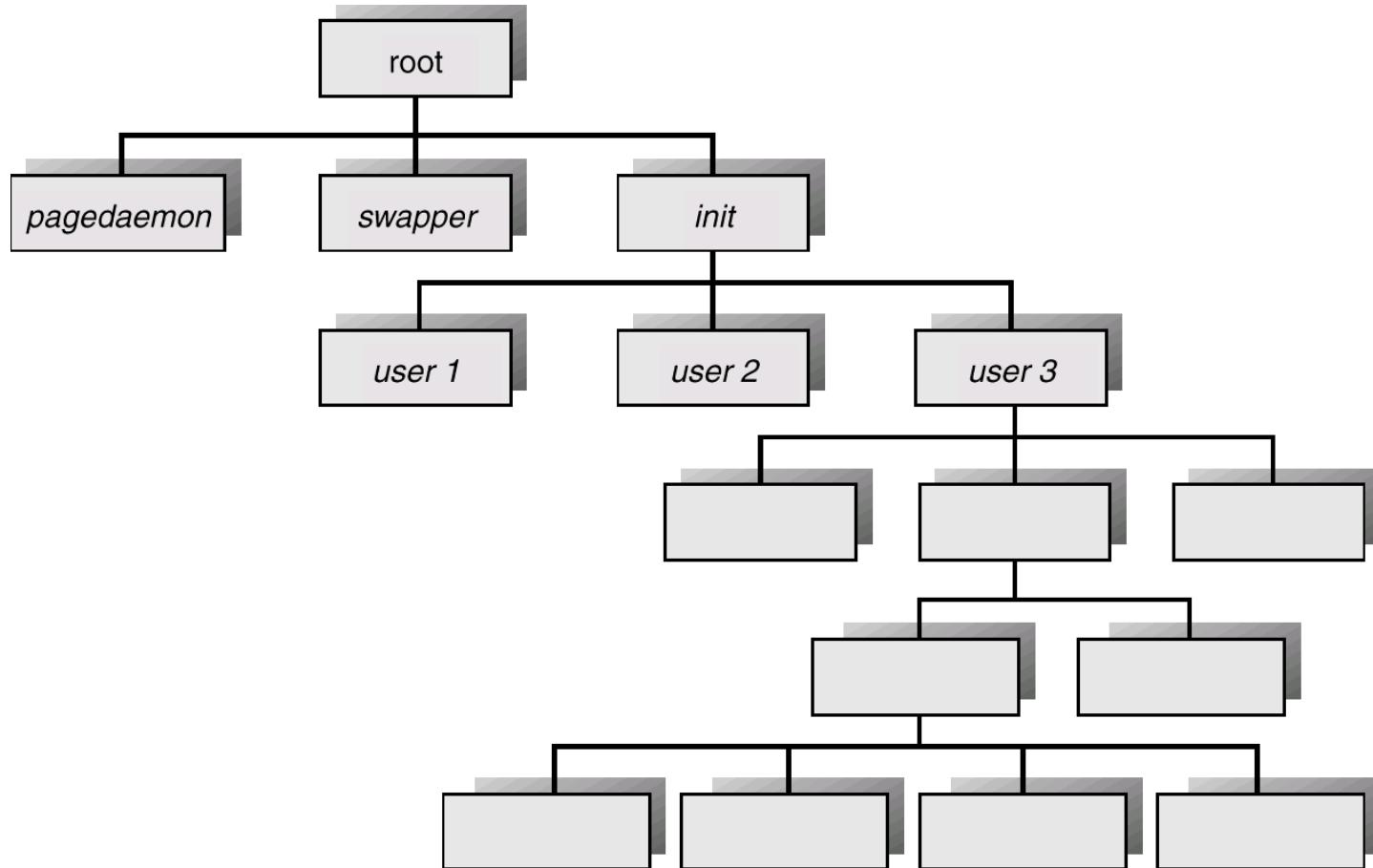
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
 - Parent and children share all resources.
 - Children share subset of parent's resources.
 - Parent and child share no resources.
- Execution
 - Parent and children execute concurrently.
 - Parent waits until children terminate.

Process Creation (Cont.)

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



A Tree of Processes On A Typical UNIX System



Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
 - Output data from child to parent (via **wait**).
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting.
 - Operating system does not allow child to continue if its parent terminates.
 - Cascading termination.

Cooperating Processes

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

Producer-Consumer Problem

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

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
var n;  
  
type item = ... ;  
  
var buffer. array [0..n-1] of item;  
in, out: 0..n-1;
```

- Producer process

```
repeat  
  ...  
  produce an item in nextp  
  ...  
  while in+1 mod n = out do no-op;  
  buffer [in] :=nextp;  
  in :=in+1 mod n;  
until false;
```

Bounded-Buffer (Cont.)

- Consumer process

```
repeat
    while in = out do no-op;
    nextc := buffer [out];
    out := out+1 mod n;
    ...
    consume the item in nextc
    ...
until false;
```

- Solution is correct, but can only fill up $n-1$ buffer.

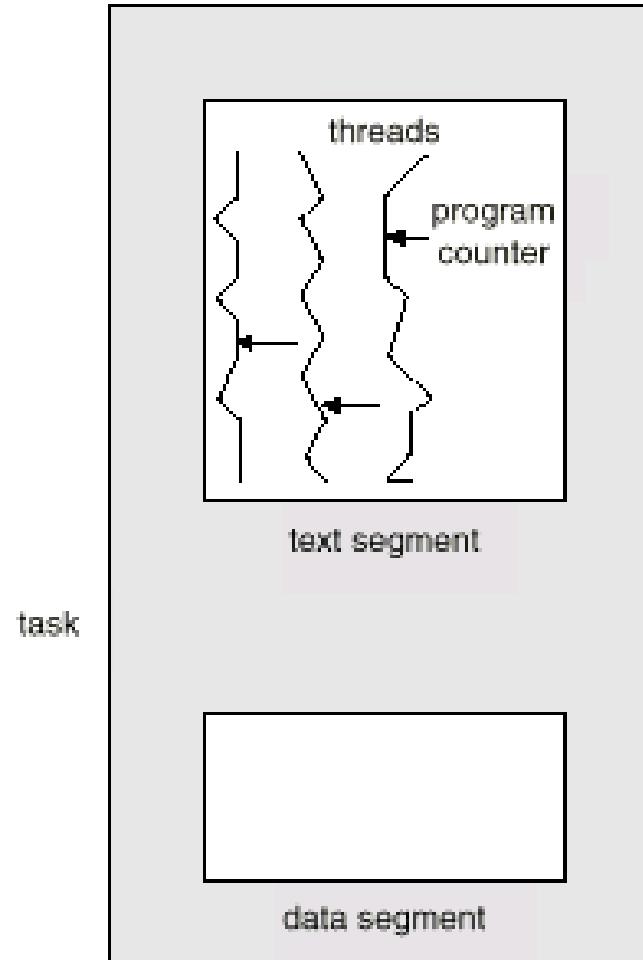
Threads

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
 - program counter
 - register set
 - stack space
- A thread shares with its peer threads its:
 - code section
 - data section
 - operating-system resourcescollectively known as a *task*.
- A traditional or *heavyweight* process is equal to a task with one thread

Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
 - Cooperation of multiple threads in same job confers higher throughput and improved performance.
 - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

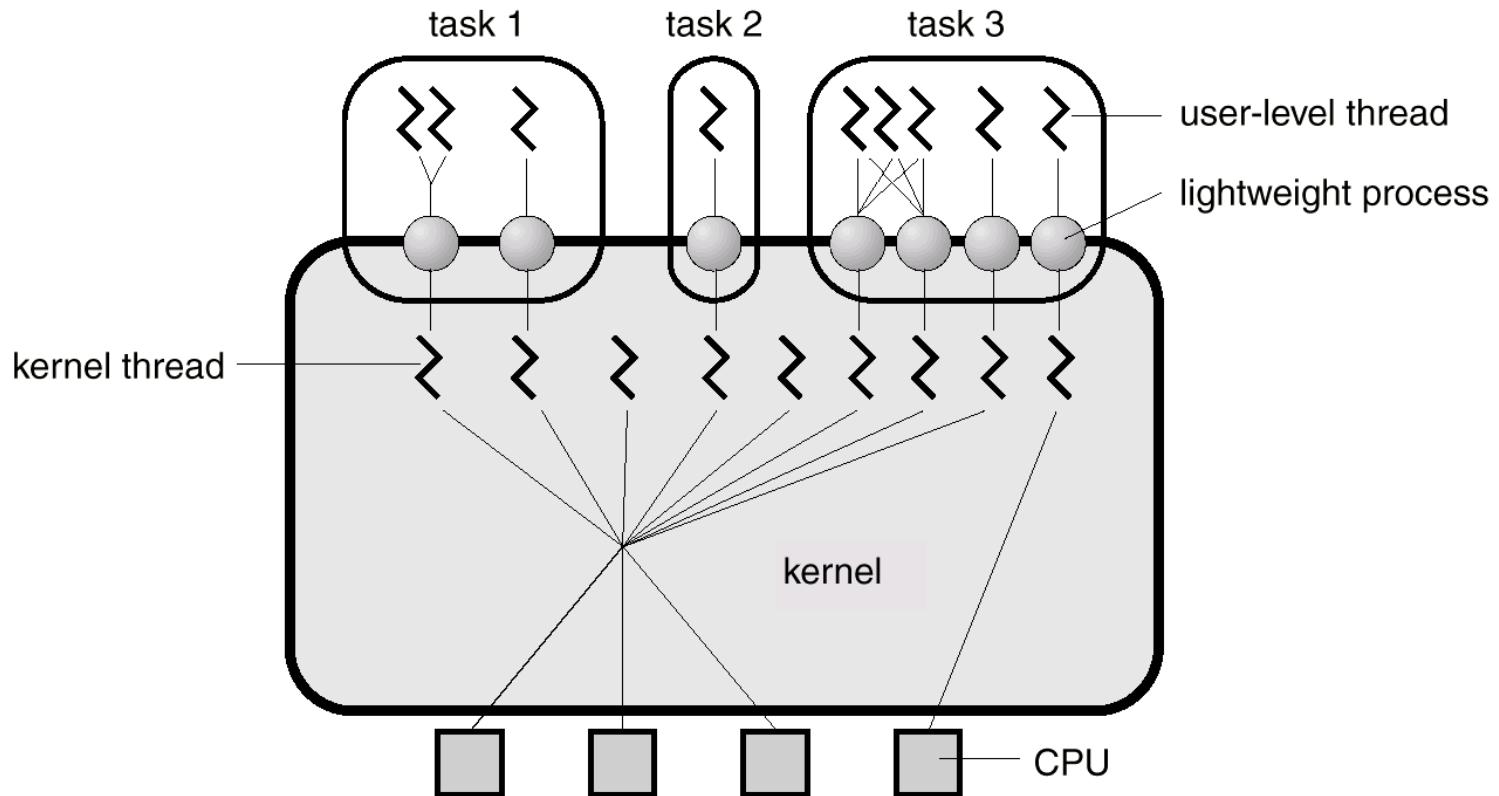
Multiple Threads within a Task



Threads Support in Solaris 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
 - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
 - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
 - User-level thread: only ned stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

Solaris 2 Threads



Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
 - **send(*message*)** – message size fixed or variable
 - **receive(*message*)**
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

Implementation Questions

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

Direct Communication

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

Indirect Communication

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

Indirect Communication

(Continued)

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

Buffering

- Queue of messages attached to the link;
implemented in one of three ways.
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous).
 2. Bounded capacity – finite length of n messages
Sender must wait if link full.
 3. Unbounded capacity – infinite length
Sender never waits.

Exception Conditions – Error Recovery

- Process terminates
- Lost messages
- Scrambled Messages

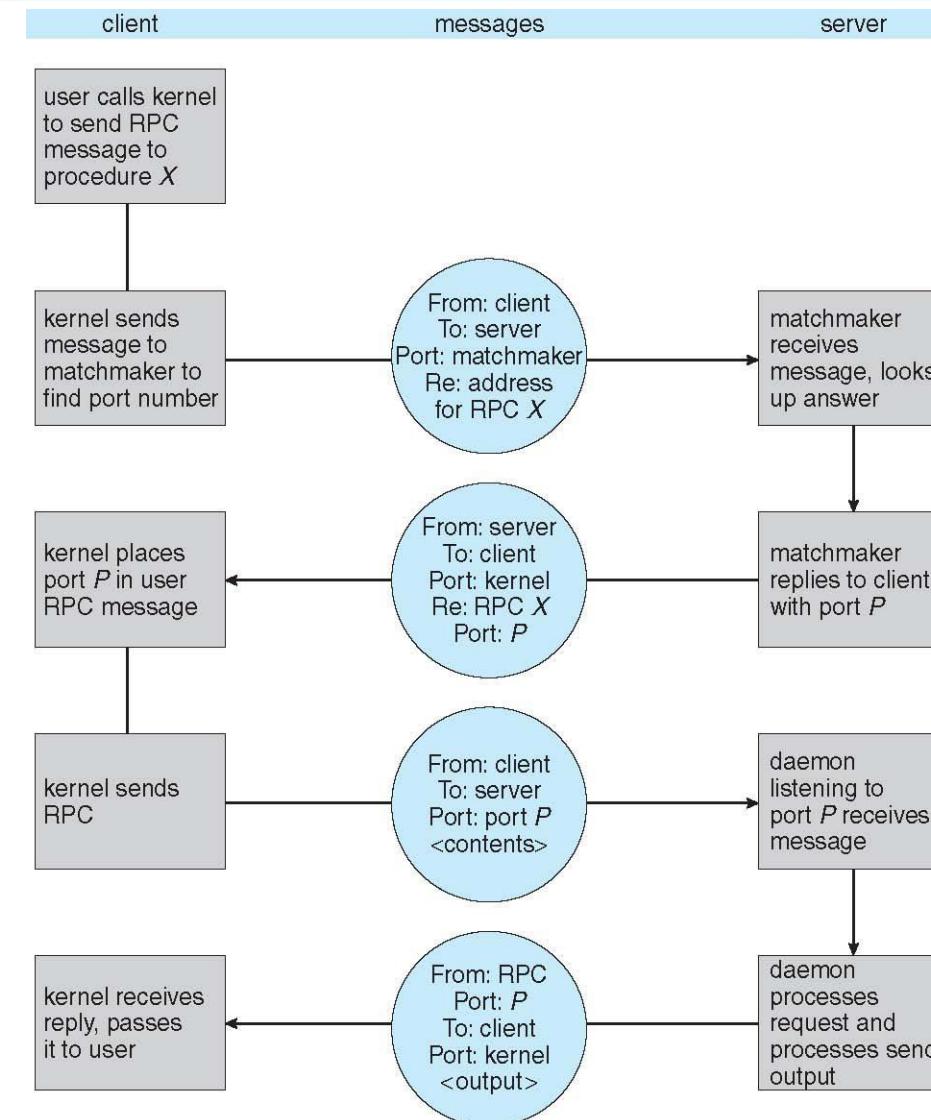
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
 - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
 - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC

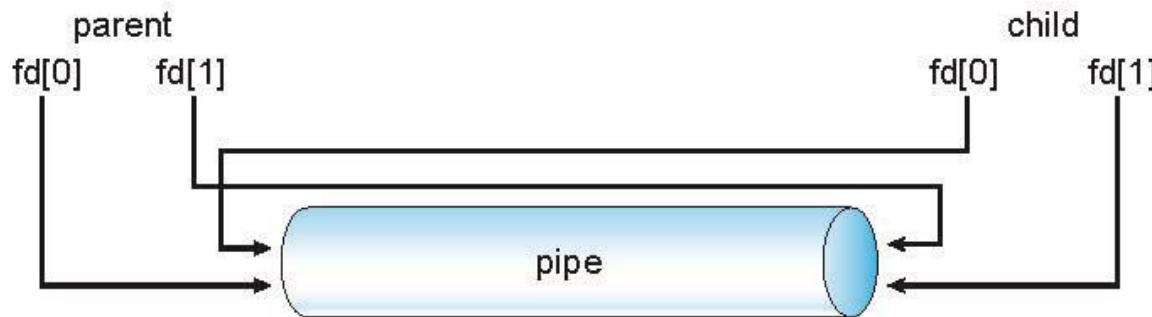


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
 - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

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



Named Pipes

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

References

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9th Edition*

Thank You

CSE-202 OPERATING SYSTEM

Module II: Process Management

Inter Process Communication

Course Learning Objectives:

- To understand the critical-section problem, whose solutions can be used to ensure the consistency of shared data

Operating System

Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

Reference Book

- **Operating system: William Stalling , Pearson Education**



Module II: Process Management

Process concept, State model, process scheduling, job and process synchronization, structure of process management, Threads.

Interprocess Communication and Synchronization: Principle of Concurrency, Producer Consumer Problem, Critical Section problem, Semaphores, Hardware Synchronization, Critical Regions, Conditional critical region, Monitor, Inter Process Communication.

CPU Scheduling: Job scheduling functions, Process scheduling, Scheduling Algorithms, Non Preemptive and preemptive Strategies, Algorithm Evaluation, Multiprocessor Scheduling.

Deadlock: System Deadlock Model, Deadlock Characterization, Methods for handling deadlock, Prevention strategies, Avoidance and Detection, Recovery from deadlock combined approach.

Topics to be Covered

- Background
- The Critical-Section Problem
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors

Background

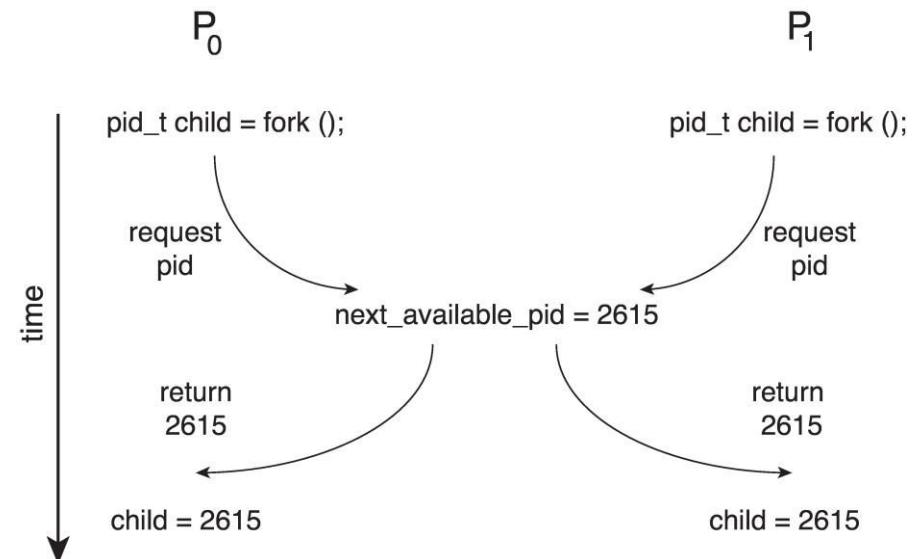
- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.

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)

Race Condition

- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!



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);

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

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

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?

- What if the critical section is code that runs for an hour?
- Can some processes starve – never enter their critical section.
 - What if there are two CPUs?

Software Solution 1

- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
 - `int turn;`
- The variable `turn` indicates whose turn it is to enter the critical section
- Initially `turn = 1`

Algorithm for Process P_i

```
while (true) {  
  
    turn = i;  
    while (turn == j)  
        ;  
  
    /* critical section */  
  
    turn = j;  
  
    /* remainder section */  
  
}
```

Correctness of the Software Solution

- Mutual exclusion is preserved
 - p_i enters critical section if and only if:
 $\text{turn} = I$
and turn cannot be both 0 and 1 at the same time
- What about the Progress requirement?
 - If Process 1 wants to enter the critical section and Process 2 is not interested in entering the critical section, can Process 1 enter?
- What about the Bounded-waiting requirement?

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

Algorithm for Process P_i

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

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
 1. Mutual exclusion is preserved
 - P_i enters CS only if:
either `flag[j] = false` or `turn = i`
 2. Progress requirement is satisfied
 3. Bounded-waiting requirement is met

Peterson's Solution

- Solution works for 2 process.
- What about modifying it to handle 10 processes?
- Solution requires **busy waiting**
 - Processes waste CPU cycles to ask if they can enter the critical section

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

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

- Thread 2 performs

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

- What is the expected output?

Modern Architecture Example

(Cont.)

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

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

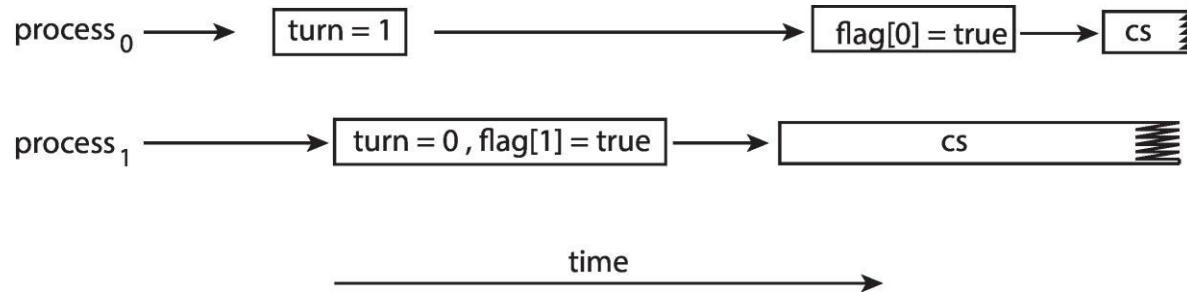
for Thread 2 may be reordered

- If this occurs, the output may be

0

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



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

Synchronization Hardware

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

Memory Barrier Instructions

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

Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

- Thread 2 now performs
- ```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that the value of **flag** is loaded before the value of **x**.
- For Thread 2 we ensure that the assignment to **x** occurs before the assignment **flag**.

# Hardware Instructions

---

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

# The test\_and\_set Instruction

- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**

# Solution Using test\_and\_set()

- Shared Boolean variable `lock`, initialized to `false`
- Solution:

```
do {
 while (test_and_set(&lock))
 ; /* do nothing */

 /* critical section */

 lock = false;
 /* remainder section */
} while (true);
```

- Based on busy waiting
- Does it solve the critical-section problem?

# The compare\_and\_swap Instruction

- Definition

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

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

# Solution using

## compare\_and\_swap

- Shared integer `lock` initialized to 0;
- Solution:

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

 /* critical section */

 lock = 0;

 /* remainder section */
}
```

- Based on busy waiting
- Does it solve the critical-section problem?

# Bounded-waiting with compare-and-swap

```
while (true) {
 waiting[i] = true;
 key = 1;
 while (waiting[i] && key == 1)
 key = compare_and_swap(&lock,0,1);
 waiting[i] = false;
 /* critical section */
 j = (i + 1) % n;
 while ((j != i) && !waiting[j])
 j = (j + 1) % n;
 if (j == i)
 lock = 0;
 else
 waiting[j] = false;
 /* remainder section */
}
```

# Atomic Variables

---

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and Booleans.
- For example:
  - Let `sequence` be an atomic variable
  - Let `increment()` be operation on the atomic variable `sequence`
  - The Command:

```
increment(&sequence);
```

ensures `sequence` is incremented without interruption:

# Atomic Variables

---

- The **increment()** function can be implemented as follows:

```
void increment	atomic_int *v)
{
 int temp;
 do {
 temp = *v;
 }
 while (temp !=
(compare_and_swap(v, temp, temp+1)) ;
}
```

# Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Two operations:
  - `acquire()` a lock
  - `release()` a lock
- The `acquire()` and `release()` are executed atomically
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

# Solution to CS Problem Using Mutex Locks

---

```
while (true) {
 acquire lock
 critical section
 release lock
 remainder section
}
```

# Semaphore

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

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

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

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

# Semaphore (Cont.)

---

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore  $S$  using binary semaphores
- With semaphores we can solve various synchronization problems

# Semaphore Usage Example

---

- Solution to the CS Problem
  - Create a semaphore “`mutex`” initialized to 1
  - Code:

```
wait(mutex) ;
 CS
signal(mutex) ;
```

# Semaphore Usage Example (Cont.)

- Consider  $P_1$  and  $P_2$  that with two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$ 
  - Create a semaphore “synch” initialized to 0
  - Code:

P1 :

```
S1;
 signal(synch);
```

P2 :

```
wait(synch);
S2;
```

# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Can be implemented using any of the critical sections solutions we discussed 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 for “regular” applications, where the application may spend lots of time in critical sections this is not a good solution

- 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
- Waiting queue

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

- The `wait` operation:

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

- The `sleep()` suspends the process that invoked it.

- The `signal` operation:

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

- The `wakeup(P)` operation resumes the execution of process P

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) ... wait(mutex)`
  - `wait(mutex) ... wait(mutex)`
  - Omitting of `wait(mutex)` and/or `signal(mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.
- Solution: introduce high-level programming constructs

# Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

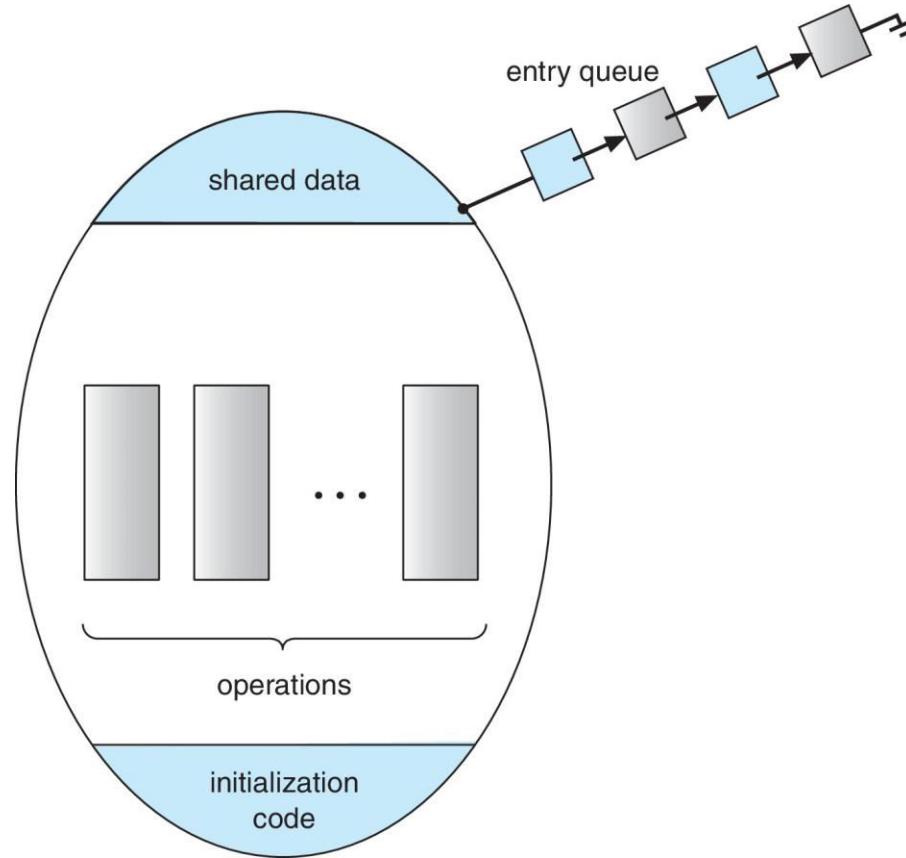
```
monitor monitor-name
{
 // shared variable declarations
 function P1 (...) { }

 function P2 (...) { }

 function Pn (...) {.....}

 initialization code (...) { ... }
}
```

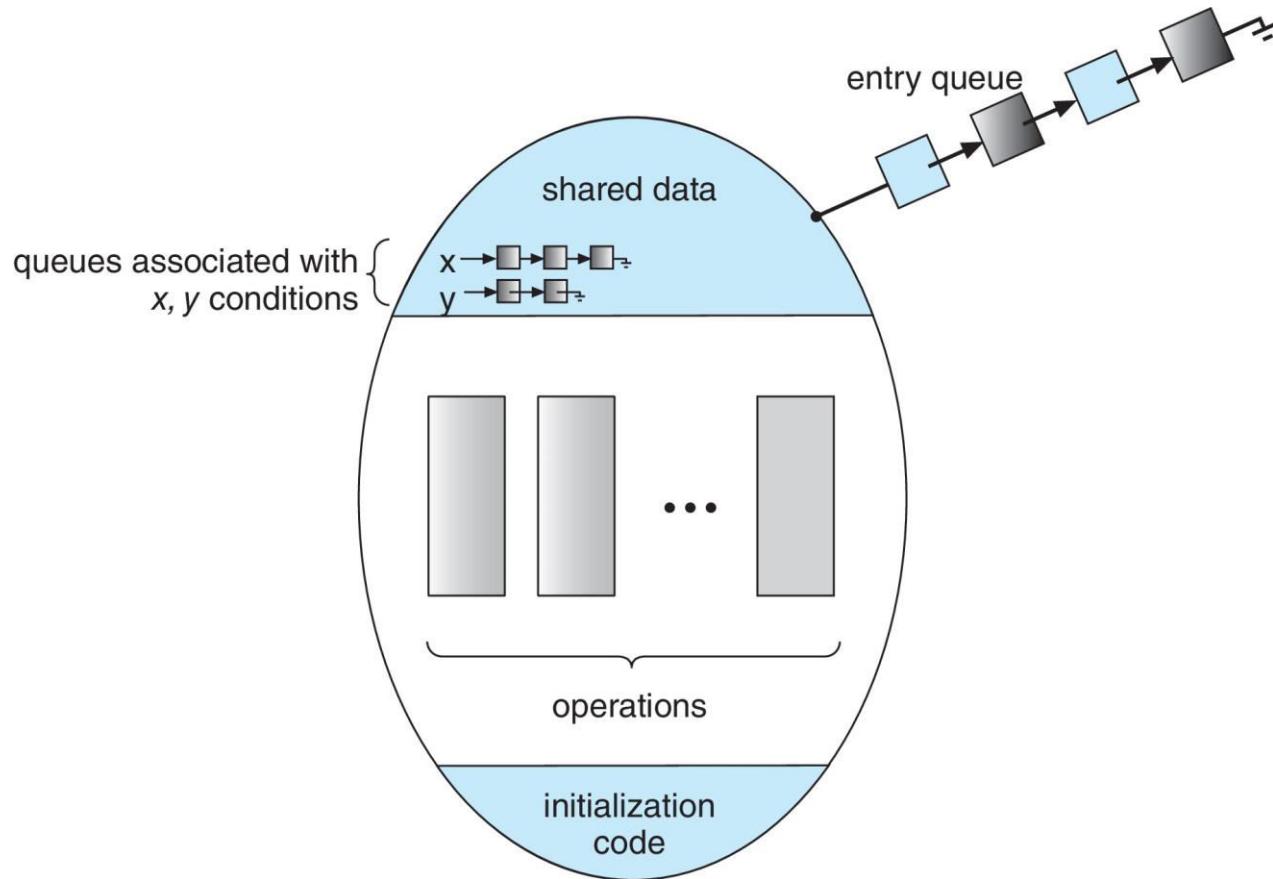
# Schematic view of a Monitor



# Condition Variables

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

# Monitor with Condition Variables



# Condition Variables Choices

---

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

# Monitor Implementation Using Semaphores



- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0; // number of processes
waiting
 inside the monitor
```

- Each function *F* will be replaced by

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

- Mutual exclusion within a monitor is ensured

# Implementation – Condition Variables

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

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

- The operation **x.wait()** can be implemented as:

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

- The operation **x.signal()** can be implemented as:

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

# Resuming Processes within a Monitor

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

# Single Resource allocation Example

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

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

...

```
R.release;
```

- Where R is an instance of type **ResourceAllocator** (shown in the next slides)

# A Monitor to Allocate Single Resource

---

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

# Single Resource Monitor (Cont.)

- Usage:

**acquire**

...  
...

**release**

- Incorrect use of monitor operations
  - **release()** ... **acquire()**
  - **acquire()** ... **acquire()**)
  - Omitting of **acquire()** and/or **release()**

# Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

# Deadlock

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

$P_0$

**wait(S) ;**

**wait(Q) ;**

...

**signal(S) ;**

**signal(Q) ;**

$P_1$

**wait(Q) ;**

**wait(S) ;**

...

**signal(Q) ;**

**signal(S) ;**

- Consider if  $P_0$  executes  $\text{wait}(S)$  and  $P_1$   $\text{wait}(Q)$ . When  $P_0$  executes  $\text{wait}(Q)$ , it must wait until  $P_1$  executes  $\text{signal}(Q)$
- However,  $P_1$  is waiting until  $P_0$  execute  $\text{signal}(S)$ .
- Since these  $\text{signal}()$  operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## **CSE-202 OPERATING SYSTEM**

### **Module II: Process Management**

#### **CPU Scheduling**

## **Course Learning Objectives:**

- Describe Basic concepts of CPU scheduling algorithms

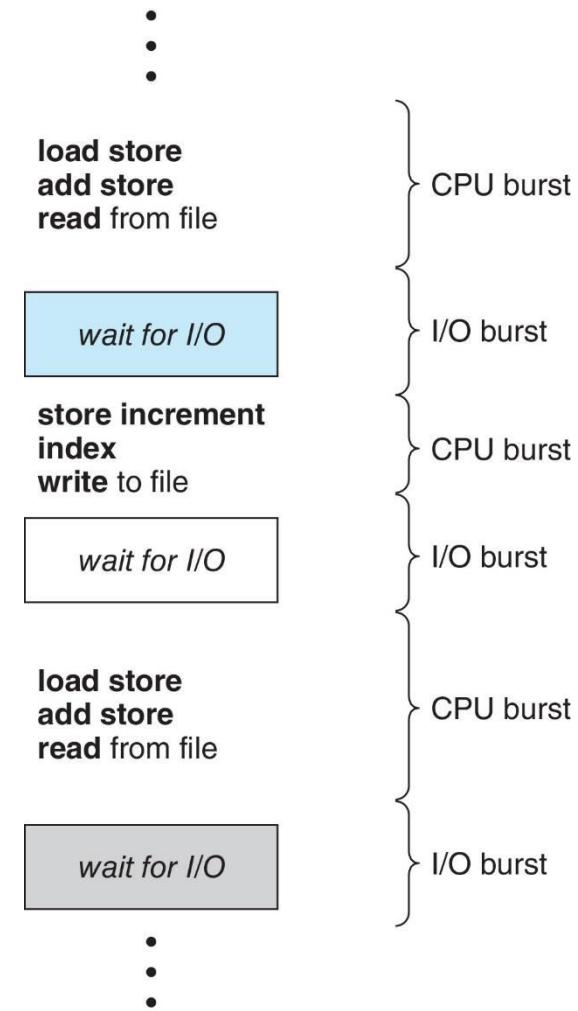
# Topics to be covered

**Basic Concepts**

**Scheduling Criterion**

# Basic Concepts

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



# Another Example

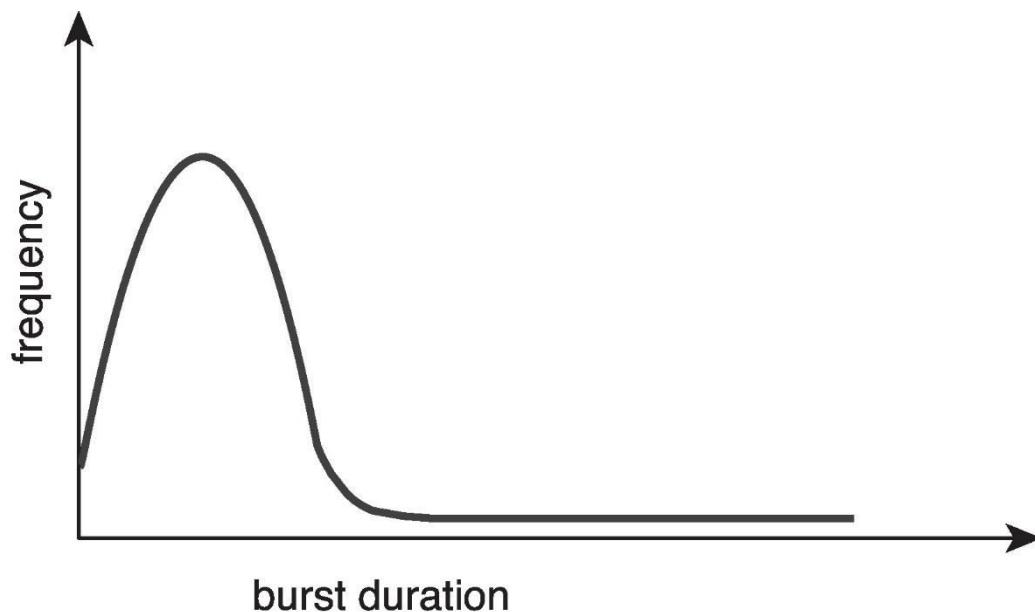
```
Printf("enter the three variables x,y,z");
scanf("%f %f %f",&x,&y,&z);
if(x>y)
{
if(x>z)
printf("x is greatest");
else
printf("z is greatest");
}
else
{
if(y>z)
printf("y is greatest");
else
printf("z is greatest");
}
getch();
```

# Histogram of CPU-burst Times

---

Large number of short bursts

Small number of longer bursts



# CPU Scheduler

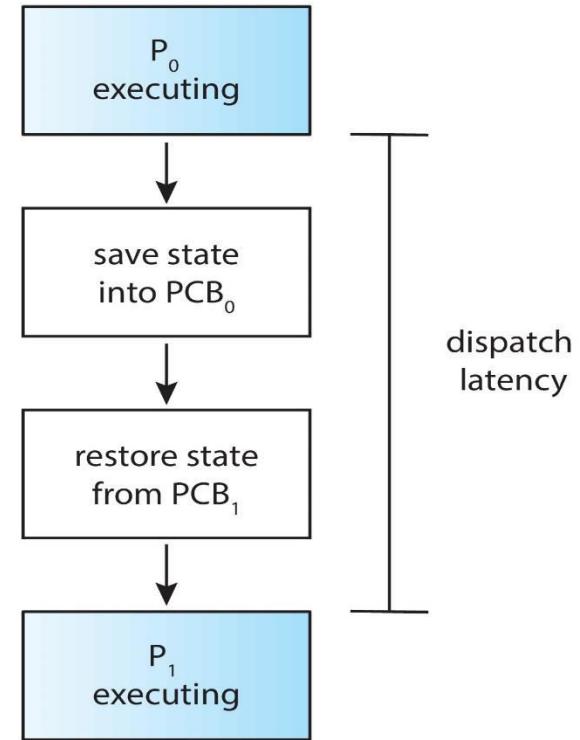
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- 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.
- For situations 2 and 3, however, there is a choice.

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive 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.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

- 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.
- This issue will be explored in detail in Chapter 6.

# Dispatcher

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



# Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.

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

# Summary

- Basic Concepts
- CPU Scheduler
- Preemptive and Nonpreemptive Scheduling
- Dispatcher
- Scheduling Criteria

# Thank You

## **CSE-202 OPERATING SYSTEM**

### **Module II: Process Management**

#### **CPU Scheduling Algorithms (Part I)**

## Course Learning Objectives:

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria

# Topics Covered

- CPU Scheduling Algorithms
  - First- Come, First-Served (FCFS) Scheduling
  - Shortest-Job-First (SJF) Scheduling
  - Round Robin (RR)

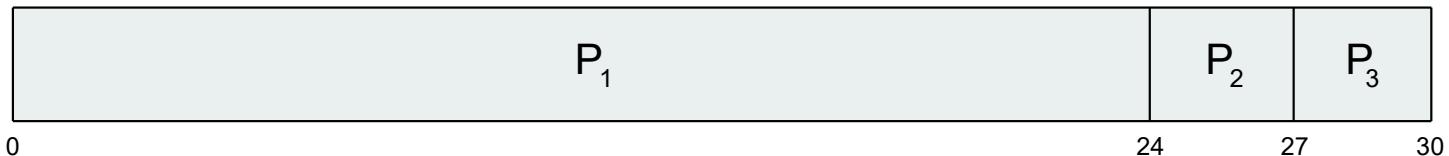
# First- Come, First-Served (FCFS) Scheduling

---

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

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

The Gantt Chart for the schedule is:



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

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



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

# Shortest-Job-First (SJF) Scheduling

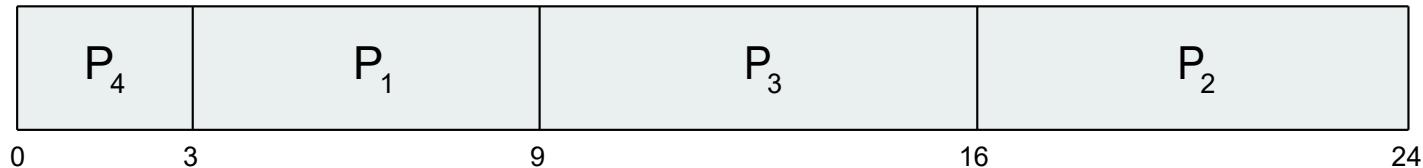
---

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
  - Estimate

# Example of SJF

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

- SJF scheduling chart



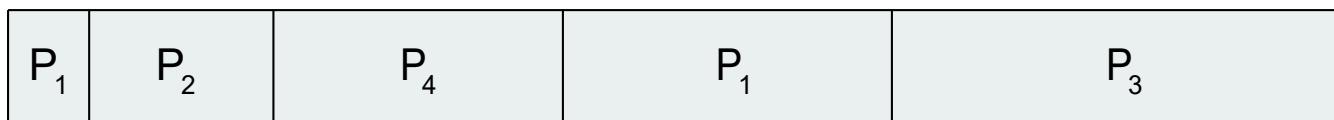
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| $P_1$          | 0                   | 8                 |
| $P_2$          | 1                   | 4                 |
| $P_3$          | 2                   | 9                 |
| $P_4$          | 3                   | 5                 |

- Preemptive SJF Gantt Chart*



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

# Round Robin (RR)

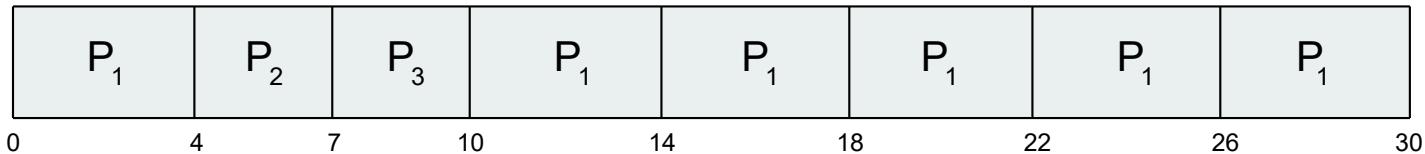
- Each process gets a small unit of CPU time (**time quantum**  $q$ ), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$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

# Example of RR with Time Quantum = 4

---

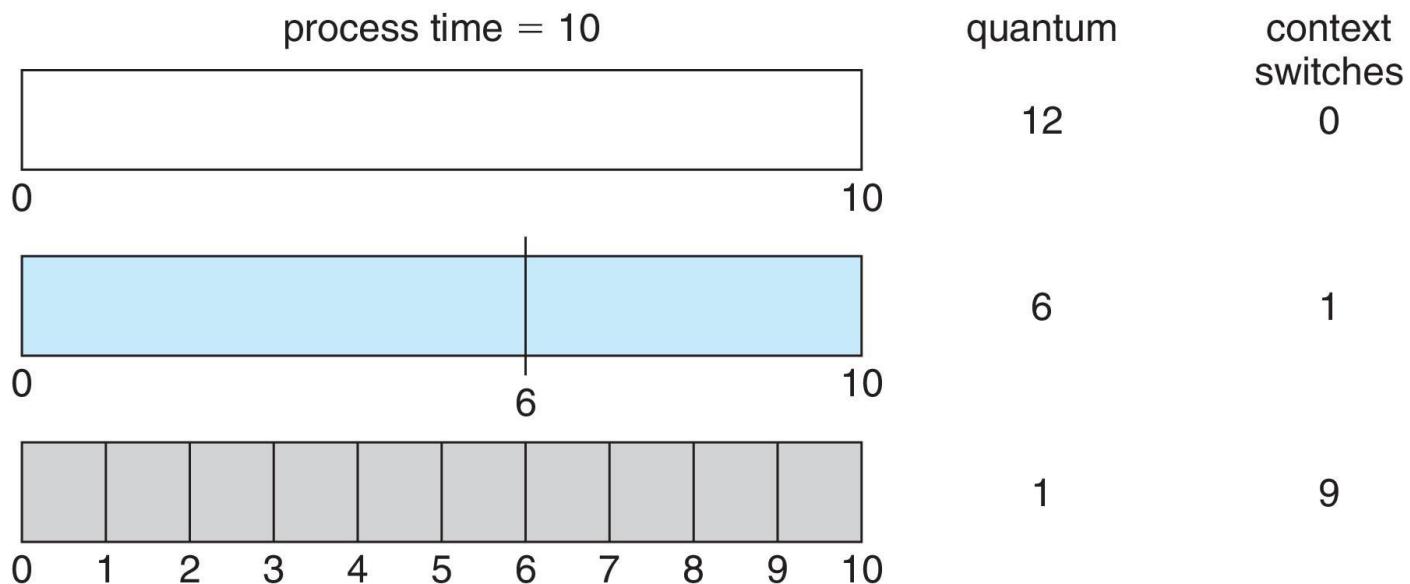
| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 24                |
| $P_2$          | 3                 |
| $P_3$          | 3                 |

- The Gantt chart is:

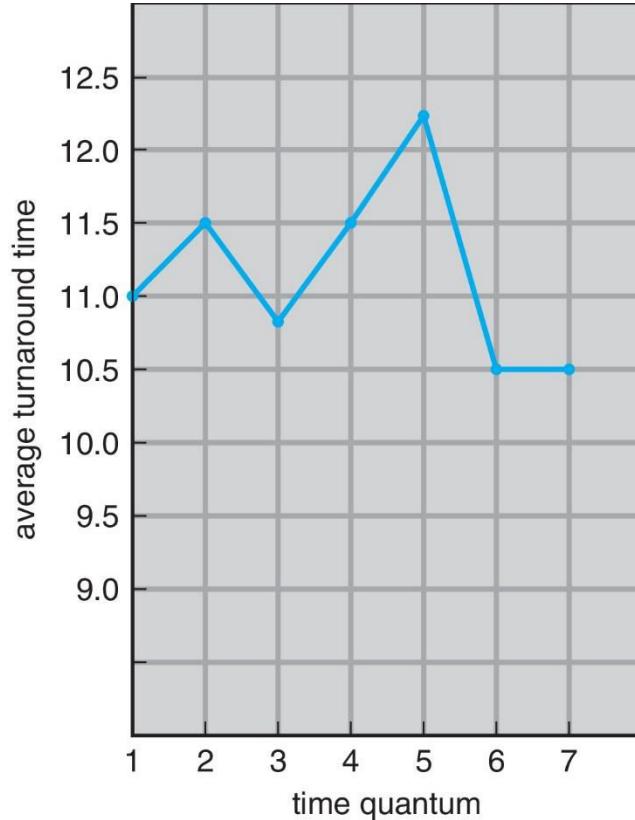


- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



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

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

# Summary

- CPU Scheduling Algorithms
  - First- Come, First-Served (FCFS) Scheduling
  - Shortest-Job-First (SJF) Scheduling
  - Round Robin (RR)

# Thank You

## **CSE-202 OPERATING SYSTEM**

### **Module II: Process Management**

#### **CPU Scheduling Algorithms (Part II)**

## **Course Learning Objectives:**

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria

# Topics Covered

---

- CPU Scheduling Algorithms
  - Priority Scheduling
  - Multilevel Queue
  - Multilevel Feedback Queue

# Priority Scheduling

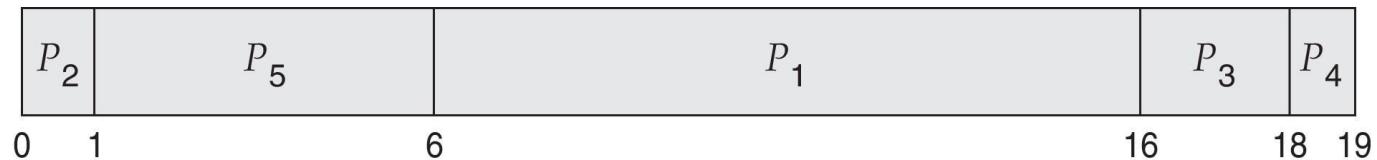
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

# Example of Priority Scheduling

---

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

- Priority scheduling Gantt Chart

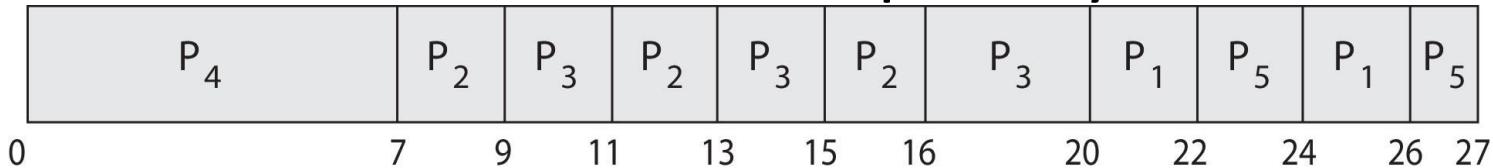


- Average waiting time = 8.2

# Priority Scheduling v/s Round-Robin

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| $P_1$          | 4                 | 3               |
| $P_2$          | 5                 | 2               |
| $P_3$          | 8                 | 2               |
| $P_4$          | 7                 | 1               |
| $P_5$          | 3                 | 3               |

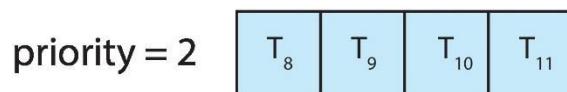
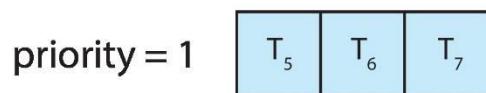
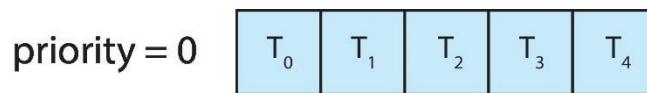
- Run the process with the highest priority.  
Processes with the same priority run round-



- Gantt Chart with time quantum = 2

# Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!

