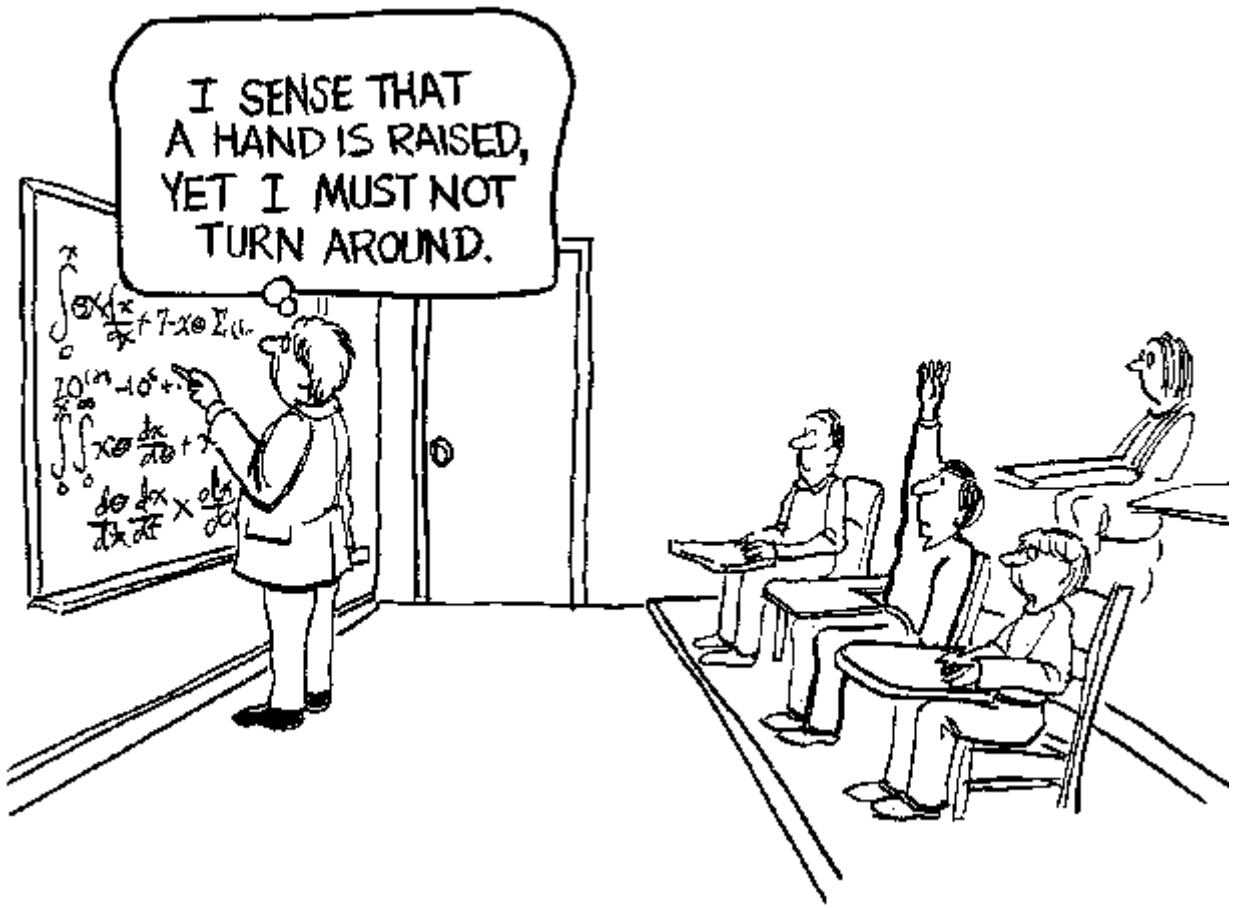
infoway
TECHNOLOGIES

Operating System Concepts

Operating Systems and Linux

Ask Questions…



Course Contents

- Introduction to Operating Systems
- History of Operating Systems
- Operating System Components
- Operating System Services
- Process Management
- Multiprogramming, threading, tasking and processing
- CPU Scheduling

Course Contents

(Contd…)

- Process Synchronization
- Deadlocks
- Memory Management
- File System
- Introduction to Network and Distributed Operating Systems
- Security and Protection

Course Contents

(Contd…)

- Introduction to Linux
- Basic Commands
- Utilities
- Linux Shell
- GUIs
- Vi Editor

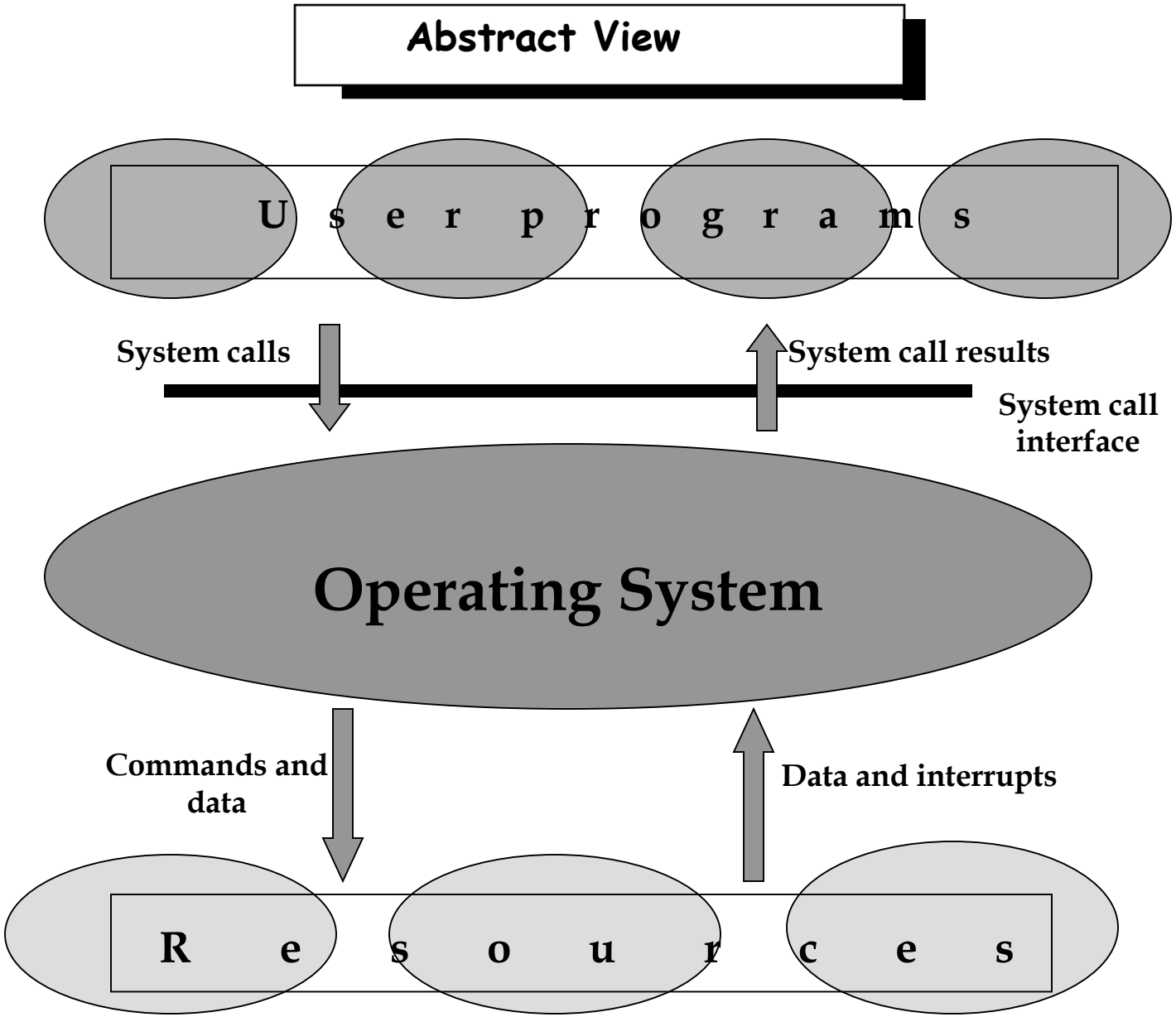
Course Contents

(Contd…)

- Emacs Editor
- Bourne Again Shell
- Networking
- Shell Programming
- Advanced Shell Programming
- Programming Tools

Introduction to Operating System

- An Operating System is a program that acts as an interface between user of the computer and the computer hardware.
- Controller of all computer resources.
- It is program without which no computer will work.


Abstract View
User programs
System calls
System call results
System call interface
Operating System
Commands and data
Data and interrupts
Resources

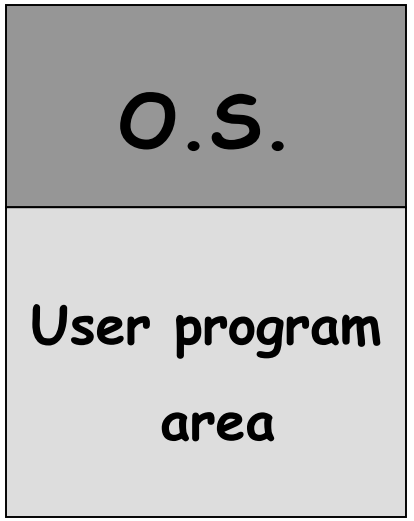
History of Operating System in Brief

- The 1st Generation (1945-1955): Vacuum Tubes and Plugboards.
- The 2nd Generation (1955-1965): Transistors and Batch Systems.
- The 3rd Generation (1965-1980): ICs and Multiprogramming.
- The 4th Generation (1980-1990): Personal Computers
- The latest Operating Systems (Network, Distributed, etc.).

Types of Operating Systems

- **Simple Batch Systems:**
  - No user interaction with the system.
  - Jobs with similar needs were grouped into Batches.
  - Reads a stream of jobs, operates on it, generates output.
  - First Come, First Served.
  - Can use **Spooling**.

Memory layout of a Batch system

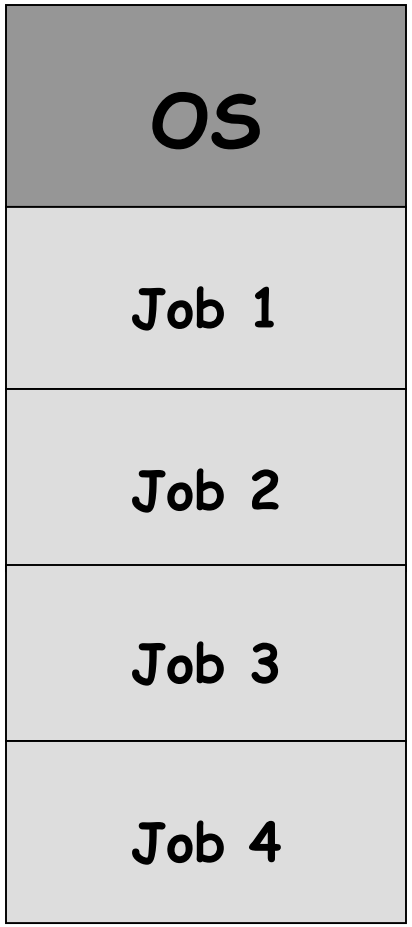


Types of Operating Systems (Contd…)

- **Multi-programmed Batched Systems:**
  - Spooling results in several jobs waiting to get processed.
  - If several jobs are available on Direct Access device, **Job Scheduling** becomes possible.
  - An important aspect of Job Scheduling is ability to **MULTIPROGRAM.**
  - Multiprogramming **increases** CPU utilization.
  - When a job needs to wait, CPU switches to another job.
  - CPU never sits idle.

Types of Operating Systems

(Contd…)



Memory layout of a Multiprogramming system

Types of Operating Systems (Contd…)

- **Time Sharing Systems:**
  - Also called as Multi-tasking systems.
  - Logical extension of Multiprogramming systems.
  - Multiple jobs are executed by CPU switching between them.
  - Switching is very frequent, so that users can interact with programs while it is running.
  - An interactive or Hands-on computer system.
  - Allows many users to share the computer resources simultaneously.

Types of Operating Systems (Contd…)

- **Personal Computer Systems:**
  - System dedicated to a single user.
  - System provides maximum user convenience and responsiveness.
  - CPU utilization is NOT the prime concern.

Types of Operating Systems (Contd…)

- **Parallel Systems:**
  - More than 1 processor. Multiprocessing.
  - Processors share the computer bus, the clock, memory and peripheral devices.
  - Referred to as Tightly Coupled systems.
  - Increased Throughput.
  - Speed-up ratio with "n" processors is somewhat less than "n".

Types of Operating Systems (Contd…)

- Graceful degradation.
  - Ability to provide service proportional to the level of surviving hardware.
- Fault tolerant.
- Common Multiprocessing systems are:
  - **Symmetric Multiprocessing:**
    - Two or more similar processors connected via a high-bandwidth link and managed by one operating system, where each processor has equal access to I/O devices.
  - **Asymmetric Multiprocessing:**
    - Each processor is assigned a specific task.
    - A master processor controls the system, and allocates work to slave processors.
    - Other processors may have predefined work.

Types of Operating Systems (Contd…)

- **Distributed Systems:**
  - Processors DO NOT share the memory or the clock.
  - Each processor has it's own memory.
  - Loosely coupled systems.
  - Gives the impression that there is a single OS controlling the network.

Types of Operating Systems (Contd…)

- **Network Operating Systems:**
  - Provides file sharing.
  - Provides communication scheme.
  - Runs independently from other computers on the network.

Types of Operating Systems (Contd…)

- **Real Time Systems:**
  - Used when processing must be done within the fixed time constraints.
  - Considered to work correctly, only if it returns correct results with given time constraints.
  - **Examples:** process control in industrial plants, robotics, air traffic control, telecommunications, military command and control systems, etc.

Types of Operating Systems (Contd…)

Two types of RTOS:

- **Hard Real-Time Systems:**
  - Secondary storage limited or absent, data stored in short-term memory, or read-only memory (ROM).
  - Not supported by general-purpose operating systems.
- **Soft Real-Time Systems:**
  - Limited utility in industrial control or robotics
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

Operating System Components

- **Process Management:**
  - Creation and deletion of user processes.
  - Creation and deletion of system processes.
  - Suspension and resumption of processes.
  - Mechanisms for process synchronization.
  - Mechanisms for process communication.
  - Mechanisms for deadlock handling.

- **Memory Management:**
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes are to be loaded into memory when memory space becomes available.
  - Allocate and deallocate memory space as needed.

Operating System Components

(Contd…)

- **Secondary Storage Management:**
  - Free space management.
  - Storage allocation.
  - Disk scheduling.

Operating System Components

(Contd…)

- **I/O System:**

  consists of:

  – A buffer caching system.

  – A general device driver code.

  – Drivers for specific hardware devices.

- **File Management:**
  - Creation and deletion of files.
  - Creation and deletion of directories.
  - Support of primitives for manipulating files and directories.
  - Mapping of files onto disk storage.
  - Backup of files on stable (non volatile) storage.

Operating System Components (Contd…)

- **Protection System:**
  - Memory addressing hardware.
  - Timers.
  - Users are not allowed to do their own I/O.

- **Networking:**
  - Access to shared resources across the network.
  - Communication protocols like TCP/IP and UDP/IP.

Operating System Components

(Contd…)

- **Command Interpreter System:**
  - One of the most important system programs for an O.S.
  - Called as ***shell*** in UNIX.

Operating System Services

- **Program execution**
  - System must be able to load the program into the memory and to run it.
- **I/O Operations**
  - A running program may require I/O (file of an I/O device).
  - System must provide means to control I/O.
- **File System Manipulation**
  - System must provide means to create, delete, modify files.

Operating System Services (Contd…)

- **Communications**
  - One process needs to exchange information with other process.
  - System must provide means of communications between processes.
    - Processes on the same computer.
      - Shared memory.
    - Processes on different computers.
      - Message Passing, networking.

- **Error Detection**
  - System must be able to handles error occurring at different areas like: CPU, Memory, I/O devices, Network etc…

Operating System Services (Contd…)

- **Resource Allocation**
  - CPU cycles, memory, file storage etc…
  - Availability of resources to multiple users and multiple jobs.
- **Accounting**
  - WHO is using WHAT???
- **Protection**
  - Controlled access to the resources.
  - Security between different users, processes, i/o devices etc…
  - Memory privileges.

Operating System Services

- **Networking Support**
- **User Interface**
  - Command shell, GUI.
  - System Calls.

Processes

Presentation Outline

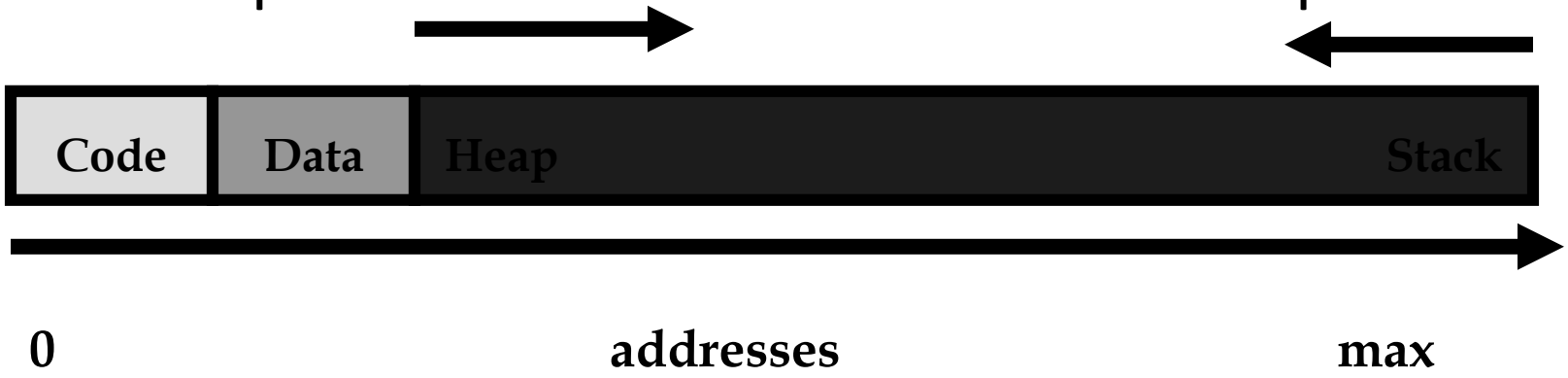
- The Process
- Process Address Space
- Process Life Cycle
- Scheduling
- Process Control Block (PCB)
- Schedulers
- Threads
- CPU Scheduling
- Criteria for Scheduling
- Pre-emptive and Non-pre-emptive Scheduling
- Scheduling Algorithms

The Process

- **What is a Process?**
  - A CPU activity.
  - A program under execution.
  - More than a program code!!! It includes:
    - Current Activity, i.e. **The Program Counter & Register Contents**.
    - A Stack, **that contains temporary data, subroutine parameters, return address, temporary variables, etc…**
    - A Data Section, **that contains global variables.**
  - All above forms the Process Space.

Process Address Space

- Representation of a Process Address Space.



Code	**Executable code of program.**
Data	**Global variables.**
Heap	**Runtime allocation of memory.**
Stack	**Temporary variables.**

Process Life Cycle

- User initiates a program.
- The OS allocates a process to represent the execution of this program.
- For this, it:

  Creates a **Process Control Block** (PCB) to manage the process during its lifetime.

  Each process is represented in OS by a PCB, also called Task Control Block.

  - Allocates resources to the new process i.e. memory, I/O devices, secondary storage, etc.

Process Life Cycle

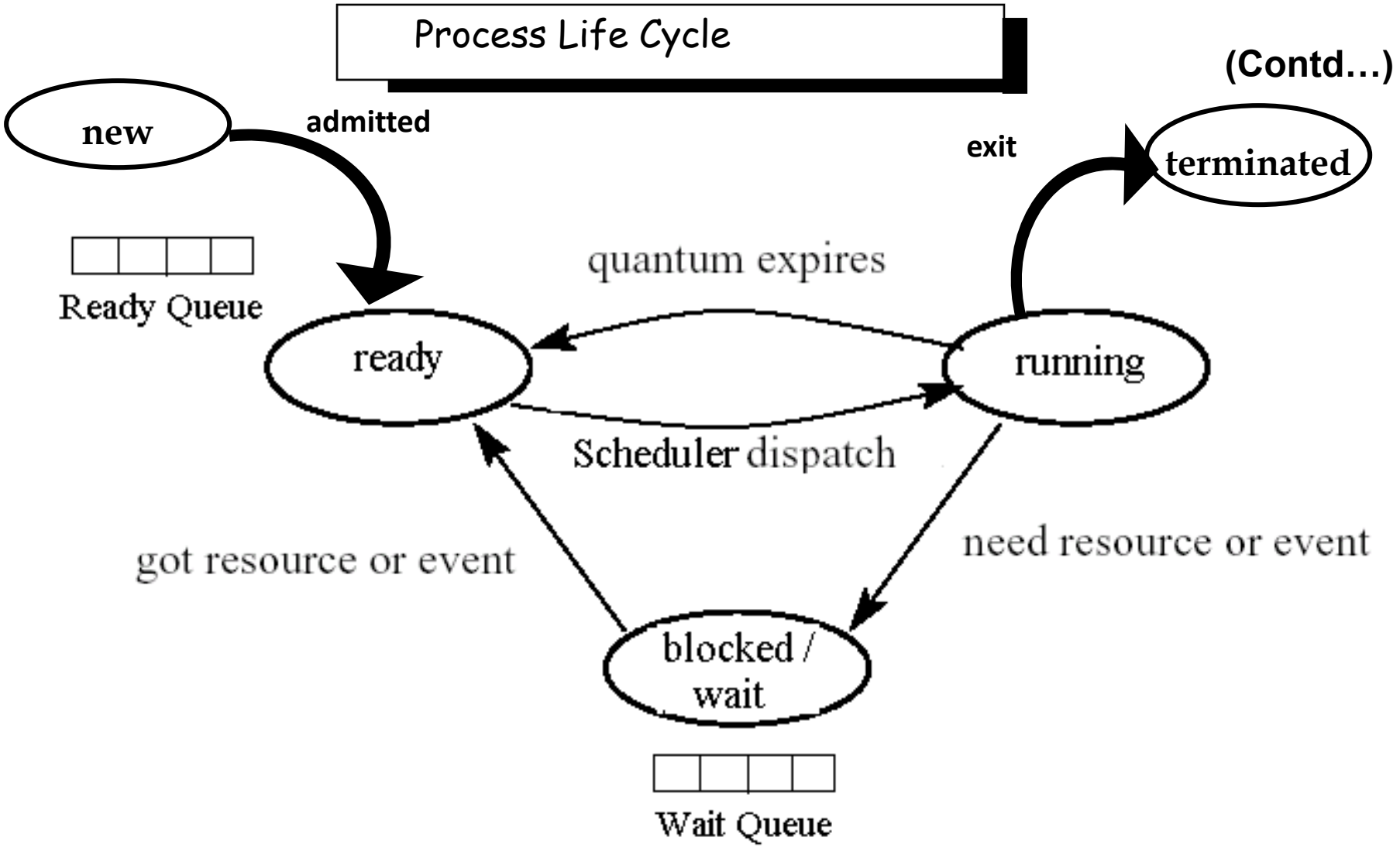
(Contd…)

- Process States:

  As the process executes, it changes states.

New	The process is being created.
Running	Instructions are being executed.
Waiting	Process is waiting for some events to occur.
Ready	Waiting to be assigned to a processor.
Terminated	Process has finished the execution.

- Only one process can be running on any processor at any instant.
- Many processes may be waiting or ready for execution.



Simple Process State Transition Diagram

Process Life Cycle

(Contd…)

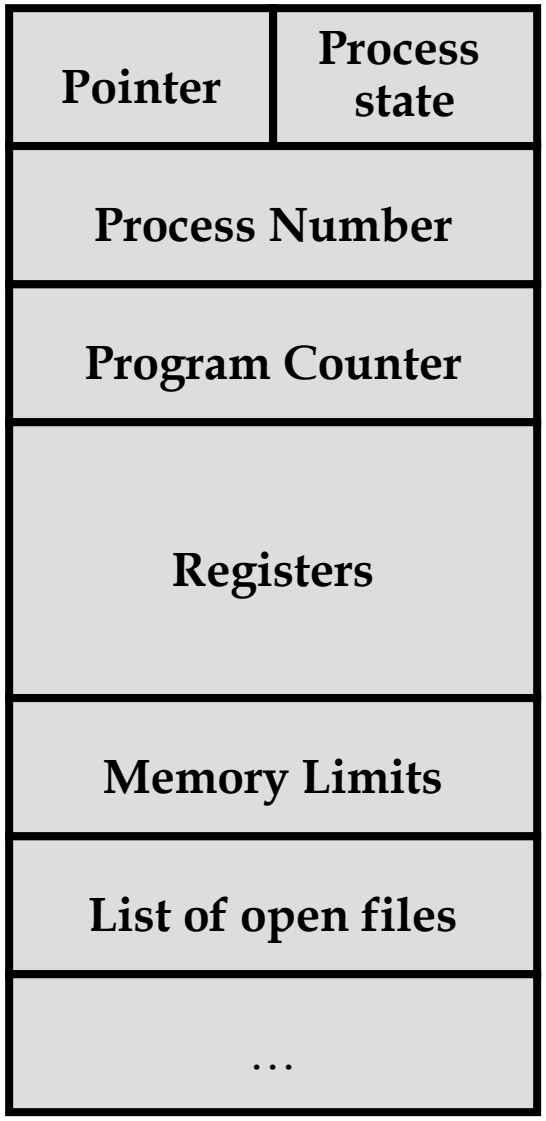
- **When does a process go into the 'wait' state ?**
  - When it waits for an event. e.g.
    - for data from a (relatively) slow disk
    - for input from a keyboard
    - for another thread to leave a critical section
    - for busy device to become idle

- The OS scheduler only allocates the processor to processes that are not blocked, since blocked processes have nothing to do while they are blocked.

Scheduling

- **Scheduling**: The assignment of CPU to a process.
- **Preemptions**: Removal of CPU control from a process.
- **Context Switch**: Switching the CPU to another process.
  - Save the state of old process.
  - Load the saved state of new process.
- **PCB** is used to save and load the information/context of a process.

