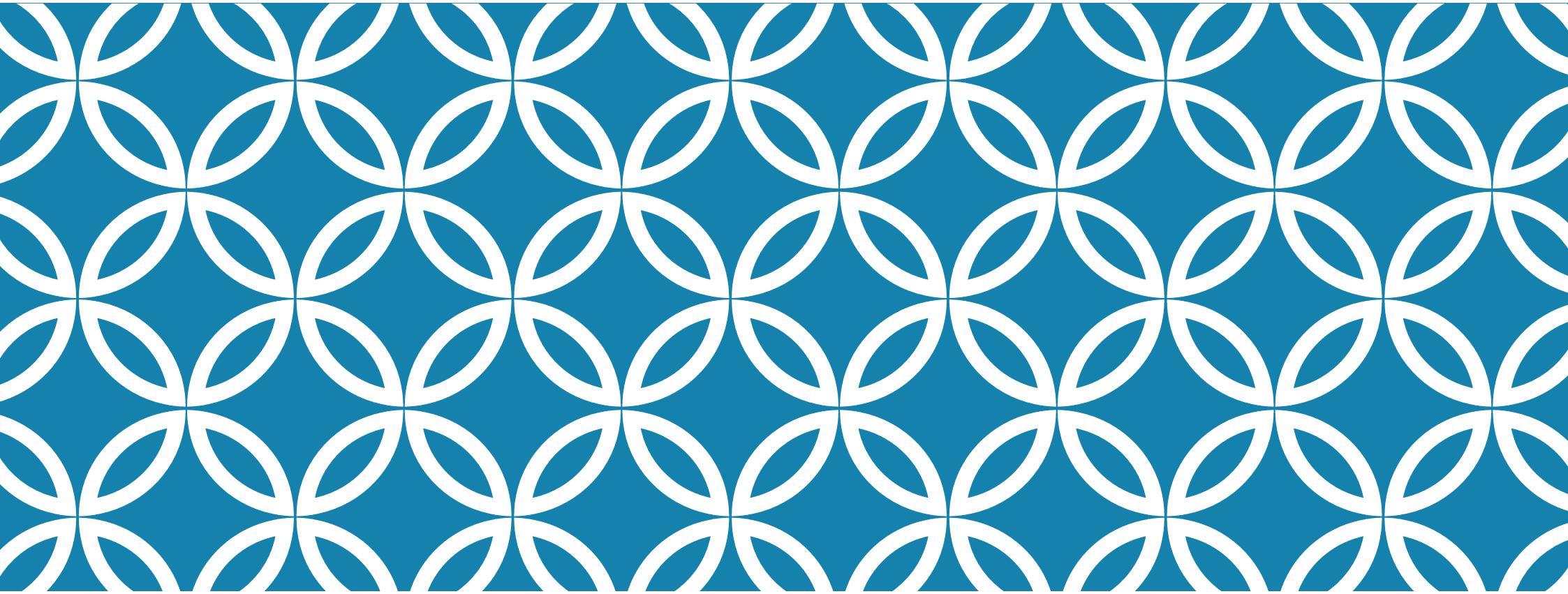


OPERATING SYSTEM

SYLLABUS

Unit 1 (6 hours)

Introduction: Operating Systems (OS) definition and its purpose, Multiprogrammed and Time Sharing Systems, OS Structure, OS Operations: Dual and Multi-mode, OS as resource manager.

Unit 2 (9 hours)

Operating System Structures: OS Services, System Calls: Process Control, File Management, Device Management, and Information Maintenance, Inter-process Communication, and Protection, System programs, OS structure- Simple, Layered, Microkernel, and Modular.

Unit 3 (10 hours)

Process Management: Process Concept, States, Process Control Block, Process Scheduling, Schedulers, Context Switch, Operation on processes, Threads, Multicore Programming, Multithreading Models, PThreads, Process Scheduling Algorithms: First Come First Served, Shortest-Job-First, Priority & Round-Robin, Process Synchronization: The critical section problem and Peterson's Solution, Deadlock characterization, Deadlock handling.

SYLLABUS CONTD..

Unit 4 (11 hours)

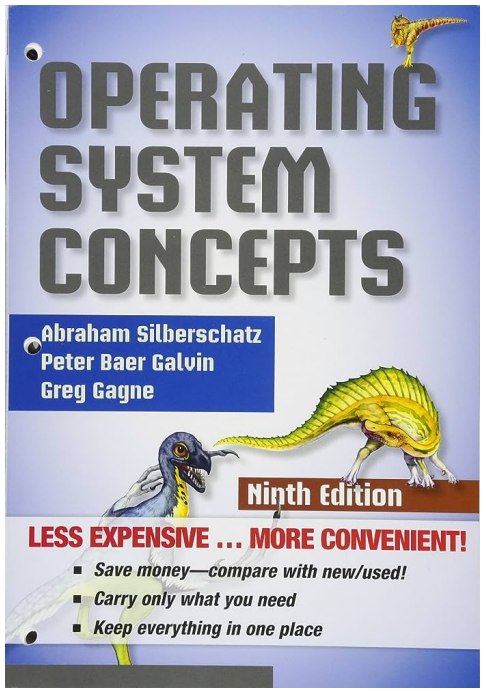
Memory Management: Physical and Logical address space, Swapping, Contiguous memory allocation strategies - fixed and variable partitions, Segmentation, Paging. Virtual Memory Management: Demand Paging and Page Replacement algorithms: FIFO Page Replacement, Optimal Page replacement, LRU page replacement.

Unit 5 (9 hours)

File System: File Concepts, File Attributes, File Access Methods, Directory Structure: Single- Level, Two-Level, Tree-Structured, and Acyclic-Graph Directories. Mass Storage Structure: Magnetic Disks, Solid-State Disks, Magnetic Tapes, Disk Scheduling algorithms: FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK Scheduling.

RECOMMENDED BOOK(S)

1. Silberschatz, A., Galvin, P. B., Gagne G. *Operating System Concepts,* 9th edition, John Wiley Publications, 2016.
2. Tanenbaum, A. S. *Modern Operating Systems*, 3rd edition, Pearson Education, 2007.

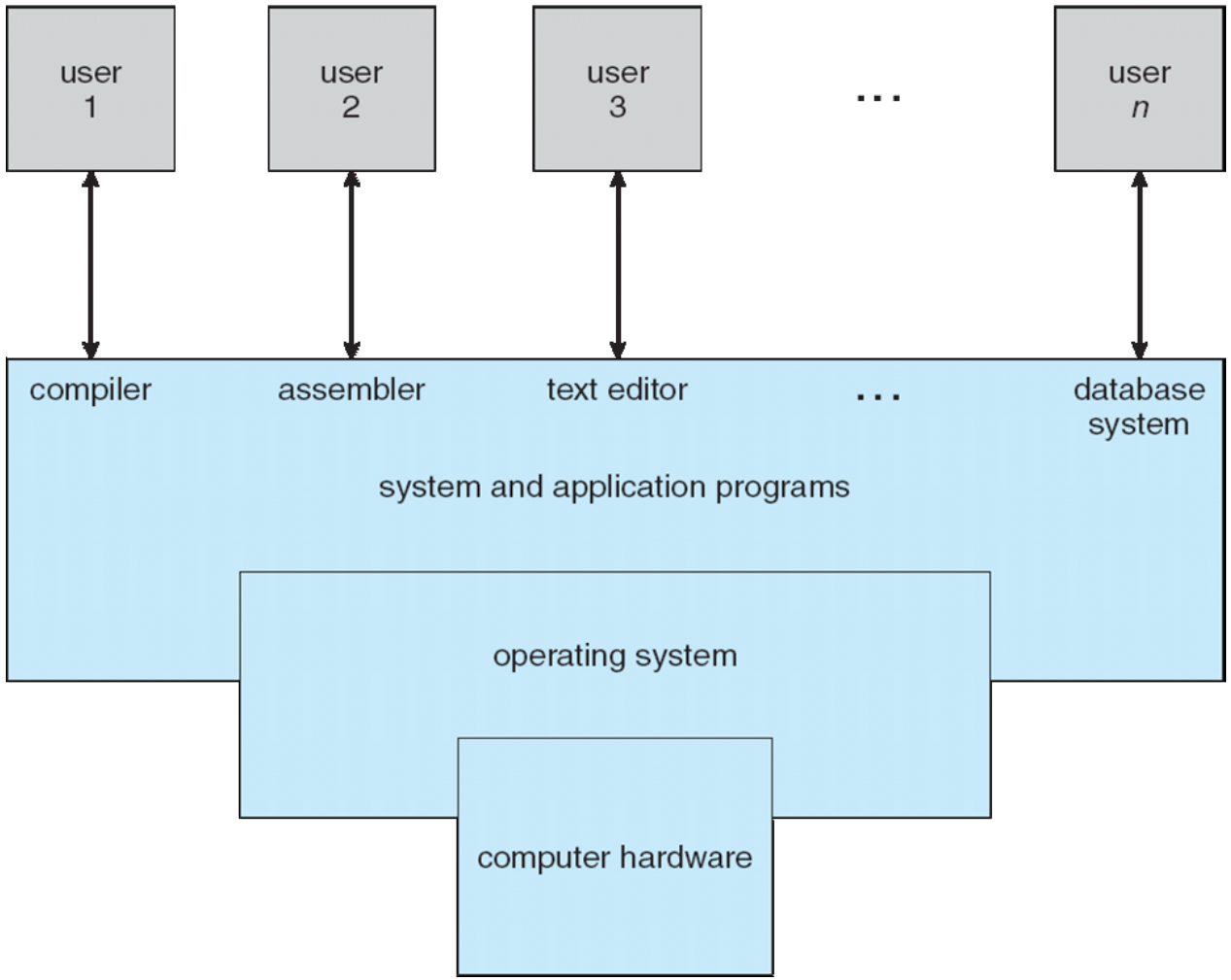


COMPUTER SYSTEM STRUCTURE

Computer system can be divided into four components:

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

FOUR COMPONENTS OF A COMPUTER SYSTEM



WHAT IS AN OPERATING SYSTEM?

- A program that acts as an intermediary between the user of a computer and the computer hardware.
- An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information.
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

WHAT OPERATING SYSTEMS DO

Depends on the point of view

Users want convenience, **ease of use** and **good performance**

- Don't care about **resource utilization**

But shared computer such as **mainframe** or **minicomputer** must keep all users happy

Handheld computers are resource poor, optimized for usability and battery life

Some computers have little or no user interface, such as embedded computers in devices and automobiles

OPERATING SYSTEM DEFINITION

OS is a **resource allocator**

- Manages all resources
- Decides between conflicting requests for efficient and fair resource use

OS is a **control program**

- Controls execution of programs to prevent errors and improper use of the computer

FUNCTIONS OF OPERATING SYSTEM

- Convenience

An OS makes a computer more convenient to use.

- Efficiency

An OS allows the computer system resources to be used in an efficient manner.

- Ability to Evolve

An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfering with service.

COMPUTER SYSTEM ARCHITECTURE

- Single Processor Systems
- Multiprocessor Systems
- Clustered Systems

SINGLE PROCESSOR SYSTEMS

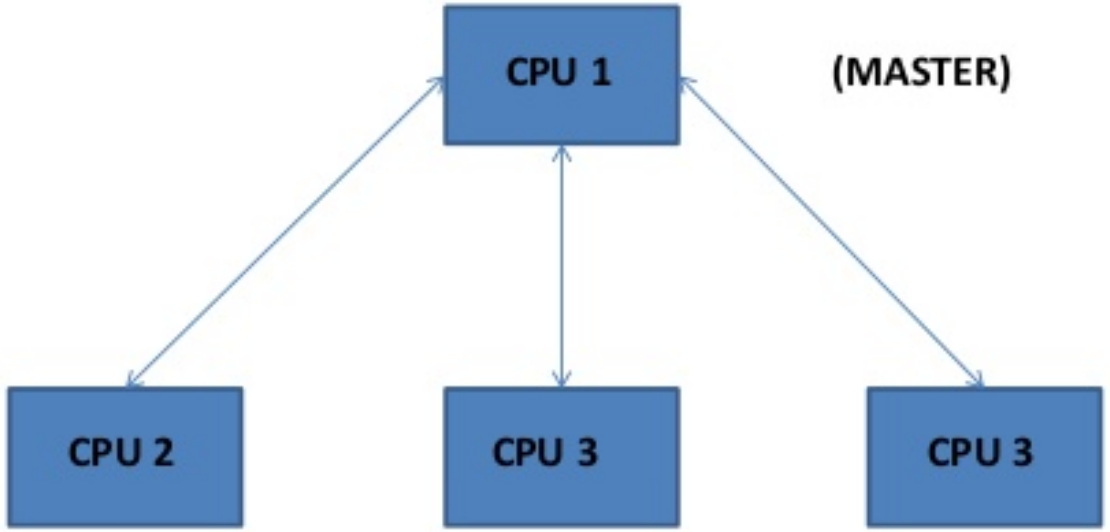
- One main CPU capable of executing a general-purpose instruction set.
- Includes special-purpose processors as well
  - I/O processors
  - Disk controller processor
- Special purpose processors run a limited instruction set and do not run user processes.
- The operating system can or cannot communicate with these processors, depending on the system architecture.

MULTIPROCESSOR SYSTEMS

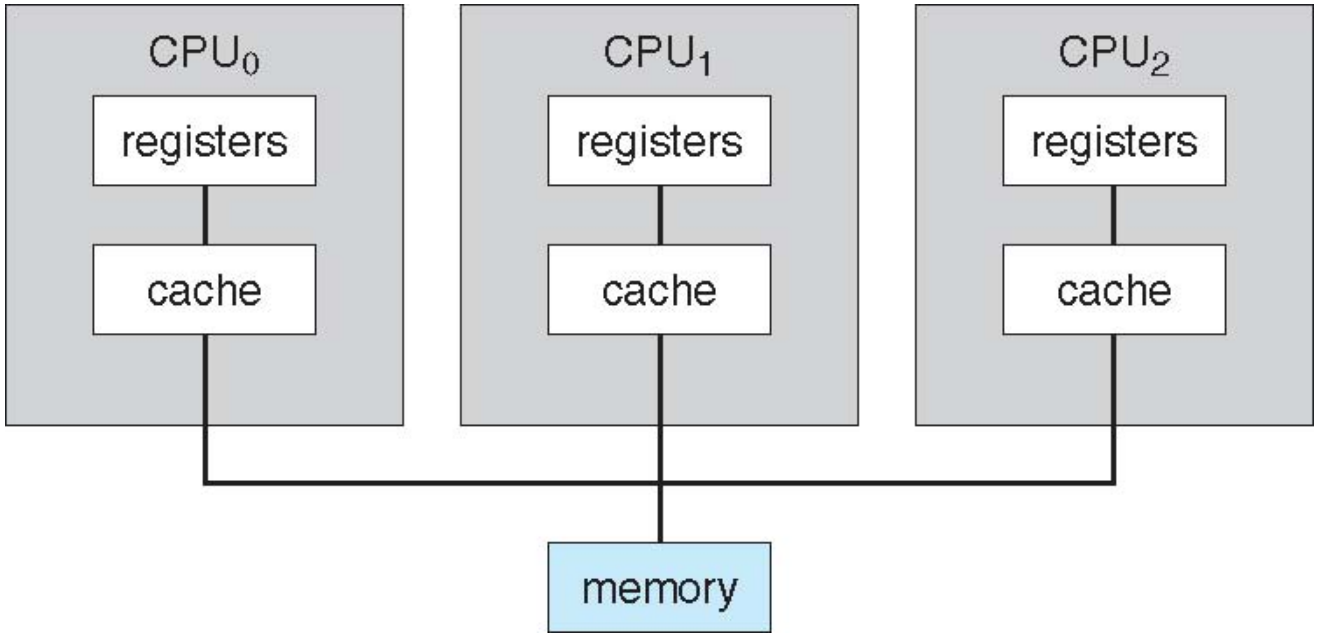
- Also known as parallel systems or tightly coupled systems.
- Two or more processors in close communication.
- Advantages for multiprocessor systems:
  - Increased throughput
  - Economy of scale
  - Increased reliability
- Multiprocessor systems are of two types:
  - Asymmetric multiprocessing- each processor is assigned a specific task by master processor.
  - Symmetric multiprocessing- each processor performs all tasks-all processors are peers

ASYMMETRIC MULTIPROCESSOR SYSTEM

Different processors do different tasks; one may be a Master and it may control other CPUs.



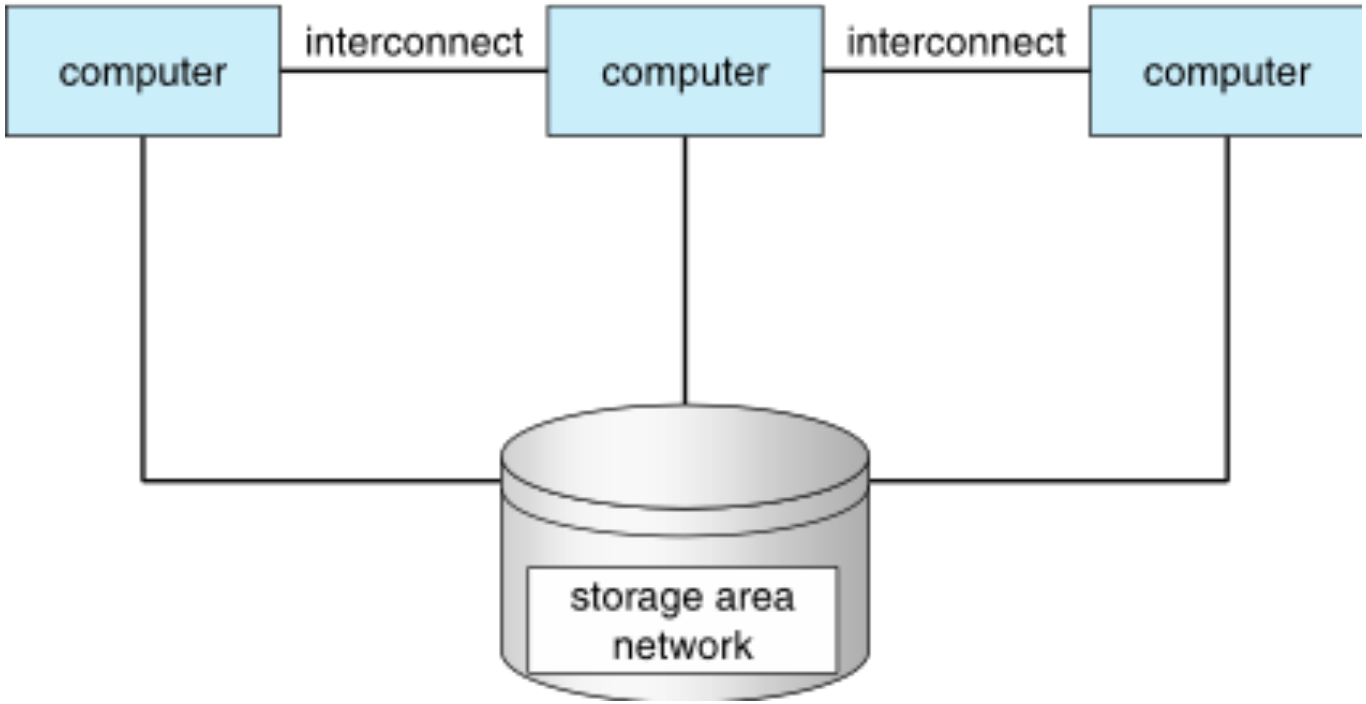
SYMMETRIC MULTIPROCESSOR SYSTEM



CLUSTERED SYSTEMS

- Clustered systems are composed of two or more individual systems(or nodes) joined together.
- Clustered systems share storage and are closely linked via a LAN.
- Provide significantly greater computational power than single processor or multiprocessor
  - They are capable of running an application concurrently on all computers in the cluster.

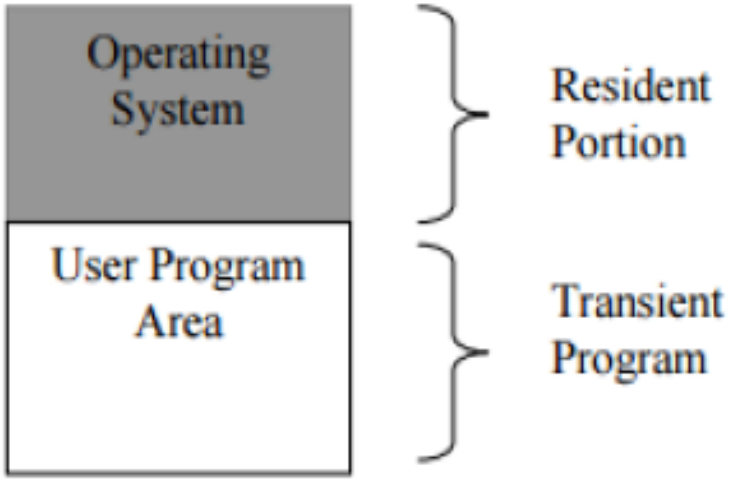
CLUSTERED SYSTEMS



BATCH SYSTEMS

- In batch processing, similar jobs were submitted to the CPU for processing and were run together.
- The main function of a batch processing system is to automatically keep executing the jobs in a batch.
- Computer system may be dedicated to a single program until its completion
- Jobs are processed in the order of submission i.e. first come first served fashion.

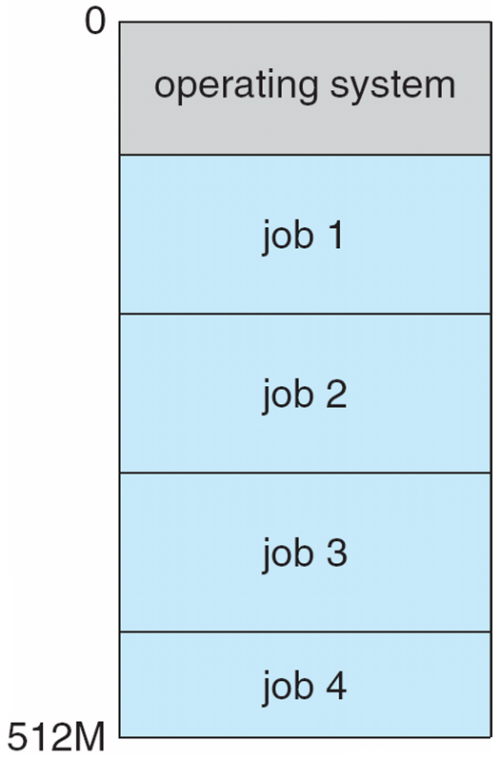
MEMORY LAYOUT FOR BATCH SYSTEMS



MULTIPROGRAMMING

- When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system.
- Multiprogramming assumes a single processor that is being shared.
- It increases CPU utilization by organizing jobs so that the CPU always has one to execute.
- The operating system keeps several jobs in memory at a time.
- The operating system picks and begins to execute one of the job in the memory.

MEMORY LAYOUT FOR MULTIPROGRAMMING



TIME SHARING SYSTEMS

- Time sharing, or multitasking, is a logical extension of multiprogramming.
- A time-shared operating system allows the many users to share the computer simultaneously.
- Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.
- As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.

REAL TIME EMBEDDED SYSTEMS

- Embedded systems are having little or no user interface
- Embedded systems are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens.
- They almost always run real-time operating systems.
  - A real time system has well defined, fixed time constraints.
  - Processing must be done within the defined constraints, or the system will fail.
  - There are two types of real systems: Hard real time and soft real time

MULTIMEDIA SYSTEMS

- Multimedia data consist of audio and video files as well as conventional files.
- These data differ from conventional data in that multimedia data must be delivered according to certain time restrictions.
- Multimedia files include audio files such as MP3, DVD movies, video conferencing, and short video clips or live webcasts of speeches or sporting events.
- These need not be either audio or video; rather a combination of both.

HANDHELD SYSTEMS

- Handheld systems include personal digital assistants(PDAs), such as Palm and Pocket-PCs, and cellular telephones.
- One of the main challenges of these systems is limited size of such devices.
  - Because of their size, most handheld systems have small amounts of memory, slow processors, and small display screens.
- Some handheld devices use wireless technology, such as BlueTooth, allowing remote access to e-mail and Web browsing.
- Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability.

TWO VIEWS OF OPERATING SYSTEM

User View

- Refers to the interface being used.
- Such systems are designed:
  - To **maximize the work** that the user is performing.
  - **ease of use** with some attention paid to performance, and none paid to resource utilization.

System View

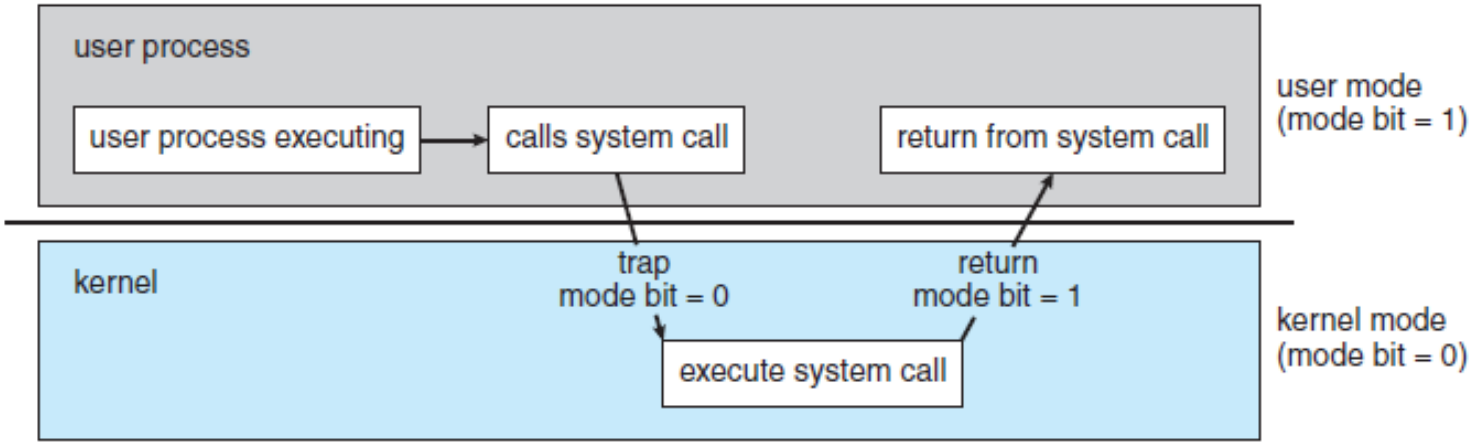
- Resource allocator
  - managing hardware and software efficiently
  - decides between conflicting requests, controls execution of programs etc.

OPERATING SYSTEM OPERATIONS

- Modern operating systems are interrupt driven.
- Events are almost always signaled by the occurrence of an interrupt or a trap.
  - A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program.
- For each type of interrupt, separate segments of code in the operating system determine what action should be taken.
- An interrupt service routine is provided to deal with the interrupt.

OPERATING SYSTEM OPERATIONS

- In order to ensure the proper execution of the operating system, there are two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode).
  - Privileged instructions are run in kernel mode.
- A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). Example of dual mode: Windows 7.



OPERATING SYSTEM OPERATIONS

- The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode).
- CPUs that support virtualization frequently have a separate mode to indicate when the virtual machine manager (VMM)—and the virtualization management software—is in control of the system.
- In this mode, the VMM has more privileges than user processes but fewer than the kernel.
- Example of multi-mode: The Intel 64 family of CPUs supports four privilege levels, and supports virtualization but does not have a separate mode for virtualization.

OPERATING SYSTEM OPERATIONS

- It must be ensured that the operating system maintains control over the CPU.
- One cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system.
- To accomplish this goal, a timer can be used.
  - A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable.
- A timer can be used to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run.