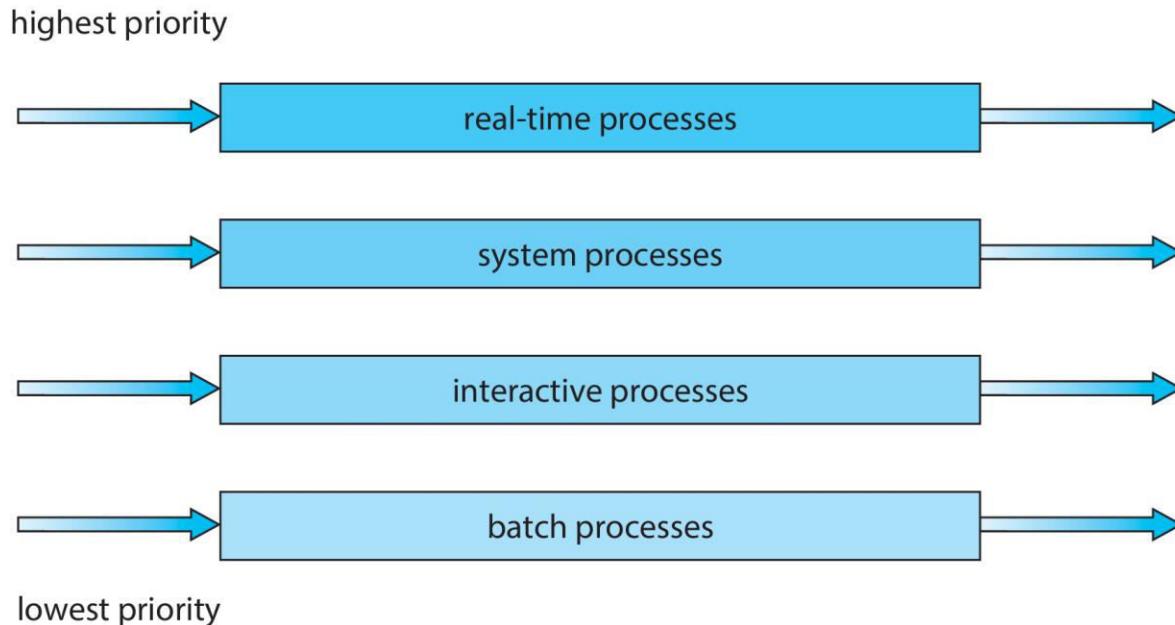


●  
●  
●



# Multilevel Queue

- Prioritization based upon process type

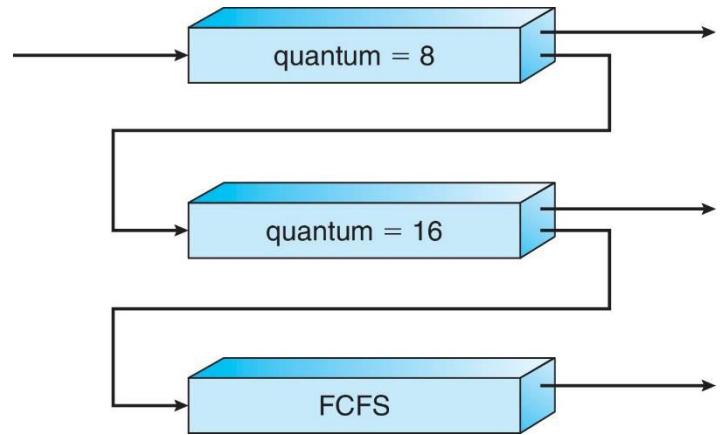


# Multilevel Feedback Queue

- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

# Example of Multilevel Feedback Queue

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



# Summary

- CPU Scheduling Algorithms
  - Priority Scheduling
  - Multilevel Queue
  - Multilevel Feedback Queue

# Thank You

## **CSE-202 OPERATING SYSTEM**

### **Module II: Process Management**

#### **Deadlocks**

## **Course Learning Objectives:**

- To understand deadlocks, which prevent sets of concurrent processes from completing their tasks
- To understand different methods for avoiding, detecting deadlocks and recovery of deadlock in a computer system

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling , Pearson Education**

# Topics to be Covered

---

- System Deadlock Model
- Deadlock Characterization
- Methods for handling deadlock
- Prevention strategies
- Avoidance and Detection
- Recovery from deadlock combined approach.

# System Model

---

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

# Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

# Deadlock Characterization

---

- **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Resource-Allocation Graph

---

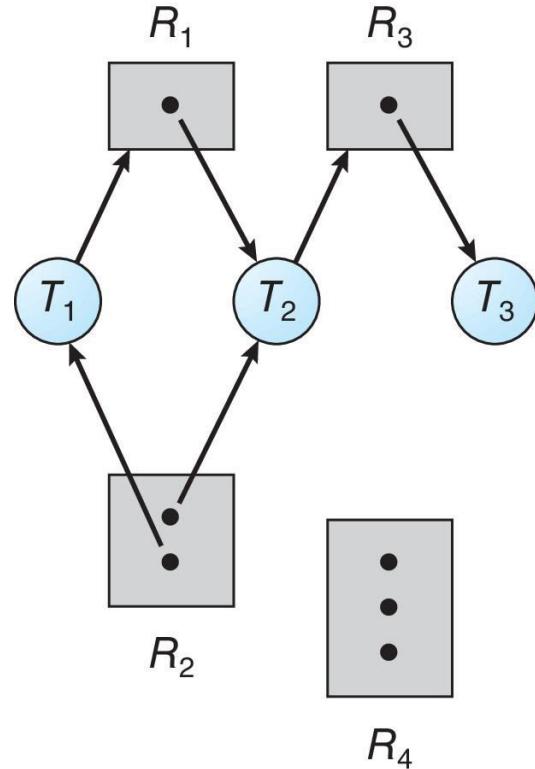
A set of vertices  $V$  and a set of edges  $E$ .

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

# Resource Allocation Graph

## Example

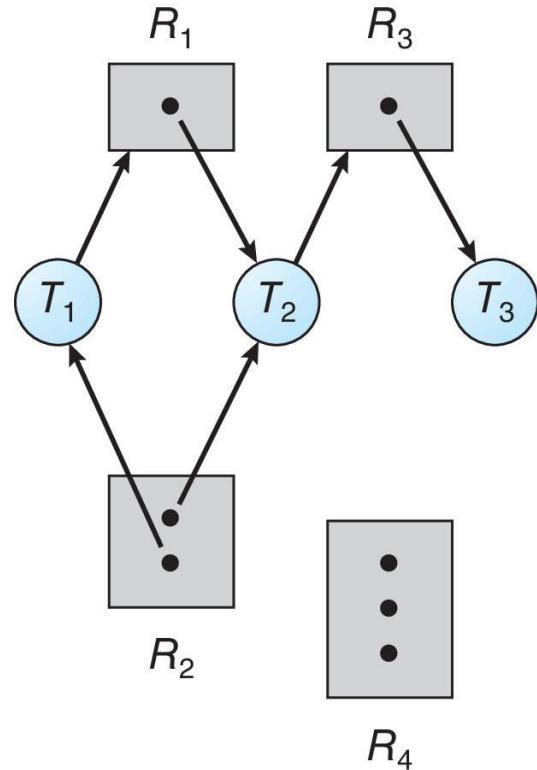
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1



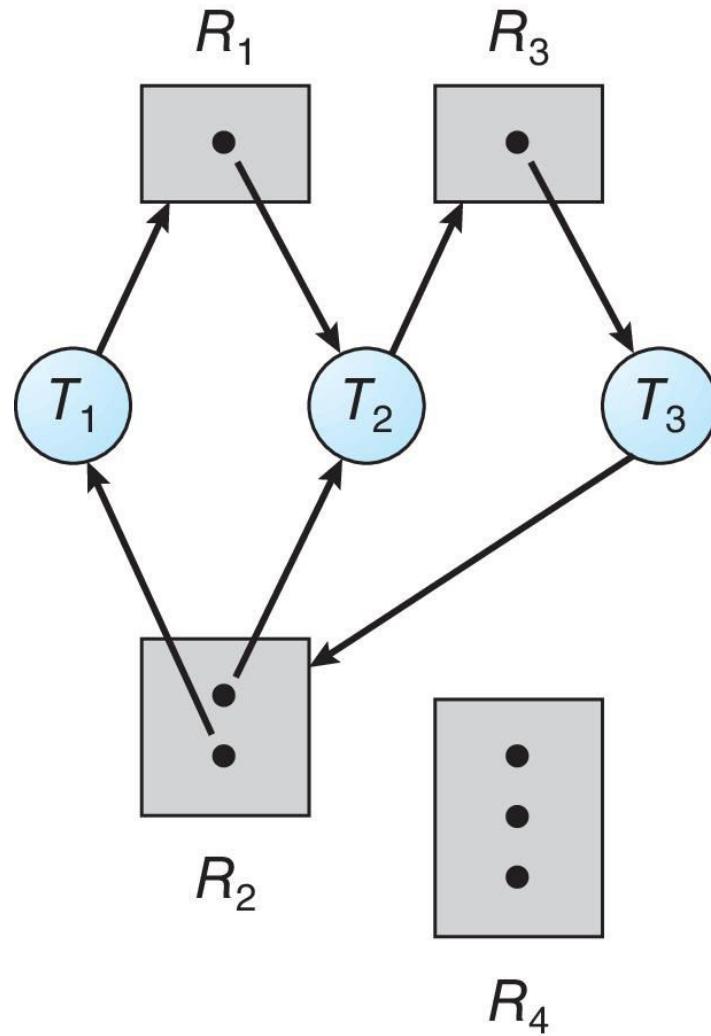
# Resource Allocation Graph

## Example

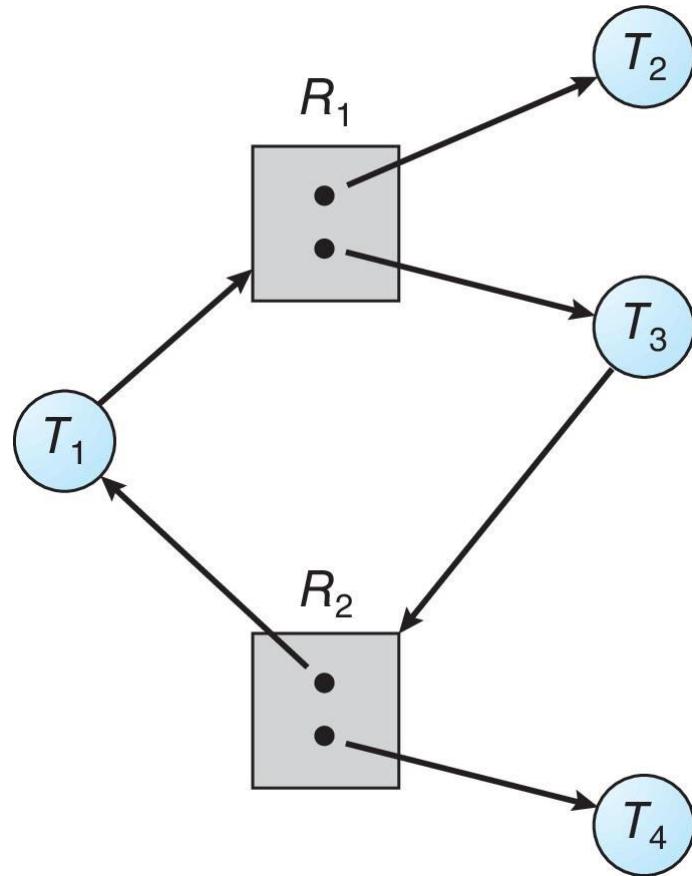
- T<sub>2</sub> holds one instance of R<sub>1</sub>, one instance of R<sub>2</sub>, and is waiting for an instance of R<sub>3</sub>
- T<sub>3</sub> is holding one instance of R<sub>3</sub>



# Resource Allocation Graph with a Deadlock



# Graph with a Cycle But no Deadlock



# Basic Facts

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

# Methods for Handling Deadlocks

---

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.

# Deadlock Prevention

---

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

# Deadlock Prevention

---

Invalidate one of the four necessary conditions for deadlock:

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

# Deadlock Prevention (Cont.)

---

- **No Preemption:**

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- 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

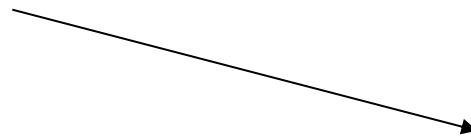
# Deadlock Prevention (Cont.)

---

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

# Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

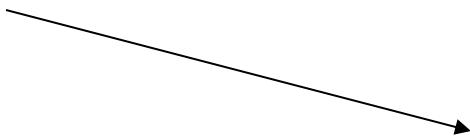


**first\_mutex = 1**

**second\_mutex = 5**

# Circular Wait

- code for **thread\_two** could not be written as follows:



```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
 pthread_mutex_lock(&first_mutex);
 pthread_mutex_lock(&second_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&second_mutex);
 pthread_mutex_unlock(&first_mutex);

 pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
 pthread_mutex_lock(&second_mutex);
 pthread_mutex_lock(&first_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&first_mutex);
 pthread_mutex_unlock(&second_mutex);

 pthread_exit(0);
}
```

# Deadlock Avoidance

---

Requires that the system has some additional a priori information available

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

# Deadlock Avoidance

---

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

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

# Safe State

---

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$

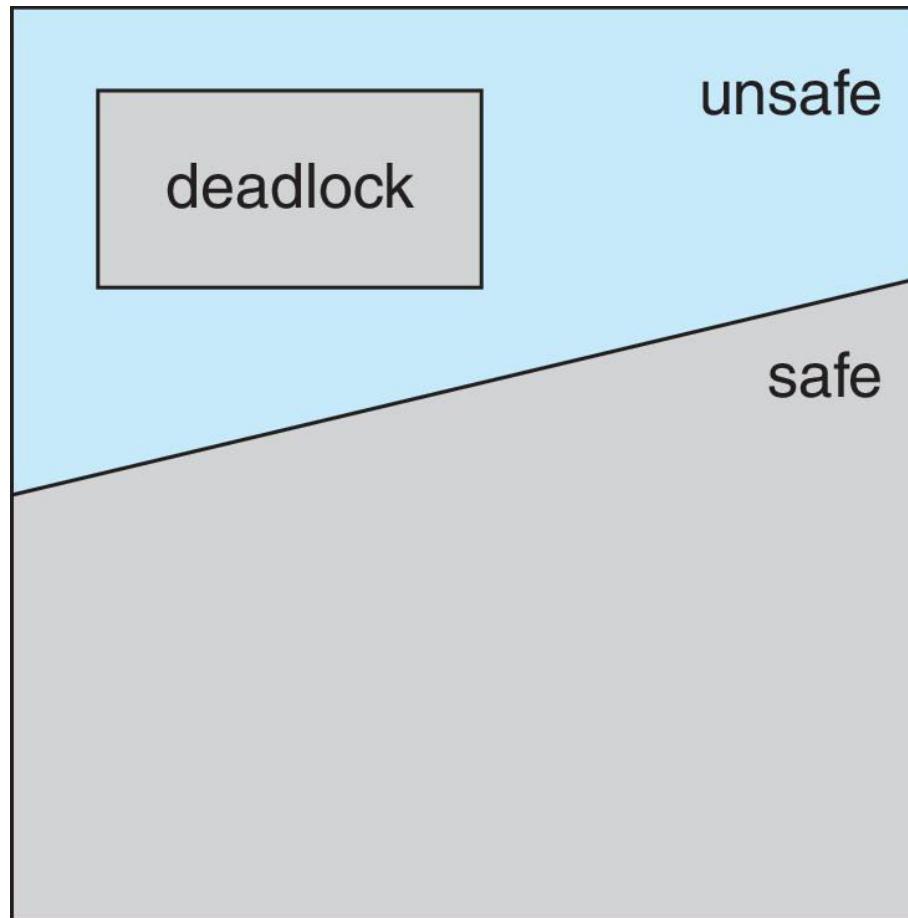
# Safe State

- That is:
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

# Basic Facts

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

# Safe, Unsafe, Deadlock State



# Avoidance Algorithms

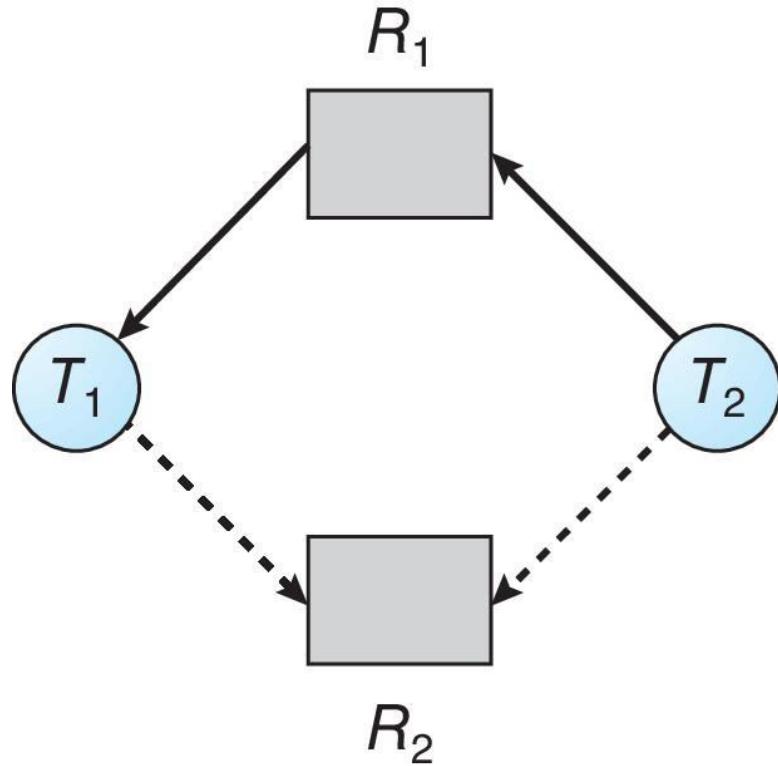
---

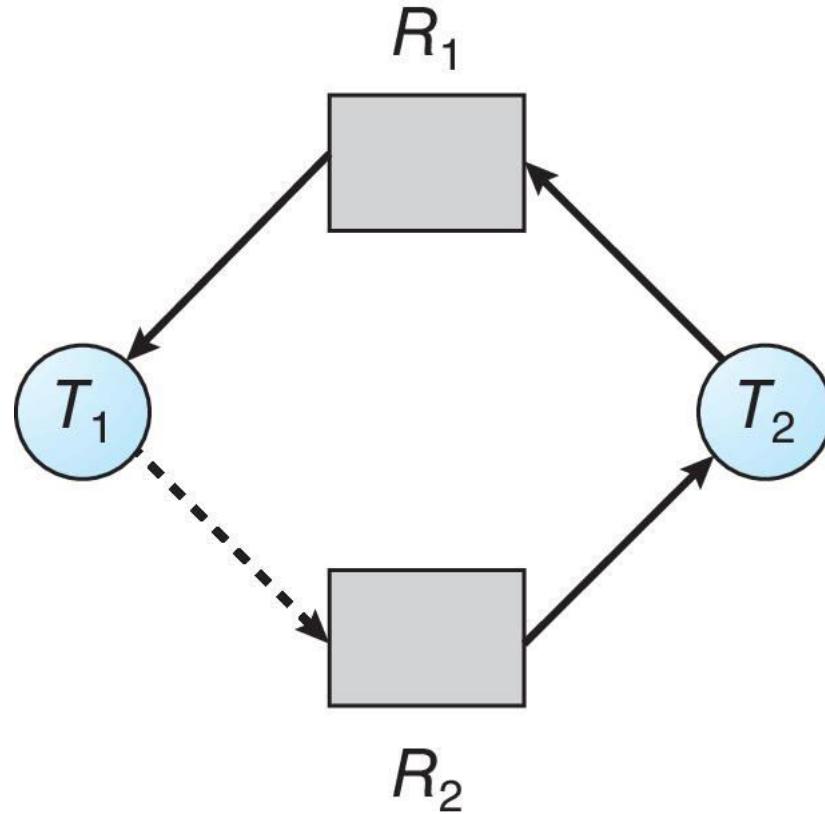
- Single instance of a resource type
  - Use a modified resource-allocation graph
  
- Multiple instances of a resource type
  - Use the Banker's Algorithm

- **Claim edge**  $P_i \rightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$
- **Request edge**  $P_i \rightarrow R_j$  indicates that process  $P_i$  requests resource  $R_j$ 
  - Claim edge converts to request edge when a process requests a resource

- **Assignment edge**  $R_j \rightarrow P_i$  indicates that resource  $R_j$  was allocated to process  $P_i$ 
  - Request edge converts to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

# Resource-Allocation Graph





# Resource-Allocation Graph

## Algorithm

---

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

# Banker's Algorithm

---

- Multiple instances of resources
- 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

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  $R_j$

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

- **Need:**  $n \times m$  matrix. If  $Need[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$Need [i,j] = Max[i,j] - Allocation [i,j]$$

# Safety Algorithm

---

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

***Work = Available***

***Finish [i] = false*** for  $i = 0, 1, \dots, n-1$

2. Find an  $i$  such that both:

(a) ***Finish [i] = false***

(b) ***Need<sub>i</sub> ≤ Work***

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

# Safety Algorithm

---

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

$\text{Finish}[i] = \text{true}$

go to step 2

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

**$Request_i$** , = request vector for process  $P_i$ . If  **$Request_i[j]$**  =  $k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $Request_i \leq Available$ , go to step 3.  
Otherwise  $P_i$  must wait, since resources are not available

## Resource-Request Algorithm for Process $P_i$

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> |          |          |
|-------|-------------------|----------|----------|------------|----------|----------|------------------|----------|----------|
|       | <i>A</i>          | <i>B</i> | <i>C</i> | <i>A</i>   | <i>B</i> | <i>C</i> | <i>A</i>         | <i>B</i> | <i>C</i> |
| $P_0$ | 0                 | 1        | 0        | 7          | 5        | 3        | 3                | 3        | 2        |
| $P_1$ | 2                 | 0        | 0        | 3          | 2        | 2        |                  |          |          |
| $P_2$ | 3                 | 0        | 2        | 9          | 0        | 2        |                  |          |          |
| $P_3$ | 2                 | 1        | 1        | 2          | 2        | 2        |                  |          |          |
| $P_4$ | 0                 | 0        | 2        | 4          | 3        | 3        |                  |          |          |

# Example (Cont.)

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

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

# Example (Cont.)

- Snapshot at time  $T_0$ :

|       | <u>Allocation</u> |          |          | <u>Max</u> | <u>Available</u> | <u>Need</u> |
|-------|-------------------|----------|----------|------------|------------------|-------------|
|       | <i>A</i>          | <i>B</i> | <i>C</i> | <i>A</i>   | <i>B</i>         | <i>C</i>    |
| $P_0$ | 0                 | 1        | 0        | 7          | 5                | 3           |
| $P_1$ | 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           |

- 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)

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

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

- Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  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

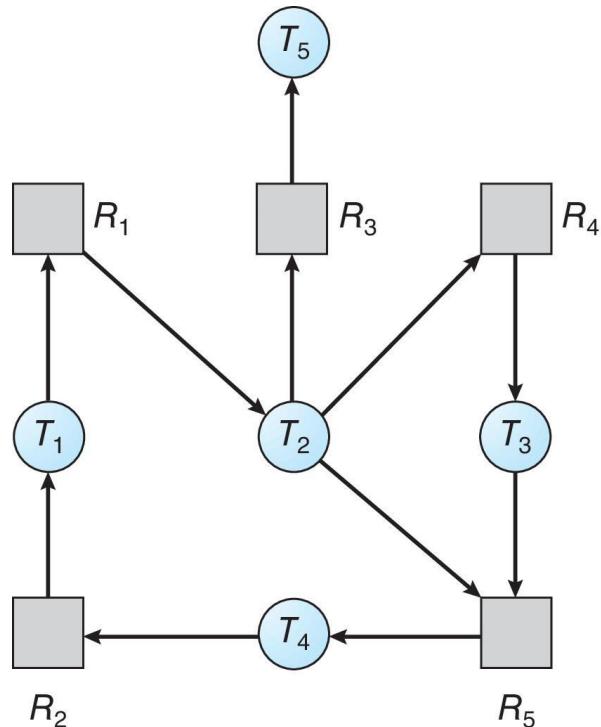
---

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

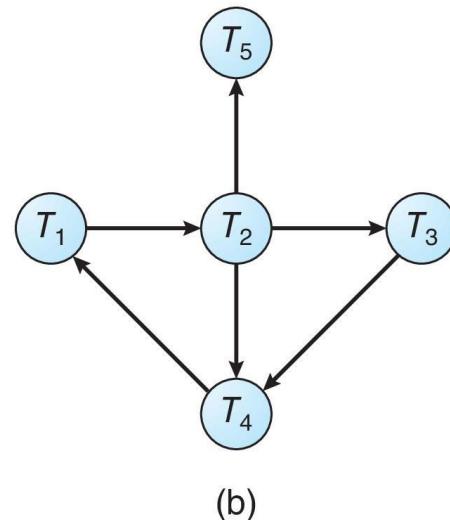
# Single Instance of Each Resource Type

- 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. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph

## Resource-Allocation Graph and Wait-for Graph



(a)



(b)

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<sub>i</sub>***  $\neq 0$ , then ***Finish[i] = false***; otherwise, ***Finish[i] = true***

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

- a) ***Finish[i] == false***
- b) ***Request<sub>i</sub> ≤ 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  **$Finish[i] == false$** , for some  $i$ ,  $1 \leq i \leq n$ ,

then the system is in deadlock state.

Moreover, if  **$Finish[i] == false$** , 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  $Finish[i] = true$  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 | 2 |
| $P_2$ | 0 | 0 | 1 |
| $P_3$ | 1 | 0 | 0 |
| $P_4$ | 0 | 0 | 2 |

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

# Detection-Algorithm Usage

---

- When, and how often, to invoke 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.

- 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?

- **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, include number of rollback in cost factor

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## CSE-202 OPERATING SYSTEM

### Module III: Memory Management

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling , Pearson Education**

# Topics to be covered

**Background, Basic Concepts**

**Swapping, Contiguous Allocation**

**Paging, Segmentation**

**Segmentation with Paging**

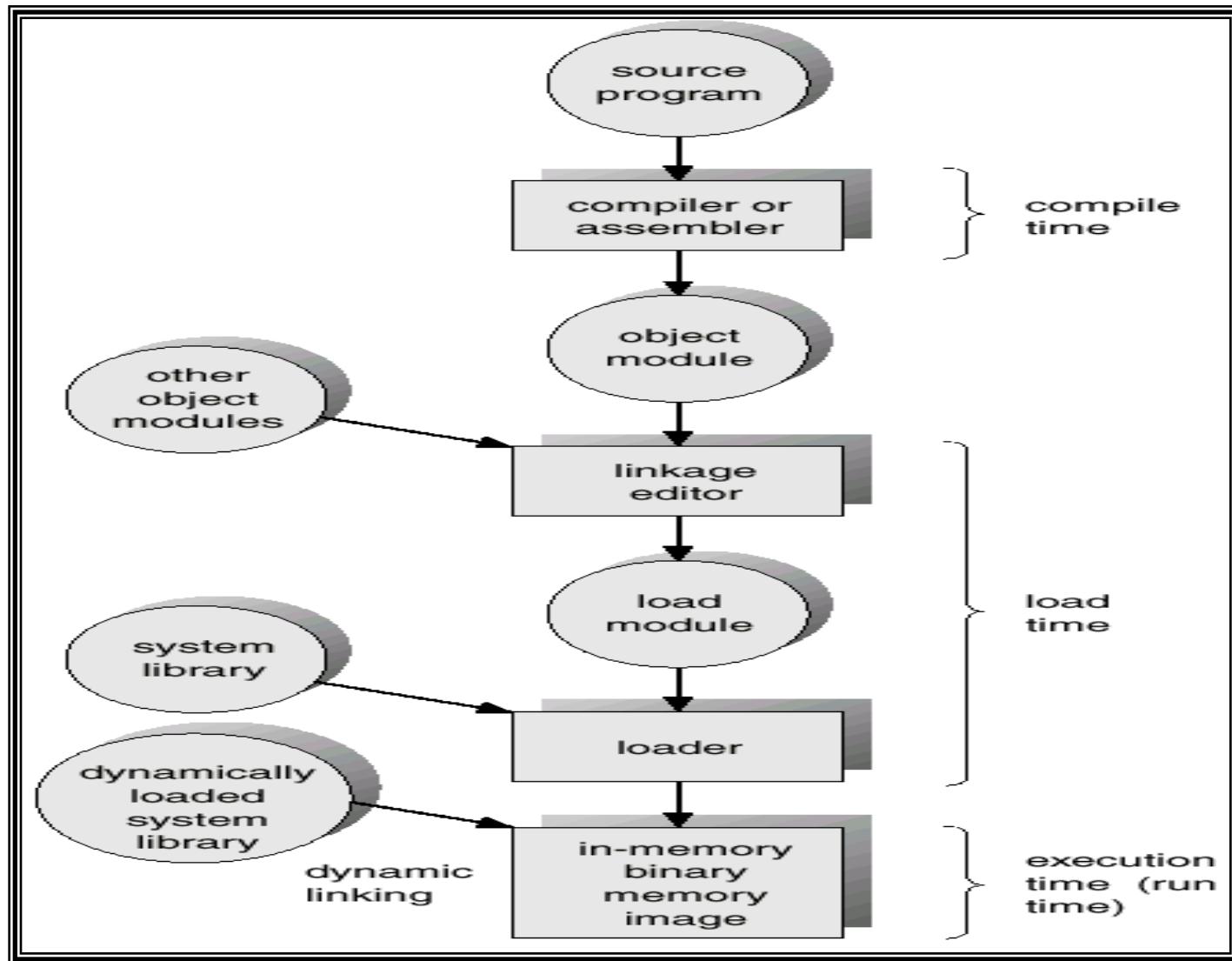
# Background

- Program must be brought into memory and placed within a process for it to be run.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.

**Address binding of instructions and data to memory addresses can happen at three different stages.**

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

# Multistep Processing of a User Program



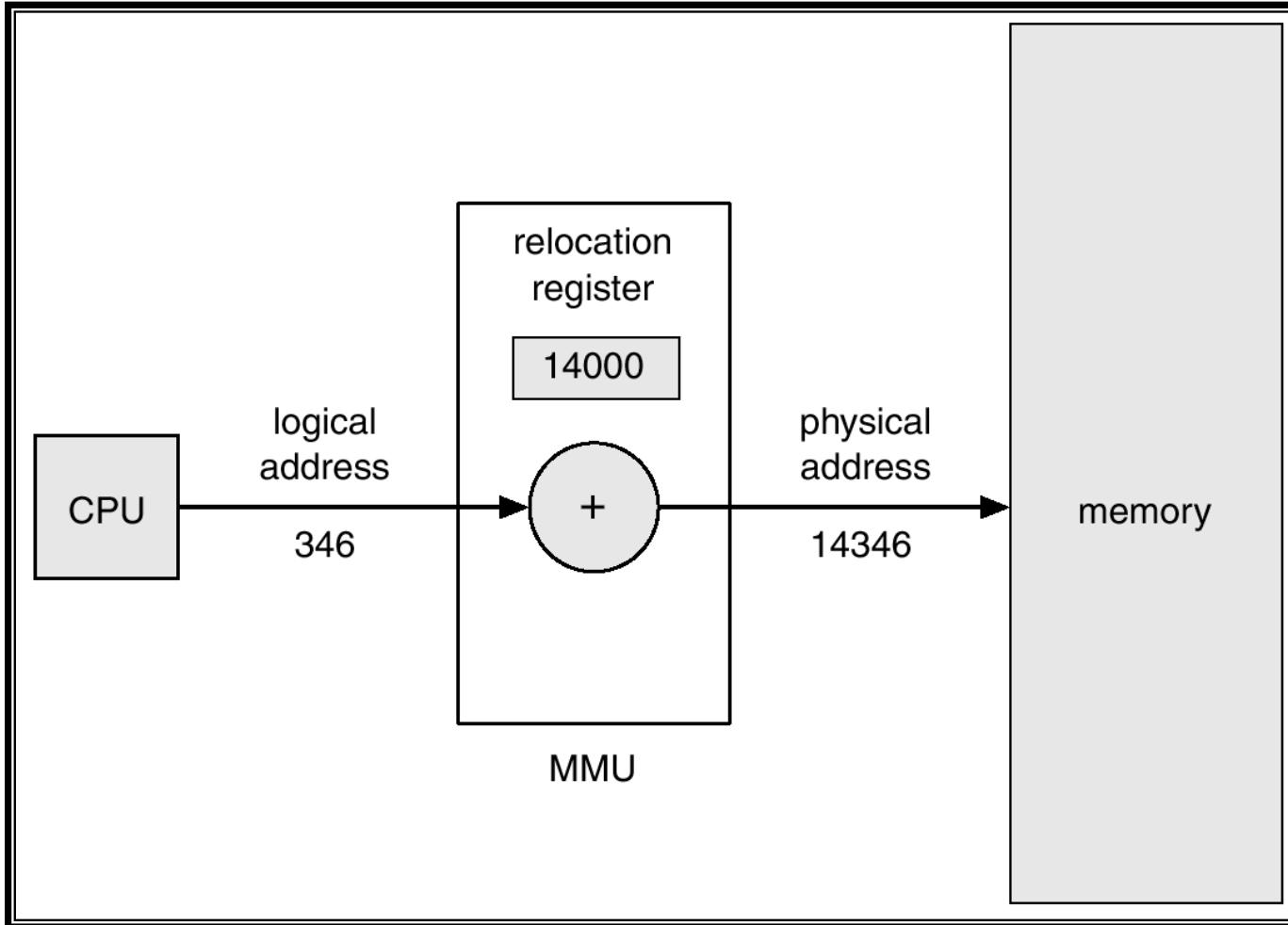
# Logical vs. Physical Address Space

---

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

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

# Dynamic relocation using a relocation register



# Dynamic Loading

---

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design.

# Dynamic Linking

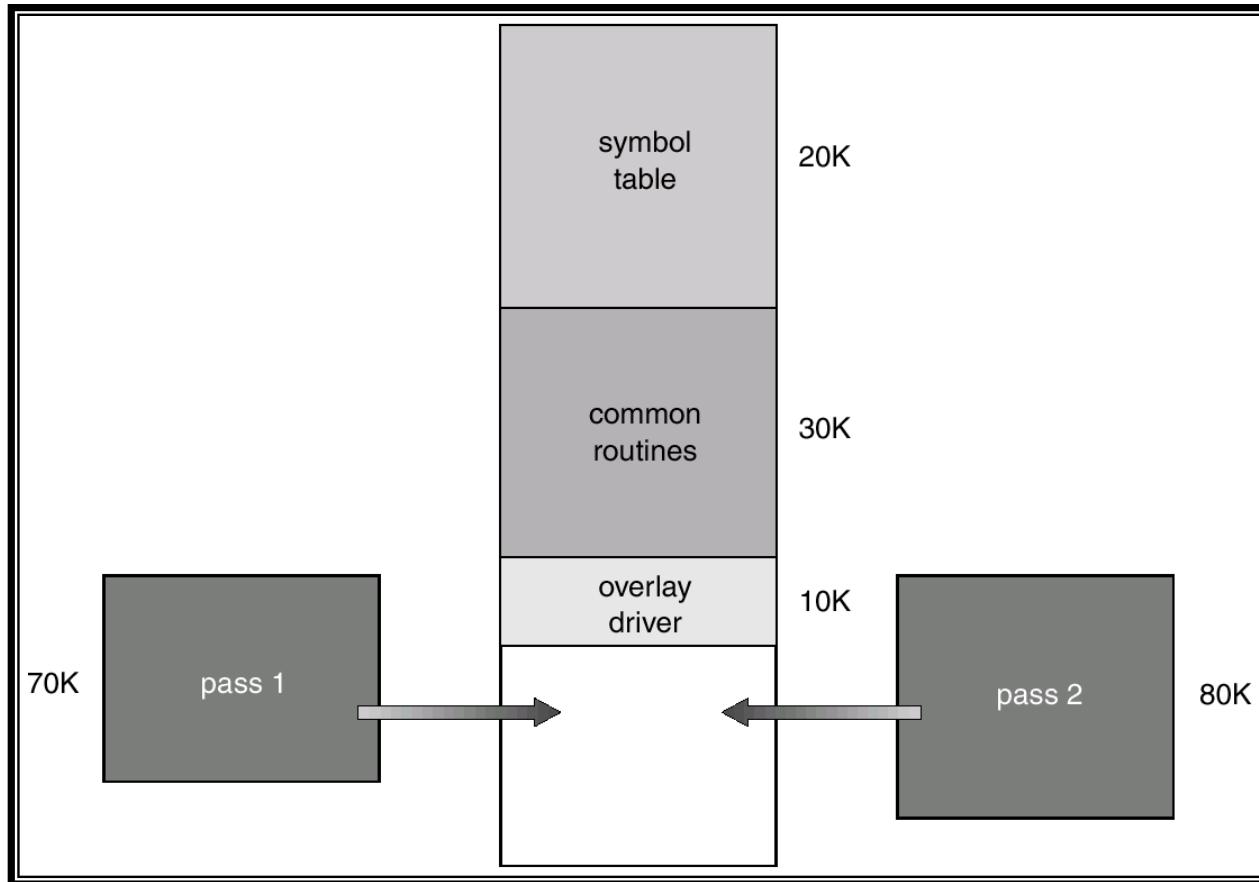
---

- 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 needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

# Overlays

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

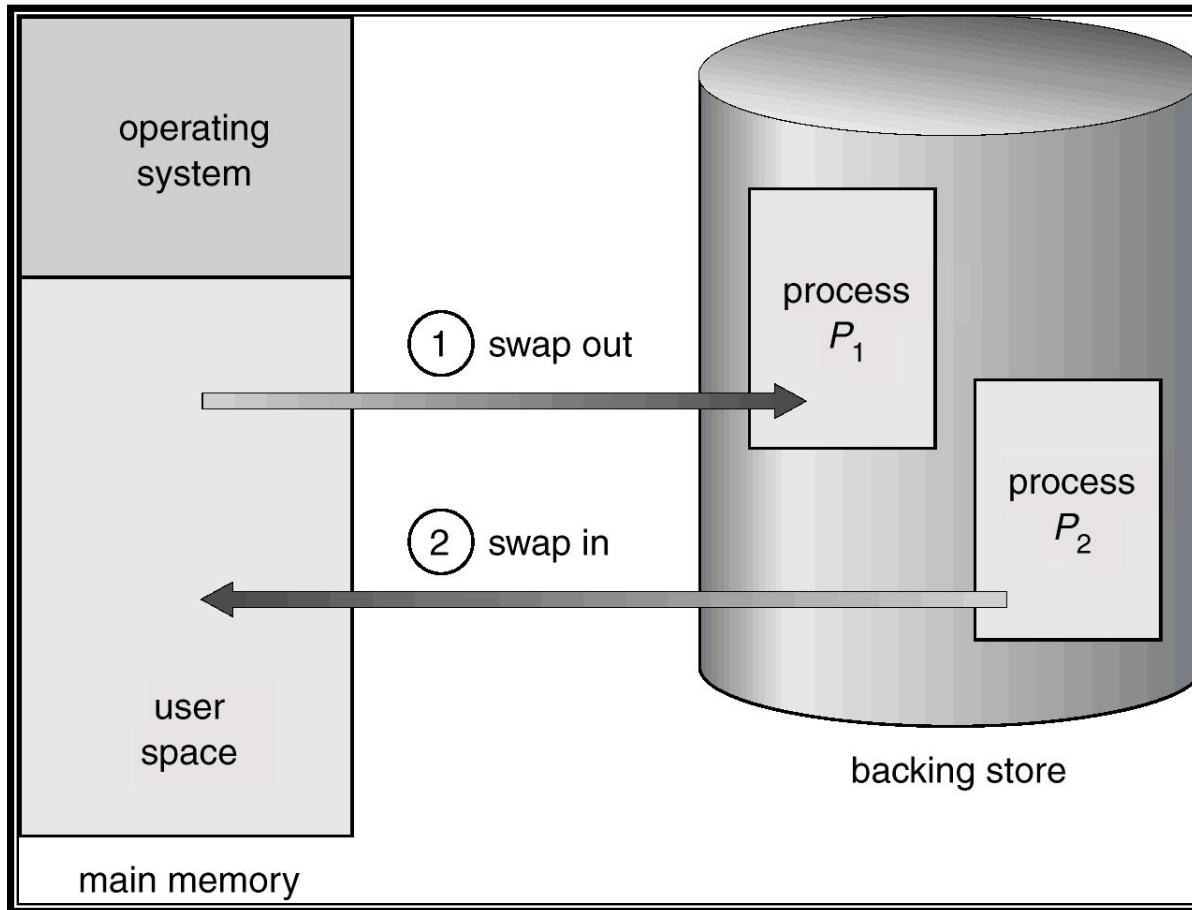
# Overlays for a Two-Pass Assembler



# Swapping

- ▶ A process can be *swapped* temporarily out of memory to a *Backing store*, and then brought back into memory for continued execution.
- ▶ Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- ▶ *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- ▶ Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- ▶ Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

# Schematic View of Swapping

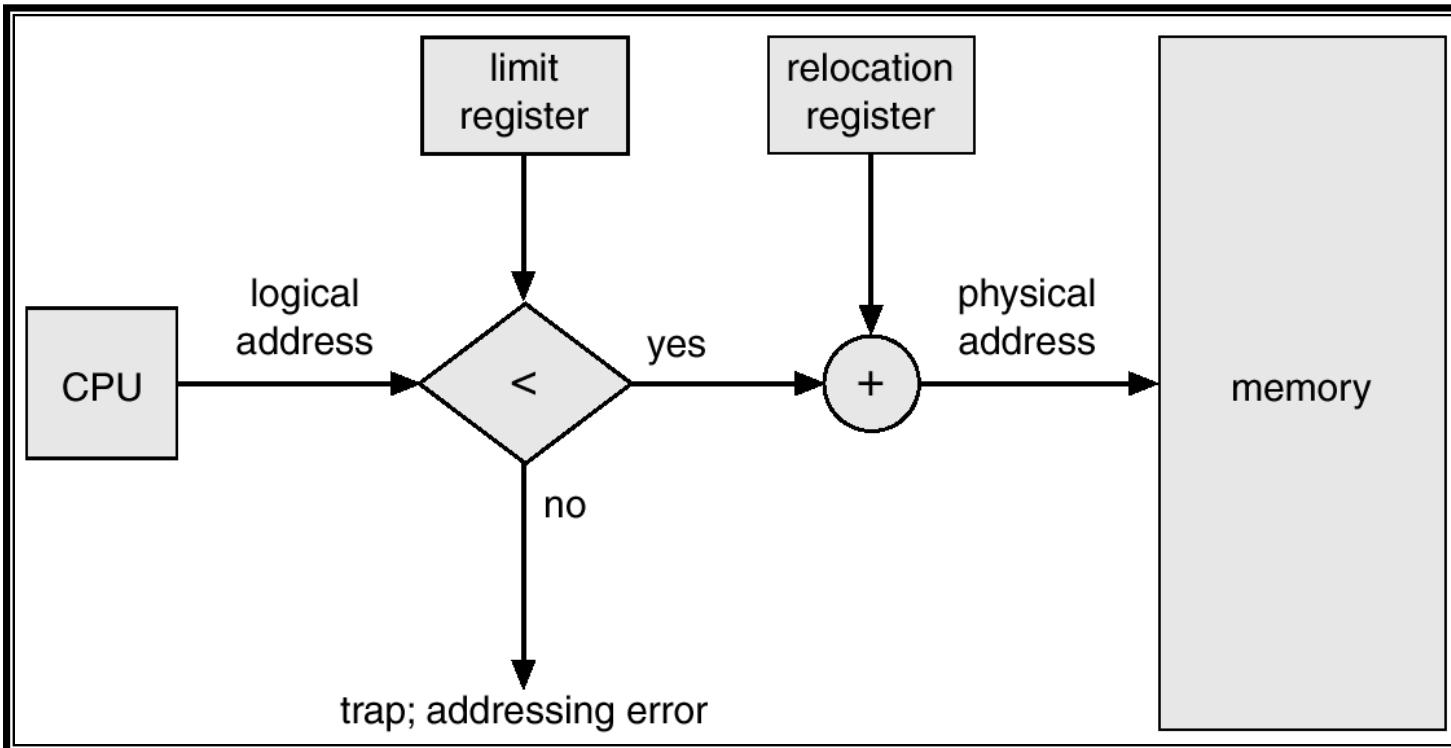


# Contiguous Allocation

---

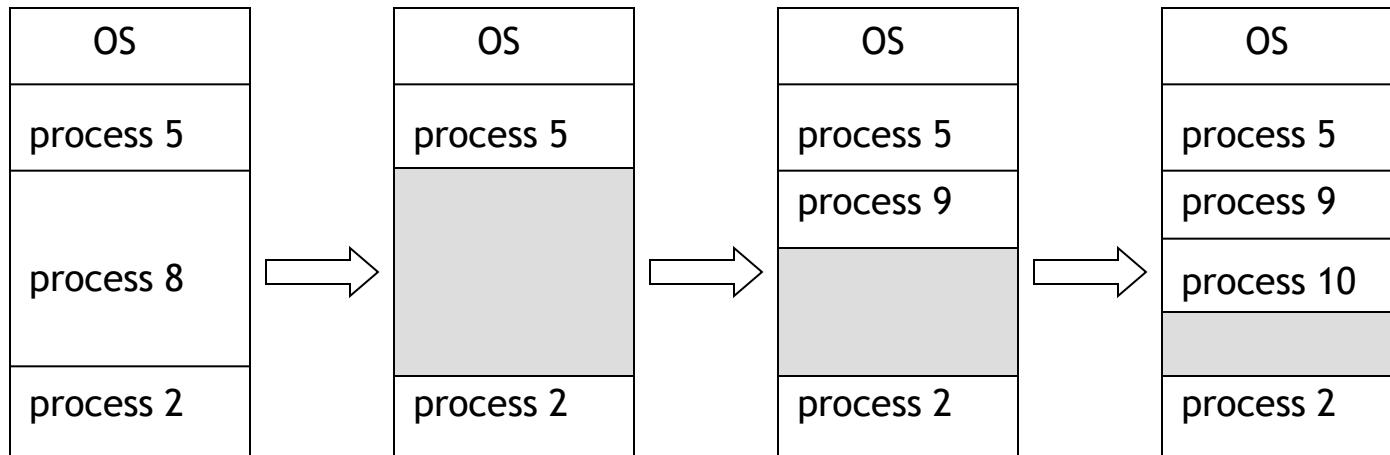
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

# Hardware Support for Relocation and Limit Registers



# Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - *Hole* – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    - a) allocated partitions
    - b) free partitions (hole)



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

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

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

- Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in that order). How would the first Fit, best fit and Worst Fit algorithms place the processes A: 212 KB, B: 417 KB C: 112 KB and D: 426 KB

Which algorithm makes the most optimal use of the memory

# Fragmentation

---

- ▶ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- ▶ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- ▶ Reduce external fragmentation by compaction
  - ▶ Shuffle memory contents to place all free memory together in one large block.
  - ▶ Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - ▶ I/O problem
    - ▶ Latch job in memory while it is involved in I/O.
    - ▶ Do I/O only into OS buffers.