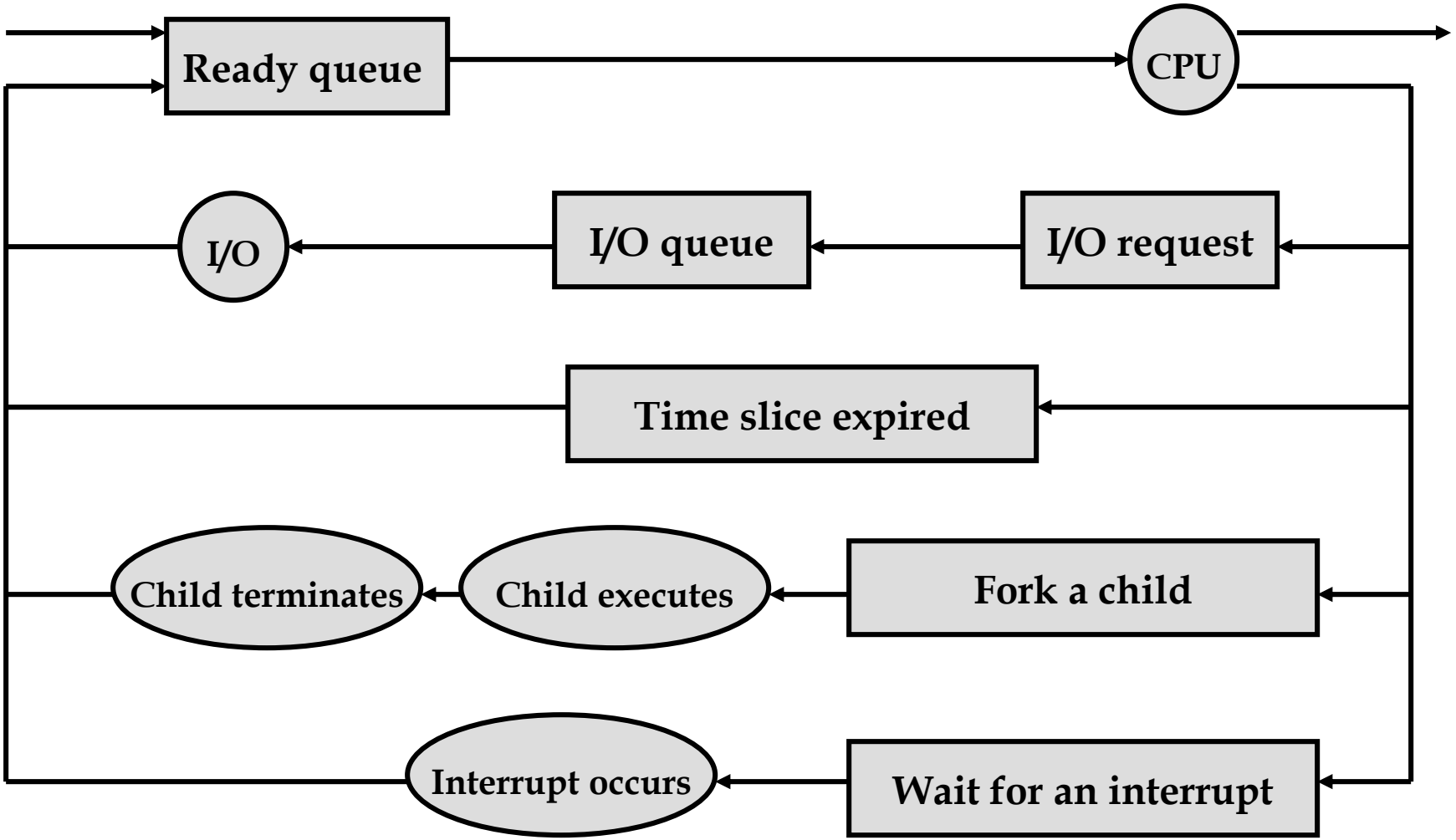
What is a PCB?



- **Process State**
  - New, ready, waiting, running, halted…
- **Program Counter**
  - Address of next instruction to be executed.
- **CPU Registers**
  - Accumulators, index-register, stack-pointer, general purpose registers…
- **Memory management info**
  - Values of base and limit registers, page tables, segment tables…
- **Accounting info**
  - Amount of CPU time used, time limits…
- **I/O status info**
  - List of I/O devices allocated to the process, open files…

Schedulers

Queuing-diagram of Process Scheduling



Threads

- Multithreading is the ability of an operating system to execute different parts of a program, called **Threads**, simultaneously.
- What's the difference between a thread and a process ?
  - Threads belong to the same process, also called as **Light Weight Process**.

Share	Don't share
Code Section Data Section OS resources – global memory, files, signals etc.	Program counter (PC) Stack State Registers

CPU Scheduling

- **Objectives:**
  - Maximize **CPU Utilization**
  - Maximize System **Throughput**
    - Number of processes completed in unit time.
  - Minimize **Turnaround Time**
    - Time measured from process creation to termination.
  - Minimize **Waiting Time**
    - Amount of time a process has been waiting in the ready queue.
  - Minimize **Response Time**
    - Time measured from process creation to the time of first output.
  - Ensure **Fairness (No starvation)**

Criteria For Scheduling

- The criteria:
  - The type of process – real time, normal and its Priority.
    - What if the priorities of two processes are equal?
  - Resource requirements.
  - CPU time already consumed.
  - Wait time.
- A running process can be modeled as alternating series of **CPU bursts** and **I/O bursts**
  - During a CPU burst, a process is executing instructions.
  - During an I/O burst, a process is waiting for an I/O operation to be performed and is not executing instructions.

Pre-emptive and Non-Pre-emptive Scheduling

- The job of the CPU scheduler is to select a process from the ready queue, based on some scheduling policy.
- The policy could be preemptive or non-preemptive.
  - A **Non-preemptive** scheduler runs only when the running process gives up the processor through its own actions, e.g.,
    - **The process terminates.**
    - **The process blocks because of an I/O or synchronization operation.**
    - **The process performs a 'yield' system call.**
  - A **Preemptive** scheduler may, force a running process to stop running.
    - Can bring process from Running State to Ready State.
    - Can bring process from Waiting State to Ready State.

Scheduling Algorithms

- Deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU.
- Different CPU scheduling algorithms exist:
  - First-Come-First-Served (FCFS)
  - Shortest-Job-First (SJF)
  - Priority Scheduling
  - Round-Robin (RR)
  - Multi-Level Queue Scheduling

First-Come-First-Served

- Simplest scheduling algorithm.
- Non-pre-emptive type of scheduling.
- Process which requests the CPU first is allocated the CPU first.
- The implementation of FCFS is managed with a FIFO queue.

First-Come-First-Served (Contd…)

- First Come, First Served (Non-preemptive)
  - Process Burst Time
    - P1 24
    - P2 3
    - P3 3
  - Suppose that the processes arrive in the order: P1 , P2 , P3
    The Gantt Chart for the schedule is:

Waiting time for P1 = 0, P2 = 24, P3 = 27
Average waiting time = (0 + 24 + 27) / 3 = 17 ms.

First-Come-First-Served

(Contd…)

- Suppose that the processes arrive in the order

  P2 , P3 , P1 .

  – 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 ms.
Much better than previous case.
CONVOY EFFECT: Short process behind long process

Shortest-Job-First (SJF)

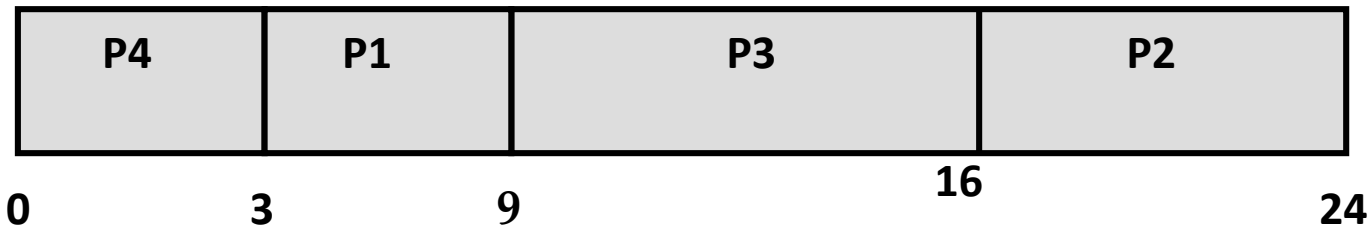
- Shortest Job First
  - Associates with each process the length of its **next CPU burst**.
  - Use these lengths to schedule the process with the shortest time.
  - If two processes have the same **next CPU burst**, **FCFS** is used.
- Two schemes:
  - **Non-preemptive**.
    - Once a CPU is given to the process it cannot be preempted until it completes its CPU burst.
  - **Preemptive** ***(Shortest-Remaining-Time-First)***.
    - If a new process arrives with CPU burst length less than remaining time of current executing process, then preempt it.
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Shortest-Job-First (SJF) (Contd…)

- **Example:**

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

  - The Gantt chart for the schedule is:



Waiting time for P4 = 0, P1 = 3, P3 = 9, P2 = 16
Average waiting time = (0 + 3 + 9 + 16) / 4 = 7.

Priority Scheduling

- Shortest-Job-First (SJF) is a special case of the general ***priority*** scheduling algorithm.
- A priority number (integer) is associated with each process.
  - The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
    - Preemptive.
    - Non-preemptive.
  - Problem: **Starvation**.
    - Low priority processes may never execute.
  - Solution: ***Aging***.
    - As time progresses increase the priority of the process.
- **What if priorities are same?**

Round-Robin (RR)

- Round-Robin (RR)
  - Preemptive version of FCFS.
  - Running process is preempted after a fixed time quantum, if it has not already blocked.
  - Preempted process goes to the end of ready queue.
  - Especially designed for time-sharing systems.

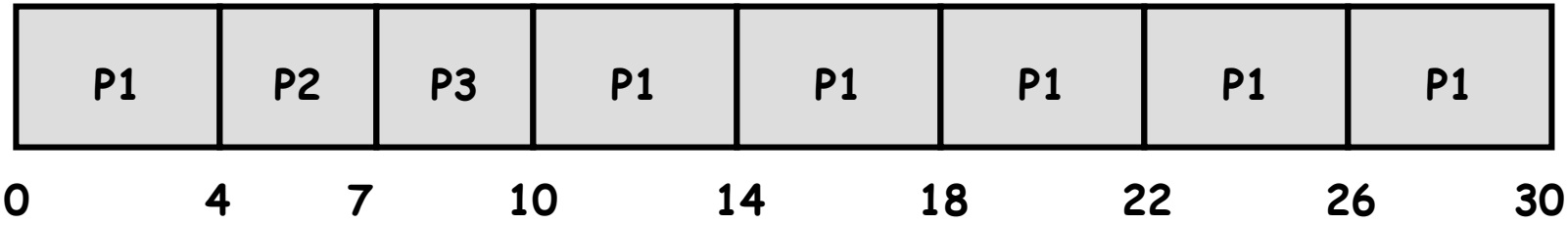
Round-Robin (RR)

(Contd…)

- **Example:**

Process	Burst Time
P1	24
P2	3
P3	3

Let time quantum = 4 ms.

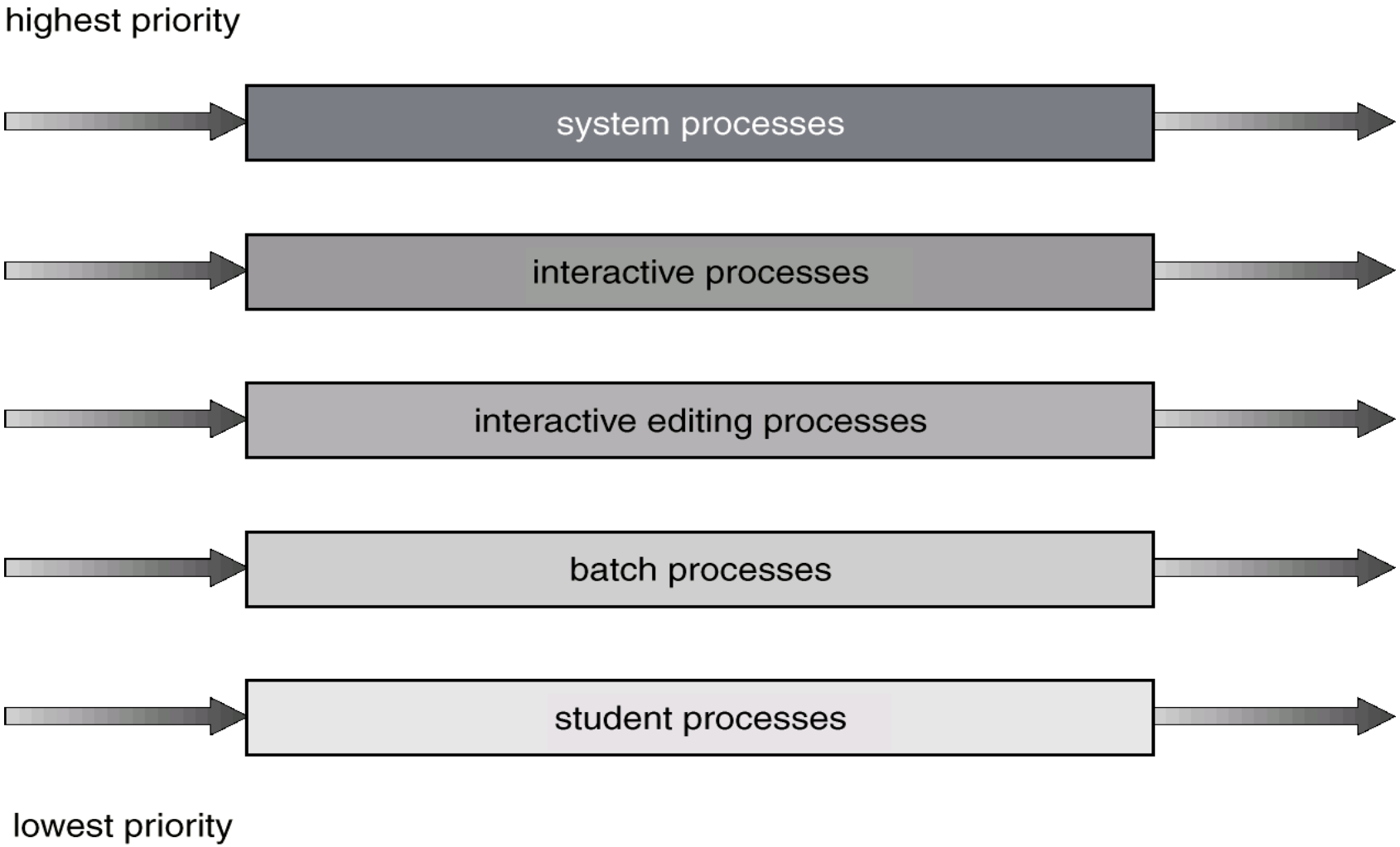


Waiting time for P1= 6, P2 = 4, P3 = 7
Average Waiting time = (6 + 4 + 7) / 3 = 5.66 ms.
Typically, higher average turnaround than SJF, but better response.

Multi-Level Queue Scheduling

- Ready queue is partitioned into separate queues
  - Foreground (interactive).
  - Background (batch).
- Each queue has its own scheduling algorithm.
  - Foreground – RR.
  - Background – FCFS.
- Scheduling must be done between the queues.
  - Fixed priority scheduling; i.e., serve all from foreground then from background. Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time
    - 80% to foreground in RR.
    - 20% to background in FCFS.

Multi-Level Queue Scheduling (Contd…)

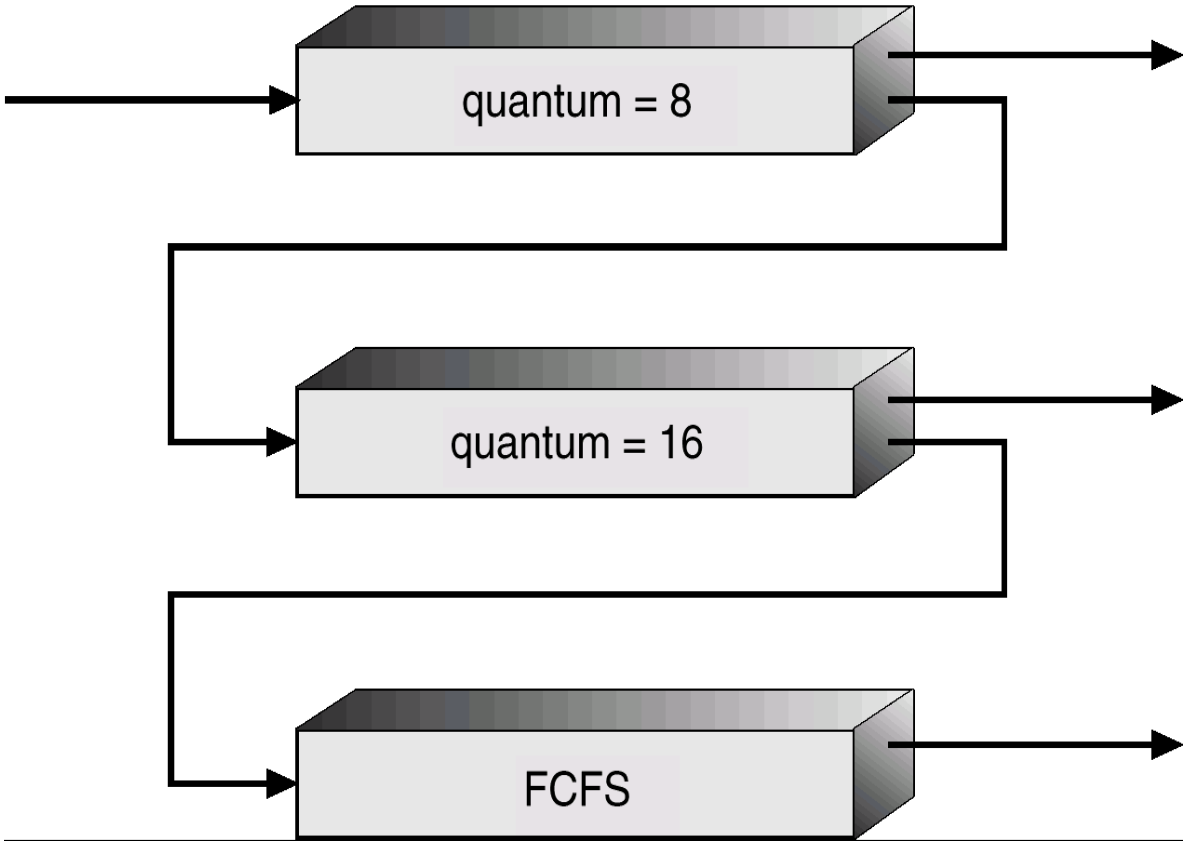


Multi-Level Feedback Queues

- A process can move between the various queues; aging can be implemented this way.

- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multi-Level Feedback Queues (Contd…)



Multiple Processor Scheduling

- Complex form of scheduling.
- Processors can be identical **(a homogeneous system)** or different **(a heterogeneous system).**
- Load sharing can occur for identical processors.

Process Synchronization

Presentation Outline

- Concurrency
- Synchronization
- Mutual Exclusion with Busy Waiting
  - Disable Interrupts
  - Lock Variables
  - The TSL Instruction
- Sleep and Wakeup
- The Producer-Consumer problem
- Semaphores
- Language Constructs
  - Monitors
  - Critical Region
- Classical IPC Problems

Concurrency

- Concurrent access to shared data can lead to data inconsistency.
- On multiprocessors, several processes can execute simultaneously, one on each processor.
- On uni-processors, only one process executes at a time.
  - Because of preemption and timesharing, processes appear to run concurrently.
- Concurrent processes may need to communicate.
  - Often through shared data structures.
  - This may lead to race conditions.

◆ Therefore the need for synchronization.

Synchronization

- **Critical section**
  - The part of a program in which the shared object is accessed is called a Critical Section.
- **Race condition**
  - A situation where several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on the sequence in which access takes place.
- Only one process at a time must manipulate the shared data.
  - For this, some form of **Synchronization** must be provided.
- A common synchronization problem is to enforce **Mutual Exclusion**.
  - Making sure that only one process at a time uses a shared object, e.g., a variable or a device.

Mutual Exclusion with Busy Waiting

- Different proposals for achieving mutual exclusion:
  - Disable Interrupts
  - Lock Variables
  - The TSL Instruction

Disable Interrupts

- Simplest solution.
- Each process disables all interrupts just after entering its critical section and re-enables them just before leaving it.
- **Unattractive approach:** gives user processes the power to turn off interrupts.

Lock Variables

- Software solution.
- A single, shared, (lock) variable is initially set to 0.
- First test the lock. If it is 0, the process sets it to 1 and enters the critical section.
- If the lock if already 1, the process just waits until it becomes 0.
- **Mutex lock primitives:**

```
mutex_lock(device_register_INTR_receive)
{
        critical section…
}
mutex_unlock(device_register_INTR_receive)
```

The TSL Instruction

- Hardware solution.
- **TEST AND SET LOCK (TSL)** instruction is used.
- Setting and clearing locks using TSL:

```
enter_region:
   tsl   register, flag    // copy flag to register and set flag to 1
   cmp register, #0        // was flag zero?
   jnz enter_region        // if it was non zero, lock was set, so loop
   ret                     // return to caller; critical section entered

leave_region:
   mov flag, #0            // store a 0 in flag
   ret                     // return to caller
```

Sleep and Wakeup

- Previous solutions require busy-waiting.
- Busy-waiting wastes precious CPU time.
- Processes can be blocked instead of wasting CPU time.
- Use SLEEP and WAKEUP pair.
- SLEEP causes the caller to block, i.e., be suspended until another process wakes it up.
- WAKEUP call has one parameter: the process that has to be awakened.
- Race conditions can definitely occur with this approach.

The Producer-Consumer Problem

- Also known as **bounded-buffer** problem.
- Two processes share a common, fixed-size buffer.
- The **producer** process puts information into the buffer.
- The **consumer** process retrieves this information.

The Producer-Consumer Problem

(Contd…)

```
#include "prototypes.h"
#define N 100     /* number of slots in the buffer */
int count = 0;      /* number of items in the buffer */

void producer(void)
{
  int item;
  while (TRUE)  {                    /* repeat forever */
        produce_item(&item);      /* generate next item */
        if (count == N)  sleep();   /* if buffer is full, go to sleep */
        enter_item(item);          /* put item in buffer */
        count = count + 1;          /* increment count of items in buffer */
        if (count == 1)  wakeup(consumer)            /* was buffer empty? */
  }
}
```

The Producer-Consumer Problem

(Contd…)

```
void consumer(void)
{
  int item;
  while (TRUE)   {                    /* repeat forever */
        if (count == 0)  sleep();     /* if buffer is empty, go to sleep */
         remove_item(&item);          /* take item out of buffer*/
         count = count - 1;           /* decrement count of items in buffer */
         if (count == N-1)  wakeup(producer);        /* was buffer full? */
         consume_item(item);          /* print item  */
  }
}
```

Semaphores

- Introduced by E.W. Dijsktra
- A semaphore is an object that has an integer value, and that support two main operations:
  - **P / Wait / Down** : If the semaphore value is non-zero, decrement the value. Otherwise, wait until the value is non-zero and then decrement it.

```
wait (S) {
        while S <= 0
                ; // no-op
        S--;
}
```

  - **V / Signal / Up** : Increment the value of the semaphore

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

Semaphores

(Contd…)

- Two kinds of semaphores:
  - Counting semaphores: can take on any non-negative value
  - Binary semaphores: take on only the values 0 and 1.
- By definition, the P and V operations of a semaphore are **Atomic.**
- Solving the **producer-consumer** problem using semaphores.

Language Constructs

- Time-dependent errors can occur if processes can share variables in an arbitrary manner.
- Several language constructs have been introduced:
  - Critical Regions
  - Monitors

Critical Regions

- A variable v of type T, which is to be shared among many processes, can be declared:

  var v: **shared** T;

- The variable **v** can be accessed only inside a **region** statement of the following form:

  region v **do** S;

- This construct means that while statement **S** is being executed, no other process can access the variable **v**.

Critical Regions

(Contd…)

- This construct guards against some simple errors associated with the semaphore solution to the critical section problem that may be made by a programmer.
- It does not necessarily eliminate time-dependent errors, but rather reduces the number of them.

Monitors

- A desire to provide the appropriate synchronization automatically has led to the development of a new language construct, called **monitor**.
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A mechanism that allows the safe and effective sharing of abstract data types among several processes.
- Syntax is identical to that of a **class**, except that the keyword **class** is replaced by the keyword **monitor**.
- **Monitors** ensure mutual exclusion; i.e. only one process at a time can be active within the monitor.