STORAGE SYSTEM

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

QUIZ

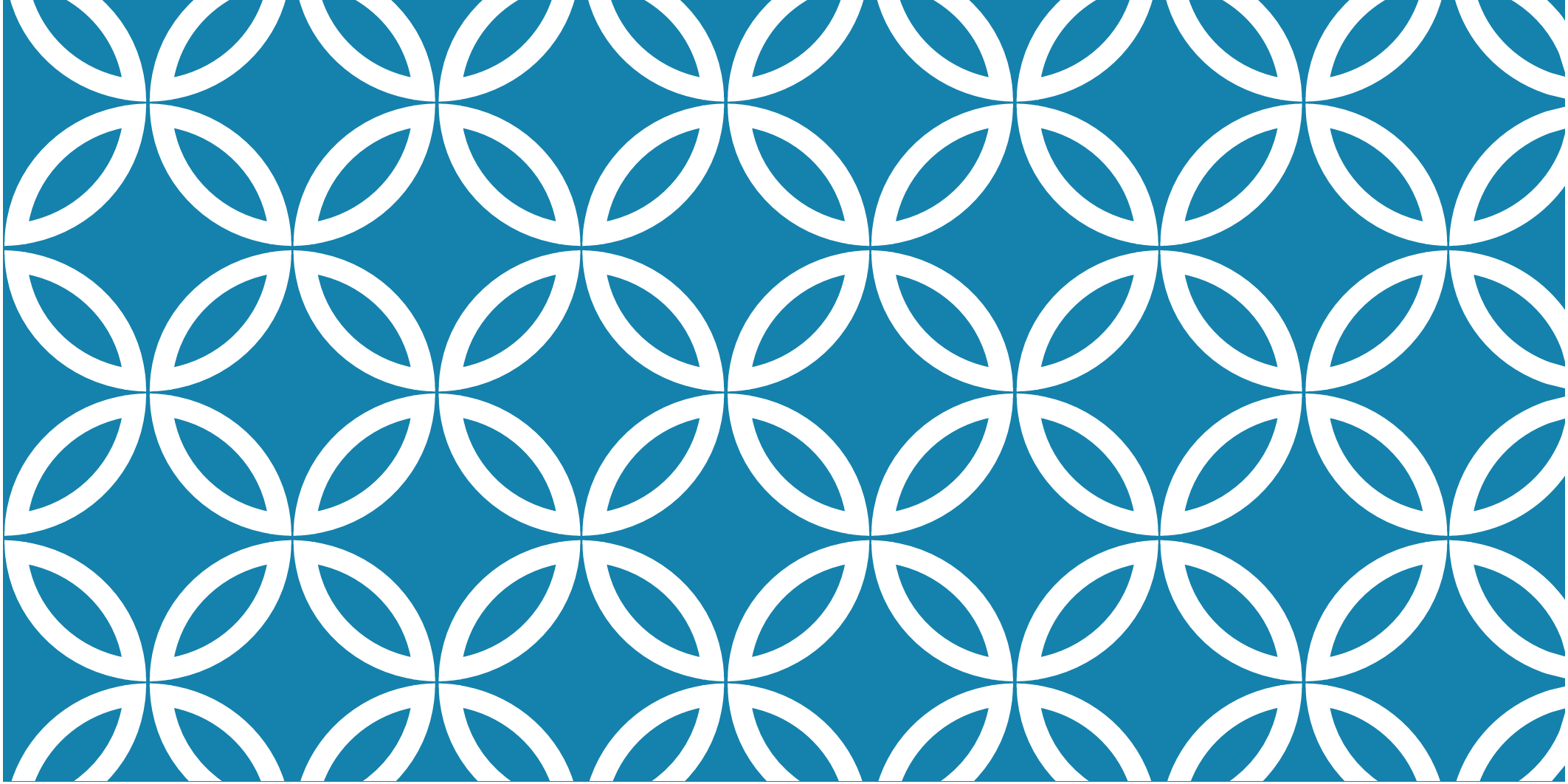
1. Which is built directly on the hardware?
   A. Application Software
   B. Operating System
   C. Database System

2. The primary purpose of an operating system is:
   A. To make the most efficient use of the computer hardware
   B. To allow people to use the computer
   C. To keep systems programmers employed

3. What does the "X" stand for in OS X?
   A. extreme
   B. extensible
   C. ten

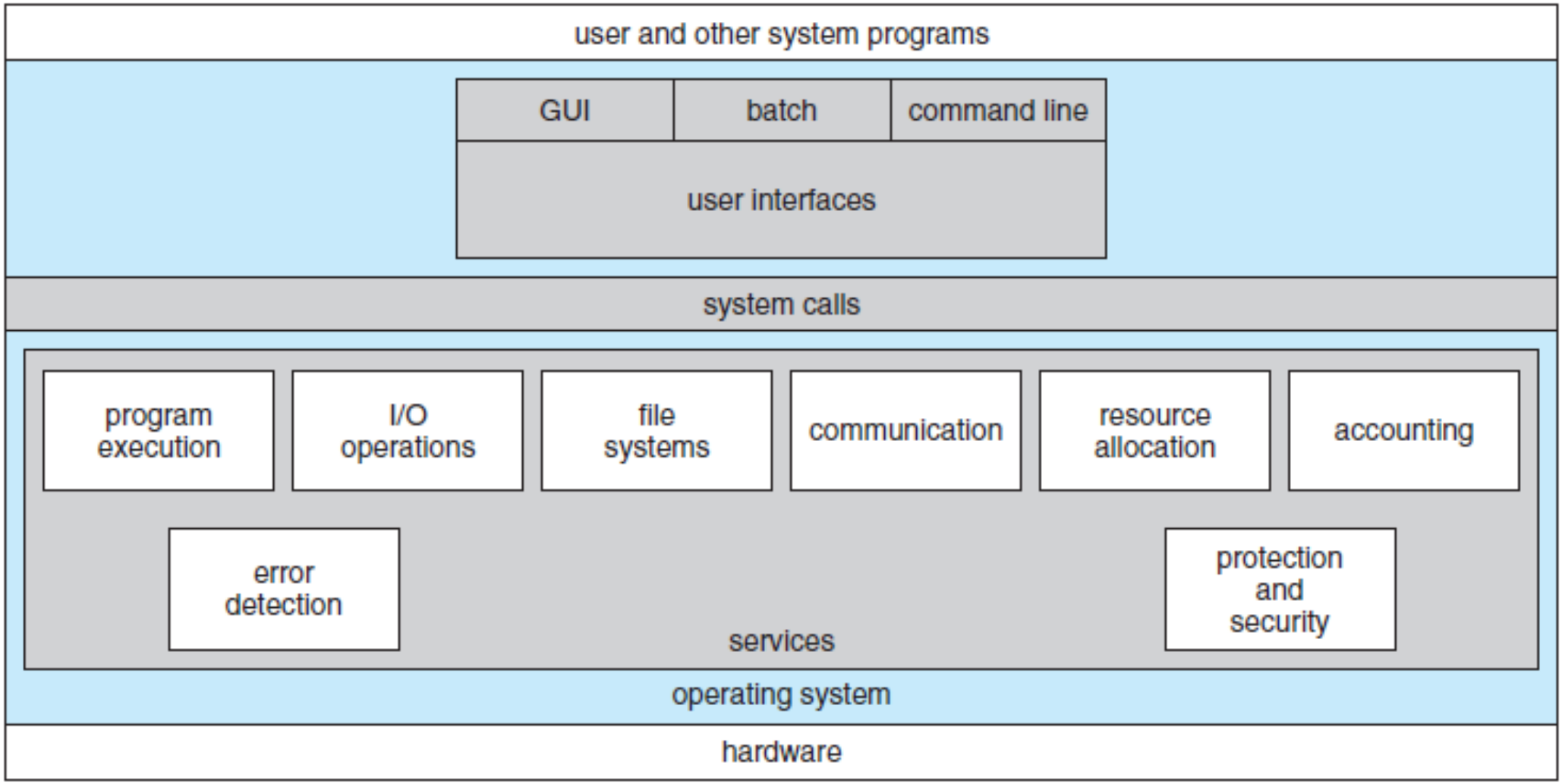
QUIZ

4. Multiprogramming systems:
   A. Are extension of time sharing systems
   B. Execute each job faster
   C. Keeps several jobs in memory at a time
5. Handheld systems have
   A. Real time operating system
   B. Special purpose processors
   C. Limited size



OPERATING SYSTEM STRUCTURES

OPERATING SYSTEM SERVICES



OPERATING SYSTEM USER INTERFACE- CLI

- CLI or **command interpreter** allows direct command entry
- On systems with multiple command interpreters to choose from, the interpreters are known as shells.
  - Example shells in UNIX
- Two ways to execute commands
  - Implemented in kernel only
  - Implemented through system programs

OPERATING SYSTEM USER INTERFACE- GUI

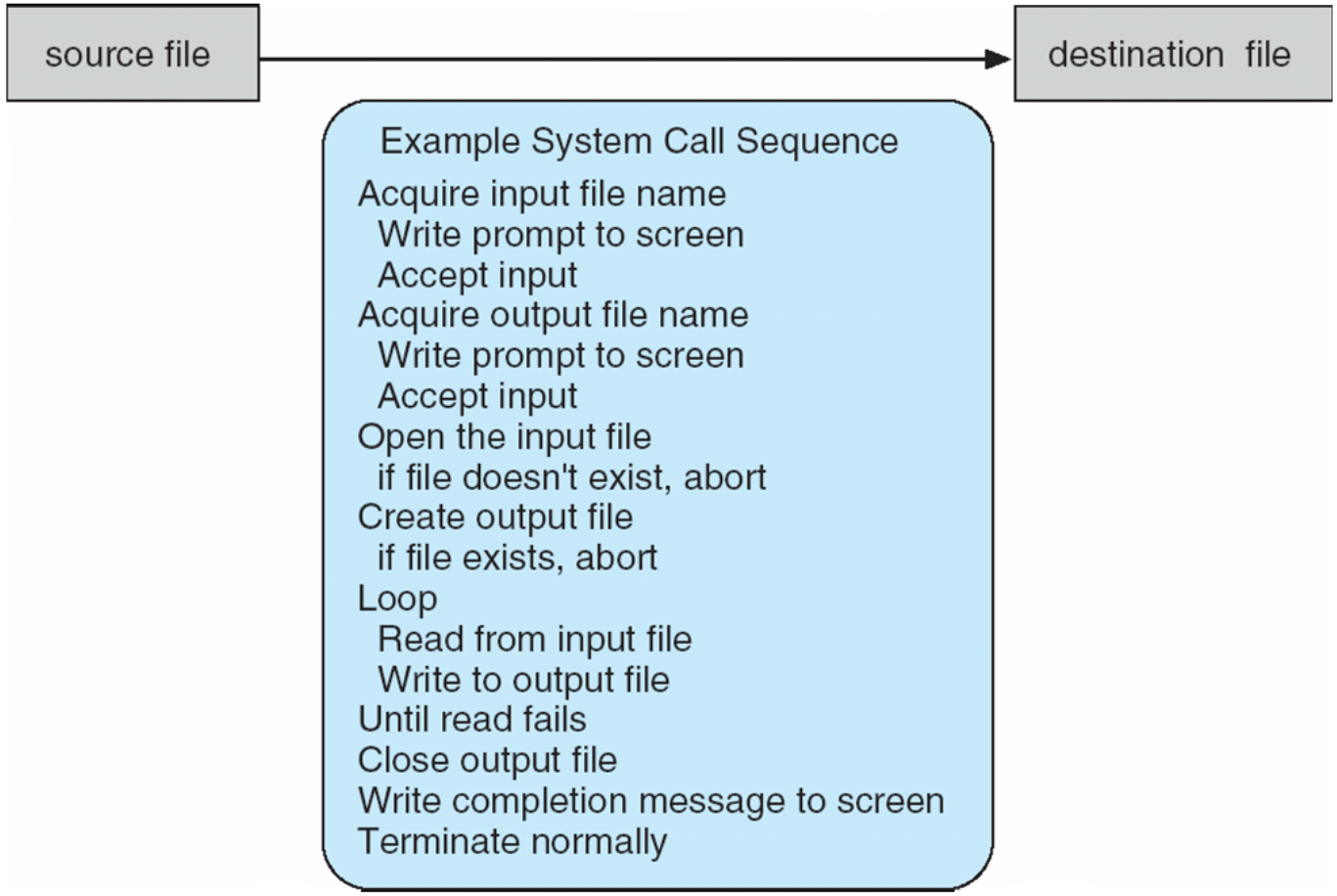
- Rather than entering commands, users employ a mouse and menu based system.
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
- First GUI by the Xerox in 1973.
- Many systems now include both CLI and GUI interfaces

SYSTEM CALLS

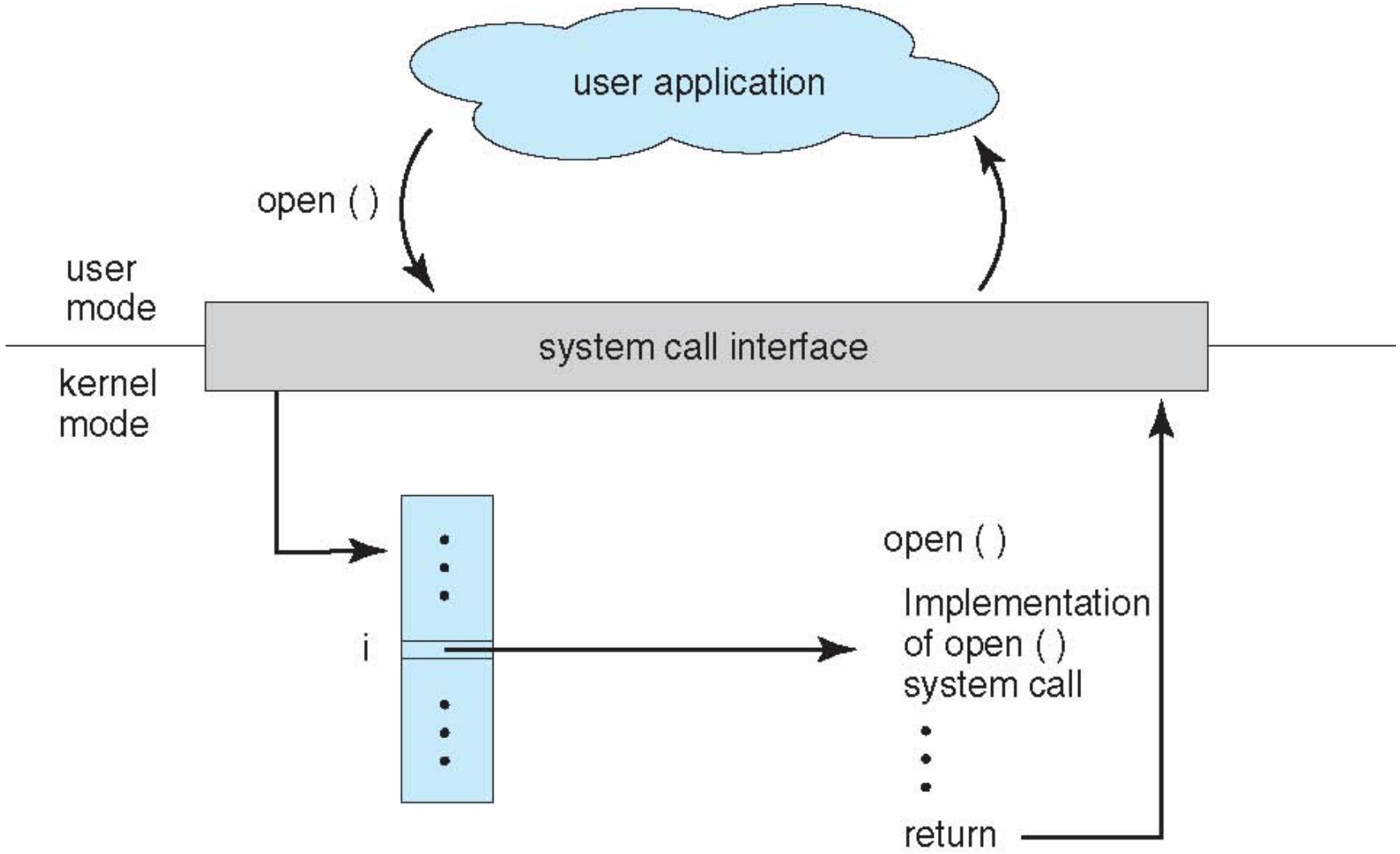
- Programming interface to the services provided by the OS.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use.
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

EXAMPLE OF SYSTEM CALLS

- System call sequence to copy the contents of one file to another file.



API – SYSTEM CALL – OS RELATIONSHIP



TYPES OF SYSTEM CALLS

Process control

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- Dump memory if error

File management

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

TYPES OF SYSTEM CALLS

Device management

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

Protection

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

TYPES OF SYSTEM CALLS

Information maintenance

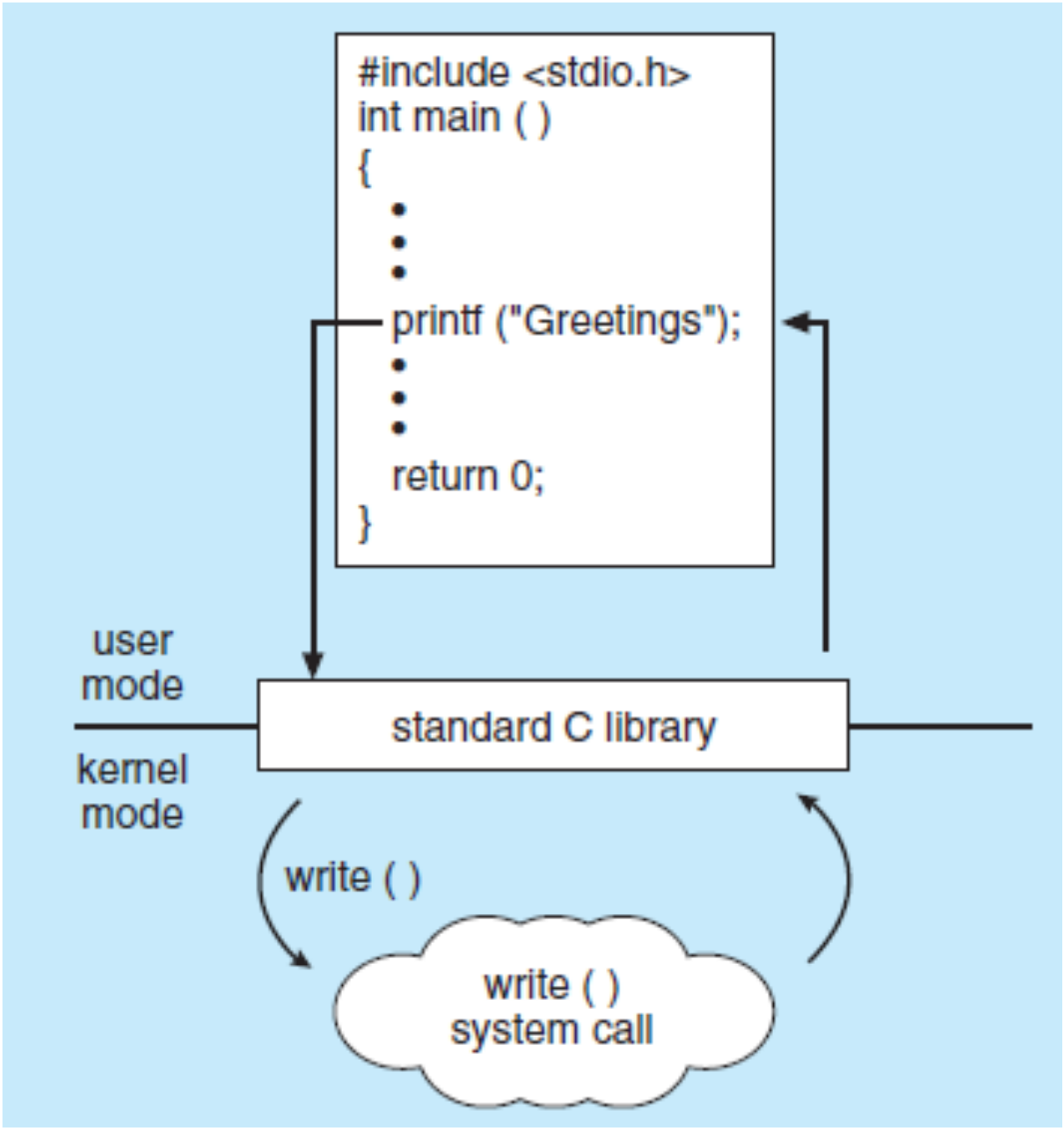
- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

Communications

- create, delete communication connection
- send, receive messages
- **Shared-memory model** create and gain access to memory regions
- transfer status information
- attach and detach remote devices

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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


#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
user mode
kernel mode
standard C library
write ( )
write ( )
system call

SYSTEM PROGRAMS

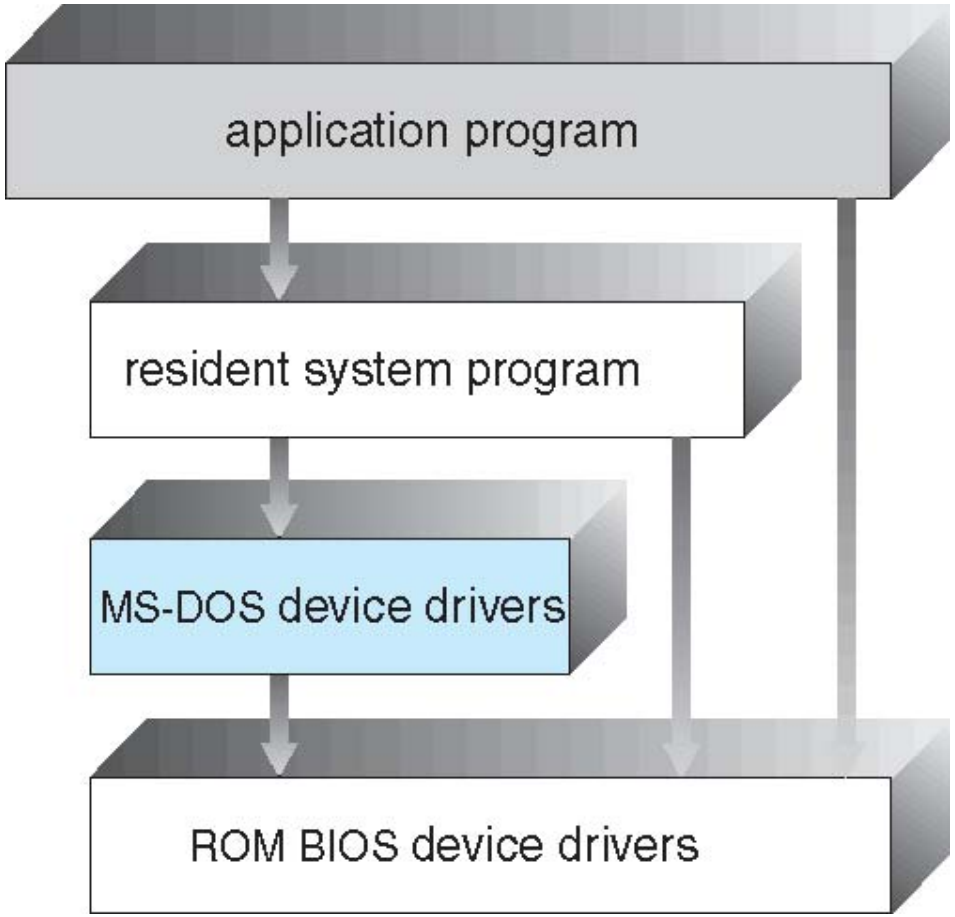
- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background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

OPERATING SYSTEM STRUCTURE

- General-purpose OS is very large program.
- Various ways to structure OS
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – Windows NT
  - Microkernel -Mach

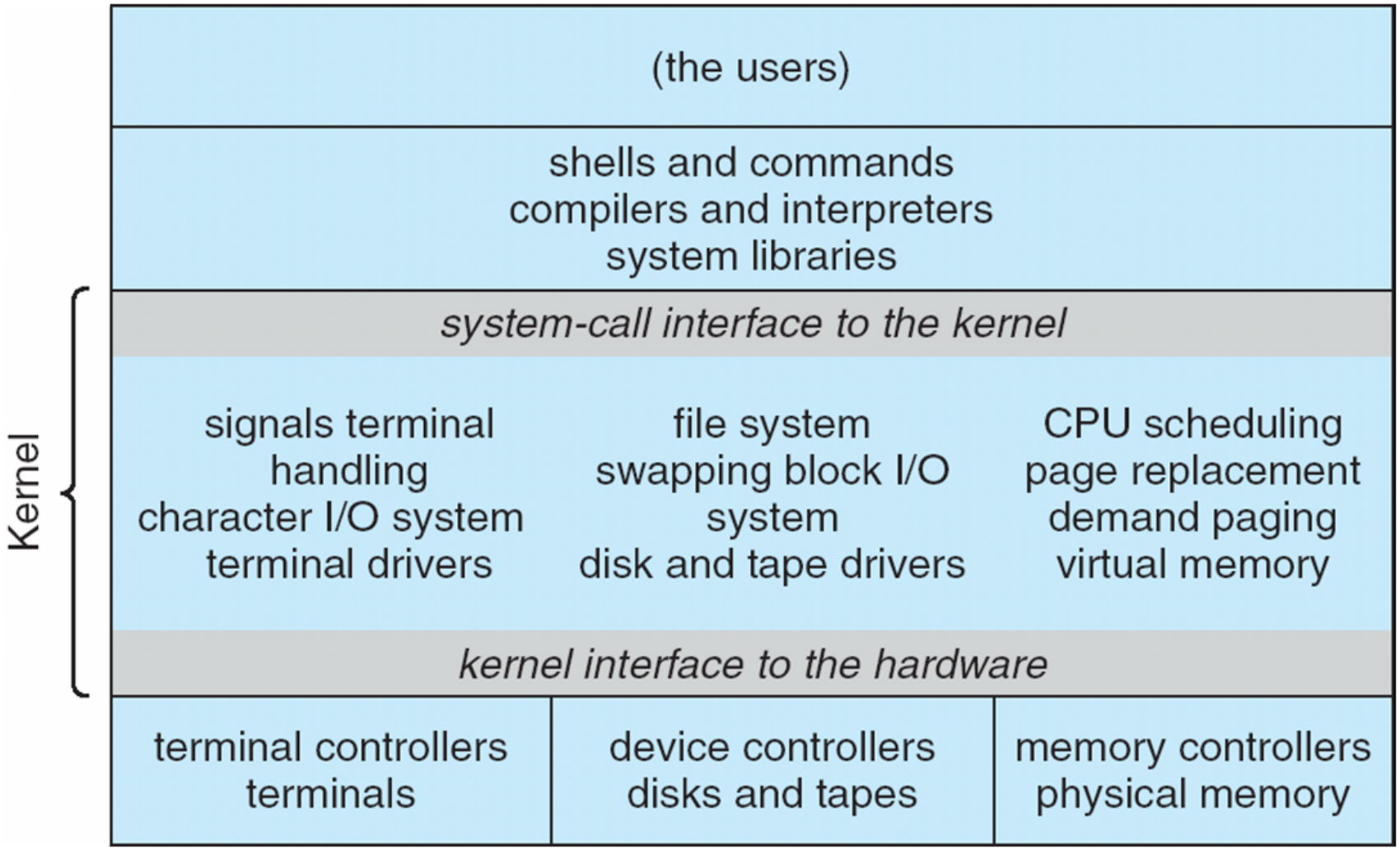
SIMPLE STRUCTURE -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space.
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.
  - Leaves MS-DOS vulnerable to malicious programs, causing entire system to crash when user programs fail.



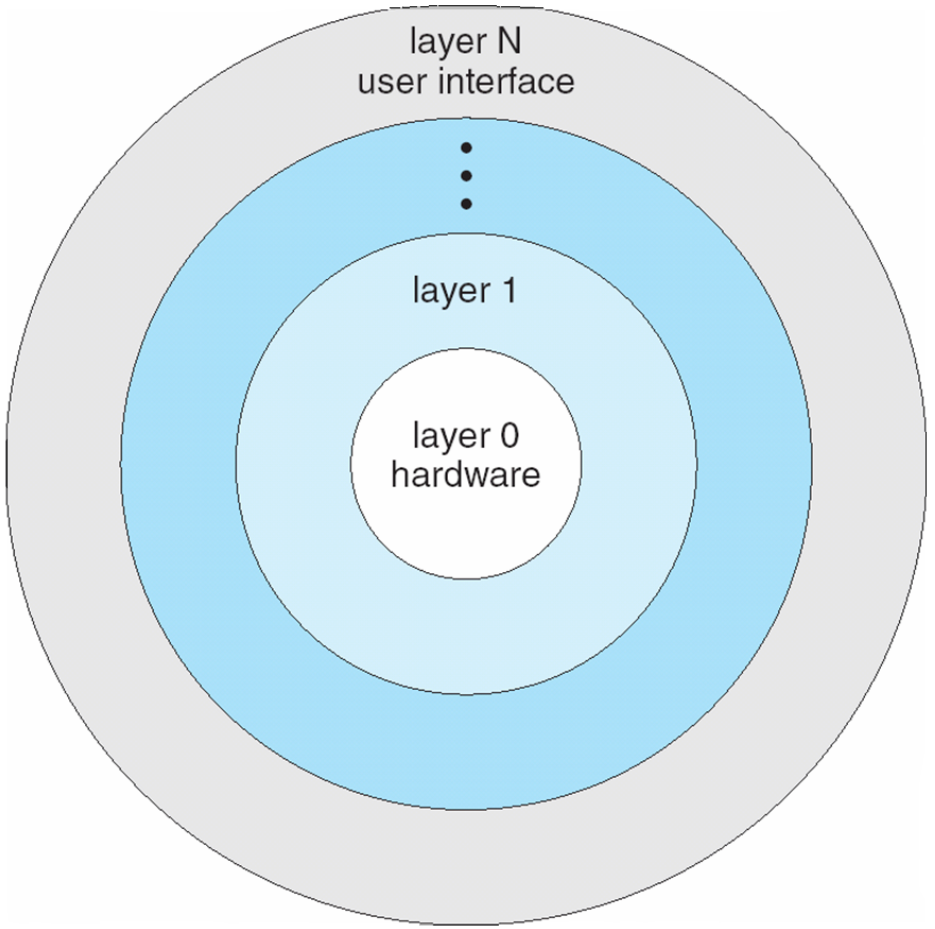
TRADITIONAL UNIX SYSTEM STRUCTURE

Beyond simple but not fully layered



LAYERED APPROACH

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

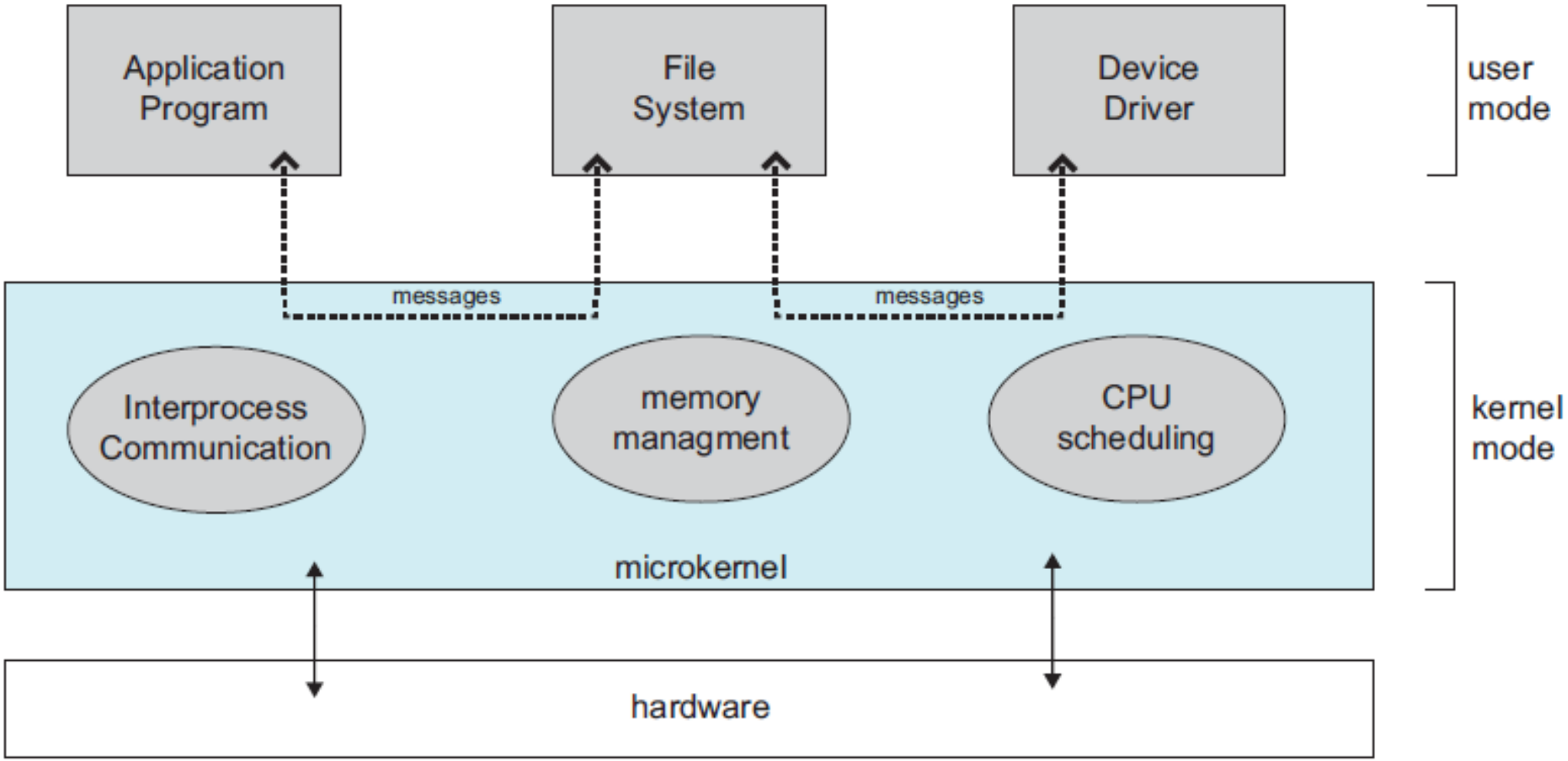


Drawback: The major difficulty with the layered approach involves appropriately defining the various layers.