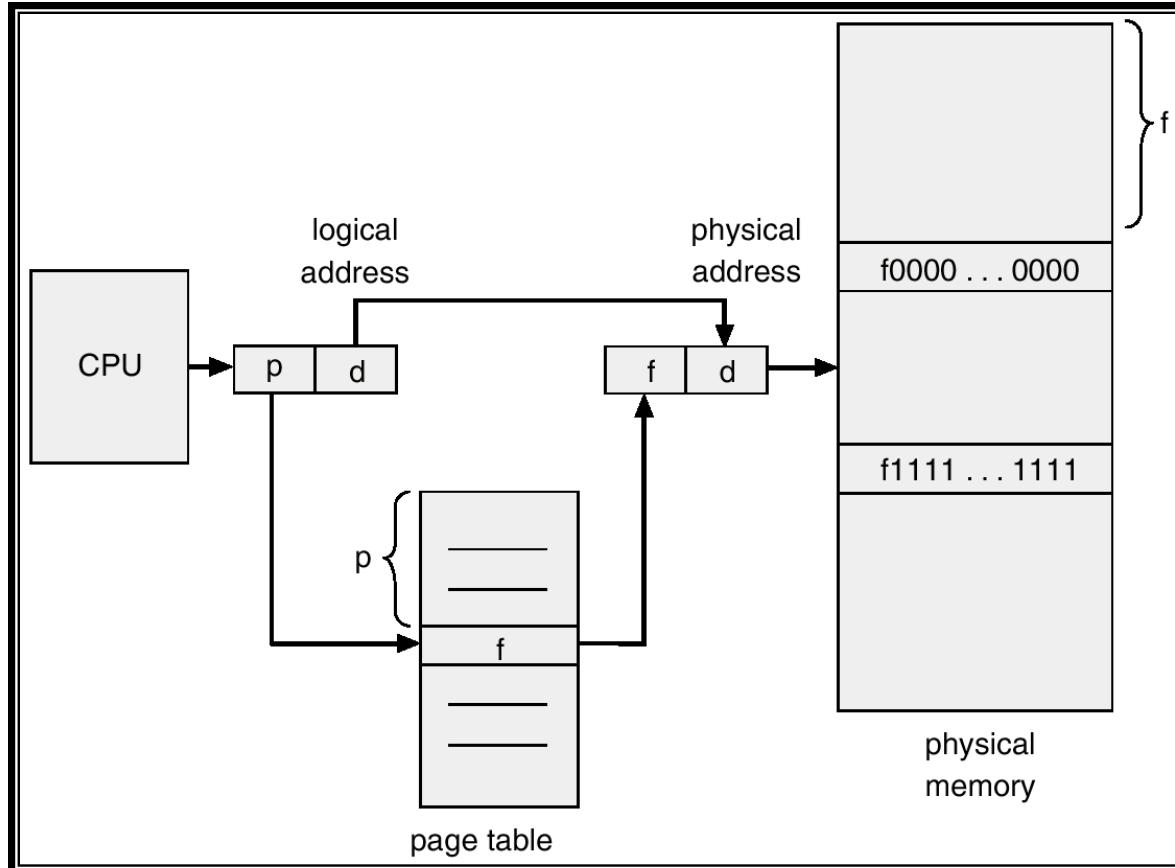
# Paging

---

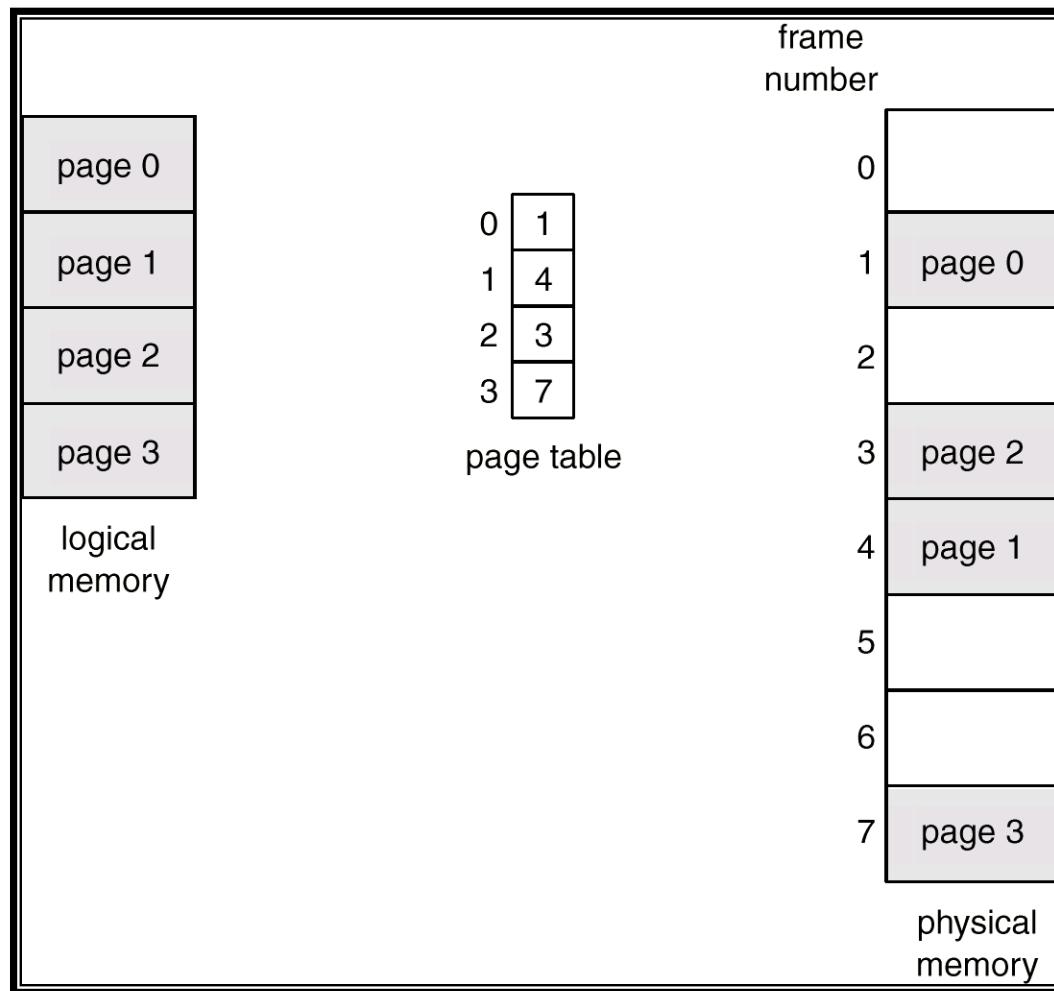
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called **pages**.
- Keep track of all free frames.
- To run a program of size  $n$  pages, need to find  $n$  free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

- 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.

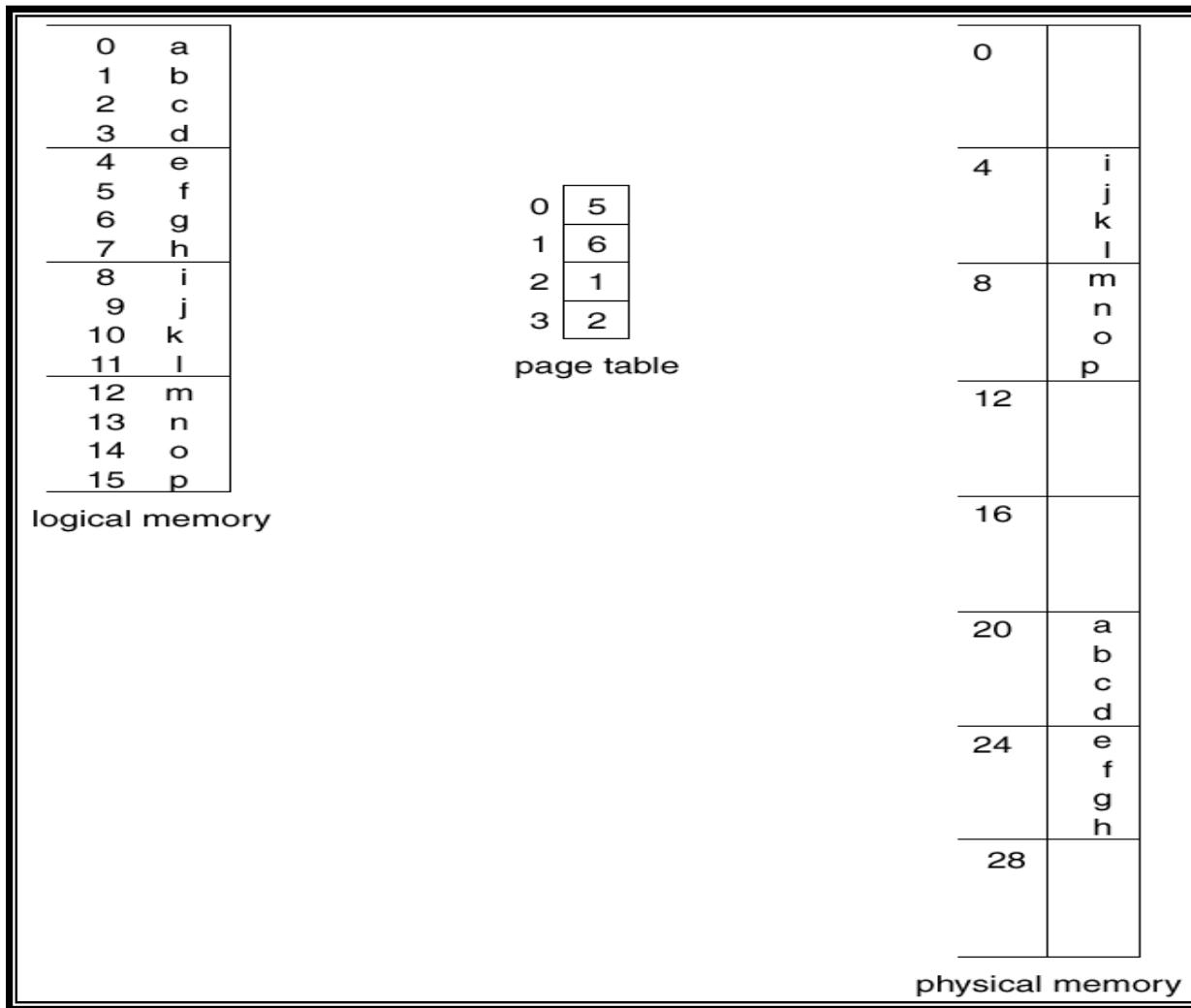
# Address Translation Architecture



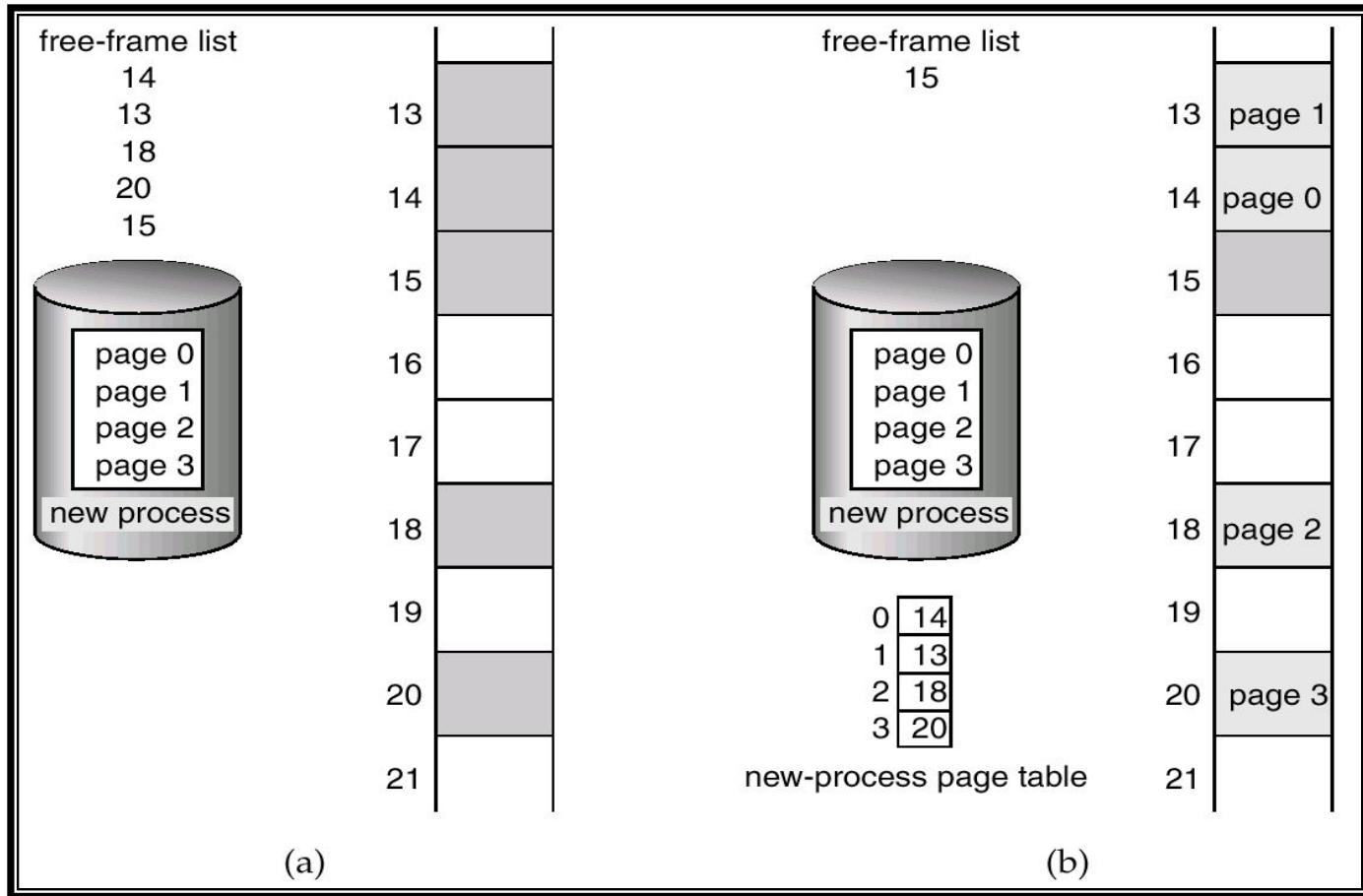
# Paging Example



# Paging Example



# Free Frames



Before allocation

After allocation

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

# Associative Memory

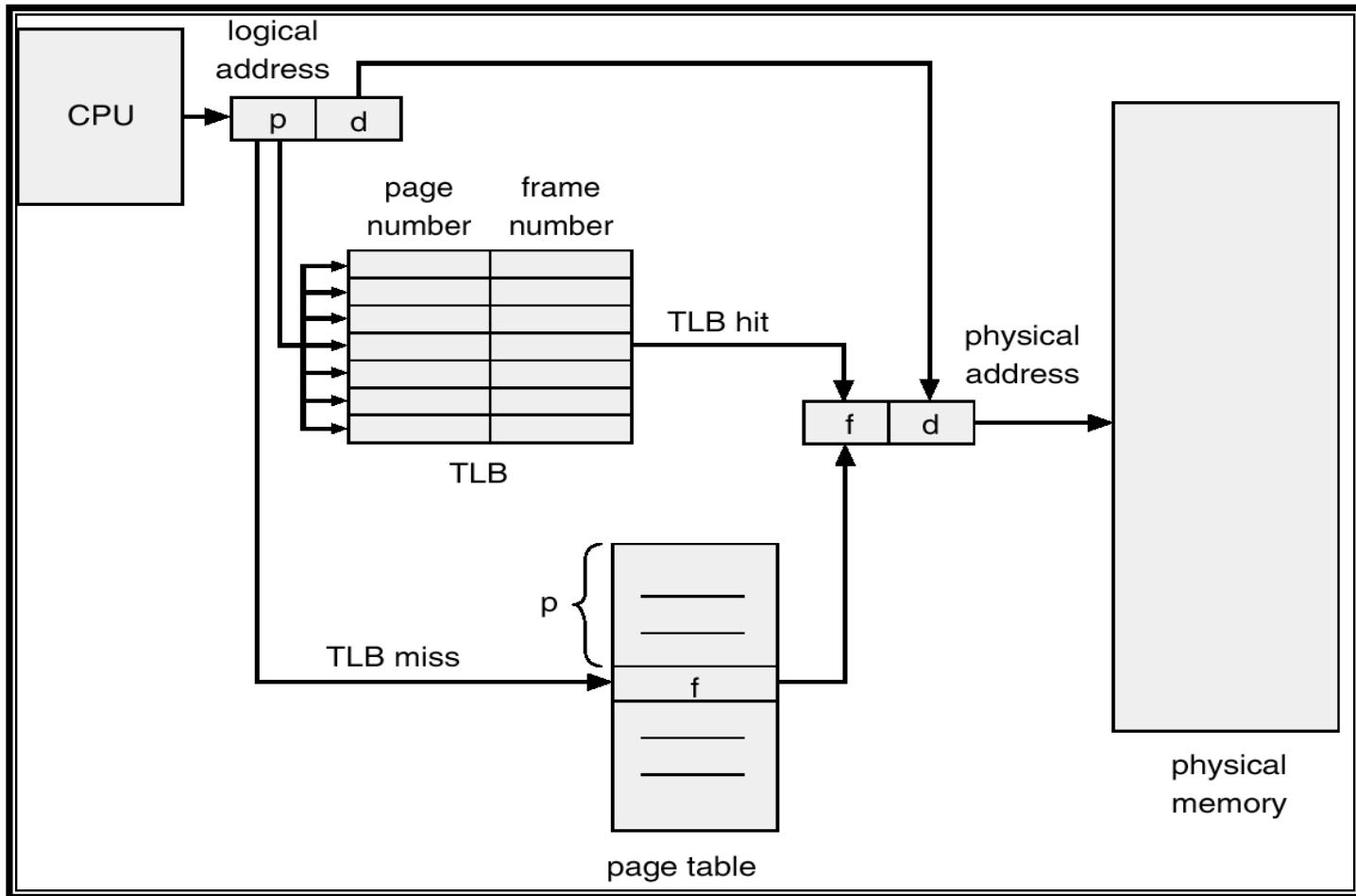
- Associative memory – parallel search

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

Address translation ( $A'$ ,  $A''$ )

- If  $A'$  is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

# Paging Hardware With TLB



# Effective Access Time

---

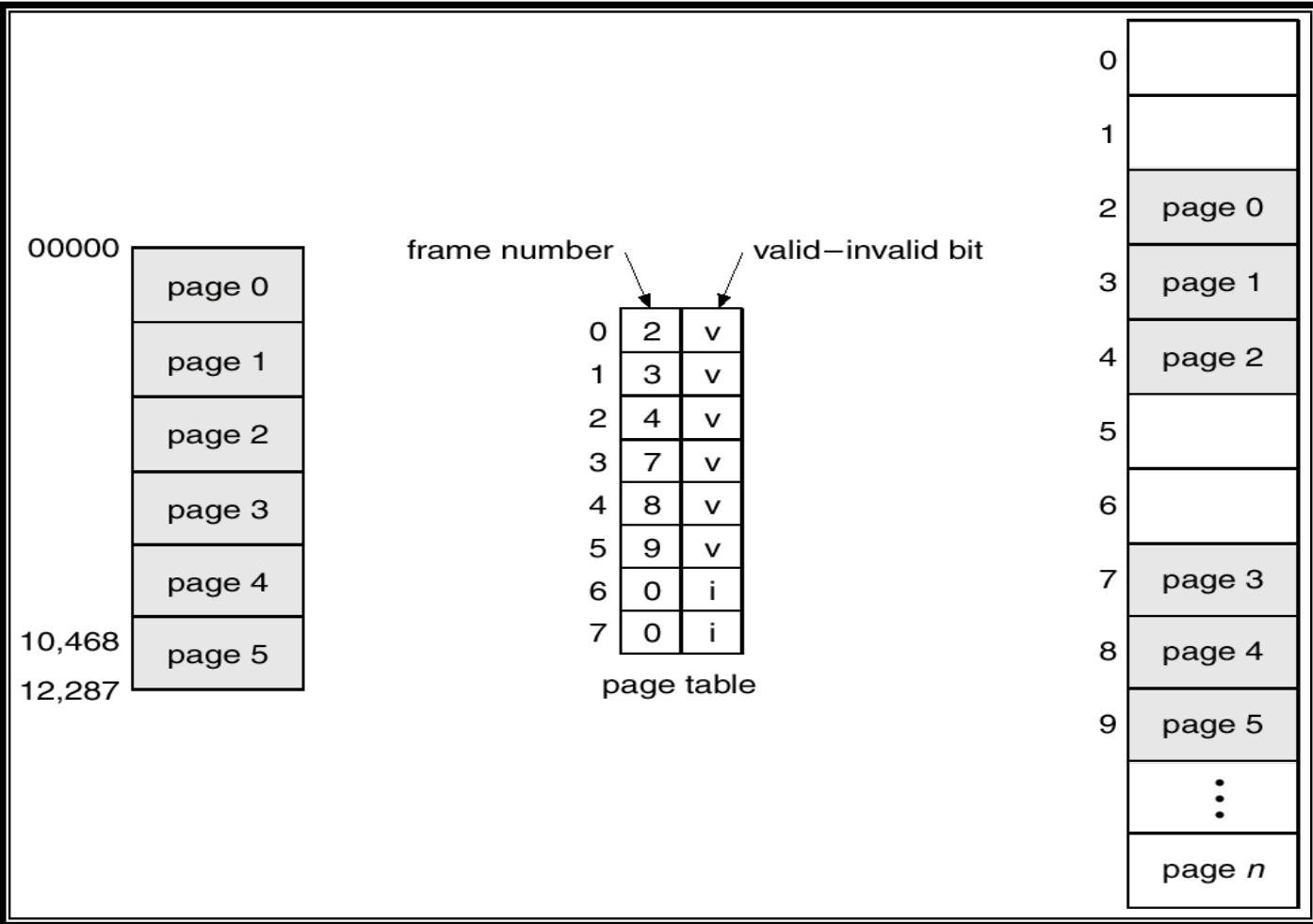
- Associative Lookup =  $\varepsilon$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio =  $\alpha$
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

# 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.



- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

# Hierarchical Page Tables

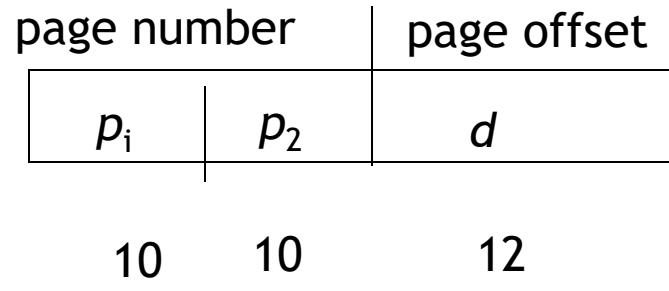
---

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

# Two-Level Paging Example

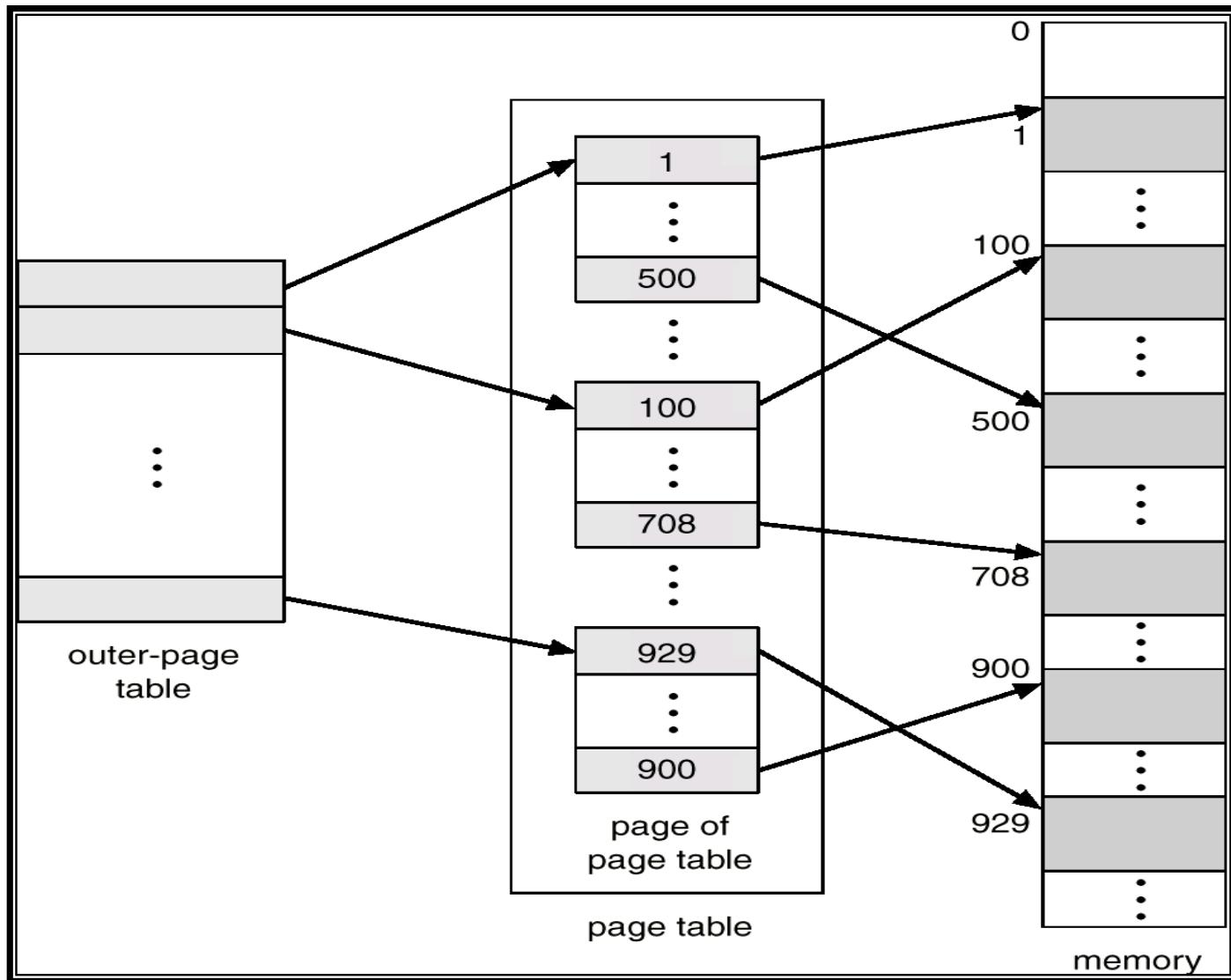
---

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number and a 10-bit page offset.



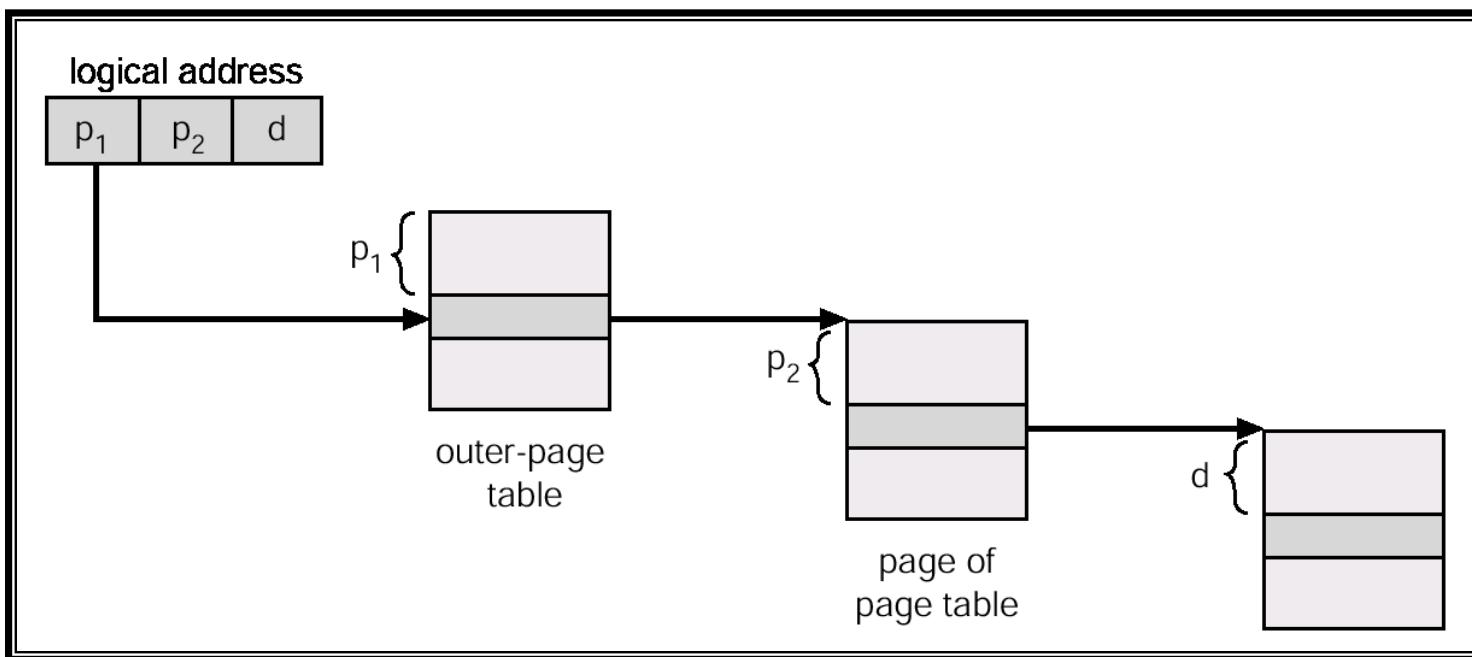
- Thus, a logical address is as follows:  
 where  $p_i$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the outer page table.

# Two-Level Page-Table Scheme



# Address-Translation Scheme

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

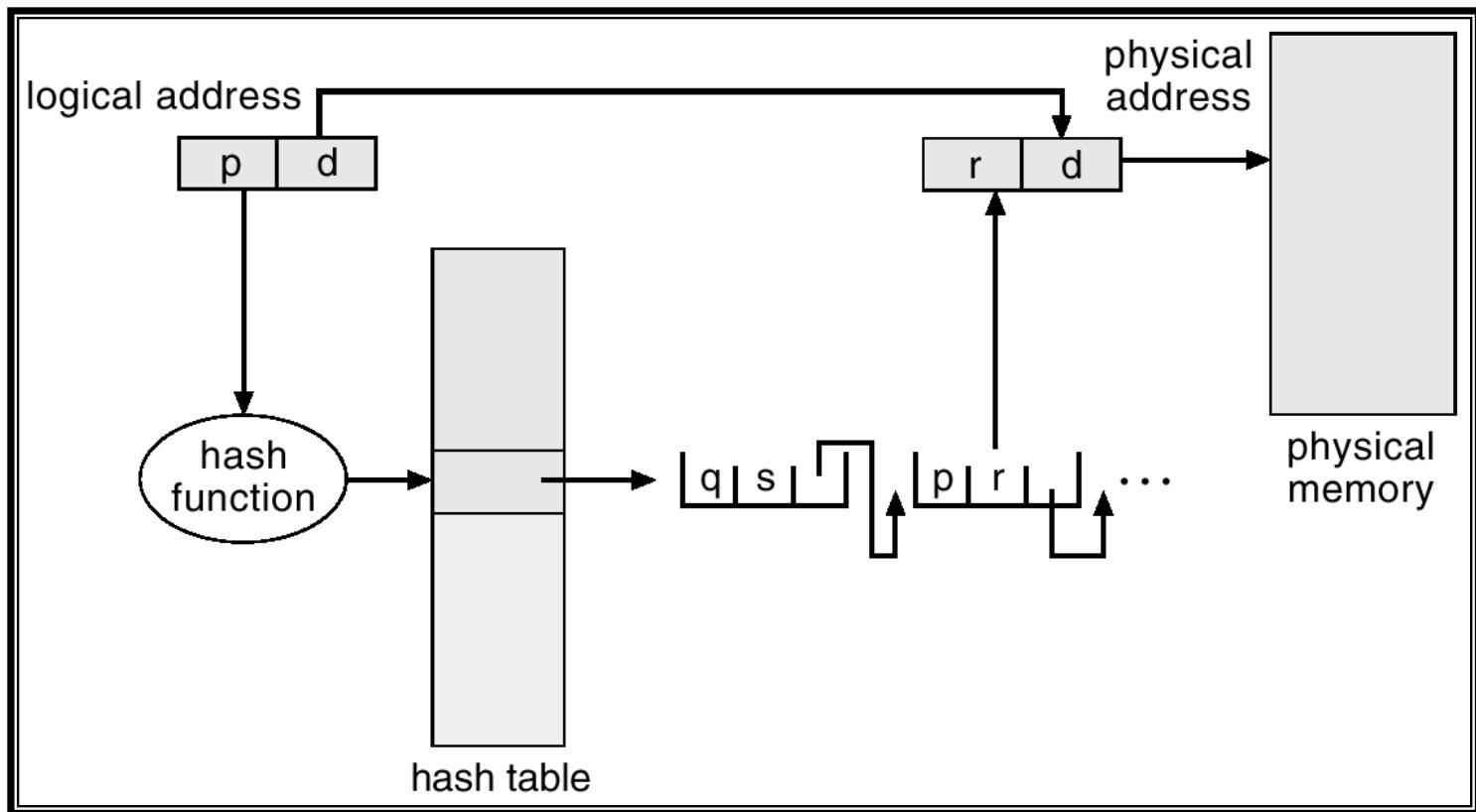


# Hashed Page Tables

---

- Common in 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.
- 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

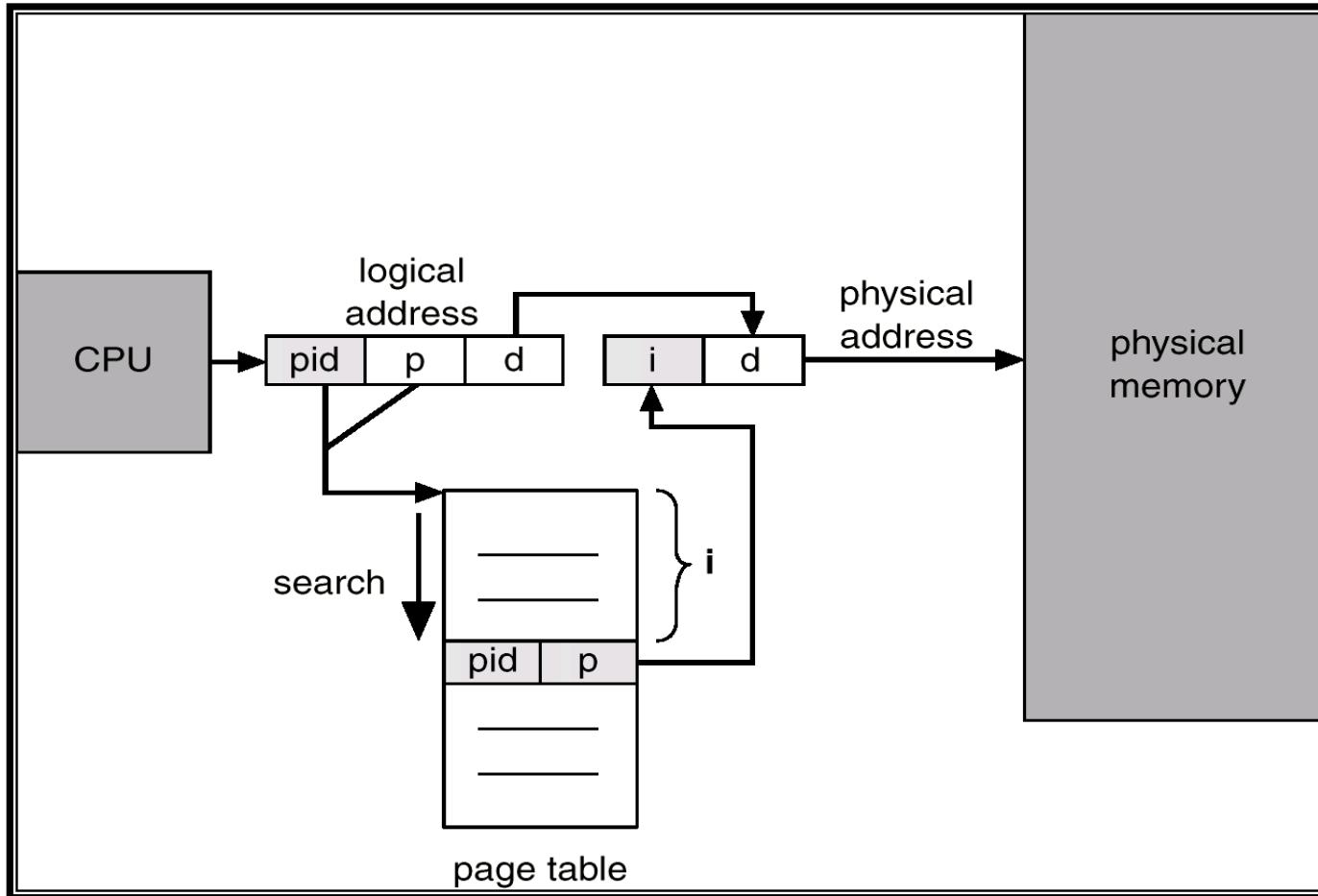


# Inverted Page Table

---

- 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.

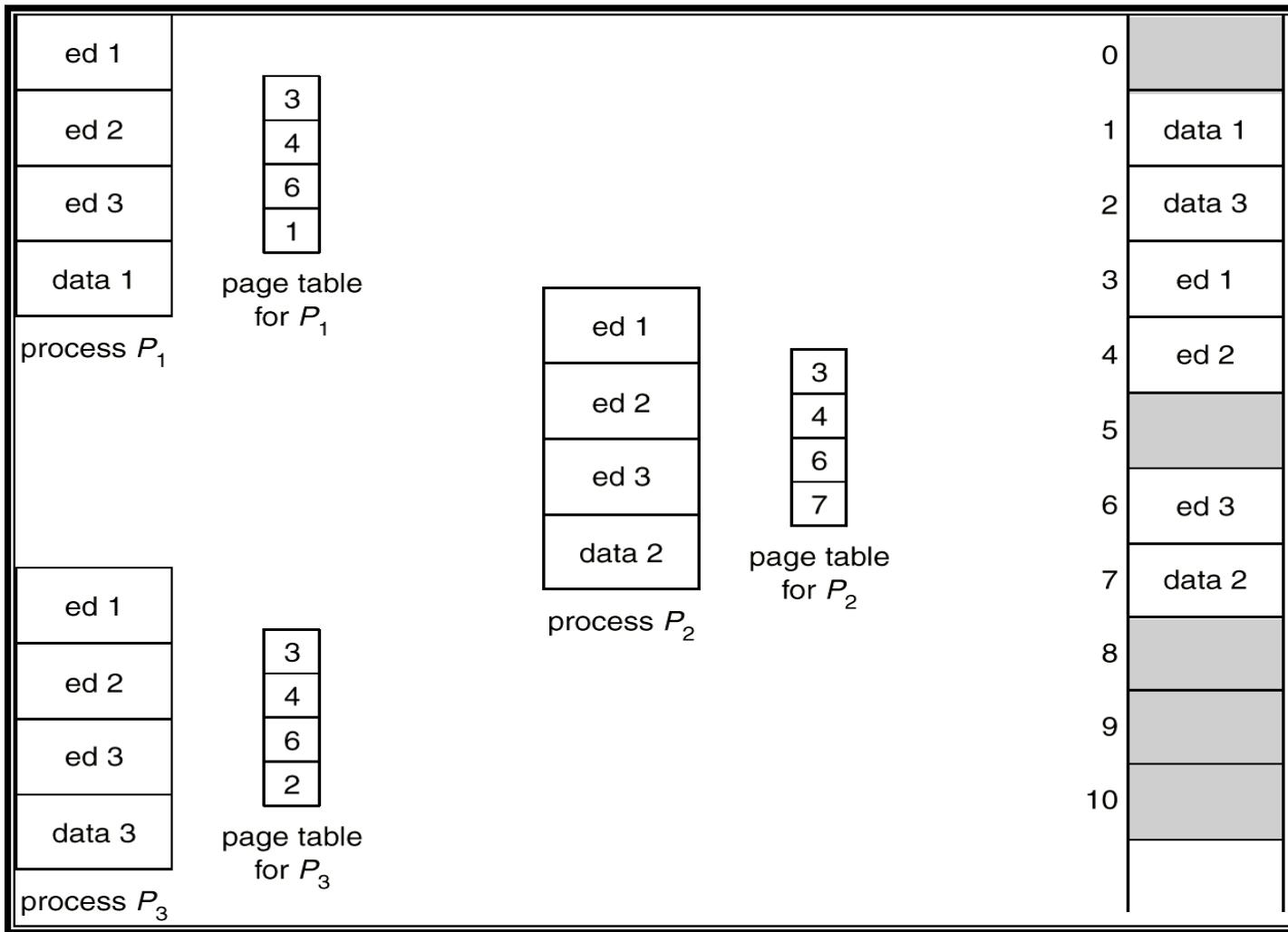
# Inverted Page Table Architecture



# 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



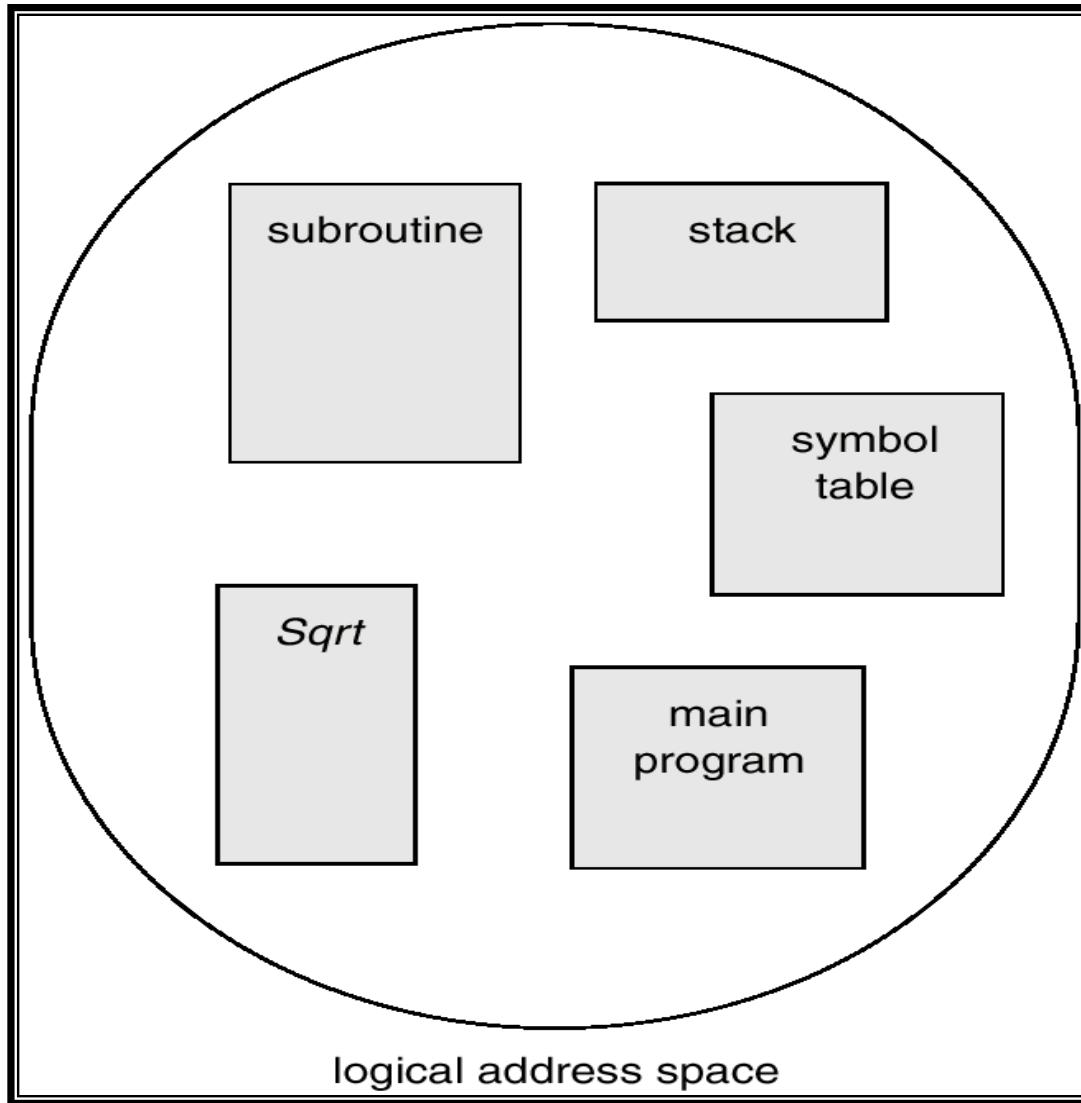
# Segmentation

---

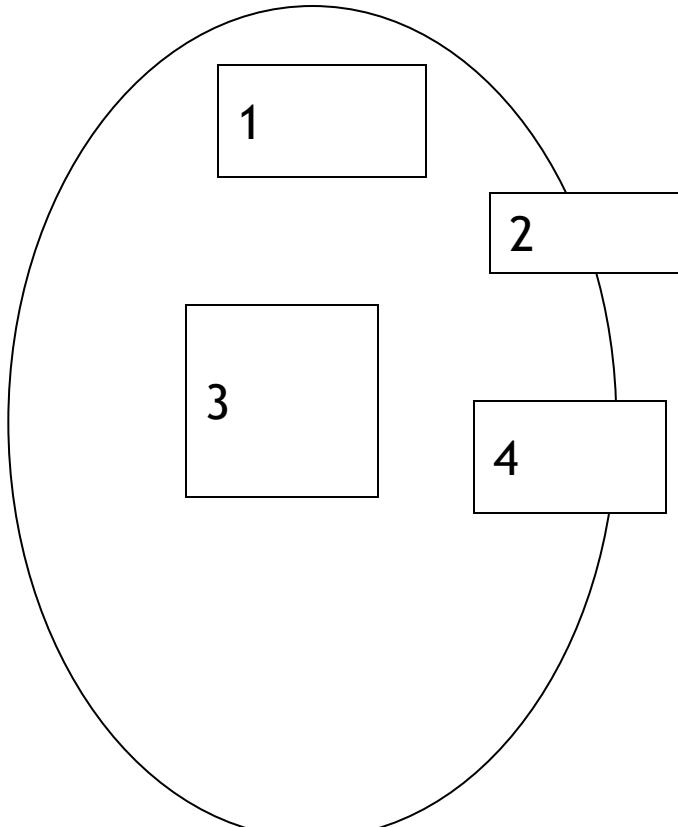
- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:

main program,  
procedure,  
function,  
method,  
object,  
local variables, global variables,  
common block,  
stack,  
symbol table, arrays