Monitors

(Contd…)

- Only one process may be active within the monitor at a time

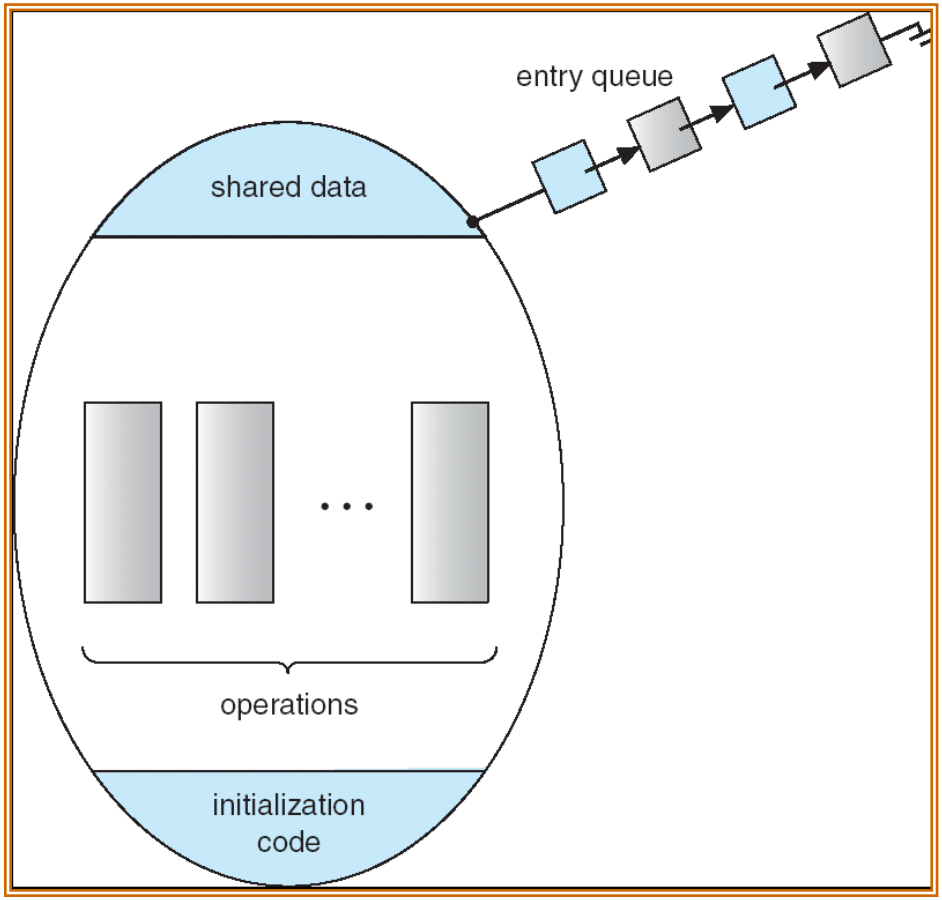
```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

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

Schematic View of a Monitor

(Contd…)



Classical IPC Problems

- Dining Philosophers Problem
- Readers and Writers Problem
- Sleeping Barber Problem
- Bounded Buffer Problem

Deadlocks

Deadlocks

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

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks

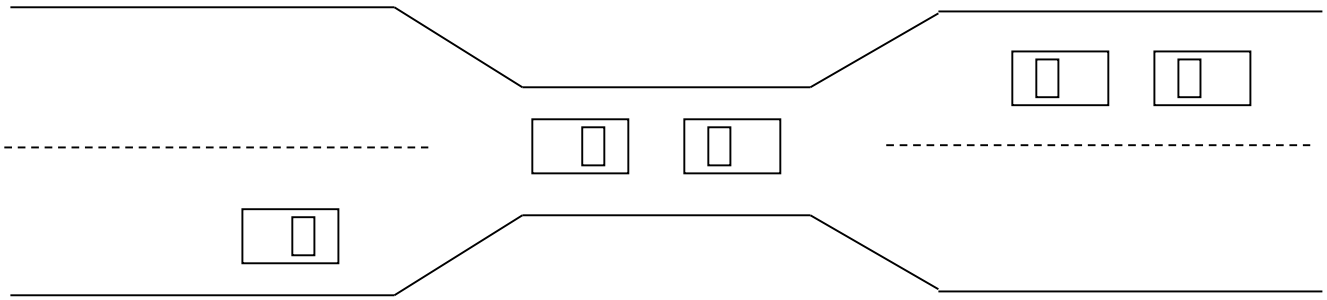
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

The Deadlock Problem

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

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

Bridge Crossing Example



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

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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

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

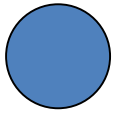
Resource-Allocation Graph

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

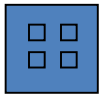
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

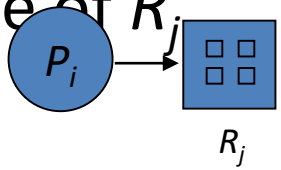
- Process



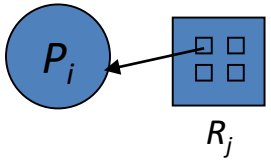
- Resource Type with 4 instances



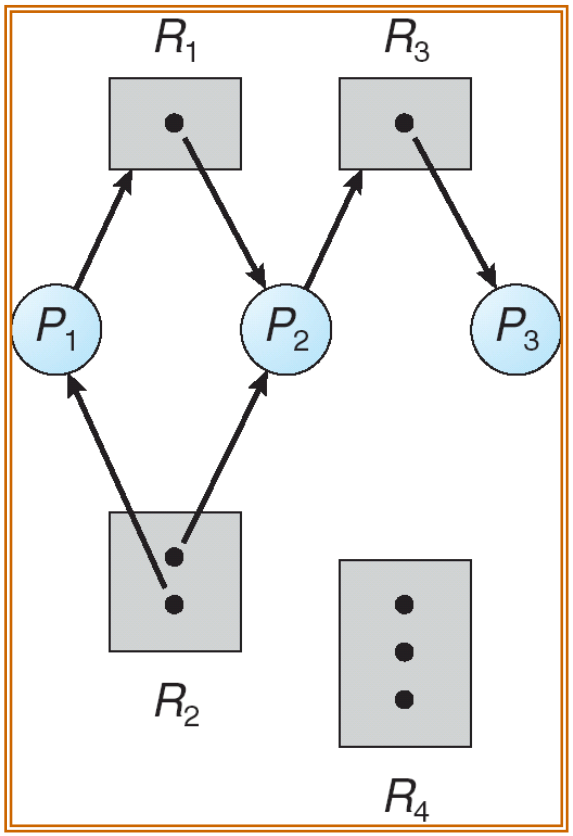
- P_i requests instance of R_j



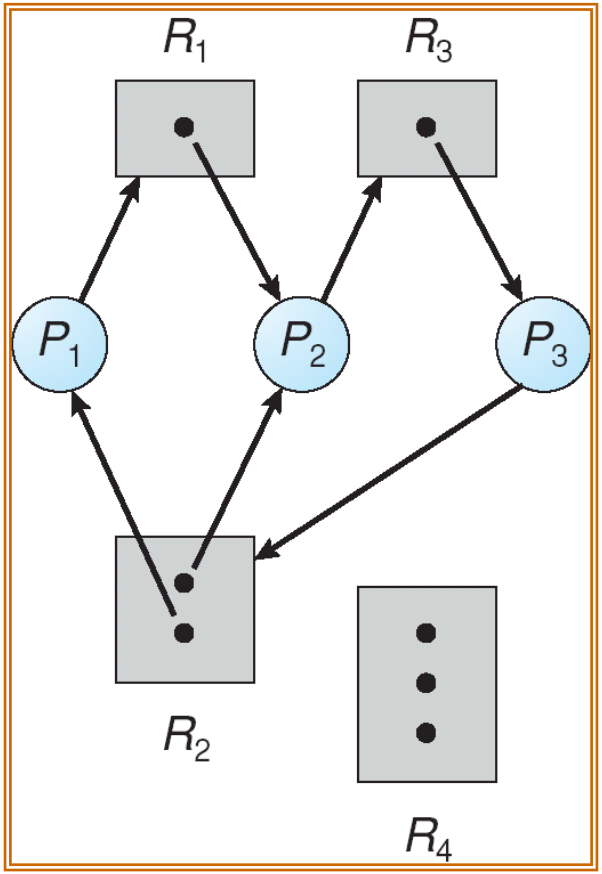
- P_i is holding an instance of R_j



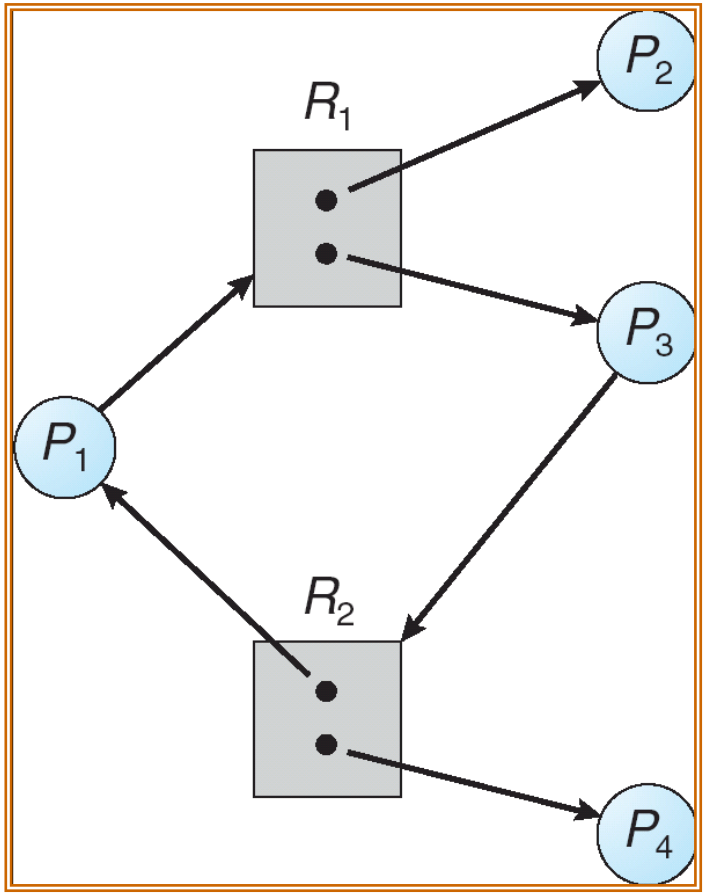
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - 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.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.

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

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

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

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

Safe State

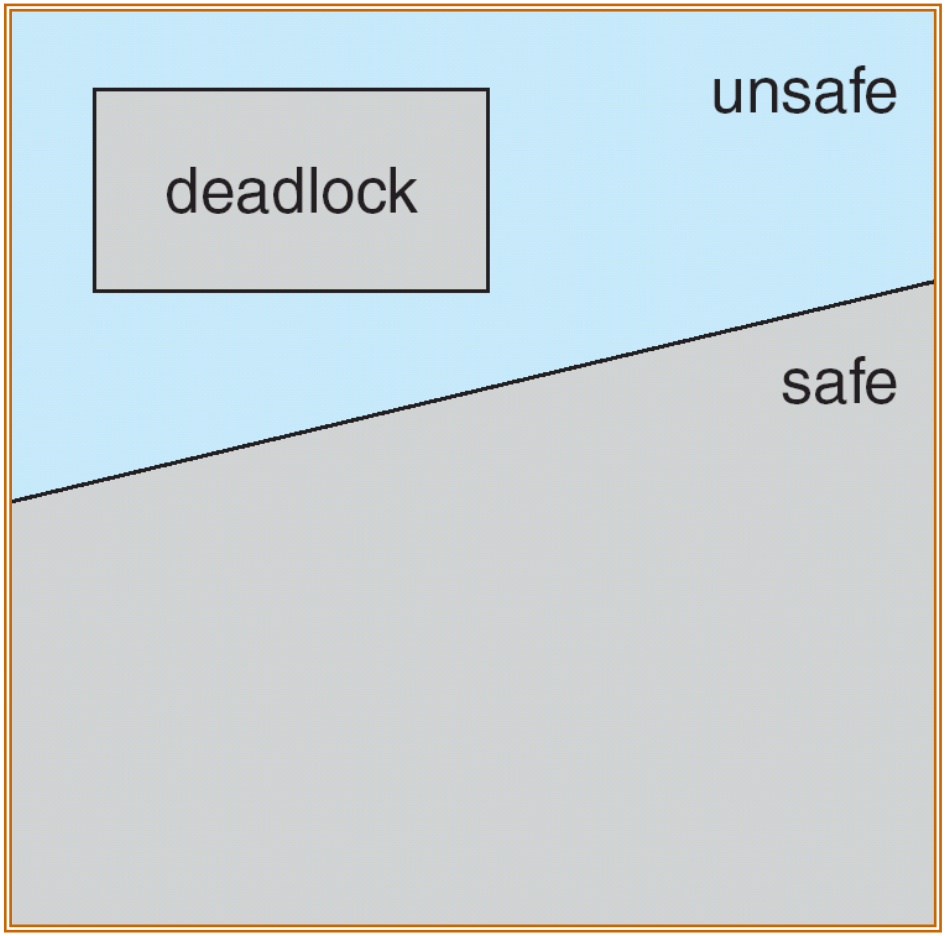
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes is 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 $j < i$.
- 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

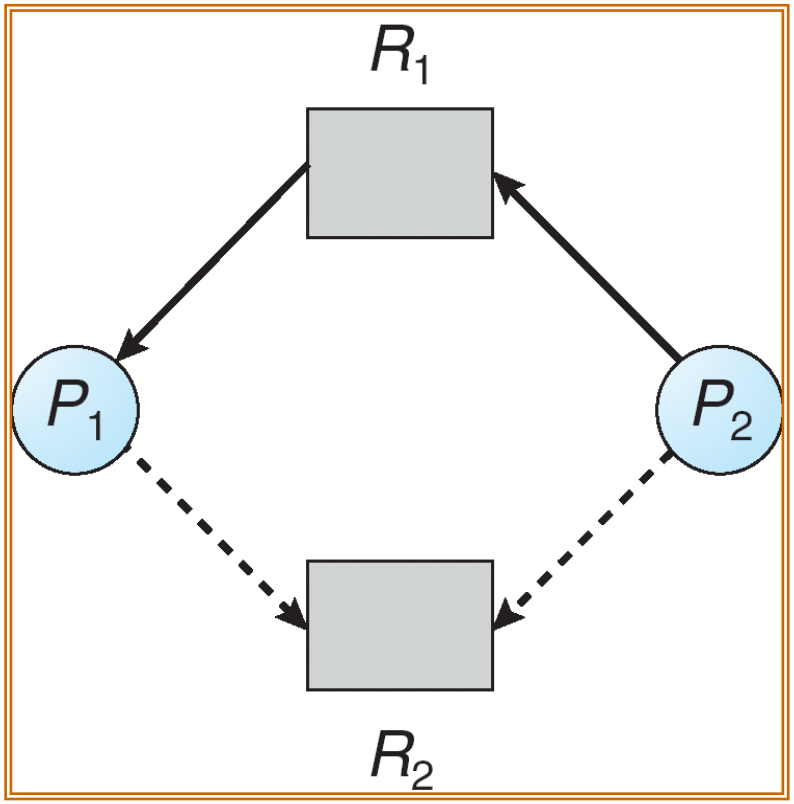
- Single instance of a resource type. Use a resource-allocation graph

- Multiple instances of a resource type. Use the banker's algorithm

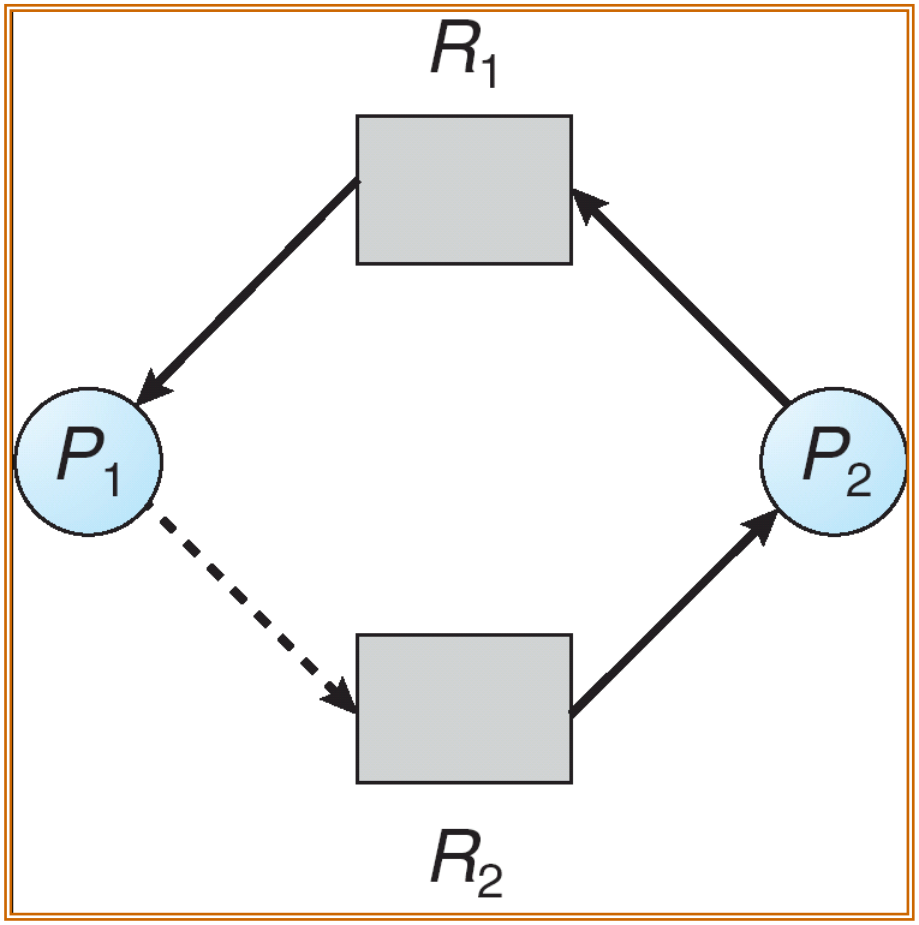
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

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

Banker's Algorithm

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

Data Structures for the Banker's Algorithm

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

- ***Available***: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- ***Max***: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- ***Allocation***: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- ***Need***: n x 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, …, *n*- 1.

2. Find and *i* such that both:

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

   (b) $Need_i \leq Work$

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

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

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

Resource-Request Algorithm for Process P_i

Request = 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.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$

   - *If safe $\Rightarrow$ the resources are allocated to Pi.*
   - *If unsafe $\Rightarrow$ Pi 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* (5instances), and *C* (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

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