MICROKERNEL SYSTEM STRUCTURE

- Moves as much from the kernel into user space.
- Removing all non-essential components from the kernel and implementing them as system and user programs.
- Smaller kernel
- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

MICROKERNEL SYSTEM STRUCTURE



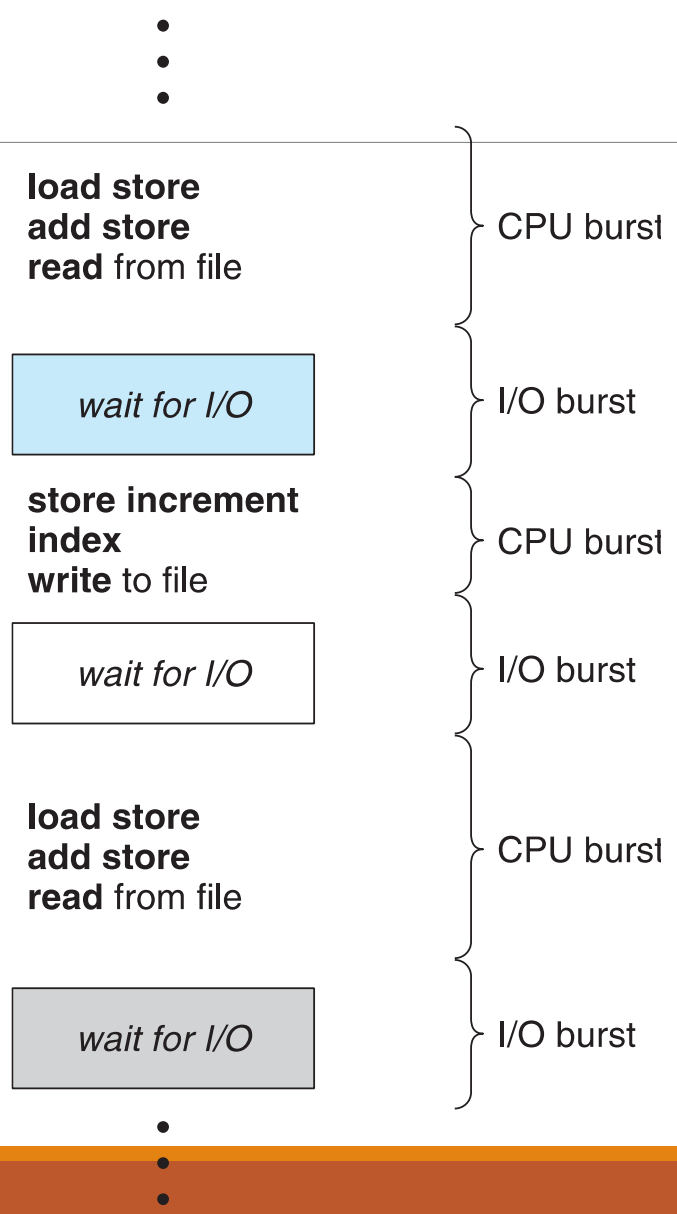
Process Scheduling

Basic Concepts

Maximum CPU utilization obtained with multiprogramming

CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait

CPU burst followed by **I/O burst**



CPU Scheduler

Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Dispatcher

Dispatcher module gives control of the CPU to the process selected.

Dispatch latency – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

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

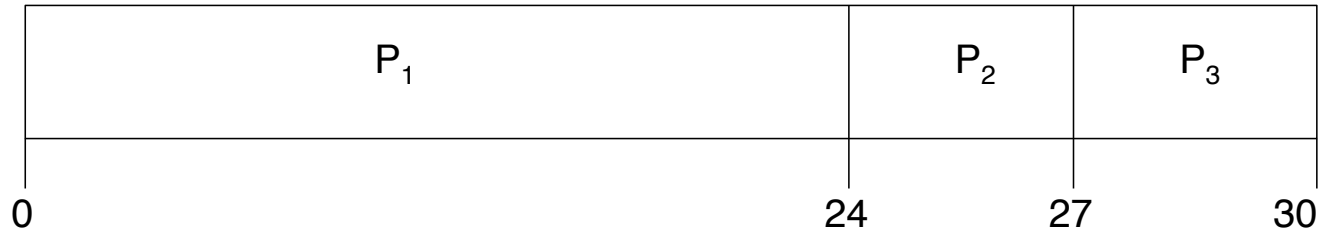
Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

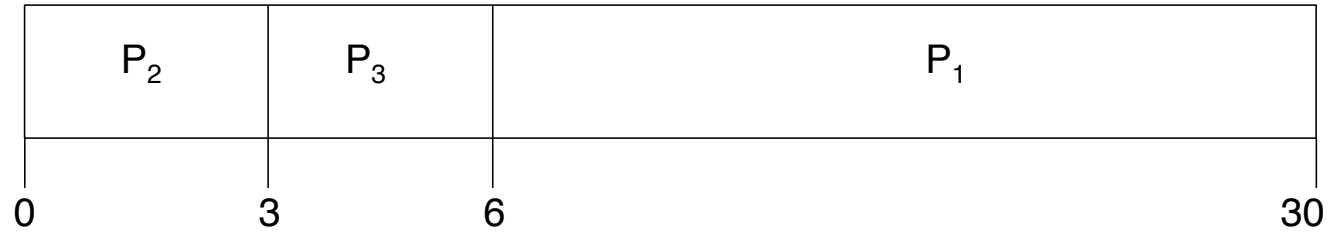
Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

The Gantt chart for the schedule is:



Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$

Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

Convoy effect - short process behind long process

Shortest-Job-First (SJF) Scheduling

Associate with each process the length of its next CPU burst

- Use these lengths to schedule the process with the shortest time

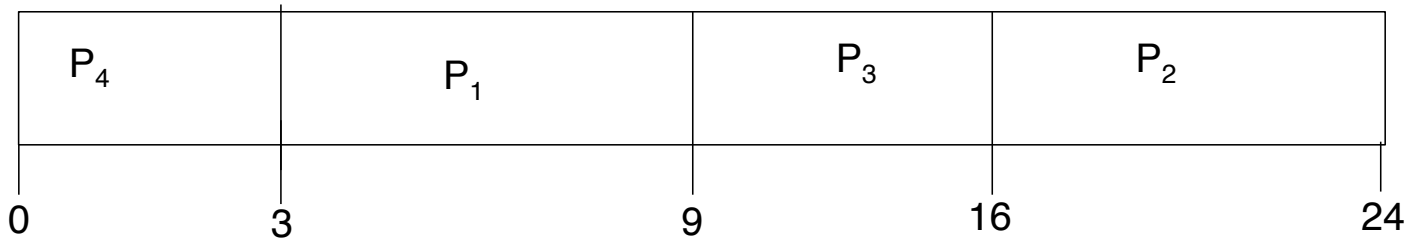
SJF is optimal – gives minimum average waiting time for a given set of processes

- The difficulty is knowing the length of the next CPU request
- Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



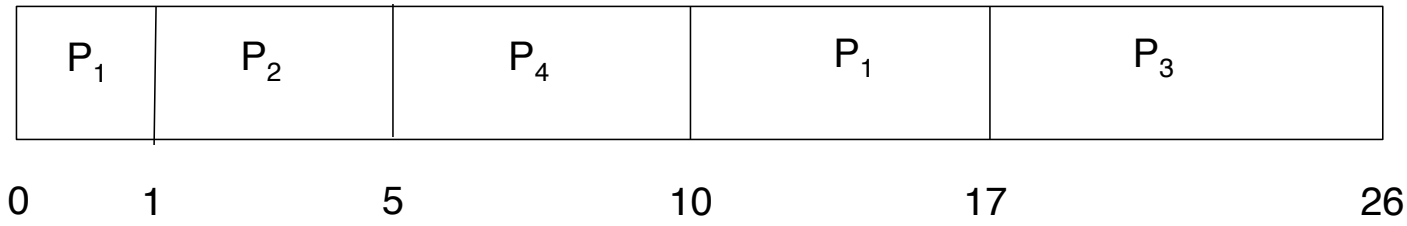
Average waiting time = (0 + 3 + 16 + 9) / 4 = 7

Example of Shortest-remaining-time-first

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

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive SJF Gantt Chart



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

Priority Scheduling

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

SJF is priority scheduling where priority is the inverse of predicted next CPU burst time.

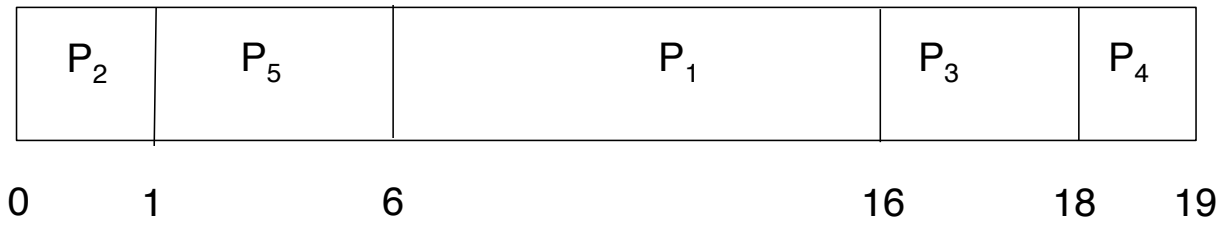
Problem ≡ **Starvation** – low priority processes may never execute

Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority scheduling Gantt Chart



Average waiting time = 8.2 msec

Round Robin (RR) Scheduling

Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

Timer interrupts every quantum to schedule next process

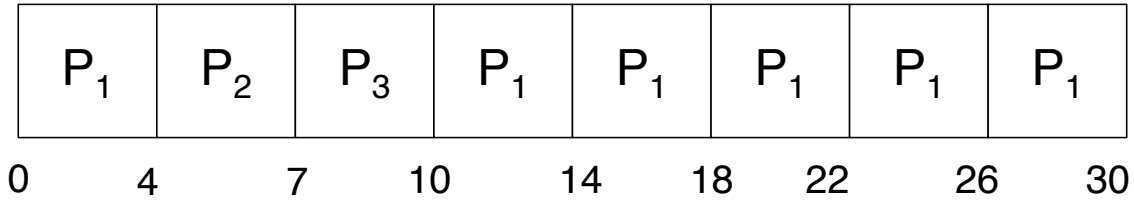
Performance

- q large $\Rightarrow$ FIFO
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

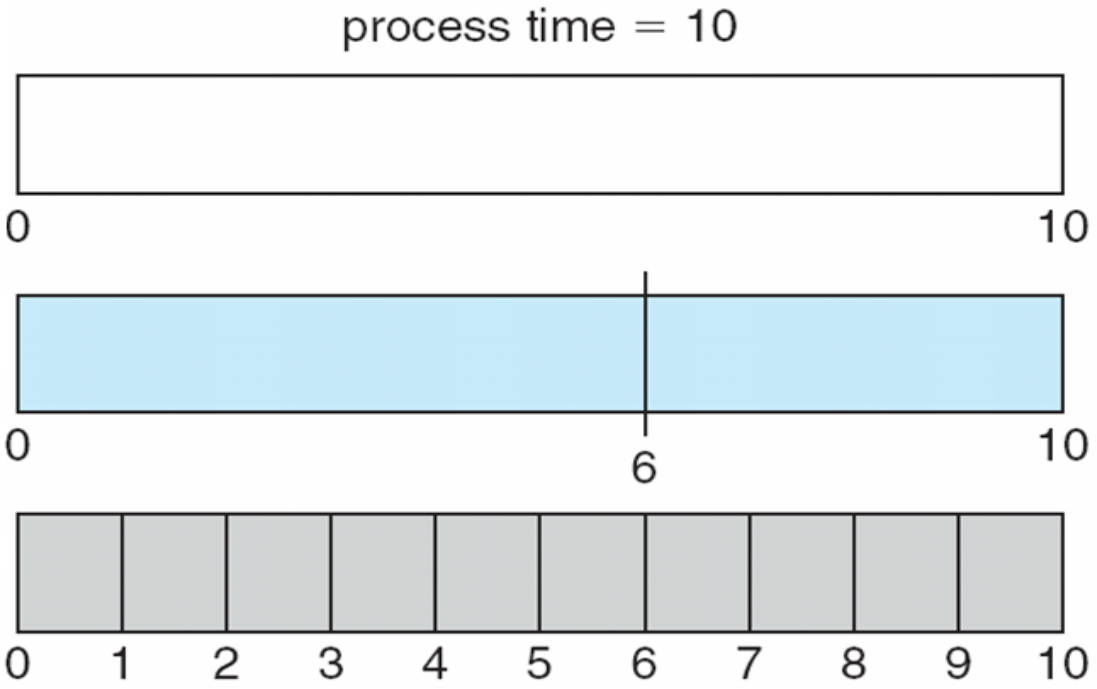
The Gantt chart is:



Typically, higher average turnaround than SJF, but ***better response***

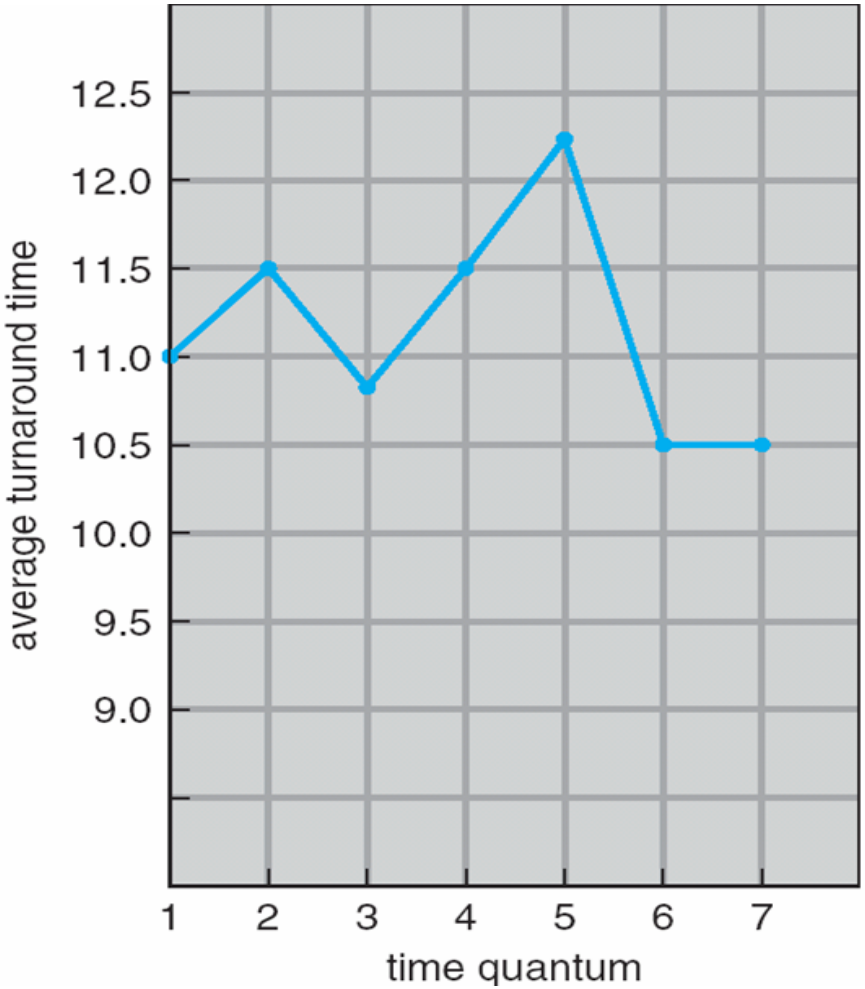
q should be large compared to context switch time

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Multilevel Queue

Ready queue is partitioned into separate queues, e.g.:

- **foreground** (interactive)
- **background** (batch)

Process permanently in a given queue.

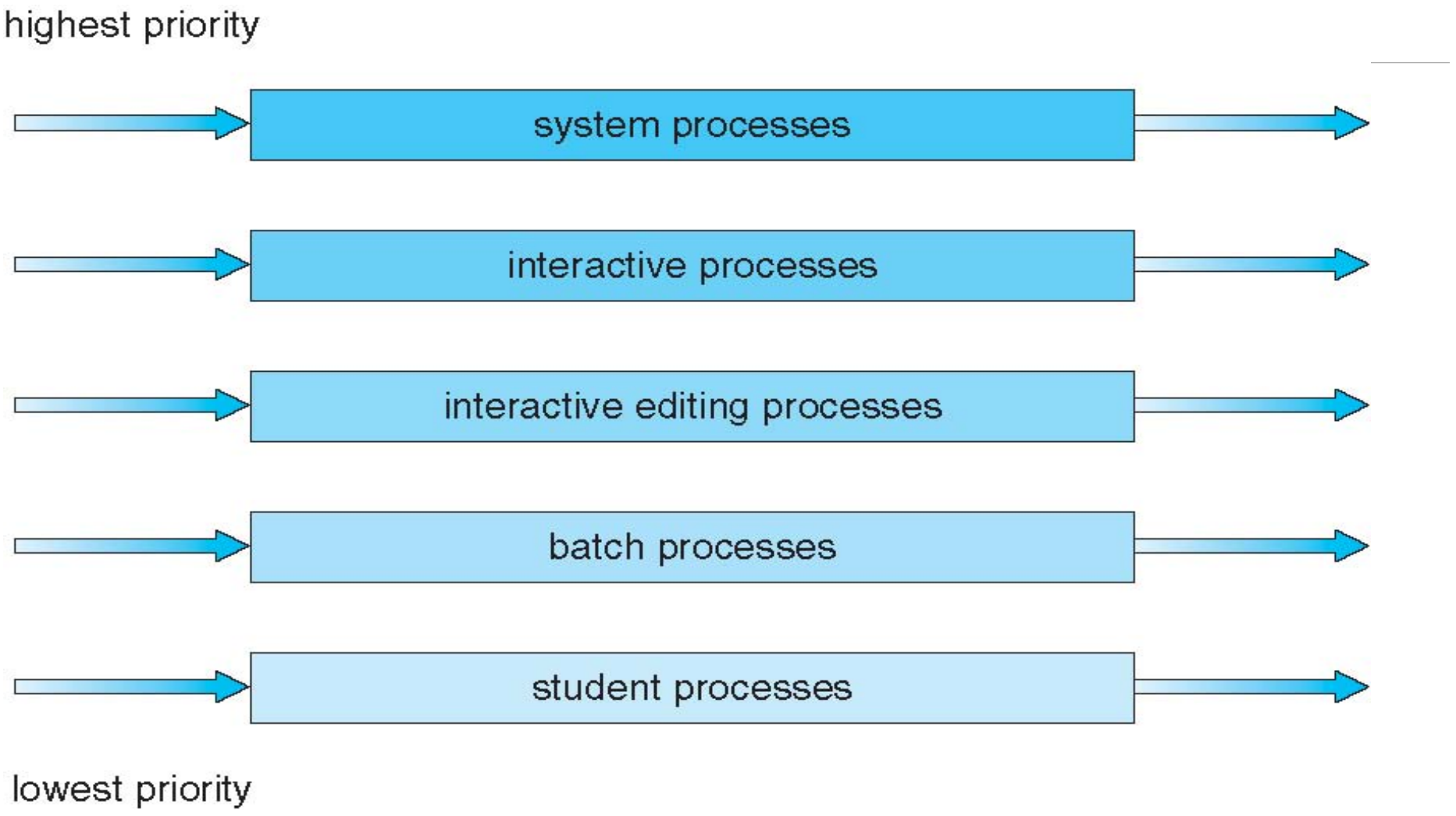
Each queue has its own scheduling algorithm:

- foreground – RR
- background – FCFS

Scheduling must be done between the queues:

- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

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

Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

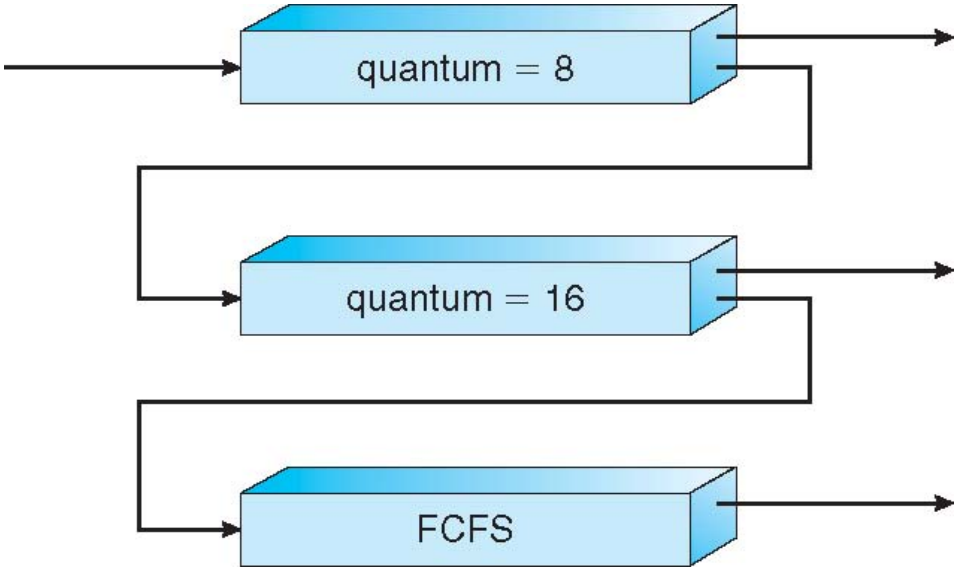
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling

- A new job enters queue Q_0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2

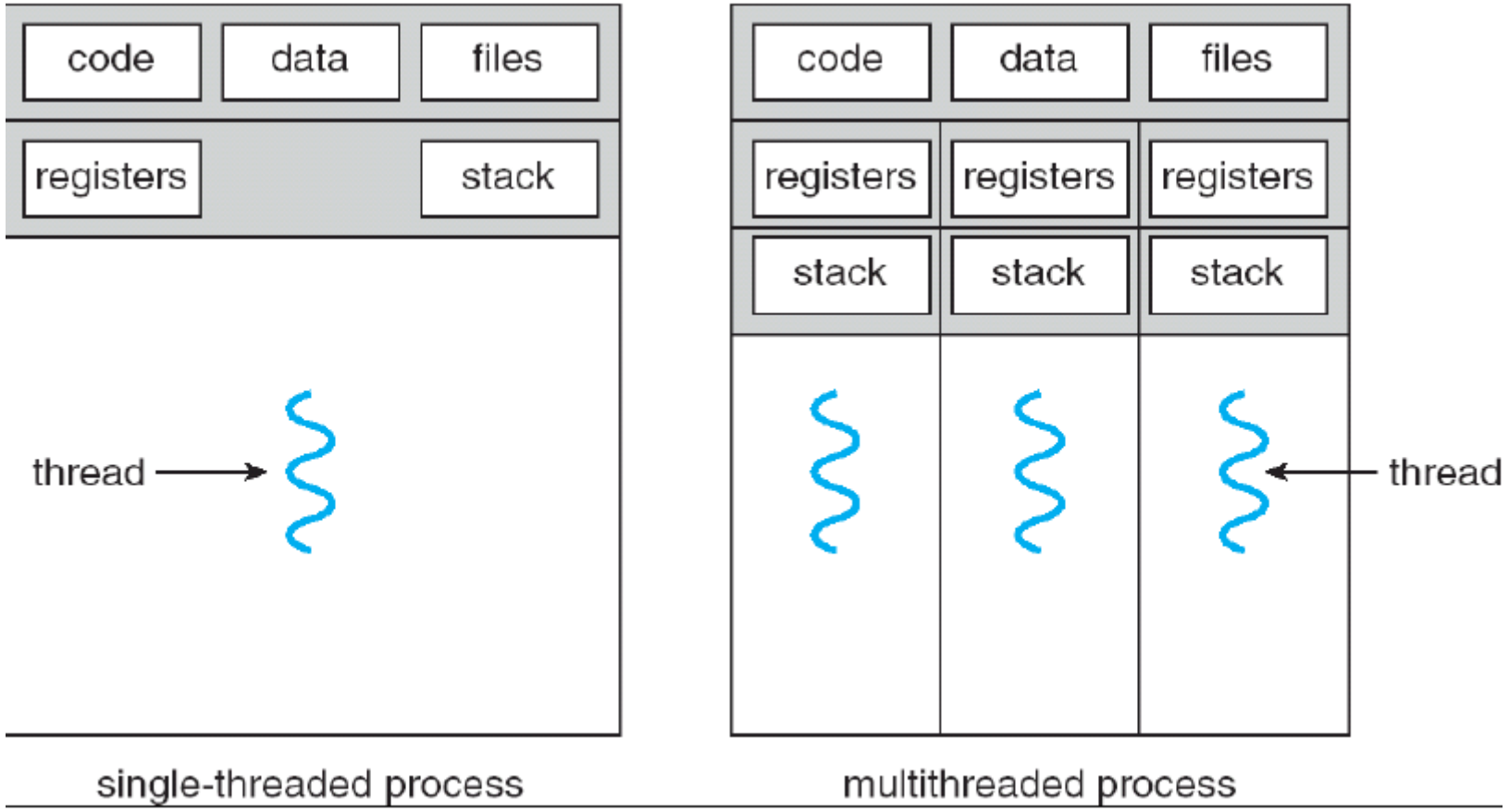


Threads

Thread

- A thread is called **a light weight process.**
- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- Threads are a popular way to improve application performance through parallelism.
- Each thread belongs to exactly one process and no thread can exist outside a process.

Single-threaded and Multi-threaded processes



Benefits of Multithreaded Environment

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Types of Thread

Thread is implemented in two ways :

1. User Level

2. Kernel Level

User Level Thread

- All of the work of thread management is done by the application and the kernel is not aware of the existence of threads.
- The thread library contains code for creating and destroying threads.
- The application begins with a single thread and begins running in that thread.

Kernel Level Threads

- Thread management is done by the Kernel.
- No thread management code in the application area.
- Kernel threads are supported directly by the operating system.
- The Kernel performs thread creation, scheduling and management in Kernel space.
- Kernel threads are generally slower to create and manage than the user threads.

User Level vs Kernel Level Threads

S.No.	User Level Threads	Kernel Level Thread
1	User level thread are faster to create and manage.	Kernel level thread are slower to create and manage.
2	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Advantages of Thread

1. Thread minimize context switching time.

2. Use of threads provides concurrency within a process.

3. Efficient communication.

4. Economy- It is more economical to create and terminate threads than processes.

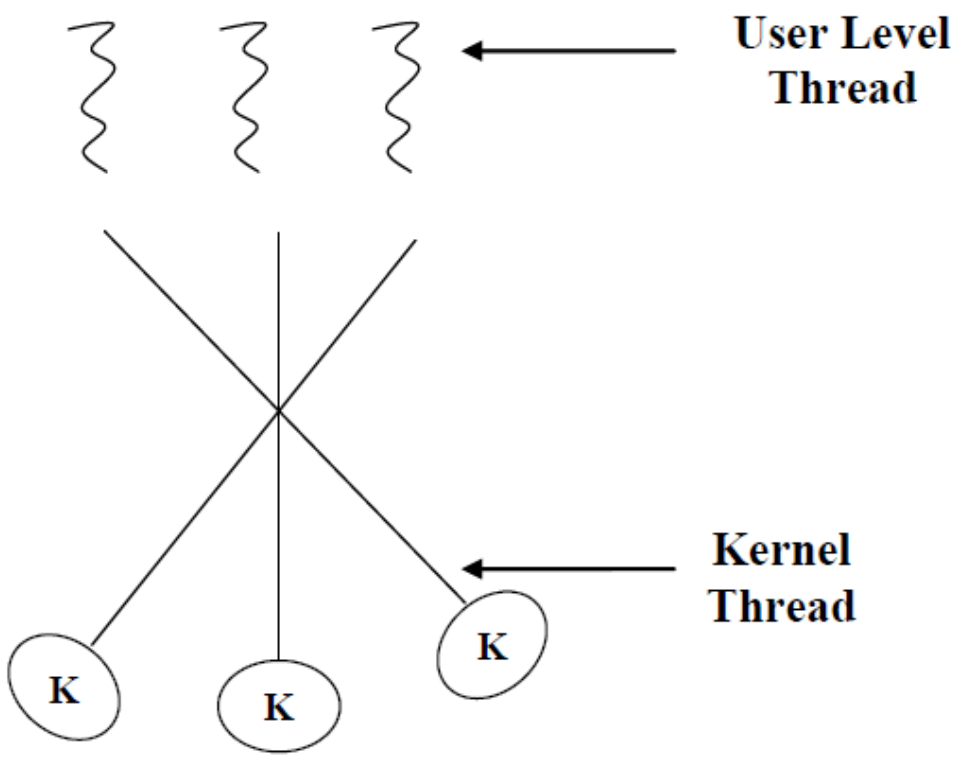
5. Utilization of multiprocessor architectures– The benefits of multithreading can be greatly increased in a multiprocessor architecture.

Multithreading Models

- Some operating system provide a combined user level thread and Kernel level thread facility.
- Solaris is a good example of the combined approach.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors.
- Multithreading models are of three types:
  1. Many to many relationship.
  2. Many to one relationship.
  3. One to one relationship.

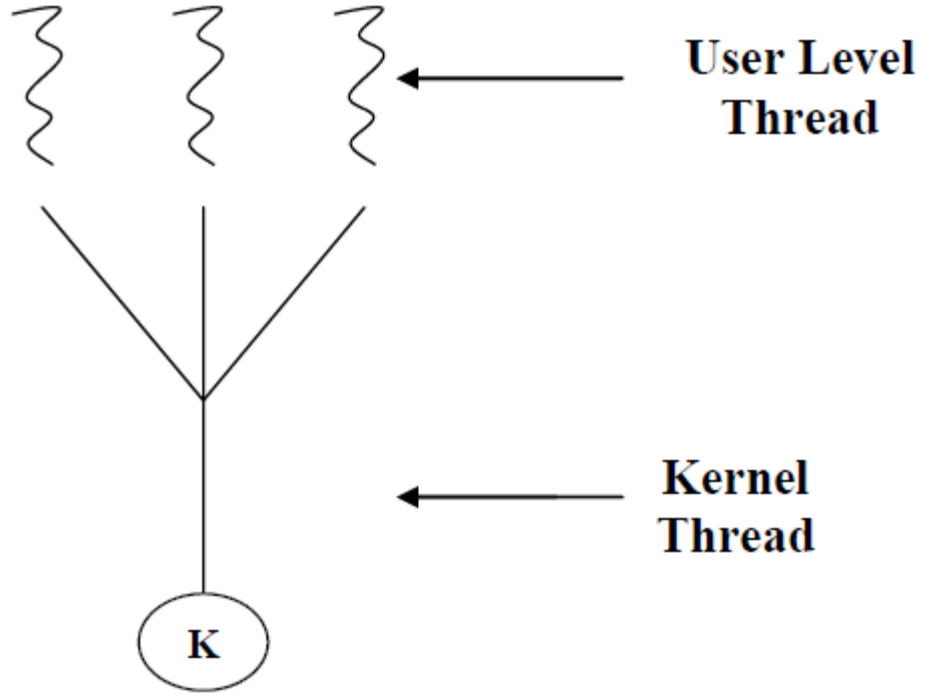
Many to Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.



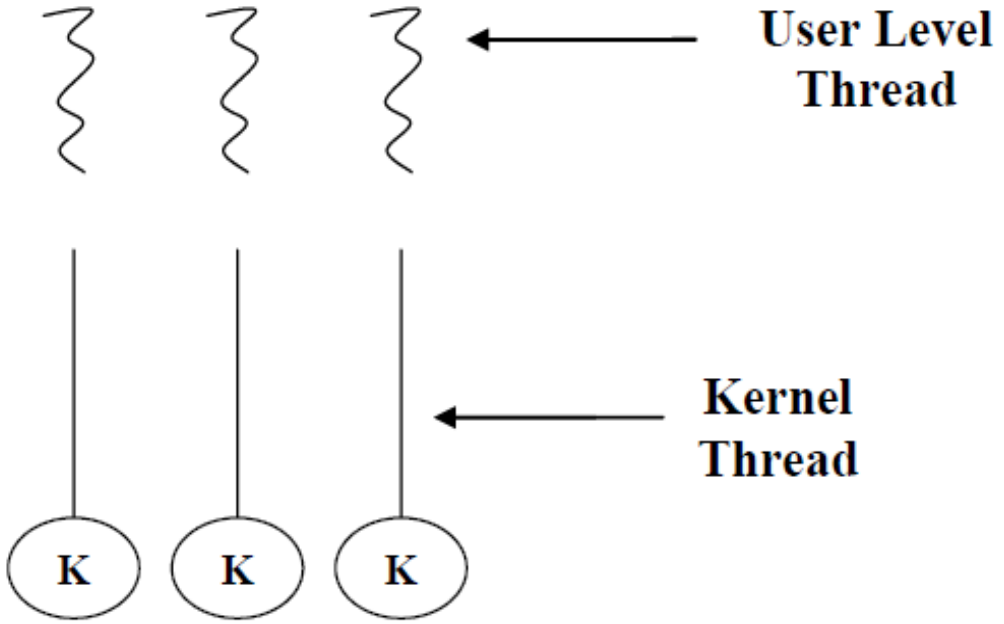
Many to One Model

- The many-to-one model maps many user-level threads to one kernel thread.
- Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors



One to One Model

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Drawback:- creating a user thread requires creating the corresponding kernel thread.