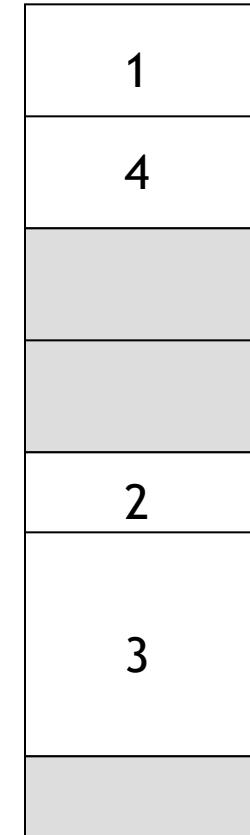
# User's View of a Program



# Logical View of Segmentation



user space



physical memory space

# Segmentation Architecture

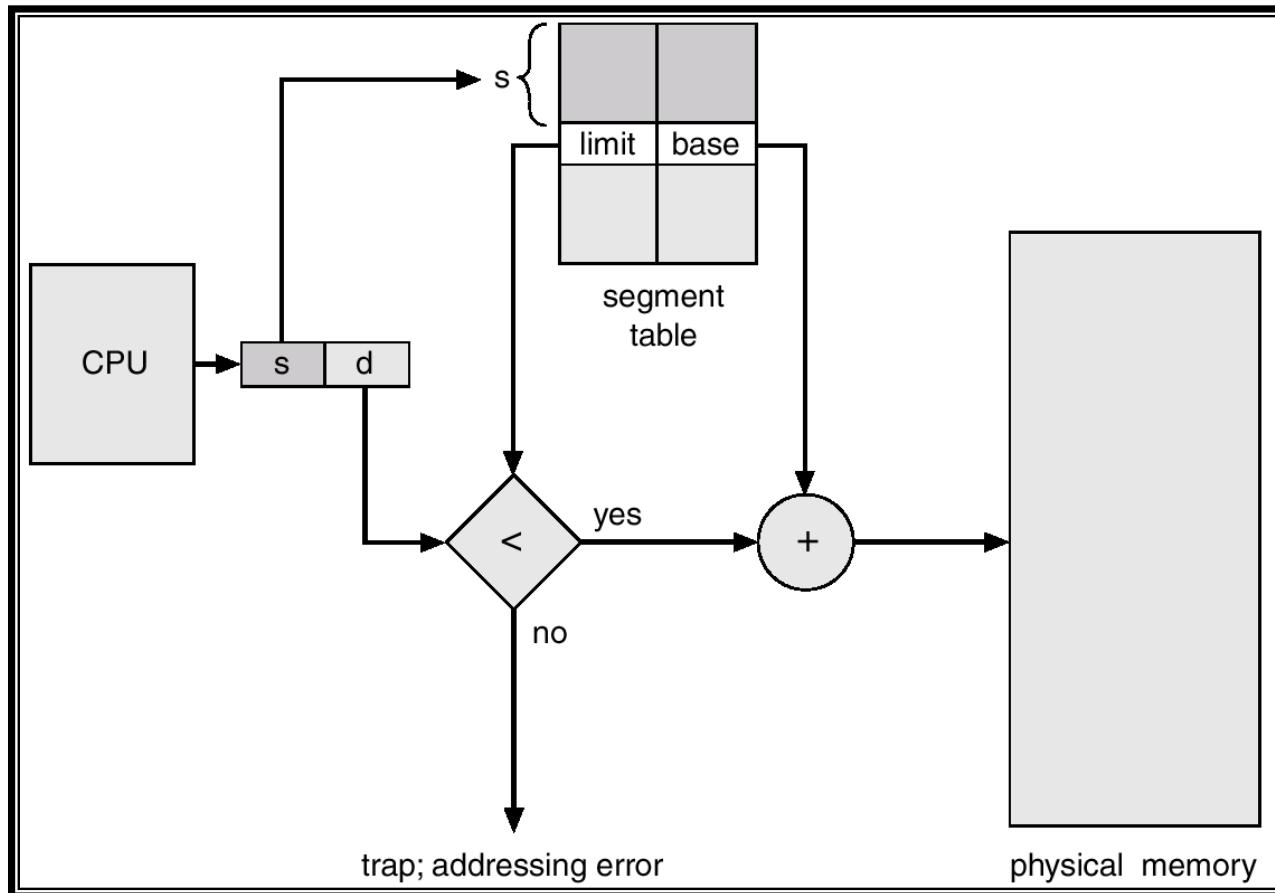
---

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

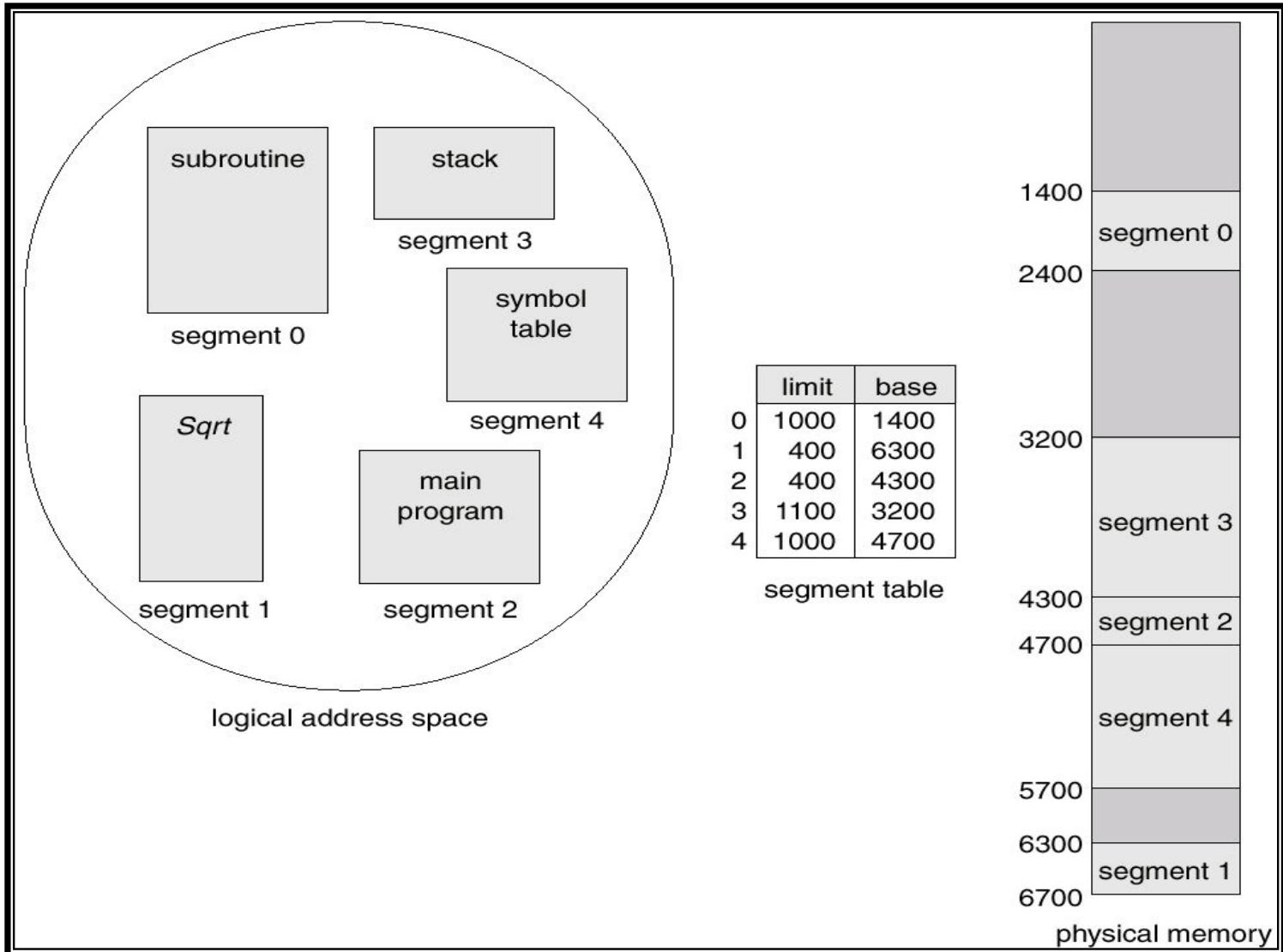
- ▶ Relocation.
  - ▶ dynamic
  - ▶ by segment table
- ▶ Sharing.
  - ▶ shared segments
  - ▶ same segment number
- ▶ Allocation.
  - ▶ first fit/best fit
  - ▶ external fragmentation

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

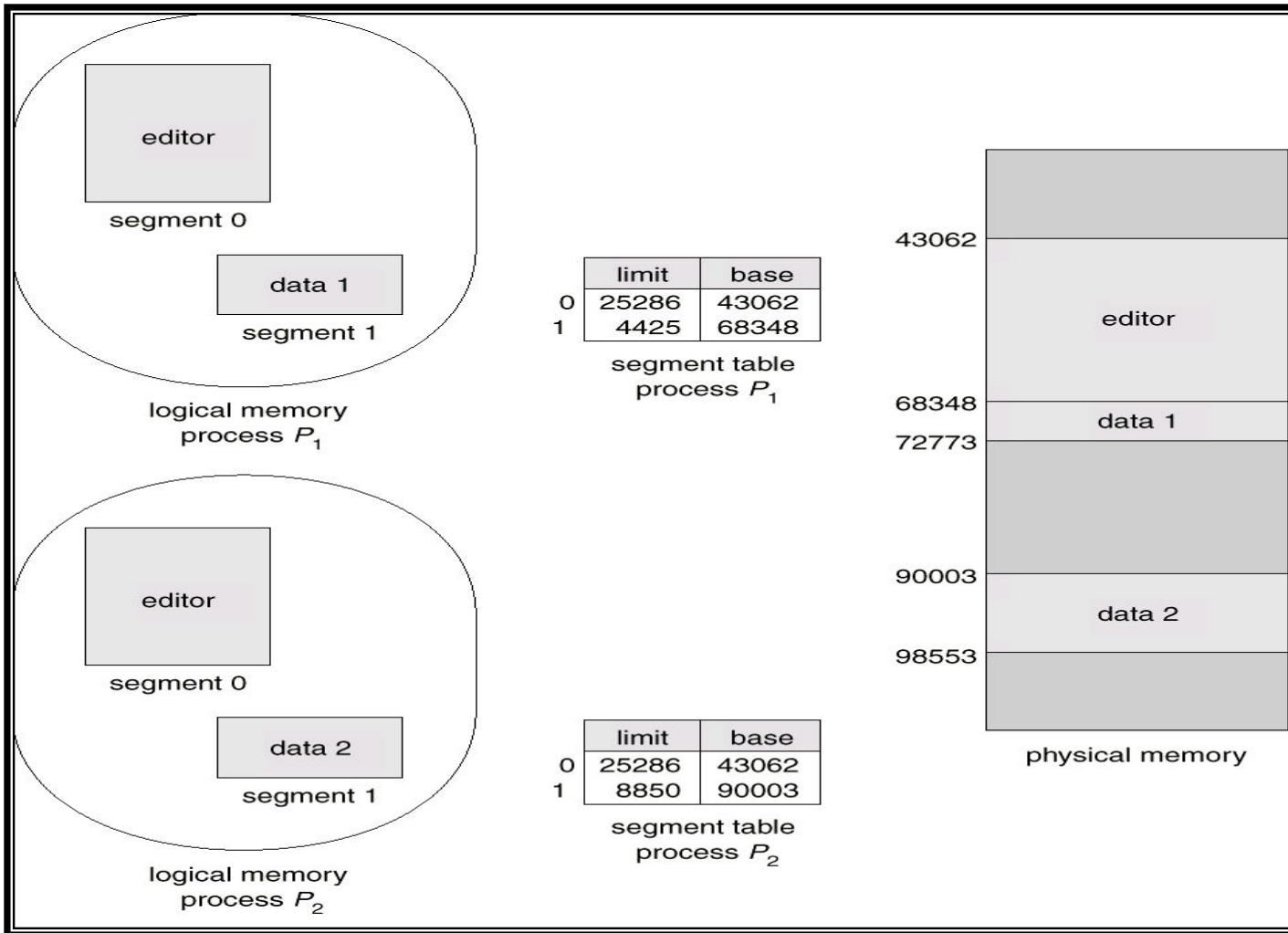
# Segmentation Hardware



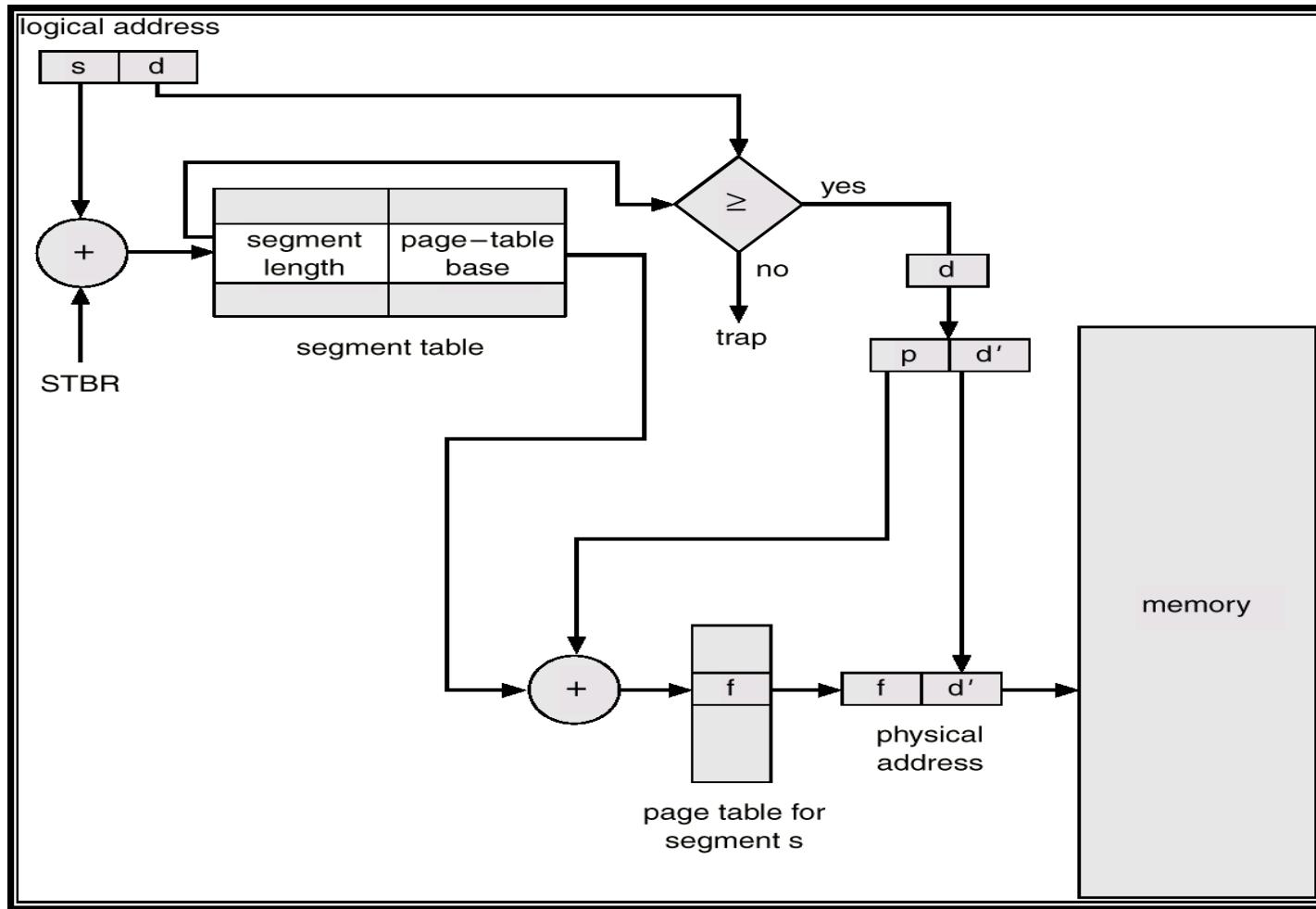
# Example of Segmentation



# Sharing of Segments

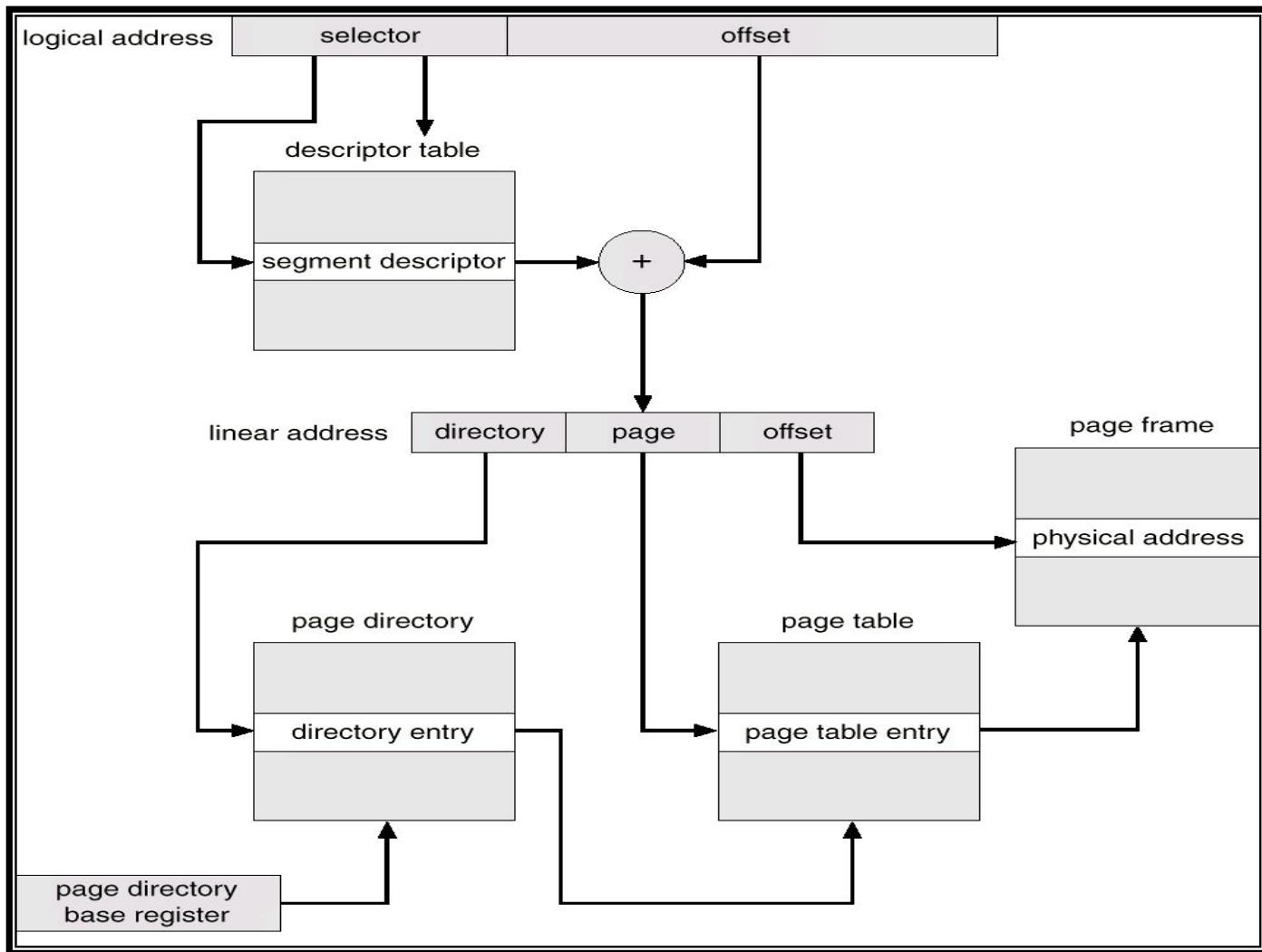


- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.



- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

# Intel 30386 Address Translation



# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## **CSE-202 OPERATING SYSTEM**

**Module III: Memory Management**  
**Page Replacement Algorithms**

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling , Pearson Education**

## **Course Learning Objectives:**

- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.

# Topics Covered

---

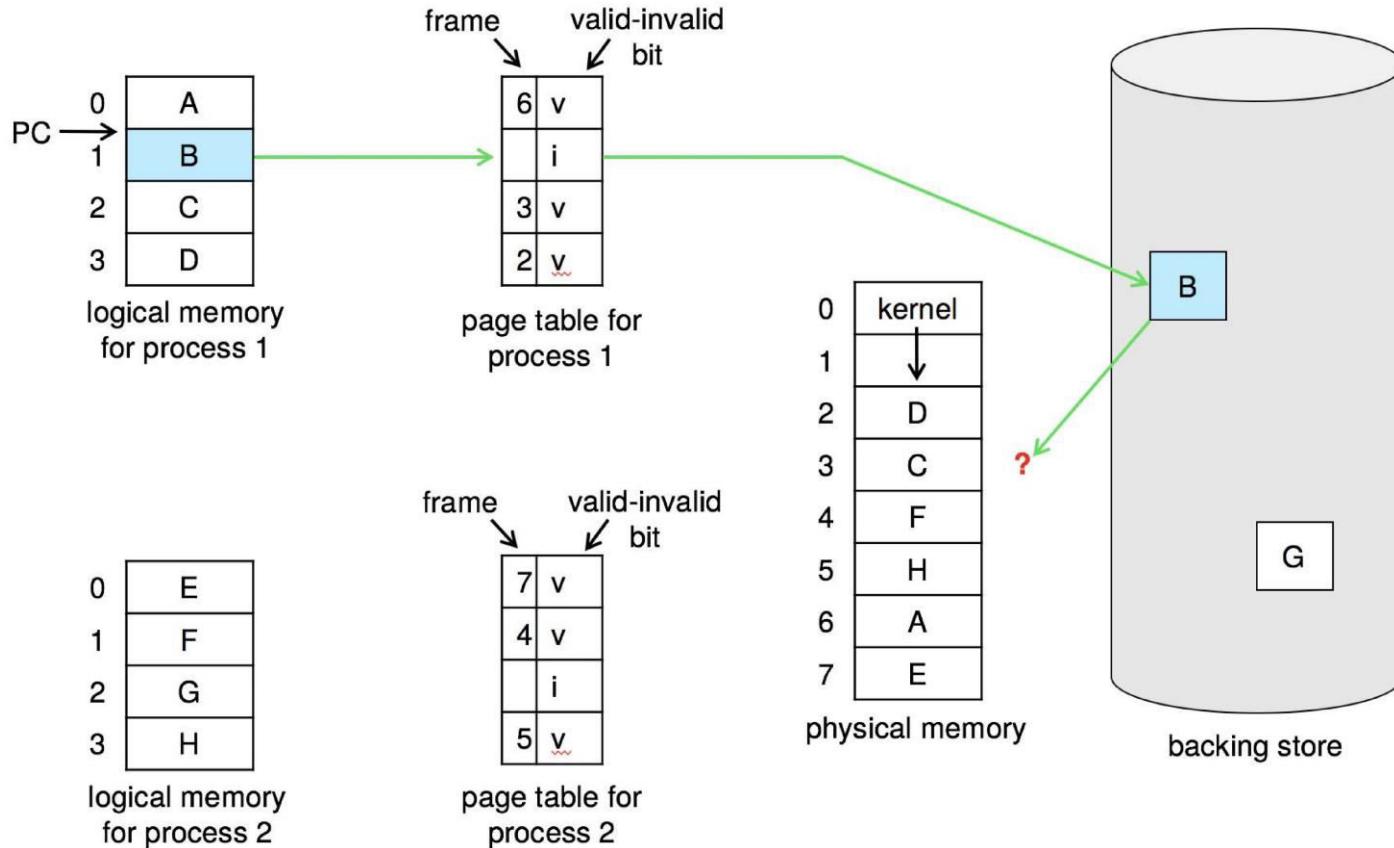
- Page Replacement
- Allocation of Frames

# Page Replacement

---

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

# Need For Page Replacement



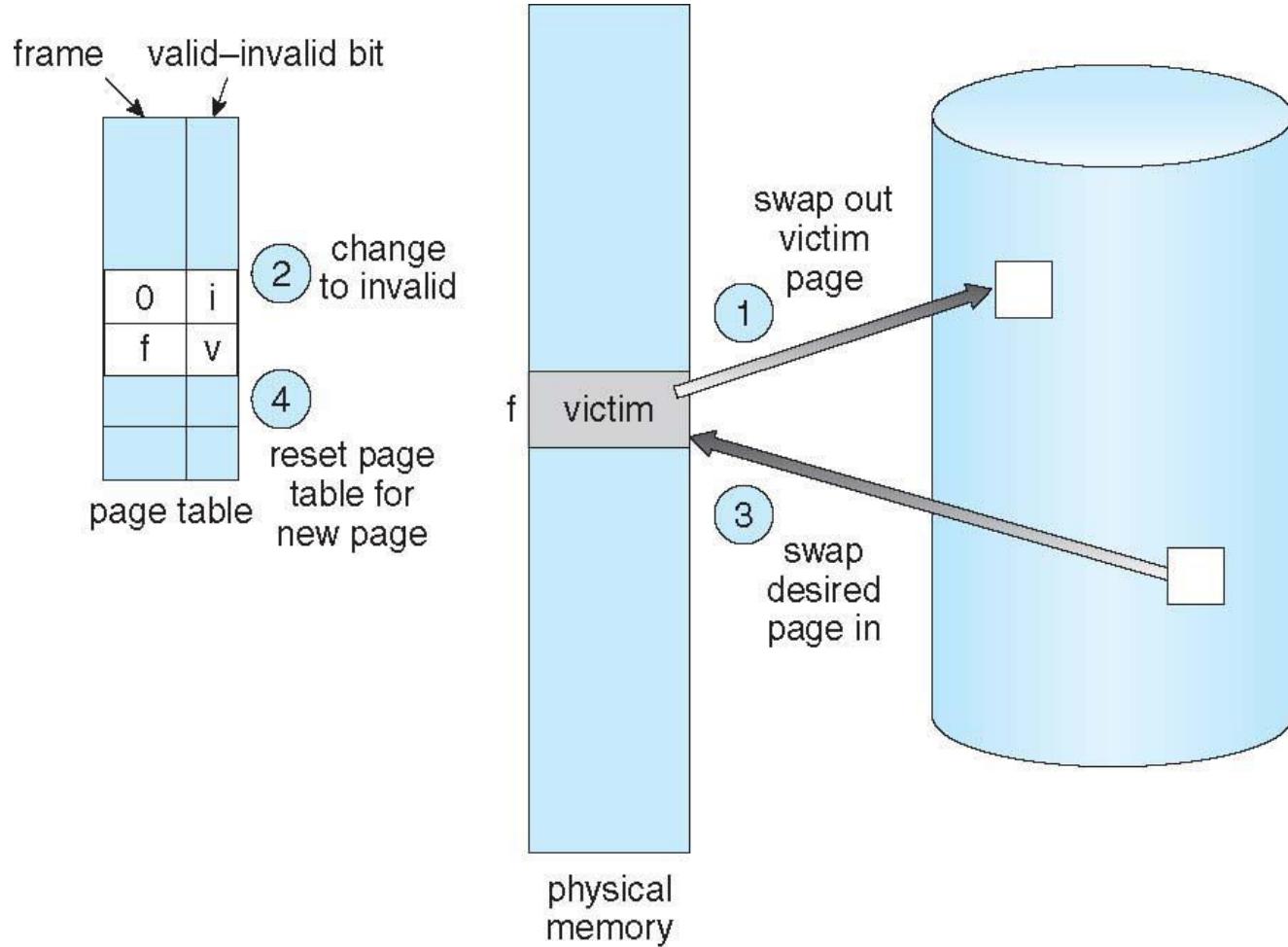
# Basic Page Replacement

---

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

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

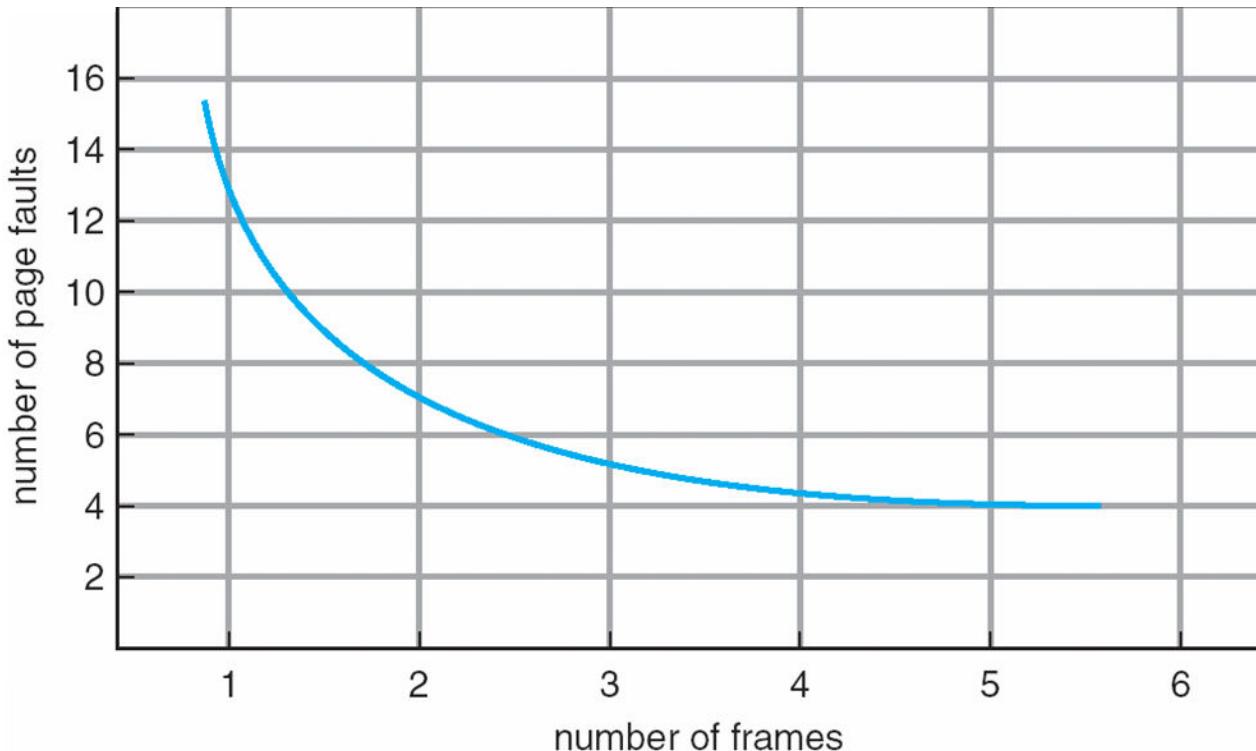
# Page Replacement



- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

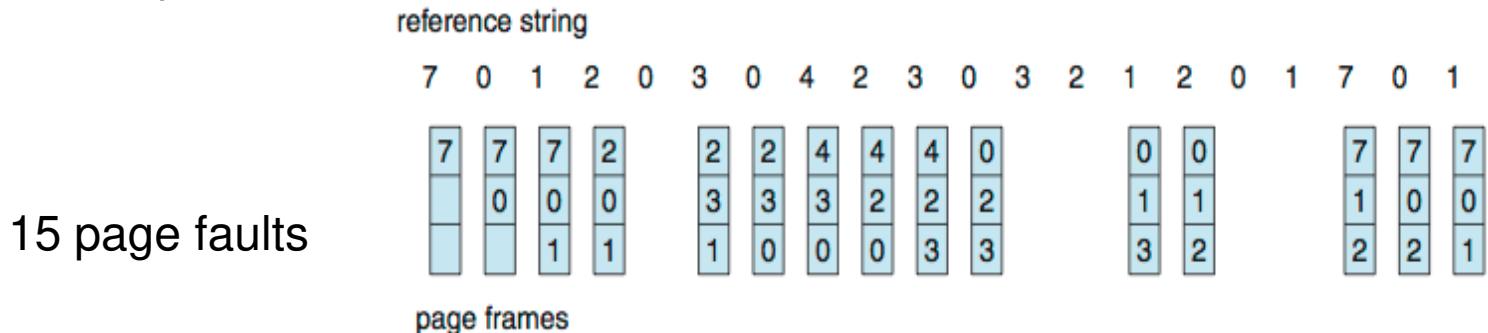
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

# Graph of Page Faults Versus the Number of Frames



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

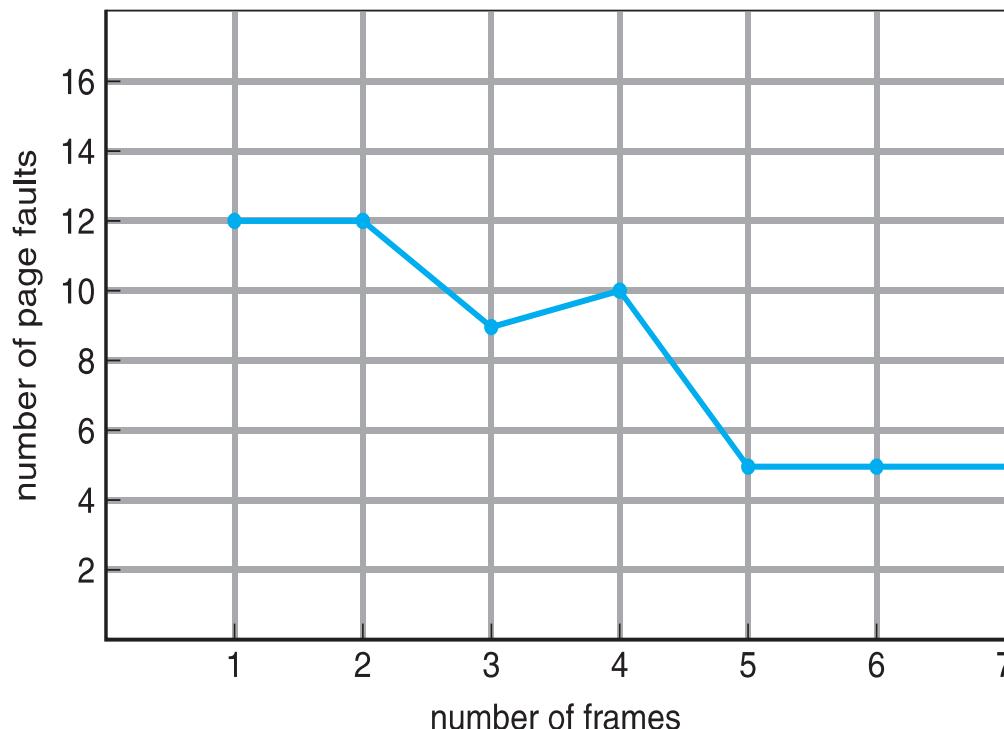
- Reference string:  
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- How to track ages of pages?
  - Just use a FIFO queue

# Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
- Graph illustrating Belady's Anomaly

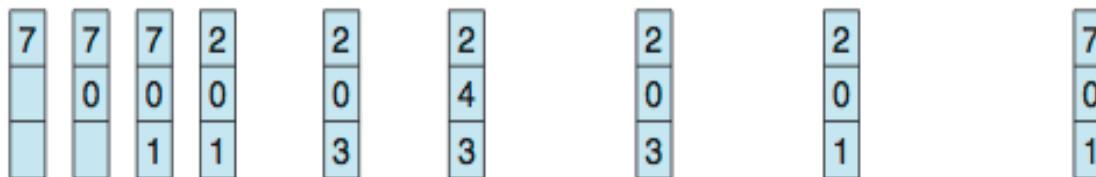


# Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs
- Optimal is an example of **stack algorithms** that don't suffer from Belady's Anomaly

reference string

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



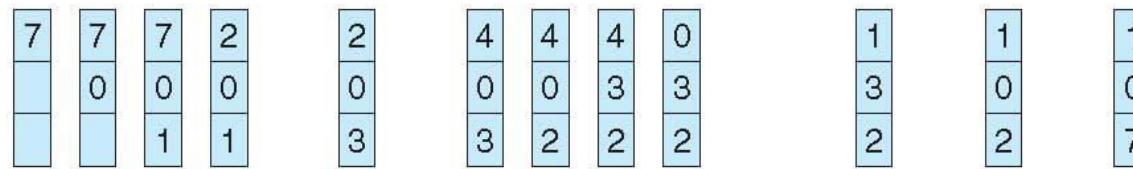
page frames

# Least Recently Used (LRU) Algorithm

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

reference string

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



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus it does not suffer from Belady's Anomaly

# LRU Algorithm Implementation

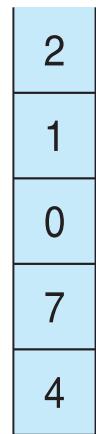
- Time-counter implementation
  - Every page entry has a time-counter variable; every time a page is referenced through this entry, copy the value of the clock into the time-counter
  - When a page needs to be changed, look at the time-counters to find smallest value
    - Search through a table is needed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

# Stack Implementation

- Use of a stack to record most recent page references

reference string

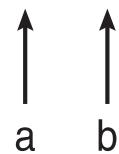
4 7 0 7 1 0 1 2 1 2 7 1 2



stack  
before  
a



stack  
after  
b



# Counting Algorithms

---

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used (LFU) Algorithm:**
  - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Applications and Page Replacement

---

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e., databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can provide direct access to the disk, getting out of the way of the applications
  - **Raw disk mode**
- Bypasses buffering, locking, etc.

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

# Summary

- We discussed how pages are loaded into memory using demand paging.
- We discussed the FIFO, optimal, and LRU page-replacement algorithms.

**Thank You**

## **CSE-202 OPERATING SYSTEM**

### **Module III: Memory Management**

### **Virtual Memory**

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling , Pearson Education**

## **Course Learning Objectives:**

- To understand the benefits of a virtual memory system
- To understand the concept of demand paging
- To understand how frames can be allocated to different processes
- To understand the consequence of not allocating enough frames to process

# Topics to be Covered

---

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

# Background

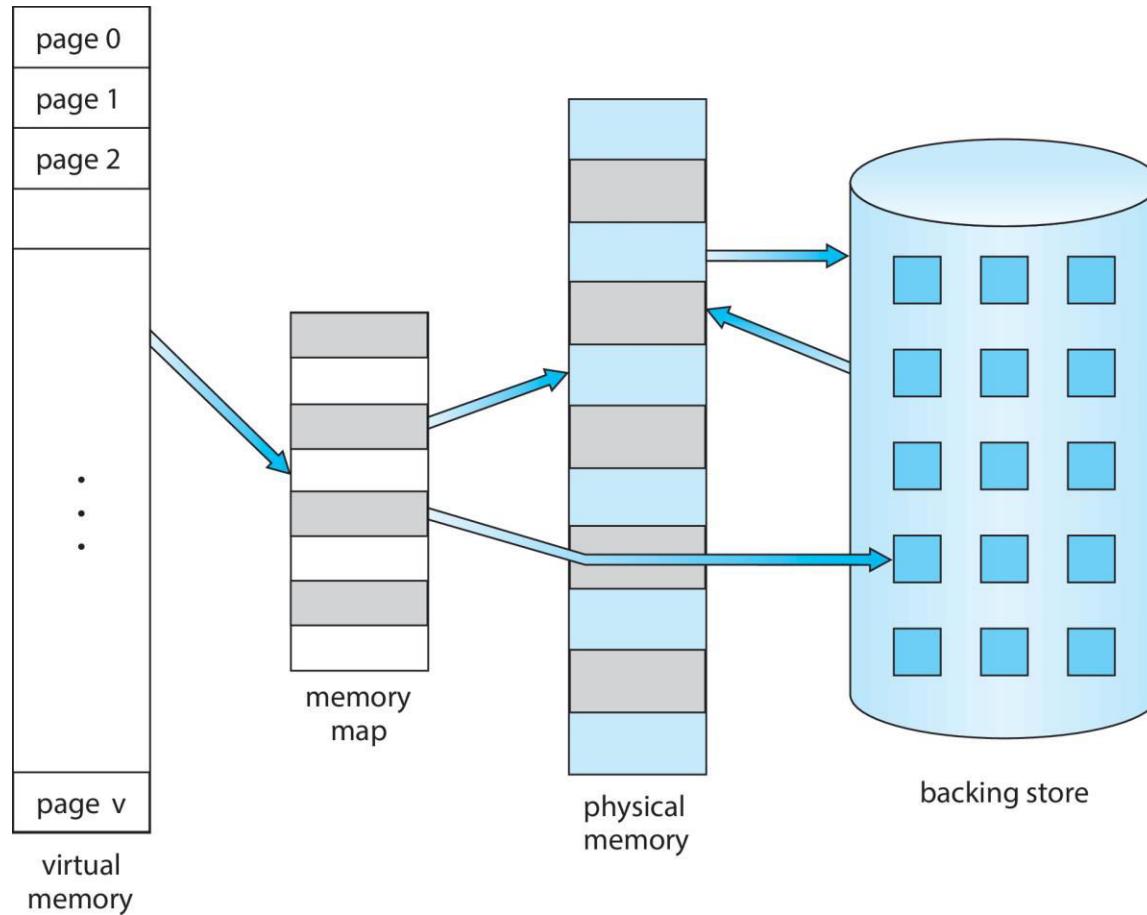
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each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

# Virtual memory

- **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

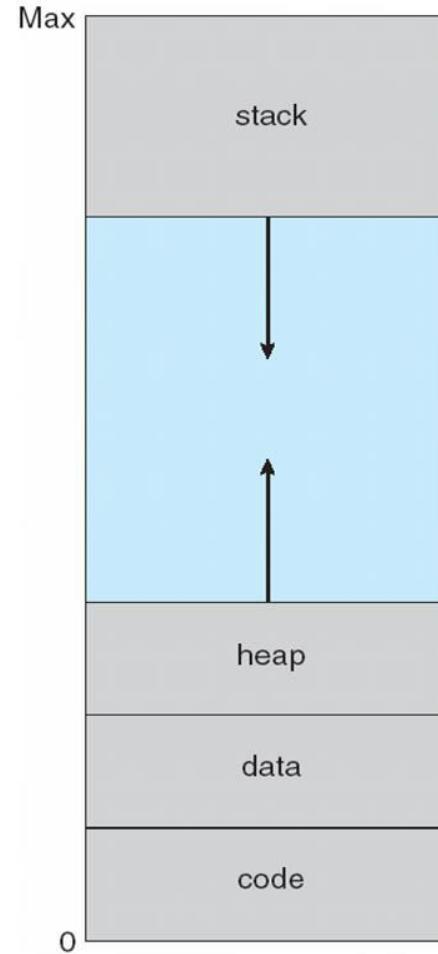
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory That is Larger Than Physical Memory

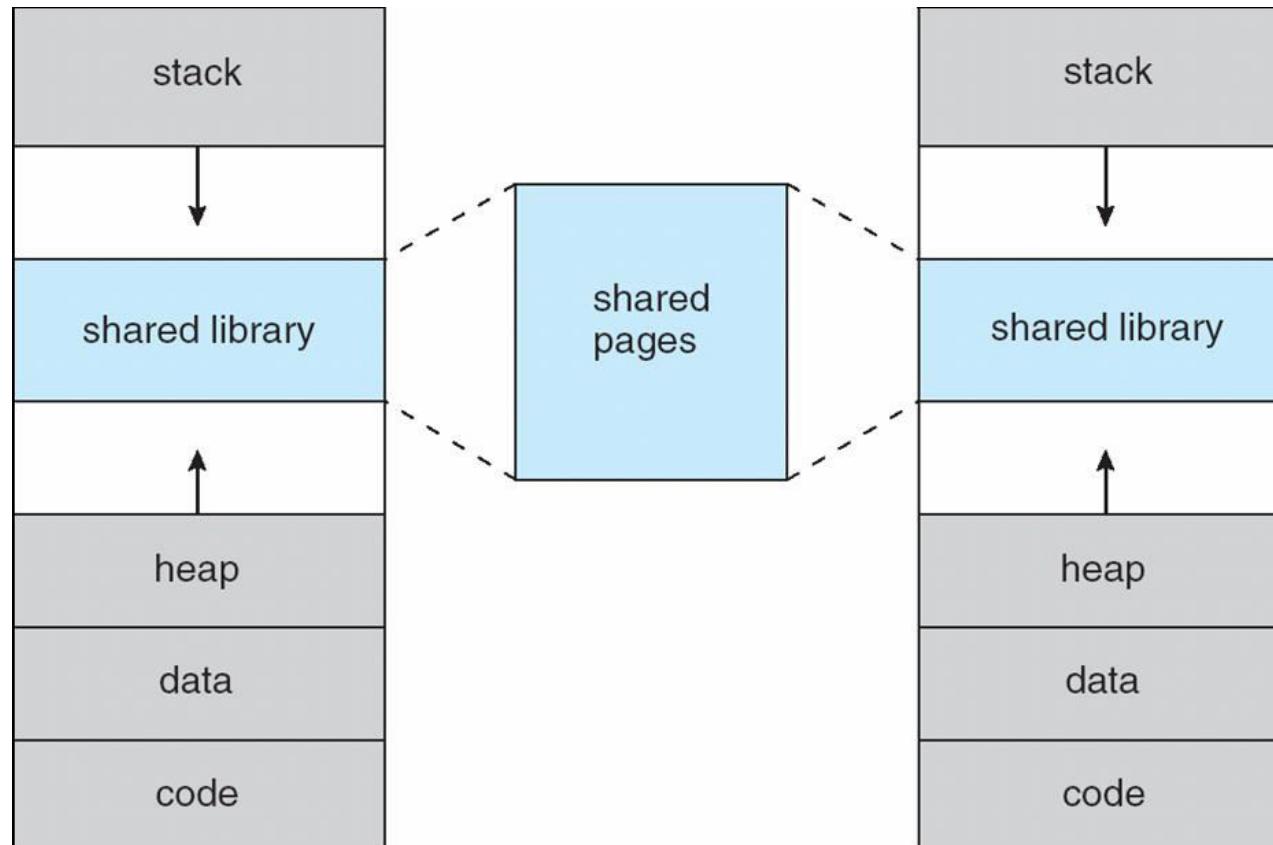


# Virtual-address Space

- Usually design logical address space for the stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during **fork()**, speeding process creation