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example: P_1 Request (1,0,2)

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

	Allocation	*Need*	*Available*
	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> 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?

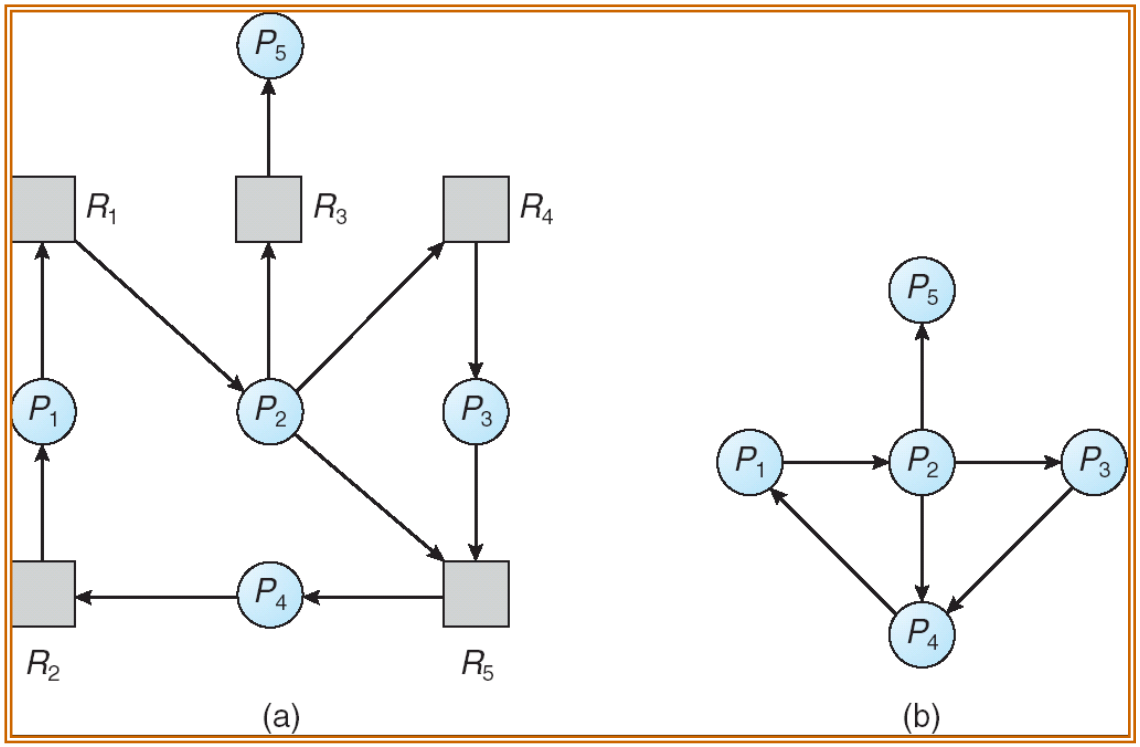
Deadlock Detection

- 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



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* x *m* matrix defines the number of resources of each type currently allocated to each process.
- ***Request**:* An *n* x *m* matrix indicates the current request of each process. If *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, …, *n*, if $Allocation_i \neq 0$, then
   Finish[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

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

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish*[*i*] == 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.

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:

	Allocation	*Request*	*Available*
	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 <P_0, P_2, P_3, P_1, P_4> 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 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

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

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- 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.

Memory Management

Memory Management

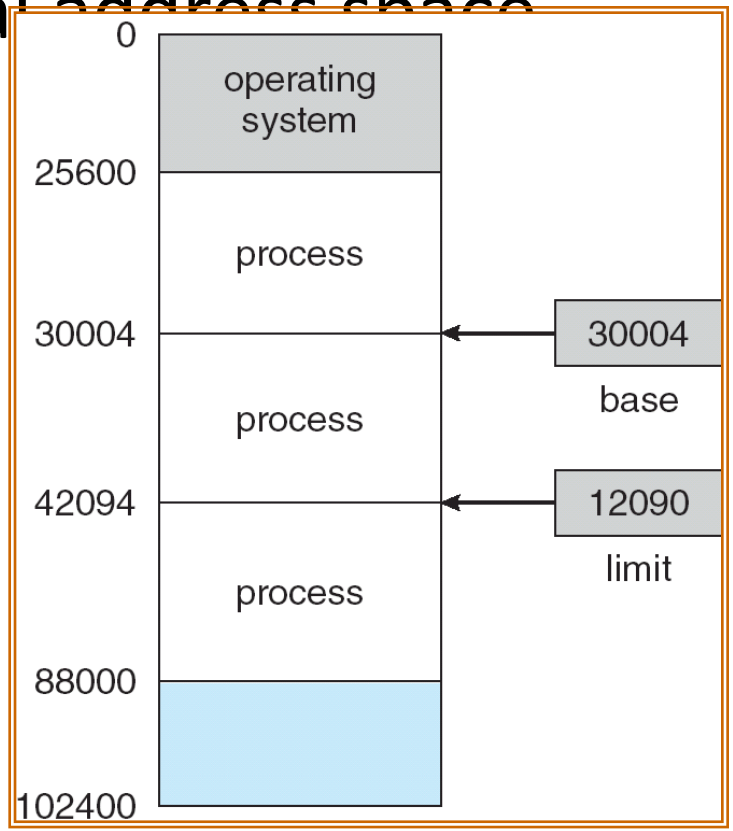
- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Segmentation

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



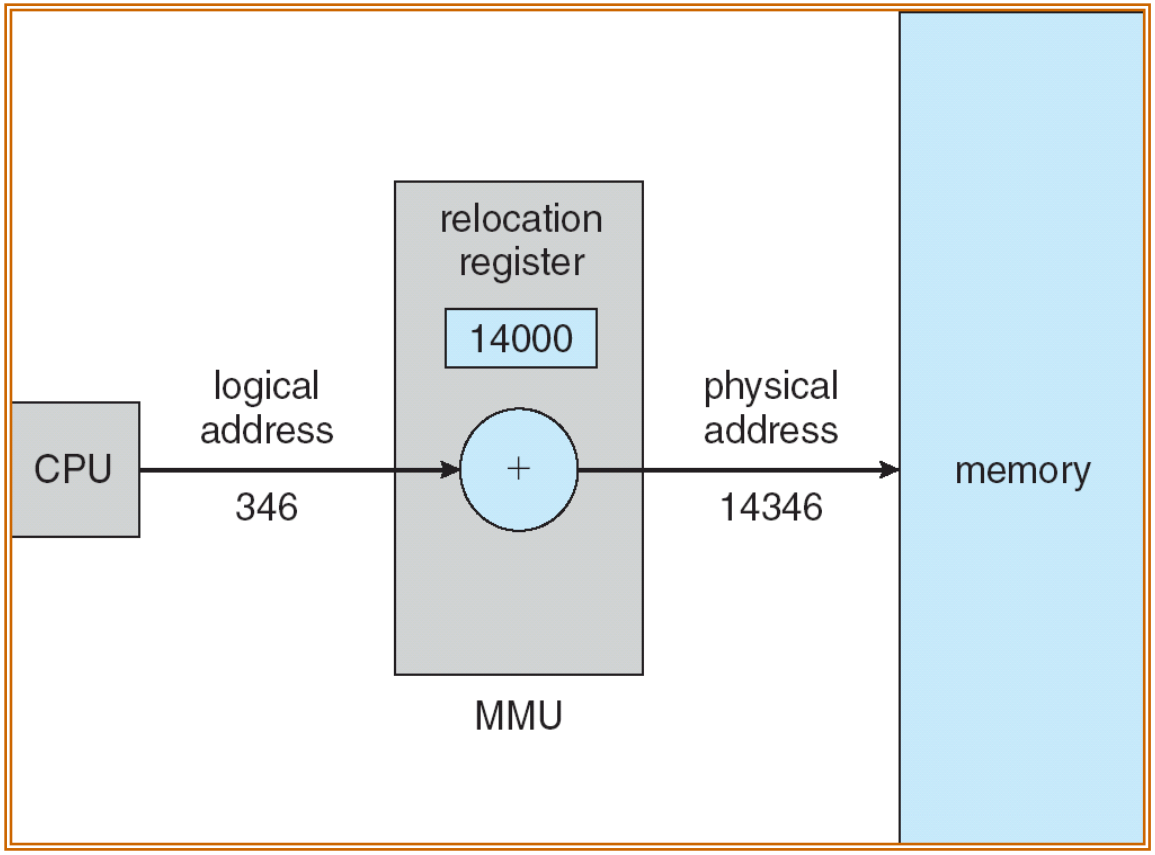
Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit, i.e. the one that is loaded into the **Memory Address Register (MAR)** of the memory
- The set of all logical addresses generated by a program is a **logical address space**; the set of all physical addresses corresponding to these logical addresses is a **physical address space**

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register (base 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

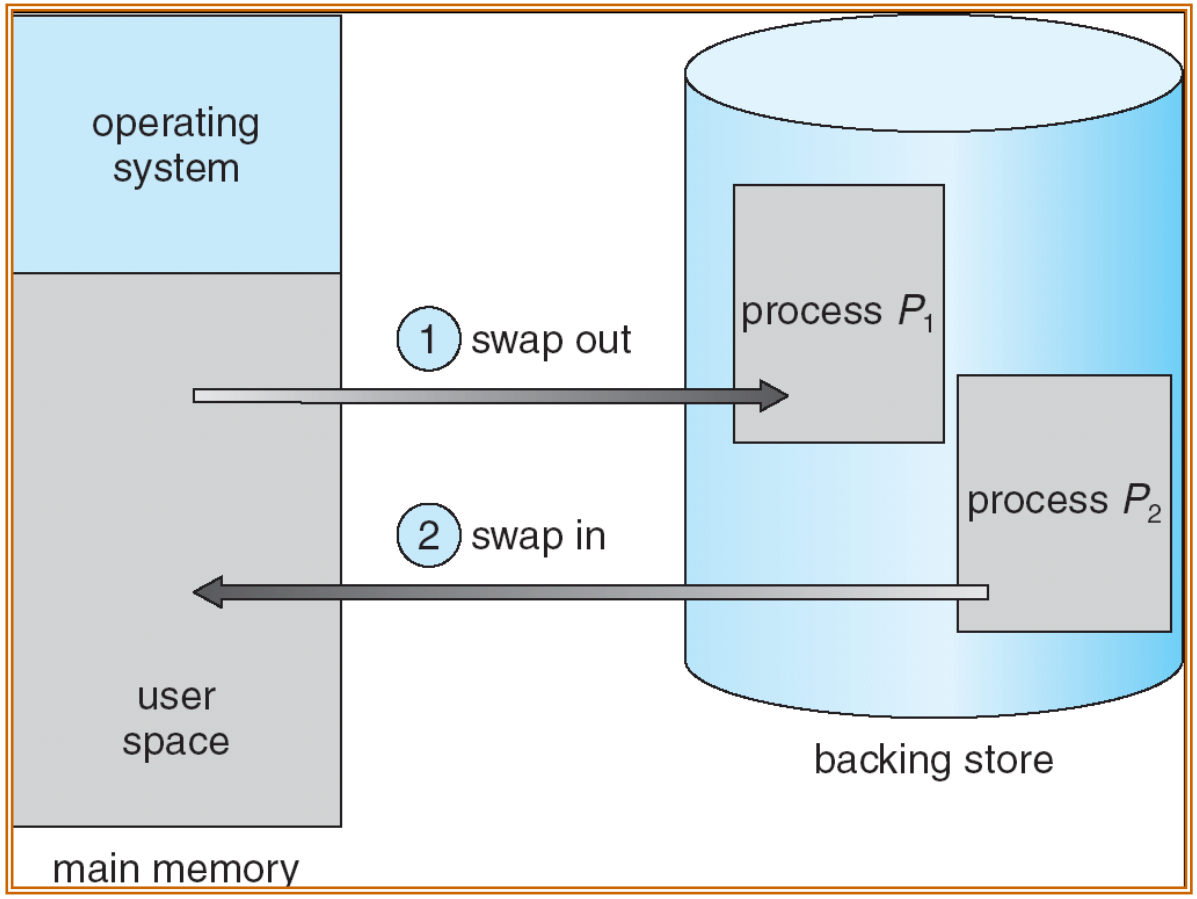


- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management

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)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

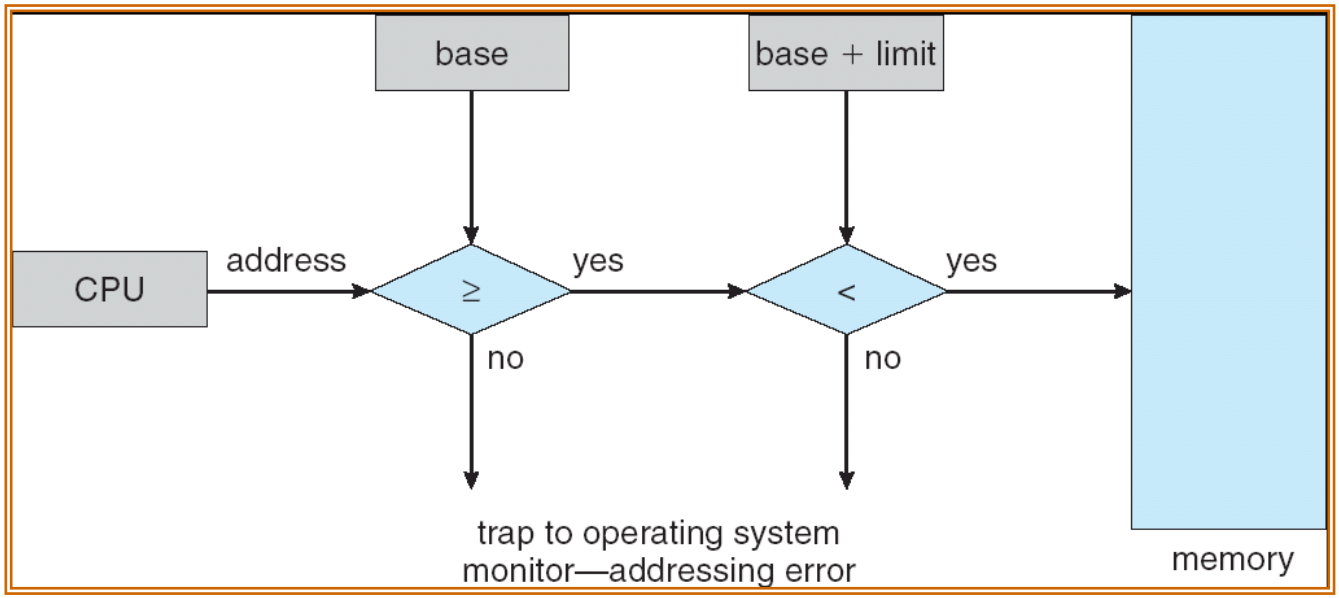
Schematic View of Swapping



Contiguous Allocation

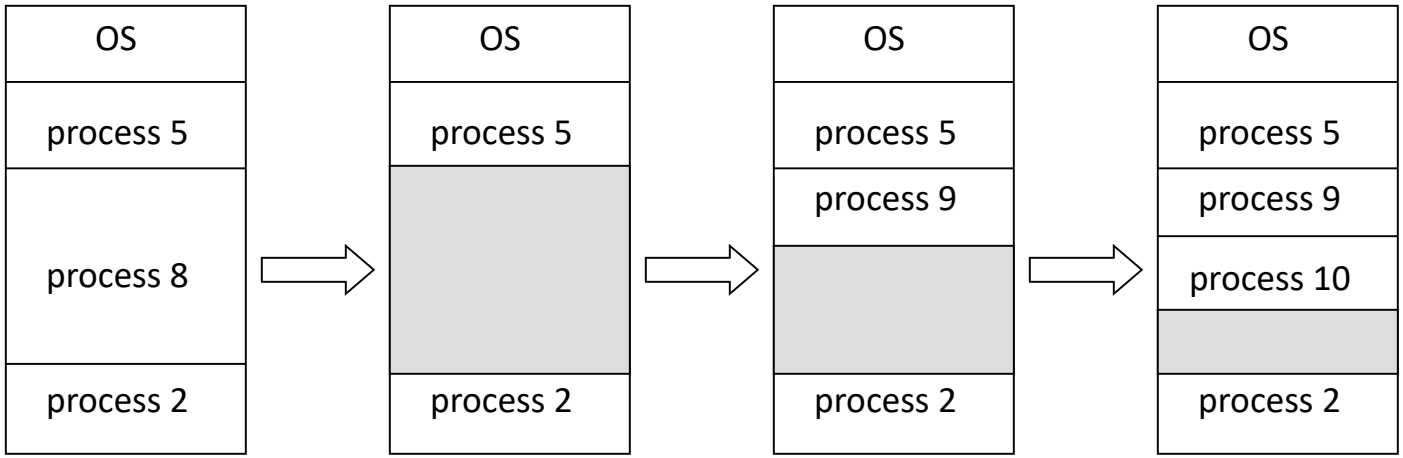
- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically* by adding the value in the relocation register. This mapped address is then sent to access memory.

HW address protection with base and limit registers



Memory Allocation

- Multiple-partition method
  - 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)



Dynamic Storage-Allocation Problem

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

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

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block

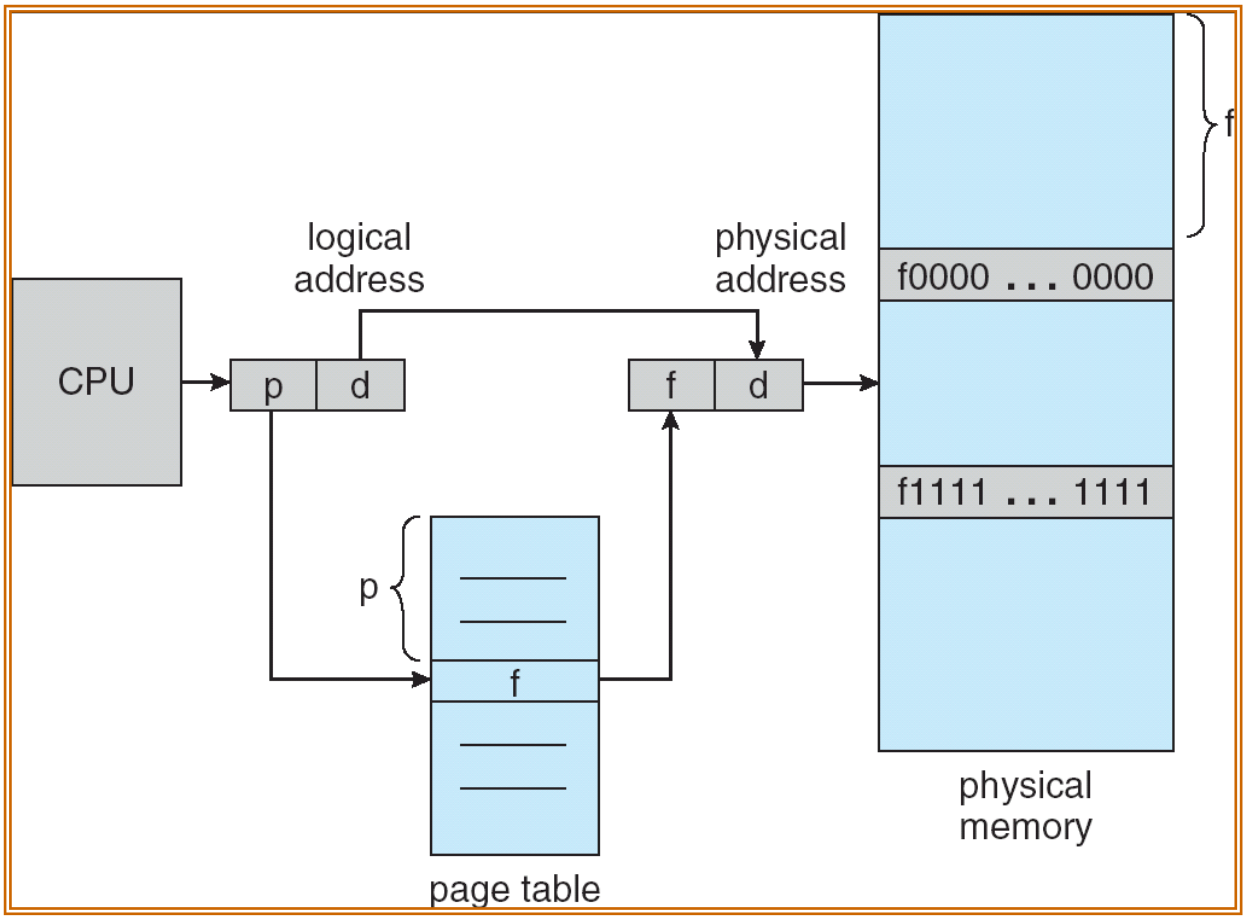
Paging

- Physical 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 8,192 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

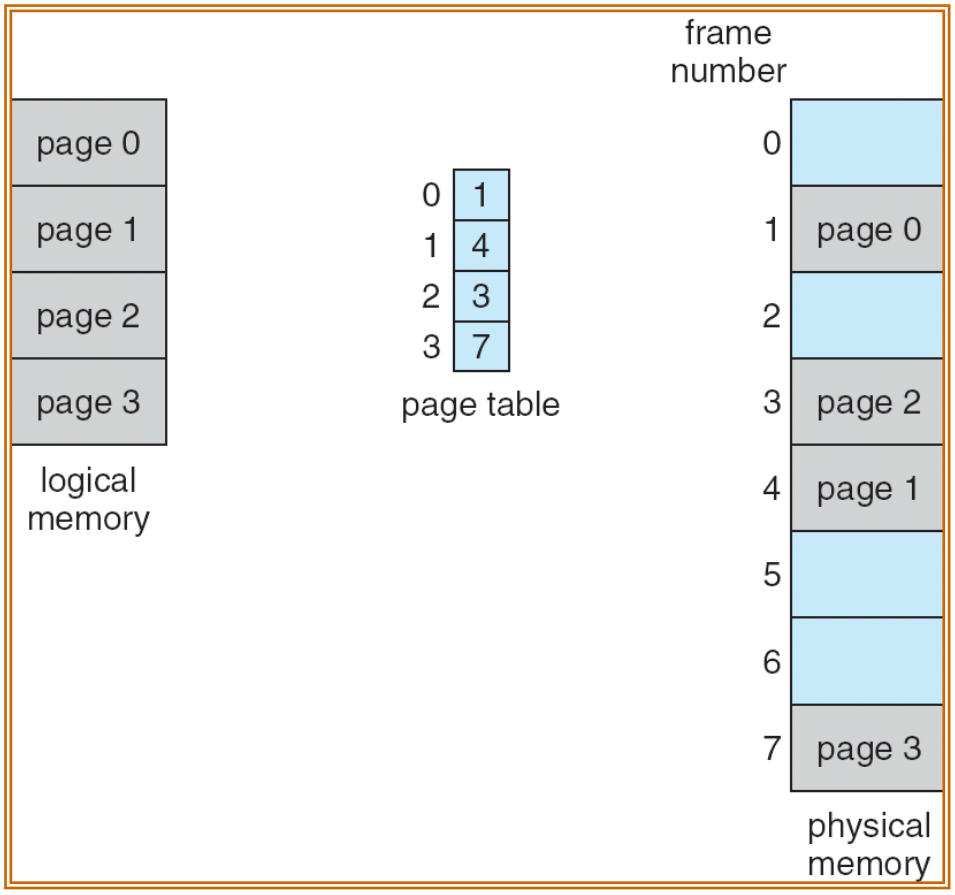
Address Translation Scheme

- Address generated by CPU (logical address) 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

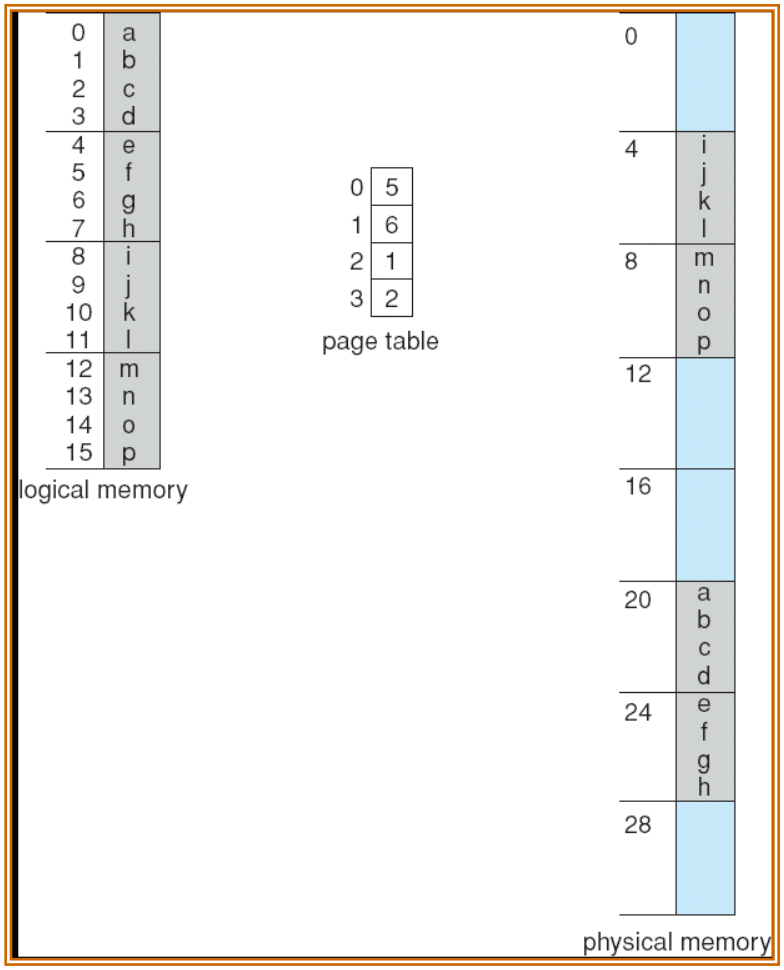
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



32-byte memory and 4-byte pages

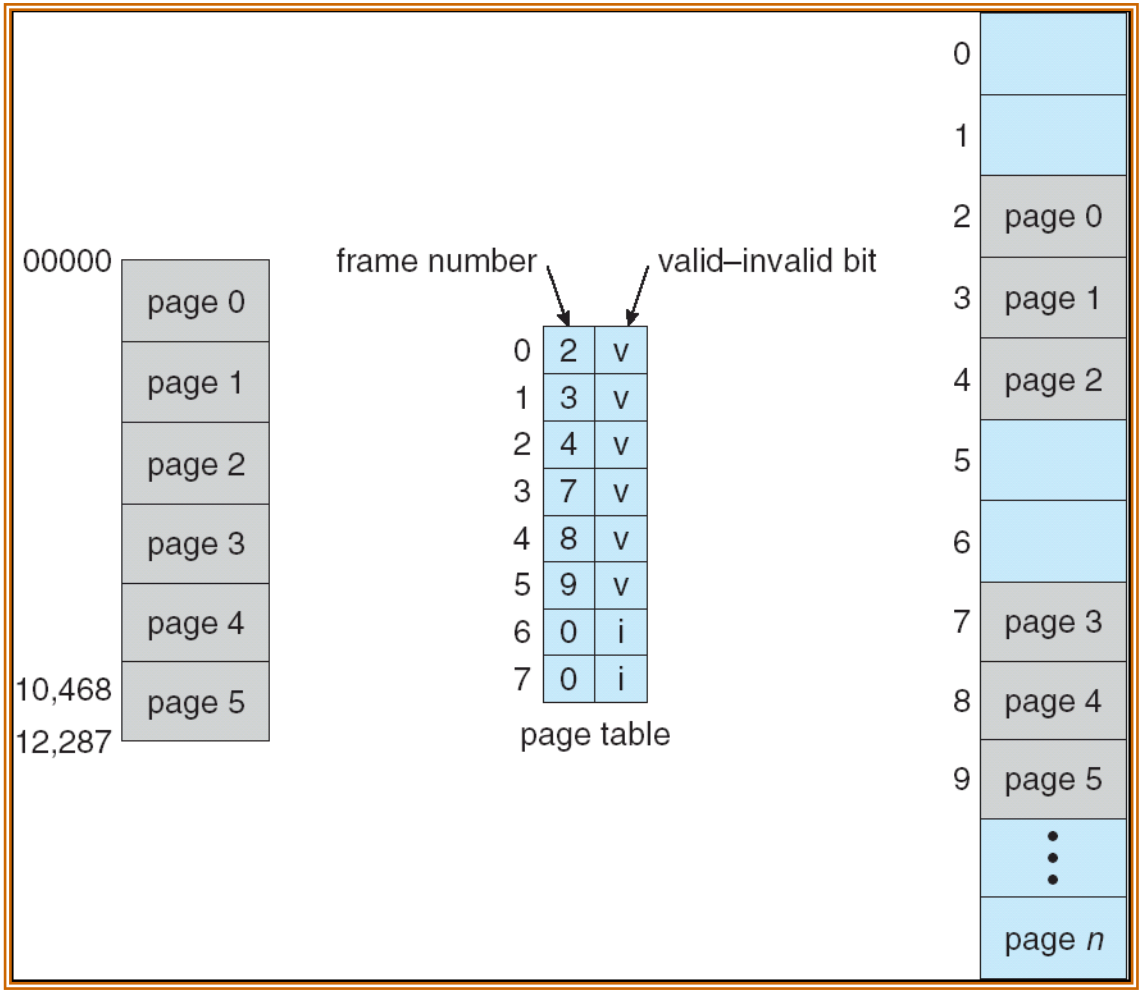
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (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.

Memory Protection

- Memory protection implemented by associating protection bit with each frame. These bits are kept in the **page table**