End of Chapter

Process Synchronization

Background

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Processes can execute concurrently.
  - Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

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

Race Condition

`counter++` could be implemented as

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

`counter--` could be implemented as

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

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition

- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a race condition.
- To guard against race condition, we need to ensure that only one process at a time can be manipulating the shared data(variable counter).

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process is in critical section, no other may be in its critical section.
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section.

Critical Section

General structure of process P_i

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

Requirements of Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** – No process running outside critical section should block the interested process from entering into critical section when critical section is free.
3. **Bounded Waiting** - No process should wait forever to enter inside the critical section. There should be a bound on giving chance to enter into critical section.

Solutions to Critical Section Problem

- Software based
- Hardware based
- Programming based

Peterson's Solution

- Software based solution to the critical section problem.
- Two process solution- i.e. valid only for two processes.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable **turn** indicates whose turn it is to enter the critical section. turn==i indicates that process $\mathbf{P_i}$ is allowed to enter its critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i and P_j

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

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

Synchronization Hardware

- Hardware features can make any programming task easier and improve system efficiency.
- Hardware instructions that allow us either to test and modify the content of a word atomically- i.e. as one uninterruptible unit- TestandSet().
- The important characteristic of TestandSet() instruction is that it is executed atomically.
  - Thus if two TestandSet() instructions are executed simultaneously, they will be executed sequentially.

Solution to Critical Section Problem using Locks

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

Synchronization Hardware

Shared Boolean variable lock

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

Semaphores

- Semaphore is an integer variable that, apart from initialization is accessed through two standard atomic operations- wait() and signal(). (Also called P() and V() respectively.)

Definition of the **wait() operation**

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

Definition of the **signal() operation**

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

Semaphores

- Semaphores can be counting semaphores and binary semaphores.
- Counting semaphores can range over an unrestricted domain, while binary semaphores can range between 0 and 1.
- Counting semaphores can be used to control access to a resource of a finite number of instances.
  - The semaphore is initialized to the number of resources available.
  - Each process that wishes to use a resource execute wait() operation.(decrements the count)
  - Each process that releases a resource executes signal() operation.(increments the count)
  - When the count for semaphore goes to 0, all resources are being used.
- When a process is in its critical section, any other process that tries to enter in its critical section must loop continuously in the entry code.
  - This continual looping is a problem in real multiprogramming system, wasting CPU cycles.

Semaphores

- To solve the busy waiting problem, the process can block itself whenever the semaphore value is not positive.
- A process that is blocked, waiting on a semaphore S, should be restarted using **wakeup()** operation when some other process executes a signal() operation.
- To implement semaphores under this definition, a semaphore is defined as a "C" structure- having one integer **value** and a list of **process**.

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

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

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

Deadlock and Starvation

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation – **indefinite blocking**

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

Classical Problems of Synchronization

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

Bounded-Buffer Problem- Producer Consumer Problem

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

Bounded Buffer Problem (Cont.)

The structure of the producer process

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

Bounded Buffer Problem (Cont.)

The structure of the consumer process

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

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at a time
- Shared Data
  - Data set
  - Semaphore **`wrt`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0
- Semaphore wrt is common to both reader and writer processes.
- Mutex semaphore is to ensure mutual exclusion when variable readcount is updated.

Readers-Writers Problem (Cont.)

The structure of a writer process

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

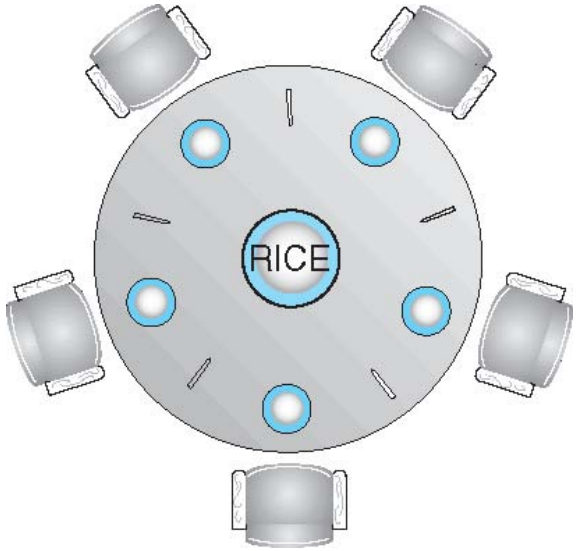
Readers-Writers Problem (Cont.)

The structure of a reader process

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

Dining-Philosophers Problem

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



Dining-Philosophers Problem Algorithm

The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
         wait (chopstick[ (i + 1) % 5] );

                    //  eat

         signal (chopstick[i] );
         signal (chopstick[ (i + 1) % 5] );

                   //  think

} while (TRUE);
```

What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

Deadlock handling

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

Problems with Semaphores

Incorrect use of semaphore operations:

- signal (mutex) …. wait (mutex)
- wait (mutex) … wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

Deadlock and starvation are possible.

Monitors

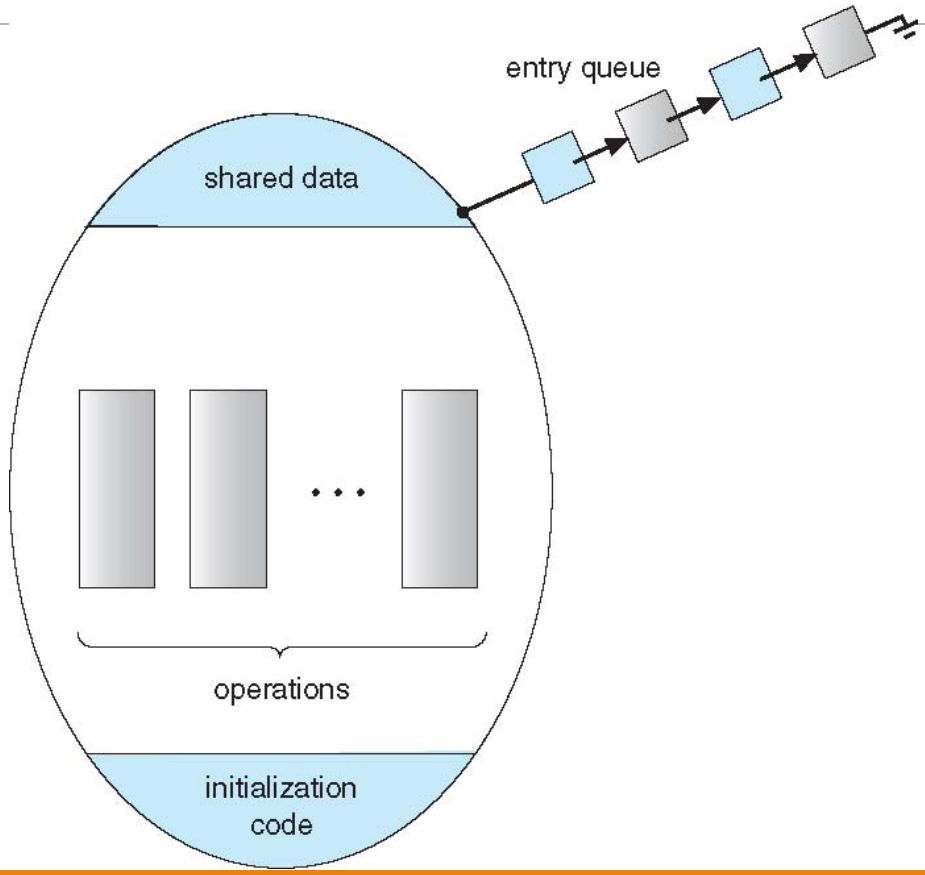
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*- encapsulates private data with public methods.
  - internal variables only accessible by code within the procedure
- A monitor is an ADT which presents a set of programmer defined operations provided mutual exclusion within the monitor.
  - A procedure defined within a monitor can access only those variables declared locally within a monitor.
- 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



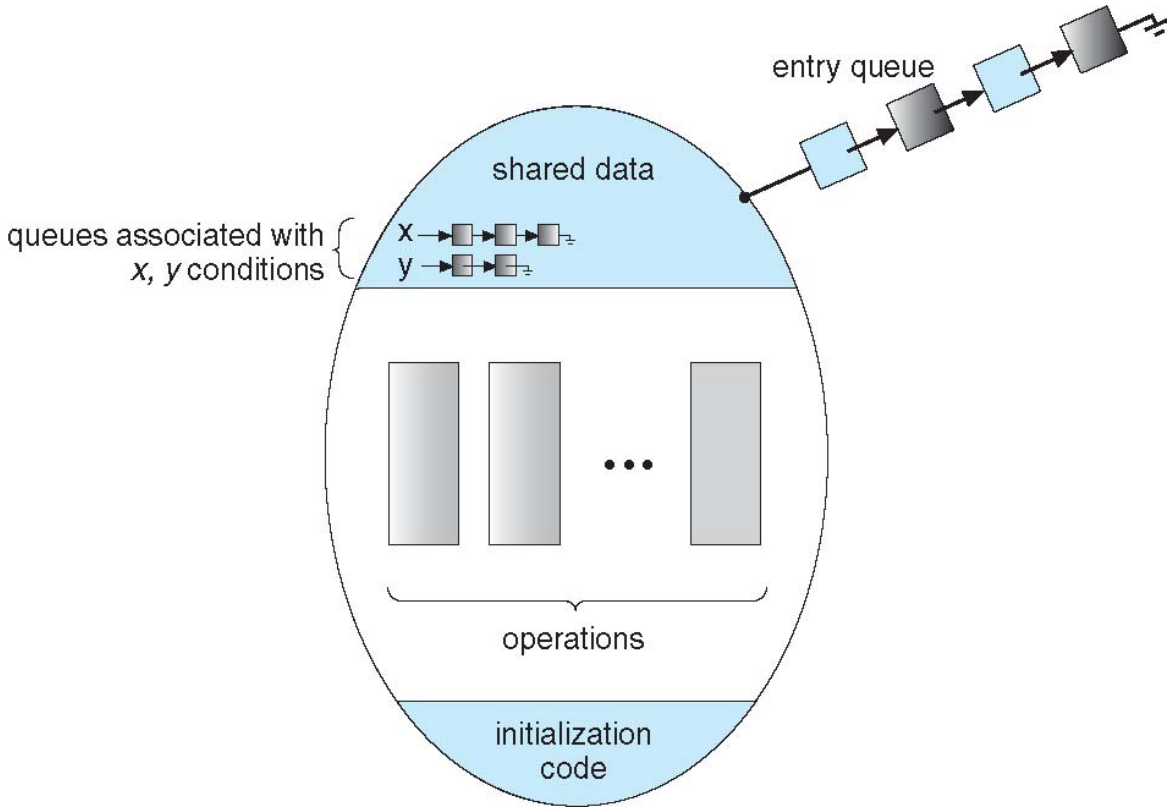
Condition Variables

```
condition x, y;
```

Two operations are allowed on a condition variable:

- **`x.wait()`** – a process that invokes the operation is suspended until x.signal()
- **`x.signal()`** – resumes one of processes (if any) that invoked x.wait()

Monitor with Condition Variables



Processes

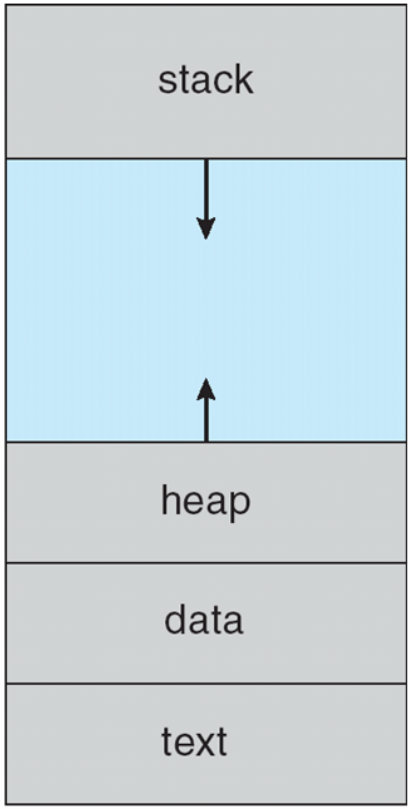
Process Concept

- Process is a **Program in execution.**
- A process defines the fundamental unit of computation for the computer.
- A process can run to completion only when all requested resources have been allocated to the process.
- Two or more processes could be executing the same program, each using their own data and resources.

Process Concept

Multiple parts of a process

- The program code, also called **text section**
- Current activity including **program counter**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



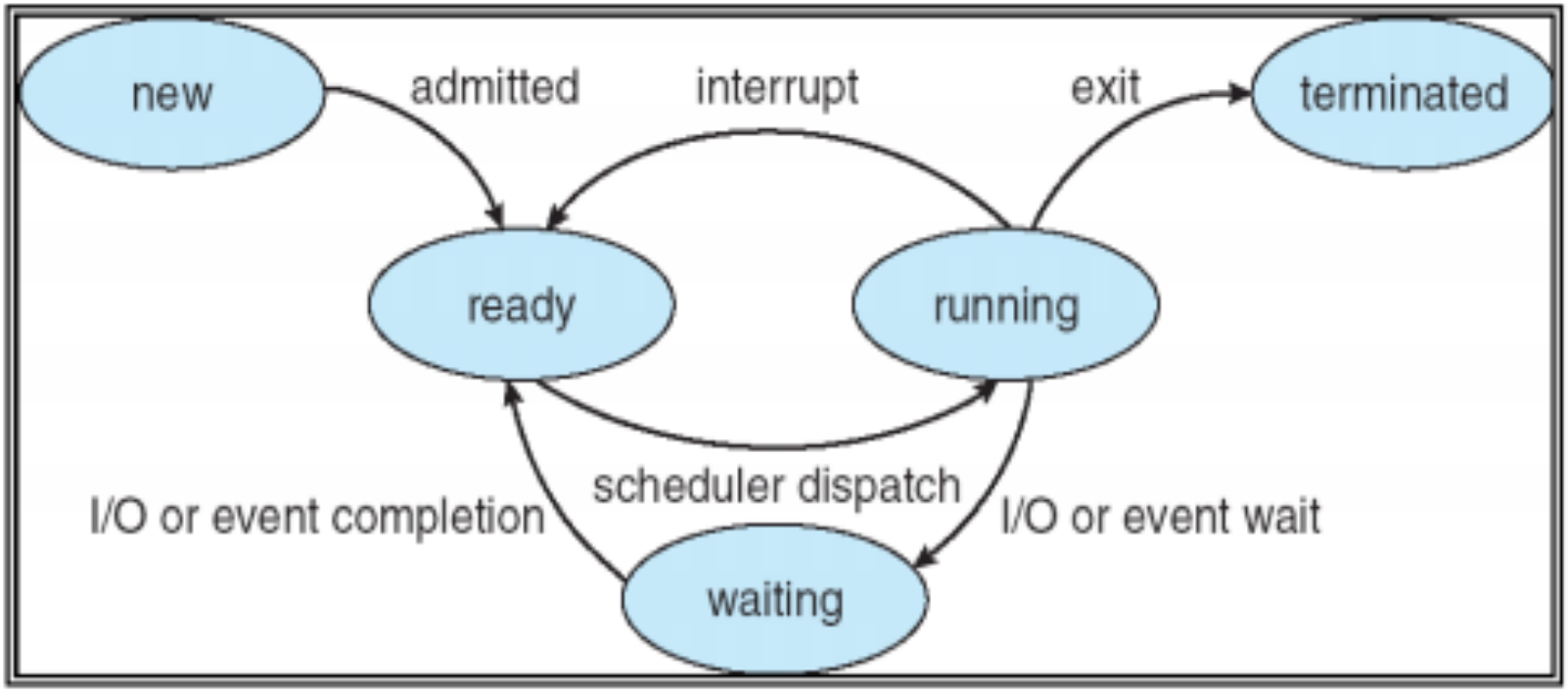
Process and Program

- Process is a dynamic entity that is a program in execution, while Program is a static entity made up of program statement.
- A process is a sequence of information executions, while Program contains the instructions.
- Process exists in a limited span of time, while a program exists at single place in space and continues to exist.
- Two or more processes could be executing the same program, each using their own data and resources.

Process States

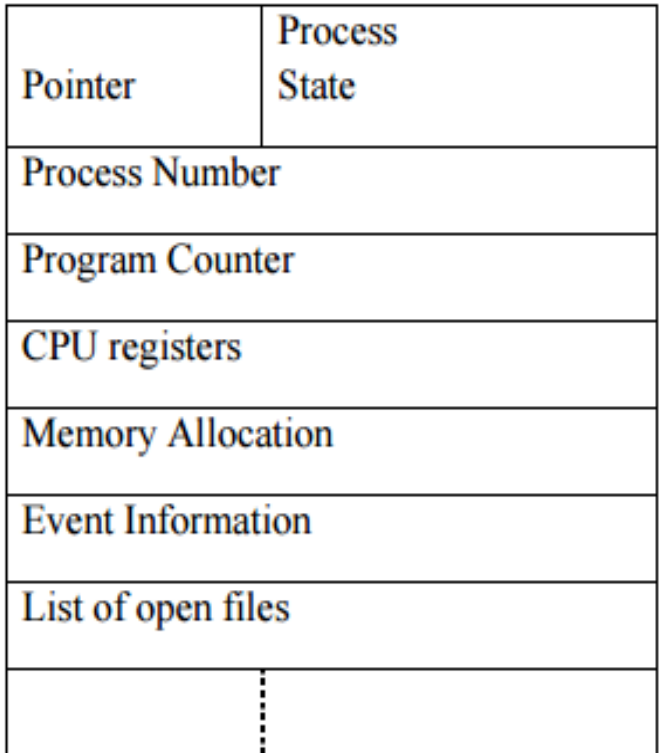
- When process executes, it changes state. Process state is defined as the current activity of the process.
- A process exists in one of the five states:
  - New: A process that just been created.
  - Ready: Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
  - Running: The process that is currently being executed. A running process possesses all the resources needed for its execution, including the processor.
  - Waiting: A process that can not execute until some event occurs such as the completion of an I/O operation. The running process may become suspended by invoking an I/O module.
  - Terminated: A process that has been released from the pool of executable processes by the operating system.

Process States

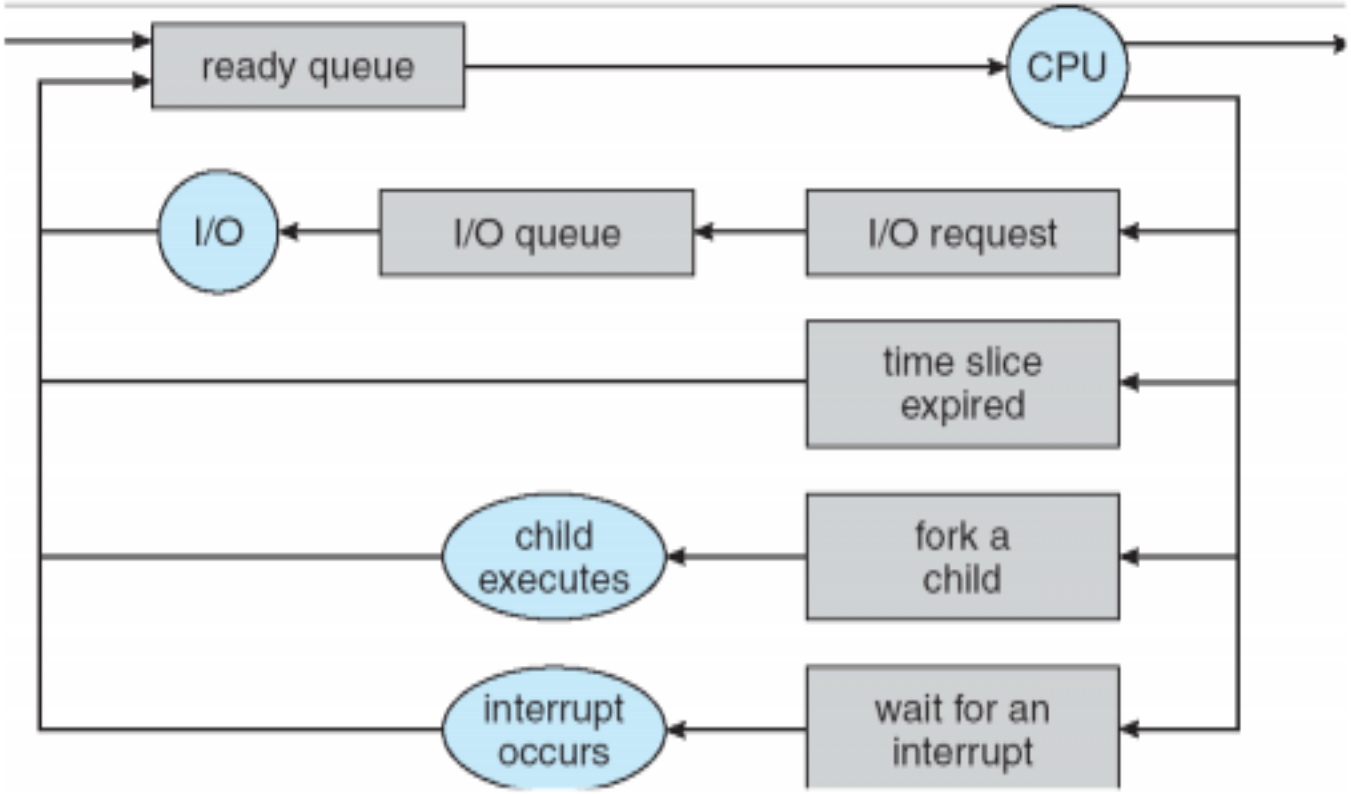


Process Control Block

- Each process contains the process control block (PCB).
- PCB is the data structure used by the operating system.
- Operating system groups all information that it needs about particular process.



Scheduling Queues



Schedulers

Schedulers are of three types.

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Long term scheduler

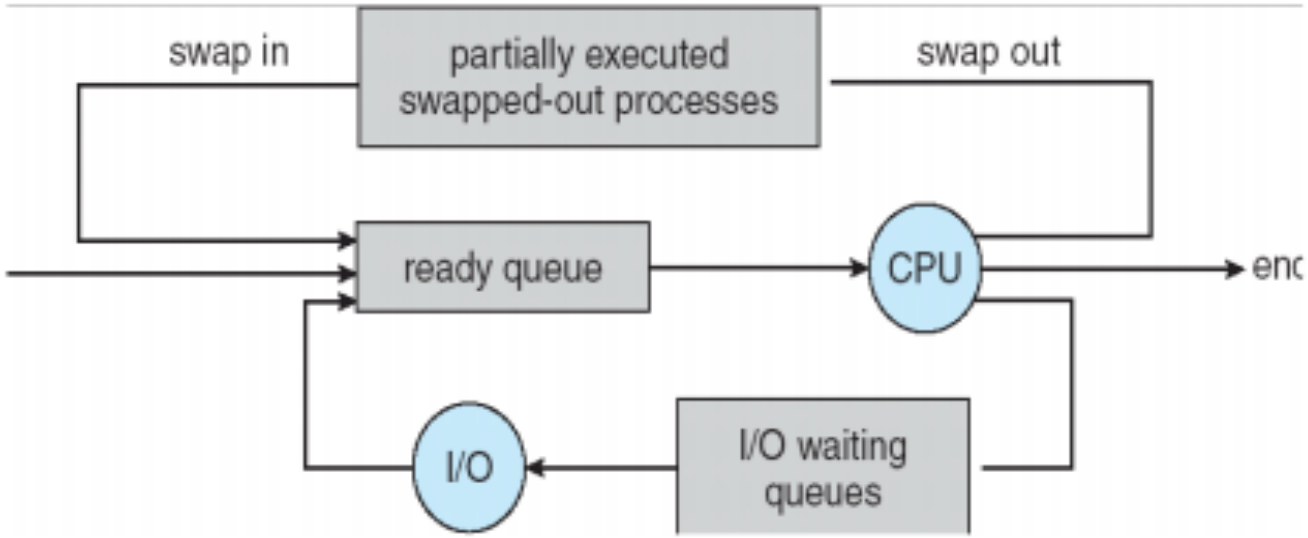
- It is also called job scheduler.
- Long term scheduler determines which programs are admitted to the system for processing.
- Job scheduler selects processes from the queue and loads them into memory for execution.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming.

Short term scheduler

- It is also called CPU scheduler.
- It is the change of ready state to running state of the process.
- CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.
- Short term scheduler also known as dispatcher.
- Short term scheduler is faster than long tern scheduler.

Medium term Scheduler

- Medium term scheduling is part of the swapping function.
- It reduces the degree of multiprogramming.
- The medium term scheduler is in charge of handling the swapped out-processes.



Context Switch

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

Operations on Processes

System must provide mechanisms for:

- process creation,
- process termination

Process Creation

Parent process create **child** processes, which, in turn create other processes, forming a **tree** of processes

Generally, process identified and managed via a **process identifier** (**pid**)

Resource sharing options

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

Process Termination

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

Interprocess Communication

Processes within a system may be ***independent*** or ***cooperating***

Cooperating process can affect or be affected by other processes, including sharing data

Reasons for cooperating processes:

- Information sharing
- Computation speedup
- Modularity
- Convenience

Cooperating processes need **interprocess communication** (**IPC**)

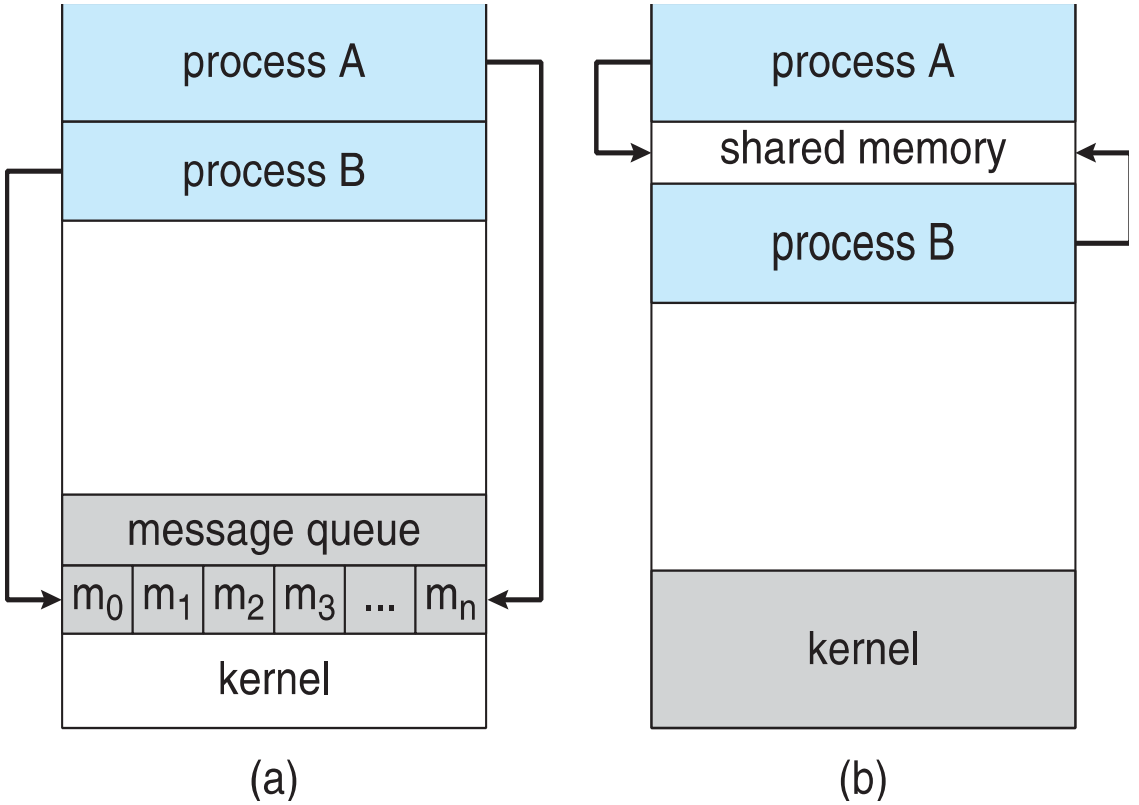
Two models of IPC

- **Shared memory**
- **Message passing**

Communications Models

(a) Message passing.
- Naming
- Synchronization
- Buffering

(b) Shared memory.



Message Passing Systems

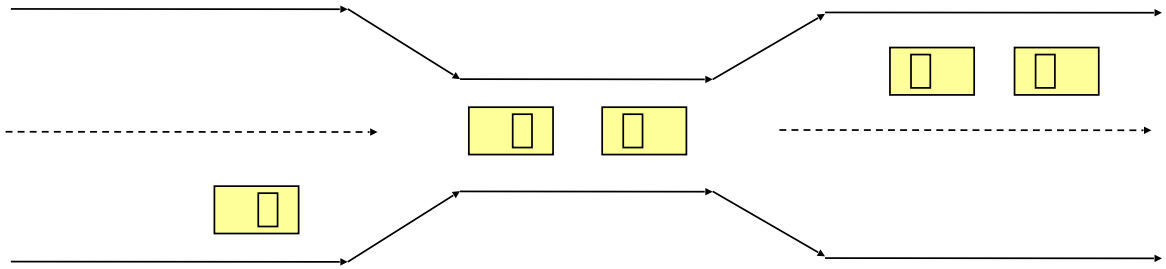
- Naming
  - Direct Communication
  - Indirect Communication (using a mailbox)
- Synchronization
  - Blocking send
  - Nonblocking send
  - Blocking receive
  - Nonblocking receive
- Buffering
  - Zero capacity
  - Bounded capacity
  - Unbounded capacity

End of Chapter

INTRODUCTION

- A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - A system has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one

BRIDGE CROSSING EXAMPLE



- Traffic only in one direction
- The resource is a one-lane bridge
- 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 *1 or more* 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

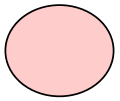
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

No preemption: a resource can be released only voluntarily by the process holding it after that process has completed its task

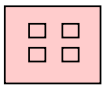
Circular wait: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by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

Resource-Allocation Graph

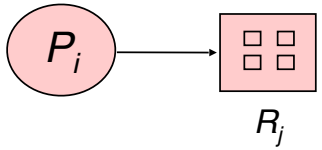
Process



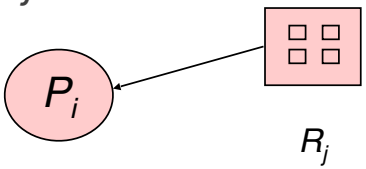
Resource Type with 4 instances



P_i requests instance of R_j

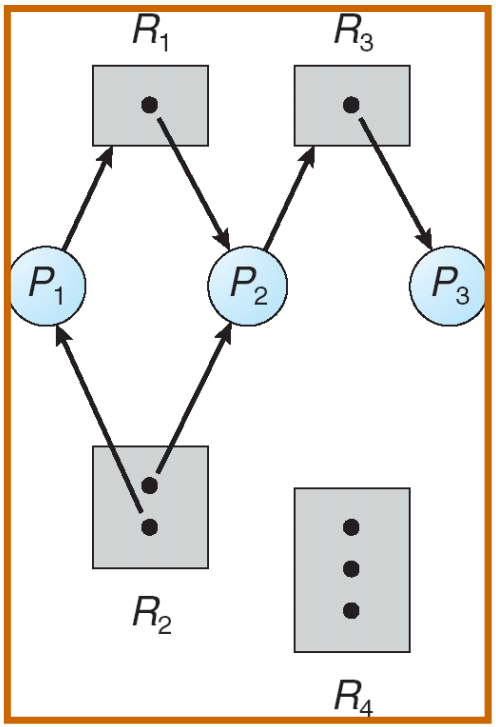


P_i is holding an instance of R_j



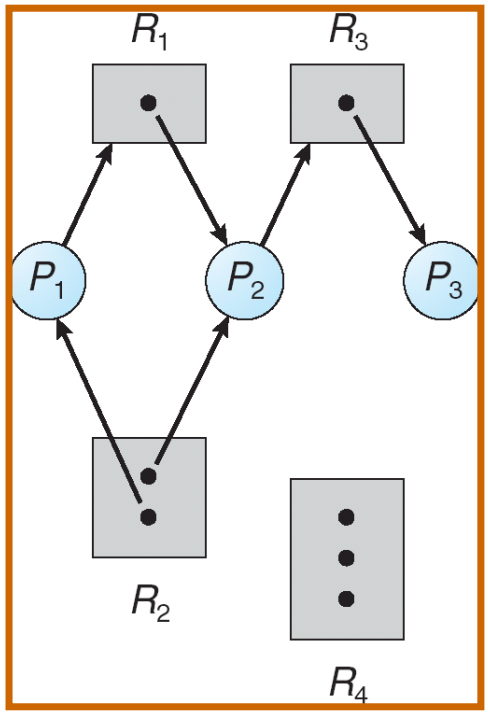
RESOURCE-ALLOCATION GRAPH

- 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 $P_i \rightarrow R_j$
- Assignment edge –
  - directed edge $R_j \rightarrow P_i$

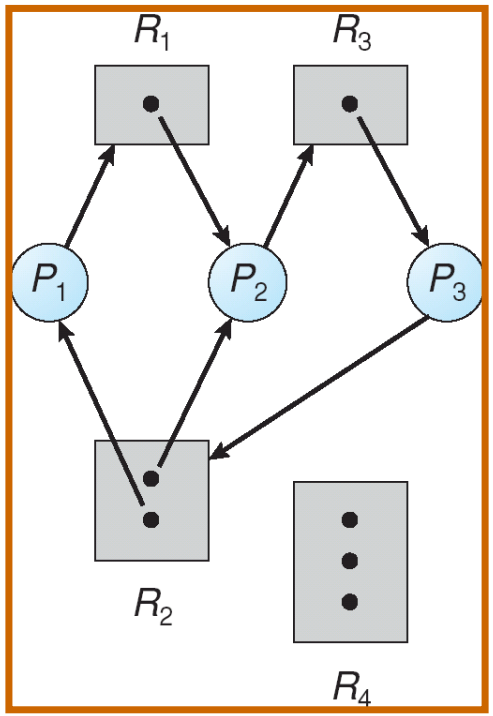


Resource Allocation Graph With A Deadlock

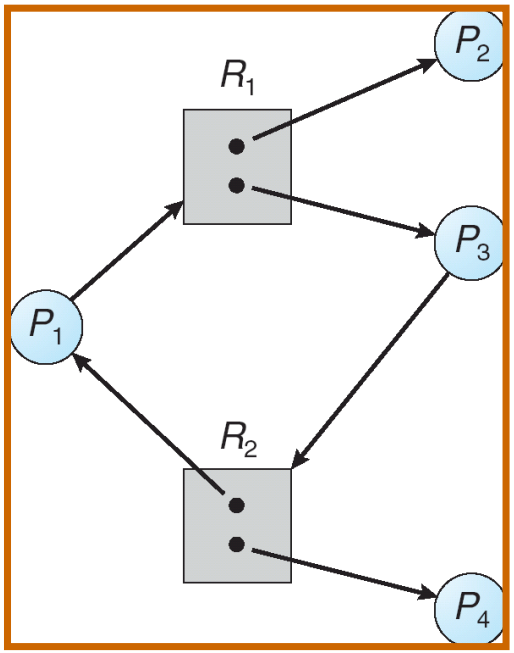
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock



Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

If a resource allocation graph contains no cycles ⇒ no deadlock

If a resource allocation graph contains a cycle and if only one instance exists per resource type ⇒ deadlock

If a resource allocation graph contains a cycle and and if several instances exists per resource type ⇒ possibility of deadlock

Methods for Handling Deadlocks

Methods for Handling Deadlocks

Prevention

- Ensure that the system will *never* enter a deadlock state.

Avoidance

- Ensure that the system will *never* enter an unsafe state.

Detection

- Allow the system to enter a deadlock state and then recover.

Do Nothing

- Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX.

Deadlock Prevention

To prevent deadlock, we can restrain the ways that a request can be made

Mutual Exclusion – The mutual-exclusion condition must hold for non-sharable resources.

Hold and Wait – We must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require a process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none
- Result: 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.
- A 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. For example:

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

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.

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

a priori: formed or conceived beforehand

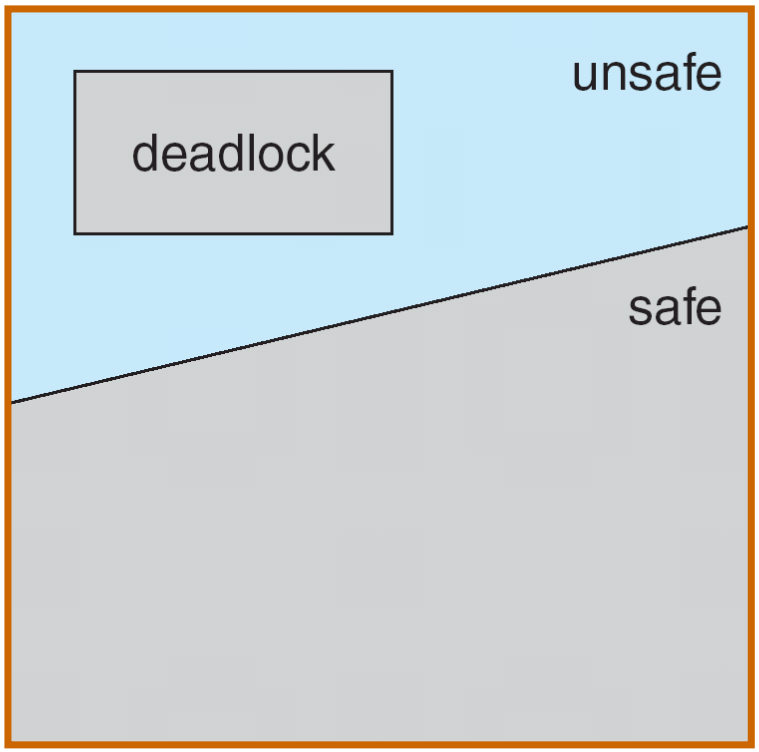
Safe State

If a system is in safe state ⇒ no deadlocks

If a system is in unsafe state ⇒ possibility of deadlock

Avoidance ⇒ ensure that a system will never enter an unsafe state

Safe, Unsafe , Deadlock State



Avoidance algorithms

For a single instance of a resource type, use a resource-allocation graph

For multiple instances of a resource type, use the banker's algorithm

Resource-Allocation Graph Scheme

Introduce a new kind of edge called a claim edge

Claim edge $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line

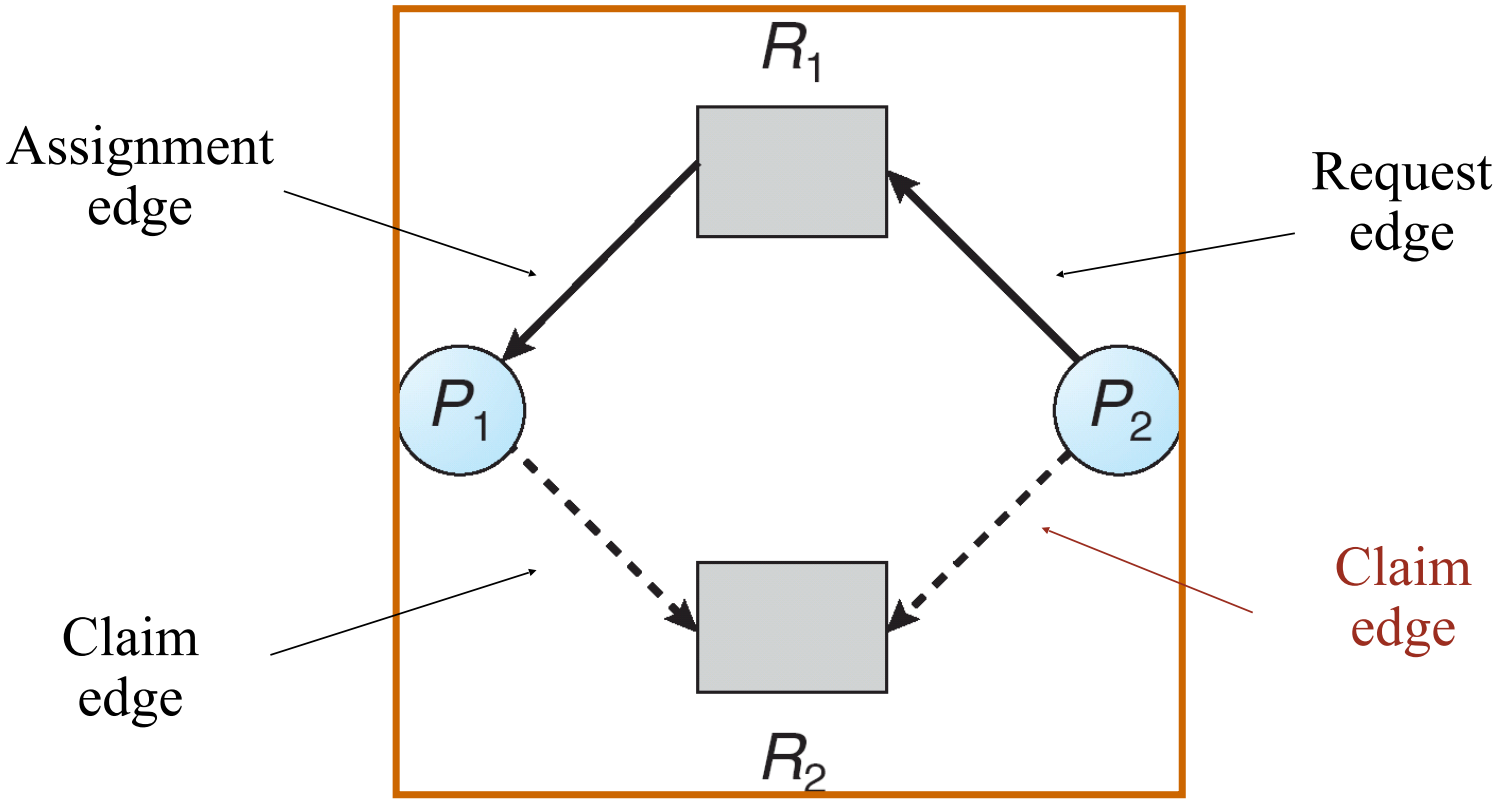
A claim edge converts to a request edge when a process **requests** a resource

A request edge converts to an assignment edge when the resource is **allocated** to the process

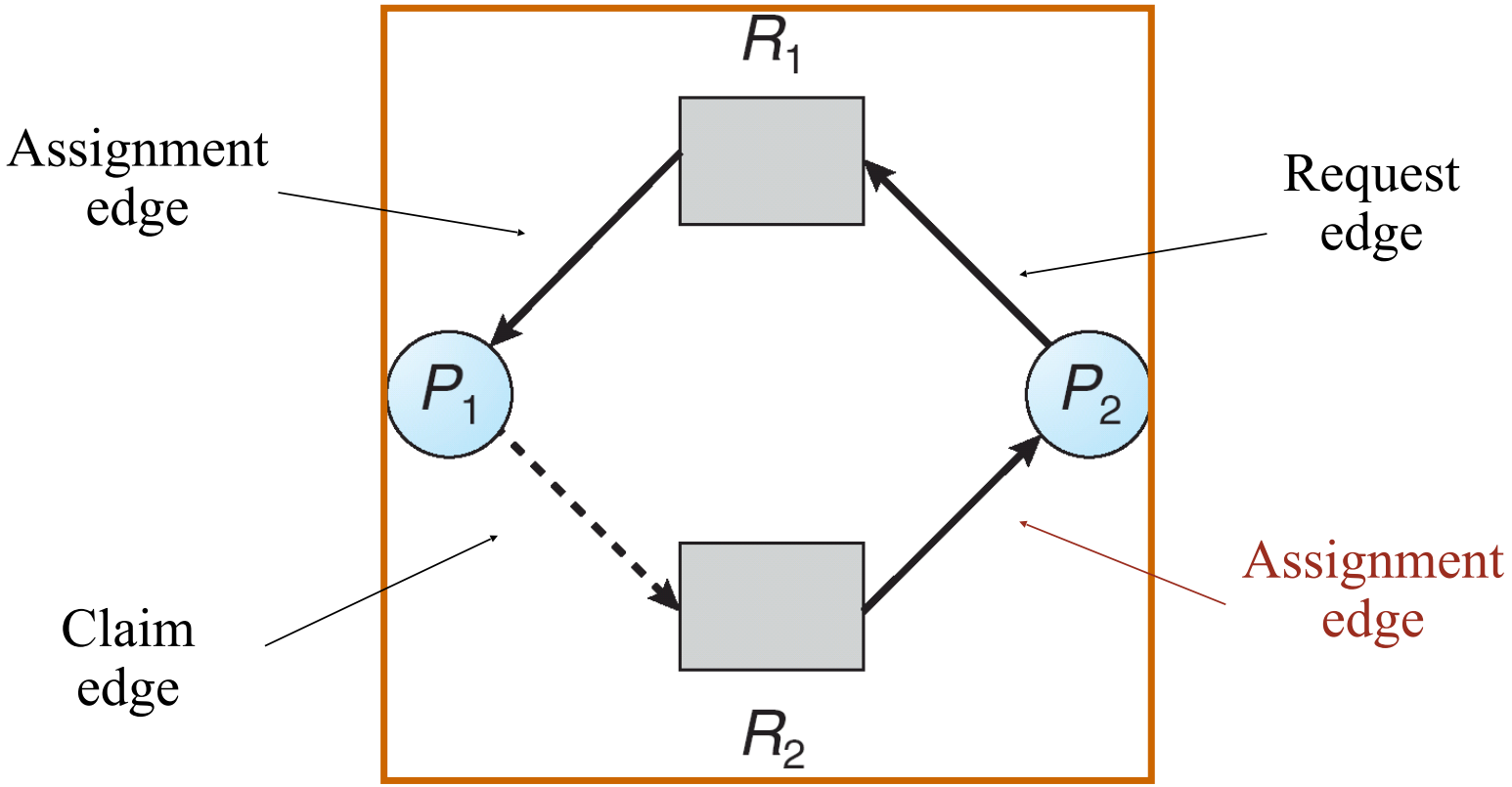
When a resource is **released** by a process, an assignment edge reconverts to a claim edge

Resources must be **claimed *a priori*** in the system

Resource-Allocation Graph with Claim Edges



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

Used when there exists **multiple** instances of a resource type

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]
```