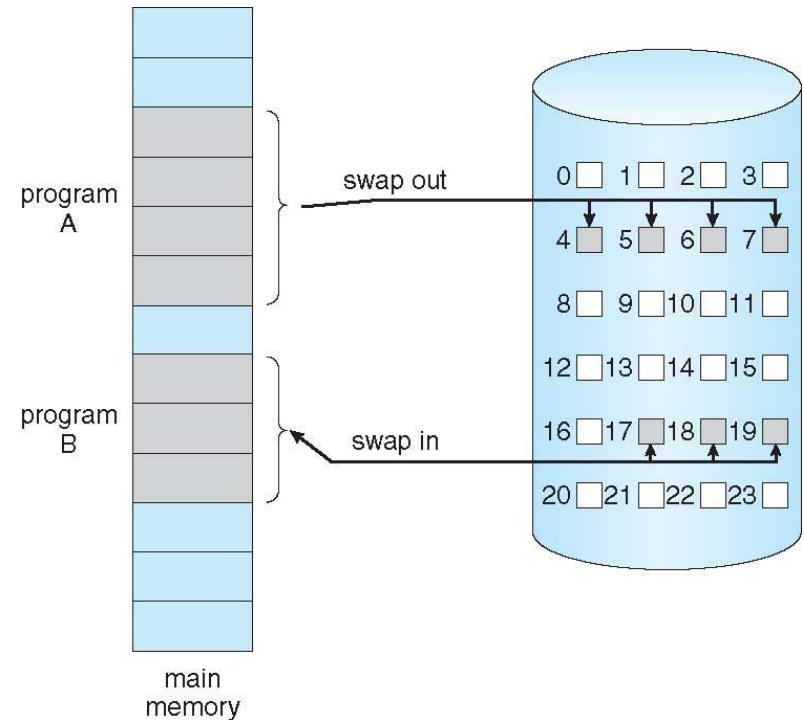


# Shared Library Using Virtual Memory



# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- invalid reference  $\Rightarrow$  abort
  - Not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



# Basic Concepts

- With swapping, the pager guesses which pages will be used before swapping them out again
- How to determine that set of pages?
- Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code
- Use page table with valid-invalid bit (see chapter 9)

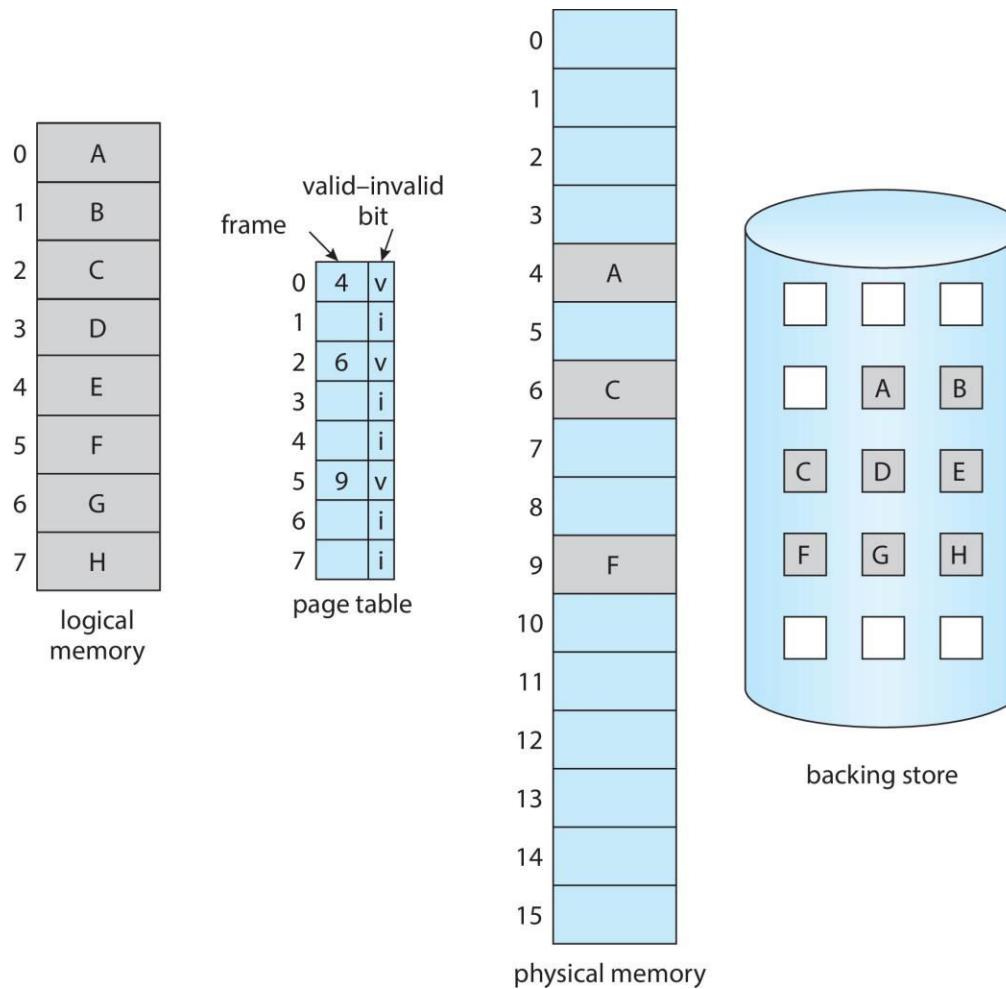
# Page table with Valid-Invalid Bit

- With each page table entry, a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:  
During MMU address translation, if valid–invalid bit in the page table entry is **i** ⇒ page fault

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

page table

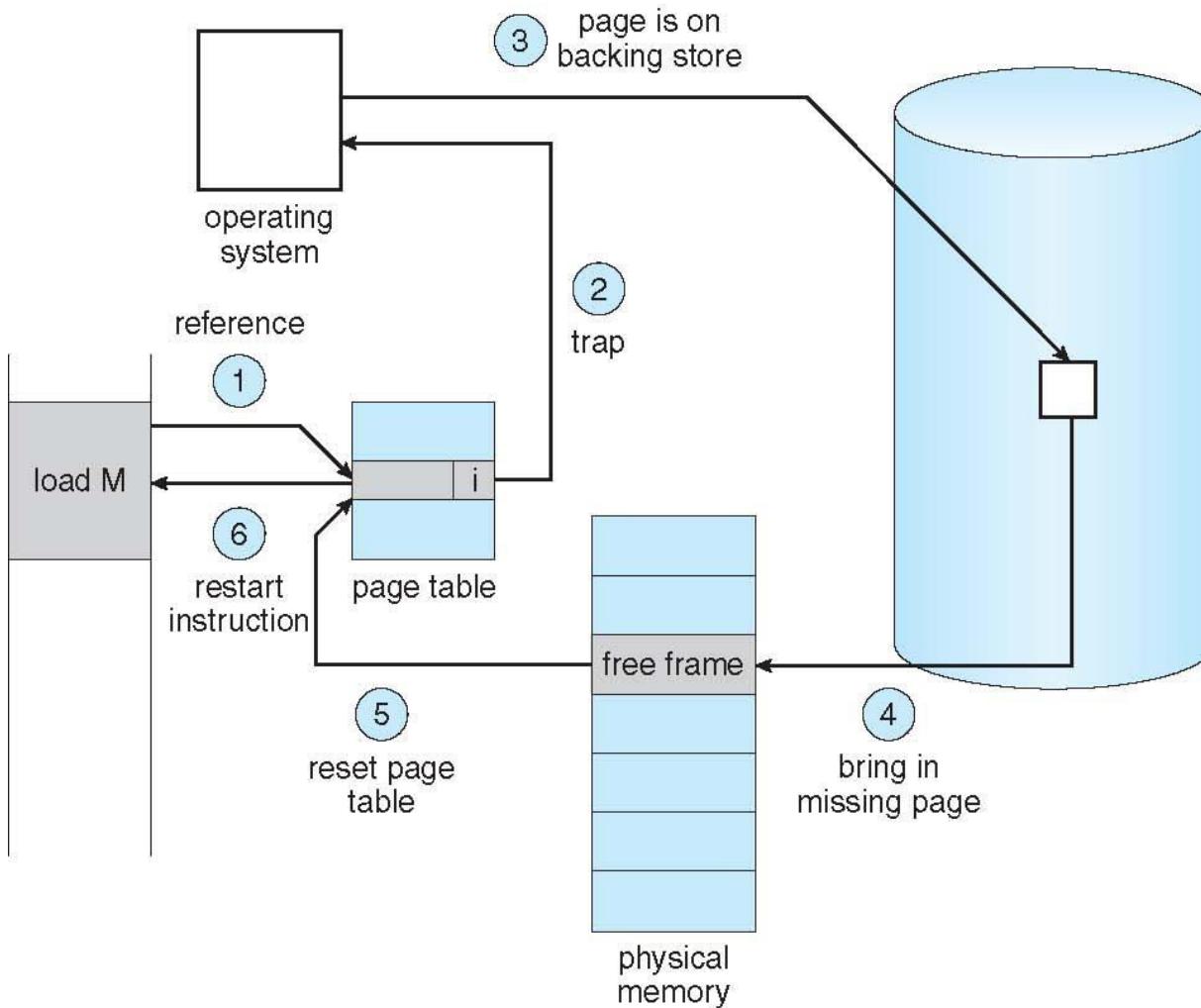
# Page Table When Some Pages Are Not in Main Memory



# Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
  - Page fault
2. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation
5. Reset 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 (Cont.)

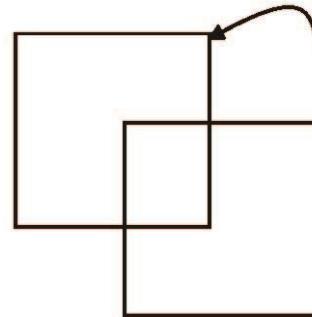


# Aspects of Demand Paging

- **Pure demand paging:** 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
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

# Instruction Restart

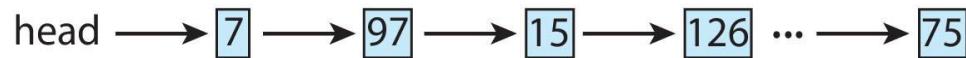
- Consider an instruction that could access several different locations
  - Block move



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

# Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

# Stages in Demand Paging – Worse Case

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:
  - a) Wait in a queue for this device until the read request is serviced
  - b) Wait for the device seek and/or latency time
  - c) Begin the transfer of the page to a free frame

# Stages in Demand Paging (Cont.)

---

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

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Input the page from disk – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault
- Effective Access Time (EAT)  
$$\text{EAT} = (1 - p) \times \text{memory access}$$
$$+ p (\text{page fault overhead})$$
$$+ \text{swap page out}$$
$$+ \text{swap page in } )$$

# Demand Paging Example

---

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$   
 $= (1 - p) \times 200 + p \times 8,000,000$   
 $= 200 + p \times 7,999,800$
- 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 Optimizations

---

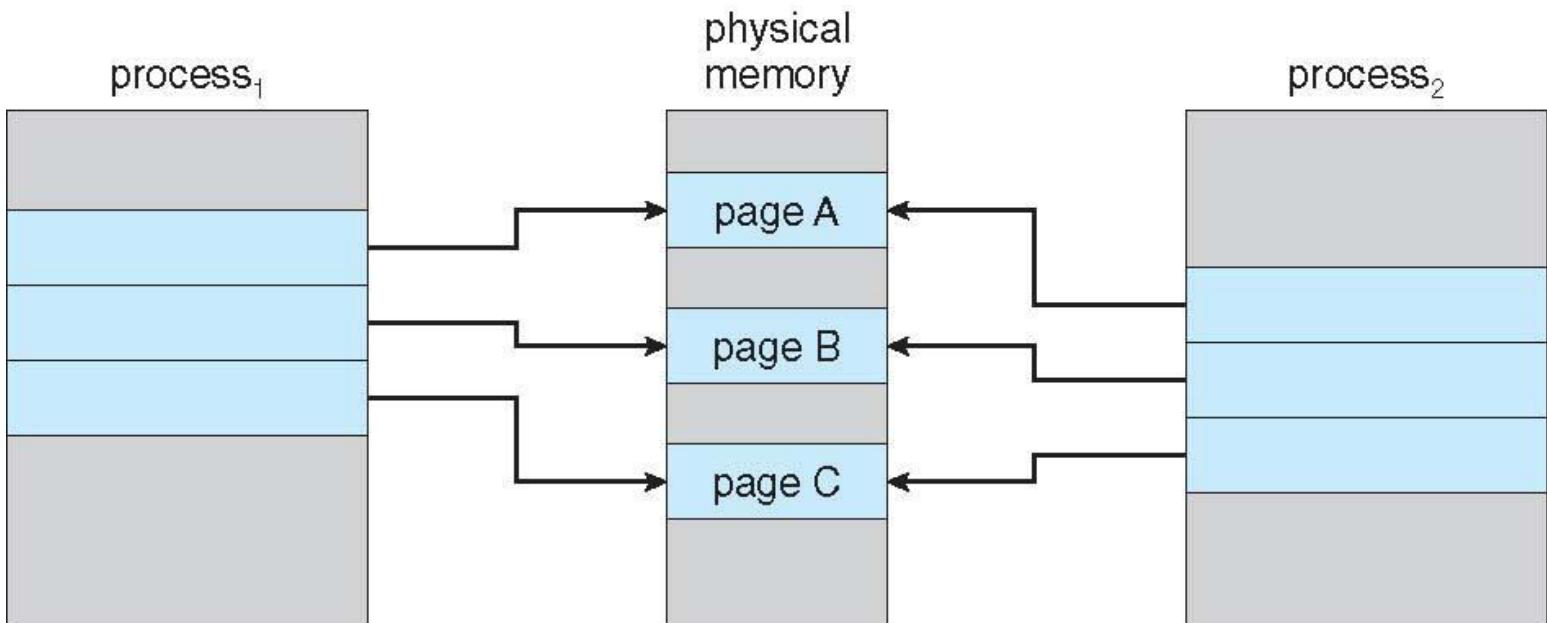
- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks; less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

# Copy-on-Write

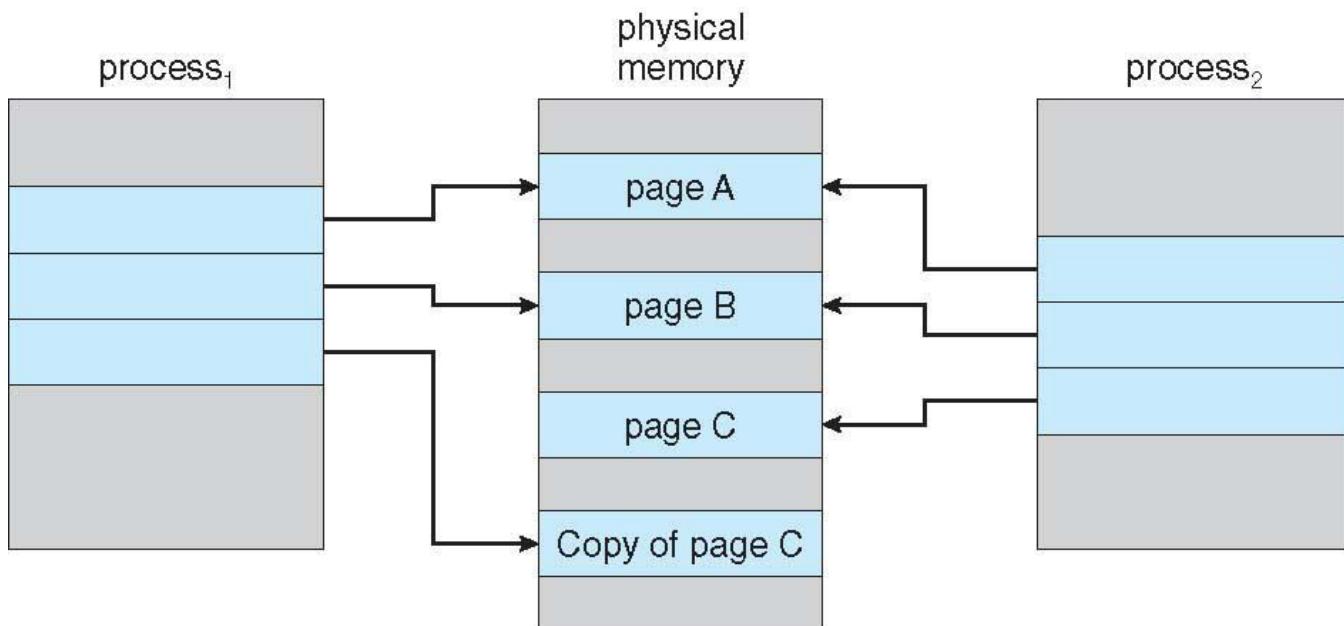
---

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
    - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# What Happens if There is no Free Frame?

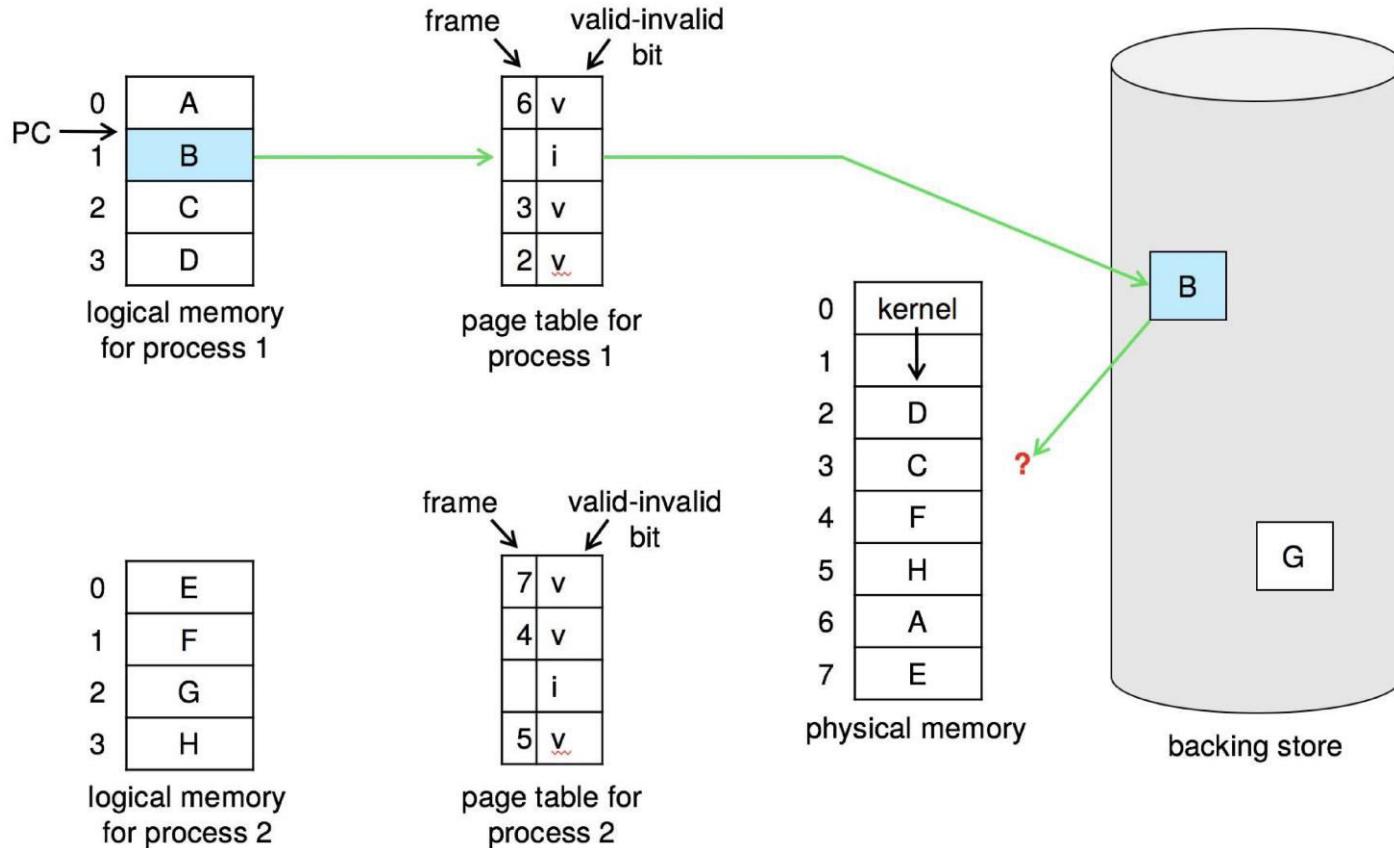
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result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

# Page Replacement

---

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

# Need For Page Replacement



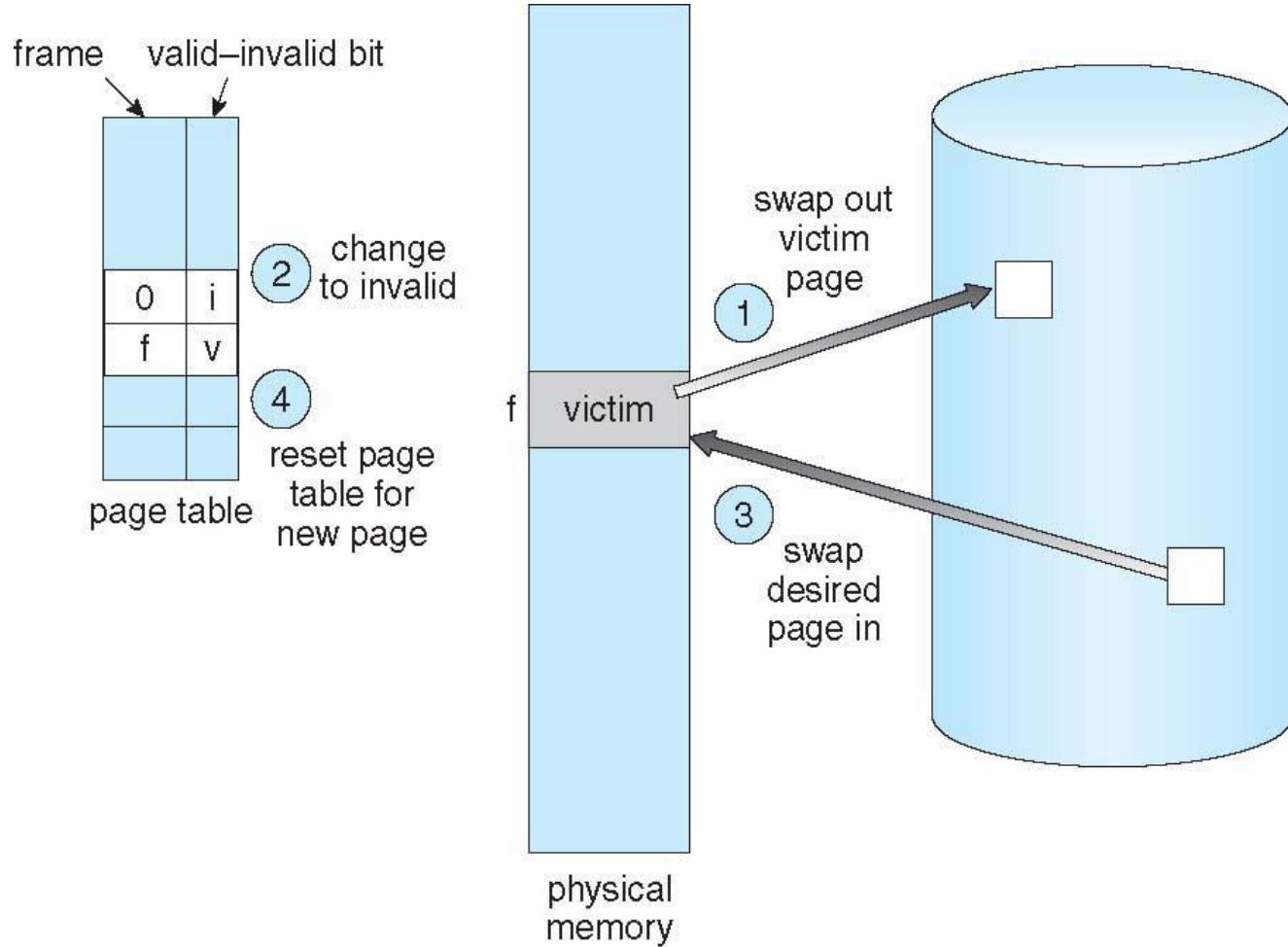
# Basic Page Replacement

---

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

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

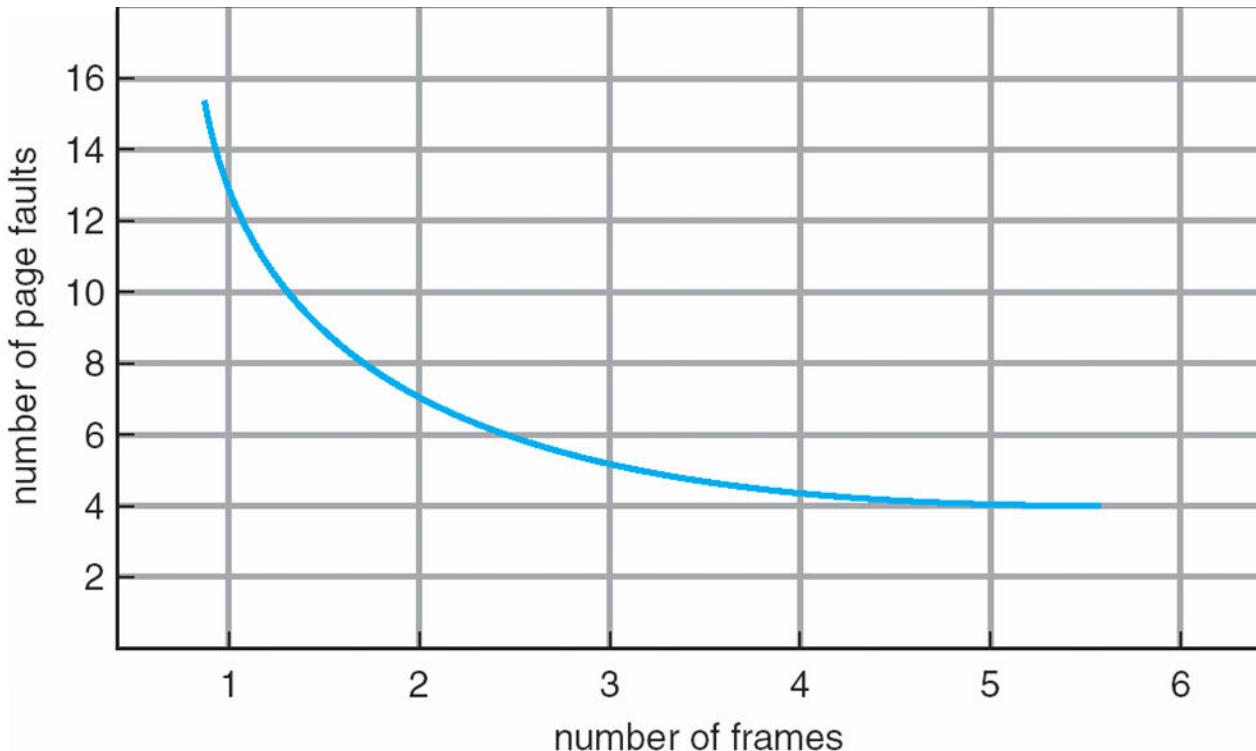
# Page Replacement



- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

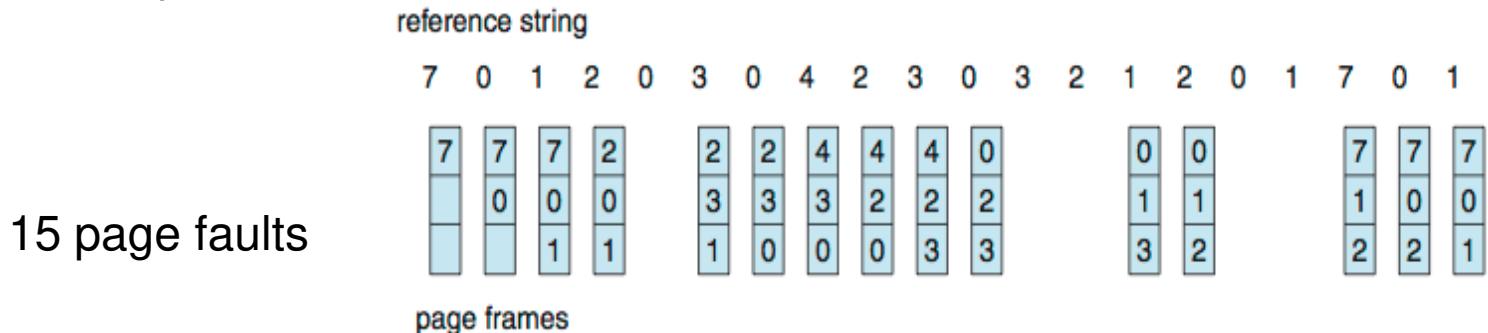
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

# Graph of Page Faults Versus the Number of Frames



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

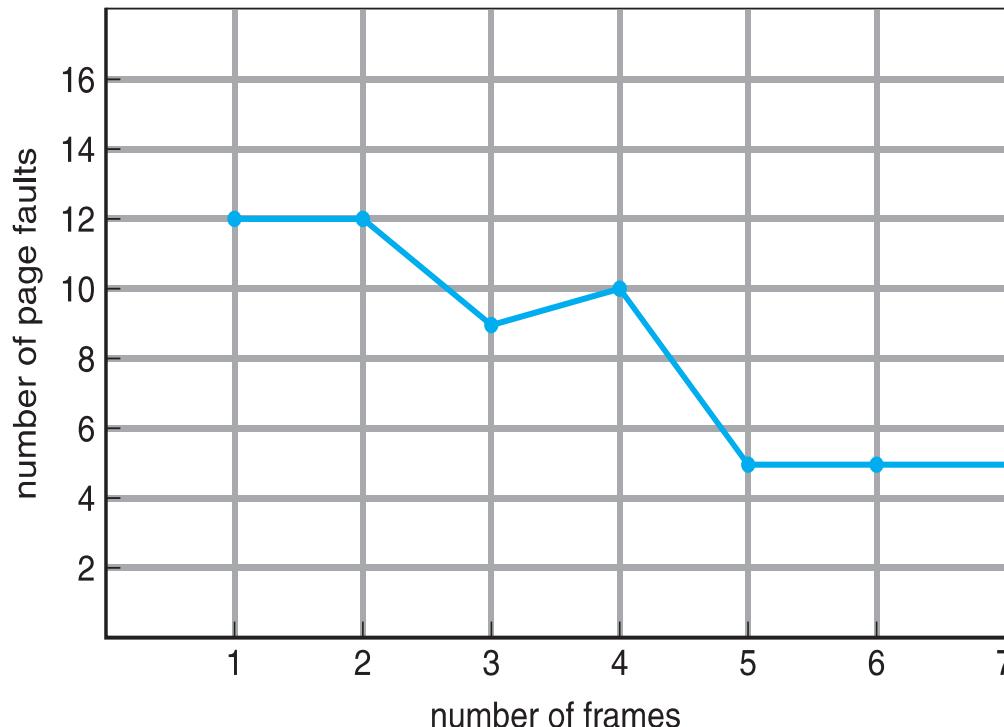
- Reference string:  
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- How to track ages of pages?
  - Just use a FIFO queue

# Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
- Graph illustrating Belady's Anomaly

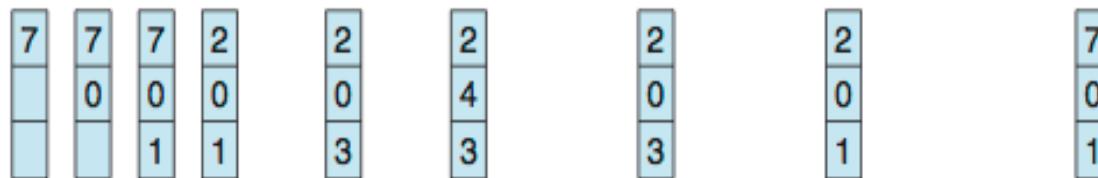


# Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs
- Optimal is an example of **stack algorithms** that don't suffer from Belady's Anomaly

reference string

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



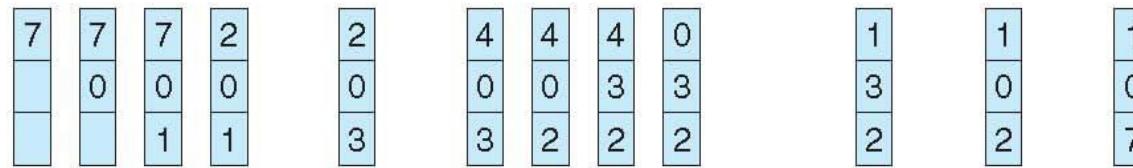
page frames

# Least Recently Used (LRU) Algorithm

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

reference string

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



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus it does not suffer from Belady's Anomaly

# LRU Algorithm Implementation

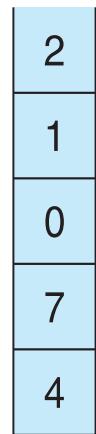
- Time-counter implementation
  - Every page entry has a time-counter variable; every time a page is referenced through this entry, copy the value of the clock into the time-counter
  - When a page needs to be changed, look at the time-counters to find smallest value
    - Search through a table is needed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

# Stack Implementation

- Use of a stack to record most recent page references

reference string

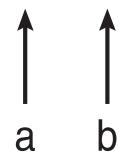
4 7 0 7 1 0 1 2 1 2 7 1 2



stack  
before  
a



stack  
after  
b



# Counting Algorithms

---

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used (LFU) Algorithm:**
  - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Applications and Page Replacement

---

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e., databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can provide direct access to the disk, getting out of the way of the applications
  - **Raw disk mode**
- Bypasses buffering, locking, etc.

# Allocation of Frames

---

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - Instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations

# Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i$  = size of process  $p_i$

$S = \sum s_i$

$m$  = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \quad 62 \quad 4$

$a_2 = \frac{127}{137} \quad 62 \quad 57$

# Global vs. Local Allocation

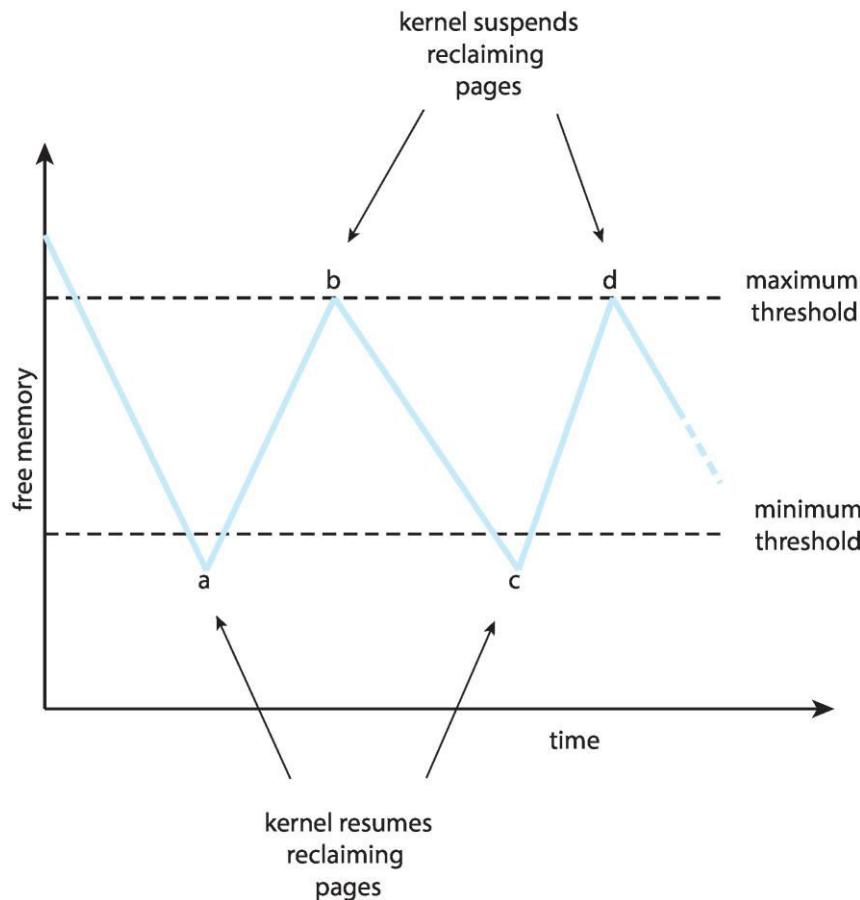
---

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly
  - Greater throughput so more commonly used
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory
  - What if a process does not have enough frames?

# Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

# Reclaiming Pages Example

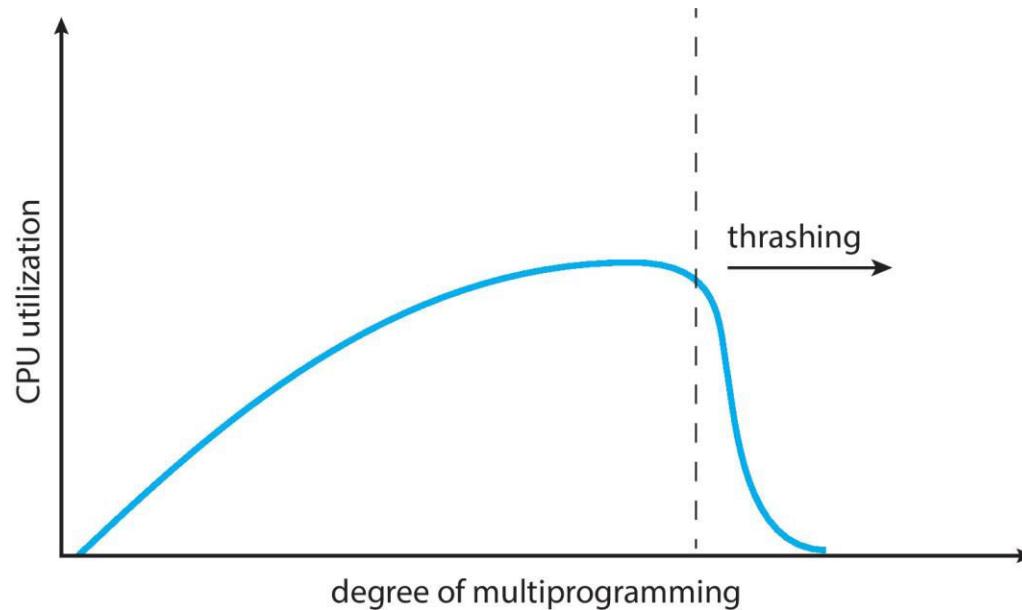


# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need the replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system

# Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out



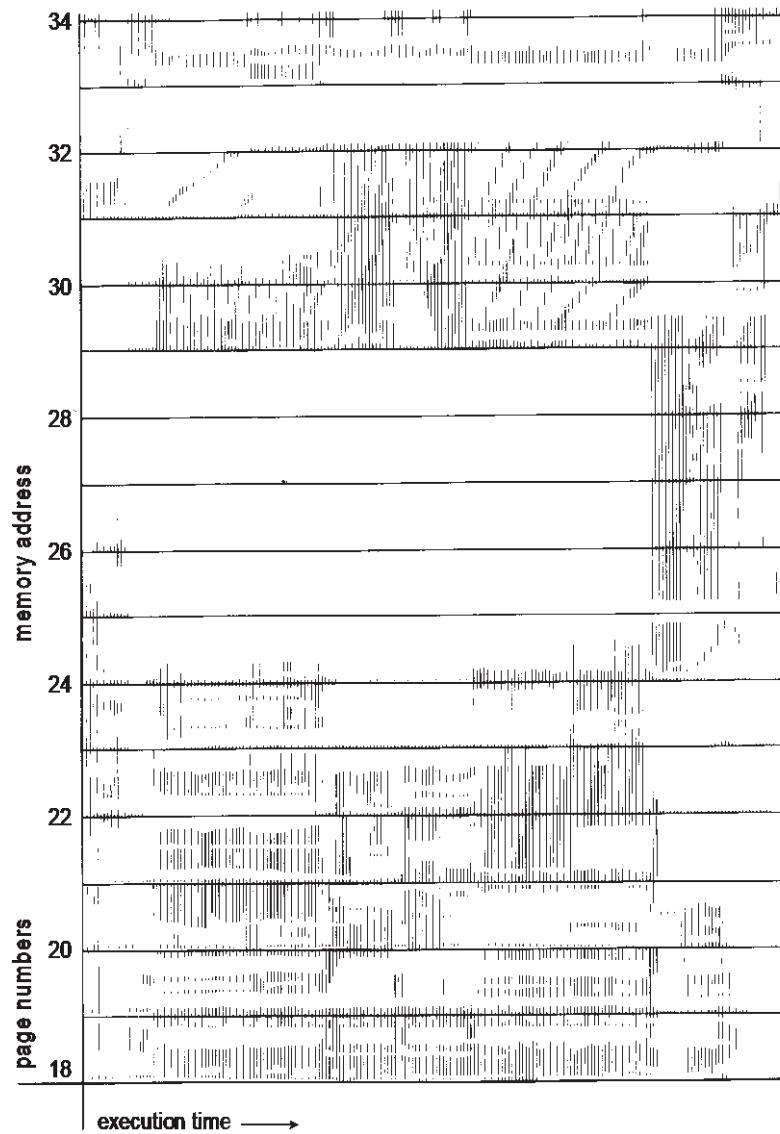
# Demand Paging and Thrashing

- Why does demand paging work?

## Locality model

- Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?  
 $\Sigma$  size of locality > total memory size
- To avoid thrashing:
  - Calculate the  $\Sigma$  size of locality
  - Policy:
    - if  $\Sigma$  size of locality > total memory size → suspend or swap out one of the processes
- Issue: how to calculate “ $\Sigma$  size of locality”

# Locality In A Memory-Reference Pattern

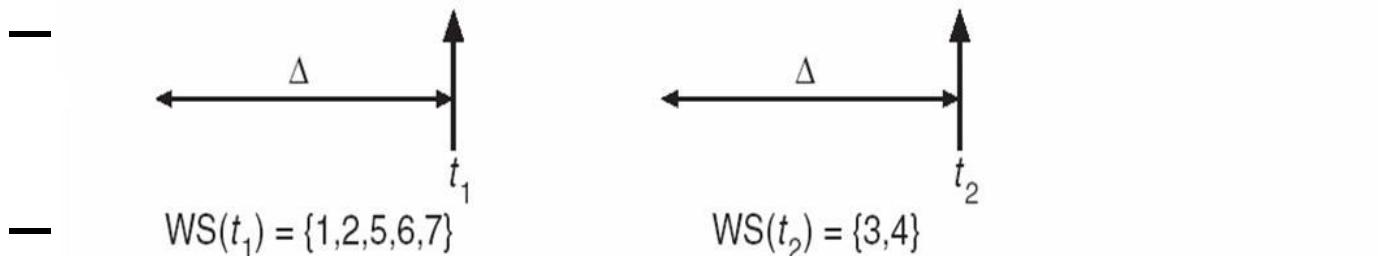


# Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
 Example: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass the

page reference table

... 2 6 1 5 7 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 ...



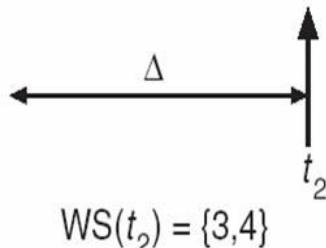
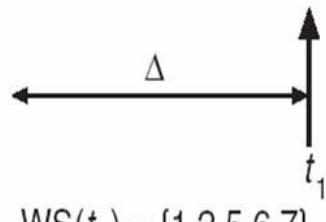
- program
- Example

# Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
Example: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass the entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality

page reference table

... 2 6 1 5 7 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 (Cont.)