- **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
  - Illegal addresses are trapped by the use of this **valid-invalid** bit

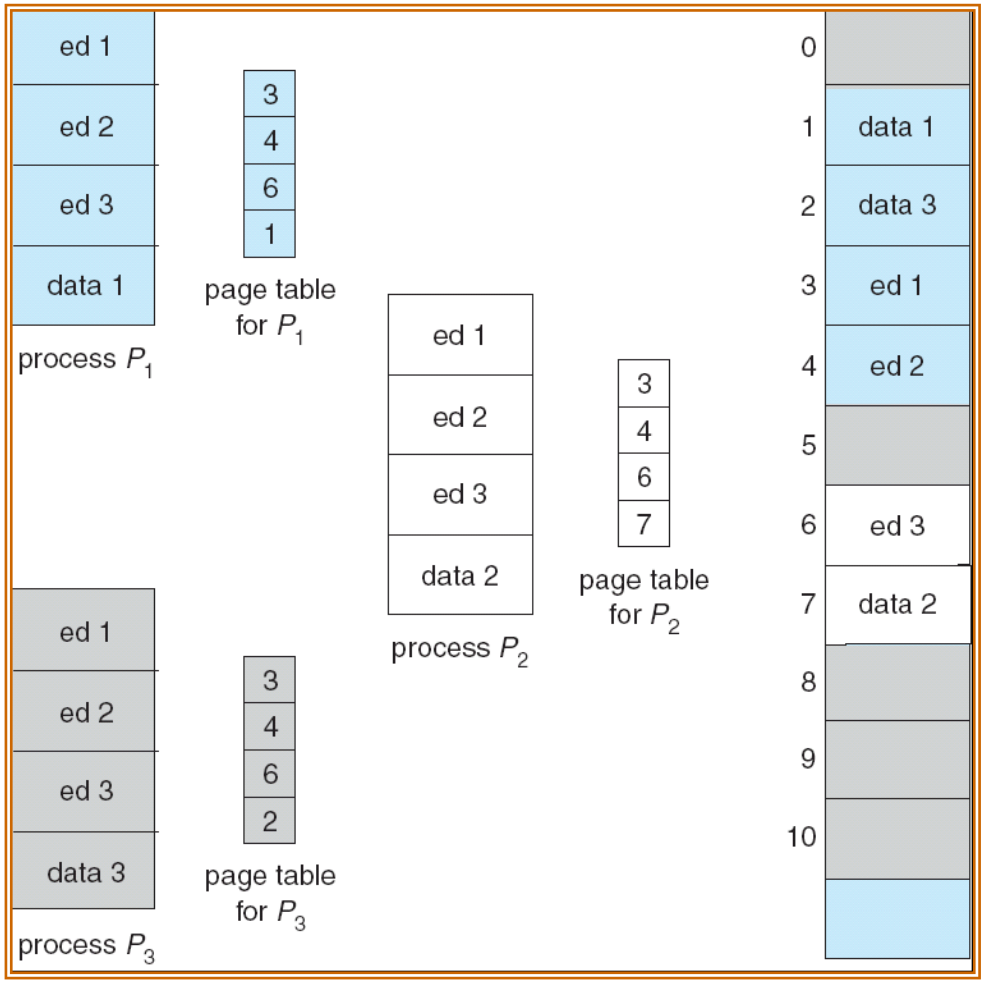
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

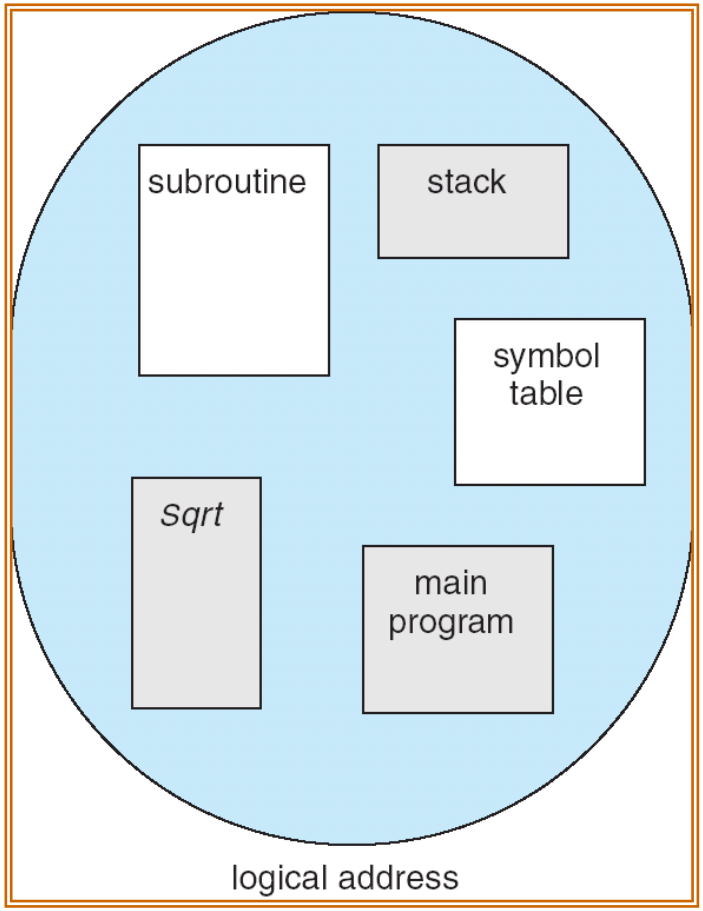


Segmentation

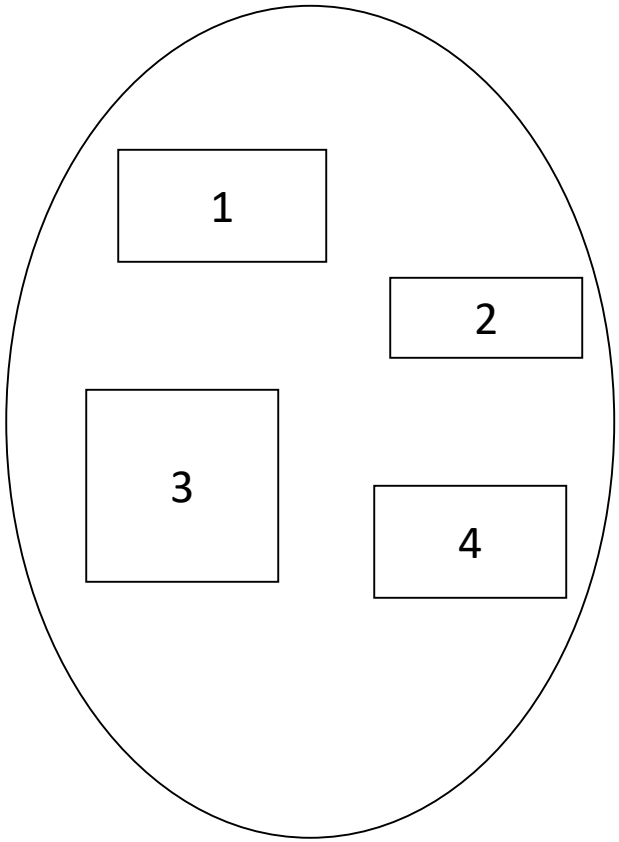
- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  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:

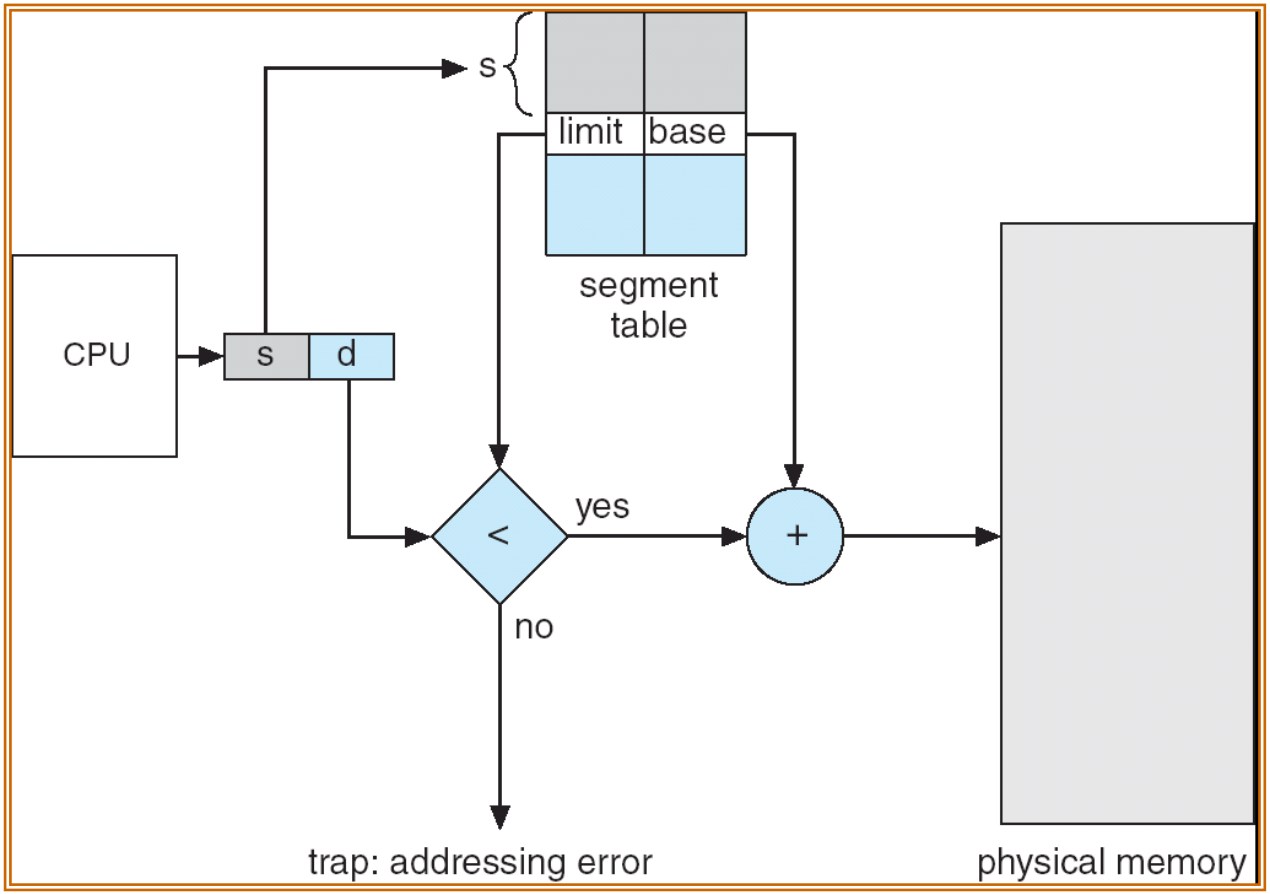
  <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses (user defined addresses) to a one dimensional address; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

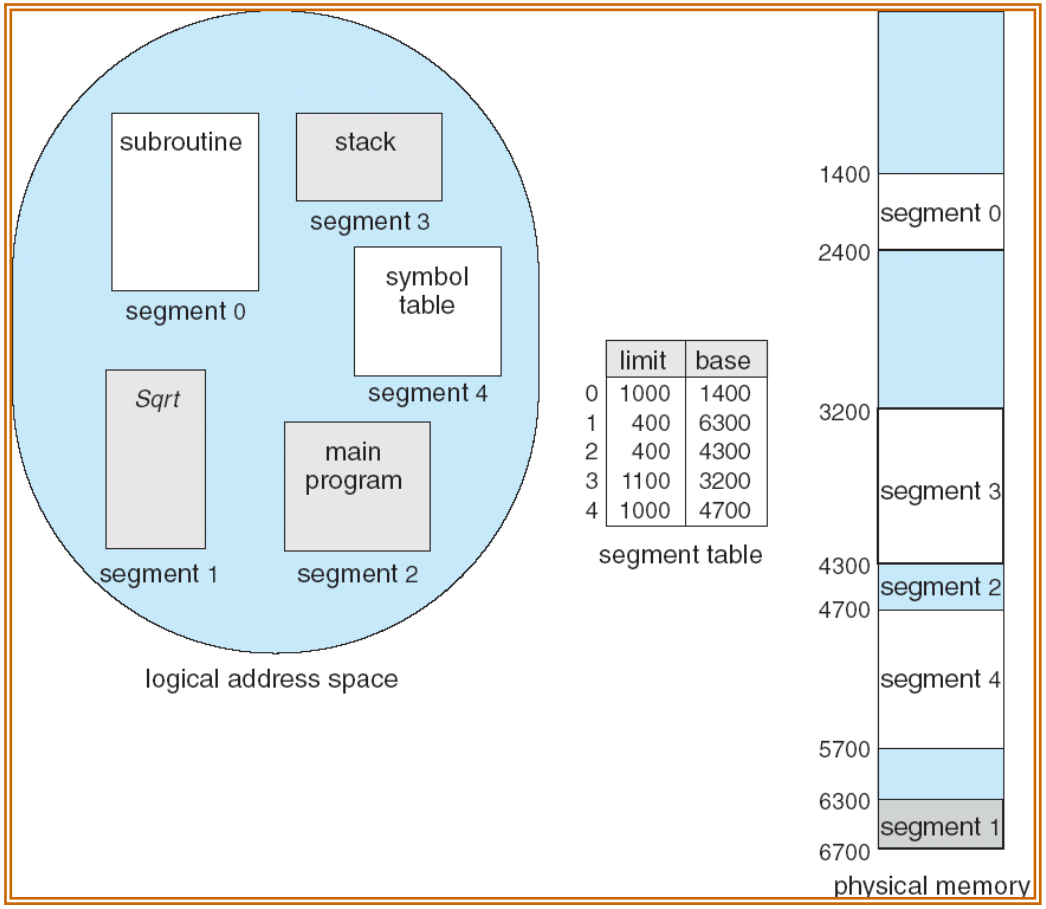
Segmentation Architecture (Cont.)

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

Segmentation Hardware



Example of Segmentation



Virtual Memory

Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

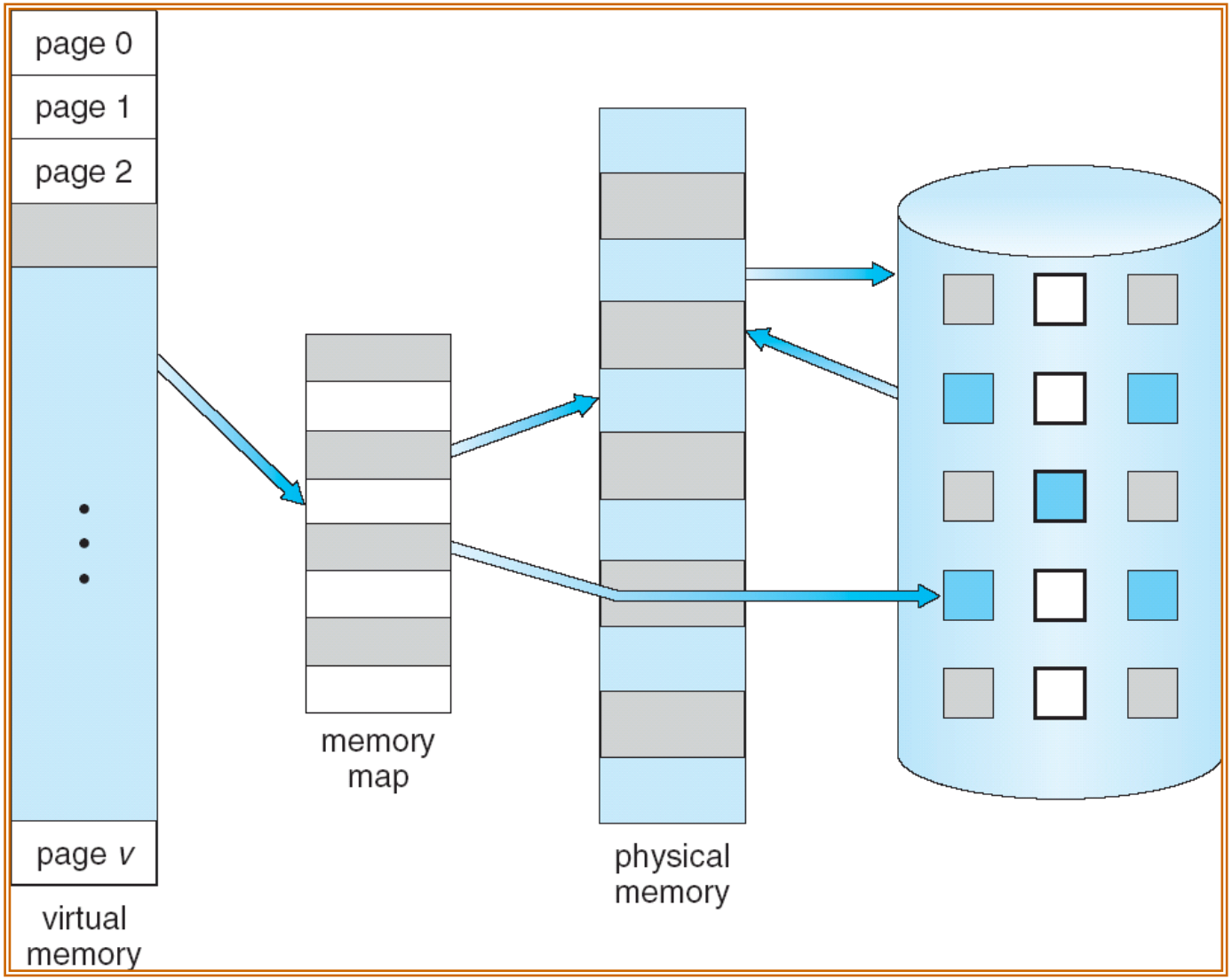
Objectives

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

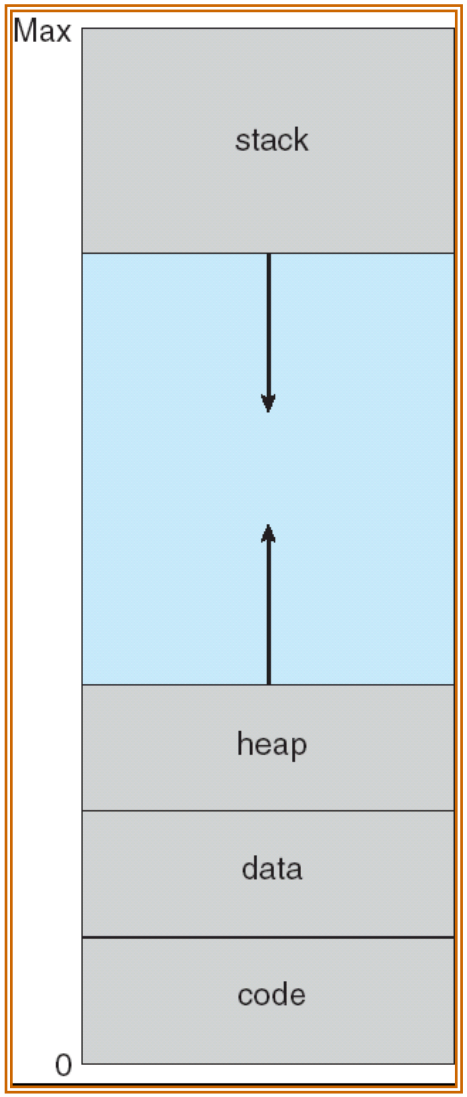
Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

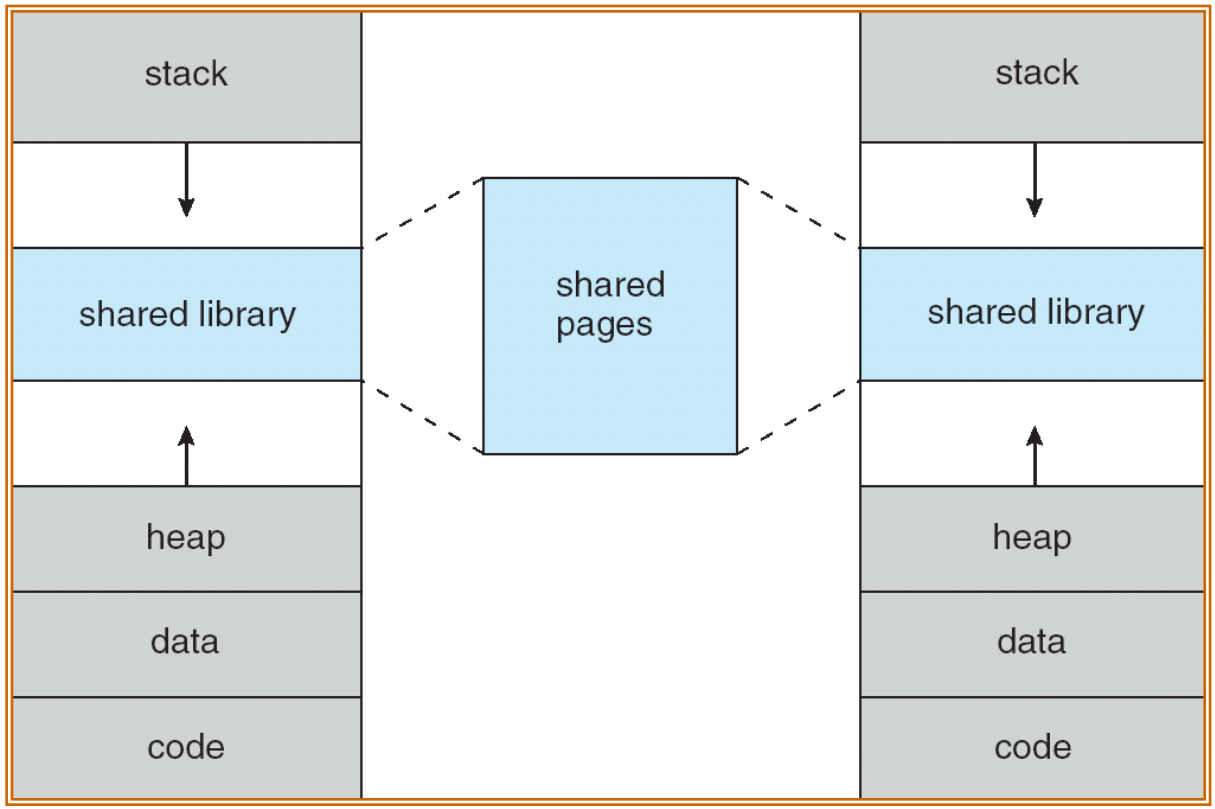
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory

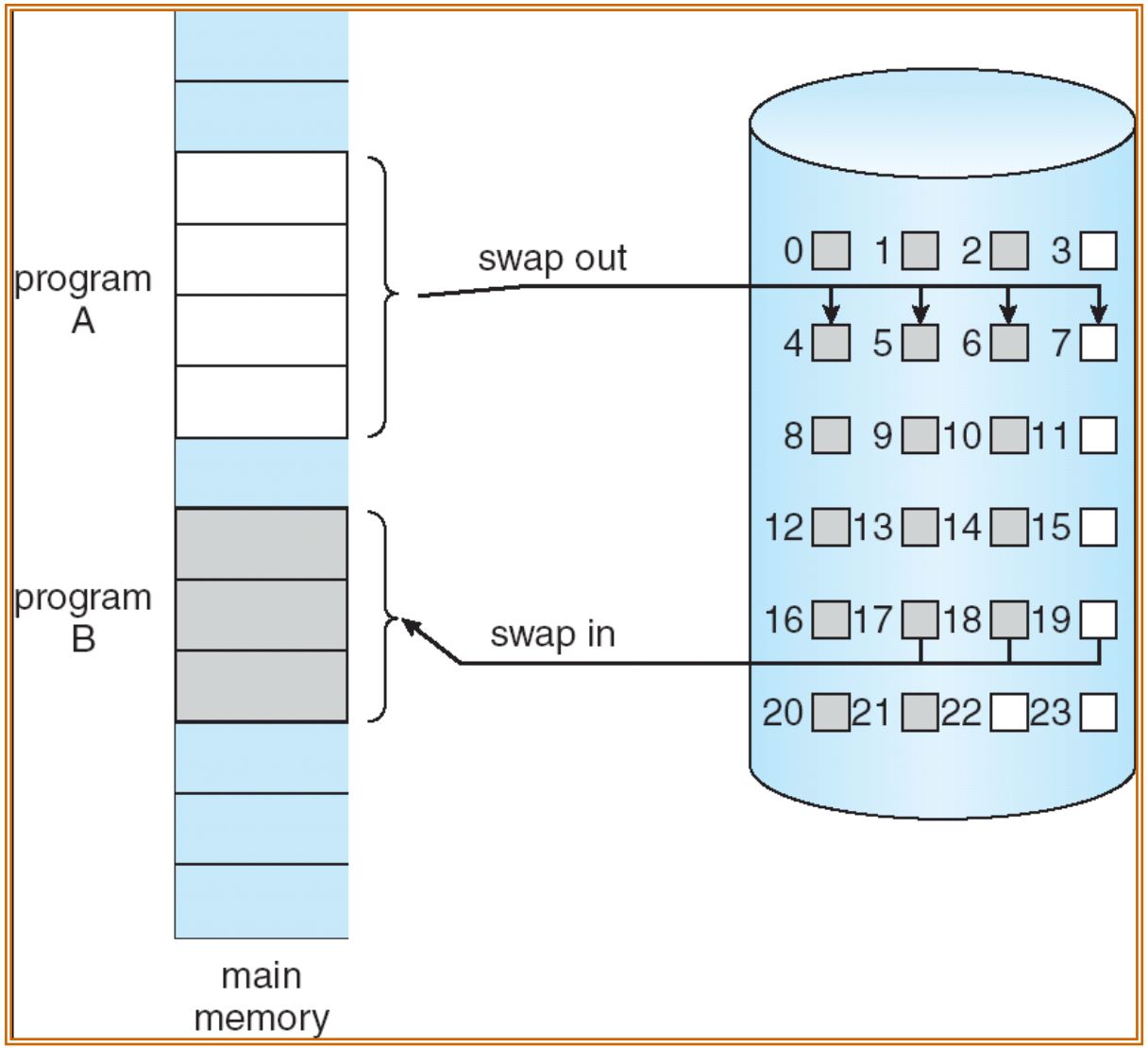


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

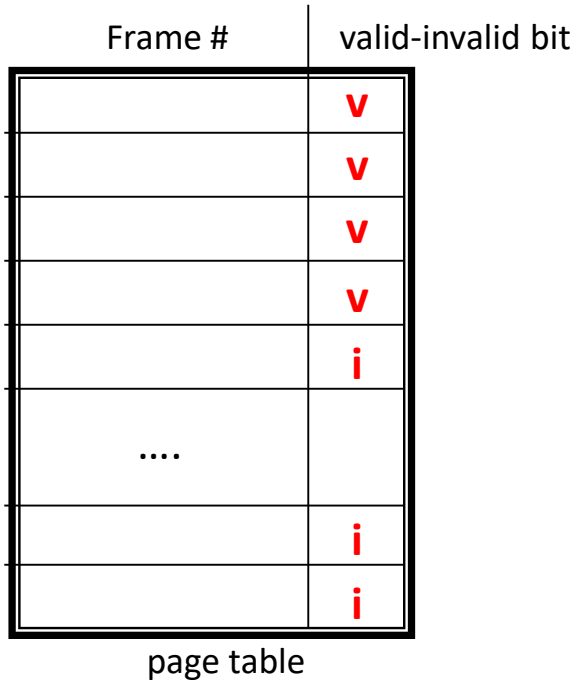
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless that page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



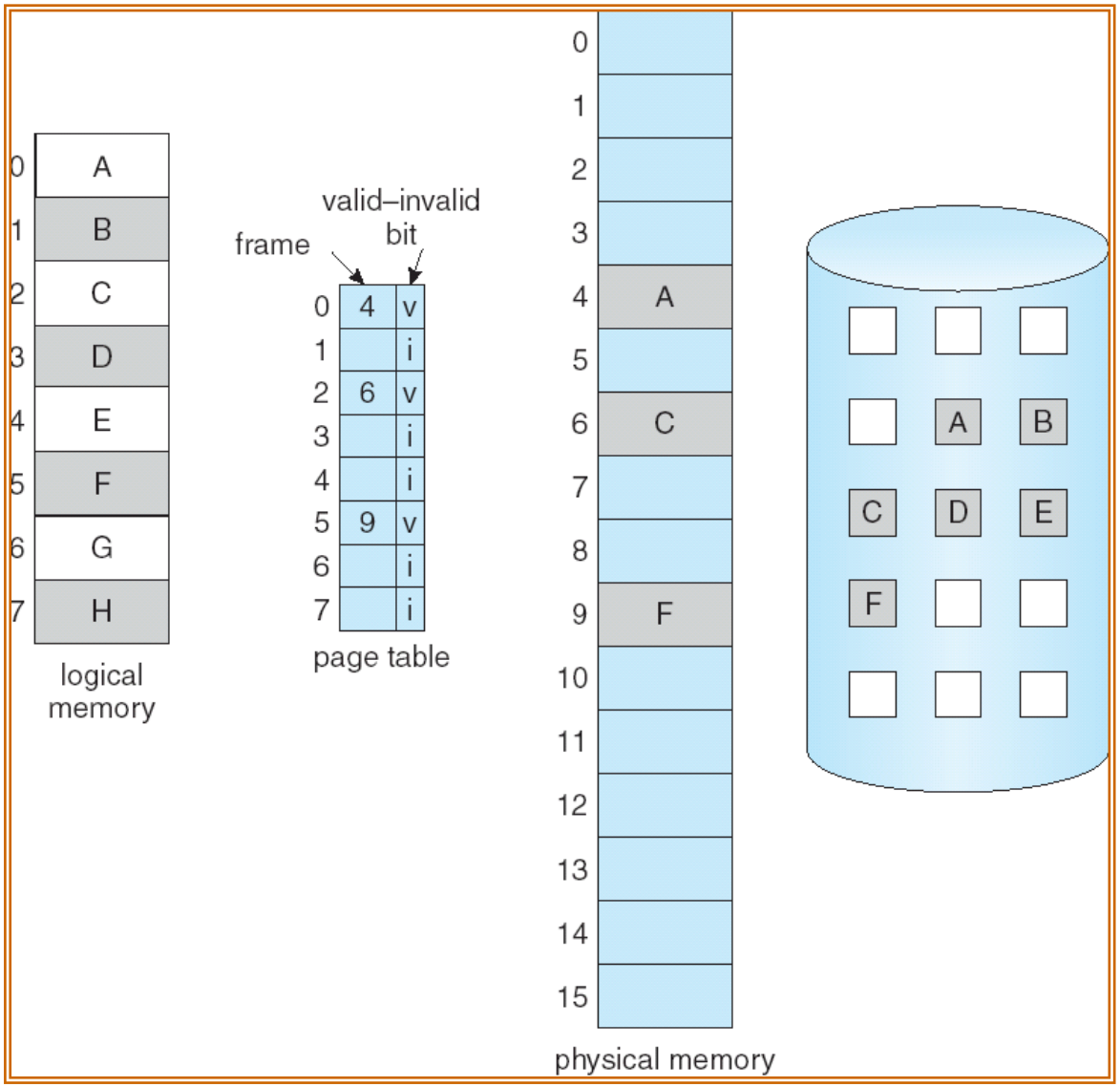
Valid-Invalid Bit

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



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

Page Table When Some Pages Are Not in Main Memory



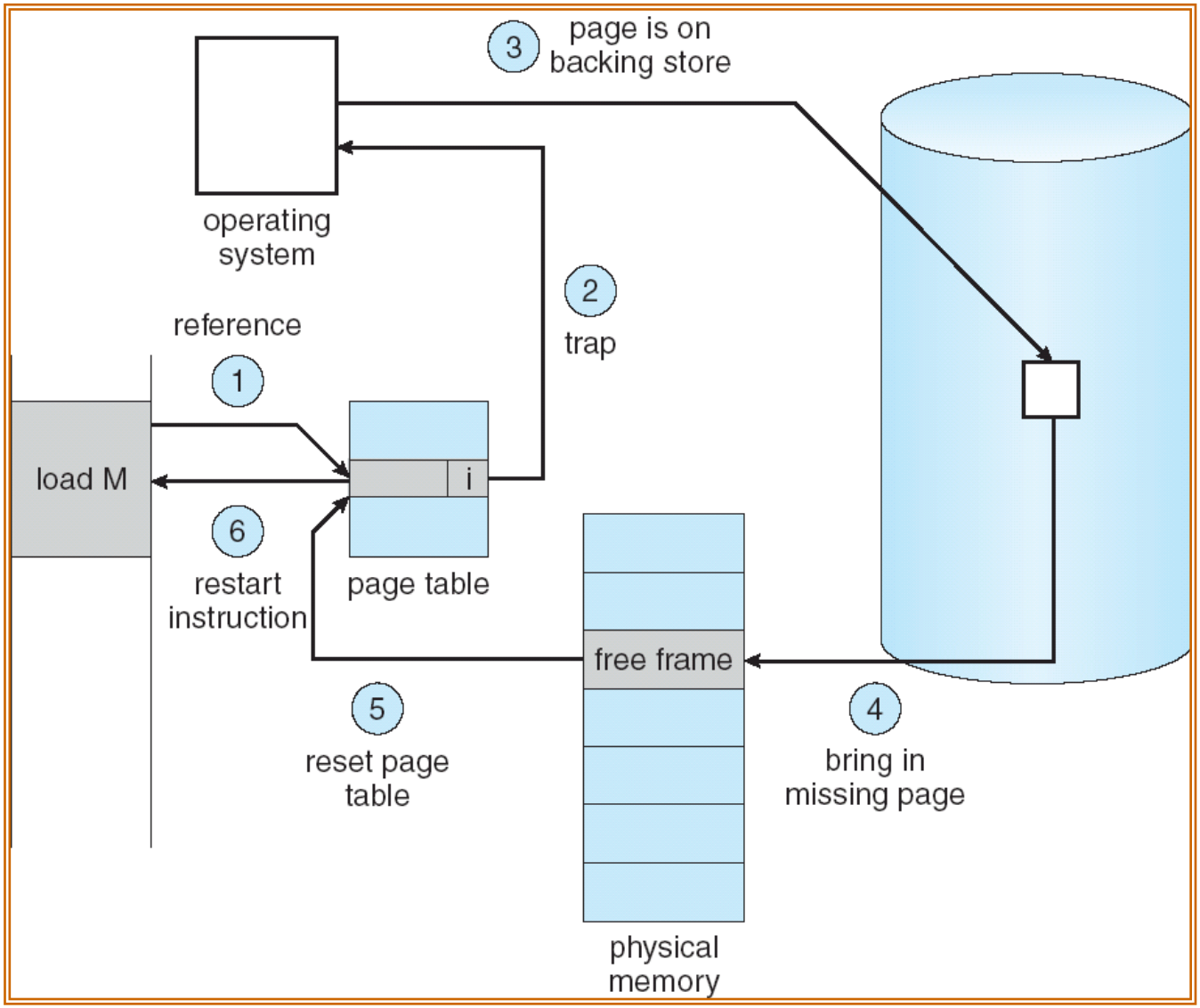
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



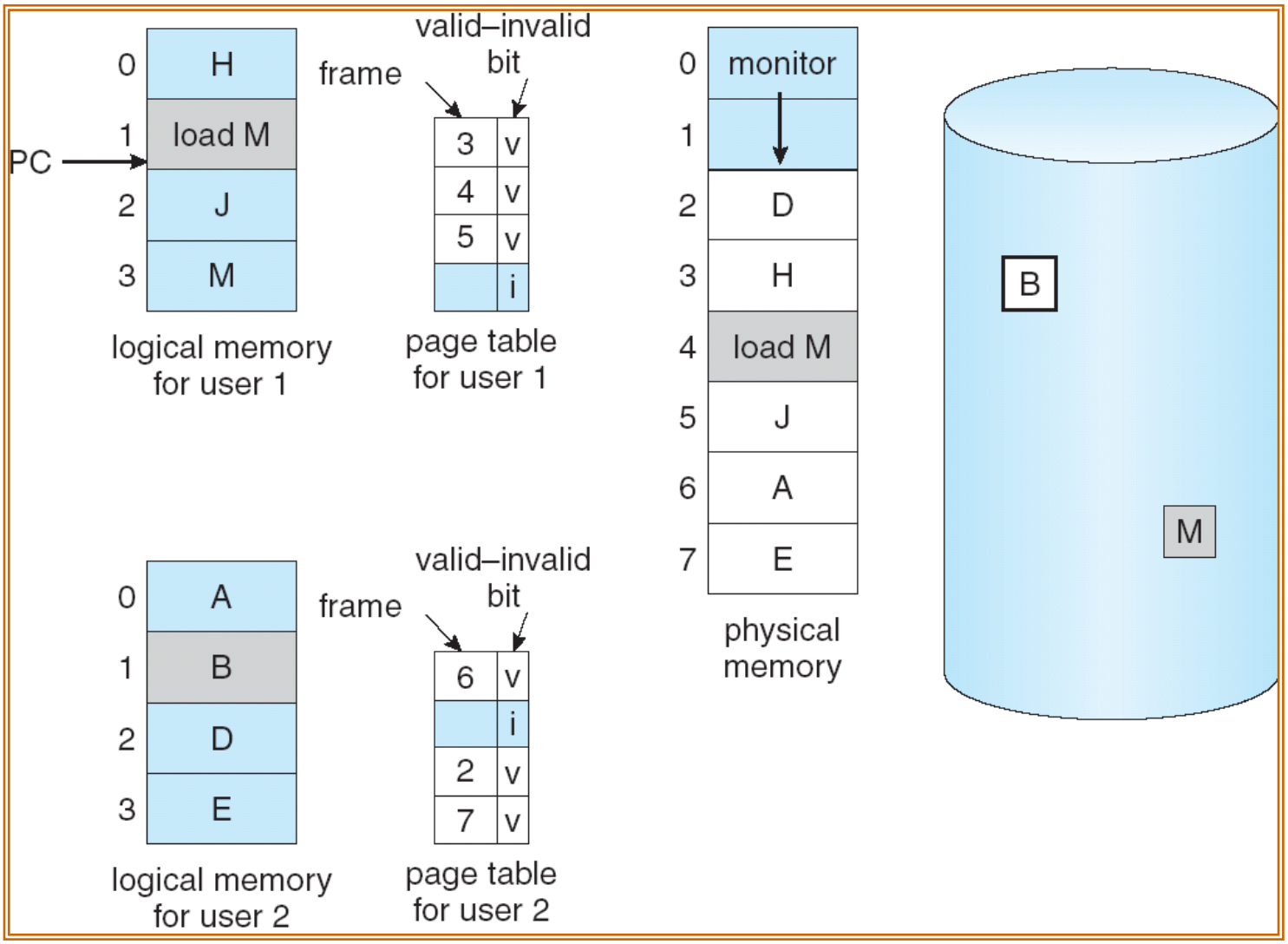
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - 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

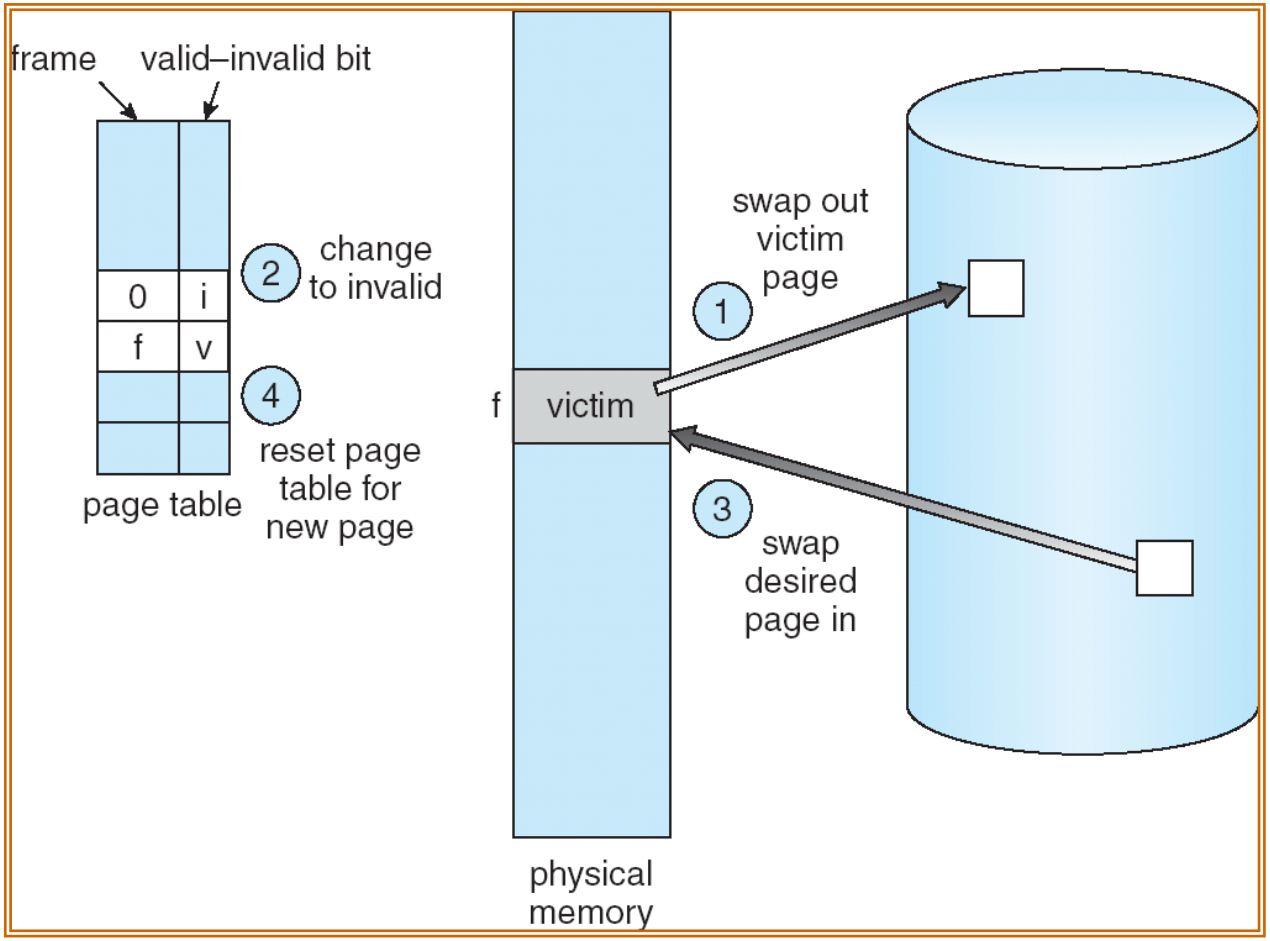
1. Find the location of the desired page on disk

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

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the user process

Page Replacement

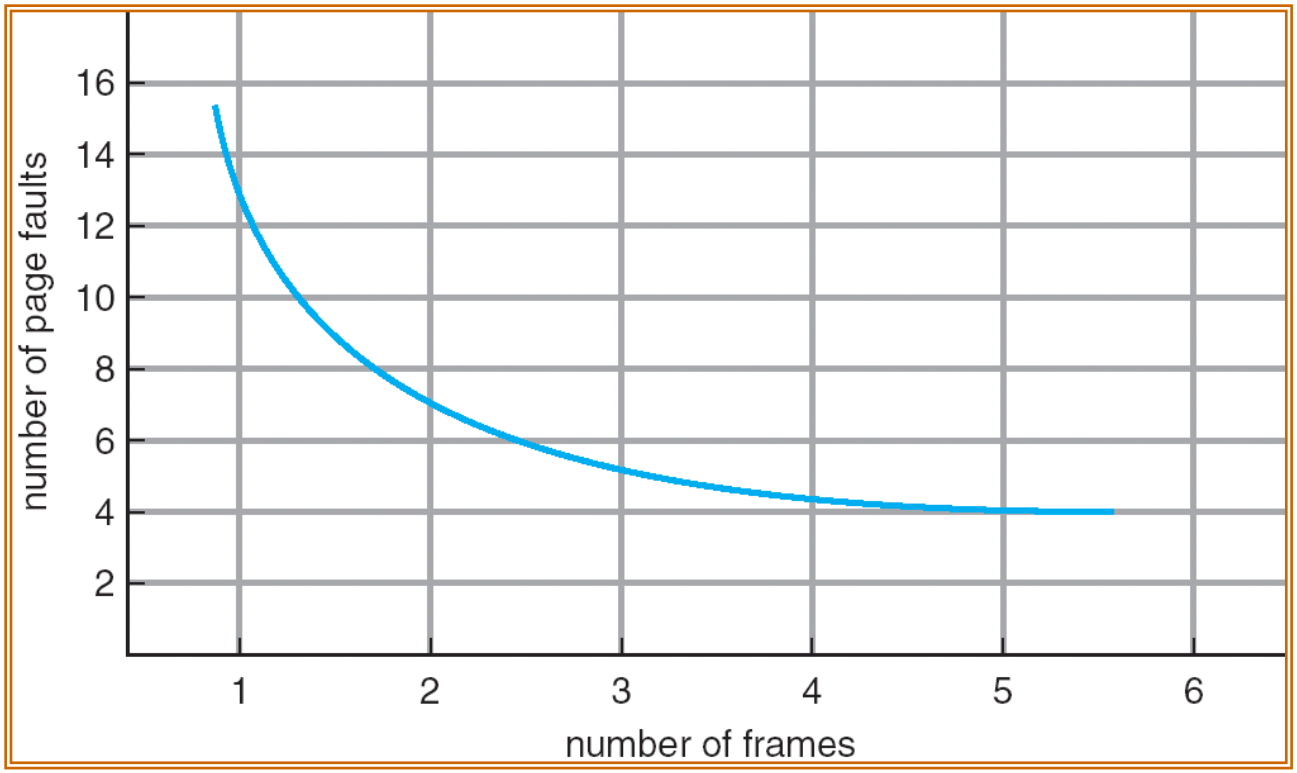


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

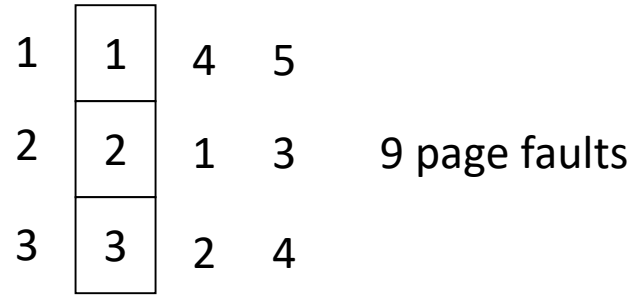
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



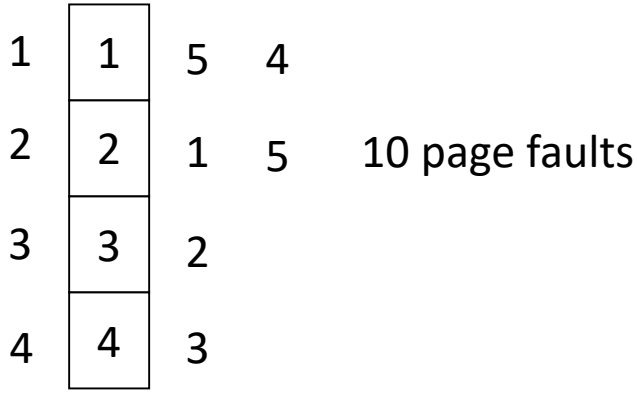
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



9 page faults

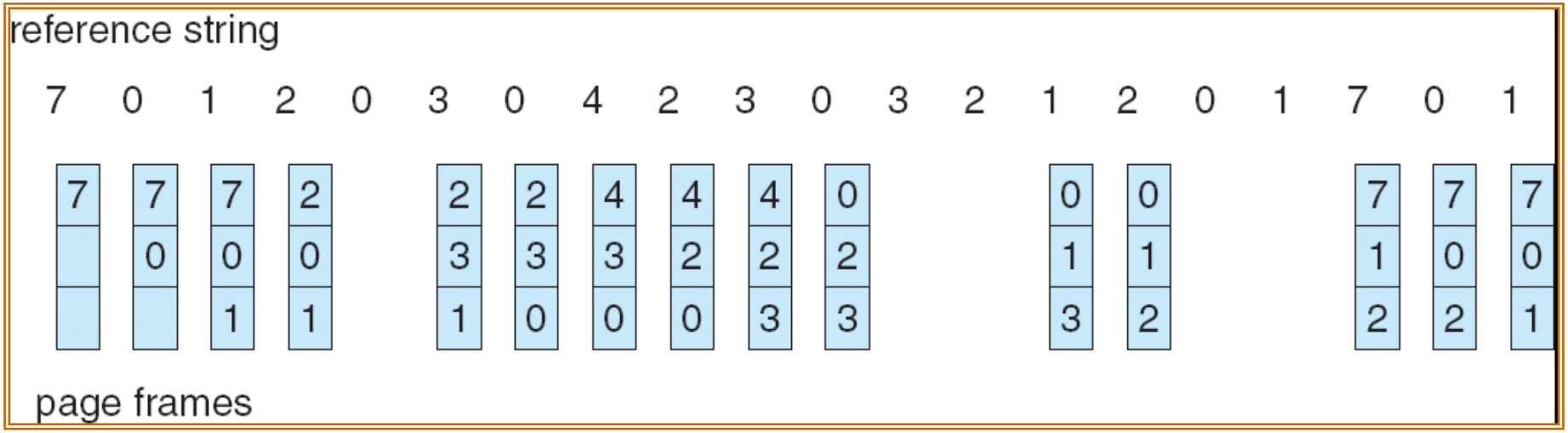
- 4 frames



10 page faults

- **Belady's Anomaly:** more frames $\Rightarrow$ more page faults

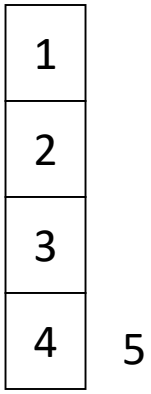
FIFO Page Replacement



Optimal Algorithm

- Replace the page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



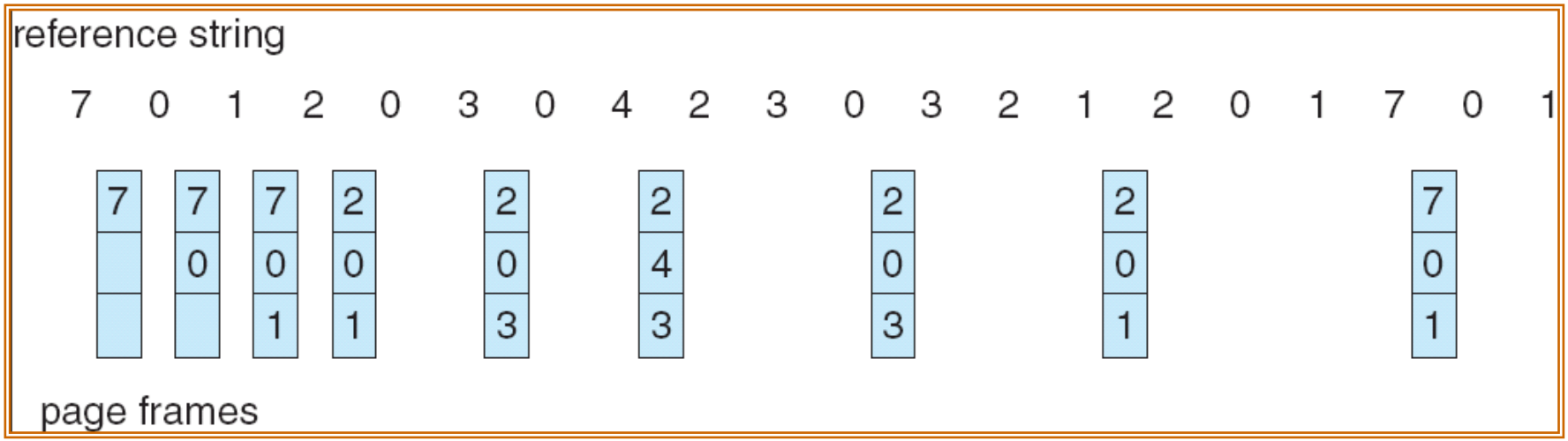
4

5

6 page faults

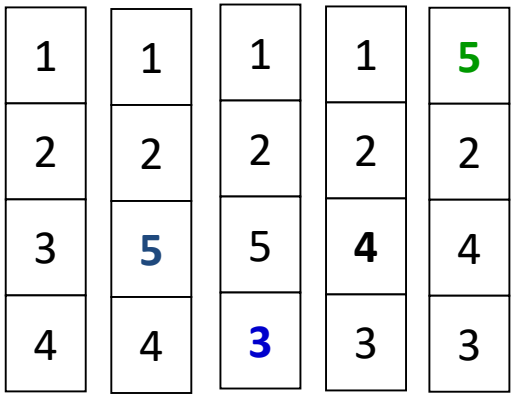
- How do you know this?
- Used for measuring how well your algorithm performs
- Lowest page fault rate

Optimal Page Replacement



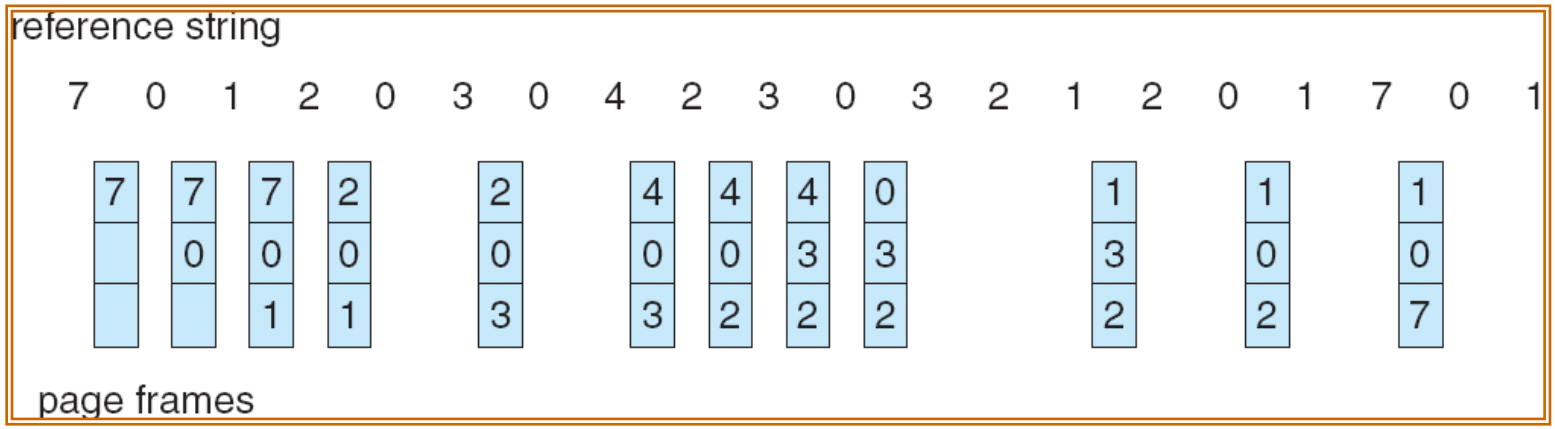
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted

File System Interface

File-System Interface

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

Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection

File Concept

- Contiguous logical address space
- Files are mapped by the Operating System onto physical devices
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program

File Structure

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

File Attributes

- **Name** – only information kept in human-readable form
- **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

File Operations

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file** (file seek)