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 ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ 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*

	Allocation	*Max*	*Need*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	P_0 7 4 3
P_1	2 0 0	3 2 2	P_1 1 2 2
P_2	3 0 2	9 0 2	P_2 6 0 0
P_3	2 1 1	2 2 2	P_3 0 1 1
P_4	0 0 2	4 3 3	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 ≤ Available (that is, (1,0,2) ≤ (3,2,2) ⇒ 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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

Allow system to enter deadlock state

Detection algorithm

Recovery scheme

Single Instance of Each Resource Type

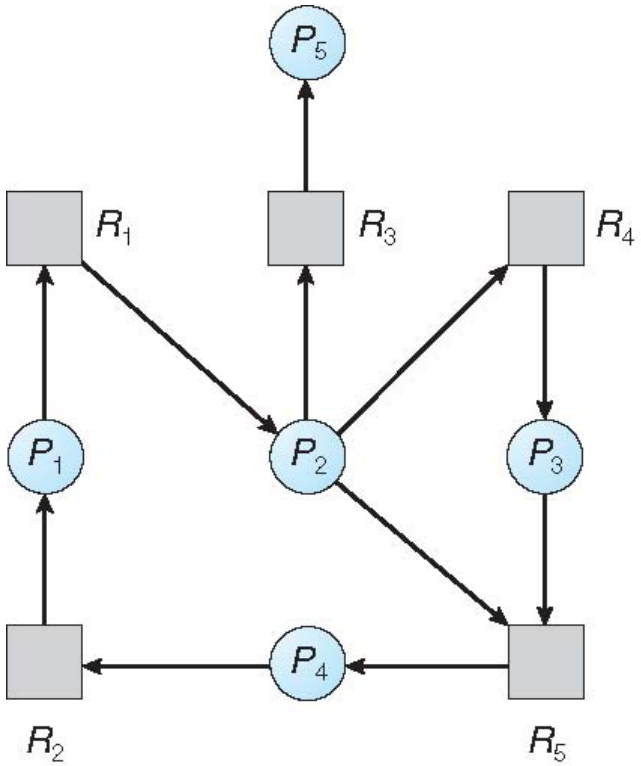
Maintain *wait-for* graph

- Nodes are processes
- $P_i \rightarrow P_j$ if P_i is waiting for P_j

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

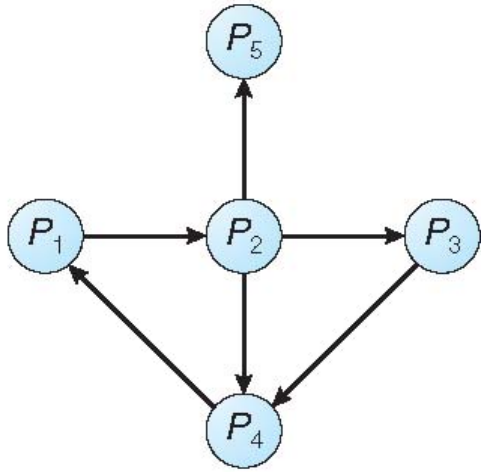
An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

EXAMPLE OF DETECTION ALGORITHM

Five processes P_0 through P_4; three resource types
A (7 instances), *B* (2 instances), and *C* (6 instances)

	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 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

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

DETECTION-ALGORITHM USAGE

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

RECOVERY FROM DEADLOCK:

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

Recovery from Deadlock: Resource Preemption

Selecting a victim – minimize cost

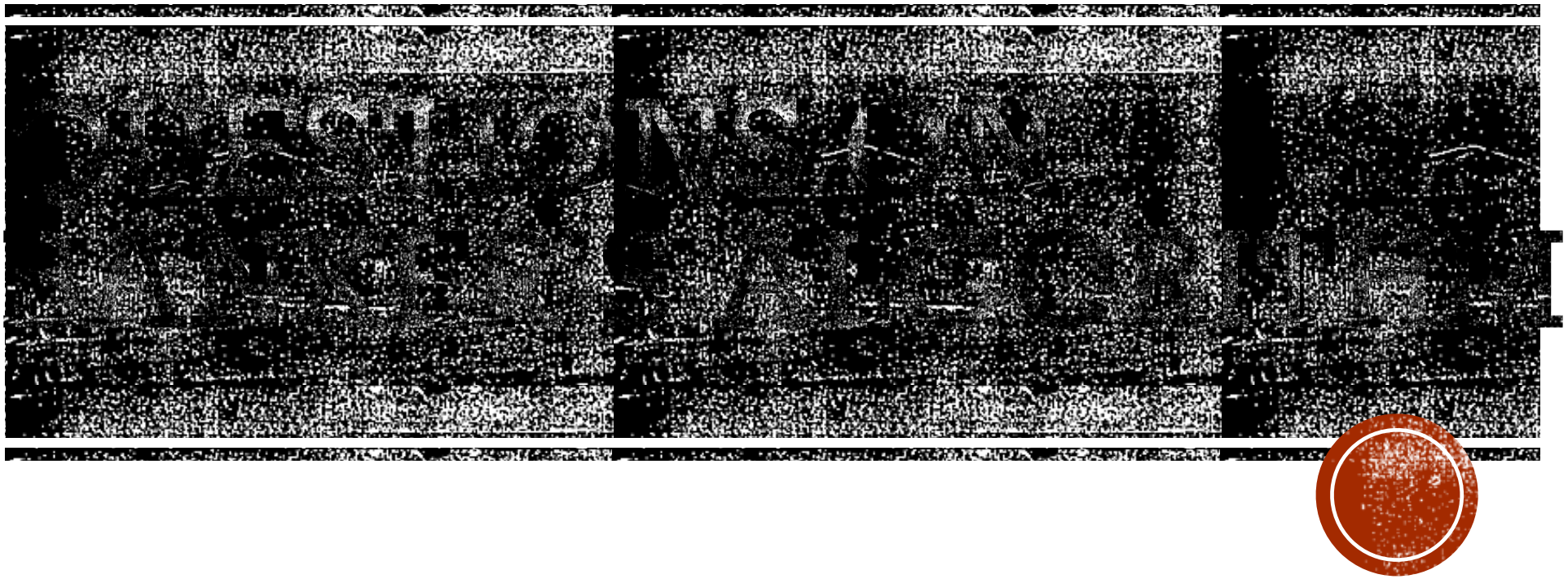
Rollback – return to some safe state, restart process for that state

Problem: starvation – same process may always be picked as victim, include number of rollback in cost factor

QUESTION

A system has 3 processes P1, P2 and P3 and 3 resources R1, R2 and R3. There are 2 instances each of R1 and R2, and one instance of R3. Given the edge set E= {R1 →□ P1, R2 →□ P2, P1 →□ R3, R1 →P2, P3→R1, R2→P3, R3→P3}.

1. Draw the resource allocation graph.
2. Is the system in a deadlock? If yes, then mention the processes in the deadlock else identify the sequence in which the processes can execute.



Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	2 1 1	4 2 3	8 1 2
P1	2 3 1	7 3 10	
P2	1 1 4	5 2 13	
P3	1 4 2	10 5 4	

a. What is the content of the Need matrix?

b. Is the system in a safe state?

FIND OUT THE SAFE SEQUENCE FOR THE FOLLOWING STATE:

	Allocation	Need	Available
	ABCD	ABCD	ABCD
P0	0010	2001	2100
P1	2001	1010	
P2	0120	2100	

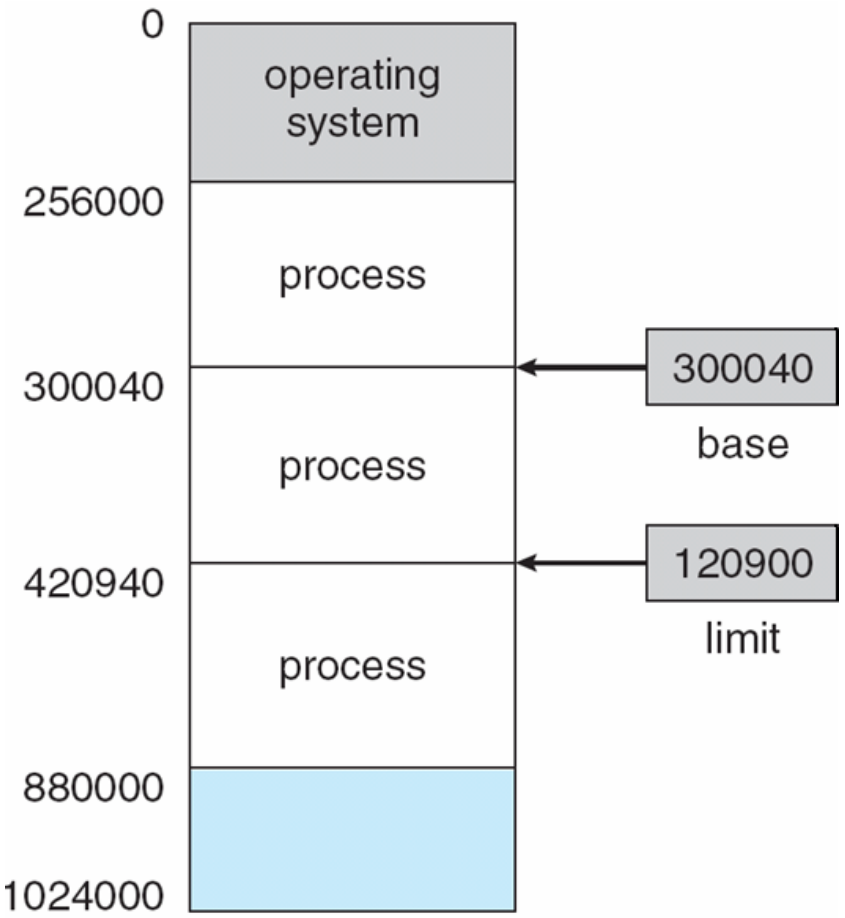
MEMORY MANAGEMENT

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



BINDING OF INSTRUCTIONS AND DATA TO MEMORY

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

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

LOGICAL VS. PHYSICAL ADDRESS SPACE

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

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

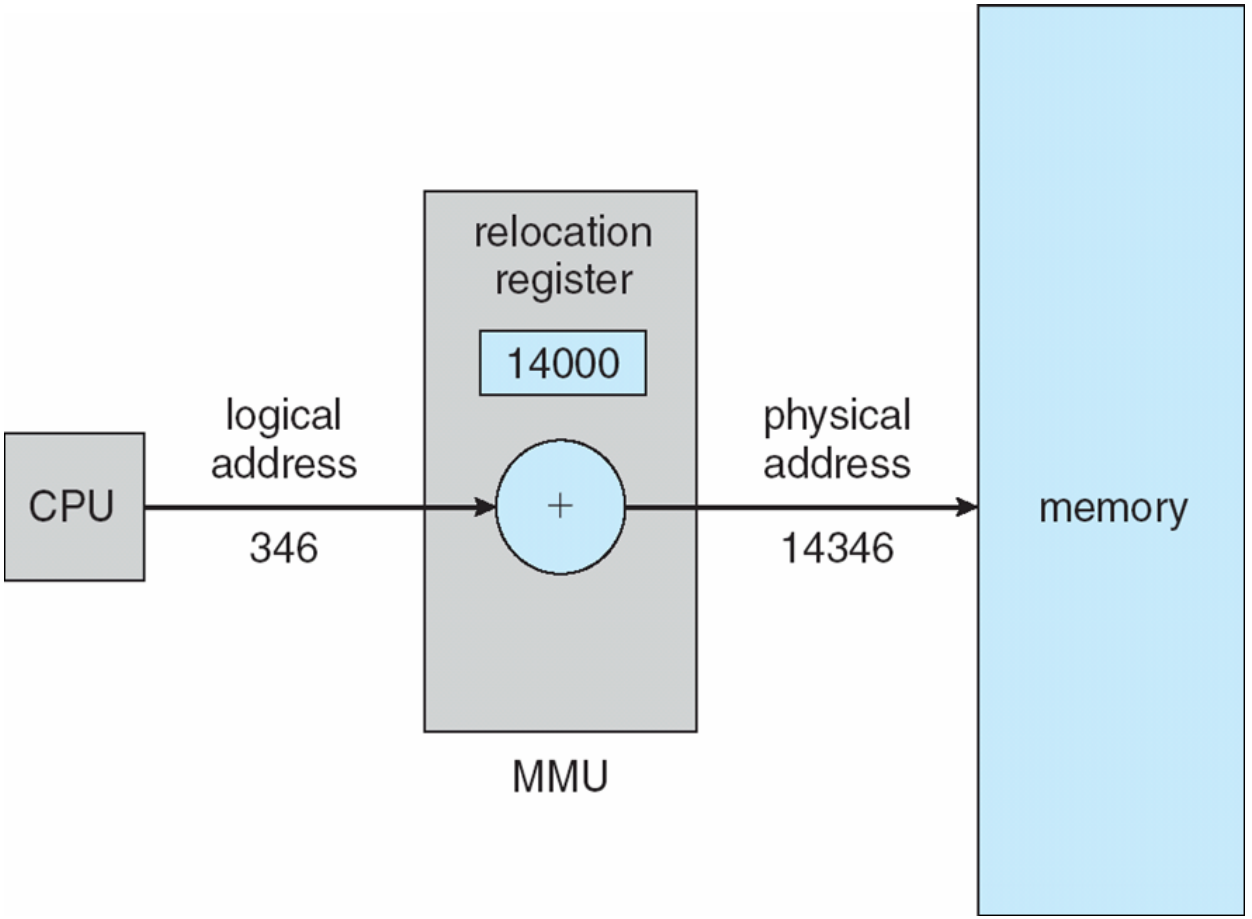
MEMORY-MANAGEMENT UNIT (MMU)

Hardware device that maps virtual to physical address

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

The user program deals with *logical* addresses; it never sees the *real* physical addresses

DYNAMIC RELOCATION USING A RELOCATION REGISTER



DYNAMIC LOADING

Routine is not loaded until it is called

Better memory-space utilization; unused routine is never loaded

Useful when large amounts of code are needed to handle infrequently occurring cases

No special support from the operating system is required implemented through program design

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

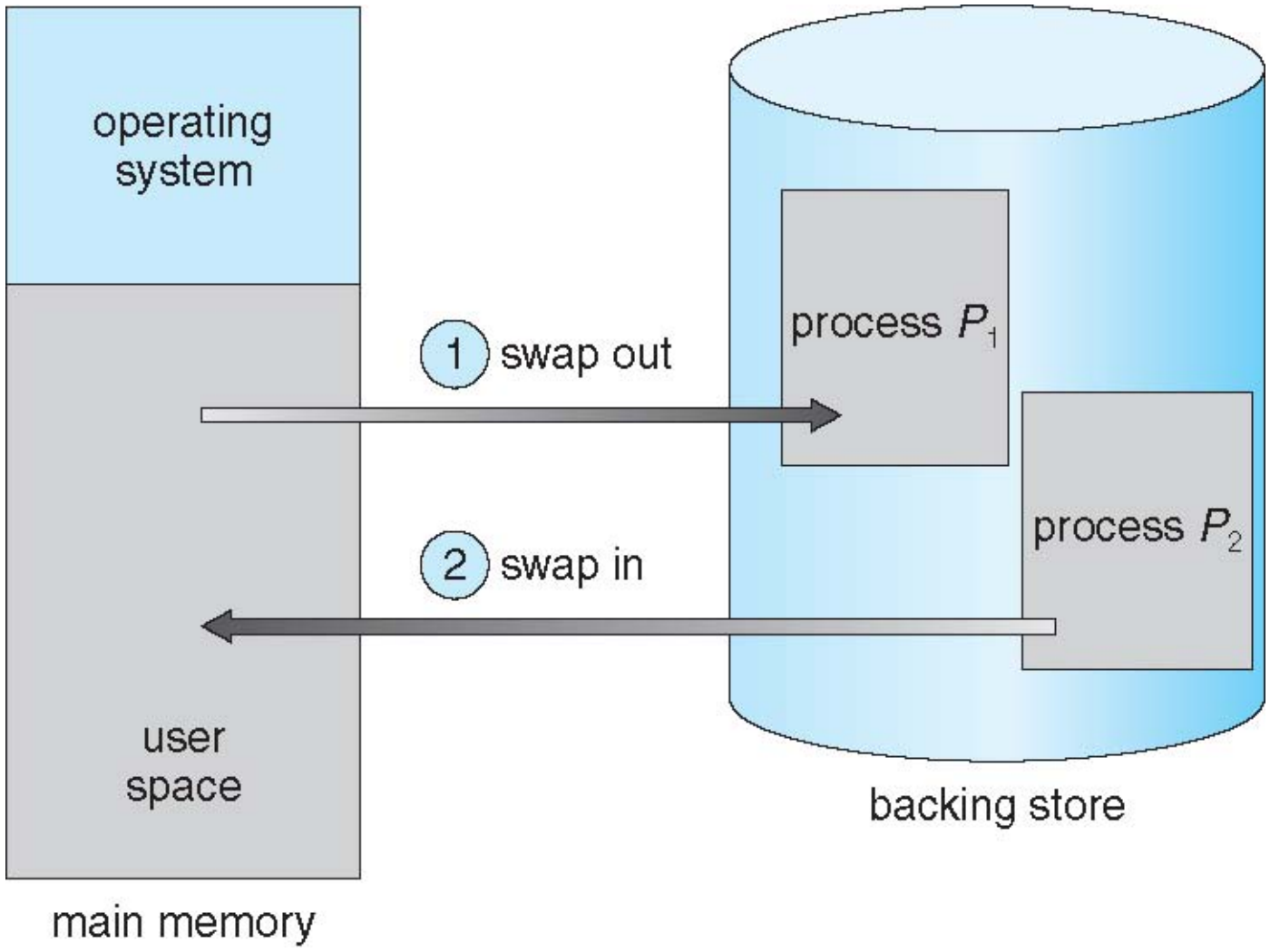
Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