---

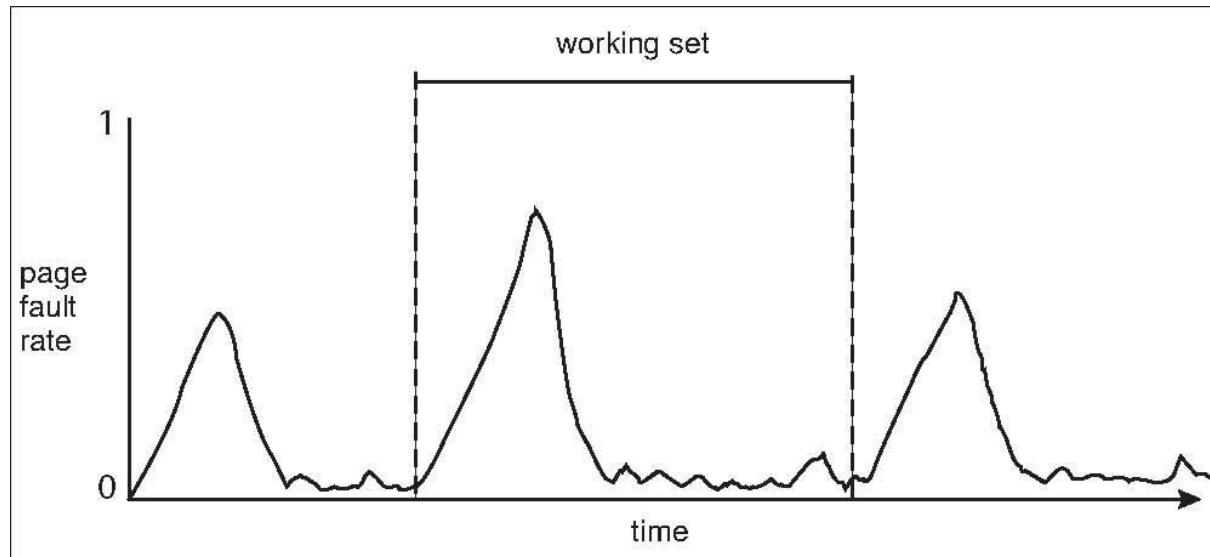
- $D = \sum WSS_i$  ≡ total demand frames
  - Approximation of locality
- $m$  = total number of frames
- If  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page  $i$  –  $B1_i$  and  $B2_i$
  - Whenever a timer interrupts copy the reference to one of the  $B_j$  and sets the values of all reference bits to 0
  - If either  $B1_i$  or  $B2_i = 1$ , it implies that Page  $i$  is in the working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

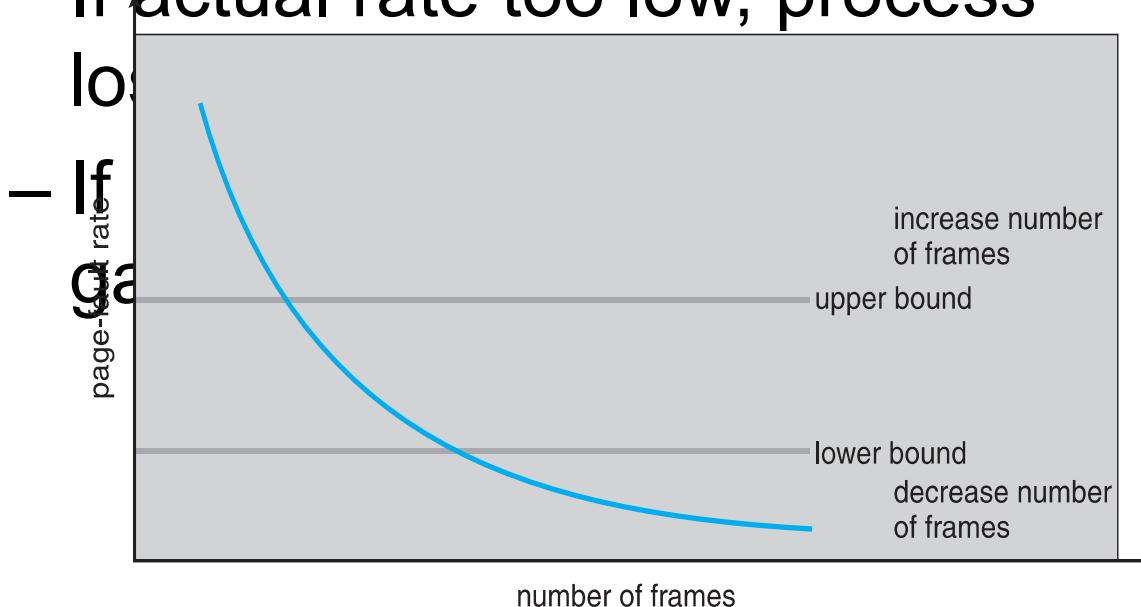
# Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



# Page-Fault Frequency Algorithm

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loss
  - If actual rate too high, decrease number of frames



# Other Considerations

---

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

# Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume  $s$  pages are prepaged and  $\alpha$  of the pages is used
  - Question: is the cost of  $s * \alpha$  save pages faults is greater or less than the cost of prepaging  $s * (1 - \alpha)$  unnecessary pages?
  - If  $\alpha$  is close to 0  $\Rightarrow$  prepaging loses
  - If  $\alpha$  is close to 1  $\Rightarrow$  prepaging wins

# Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range  $2^{12}$  (4,096 bytes) to  $2^{22}$  (4,194,304 bytes)
- On average, growing over time

# Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## CSE-202 OPERATING SYSTEM

### Module IV: I/O Systems

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling , Pearson Education**

# Course Learning Objectives:

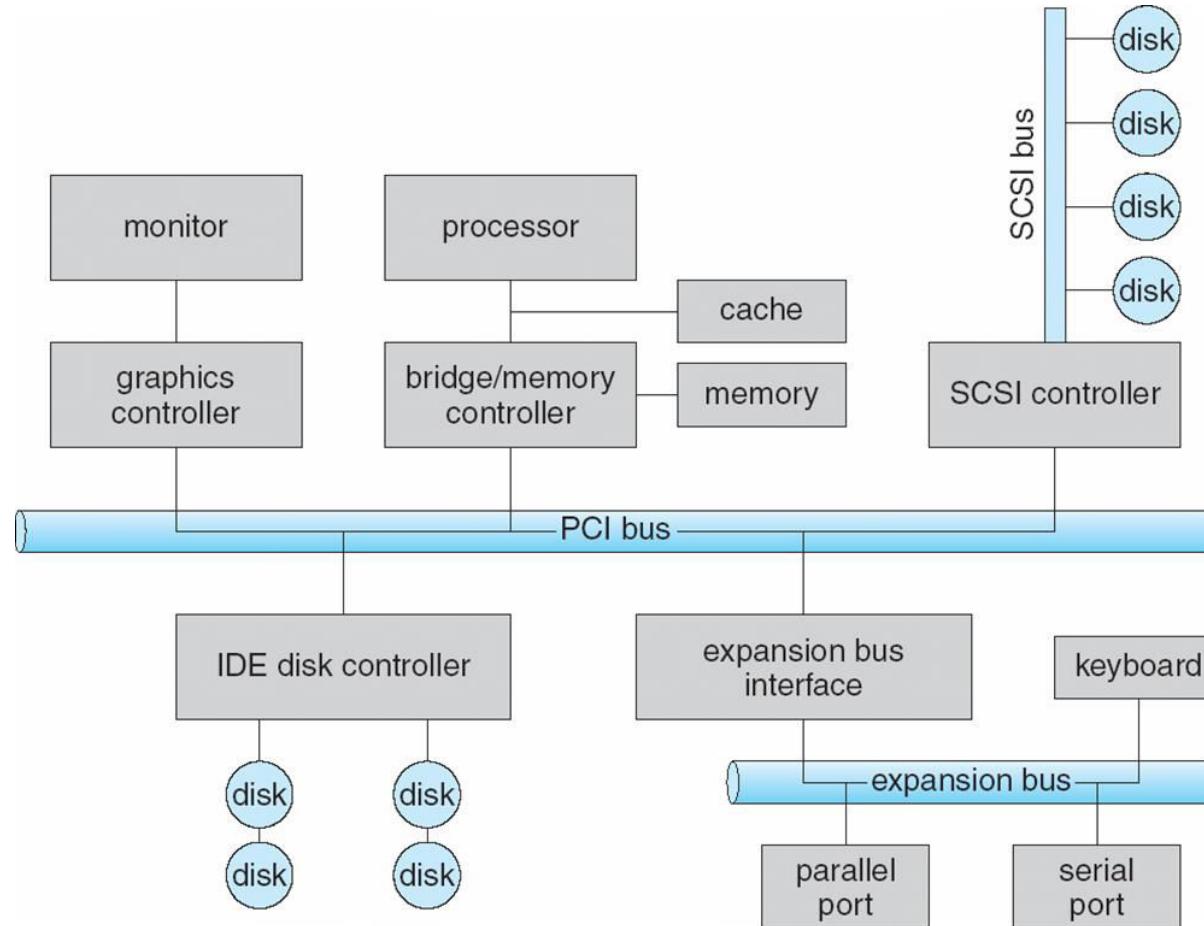
- To understand the structure and complexities of an operating system's I/O subsystem
- To understand disk scheduling algorithm
- To understand how I/O devices performance can be enhanced

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus - daisy chain** or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**PCle**)
    - **expansion bus** connects relatively slow devices
  - **Controller (host adapter)** – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc
      - Some talk to per-device controller with bus controller, microcode, memory, etc

# A Typical PC Bus Structure



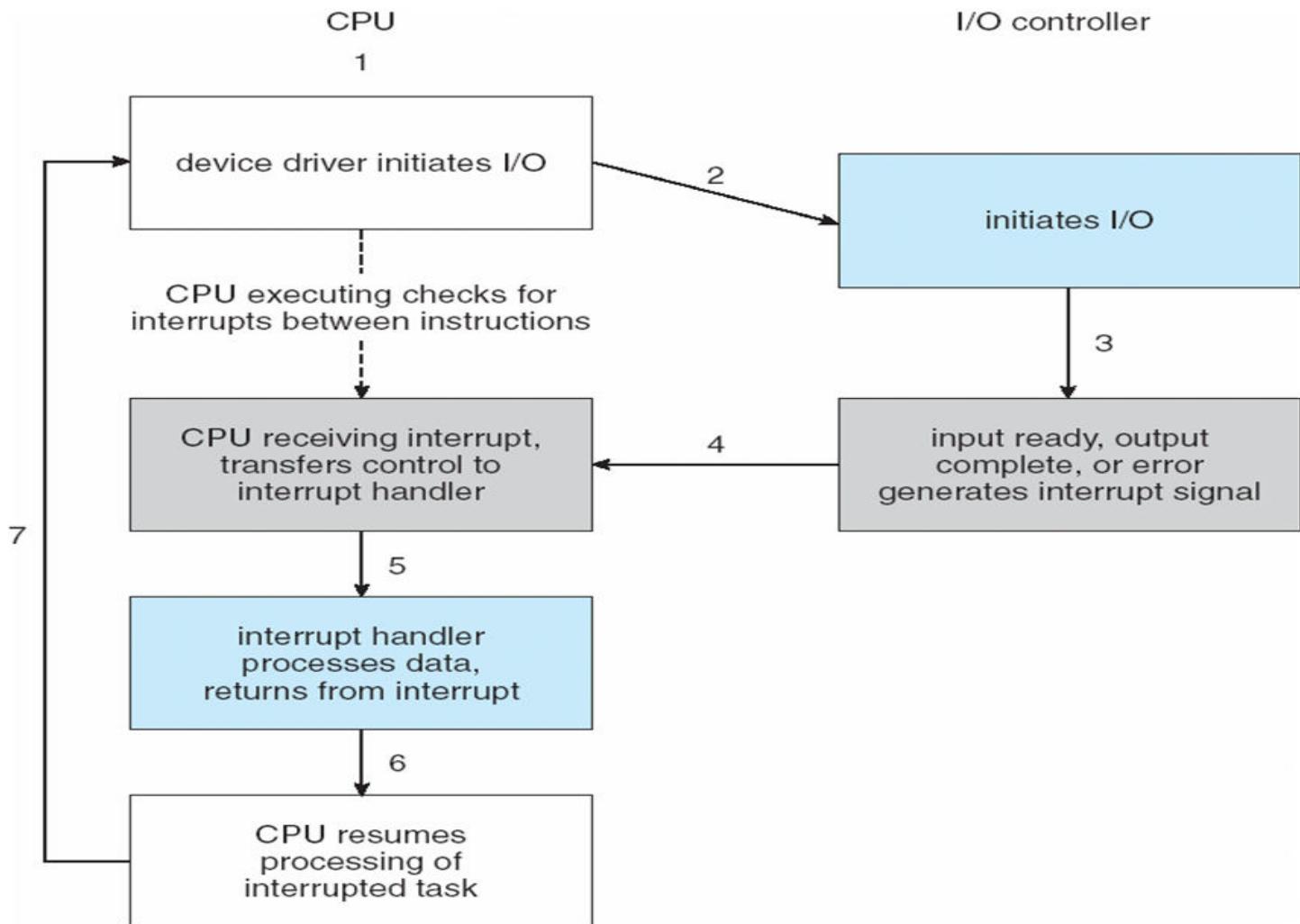
- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

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

- For each byte of I/O
  1. Read busy bit from status register until 0
  2. Host sets read or write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit, executes transfer
  5. Controller clears busy bit, error bit, command-ready bit when transfer done
- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - ▶ But if miss a cycle data overwritten / lost

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number

# Interrupt-Driven I/O Cycle

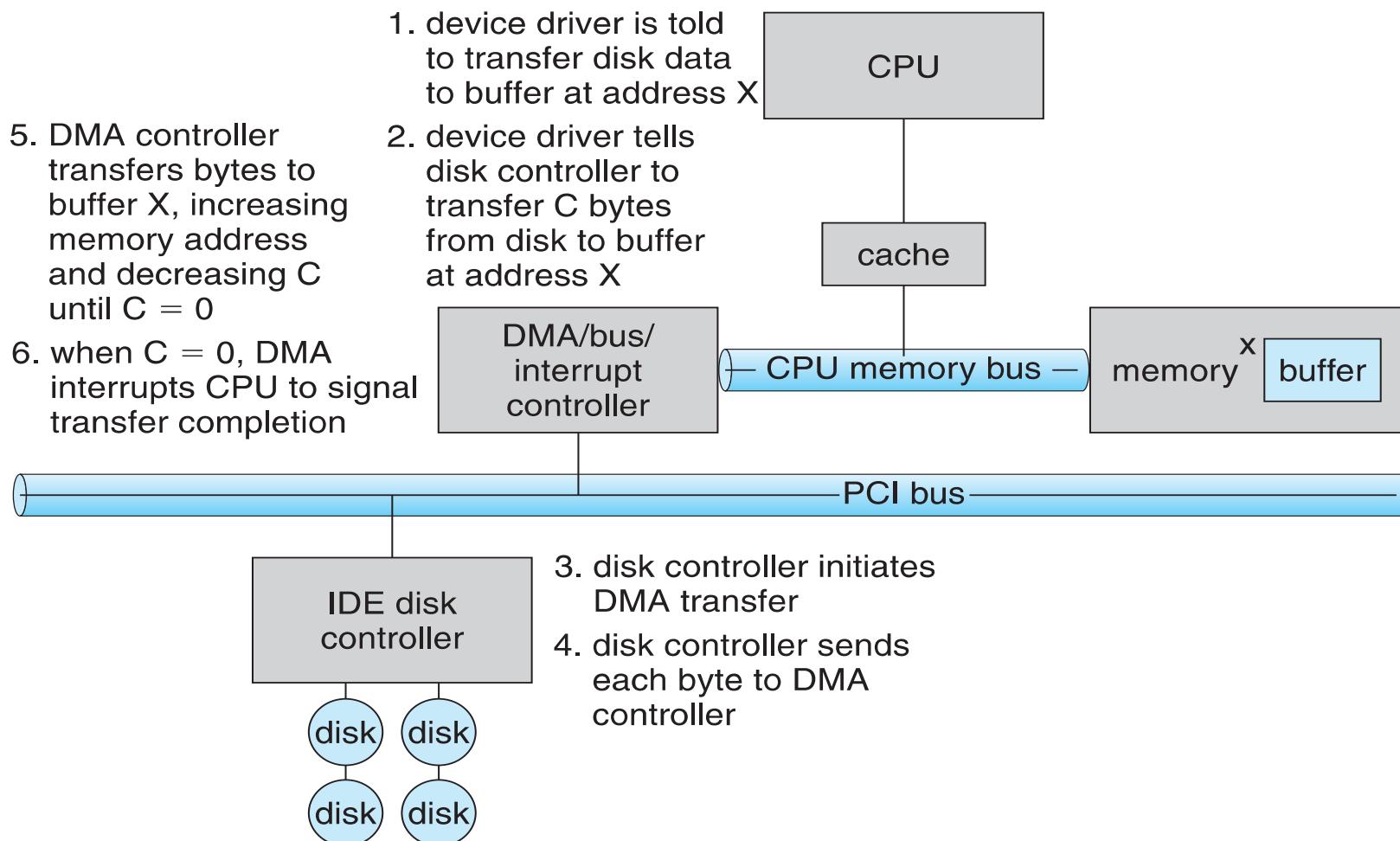


| vector number | description                            |
|---------------|----------------------------------------|
| 0             | divide error                           |
| 1             | debug exception                        |
| 2             | null interrupt                         |
| 3  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

- Interrupt mechanism also used for **exceptions**
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
    - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**

# Six Step Process to Perform DMA Transfer



- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**

| aspect             | variation                                                         | example                               |
|--------------------|-------------------------------------------------------------------|---------------------------------------|
| data-transfer mode | character<br>block                                                | terminal<br>disk                      |
| access method      | sequential<br>random                                              | modem<br>CD-ROM                       |
| transfer schedule  | synchronous<br>asynchronous                                       | tape<br>keyboard                      |
| sharing            | dedicated<br>sharable                                             | tape<br>keyboard                      |
| device speed       | latency<br>seek time<br>transfer rate<br>delay between operations |                                       |
| I/O direction      | read only<br>write only<br>read-write                             | CD-ROM<br>graphics controller<br>disk |

## (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register

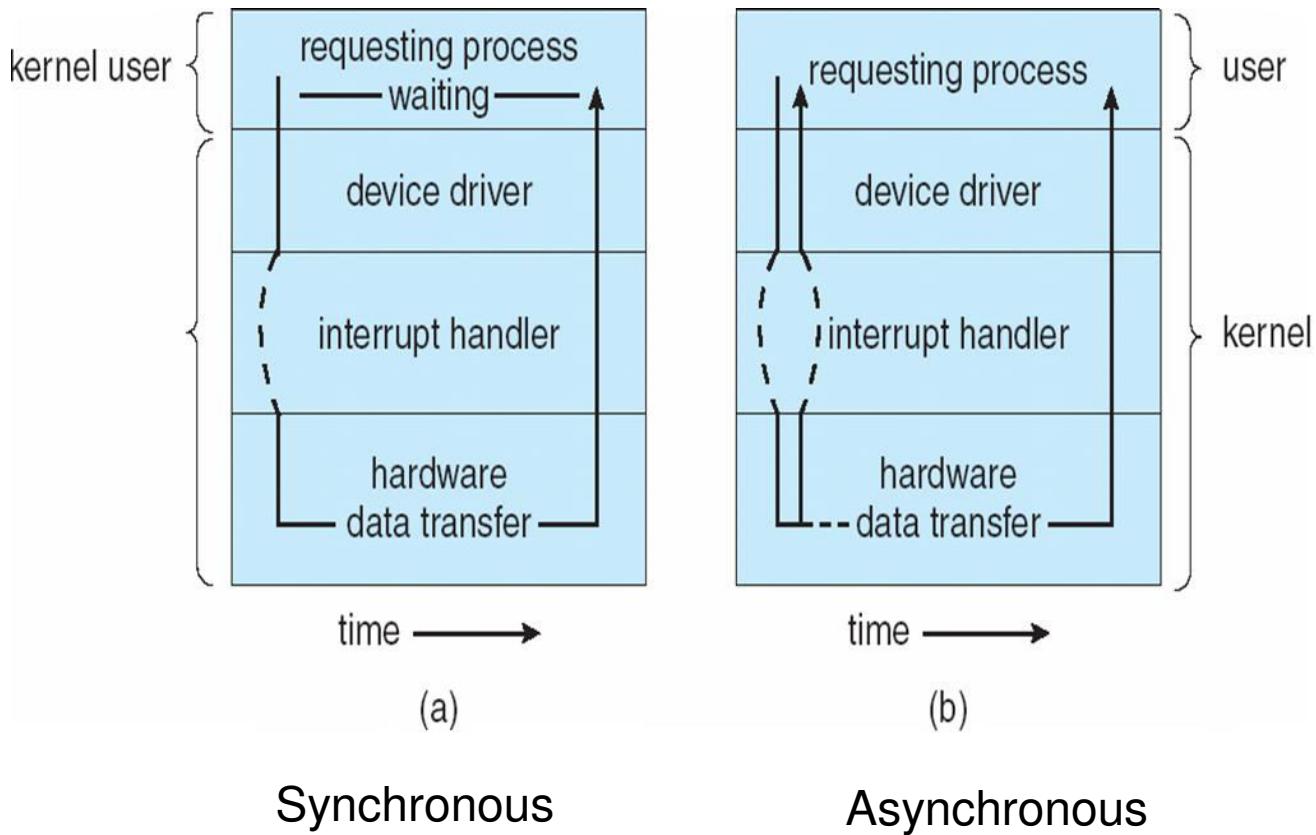
- Block devices include disk drives
  - Commands include read, write, seek
  - **Raw I/O, direct I/O**, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA
- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include **socket** interface
  - Separates network protocol from network operation
  - Includes **select()** functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

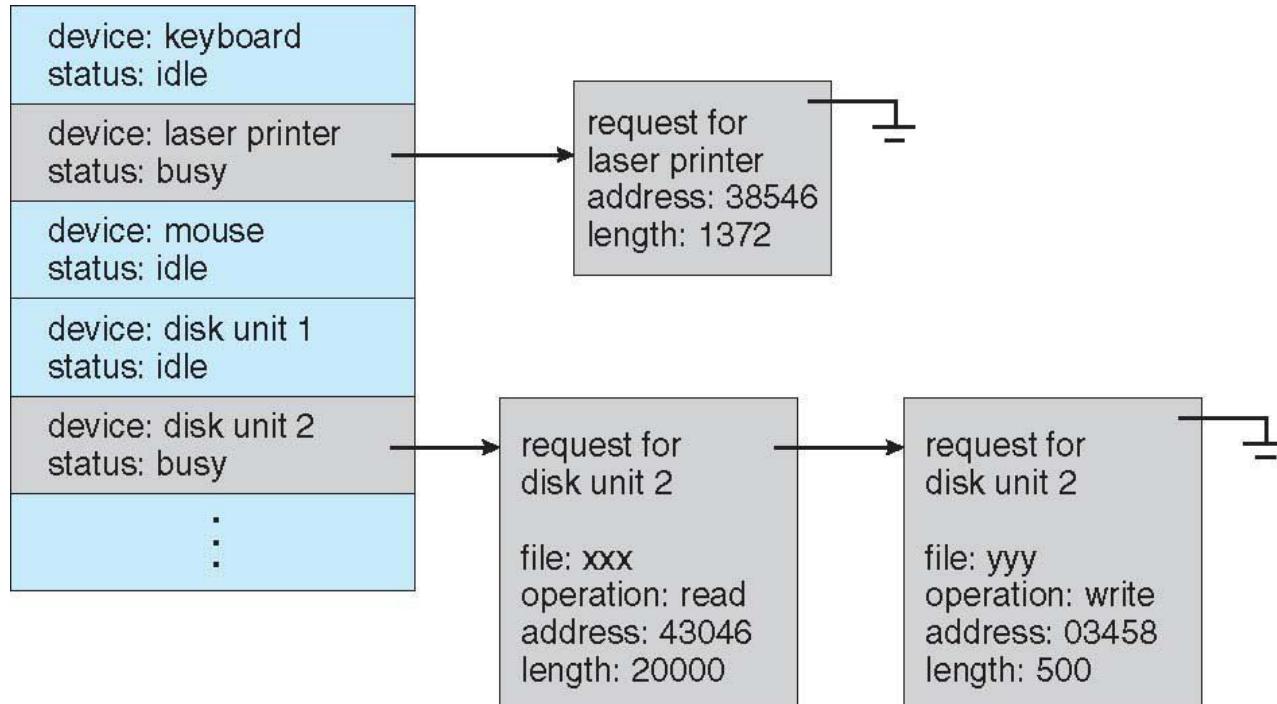
# Two I/O Methods



- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix `readve()` accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
  - Decreases context switching and system call overhead
  - Some versions provide atomicity
    - Avoid for example worry about multiple threads changing data as reads / writes occurring

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - **Double buffering** – two copies of the data
    - Kernel and user
    - Varying sizes
    - Full / being processed and not-full / being used
    - Copy-on-write can be used for efficiency in some cases

# Device-status Table



- **Caching** - faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

# Thank You

**CSE-202 OPERATING SYSTEM**

**Module IV: Disk Scheduling**

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling, Pearson Education**

## **Course Learning Objectives:**

- To understand how I/O devices performance can be enhanced

# Topics Covered

---

- Disk Scheduling Algorithms
  - FCFS Scheduling
  - Shortest Seek Time First (SSTF)
  - SCAN
  - SEEK
  - C-SCAN
  - C-SEEK
  - LOOK
  - C-LOOK

# HDD Scheduling

---

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time  $\approx$  seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

# Disk Scheduling (Cont.)

---

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists

# Disk Scheduling (Cont.)

---

- In the past, operating system responsible for queue management, disk drive head scheduling
  - Now, built into the storage devices, controllers
  - Just provide Logical Block Addressing (LBA), handle sorting of requests
    - Some of the algorithms they use described next

# Disk Scheduling (Cont.)

---

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms 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

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

head starts at 53



Total head movements incurred while servicing these requests

$$\begin{aligned}
 &= (98 - 53) + (183 - 98) + (183 - 37) + (122 - 37) + (122 - 14) + (124 - 14) + \\
 &\quad (124 - 65) + (67 - 65) \\
 &= 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = 640
 \end{aligned}$$

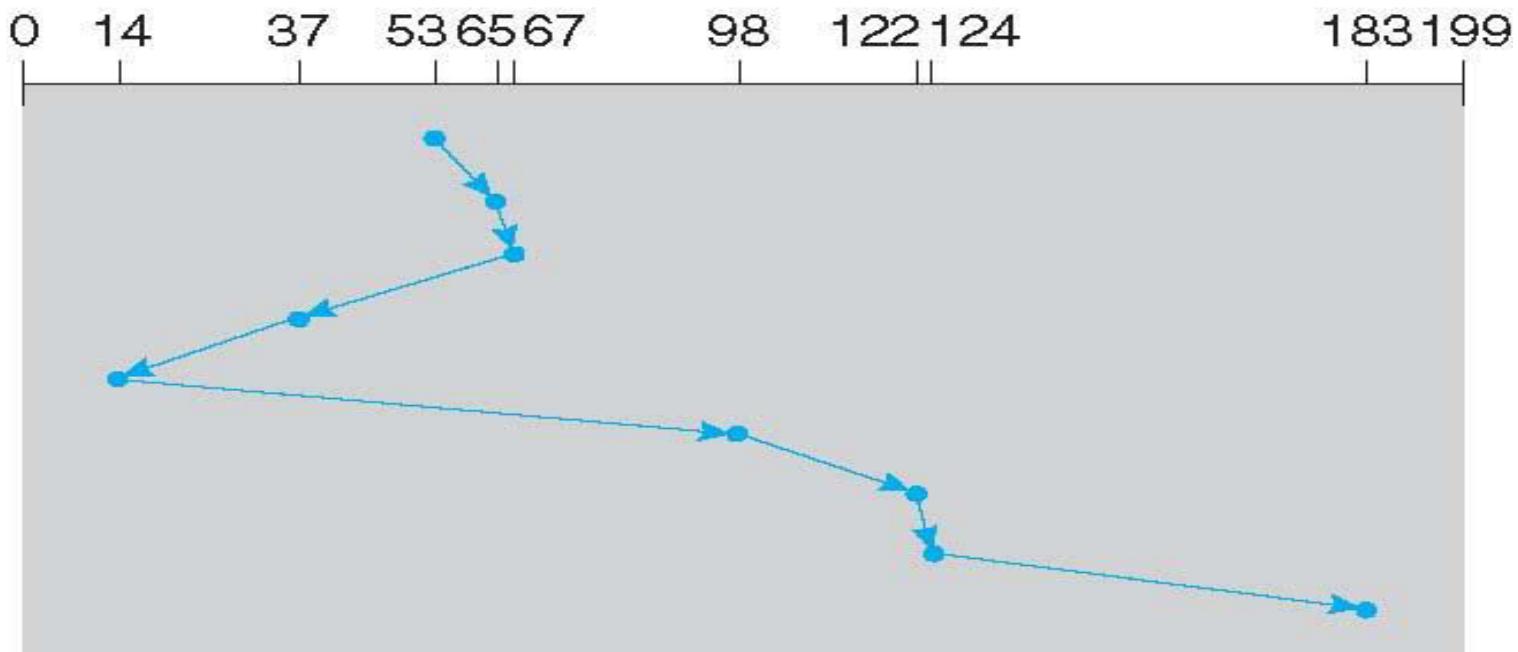
# SSTF

- 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



$$\begin{aligned} &= (65 - 53) + (67 - 65) + (67 - 37) + (37 - 14) + (98 - 14) + (122 - 98) \\ &\quad + (124 - 122) + (183 - 124) \end{aligned}$$

$$= 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59$$

$$= 236$$

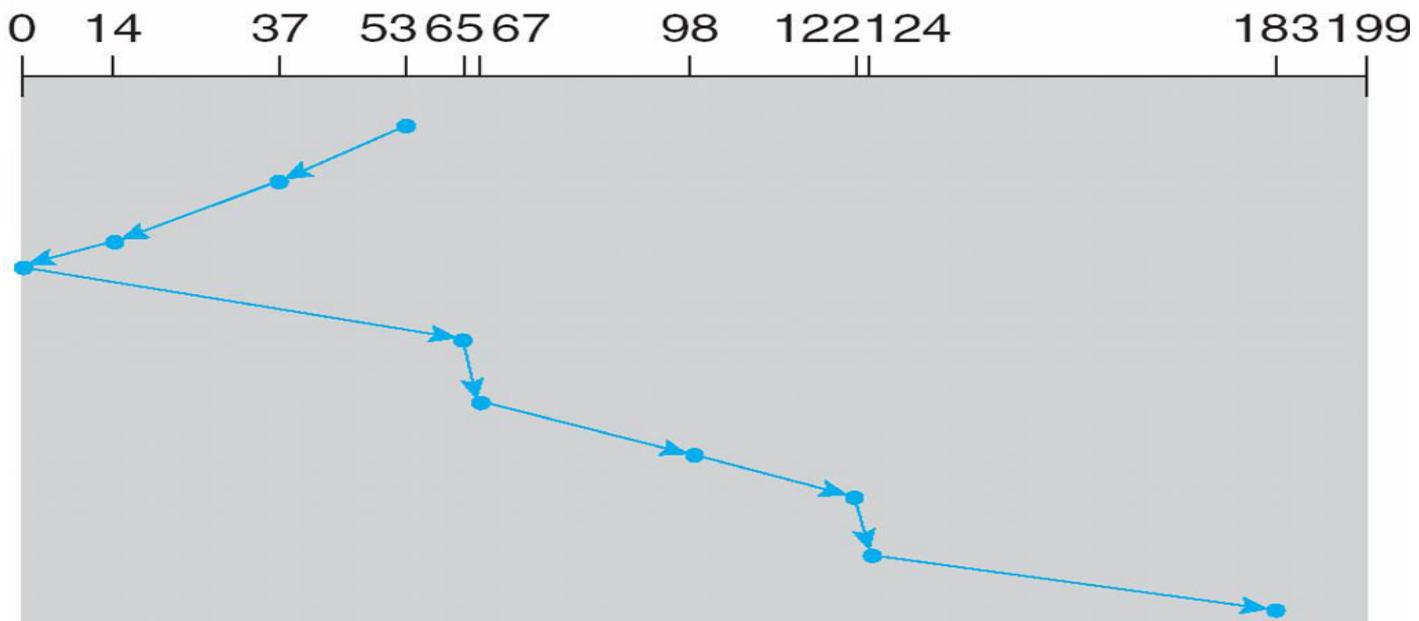
# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed, and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders

# SCAN (Cont.)

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

head starts at 53



$$\begin{aligned}
&= (53 - 37) + (37 - 14) + (14 - 0) + (65 - 0) + (67 \\
&- 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - \\
&124)
\end{aligned}$$

$$= 16 + 23 + 14 + 65 + 2 + 31 + 24 + 2 + 59$$

$$= 236$$

Alternatively  
Total head movements incurred  
while servicing these requests

$$\begin{aligned} &= (53-0) + (183 - 0) \\ &= 53 + 183 \\ &= 236 \end{aligned}$$

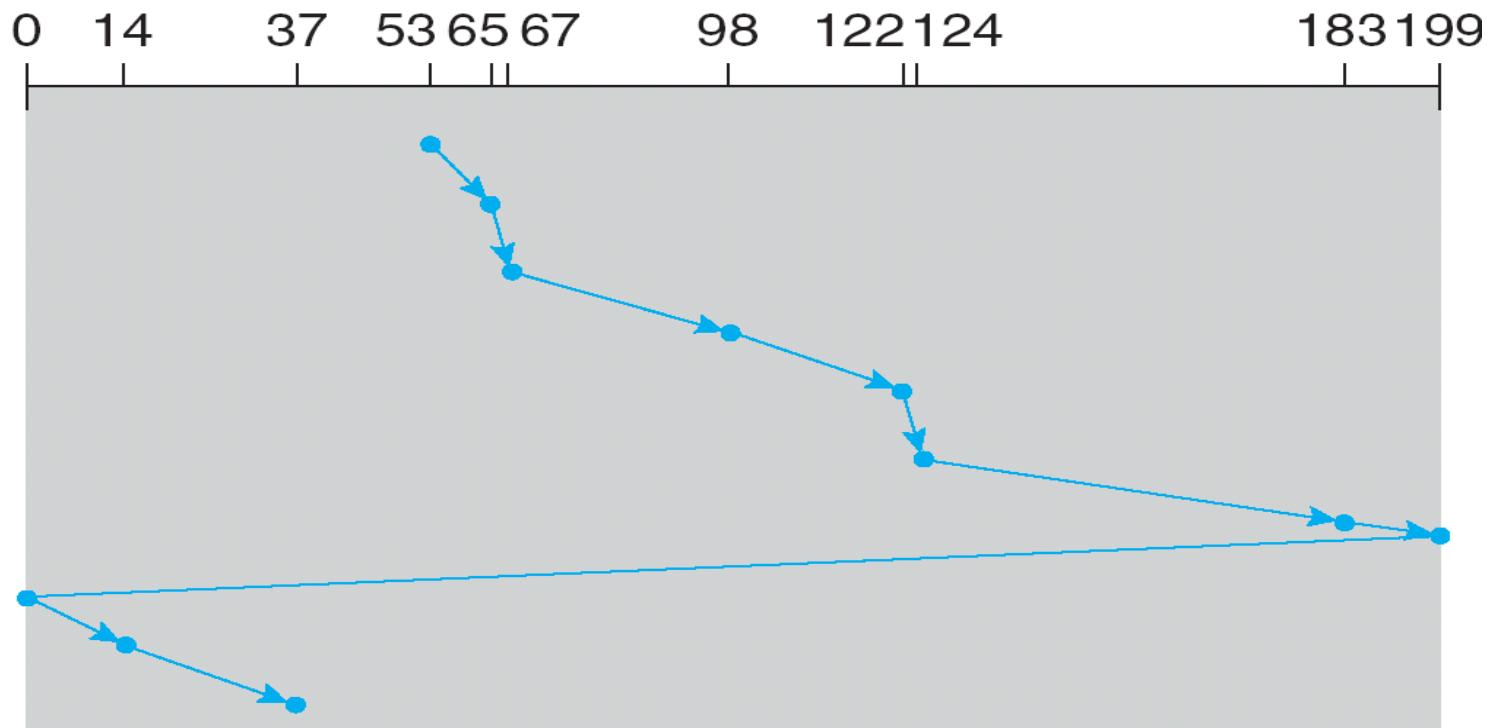
# C-SCAN

- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- 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



Total head movements incurred while servicing these requests

$$= (199 - 53) + (199 - 0) + (37 - 0)$$

$$= 146 + 199 + 37$$

$$= 382$$

# LOOK and C-LOOK

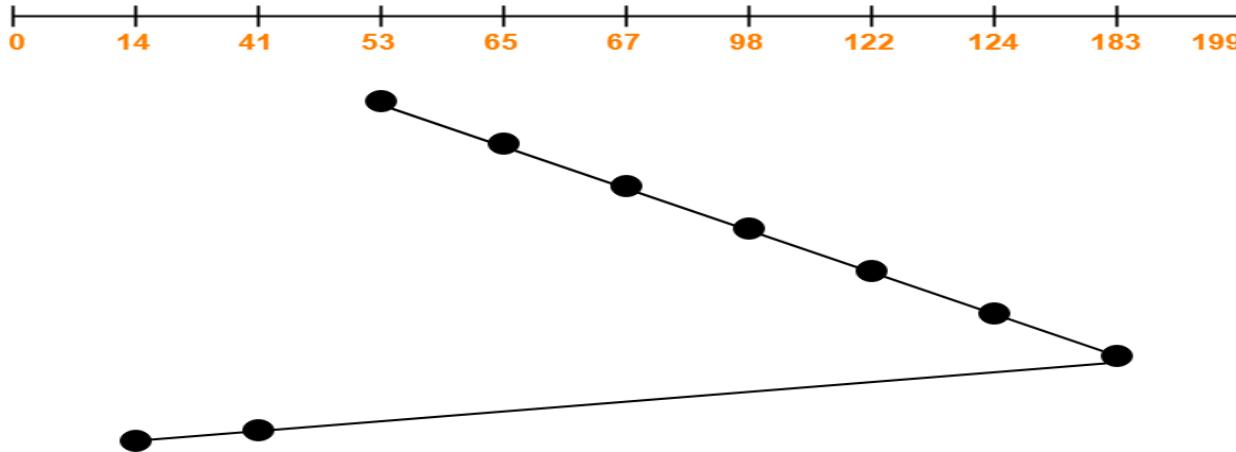
---

- Version of SCAN and 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

Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67.

The LOOK scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. T

he cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is \_\_\_\_\_.

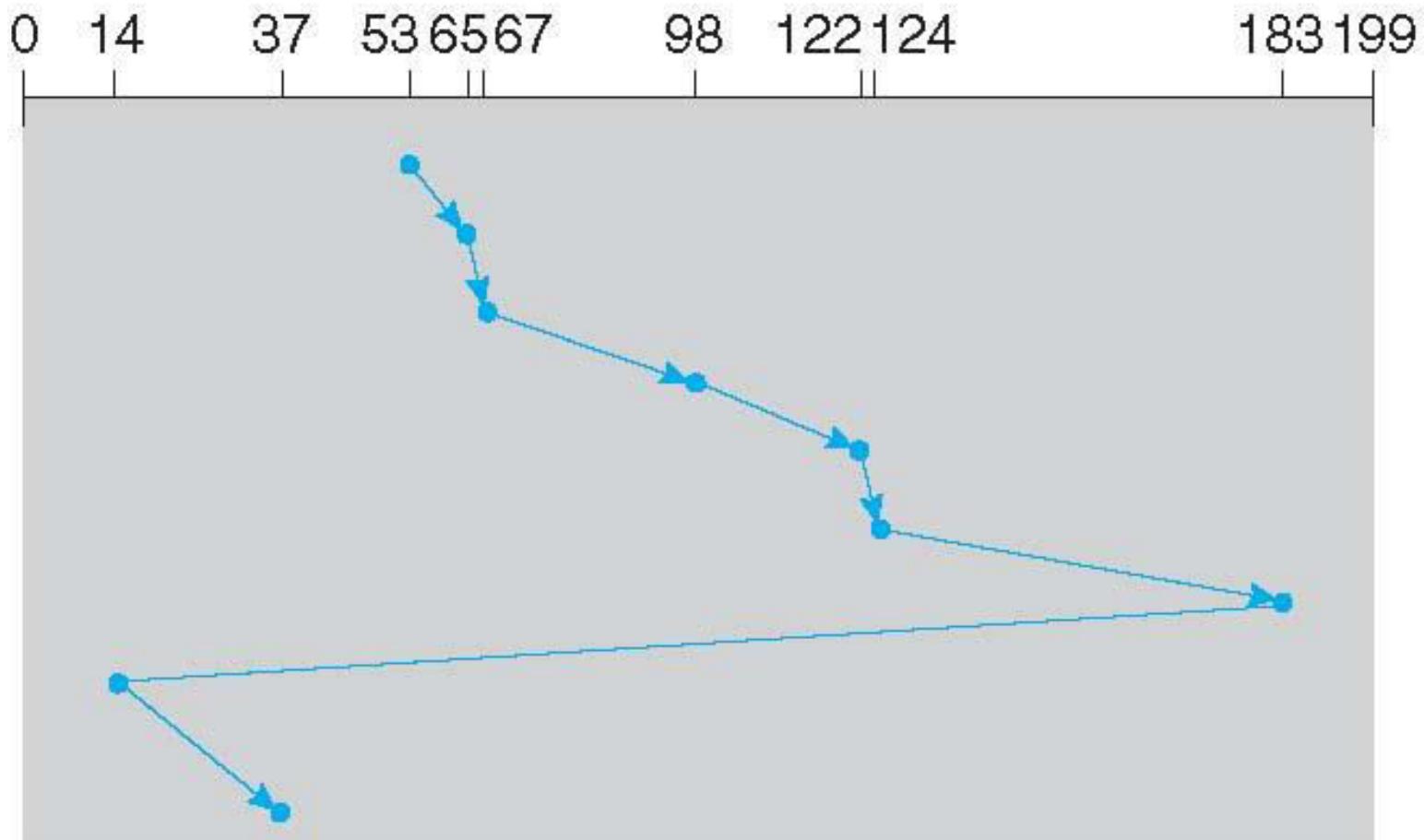


- Total head movements incurred while servicing these requests
- $= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - 124) + (183 - 41) + (41 - 14)$
- $= 12 + 2 + 31 + 24 + 2 + 59 + 142 + 27$
- $= 299$

# C-LOOK (Cont.)

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

head starts at 53



# 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 LOOK is a reasonable choice for the default algorithm

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## CSE-202 OPERATING SYSTEM

### Module V: File Concept

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling, Pearson Education**

## **Course Learning Objectives:**

- To understand basic file concept
- To understand file access mechanism
- To understand the directory Structure
- To understand how files can be shared

# Topics to be covered

---

- File Concept,
- File Organization and Access Mechanism,
- File Directories,
- Basic file system,
- File Sharing,
- Allocation method,
- Free space management.
- Policy Mechanism,
- Authentication,
- Internal excess Authorization.

# File Concept

---

- Contiguous logical address space
- Types:
  - Data
    - Numeric
    - Character
    - Binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider **text file, source file, executable file**

# File Attributes

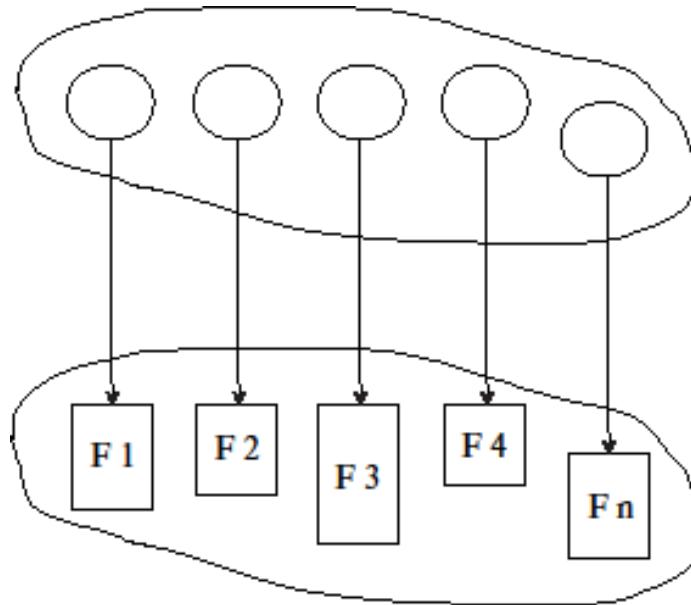
- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls 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
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

# File info Window on Mac OS X



# Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

# File Operations

- **Create**
- **Write** – at **write pointer** location
- **Read** – at **read pointer** location
- **Reposition within 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

# Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

# Open File Locking

---

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

# File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
 public static final boolean EXCLUSIVE = false;
 public static final boolean SHARED = true;
 public static void main(String arsg[]) throws IOException {
 FileLock sharedLock = null;
 FileLock exclusiveLock = null;
 try {
 RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
 // get the channel for the file
 FileChannel ch = raf.getChannel();
 // this locks the first half of the file - exclusive
 exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
 /** Now modify the data . . . */
 // release the lock
 exclusiveLock.release();
 }
 }
}
```

# File Locking Example – Java API (Cont.)