- **Delete**
- **Truncate**
- *Open(F_i)* – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close (F_i)* – move the content of entry F_i in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage 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
- 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 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

Access Methods

- **Sequential Access**

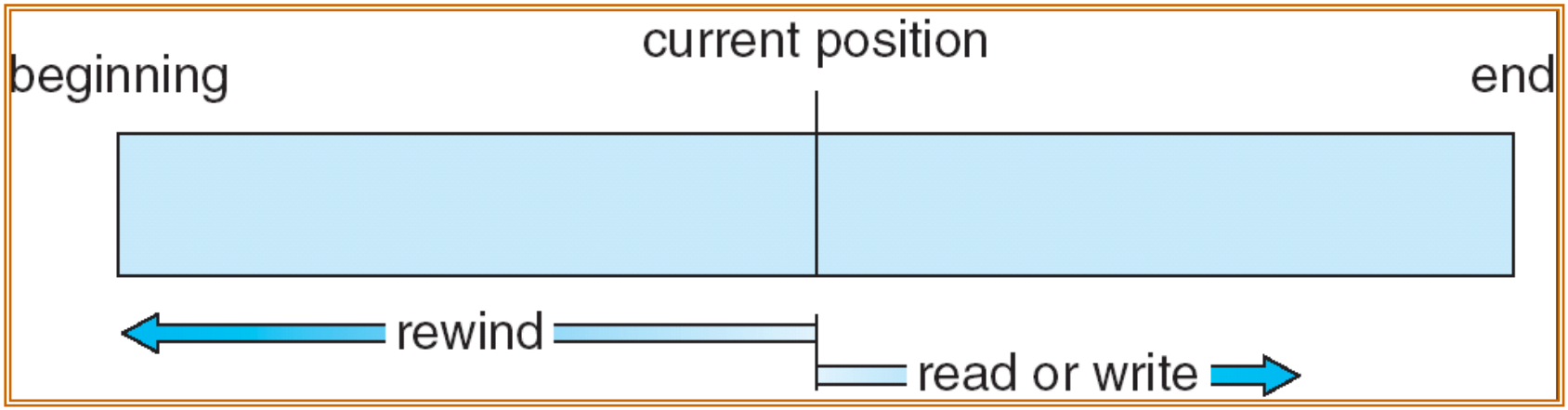
  read next
  write next
  reset

- **Direct Access**

  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

  n = relative block number

Sequential-access File

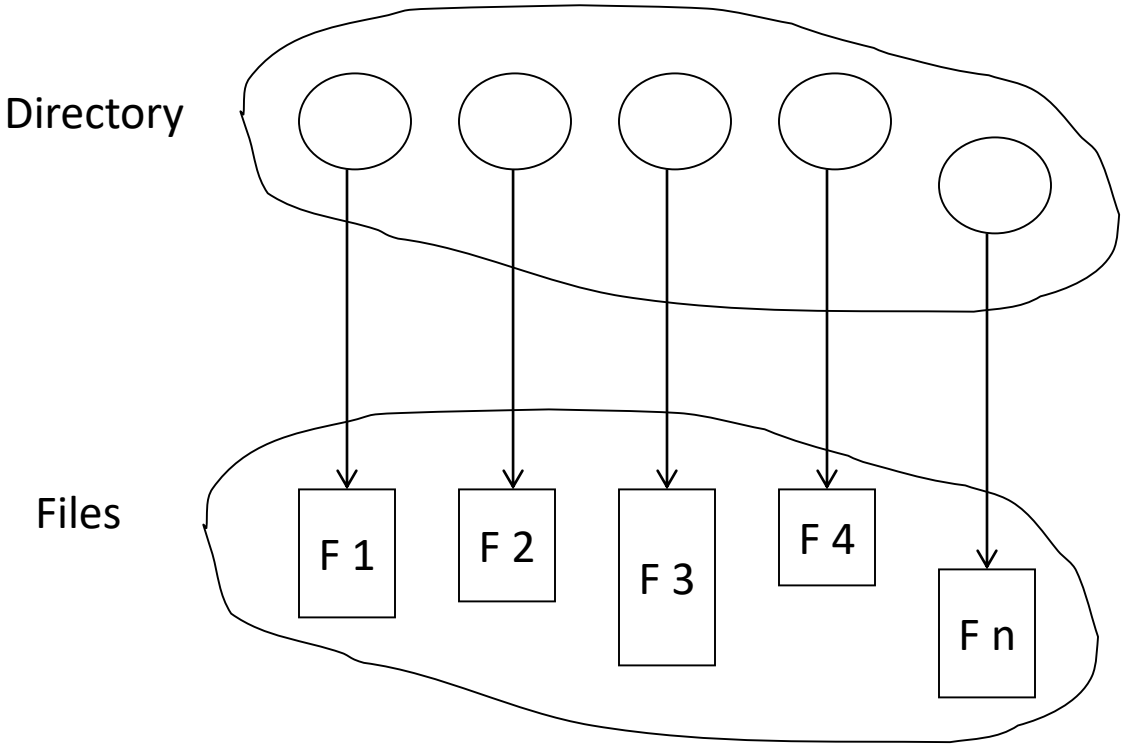


Simulation of Sequential Access on a Direct-access File

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

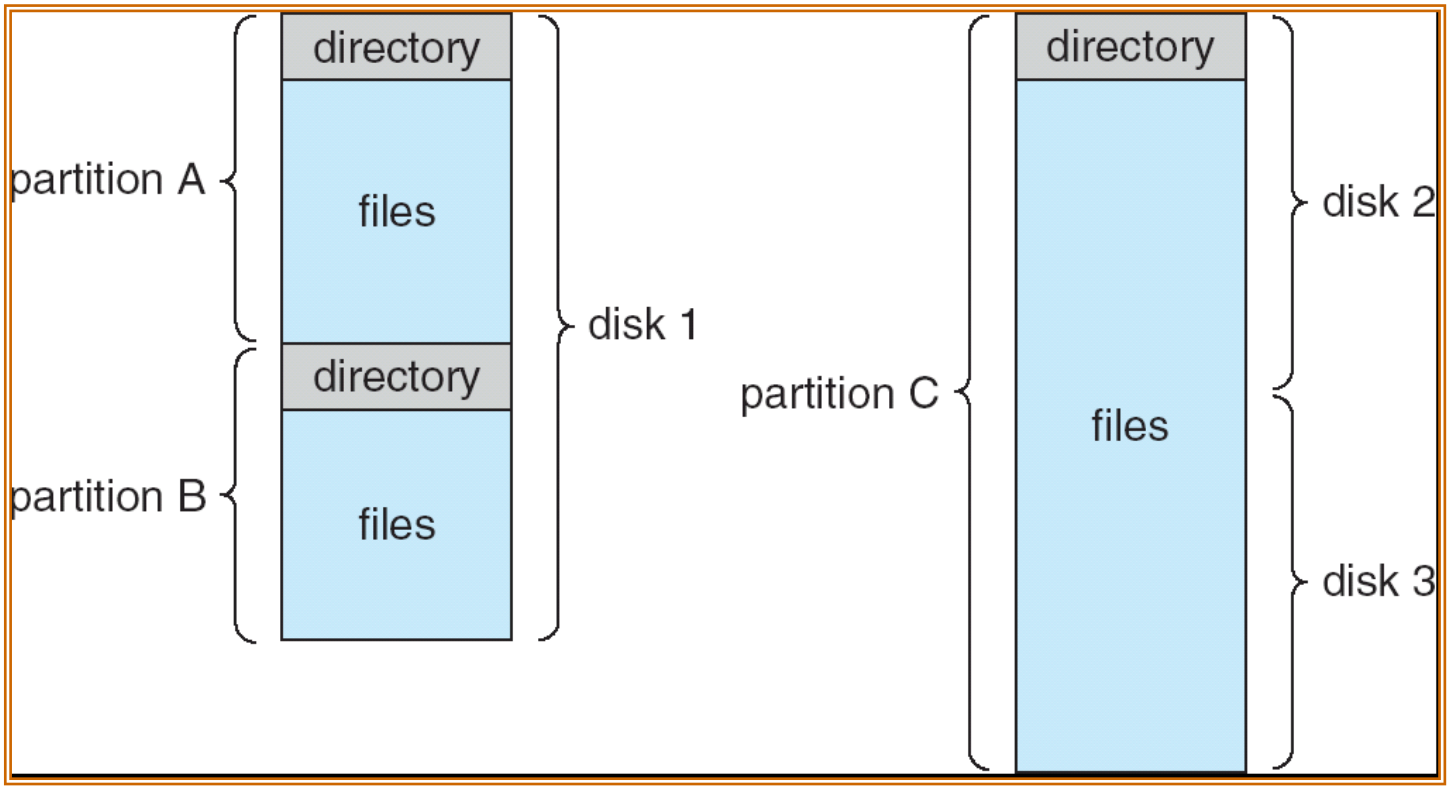
Directory Structure

- A collection of nodes containing information about all files



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

A Typical File-system Organization



Operations Performed on Directory

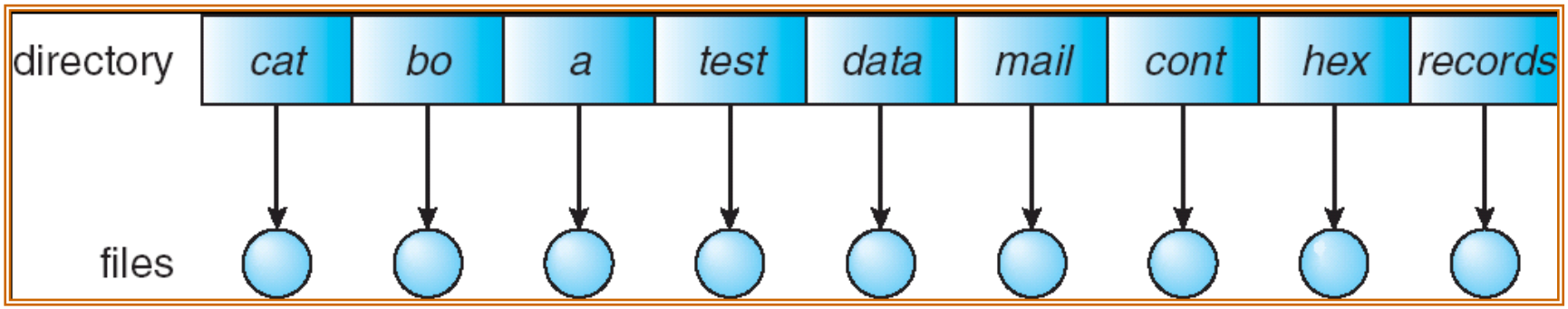
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Organize the Directory (Logically) to Obtain

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

Single-Level Directory

- A single directory for all users

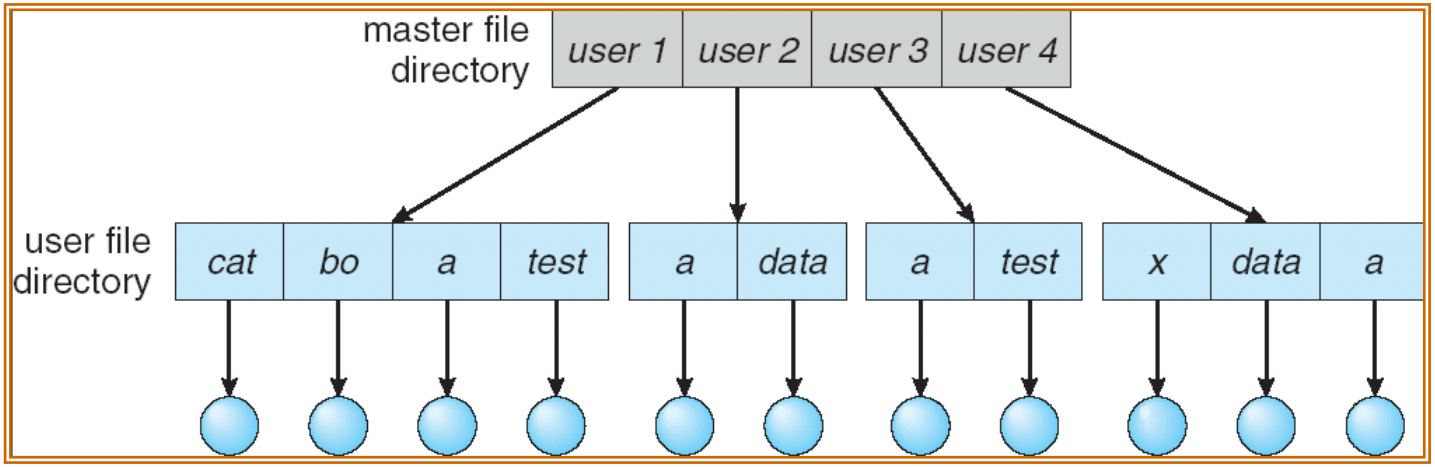


Naming problem

Grouping problem

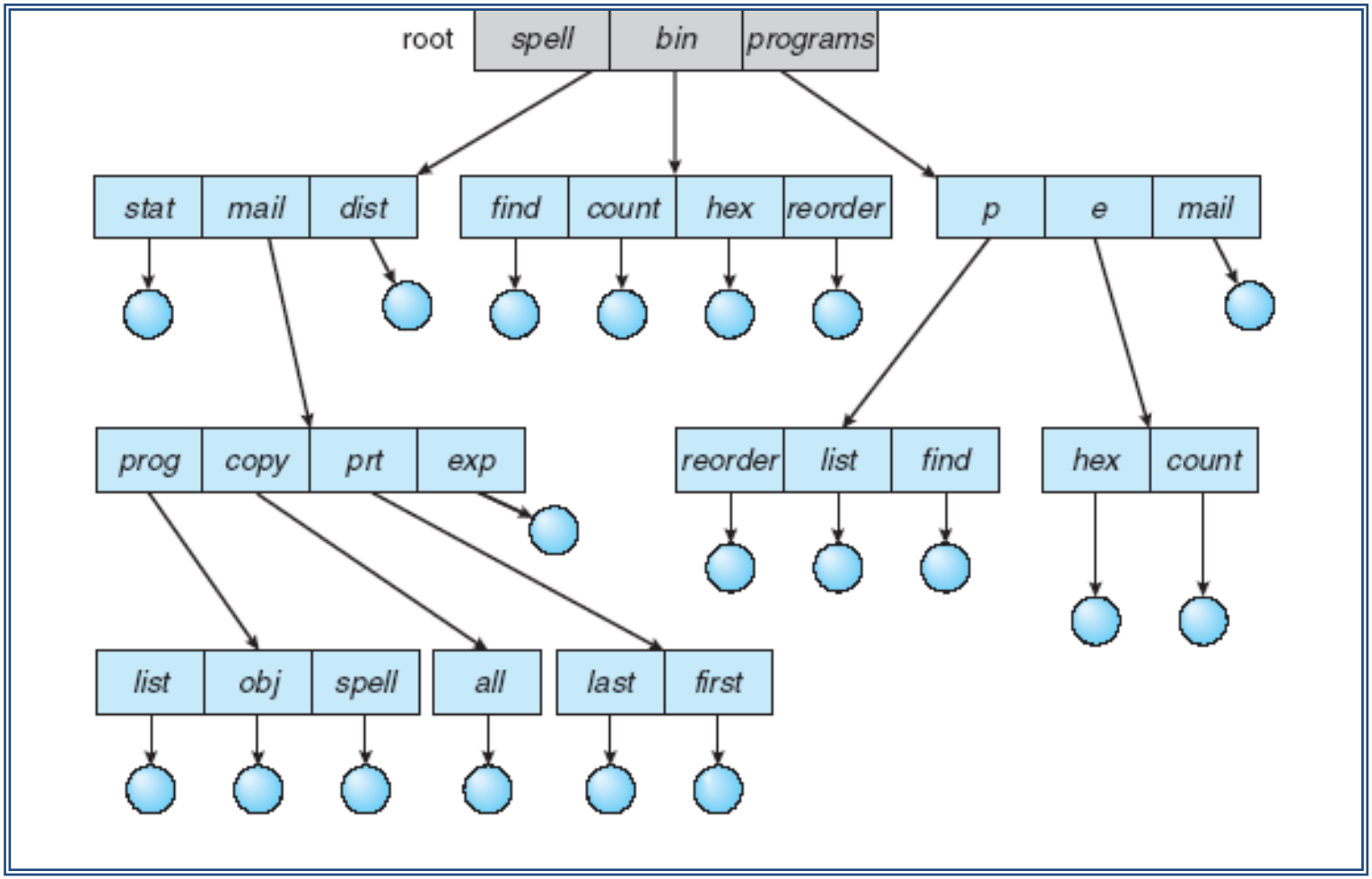
Two-Level Directory

- Separate directory for each user



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

Tree-Structured Directories



Tree-Structured Directories (Cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - cd /spell/mail/prog/list
  - type _____
  - cat _____

Tree-Structured Directories (Cont)

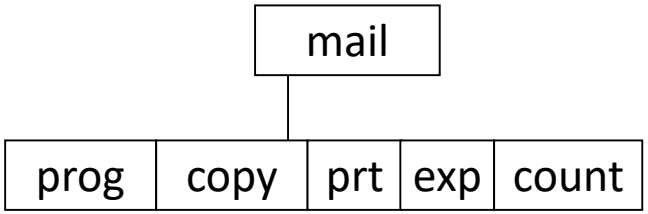
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

  Example: if in current directory /mail

  mkdir count

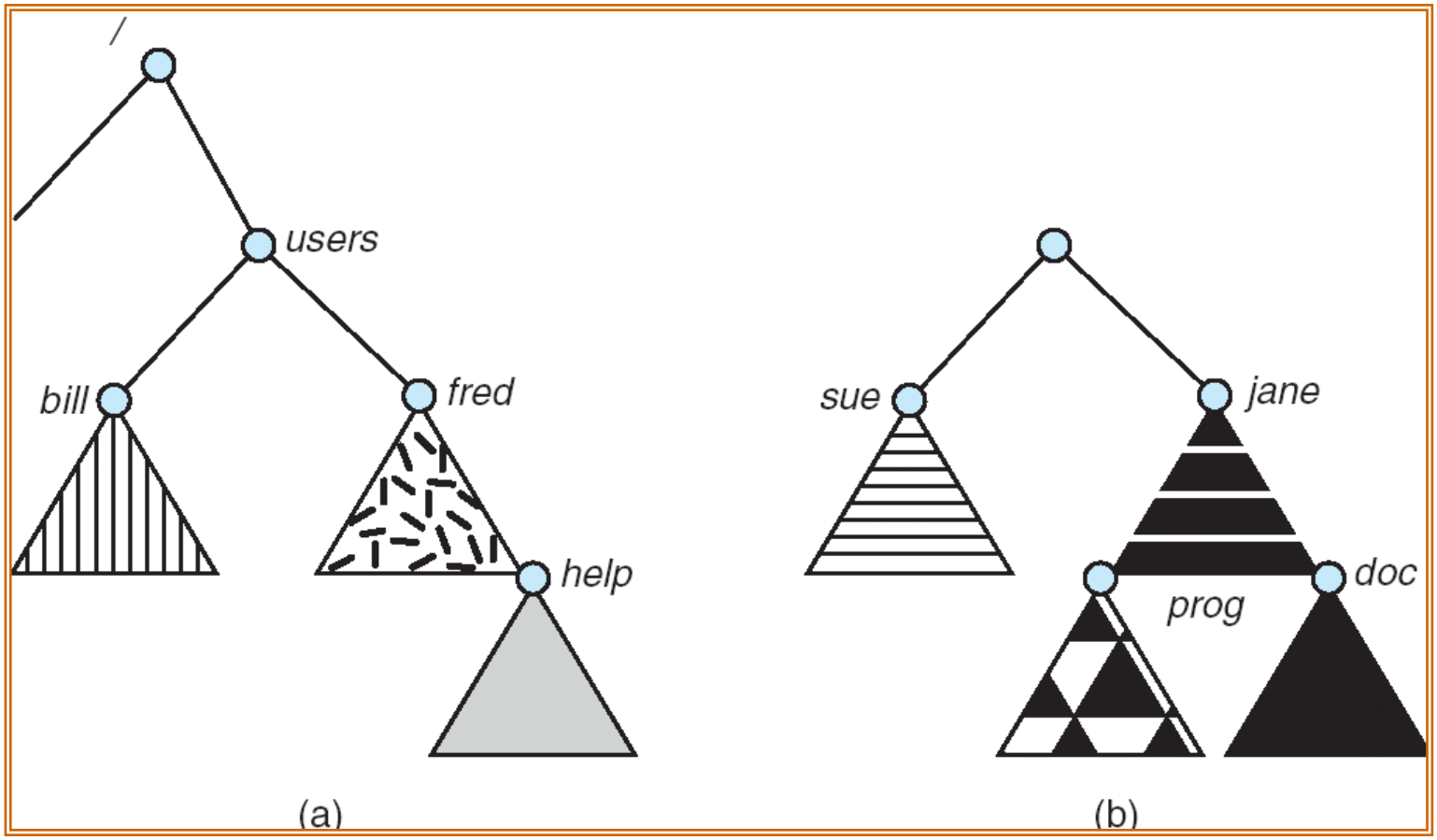


Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

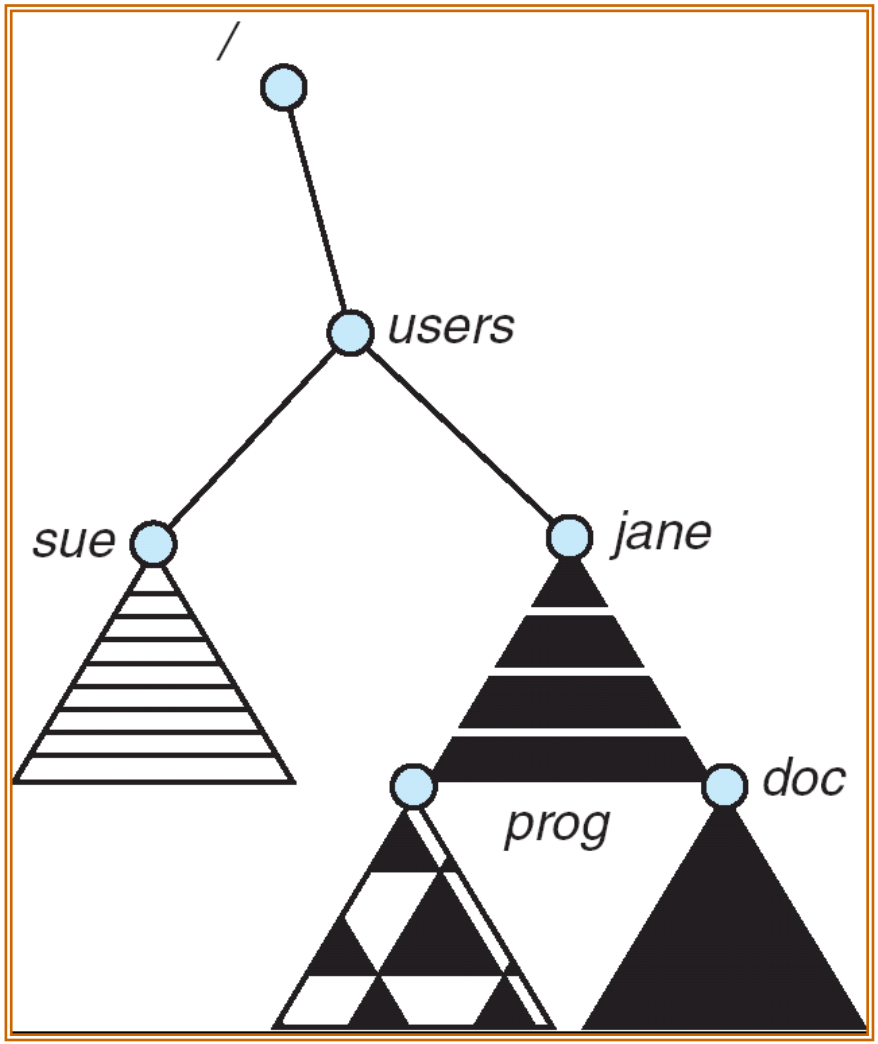
File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (figure is given in the next slide) is mounted at a **mount point**

(a) Existing. (b) Unmounted Partition



Mount Point



File Sharing

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

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights

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

File Sharing – Failure Modes

- 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 include all information in each request, allowing easy recovery but less security

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom

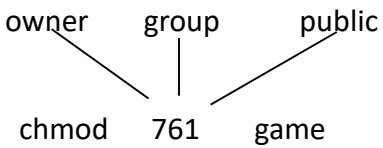
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

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

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

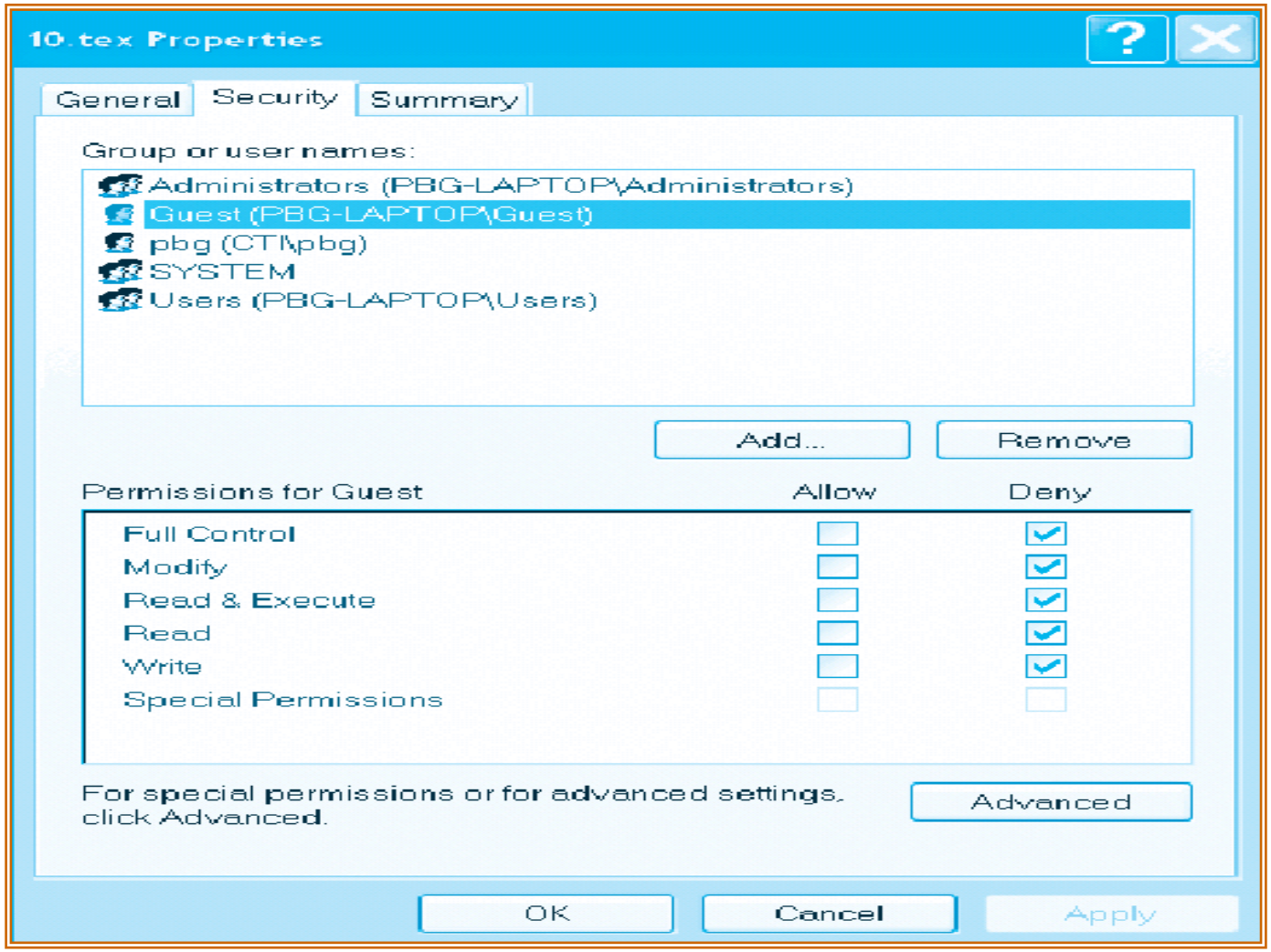
- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Windows XP 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/

Mass-Storage Systems

Mass-Storage Systems

- Overview of Mass Storage Structure
- Disk Structure
- Disk Scheduling
- Disk Management
- Swap-Space Management
- Stable-Storage Implementation
- Tertiary Storage Devices
- Operating System Issues

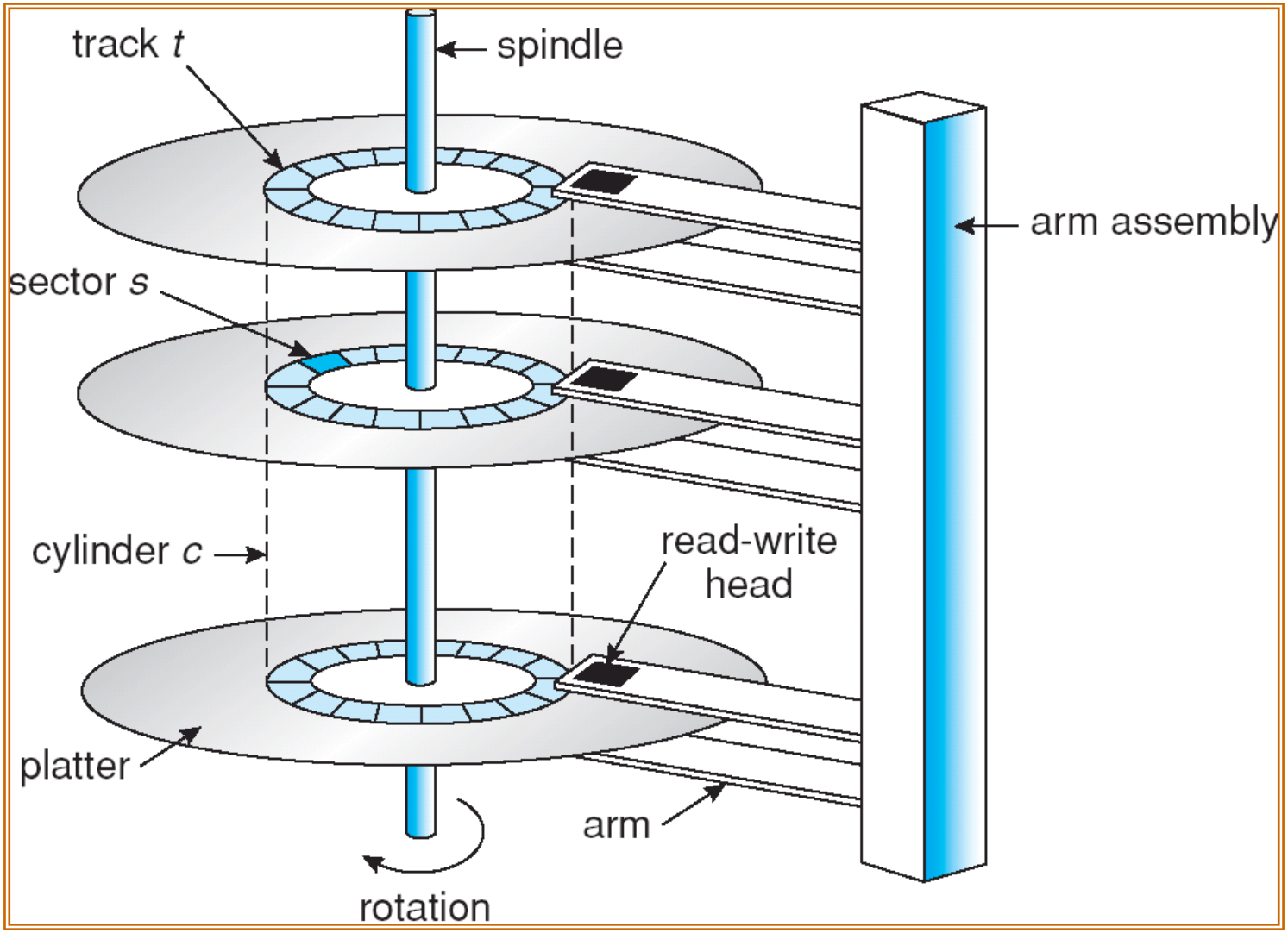
Objectives

- Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices
- Explain the performance characteristics of mass-storage devices
- Discuss operating-system services provided for mass storage.

Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 200 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 (normally cannot be repaired; the entire disk must be replaced).
- Disks can be removable
- Drive attached to computer via **I/O bus**
  - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
  - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array

Moving-head Disk Mechanism



Overview of Mass Storage Structure (Cont.)

- Magnetic tape
  - Was early secondary-storage medium
  - Relatively permanent and holds large quantities of data
  - Access time slow
  - Random access ~1000 times slower than disk
  - Mainly used for backup, storage of infrequently-used data, transfer medium between systems
  - Kept in spool and wound or rewound
  - Once data under head, transfer rates comparable to disk
  - 20-200GB typical storage
  - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - *Seek time* is the time for the disk arm to move the heads to the cylinder containing the desired sector.
  - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time ≈ 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 (contd.)

- We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

- Whenever a process needs I/O to or from the disk, it issues a system call to the O.S.

- The request specifies several pieces of information:
  - Whether this operation is input or output
  - What the disk address for the transfer is
  - What the memory address for the transfer is
  - What the number of sectors to be transferred is

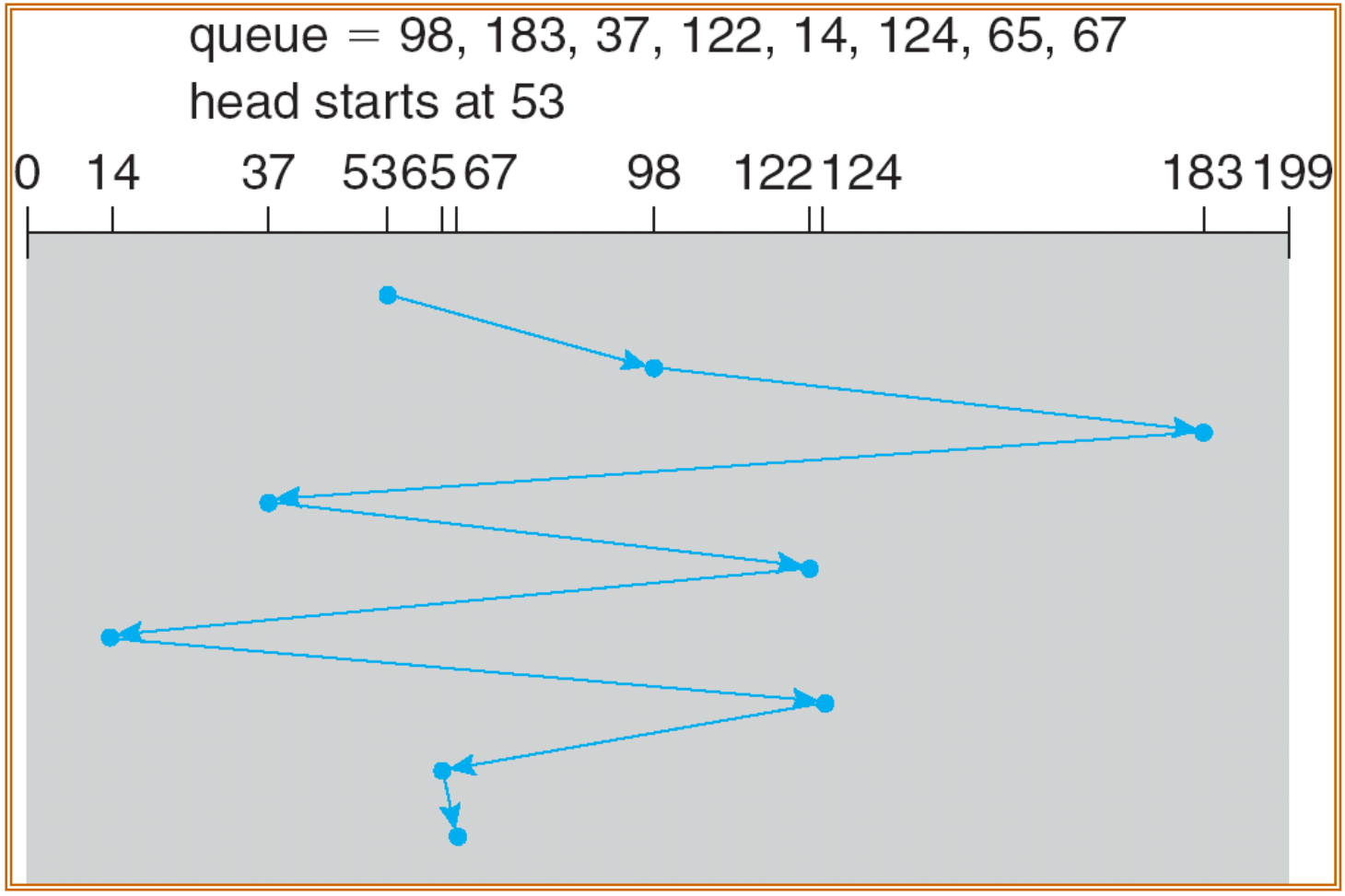
Disk Scheduling (Cont.)

- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

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

Head pointer 53

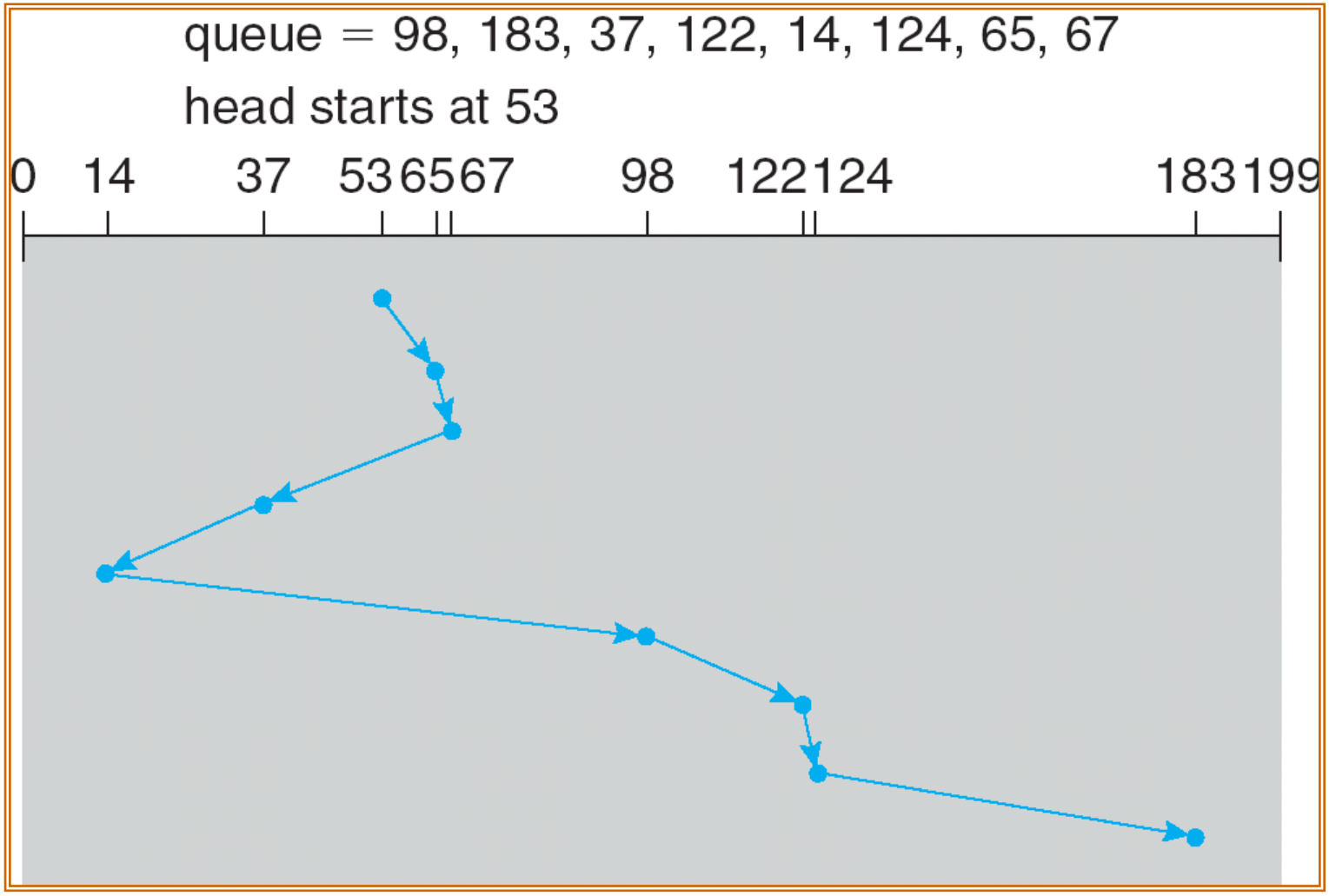
FCFS


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

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 is shown in the next slide.

SSTF (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

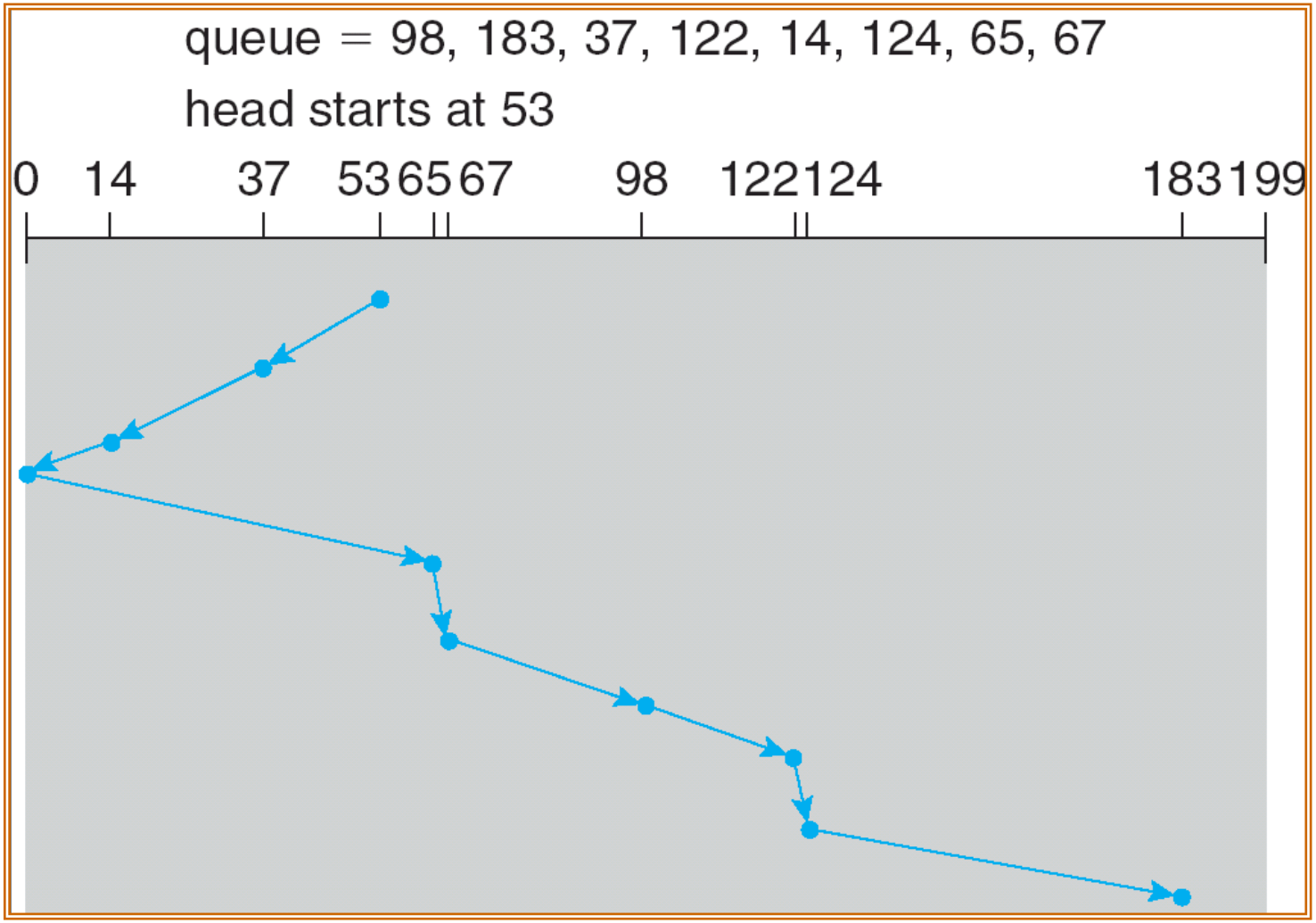
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues

- Sometimes called the *elevator algorithm*

- Illustration is shown in the next slide

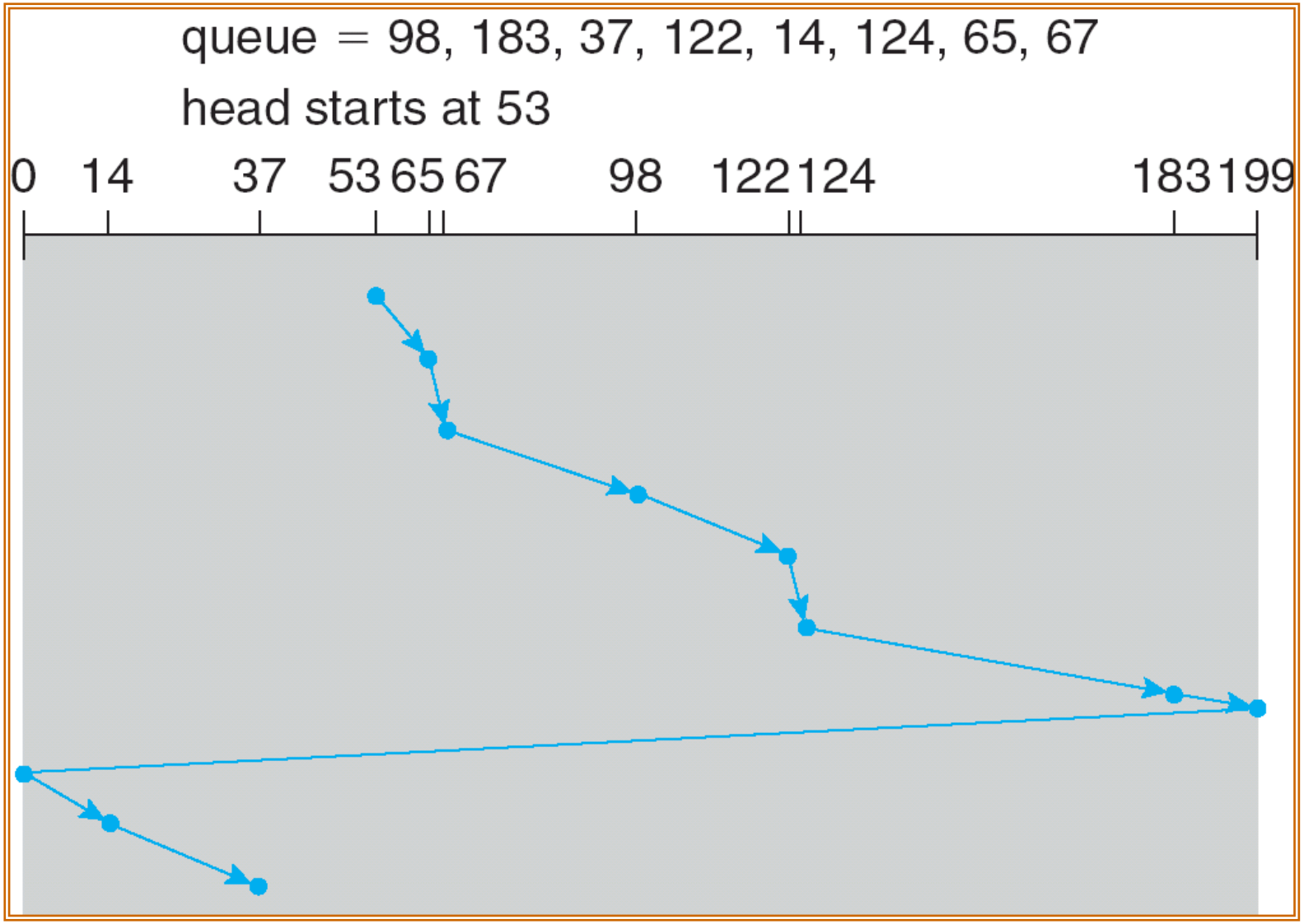
SCAN (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

C-SCAN (Cont.)

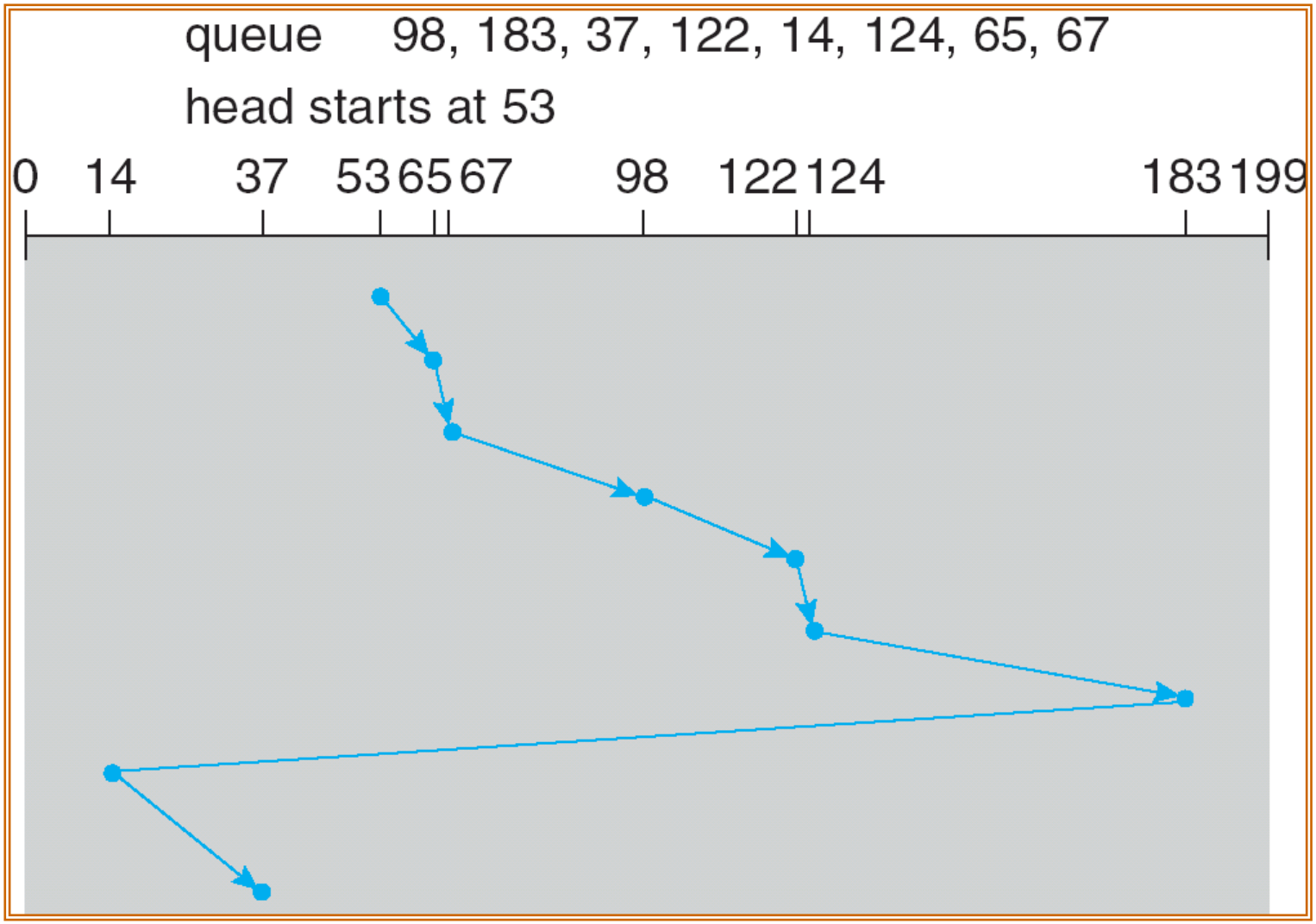

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-LOOK

- Version of C-SCAN

- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

C-LOOK (Cont.)


queue 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

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, because they are less likely to cause a starvation problem
- 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.

Disk Management

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

Swap-Space Management

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

Tertiary Storage Devices

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

Removable Disks

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

Removable Disks (Cont.)

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

WORM Disks

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

Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower

- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data

- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use

Operating System Issues

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications

- For hard disks, the OS provides two abstraction:
  - Raw device – an array of data blocks.
  - File system – the OS queues and schedules the interleaved requests from several applications.

Application Interface

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

Tape Drives

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

I/O Systems

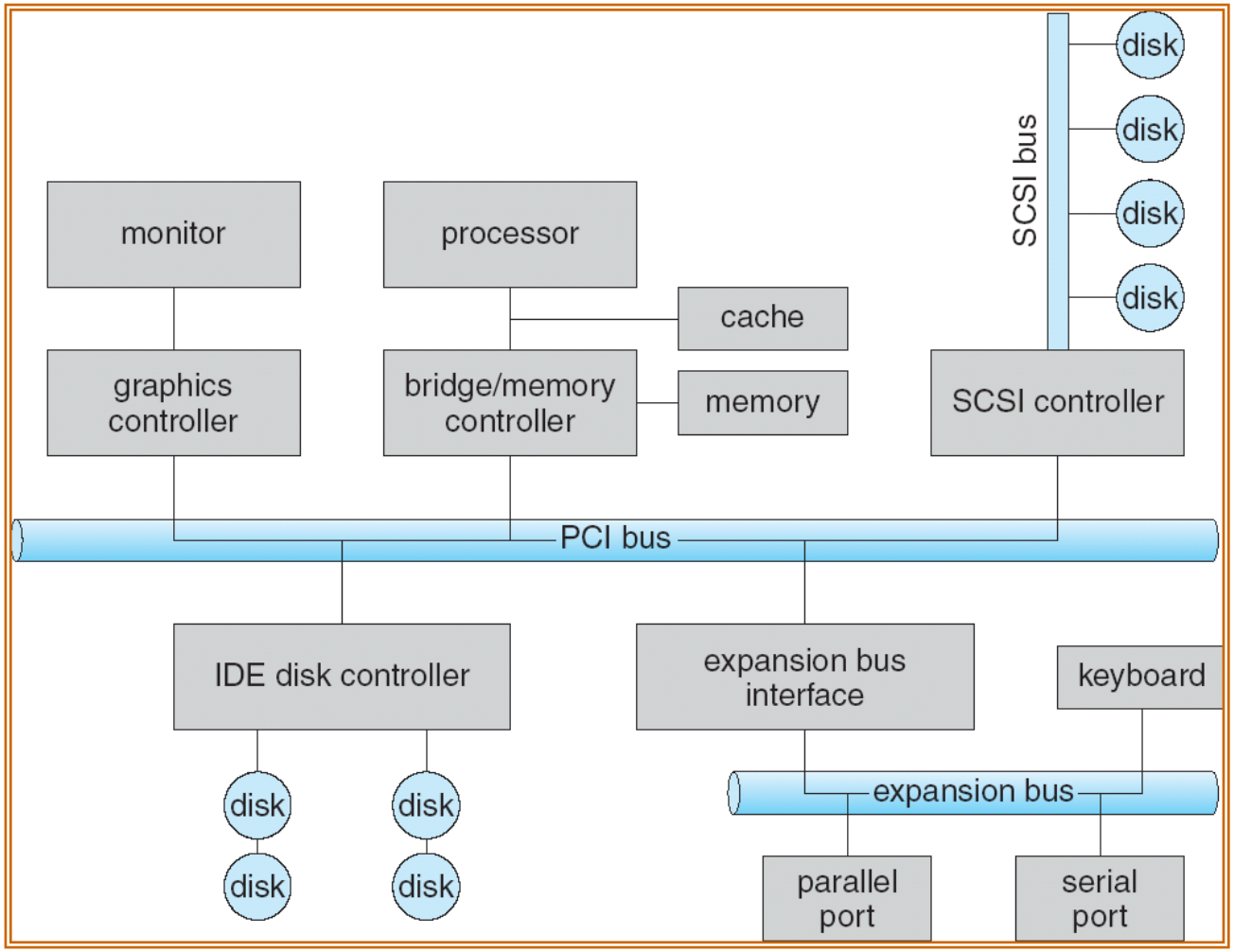
I/O Systems

- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem

I/O Hardware

- Incredible variety of I/O devices
- Common concepts
  - **Port**