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

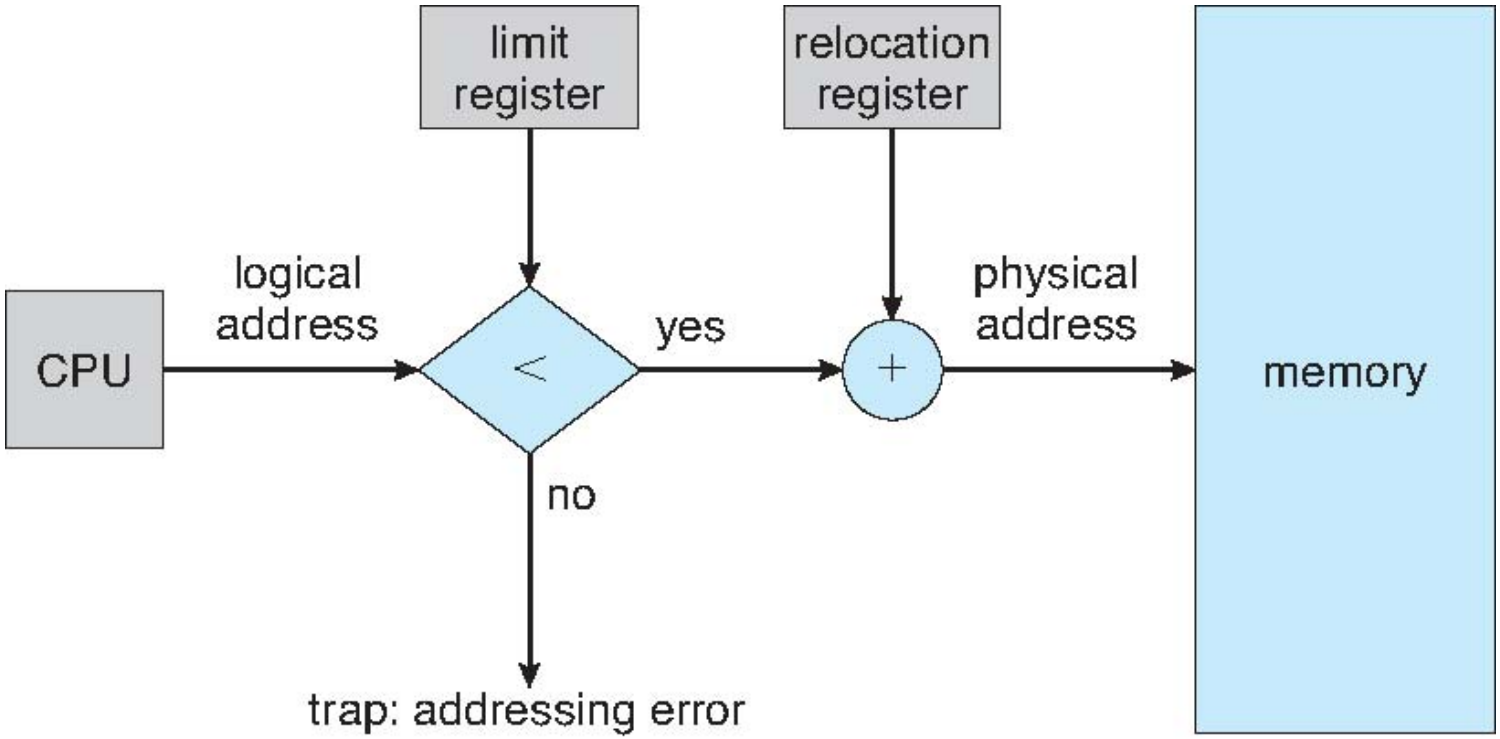
Main memory usually into two partitions:

- Resident operating system, usually held in low memory with interrupt vector
- User processes that 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*

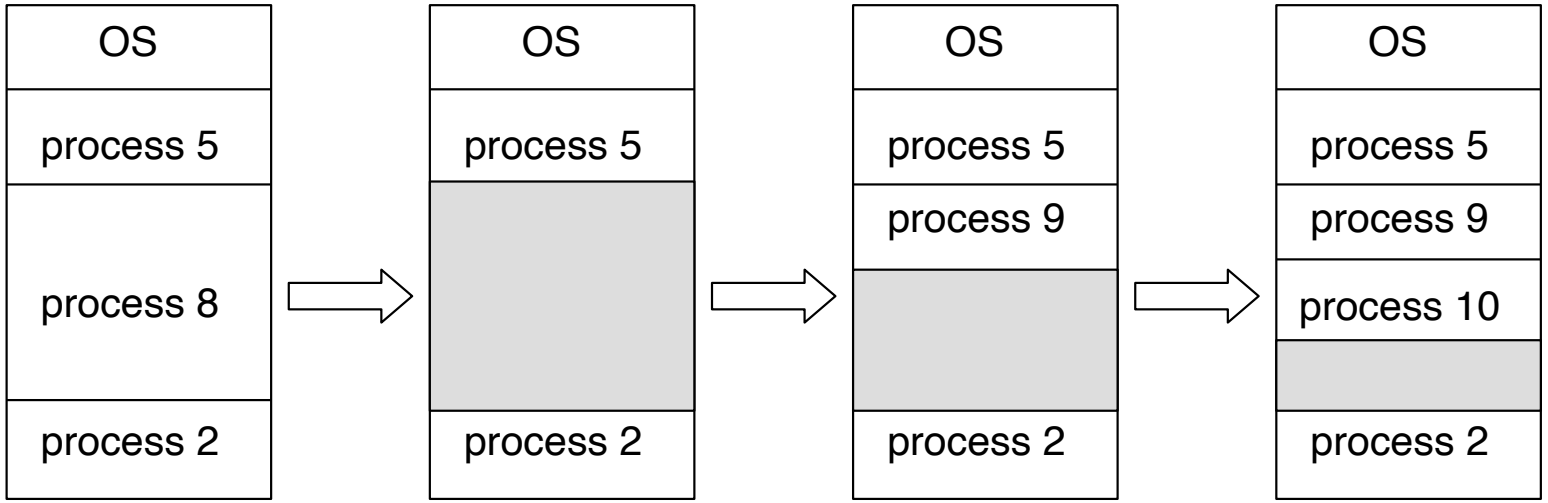
HARDWARE SUPPORT FOR RELOCATION AND LIMIT REGISTERS



CONTIGUOUS ALLOCATION (CONT.)

Multiple-partition allocation

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



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 (according to simulations)

FRAGMENTATION

External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous

Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time

PAGING

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 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

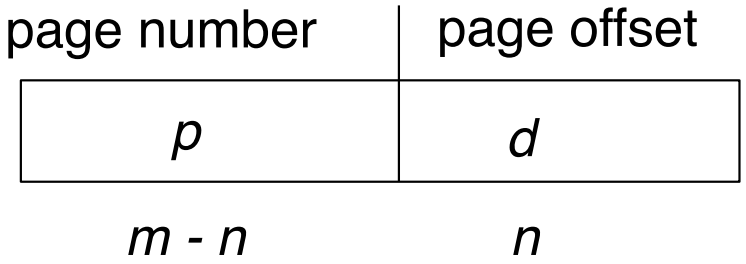
Set up a page table to translate logical to physical addresses

Internal fragmentation

ADDRESS TRANSLATION SCHEME

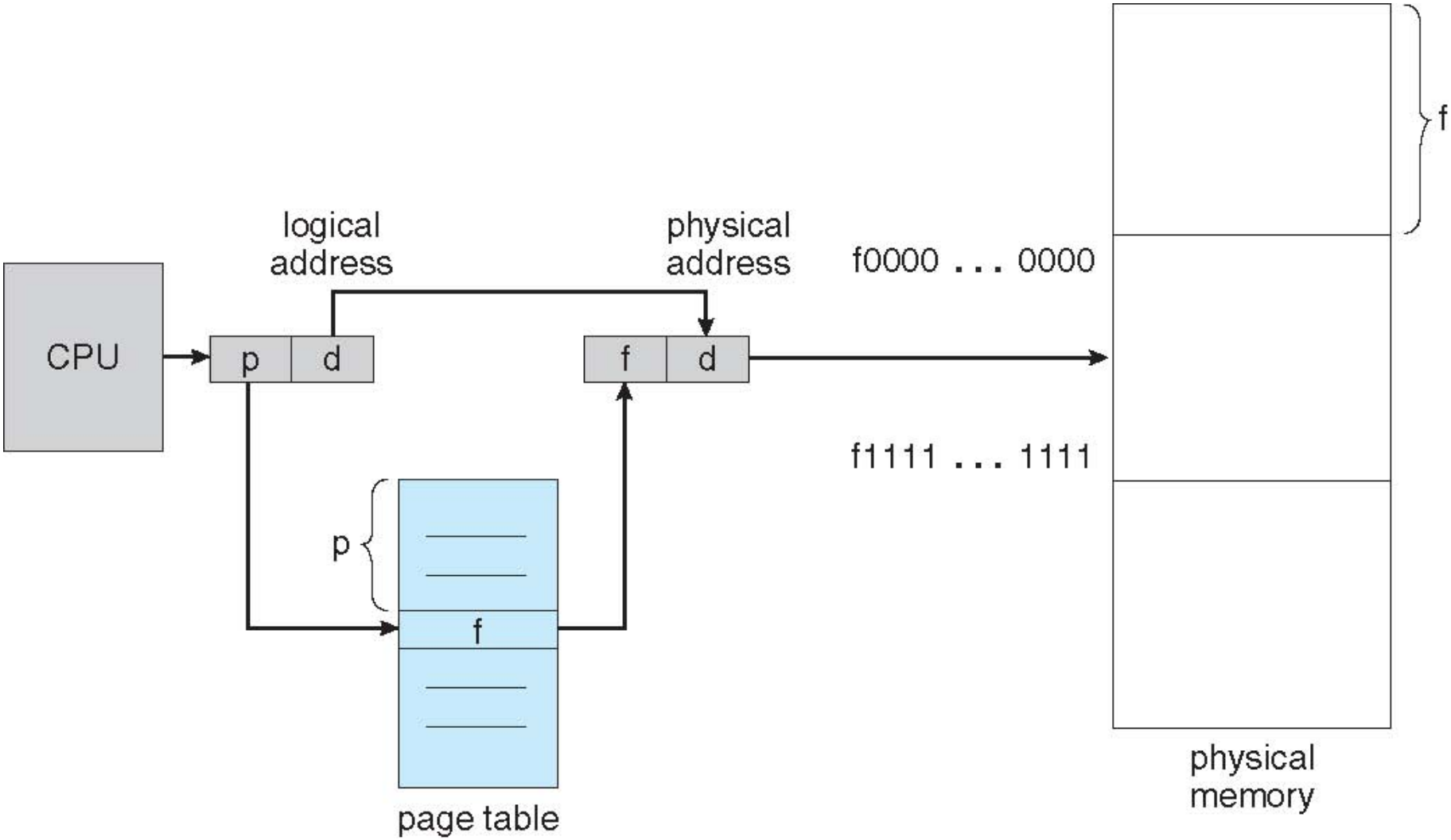
Address generated by CPU is divided into:

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

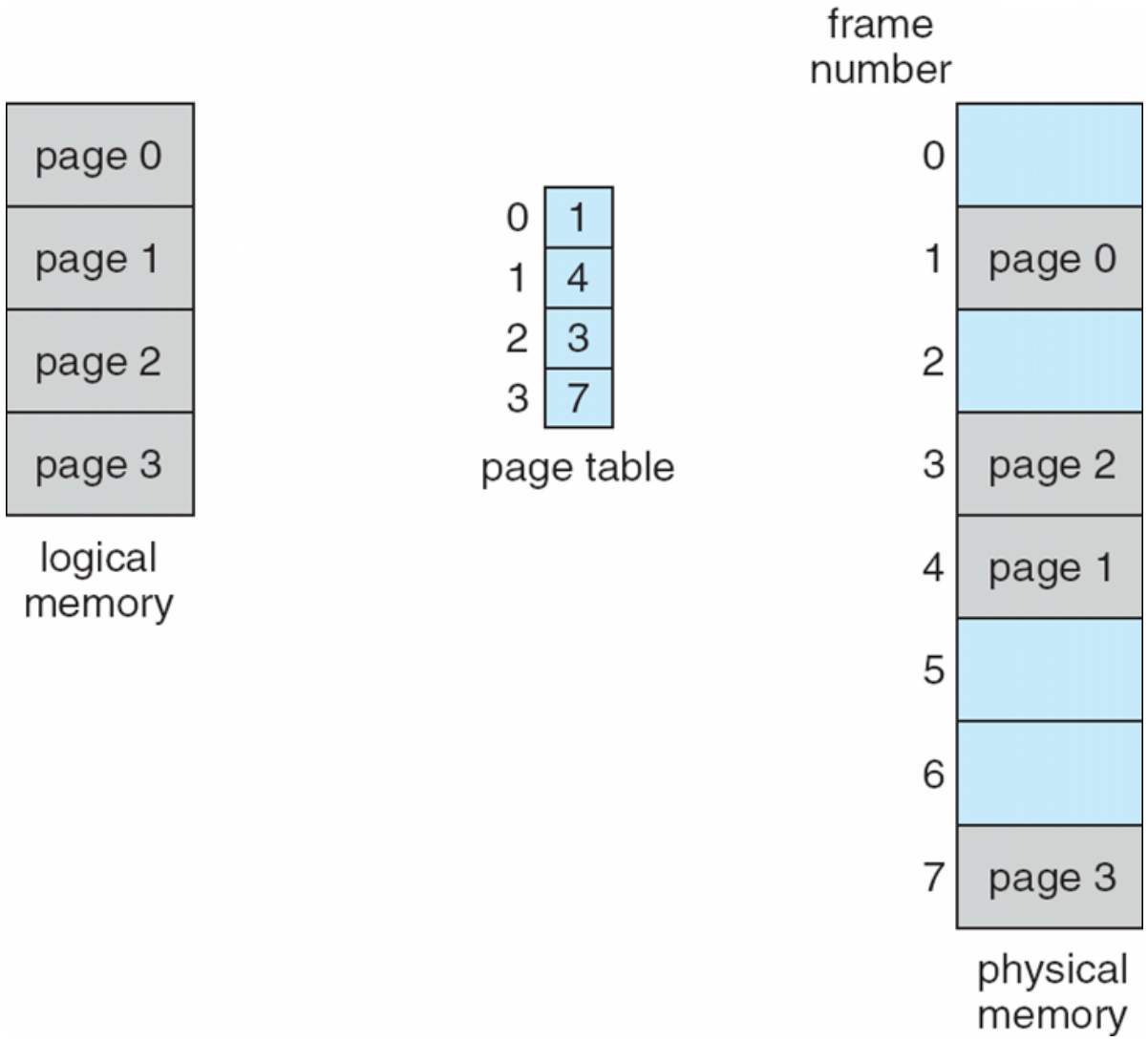


- For given logical address space 2^m *and page size* 2^n

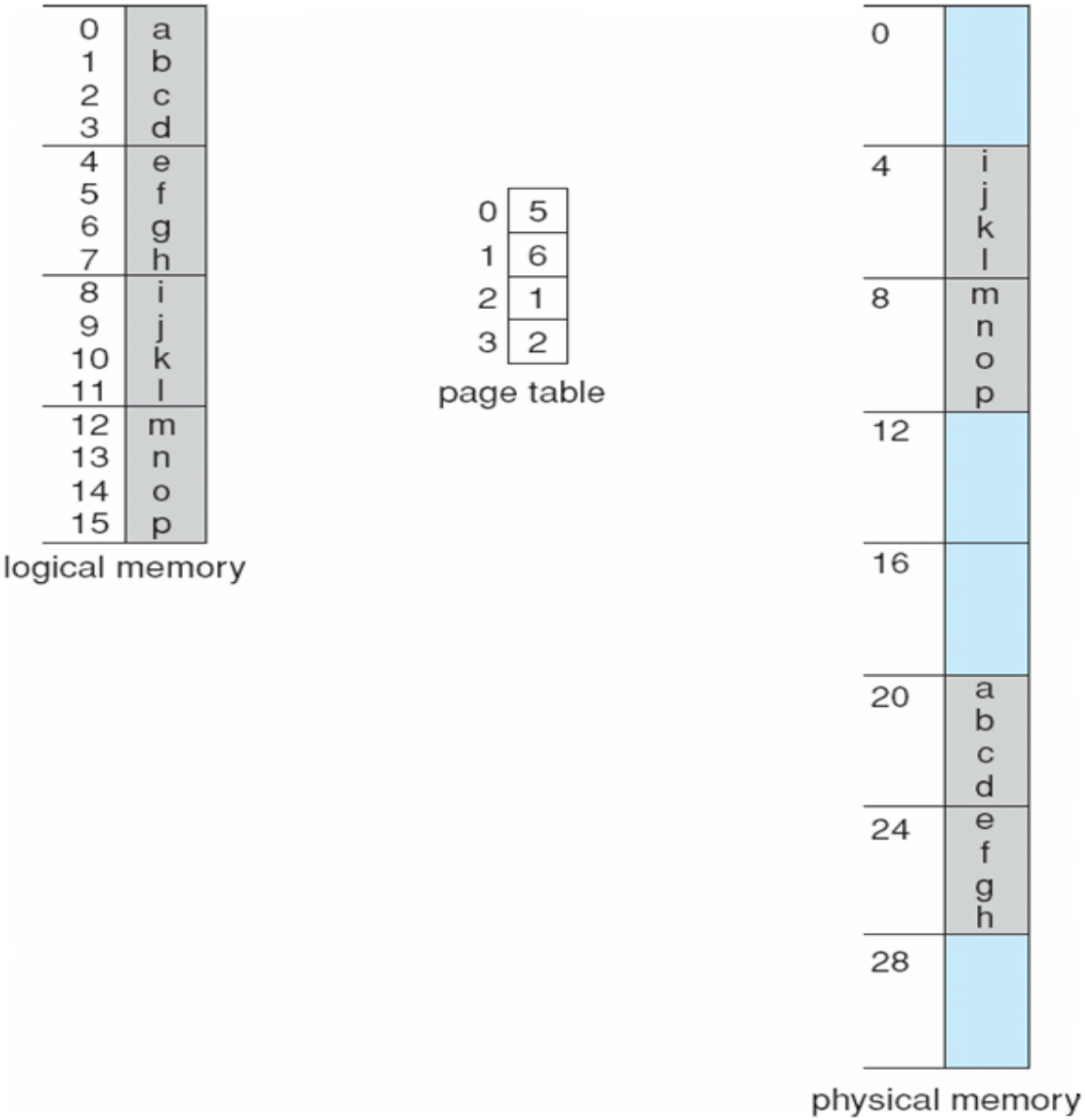
PAGING HARDWARE



PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY



PAGING EXAMPLE



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

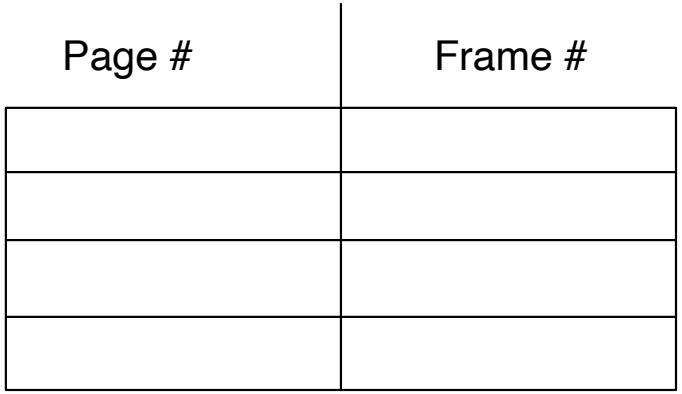
In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

ASSOCIATIVE MEMORY

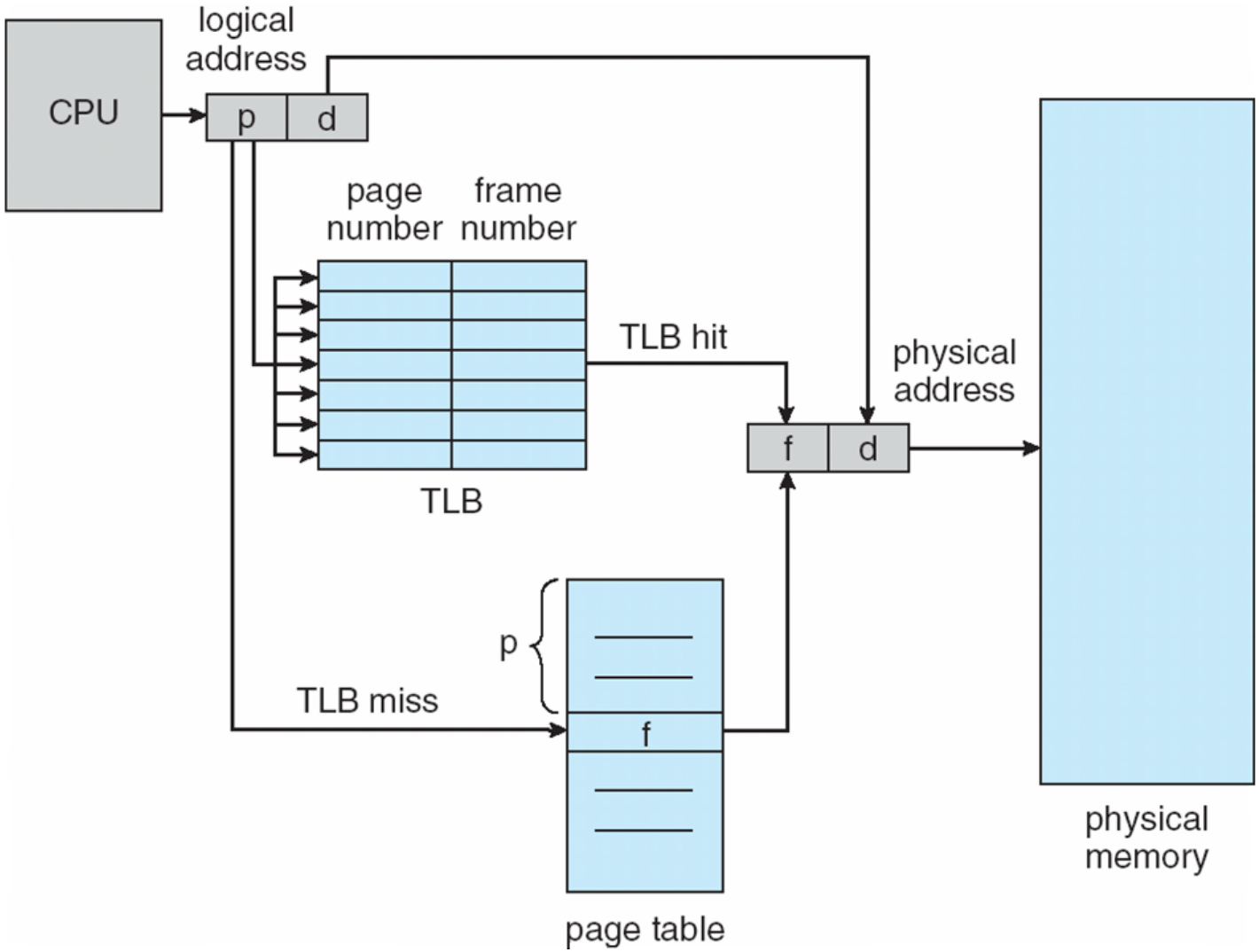
Associative memory – parallel search



Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

PAGING HARDWARE WITH TLB



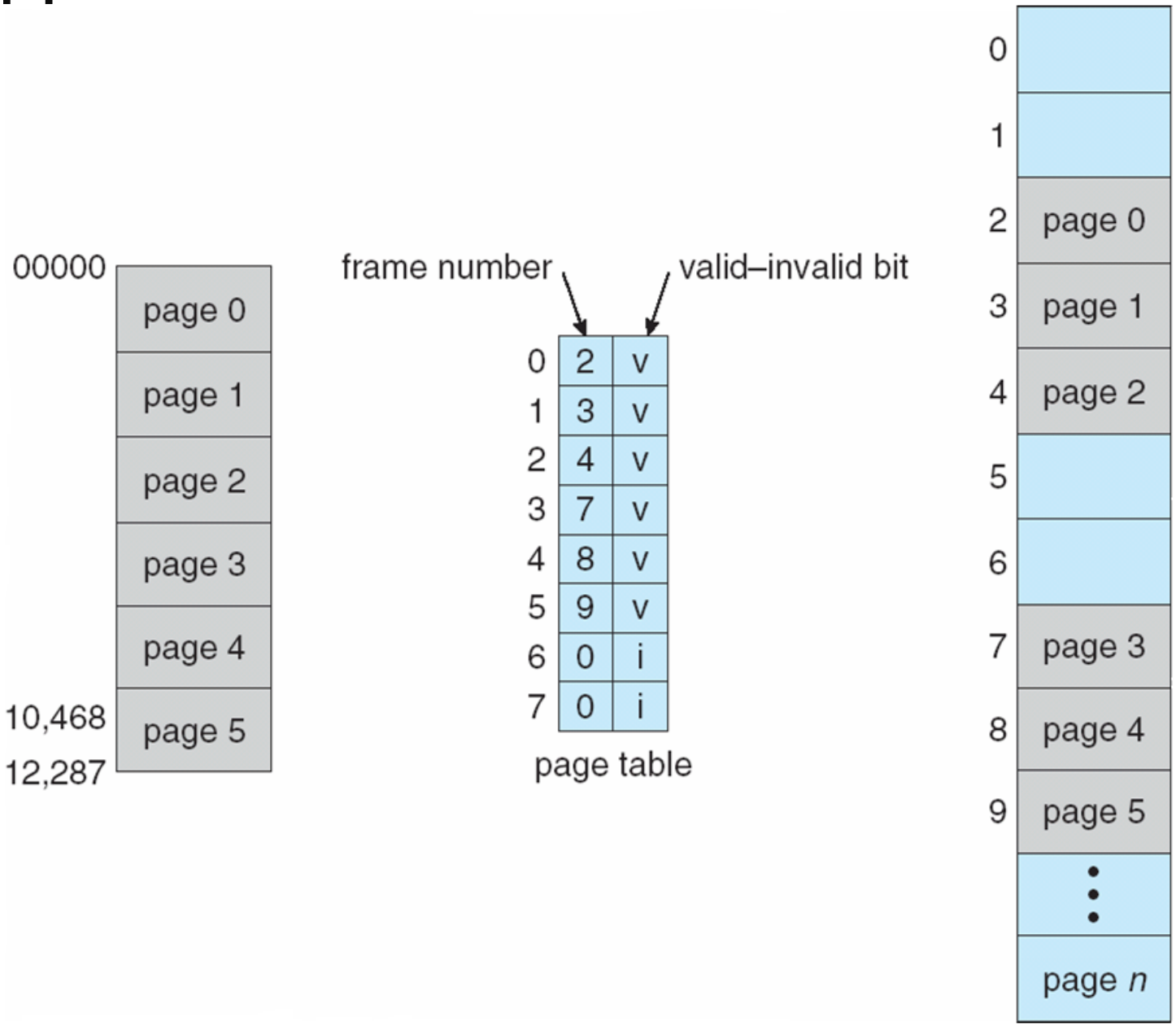
MEMORY PROTECTION

Memory protection implemented by associating protection bit with each frame

Valid-invalid bit attached to each entry in the page table:

- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space

VALID (V) OR INVALID (I) BIT IN A PAGE TABLE



SHARED PAGES

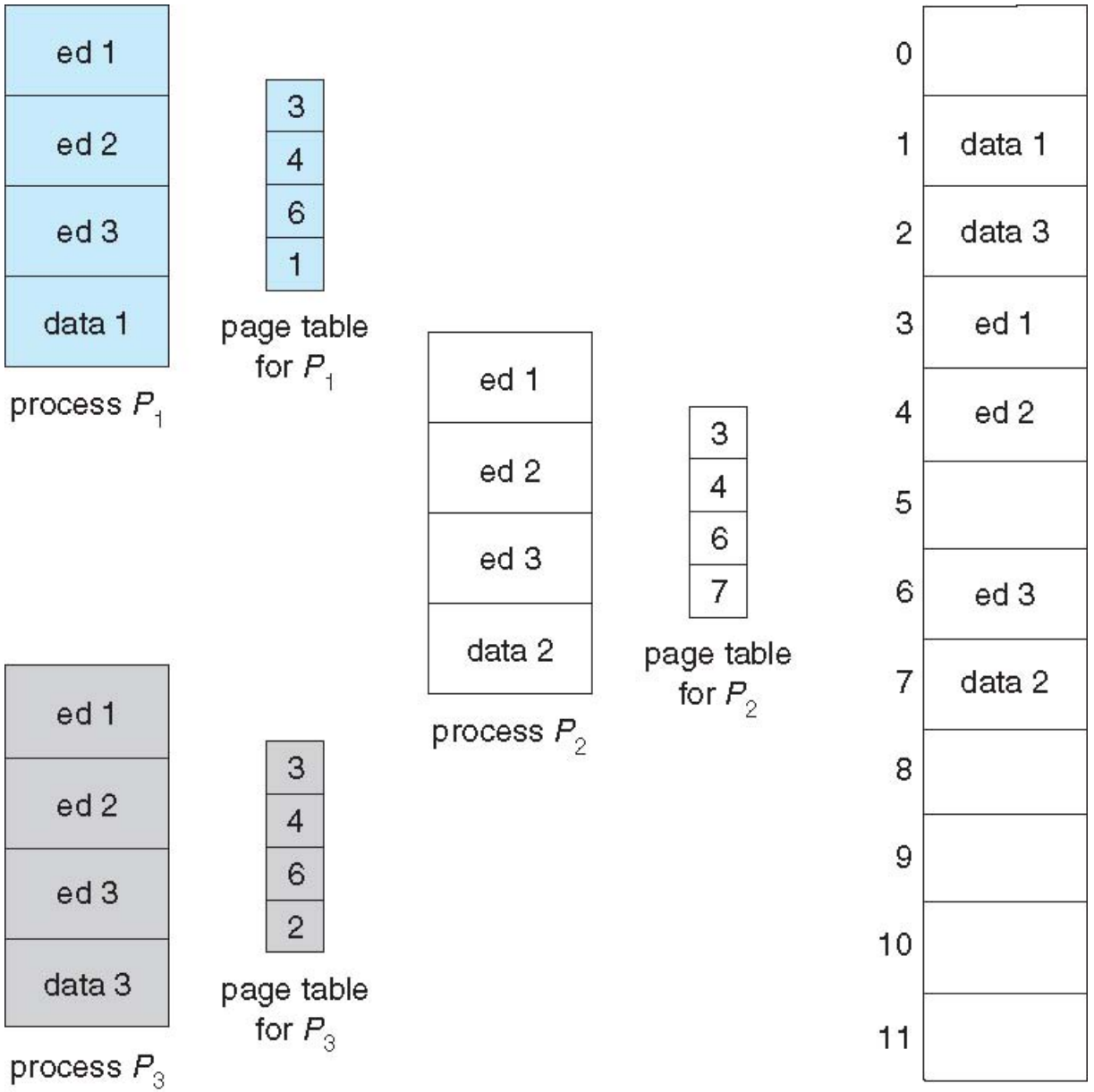
Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