```
// this locks the second half of the file - shared
sharedLock = ch.lock(raf.length()/2+1, raf.length(),
 SHARED);
/** Now read the data . . . */
// release the lock
sharedLock.release();
} catch (java.io.IOException ioe) {
 System.err.println(ioe);
}finally {
 if (exclusiveLock != null)
 exclusiveLock.release();
 if (sharedLock != null)
 sharedLock.release();
}
}
```

# File Types – Name, Extension

| file type      | usual extension          | function                                                                            |
|----------------|--------------------------|-------------------------------------------------------------------------------------|
| executable     | exe, com, bin or none    | ready-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, rtf, doc        | various word-processor formats                                                      |
| library        | lib, a, so, dll          | libraries of routines for programmers                                               |
| print or view  | ps, pdf, jpg             | 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, mp3, avi  | binary file containing audio or A/V information                                     |

# File Structure

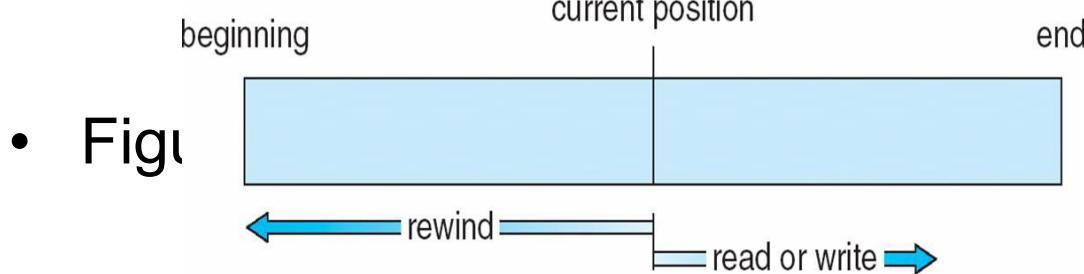
- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
 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

# Access Methods

- **Sequential Access**
- **Direct Access**

# Sequential Access

- **Operations**
  - **read next**
  - **write next**
  - **Reset**
  - **no read after last write (rewrite)**



# Direct Access

- A file is fixed length **logical records**
- Operations
  - **read *n***
  - **write *n***
  - **position to *n***
    - **read next**
    - **write next**
    - **rewrite *n***
- ***n* = relative block number**
- Relative block numbers allow OS to decide where file should be placed

## Simulation of Sequential Access on Direct-access File

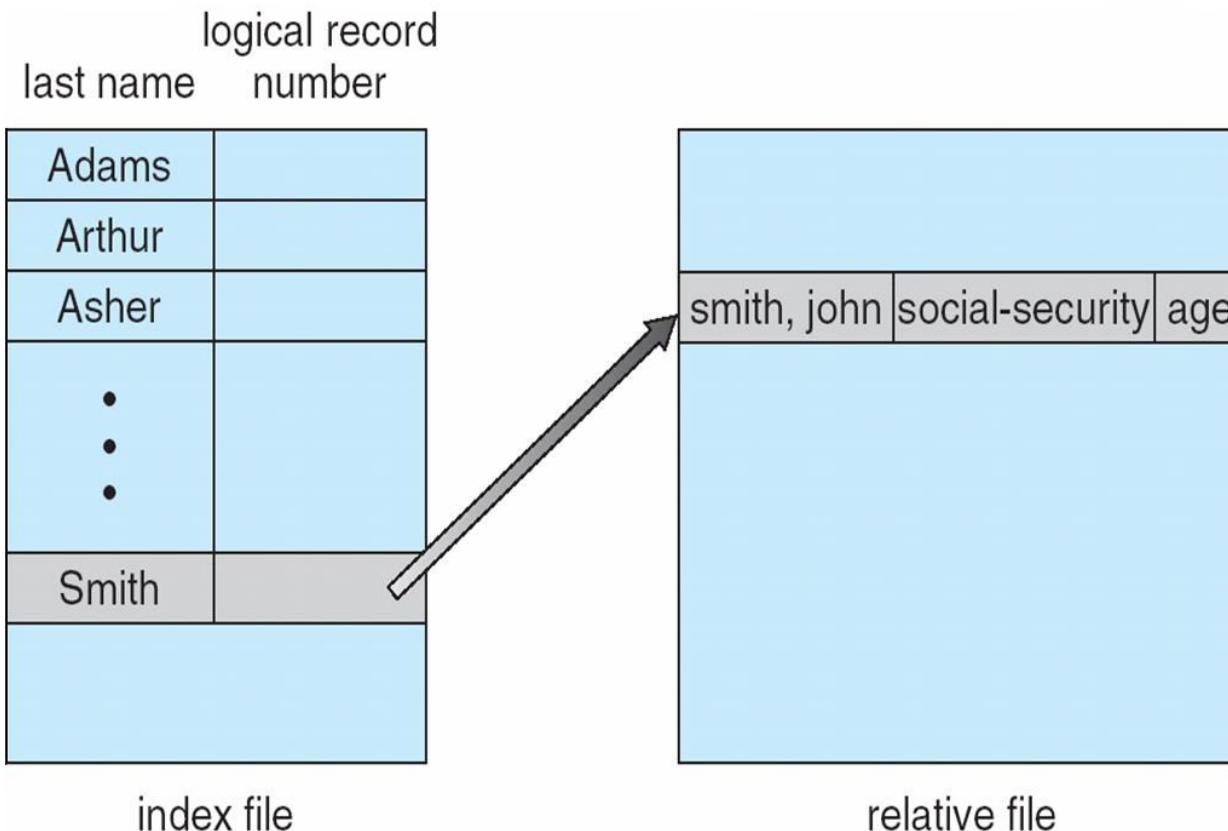
| sequential access | implementation for direct access   |
|-------------------|------------------------------------|
| <i>reset</i>      | $cp = 0;$                          |
| <i>read next</i>  | <i>read cp;</i><br>$cp = cp + 1;$  |
| <i>write next</i> | <i>write cp;</i><br>$cp = cp + 1;$ |

# Other Access Methods

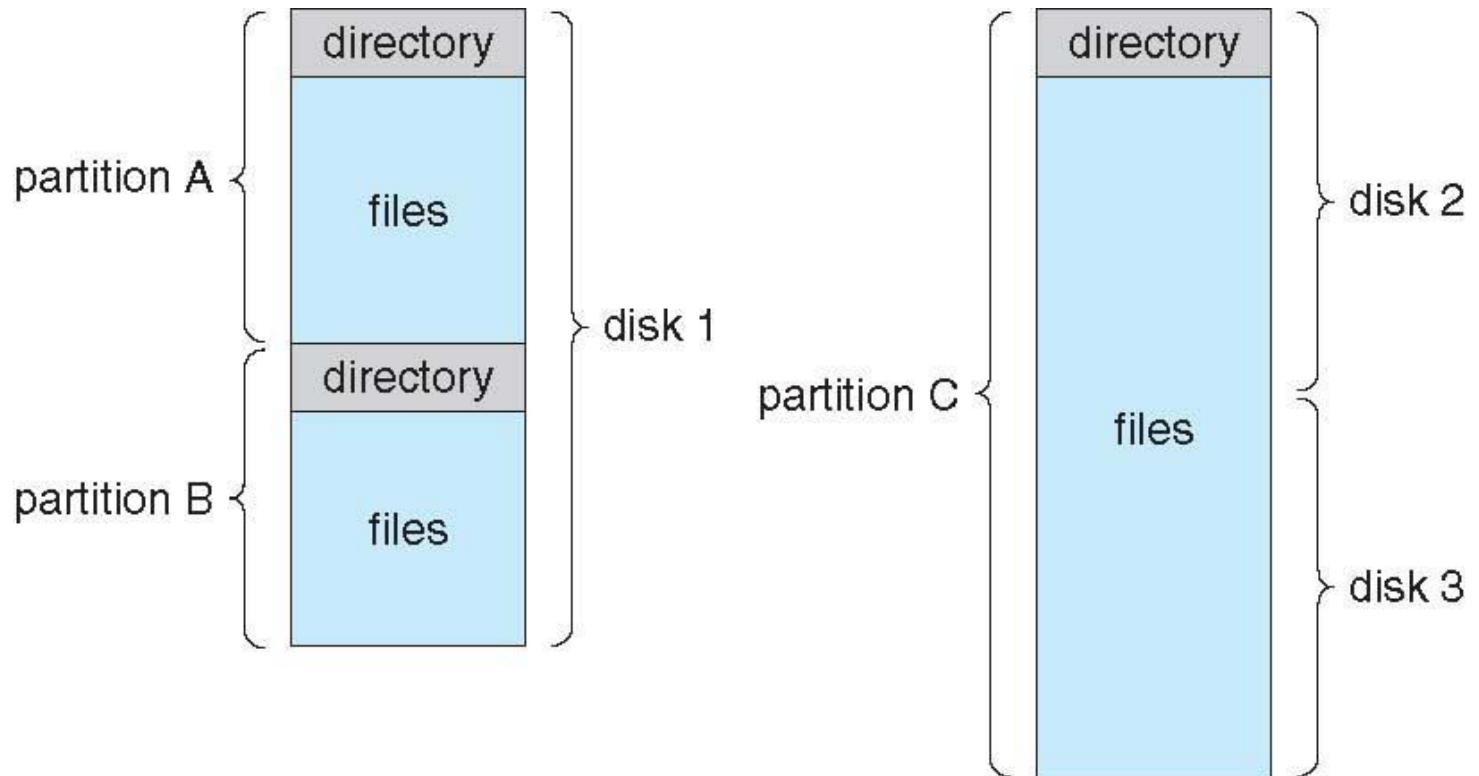
---

- Can be built on top of base methods
- General involve creation of an **index** for the file
- Keep index in memory for fast determination of location of data to be operated on (consider Universal Produce Code (UPC code) plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

# Example of Index and Relative Files



# A Typical File-system Organization



# Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

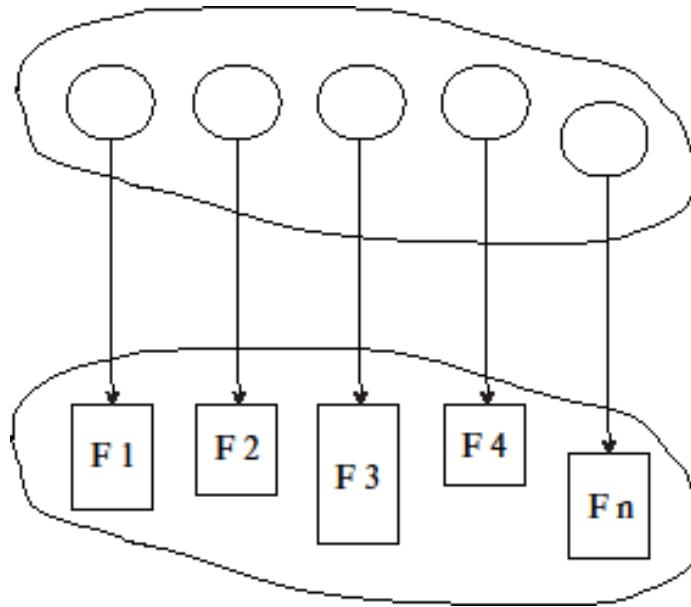
# Types of File Systems

---

- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems

# Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

# Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

# Directory Organization

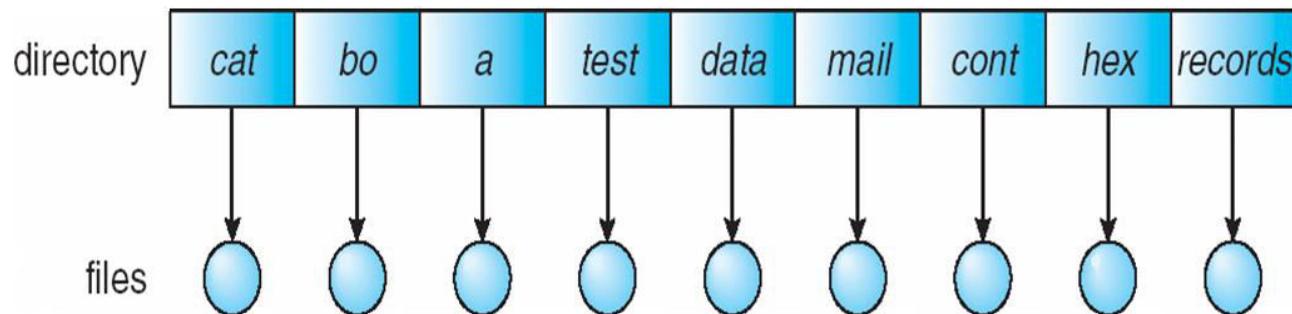
---

The directory is organized 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

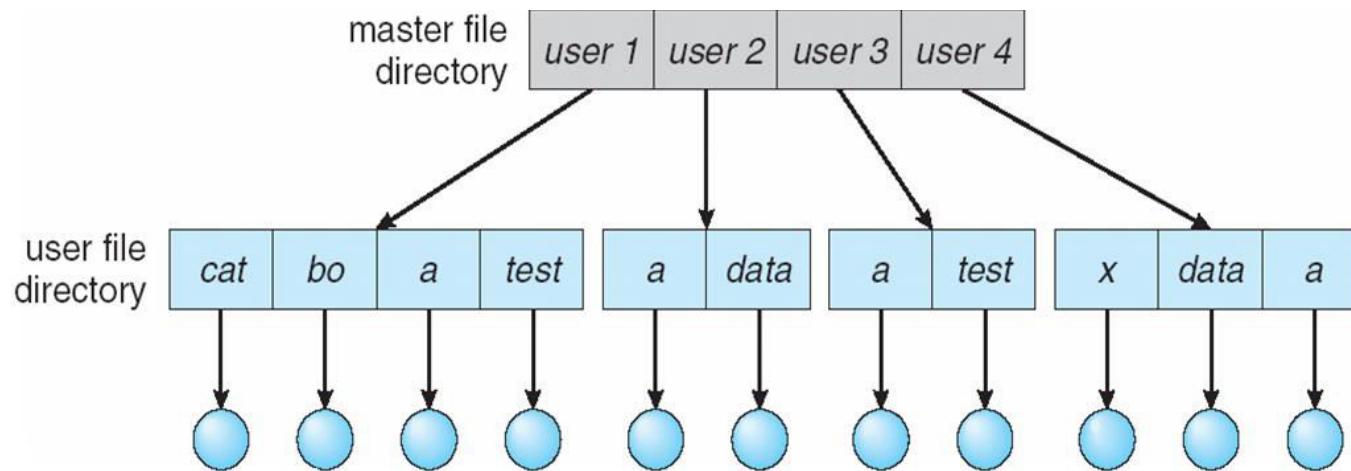
- A single directory for all users



- Naming problem
- Grouping problem

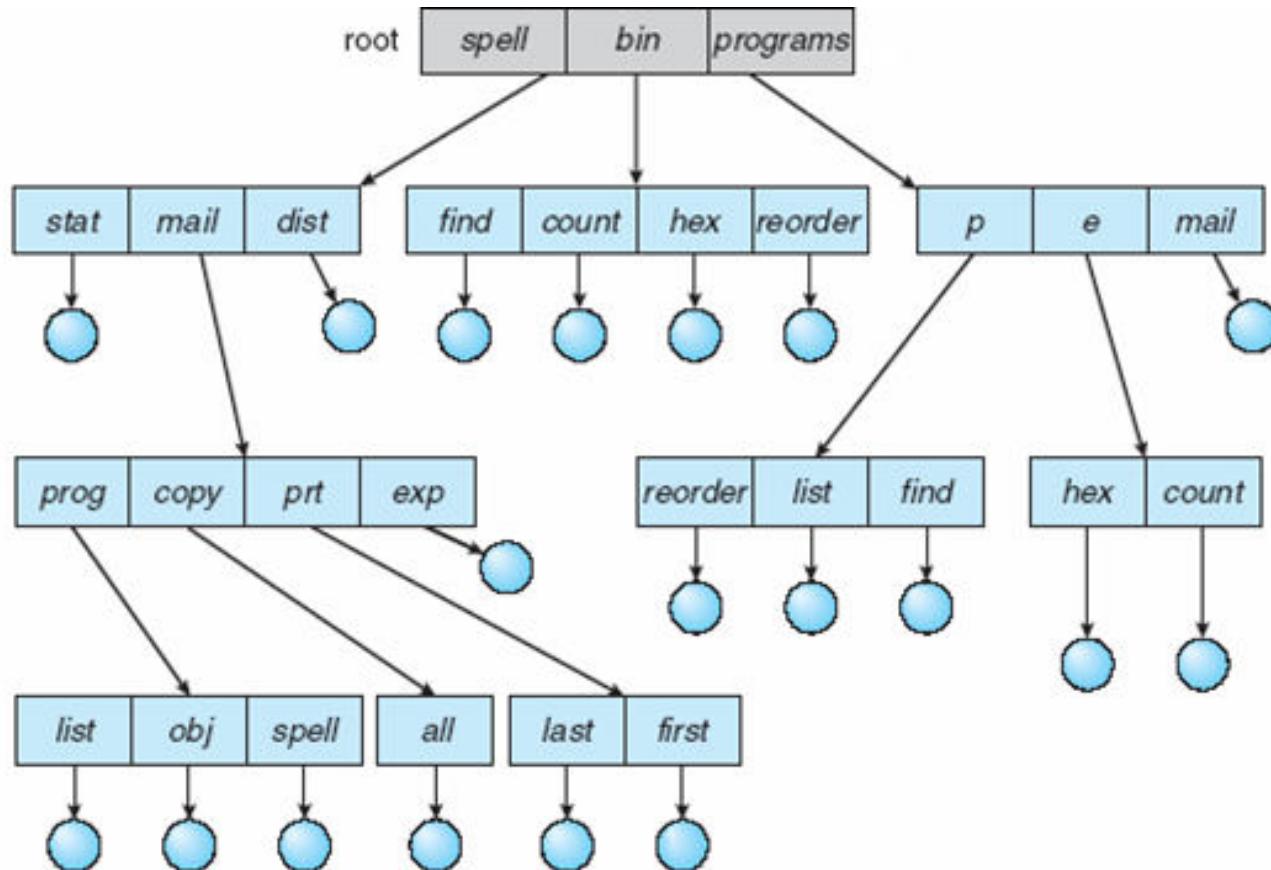
# Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

# Tree-Structured Directories

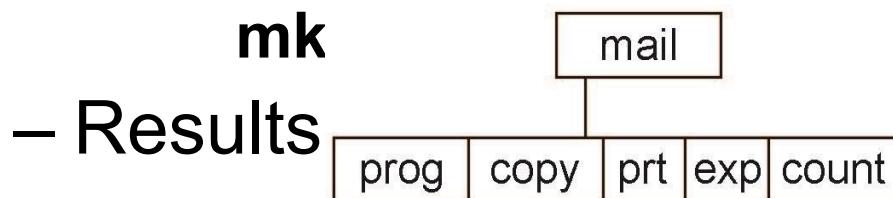


# Current Directory

- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`

# Current Directory (Cont.)

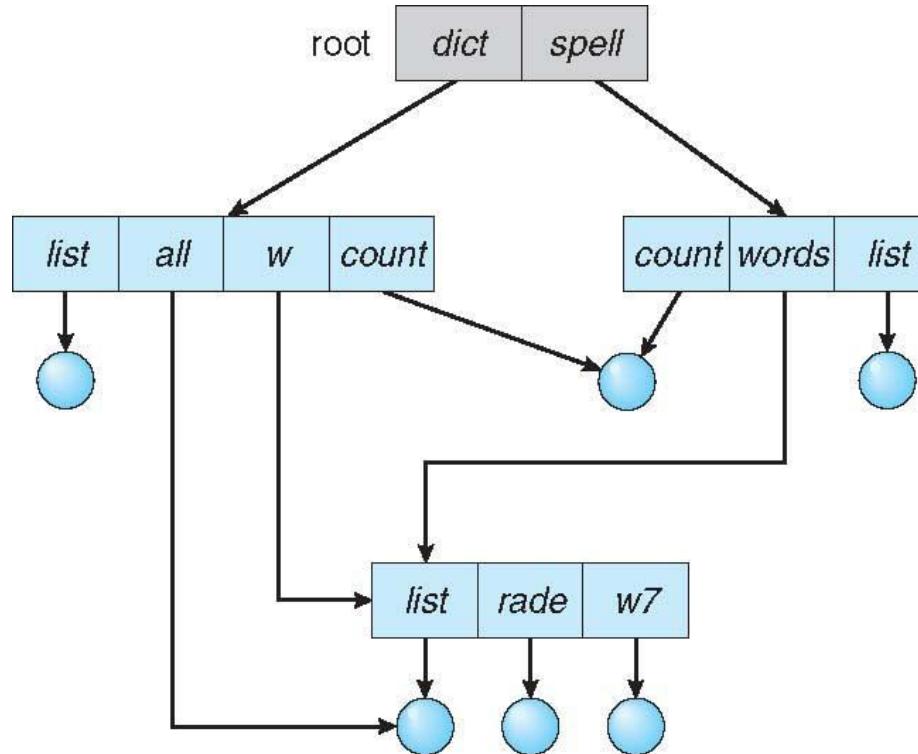
- Creating and deleting a file is done in current directory
- Example of creating a new file
  - If in current directory is **/mail**
  - The command



- Deleting “mail”  $\Rightarrow$  deleting the entire subtree rooted by “mail”

# Acyclic-Graph Directories

- Have shared subdirectories and files



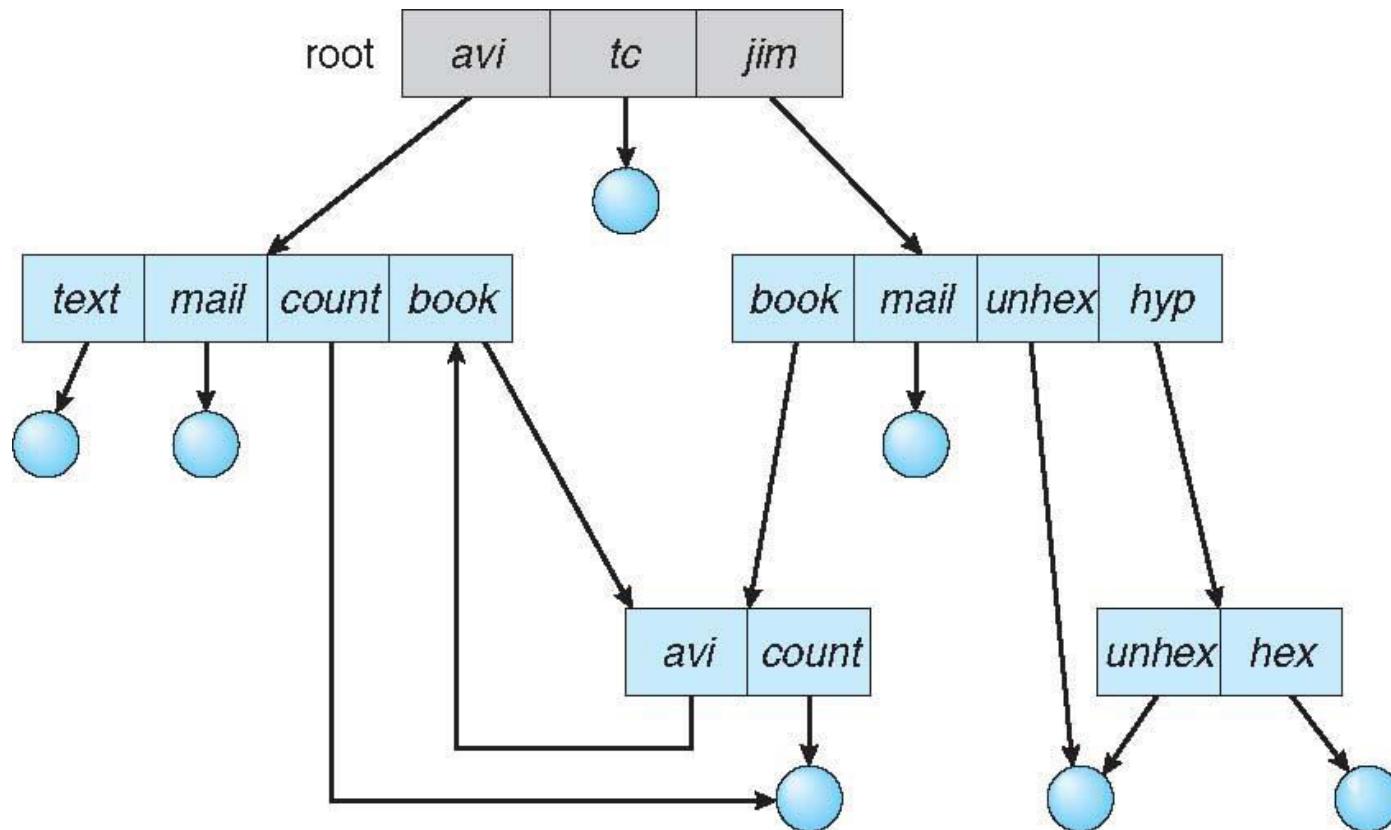
# Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If **dict** deletes w/**list** ⇒ 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
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

# General Graph Directory



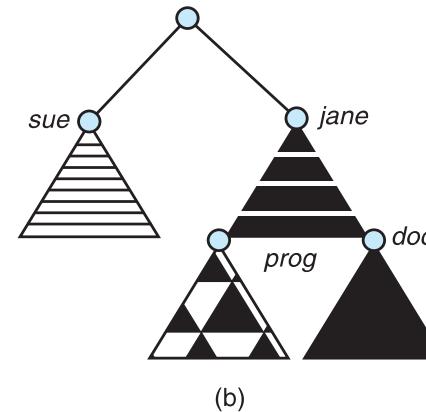
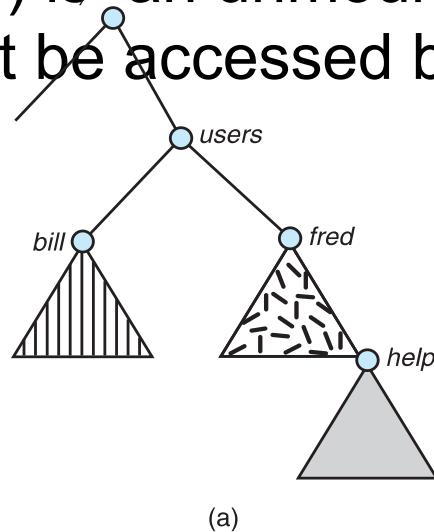
# General Graph Directory (Cont.)

---

- 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

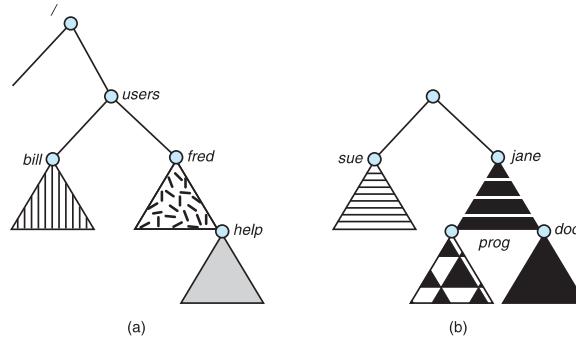
# File System Mounting

- A file system must be **mounted** before it can be accessed
- Fig (a) is a mounted file system that can be accessed by users.
- Fig. (b) is an unmounted files system that cannot be accessed by users

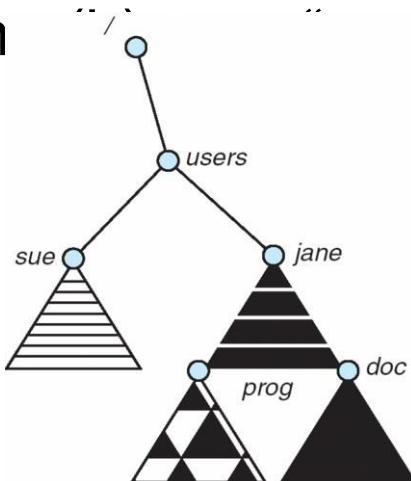


# Mount Point

- Consider the file system of previous slide:



- Mounting "users" results in



# 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
- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
  - **Group IDs** allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

# File Sharing – Remote File Systems

---

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

# File Sharing – Failure Modes

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

# File Sharing – Consistency Semantics

---

- Specify how multiple users are to access a shared file simultaneously
  - Similar to process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems)
  - Andrew File System (AFS) implemented complex remote file sharing semantics
  - Unix file system (UFS) implements:
    - Writes to an open file visible immediately to other users of the same open file
    - Sharing file pointer to allow multiple users to read and write concurrently
  - AFS has session semantics
    - Writes only visible to sessions starting after the file is closed

# Protection

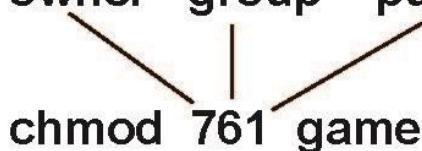
- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

# Access Lists and Groups in Unix

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

|                  |   | RWX     |
|------------------|---|---------|
| a) owner access  | 7 | ⇒ 1 1 1 |
| b) group access  | 6 | ⇒ 1 1 0 |
| c) public access | 1 | ⇒ 0 0 1 |

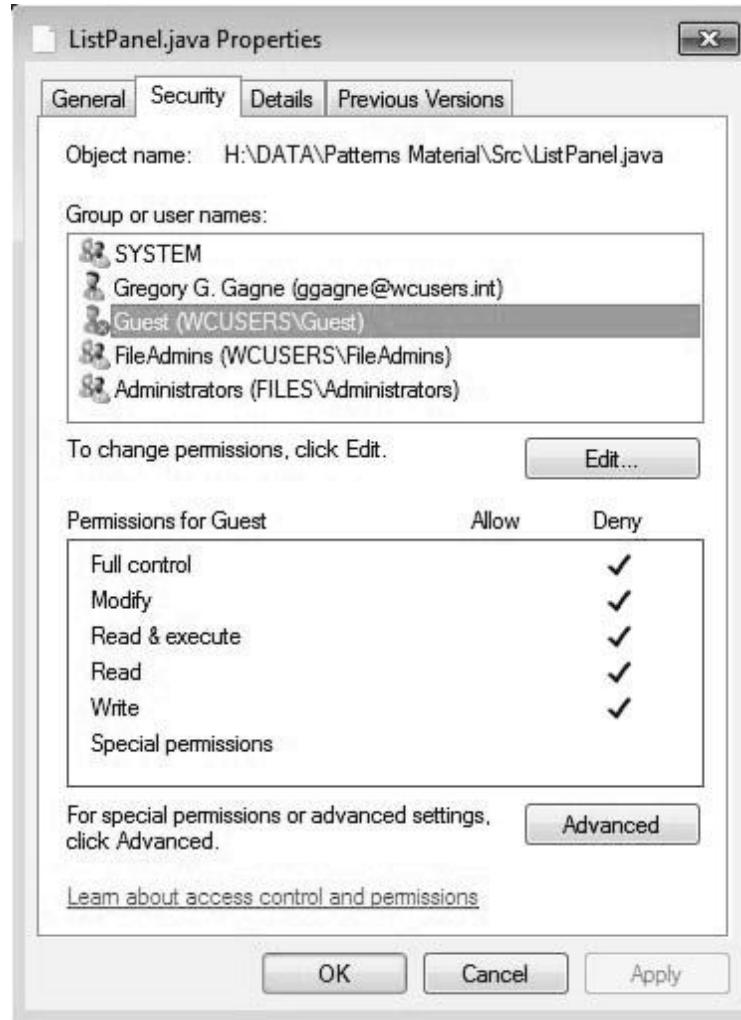
- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a file (say *game*) or subdirectory, define an appropriate acc



Attach a group to a file

`chgrp G game`

# Windows 7 Access-Control List Management



# A Sample UNIX Directory Listing

|            |   |     |         |       |              |               |
|------------|---|-----|---------|-------|--------------|---------------|
| -rw-rw-r-- | 1 | pbg | staff   | 31200 | Sep 3 08:30  | intro.ps      |
| drwx-----  | 5 | pbg | staff   | 512   | Jul 8 09:33  | private/      |
| drwxrwxr-x | 2 | pbg | staff   | 512   | Jul 8 09:35  | doc/          |
| drwxrwx--- | 2 | pbg | student | 512   | Aug 3 14:13  | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff   | 9423  | Feb 24 2003  | program.c     |
| -rwxr-xr-x | 1 | pbg | staff   | 20471 | Feb 24 2003  | program       |
| drwx--x--x | 4 | pbg | faculty | 512   | Jul 31 10:31 | lib/          |
| drwx-----  | 3 | pbg | staff   | 1024  | Aug 29 06:52 | mail/         |
| drwxrwxrwx | 3 | pbg | staff   | 512   | Jul 8 09:35  | test/         |

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## CSE-202 OPERATING SYSTEM

### Module V: File System Implementation

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling, Pearson Education**

## **Course Learning Objectives:**

- To understand how files are stored on hard disk and how operating system manages free space
- To understand how operating system implements security & protection

# Topics to be covered

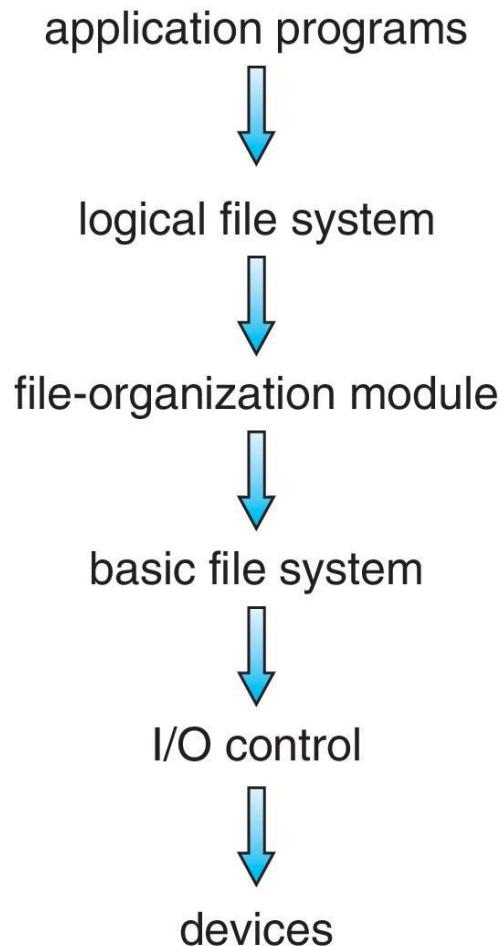
---

- Basic file system,
- File Sharing,
- Allocation method,
- Free space management.
- Policy Mechanism,
- Authentication,
- Internal access Authorization.

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block (FCB)** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

# Layered File System

---



# File System Layers

---

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

# File System Layers (Cont.)

---

- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

# File System Layers (Cont.)

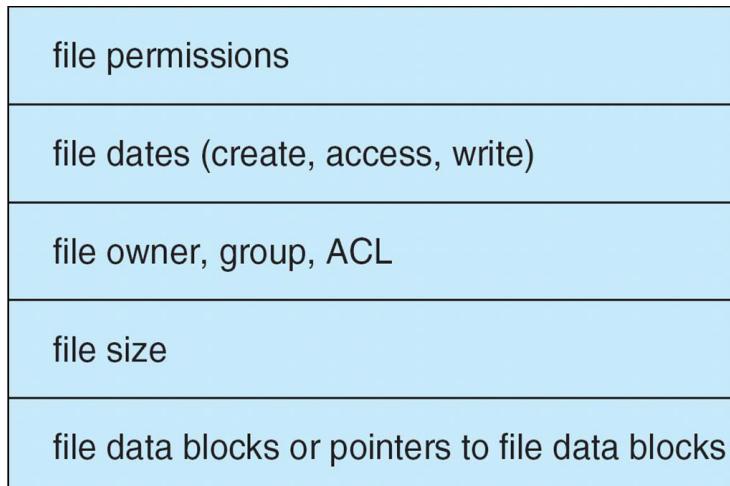
---

- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

# File-System Implementation (Cont.)

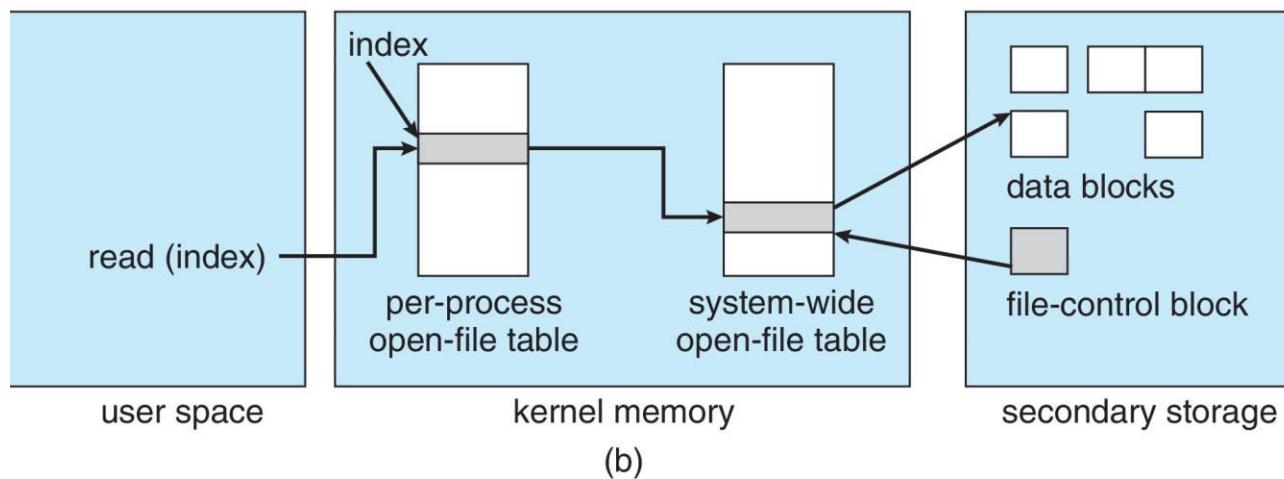
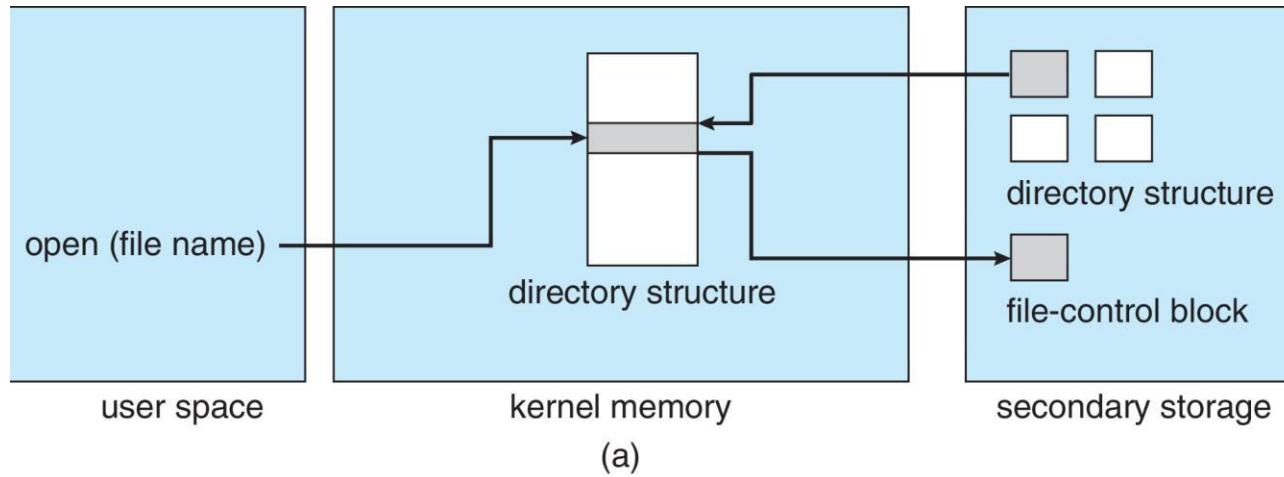
- Per-file **File Control Block (FCB)** contains many details about the file
  - Typically inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures



# In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure (a) refers to opening a file
- Figure (b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

# In-Memory File System Structures



# Directory Implementation

---

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

# Allocation Methods - Contiguous

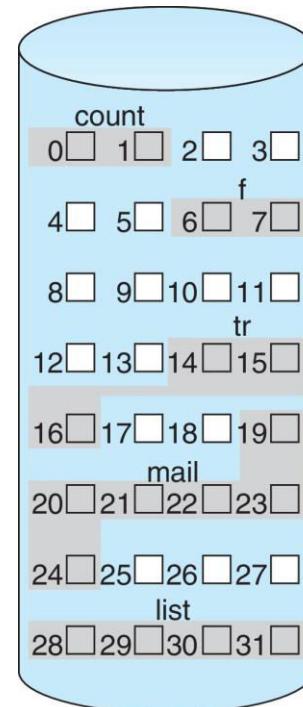
---

- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - Finding space for file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line (downtime)** or **on-line**

# Contiguous Allocation

- Mapping from logical to physical (block size =512 bytes)

LA/512  
 Q  
 R



| directory |       |        |
|-----------|-------|--------|
| file      | start | length |
| count     | 0     | 2      |
| tr        | 14    | 3      |
| mail      | 19    | 6      |
| list      | 28    | 4      |
| f         | 6     | 2      |

- Block to be accessed = starting address + Q
- Displacement into block = R

# Extent-Based Systems

---

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

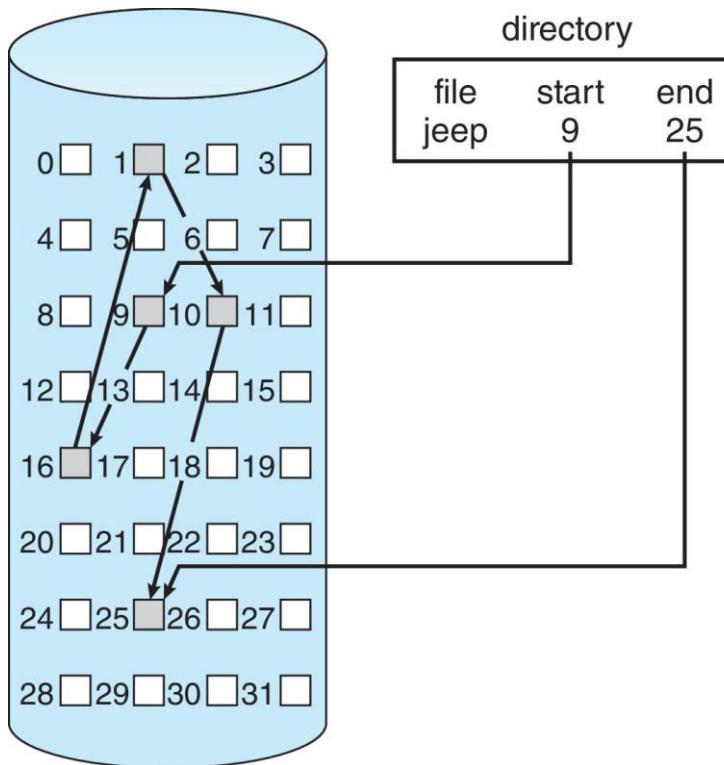
# Allocation Methods - Linked

---

- Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks

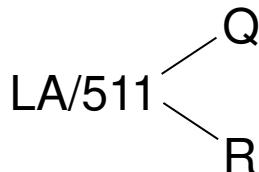
# Linked Allocation

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



# Linked Allocation

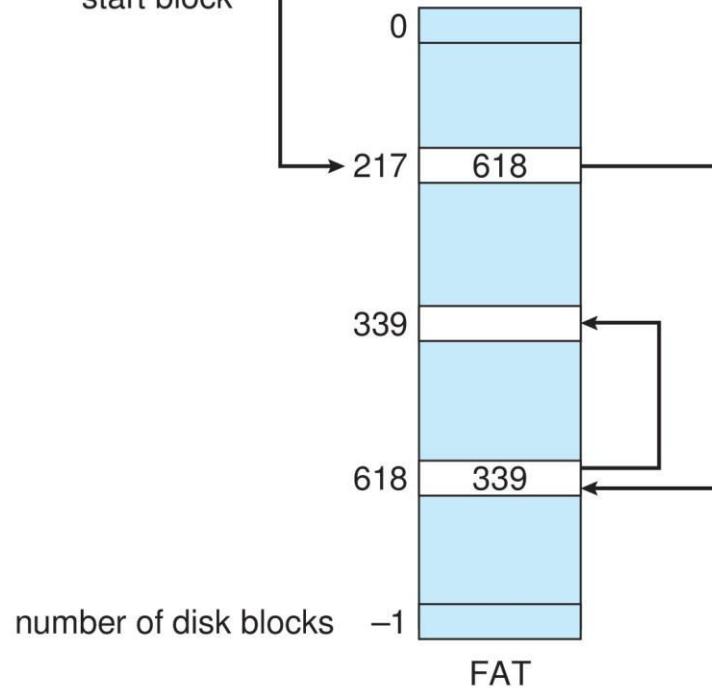
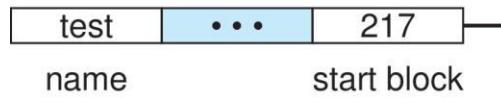
- Mapping



- Block to be accessed is the  $Q$ th block in the linked chain of blocks representing the file.
- Displacement into block =  $R + 1$

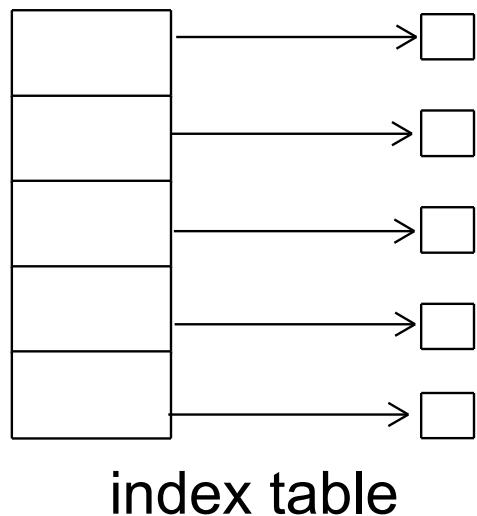
# File-Allocation Table

directory entry

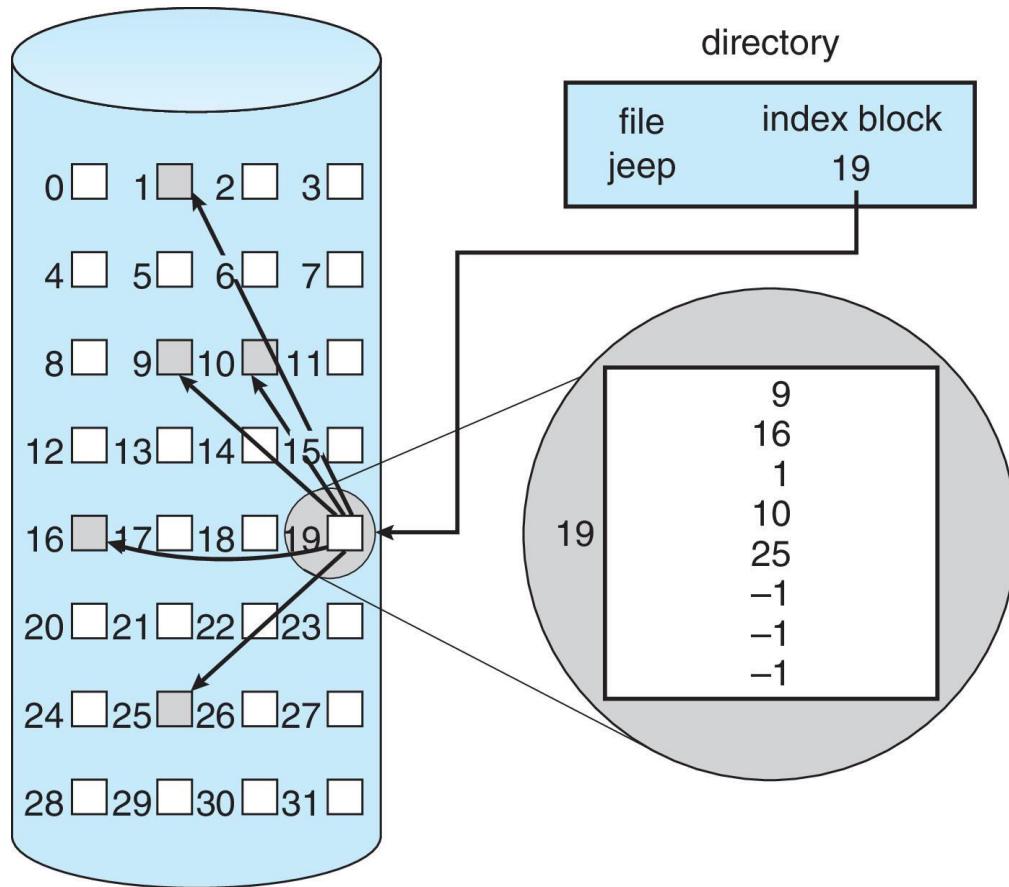


# Allocation Methods - Indexed

- Each file has its own **index block(s)** of pointers to its data blocks
- Logical view

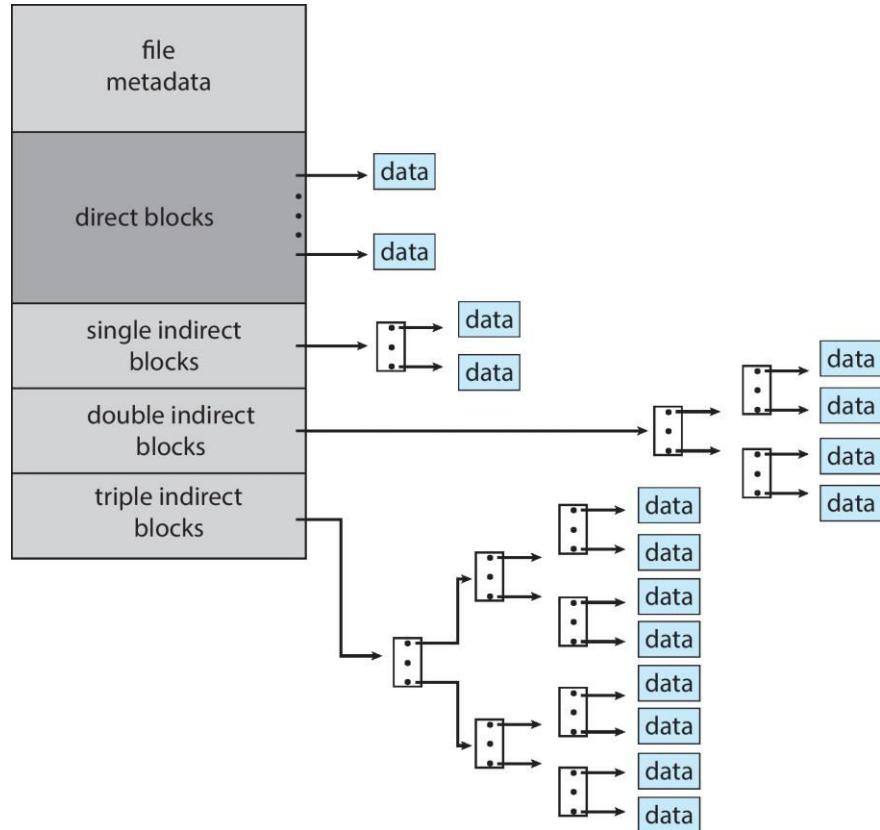


# Example of Indexed Allocation



# Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

# Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - With NVM goal is to reduce CPU cycles and overall path needed for I/O

# Free-Space Management

---

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term “block” for simplicity)
- **Bit vector** or **bit map** ( $n$  blocks)



Block number calculation

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

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

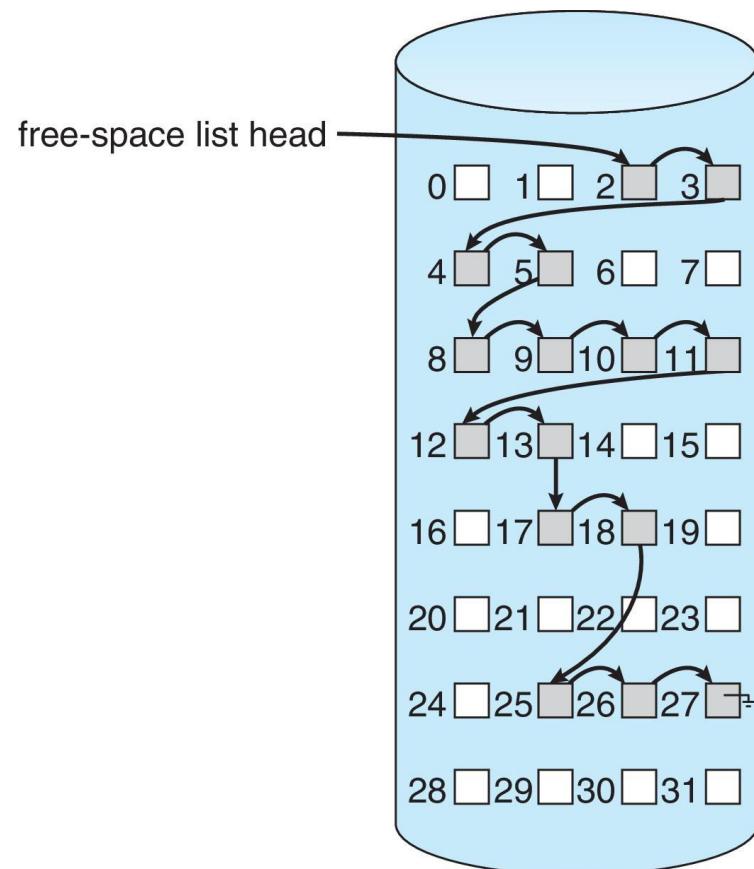
CPUs have instructions to return offset within word of first “1” bit

# Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:
    - block size = 4KB =  $2^{12}$  bytes
    - disk size =  $2^{40}$  bytes (1 terabyte)
    - $n = 2^{40}/2^{12} = 2^{28}$  bits (or 32MB)
    - if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files

# Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)



# Free-Space Management (Cont.)

---

- Grouping
  - Modify linked list to store address of next  $n-1$  free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

# Free-Space Management (Cont.)

- Space Maps
  - Used in **ZFS**
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - Uses counting algorithm
  - But records to log file rather than file system
    - Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry

# TRIMing Unused Blocks

---

- HDDs overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (like NVM) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - Can be garbage collected or if block is free, now block can be erased

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

# Efficiency and Performance (Cont.)

---

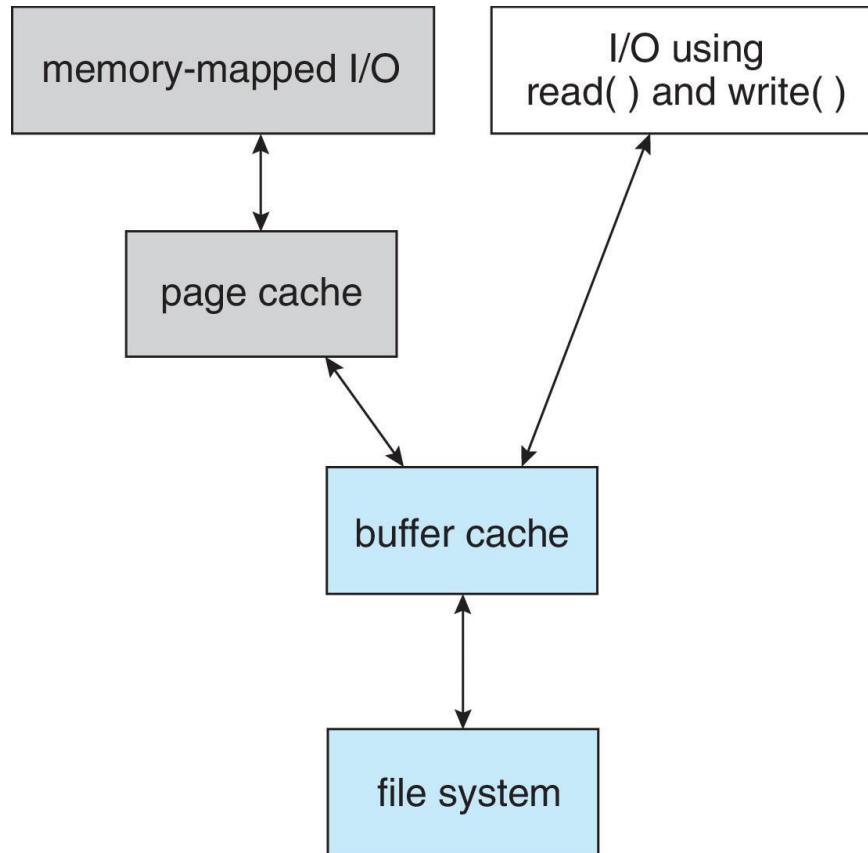
- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

# Page Cache

---

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- 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

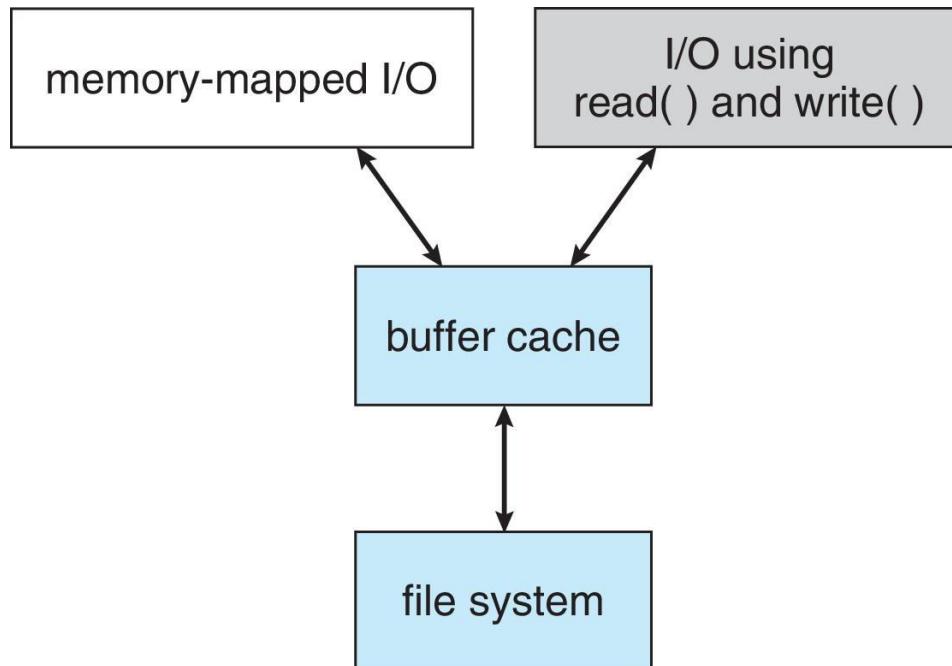


# Unified Buffer Cache

---

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?

# 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
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

# The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

[Apple File System \(APFS\)](#) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. [Space sharing](#) is a ZFS-like feature in which storage is available as one or more large free spaces ([containers](#)) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). [Fast directory sizing](#) provides quick used-space calculation and updating. [Atomic safe-save](#) is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**

## CSE-202 OPERATING SYSTEM

### Module V: Mass Storage Systems

# Operating System

## Text Book

- **Operating System Concepts : *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg , Wiley Publisher***

## Reference Book

- **Operating system: William Stalling, Pearson Education**

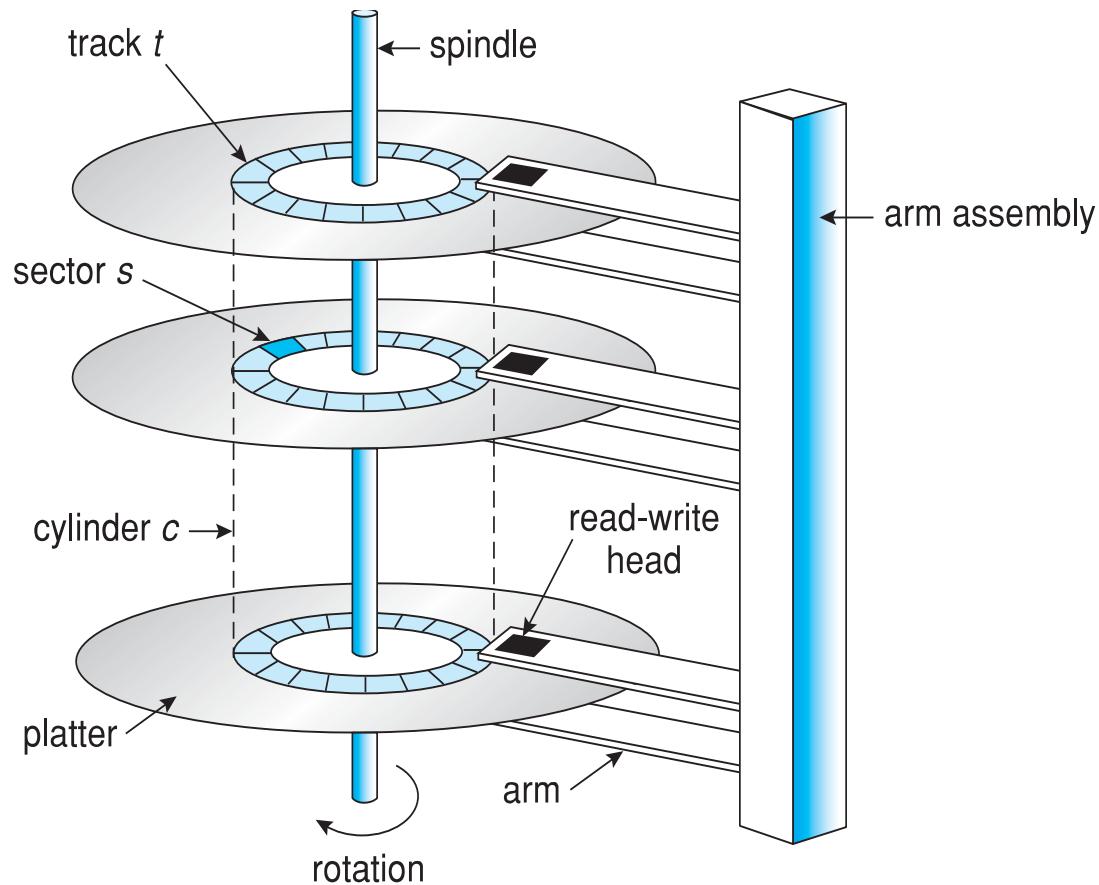