  - **Bus** (**daisy chain** or shared direct access)
  - **Controller** (**host adapter**)
- I/O instructions control devices
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped** I/O

A Typical PC Bus Structure



Device I/O Port Locations on PCs (partial)

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

Polling

- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device

Interrupts

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some **nonmaskable**
- Interrupt mechanism also used for exceptions

Direct Memory Access (DMA)

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential or random-access**
  - **Sharable or dedicated**
  - **Speed of operation**
  - **read-write, read only,** or **write only**

Characteristics of I/O Devices

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

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
  - Commands include `get, put`
  - Libraries layered on top allow line editing

Clocks and Timers

- Provide the current time
- Provide the elapsed time
- Set a timer to trigger operation **X** at time **T**
- **Programmable interval timer** used for timings, periodic interrupts

Blocking and Nonblocking I/O

- **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

Kernel I/O Subsystem (contd.)

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch

Kernel I/O Subsystem

- **Caching** - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- **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 deallocation
  - Watch out for deadlock

Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
    - Determine device holding file
    - Translate name to device representation
    - Physically read data from disk into buffer
    - Make data available to requesting process
    - Return control to process

Protection

Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Language-Based Protection

Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine language-based protection systems

Goals of Protection

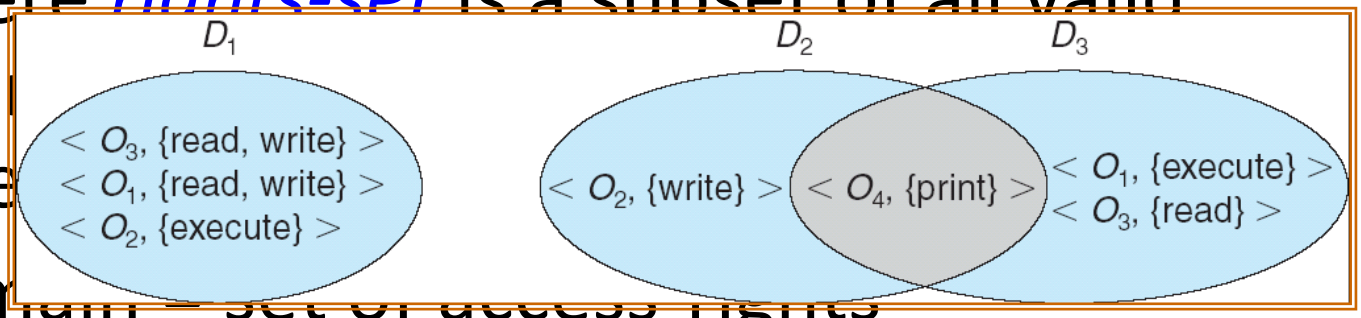
- Operating system consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.

Principles of Protection

- Guiding principle – principle of least privilege
  - Programs, users and systems should be given just enough privileges to perform their tasks

Domain Structure

- A process operates within a protection domain, which specifies the resources that it may access.
- Access-right = <*object-name*, *rights-set*> where *rights-set* is a subset of all valid ope
obje
- Don



Access Matrix

- View protection as a matrix **(*access matrix*)**
- Rows represent domains
- Columns represent objects
- *Access(i, j)* is the set of operations that a process executing in Domain_i can invoke on Object_j

Access Matrix

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- If a process in Domain *Di* tries to do “op” on object *Oj*, then “op” must be in the access matrix.

Implementation of Access Matrix

- Each column = **Access-control list (ACL)** for one object
  Defines who can perform what operation.

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

  ⋮

- Each Row = **Capability List (like a key)**
  Fore each domain, what operations allowed on what objects.

  Object 1 – Read
  Object 4 – Read, Write, Execute
  Object 5 – Read, Write, Delete, Copy

Language-Based Protection

- Specification of protection in a programming language allows for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

Protection in Java 2

- Protection is handled by the Java Virtual Machine (JVM)
- A class is assigned a protection domain when it is loaded by the JVM.
- The protection domain indicates what operations the class can (and cannot) perform.
- If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library.