SHARED PAGES EXAMPLE



SEGMENTATION

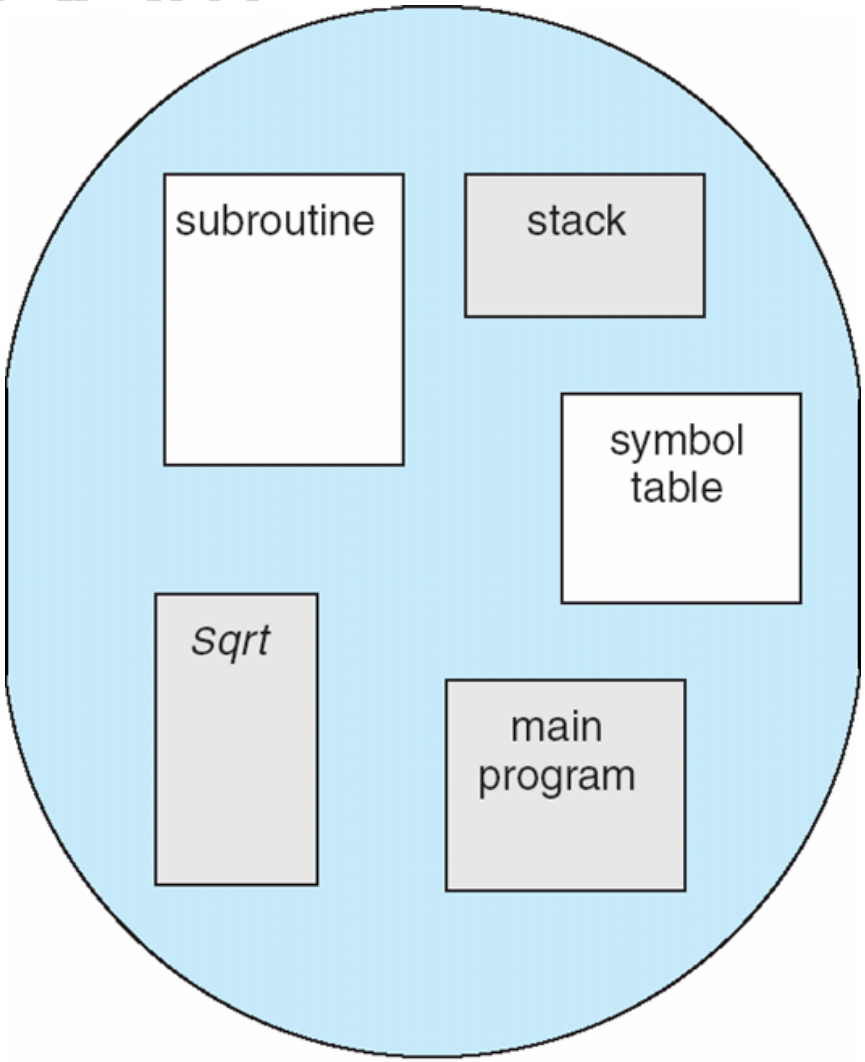
- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- Each segment has a segment name and an offset.
- Logical address consists of a two tuple:

  <segment-number, offset>

- A segment is a logical unit such as:

  procedure
  function
  method
  object
  local variables, global variables
  stack
  symbol table
  arrays

USER'S VIEW OF A PROGRAM



SEGMENTATION ARCHITECTURE

Segment table – maps two-dimensional physical addresses; each table entry has:

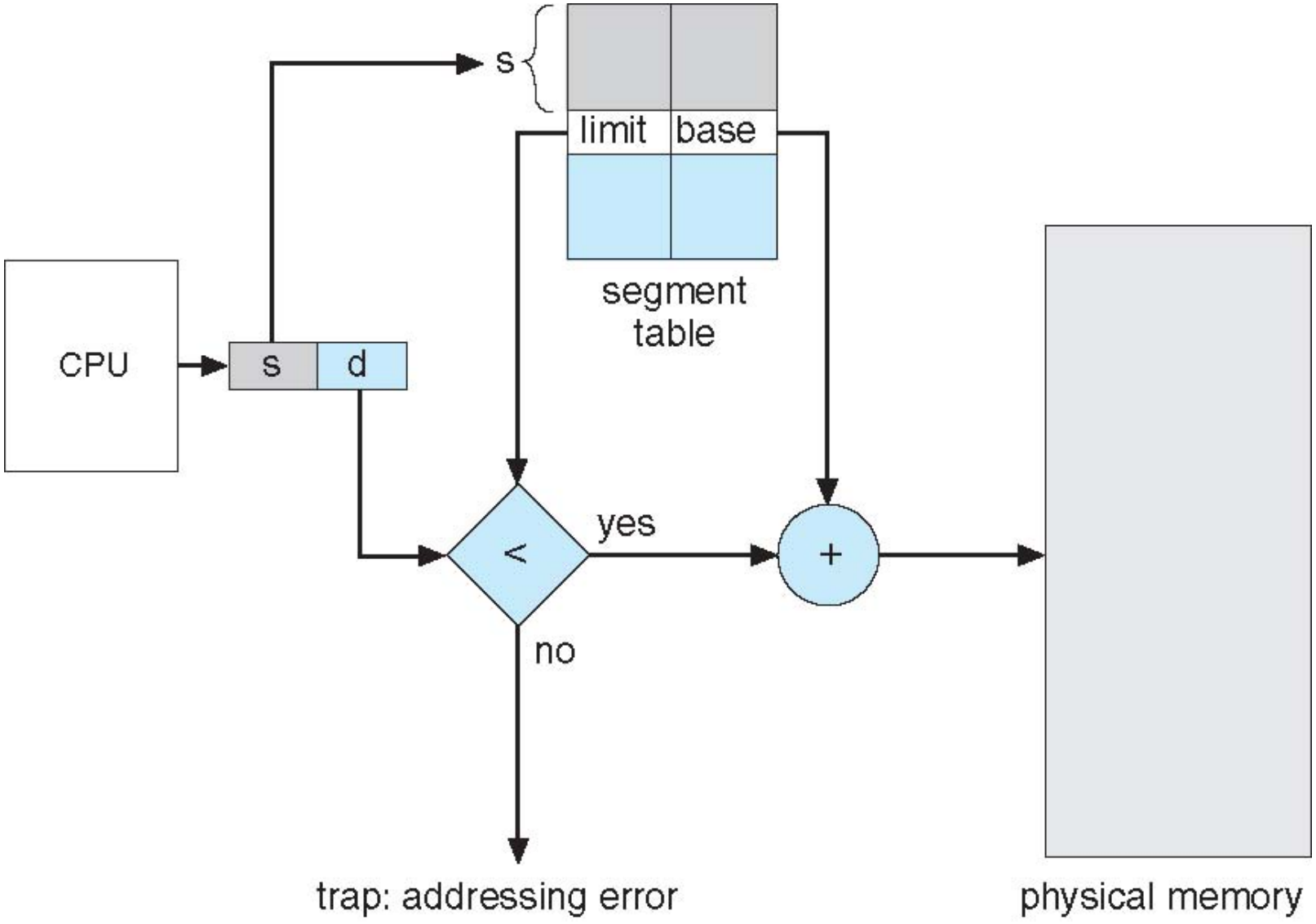
- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory

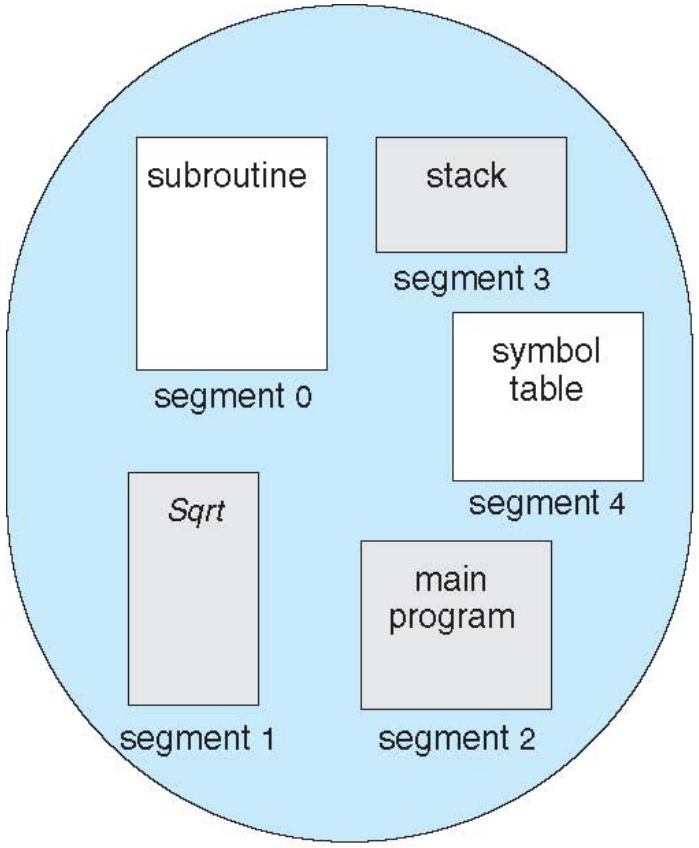
Segment-table length register (STLR) indicates number of segments used by a program;

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

SEGMENTATION HARDWARE

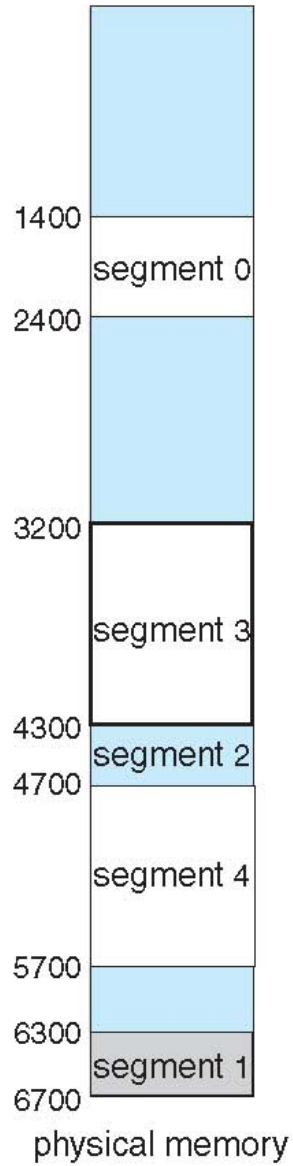


EXAMPLE OF SEGMENTATION



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



SEGMENTATION WITH PAGING

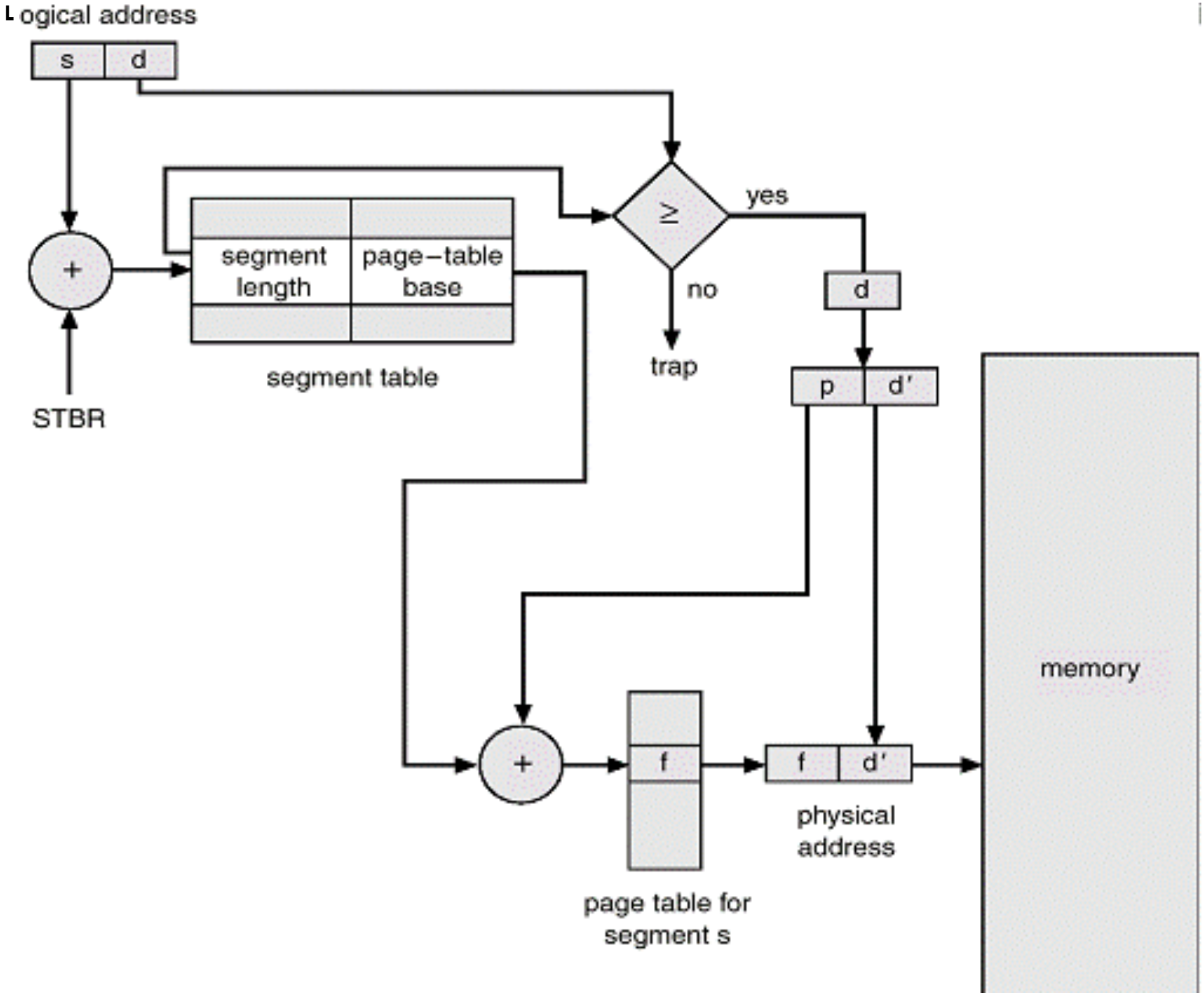
Both paging and segmentation have advantages and disadvantages, that‘s why we can combine these two methods to improve this technique for memory allocation.

- To page the segments.

Each segment is divided into a number of pages of equal size whose information is maintained in a separate page table.

Separate page table is prepared for each segment.

- If a process has four segments that is 0 to 3 then there will be 4 page tables for that process.


Logical address
s
d
+
segment length
page–table base
segment table
STBR
≥
yes
no
trap
d
p
d'
+
f
f
d'
physical address
page table for segment s
memory

END OF CHAPTER

Virtual Memory

Background

- Instructions must be in physical memory to be executed.
  - But it limits the size of a program to the size of physical memory.
- There are situations where the entire program is not needed.
  - Programs handling unusual error conditions
  - Arrays, lists and tables are often allocated more memory than needed.

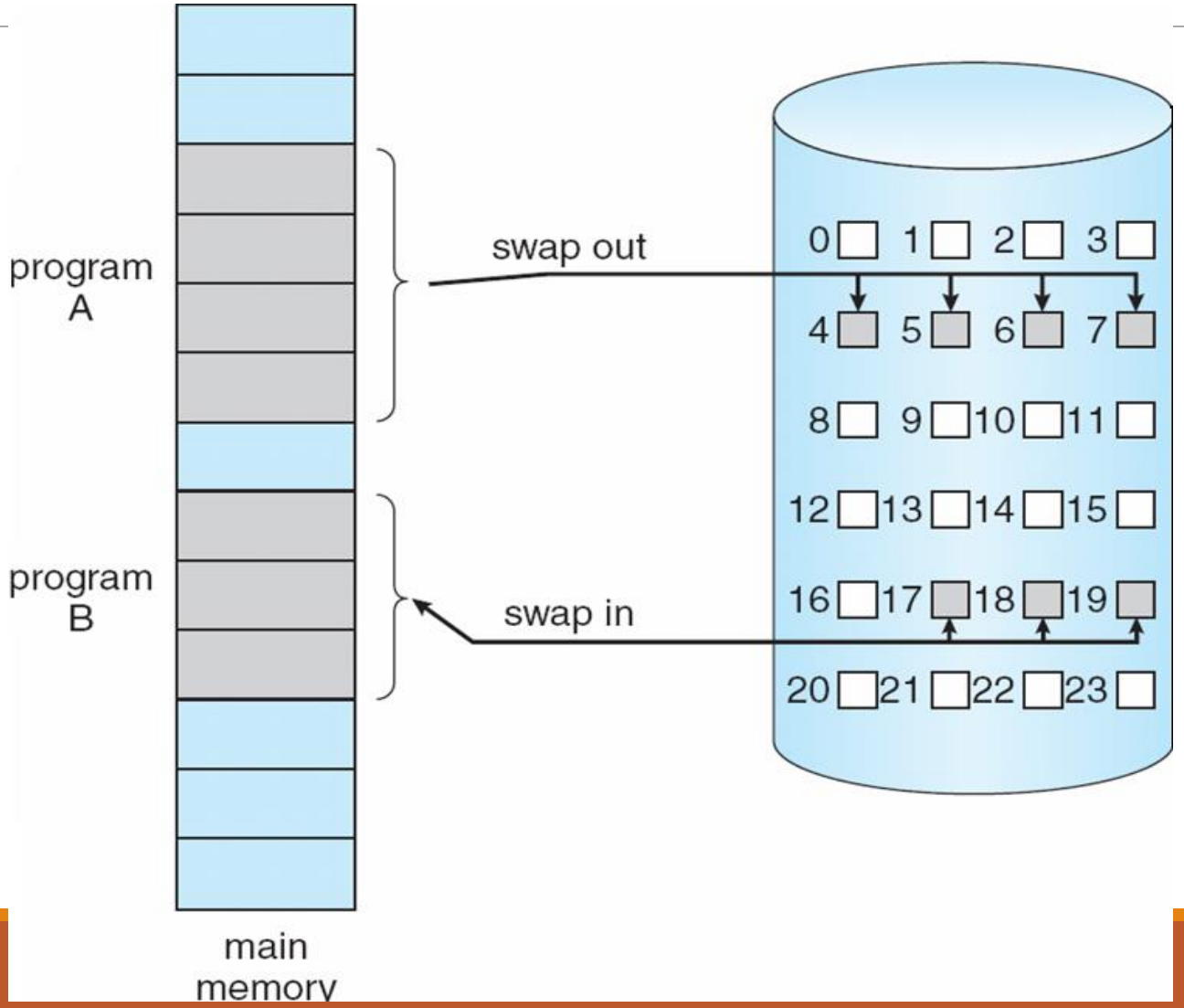
Introduction

- **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.
  - A program would no longer be constrained by the amount of physical memory.
- Virtual address space of a process refers to the logical(or virtual) view of how a process is stored in memory.

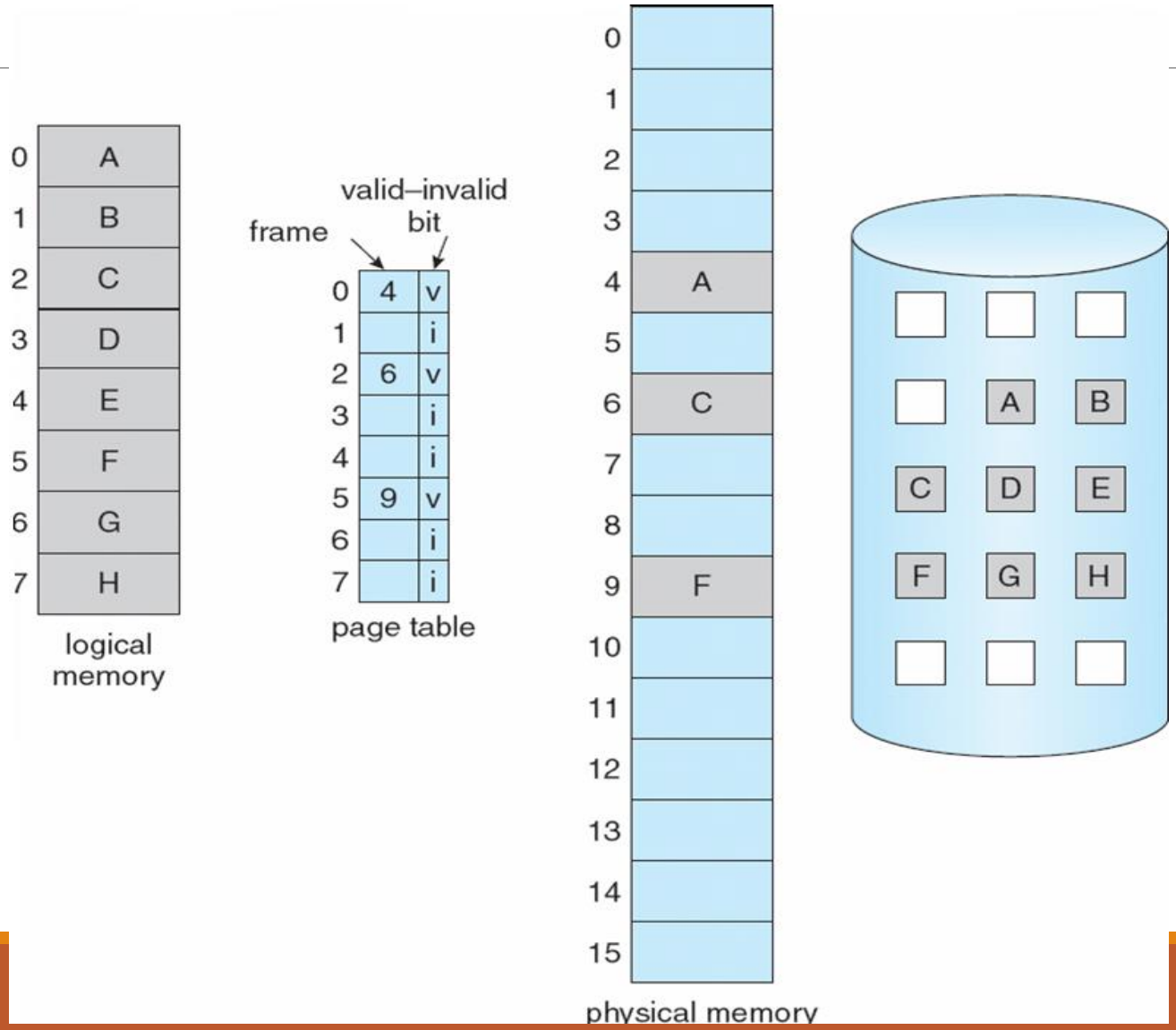
Demand Paging

- Bring a page into memory only when it is needed
  - Less memory needed
  - Faster response
  - More users
- Pages that are not accessed are never loaded into memory.
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Page Table When Some Pages Are Not in Main Memory



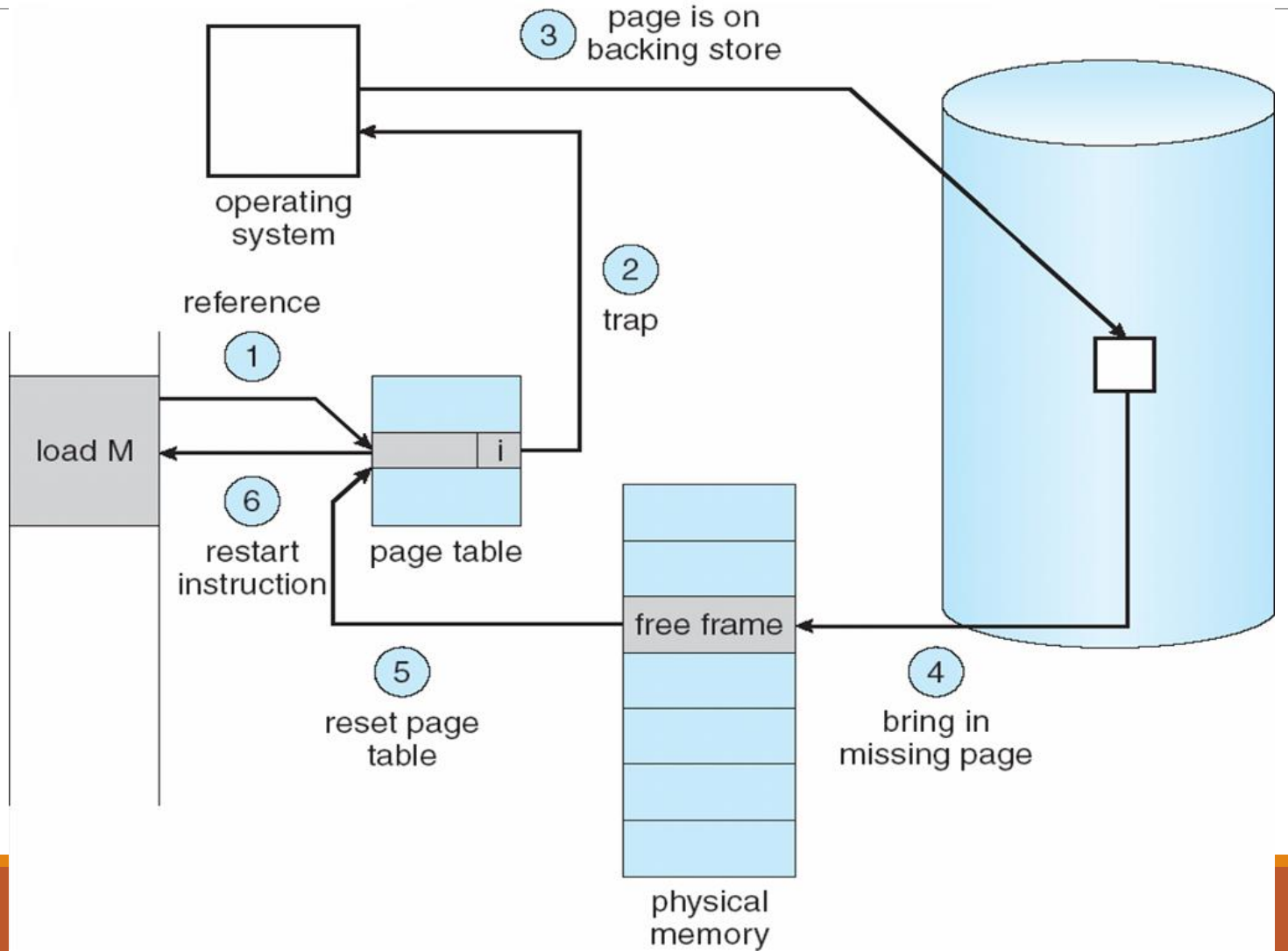
Page Fault

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 ⇒ 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

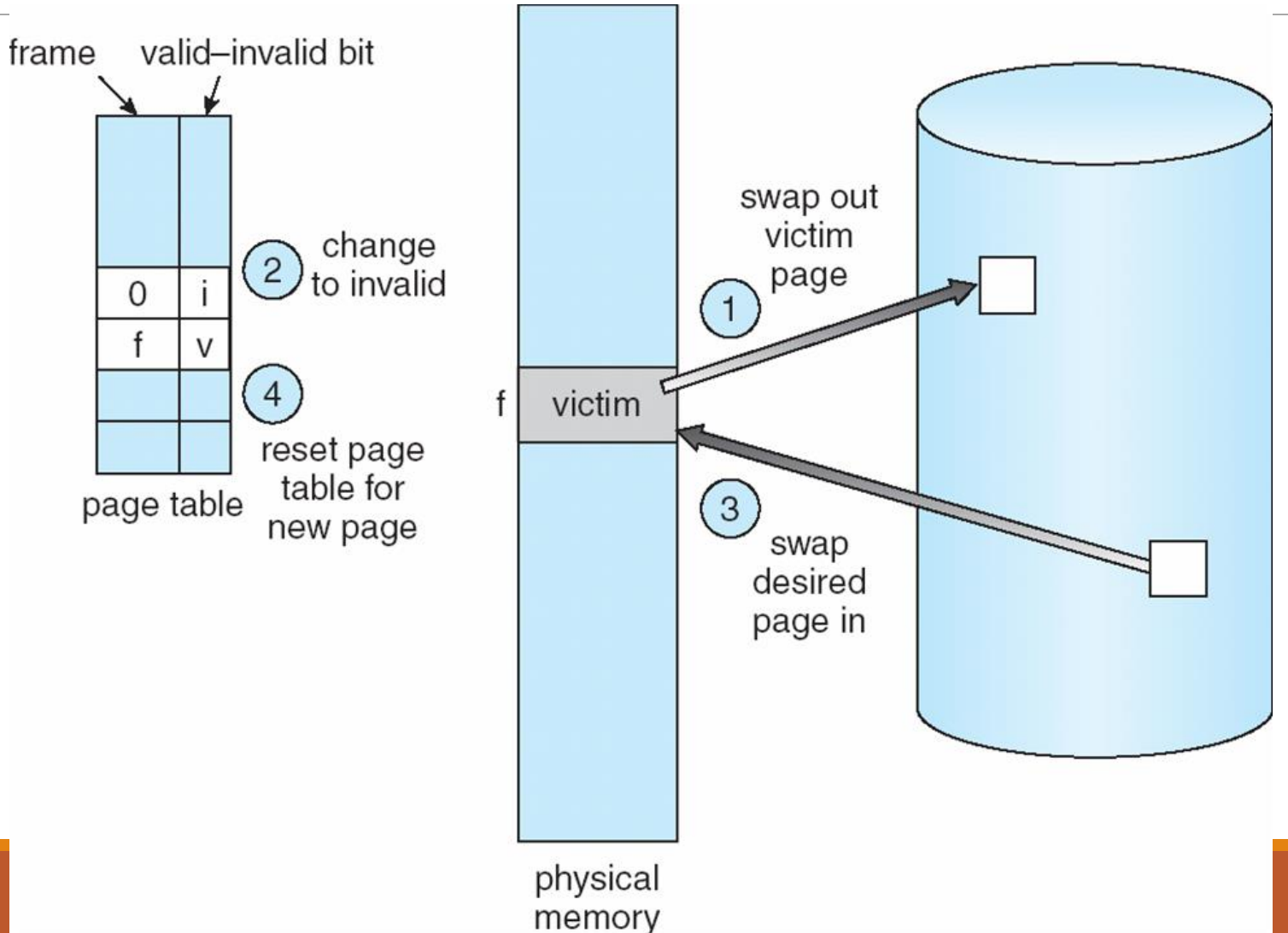
Steps in Handling a Page Fault



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 process.