## **Course Learning Objectives:**

- To understand the structure and complexities of an operating system's I/O subsystem

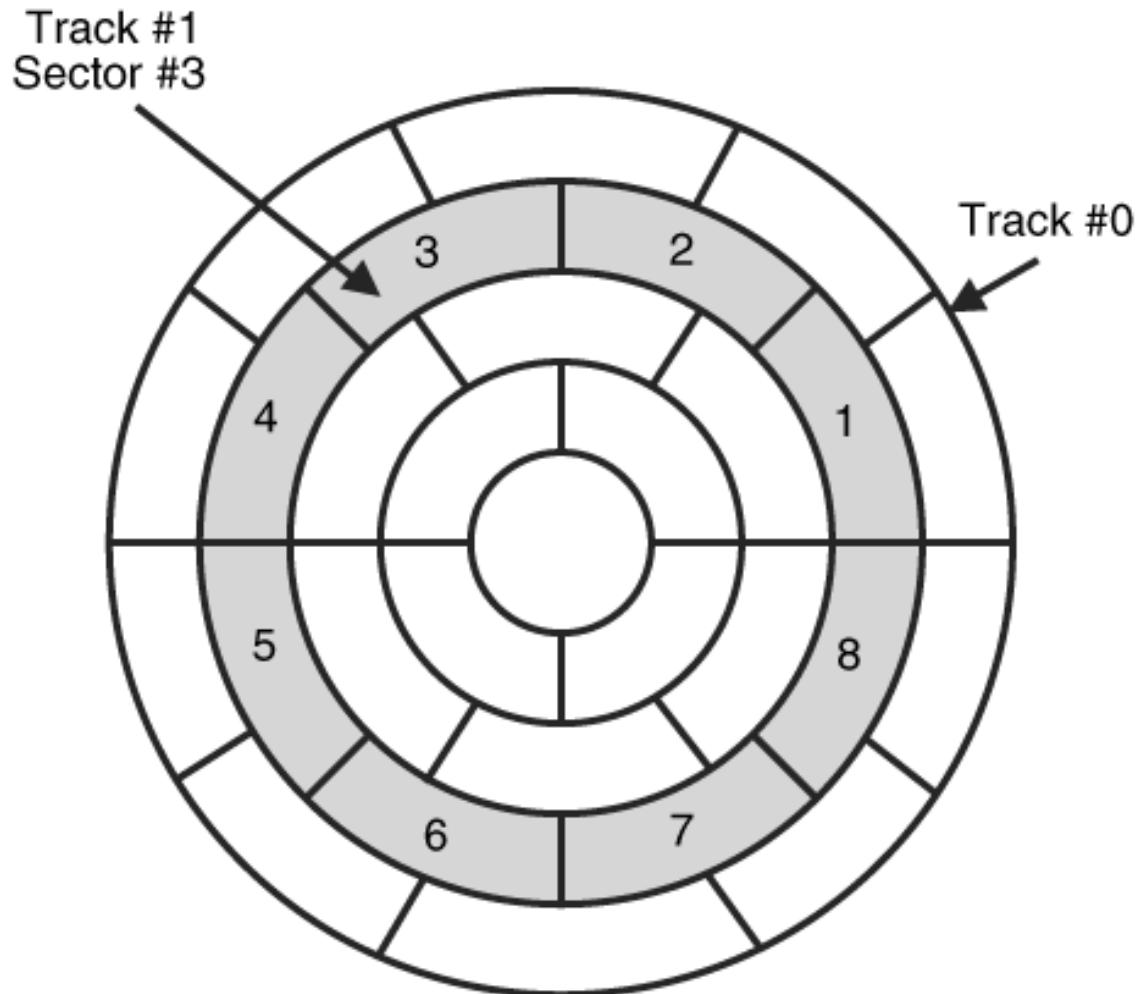
# Overview of Mass Storage Structure

- Bulk of secondary storage for modern computers is **hard disk drives (HDDs)** and **nonvolatile memory (NVM)** devices
- **HDDs** spin platters of magnetically-coated material under moving read-write heads
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface -- That's bad
- Disks can be removable

# Moving-head Disk Mechanism

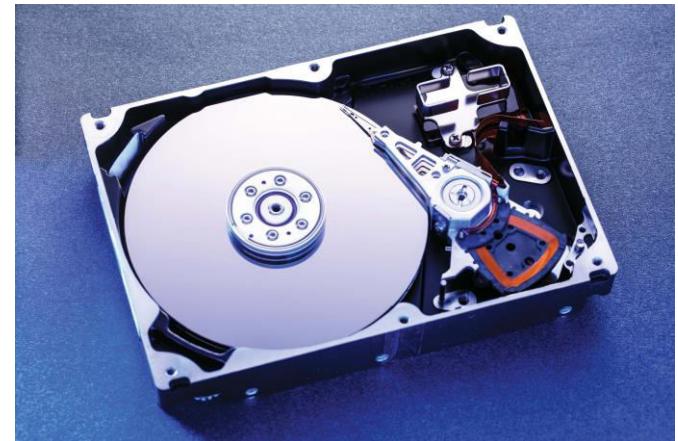


# Disk Structure



# Hard Disk Drives

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
  - Average latency =  $\frac{1}{2}$  latency



# Hard Disk Performance

---

- **Access Latency = Average access time** = average seek time + average latency (milliseconds)
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
  - 5ms + 4.17ms + 0.1ms + transfer time =
  - Transfer time =  $4\text{KB} / 1\text{Gb/s} * 8\text{Gb / GB} * 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2) = 0.031 \text{ ms}$
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

# The First Commercial Disk Drive



1956

IBM RAMDAC computer  
included the IBM Model 350  
disk storage system

5M (7 bit) characters

50 x 24" platters

Access time = < 1 second

# Nonvolatile Memory Devices

---

- If disk-drive like, then called **solid-state disks (SSDs)**
- Other forms include **USB drives** (thumb drive, flash drive), DRAM disk replacements, surface-mounted on motherboards, and main storage in devices like smartphones
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span – need careful management
- Less capacity
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency

# Nonvolatile Memory Devices

- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases happen in larger ”block” increments
  - Can only be erased a limited number of times before worn out –  
~ 100,000
  - Life span measured in **drive writes per day (DWPD)**
    - A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing



# Volatile Memory

- DRAM frequently used as mass-storage device
  - Not technically secondary storage because volatile, but can have file systems, be used like very fast secondary storage
- **RAM drives** (with many names, including RAM disks) present as raw block devices, commonly file system formatted
- Computers have buffering, caching via RAM, so why RAM drives?
  - Caches / buffers allocated / managed by programmer, operating system, hardware
  - RAM drives under user control
  - Found in all major operating systems
    - Linux /dev/ram, macOS diskutil to create them, Linux /tmp of file system type tmpfs
- Used as high speed temporary storage
  - Programs could share bulk date, quickly, by reading/writing to RAM drive

# Magnetic Tape

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.



**Figure 11.5** An LTO-6 Tape drive with tape cartridge inserted.

# Disk Attachment

- Host-attached storage accessed through I/O ports talking to **I/O busses**
- Several busses available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **serial attached SCSI (SAS)**, **universal serial bus (USB)**, and **fibre channel (FC)**.
- Most common is SATA
- Because NVM much faster than HDD, new fast interface for NVM called **NVM express (NVMe)**, connecting directly to PCI bus
- Data transfers on a bus carried out by special electronic processors called **controllers** (or **host-bus adapters, HBAs**)
  - Host controller on the computer end of the bus, device controller on device end
  - Computer places command on host controller, using memory-mapped I/O ports
    - Host controller sends messages to device controller
    - Data transferred via DMA between device and computer DRAM

# Address Mapping

---

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address should be easy
    - Except for bad sectors
    - Non-constant # of sectors per track via constant angular velocity

# HDD Scheduling

---

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time  $\approx$  seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

# Disk Scheduling (Cont.)

---

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists

# Disk Scheduling (Cont.)

---

- In the past, operating system responsible for queue management, disk drive head scheduling
  - Now, built into the storage devices, controllers
  - Just provide Logical Block Addressing (LBA), handle sorting of requests
    - Some of the algorithms they use described next

# Disk Scheduling (Cont.)

---

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms 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

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

head starts at 53



Total head movements incurred while servicing these requests

$$\begin{aligned}
 &= (98 - 53) + (183 - 98) + (183 - 37) + (122 - 37) + (122 - 14) + (124 - 14) + \\
 &\quad (124 - 65) + (67 - 65) \\
 &= 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = 640
 \end{aligned}$$

# SSTF

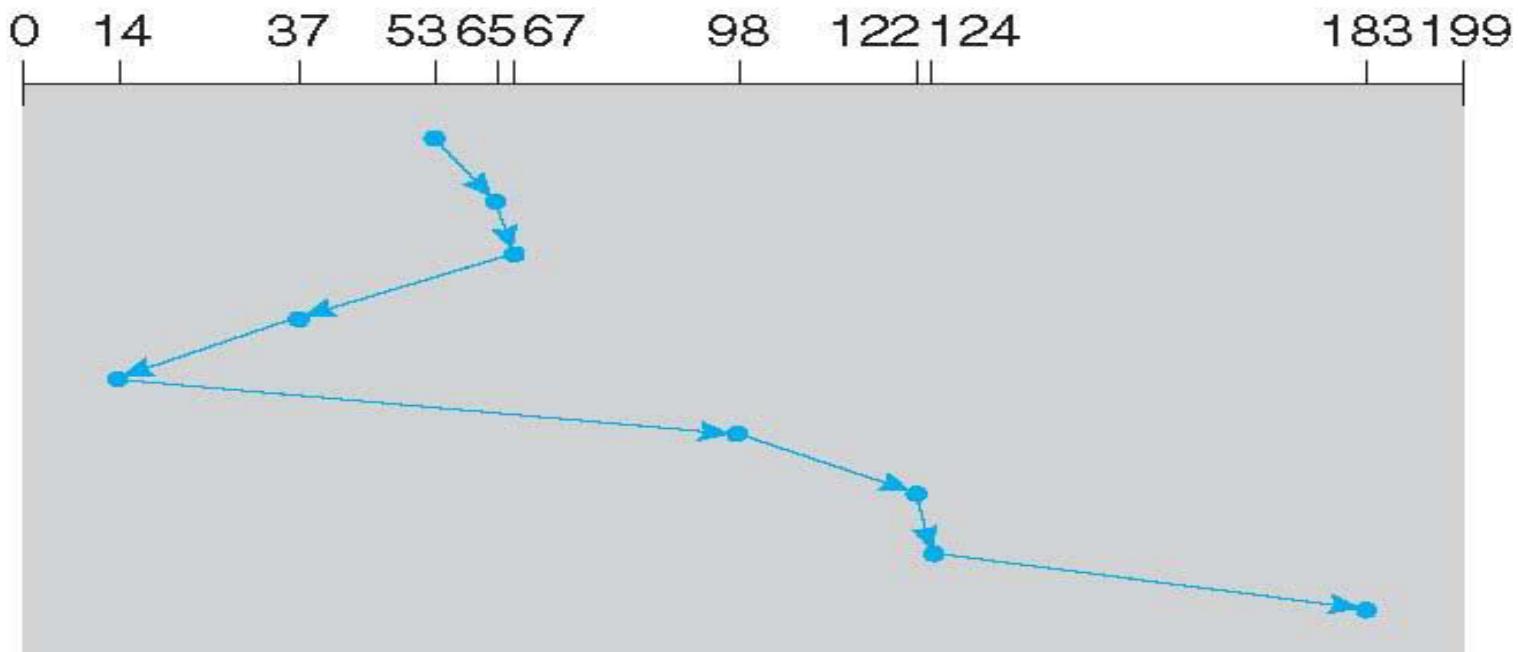
---

- 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



$$\begin{aligned} &= (65 - 53) + (67 - 65) + (67 - 37) + (37 - 14) + (98 - 14) + (122 - 98) \\ &\quad + (124 - 122) + (183 - 124) \end{aligned}$$

$$= 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59$$

$$= 236$$

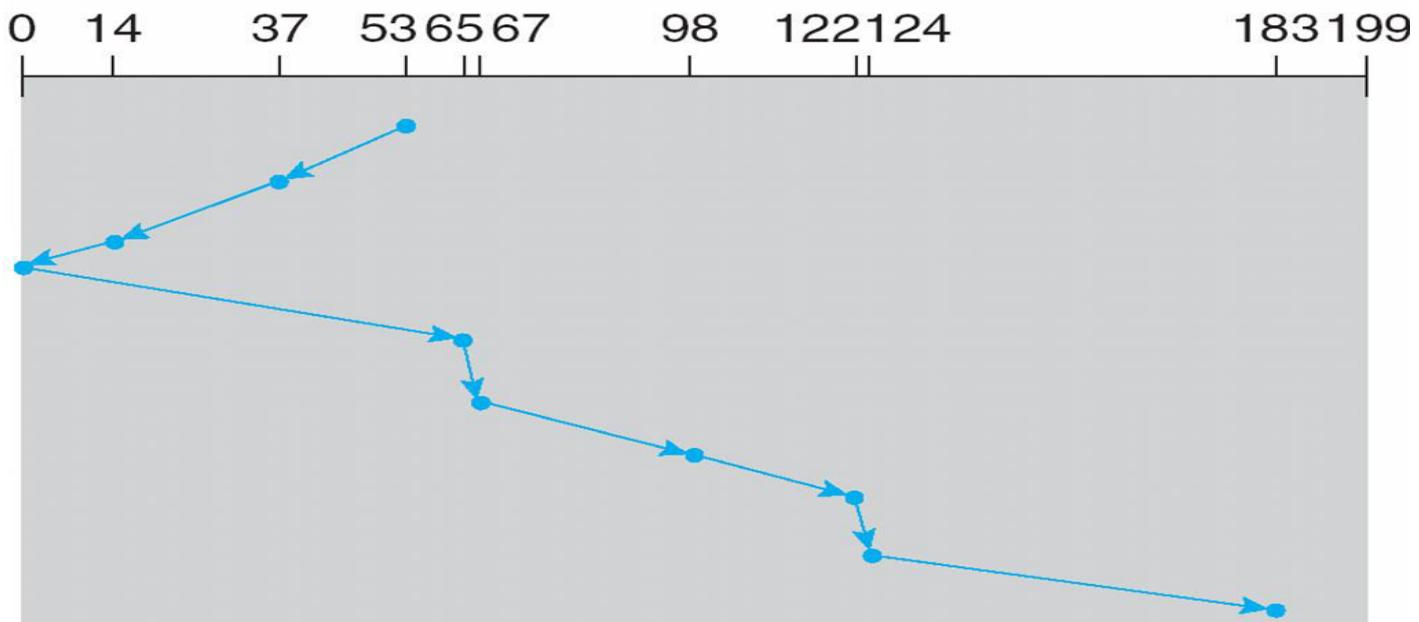
# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders

# SCAN (Cont.)

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

head starts at 53



$$\begin{aligned}
 &= (53 - 37) + (37 - 14) + (14 - 0) + (65 - 0) + (67 \\
 &- 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - \\
 &124)
 \end{aligned}$$

$$= 16 + 23 + 14 + 65 + 2 + 31 + 24 + 2 + 59$$

$$= 236$$

Alternatively  
Total head movements incurred  
while servicing these requests

$$\begin{aligned} &= (53-0) + (183 - 0) \\ &= 53 + 183 \\ &= 236 \end{aligned}$$

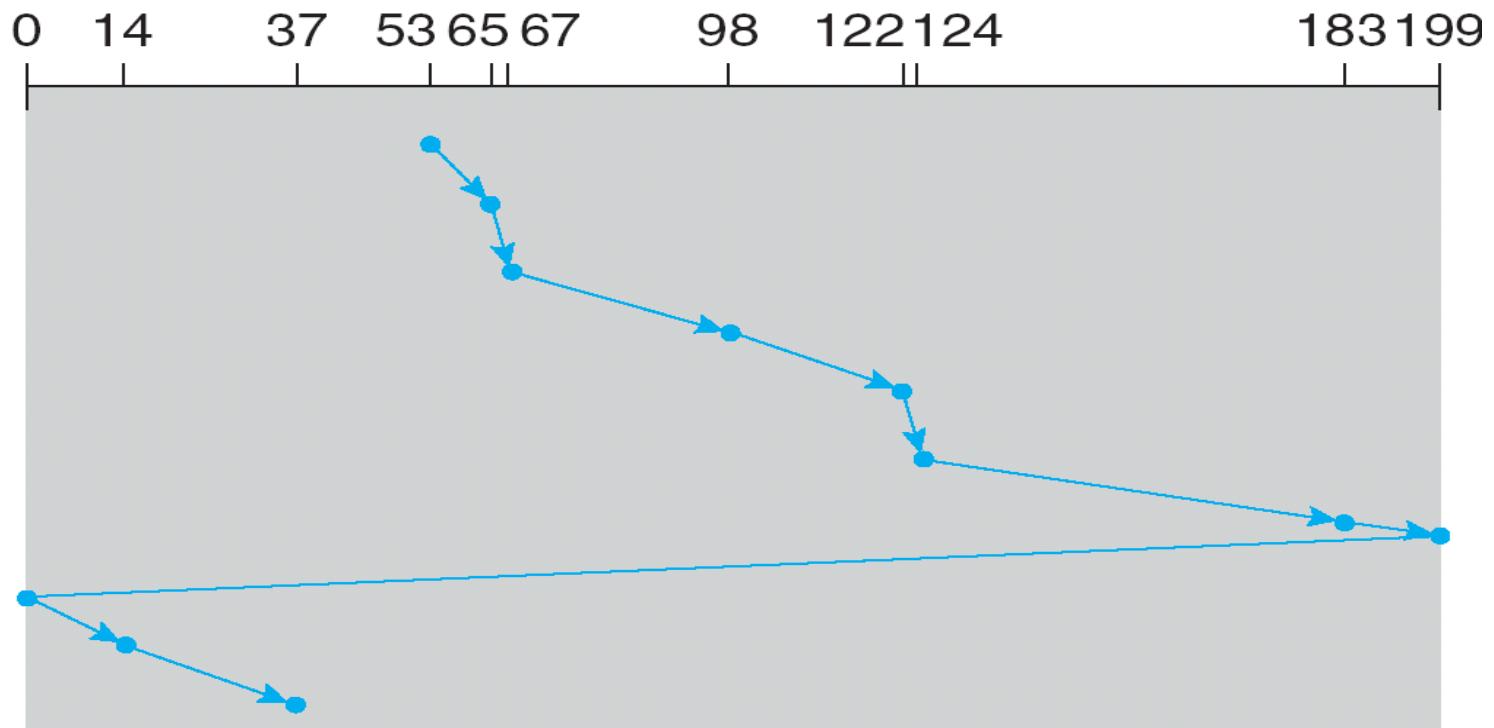
# C-SCAN

- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- 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



Total head movements incurred while servicing these requests

$$= (199 - 53) + (199 - 0) + (37 - 0)$$

$$= 146 + 199 + 37$$

$$= 382$$

# LOOK and C-LOOK

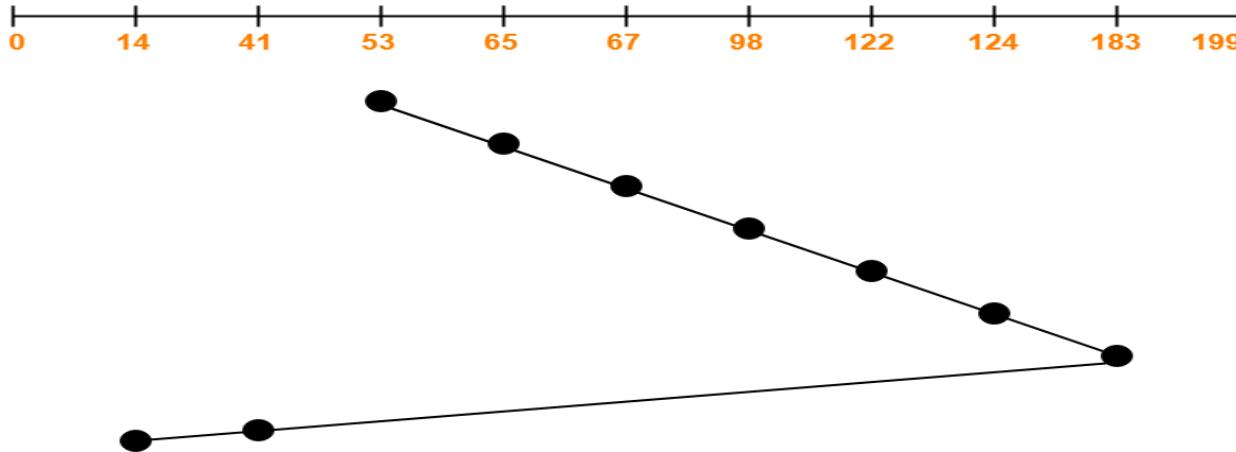
---

- Version of SCAN and 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

Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67.

The LOOK scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. T

he cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is \_\_\_\_\_.

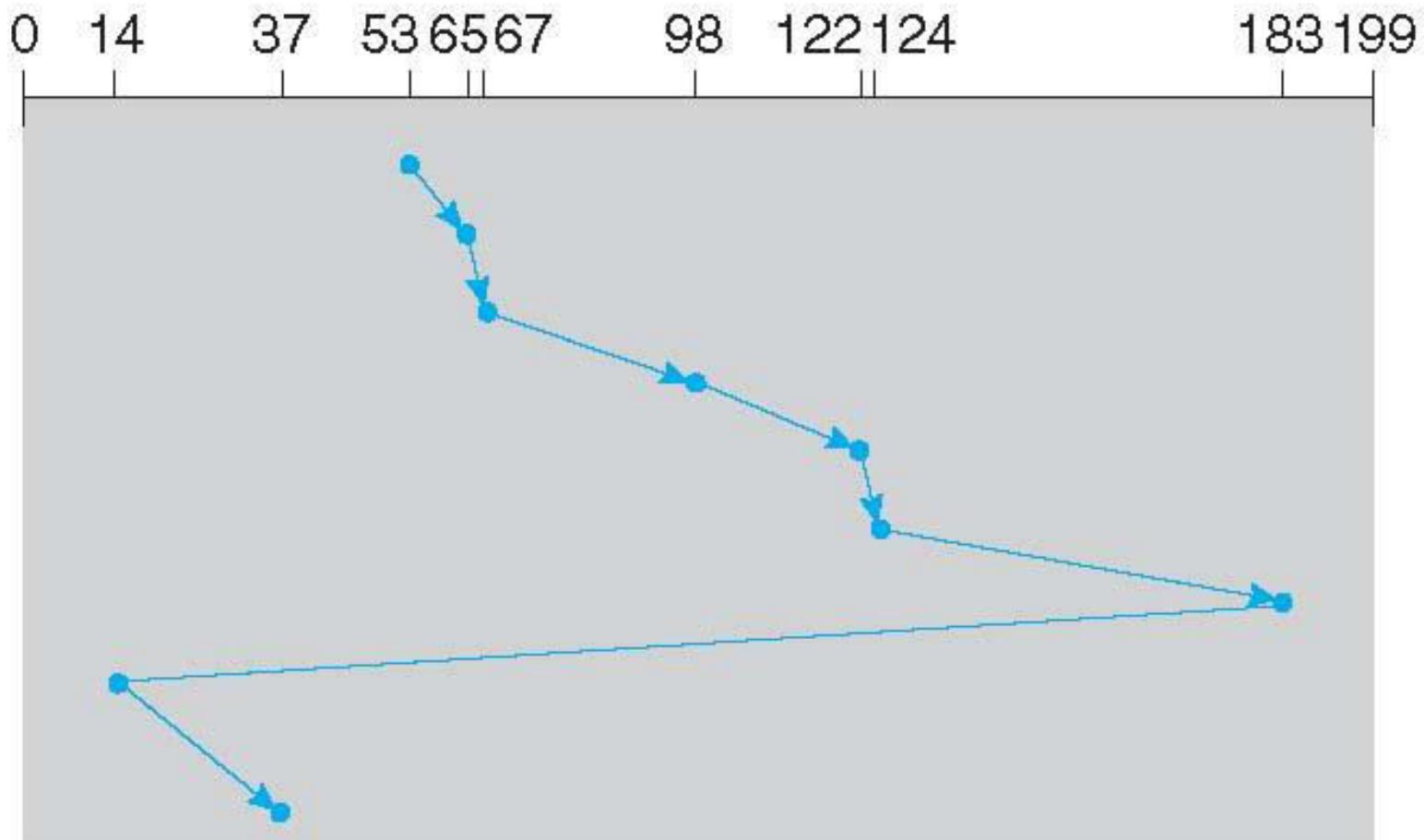


- Total head movements incurred while servicing these requests
- $= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - 124) + (183 - 41) + (41 - 14)$
- $= 12 + 2 + 31 + 24 + 2 + 59 + 142 + 27$
- $= 299$

# C-LOOK (Cont.)

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

head starts at 53



# 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 LOOK is a reasonable choice for the default algorithm

# Storage Device Management

---

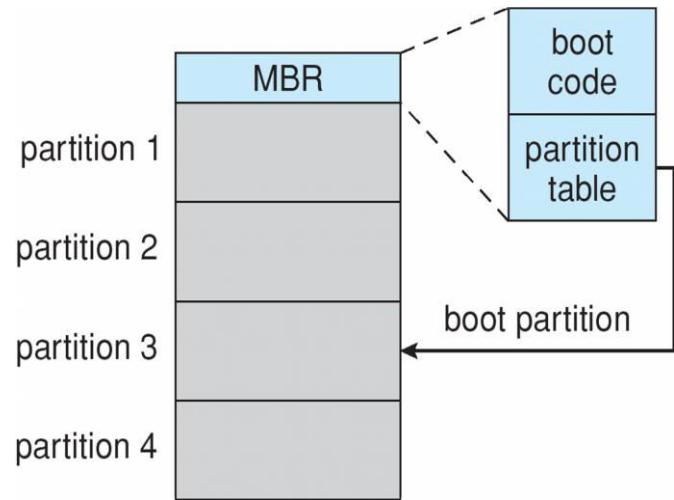
- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or “making a file system”
  - To increase efficiency most file systems group blocks into **clusters**
    - Disk I/O done in blocks
    - File I/O done in clusters

# Storage Device Management (cont.)

- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - **Mounted** at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting

# Device Storage Management (Cont.)

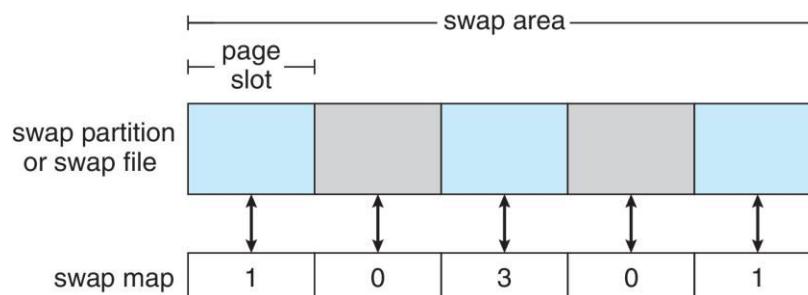
- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
  - The bootstrap is stored in ROM, firmware
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks



Booting from secondary storage in Windows

# Swap-Space Management

- Used for moving entire processes (swapping), or pages (paging), from DRAM to secondary storage when DRAM not large enough for all processes
- Operating system provides **swap space management**
  - Secondary storage slower than DRAM, so important to optimize performance
  - Usually multiple swap spaces possible – decreasing I/O load on any given device
  - Best to have dedicated devices
  - Can be in raw partition or a file within a file system (for convenience of adding)
  - Data structures for swapping on Linux systems:

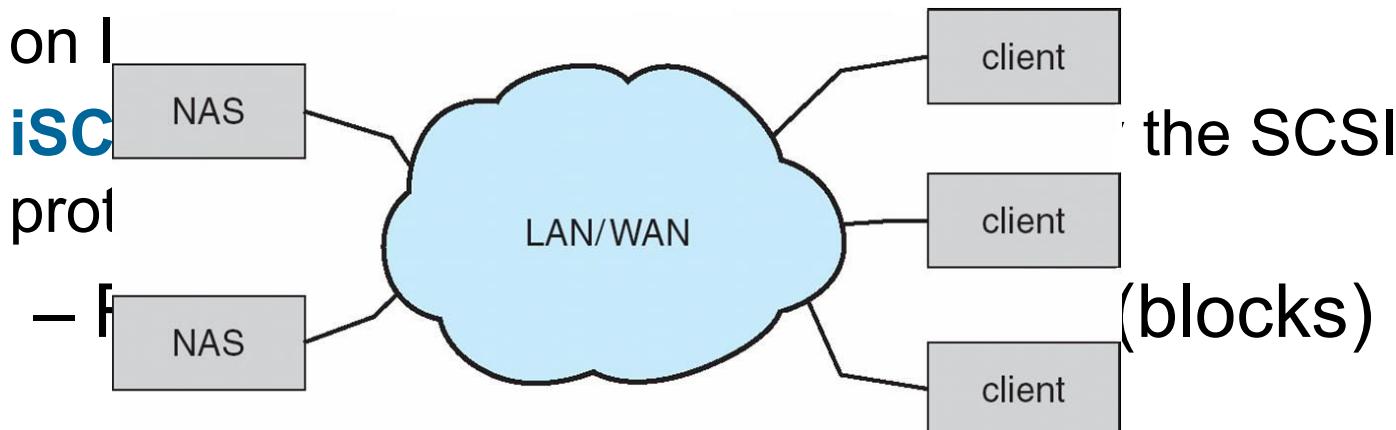


# Storage Attachment

- Computers access storage in three ways
  - Host-attached
  - Network-attached
  - Cloud
- Host attached access through local I/O ports, using one of several technologies
  - To attach many devices, use storage busses such as USB, firewire, thunderbolt
  - High-end systems use **fibre channel (FC)**
    - High-speed serial architecture using fibre or copper cables
    - Multiple hosts and storage devices can connect to the FC fabric

# Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on I
- **iSCSI** protocol
  - F



# Cloud Storage

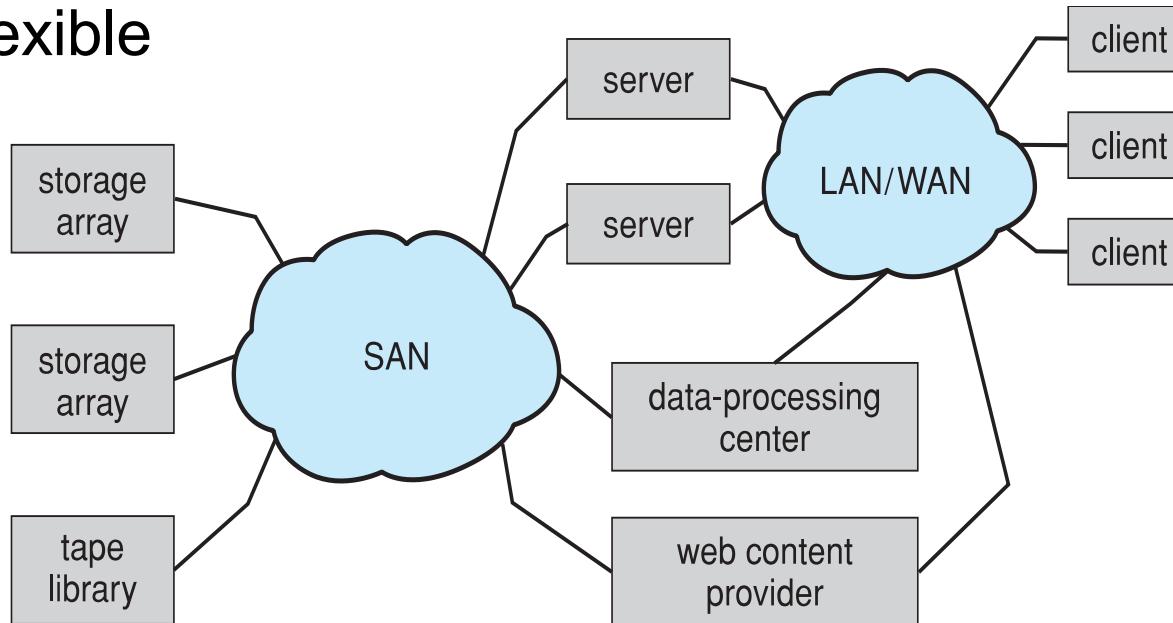
- Similar to NAS, provides access to storage across a network
  - Unlike NAS, accessed over the Internet or a WAN to remote data center
- NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access
  - Examples include Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well)

# Storage Array

- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc.)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - Snapshots, clones, thin provisioning, replication, deduplication, etc

# Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays – flexible



# Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches or **InfiniBand (IB)** network
- Hosts also attach to the switches
- Storage made available via **LUN Masking** from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE



A Storage Array

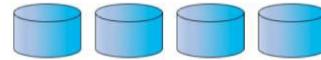
# RAID Structure

- RAID – redundant array of inexpensive disks
  - multiple disk drives provides reliability via redundancy
- Increases the mean time to failure
- Mean time to repair – exposure time when another failure could cause data loss
- Mean time to data loss based on above factors
- If mirrored disks fail independently, consider disk with 1300,000 mean time to failure and 10 hour mean time to repair
  - Mean time to data loss is  $100,000^2 / (2 * 10) = 500 * 10^6$  hours, or 57,000 years!
- Frequently combined with NVRAM to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively

# RAID (Cont.)

- Disk **striping** uses a group of disks as one storage unit
- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - **Mirroring** or **shadowing** (**RAID 1**) keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity** (**RAID 4, 5, 6**) uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

# RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



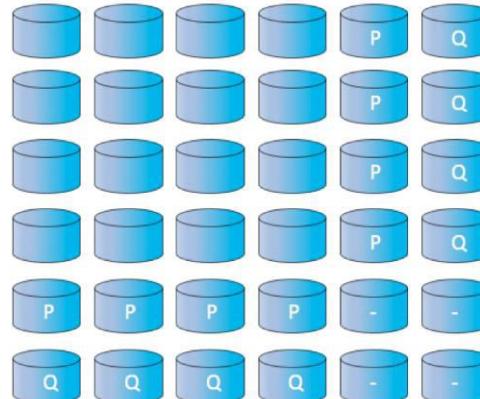
(c) RAID 4: block-interleaved parity.



(d) RAID 5: block-interleaved distributed parity.

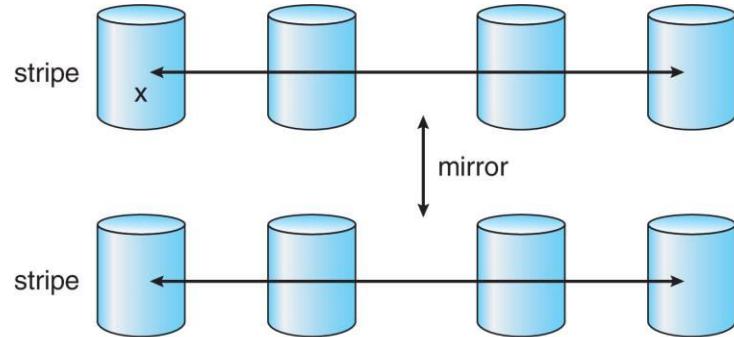


(e) RAID 6: P + Q redundancy.

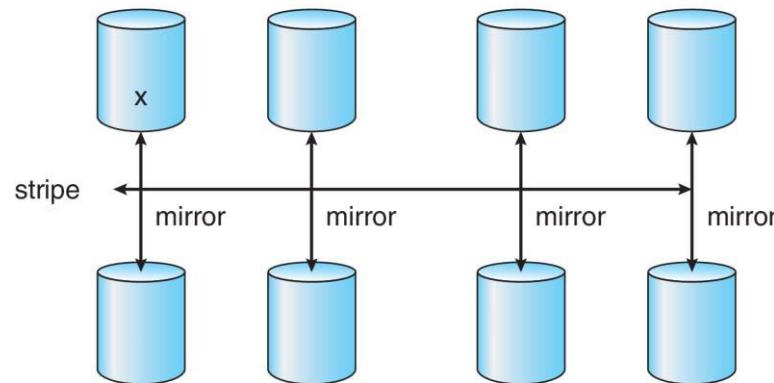


(f) Multidimensional RAID 6.

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



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

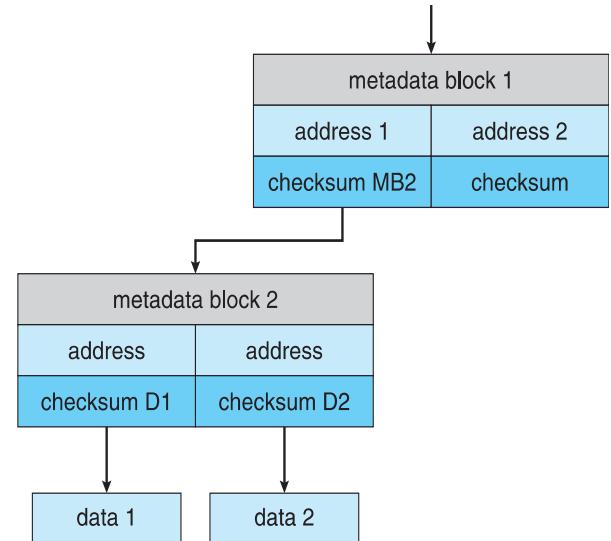
# Other Features

---

- Regardless of where RAID implemented, other useful features can be added
- **Snapshot** is a view of file system before a set of changes take place (i.e., at a point in time)
  - More in Ch 12
- Replication is automatic duplication of writes between separate sites
  - For redundancy and disaster recovery
  - Can be synchronous or asynchronous
- Hot spare disk is unused, automatically used by RAID production if a disk fails to replace the failed disk and rebuild the RAID set if possible
  - Decreases mean time to repair

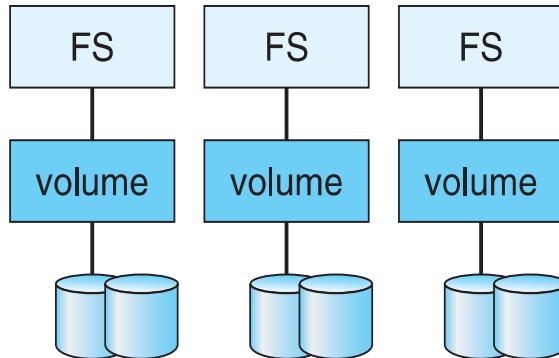
# Extensions

- RAID alone does not prevent or detect data corruption or other errors, just disk failures
- Solaris ZFS adds **checksums** of all data and metadata
- Checksums kept with pointer to object, to detect if object is the right one and whether it changed
- Can detect and correct data and metadata corruption
- ZFS also removes volumes, partitions
  - Disks allocated in **pools**
  - Filesystems with a pool share that pool, use and release space like `malloc()` and `free()` memory allocate / release calls

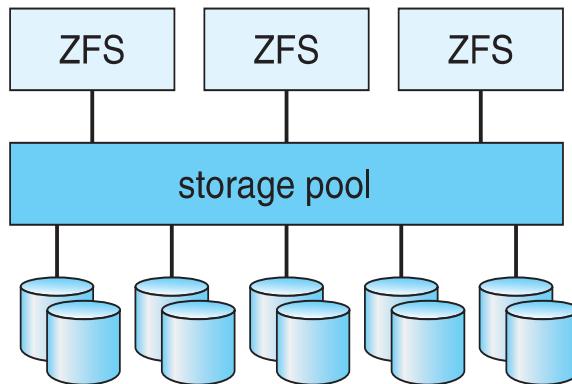


ZFS checksums all metadata and data

# Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

# Object Storage

- General-purpose computing, file systems not sufficient for very large scale
- Another approach – start with a storage pool and place objects in it
  - Object just a container of **data**
  - No way to navigate the pool to find objects (no directory structures, few services)
  - Computer-oriented, not user-oriented
- Typical sequence
  - Create an object within the pool, receive an object ID
  - Access object via that ID
  - Delete object via that ID

# Object Storage (Cont.)

---

- Object storage management software like **Hadoop file system (HDFS)** and **Ceph** determine where to store objects, manages protection
  - Typically by storing N copies, across N systems, in the object storage cluster
  - **Horizontally scalable**
  - **Content addressable, unstructured**

# References

---

- *Silberschatz, Abraham and Galvin, Peter B. and Gagne, Greg, Operating system Concepts, 9<sup>th</sup> Edition*

**Thank You**