Page Replacement



Page Replacement Algorithms

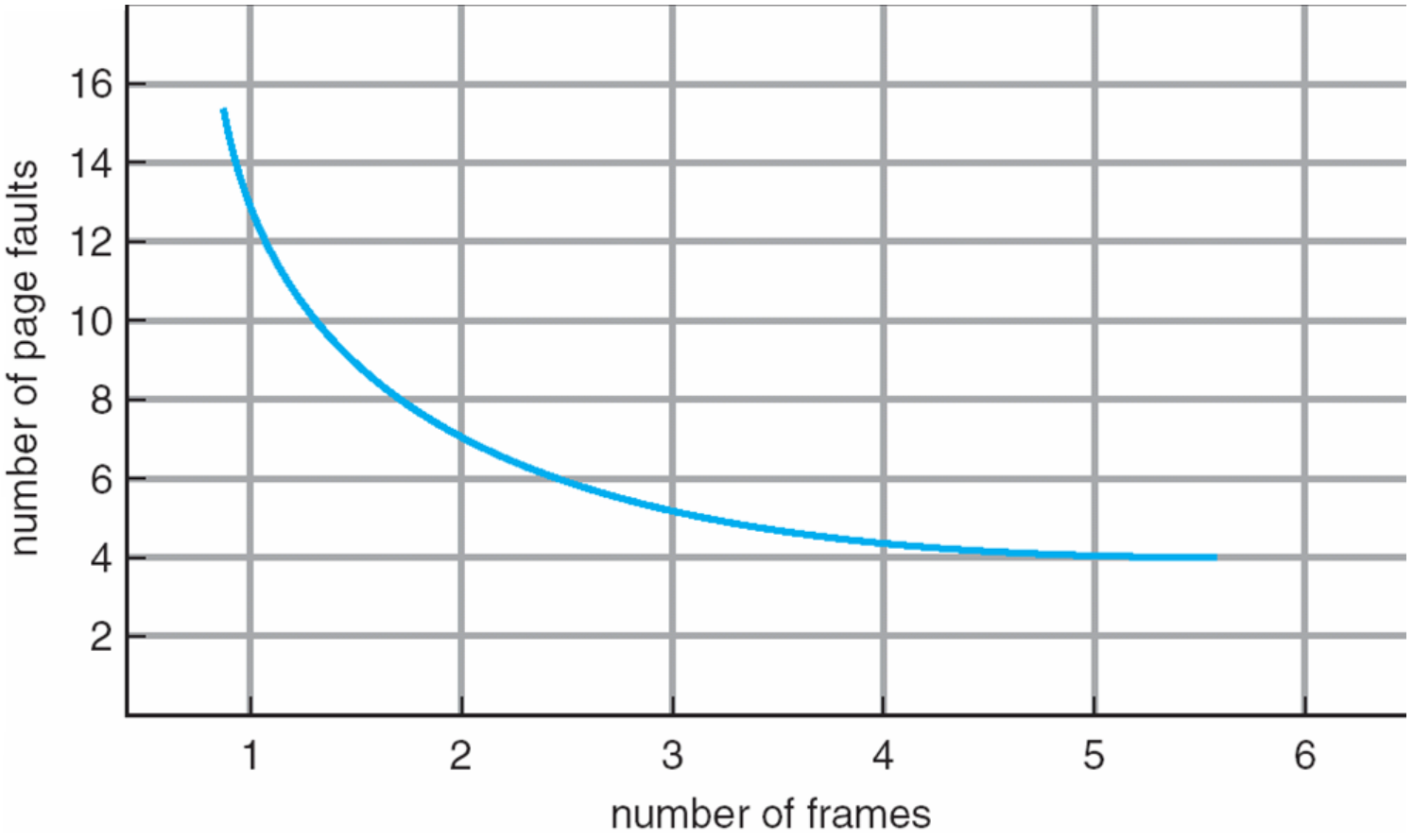
Want lowest page-fault rate

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

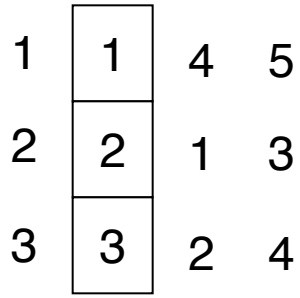
Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

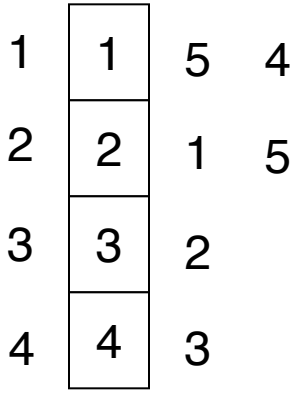
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames



9 page faults

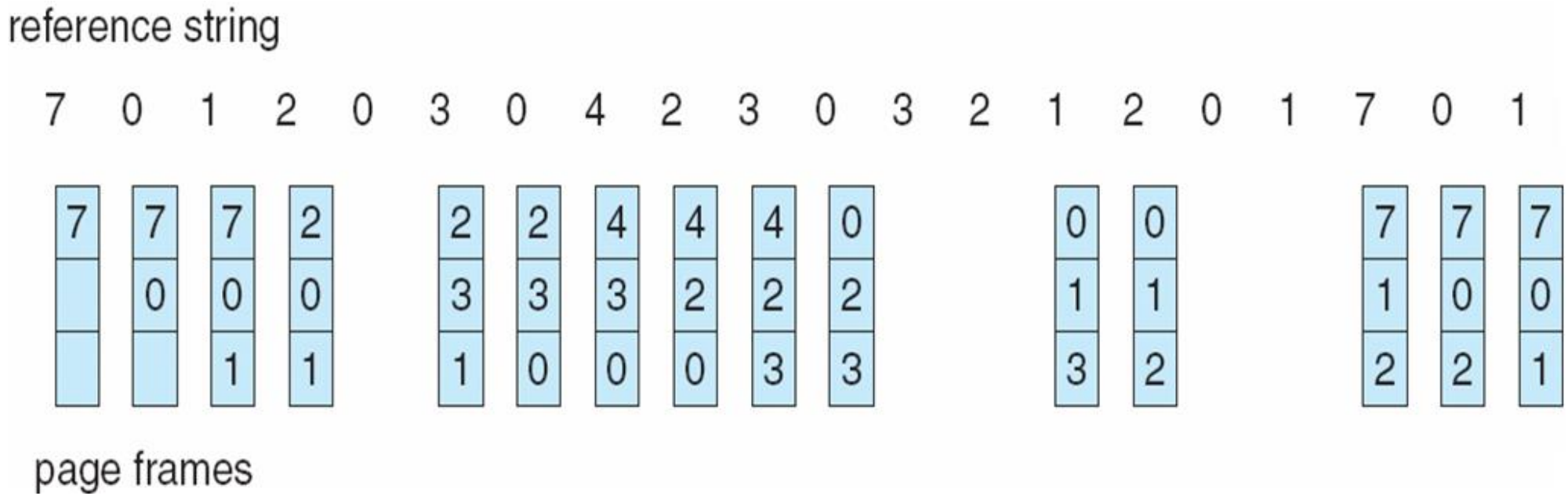
4 frames



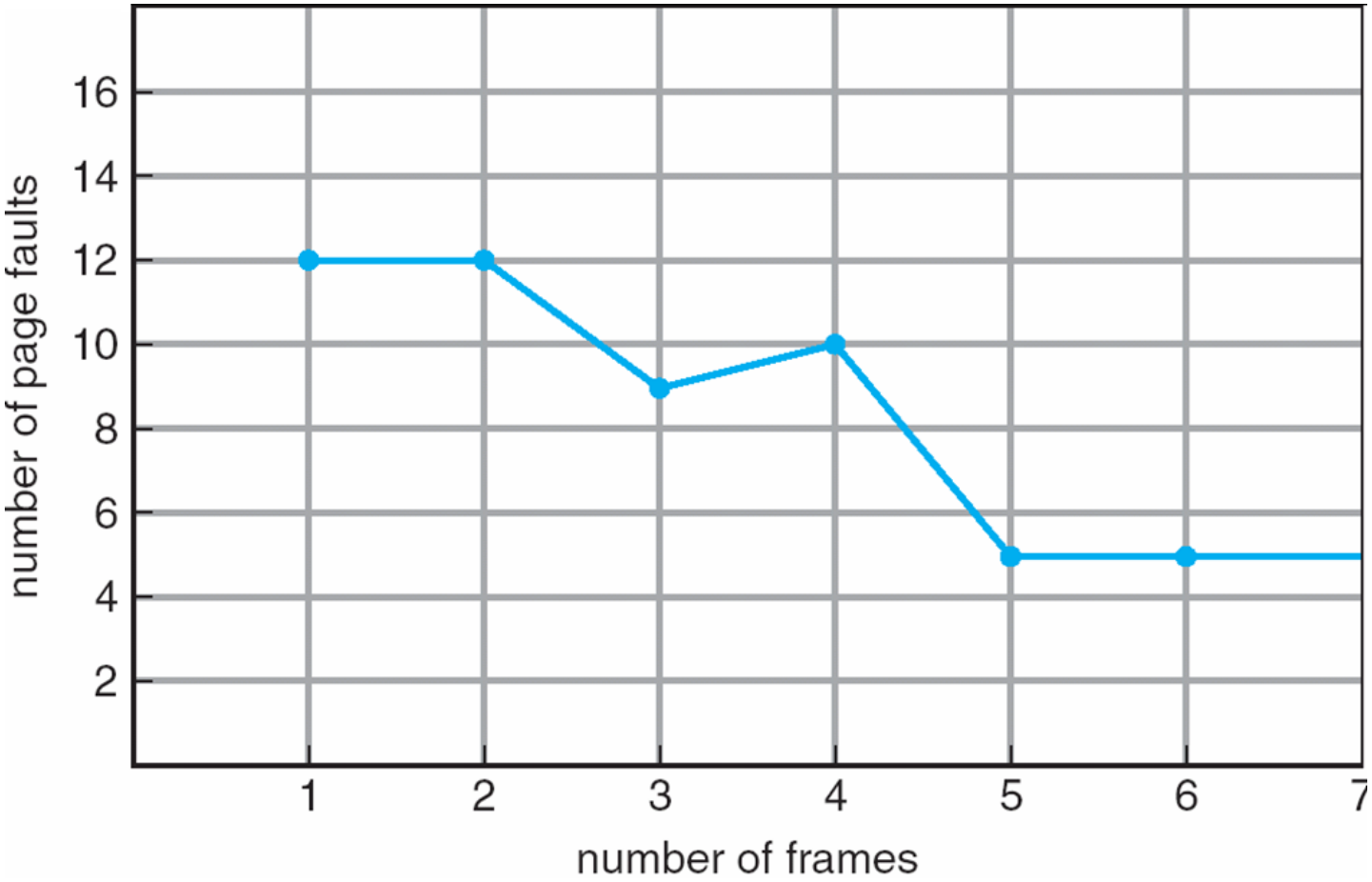
10 page faults

Belady's Anomaly: Adding more frames can cause more page faults!

FIFO Page Replacement



FIFO Illustrating Belady's Anomaly



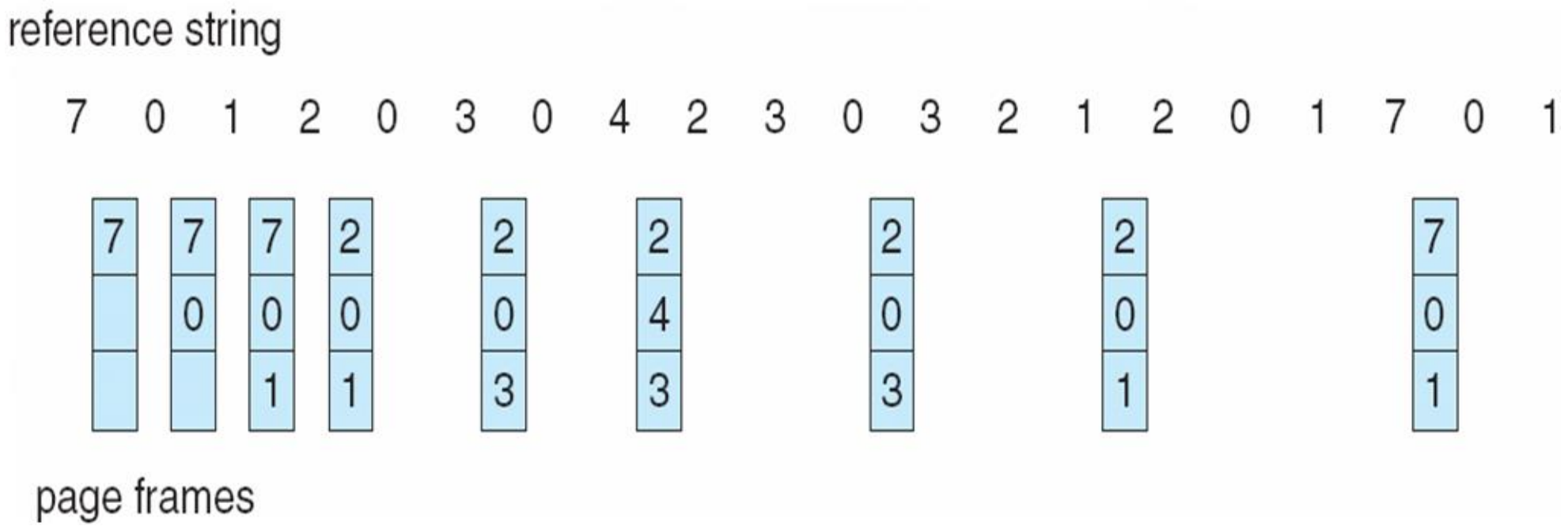
Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms.

Replace the page that will not be used

for the longest period of time.

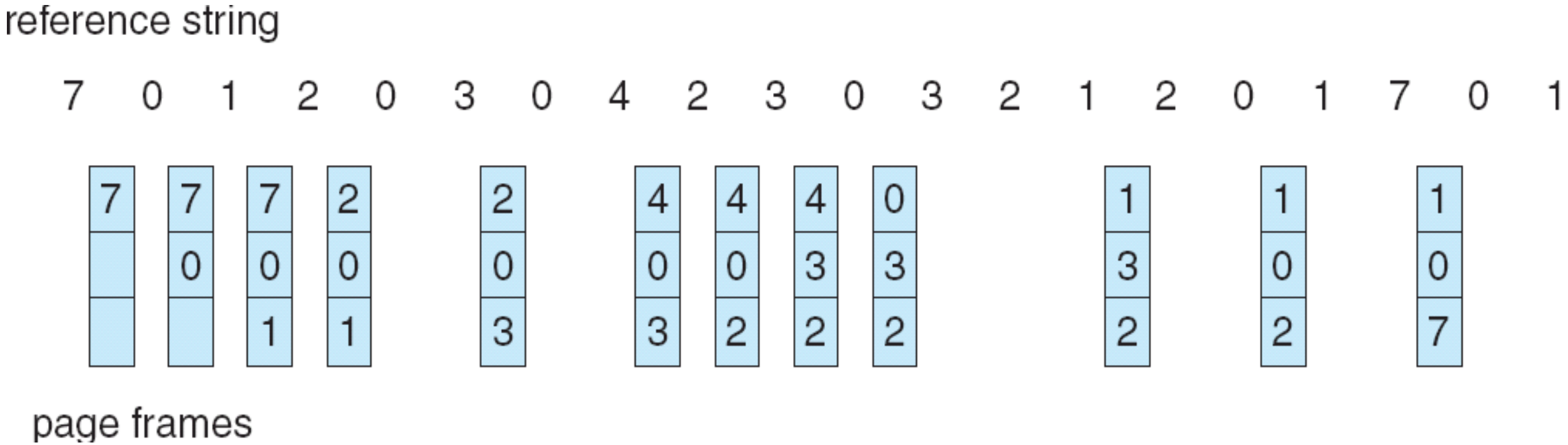
Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- LRU associates with each page the time of that page's last reference or use.
- LRU chooses the page that has not been used for the longest period of time.
- Can be thought of as the optimal page replacement looking backward in time.

LRU Page Replacement



LRU Approximation Page Replacement

LRU needs special hardware.

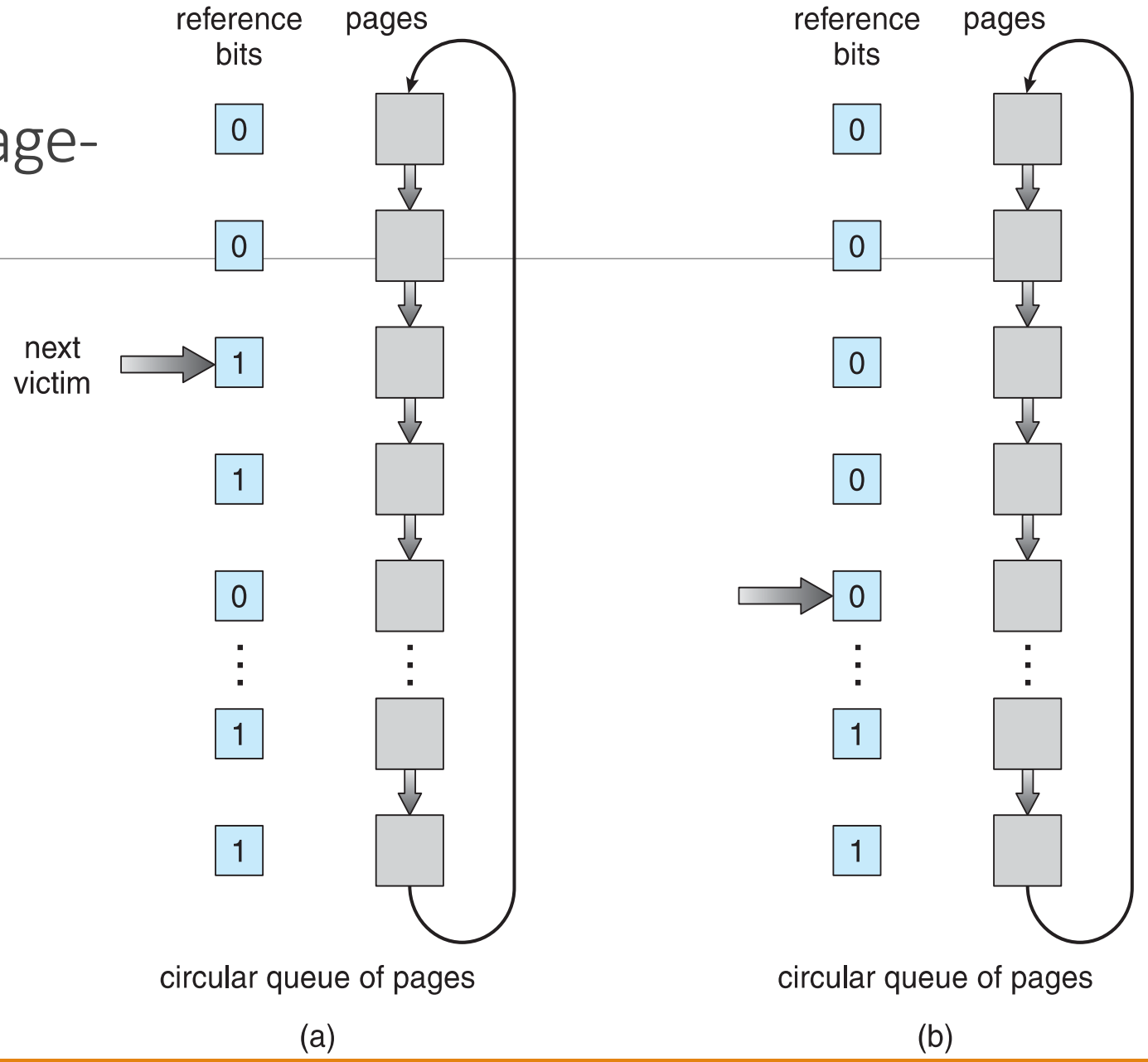
Reference bit

- With each page associate a bit, initially = 0
- When page is referenced, bit is set to 1
- Replace any page with reference bit = 0 (if one exists)

Second-chance algorithm

- If page to be replaced has
  - Reference bit = 0 -> replace it
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Allocation of Frames

Each process needs *minimum* number of pages

- Number of pages allocated to each process decreases, the page fault rate increases, slowing the process execution.
- Example: IBM 370 – 6 pages to handle SS MOVE instruction.

Two major allocation schemes

- fixed allocation
- priority allocation

Fixed Allocation

Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.

Proportional allocation – Allocate according to the size of process

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

$$S = \sum s_i$$

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

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

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

Use a proportional allocation scheme using priorities rather than size

If process P_i generates a page fault,

- select for replacement one of its frames
- select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

Local replacement – each process selects from only its own set of allocated frames

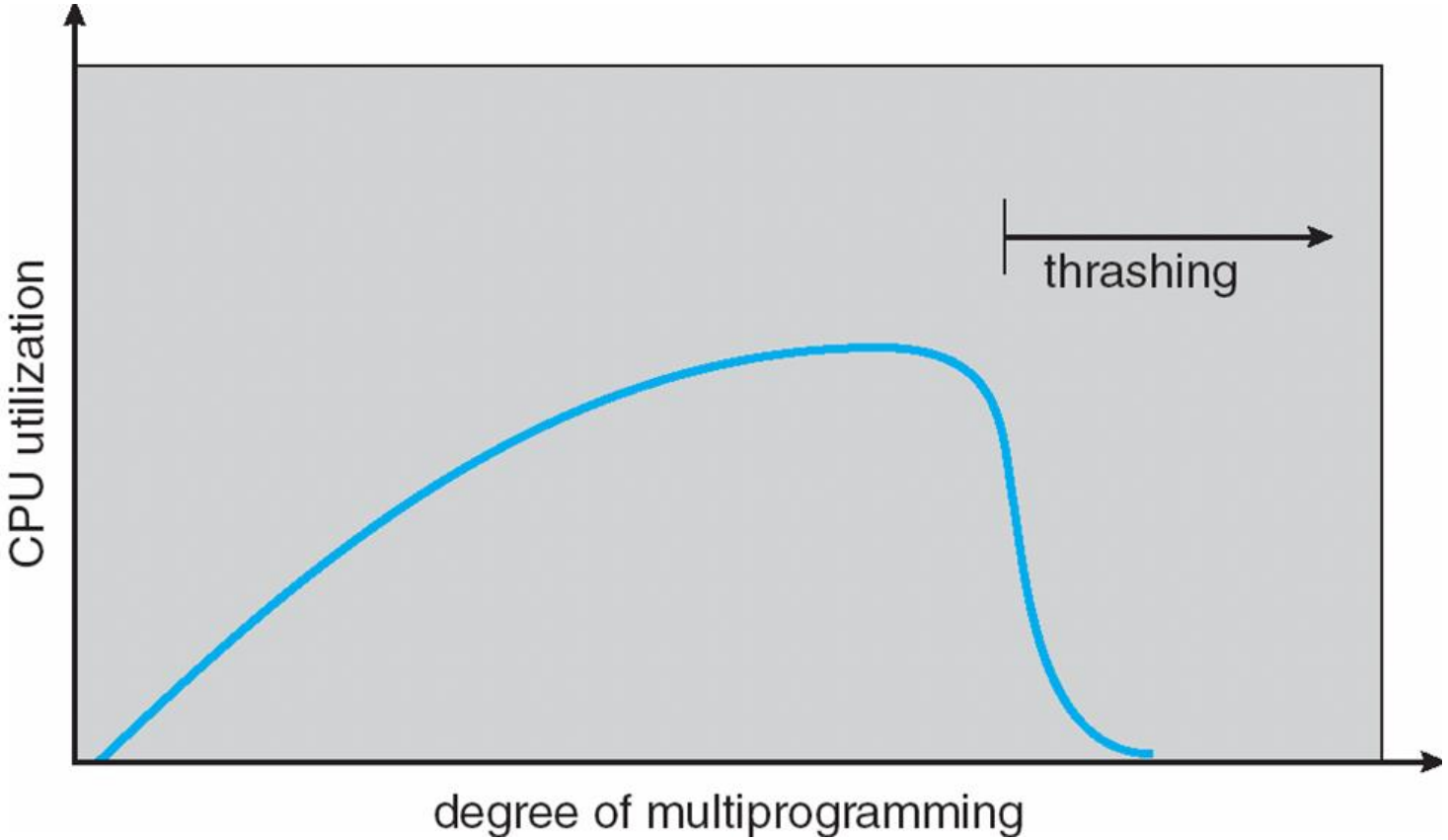
Thrashing

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…further lowering the CPU utilization.

Thrashing ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



End of Chapter

File Management

Files

- A file is a container for a collection of information.
- File represents programs and data. Data files may be numeric, alphabetic, binary or alpha numeric.
- Four terms are used for files
  - Field
  - Record
  - Database

File Attributes

File attributes vary from one operating system to another. The common attributes are:

- **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.

File Operations

Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

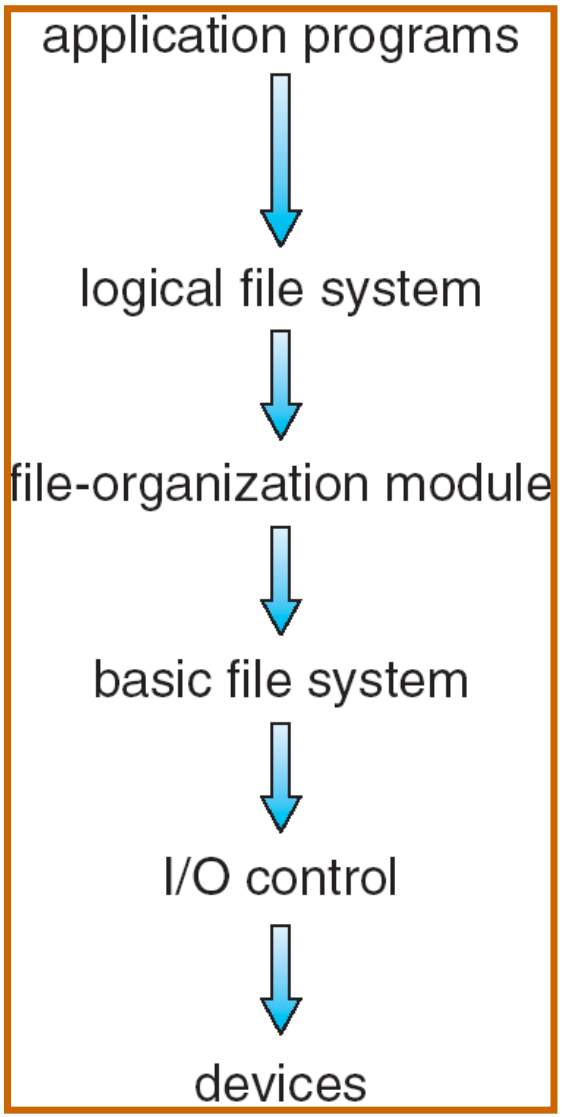
- **Create**
- **Delete**
- **Open**
- **Close**
- **Read**
- **Write**

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

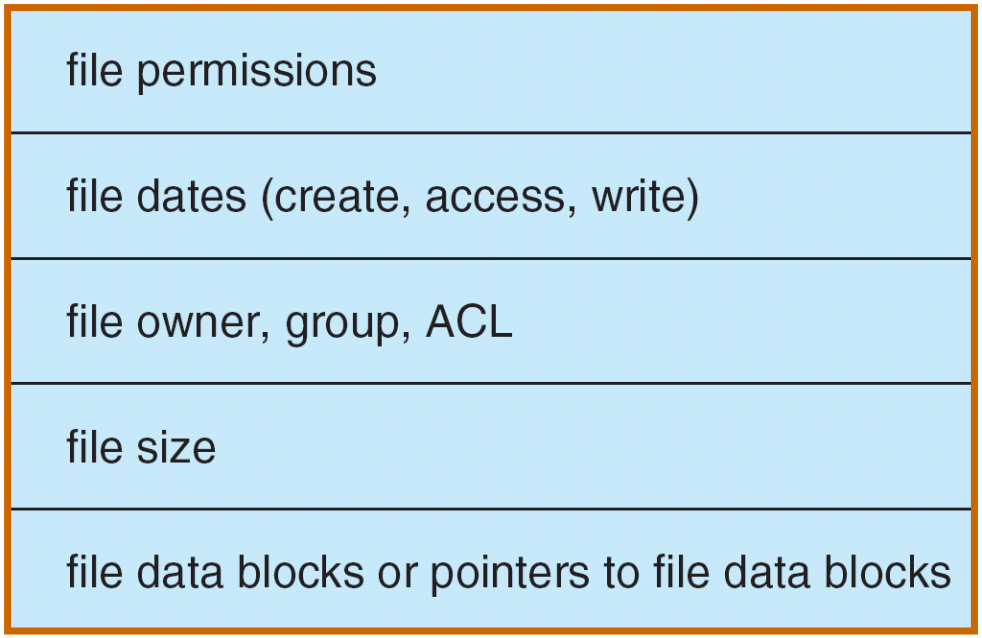
Layered File System

- Application programs
- Logical file system
  - Contains the metadata
  - Maintains the file structure via file control blocks
- File-organization module
  - Maps the logical blocks to the physical blocks
  - Manages the free space/blocks on the disk
- Basic file system
  - Issues generic commands to the appropriate device driver to read and write physical blocks on the disk
- I/O control
  - Consists of device drivers and interrupt handlers
  - Translates generic commands into hardware instructions



Layered File System

A file control block (FCB) contains information about the file, including ownership, permissions, and location of the file contents



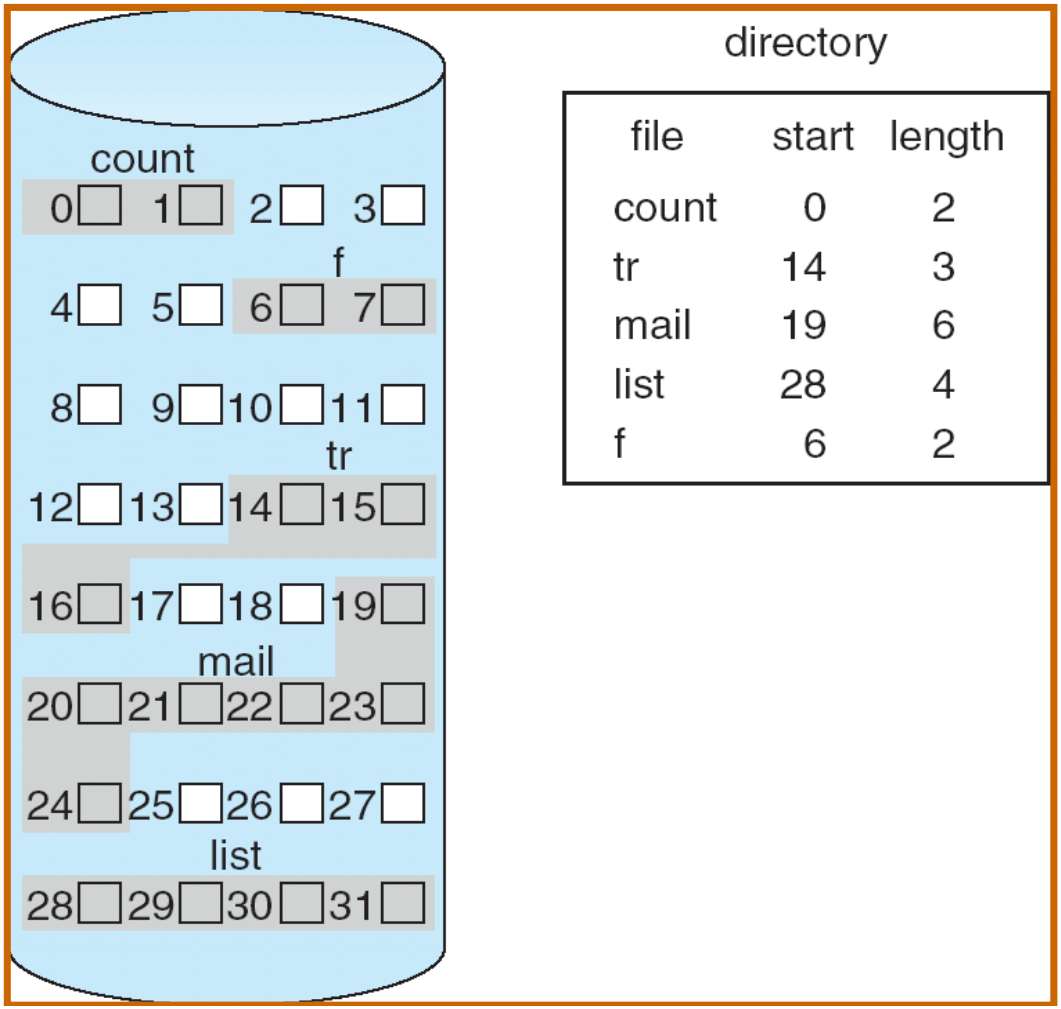
Allocation Methods

- Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly
- Three methods exist for allocating disk space
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

Contiguous Allocation

- Requires that each file occupy a set of contiguous blocks on the disk
- Accessing a file is easy – only need the starting location (block #) and length (number of blocks)
- Problems
  - Finding space for a new file (first fit, best fit, etc.)
  - External fragmentation (free space is broken into small unusable chunks)
  - Need for compaction
  - Determining space for a file, especially if it needs to grow

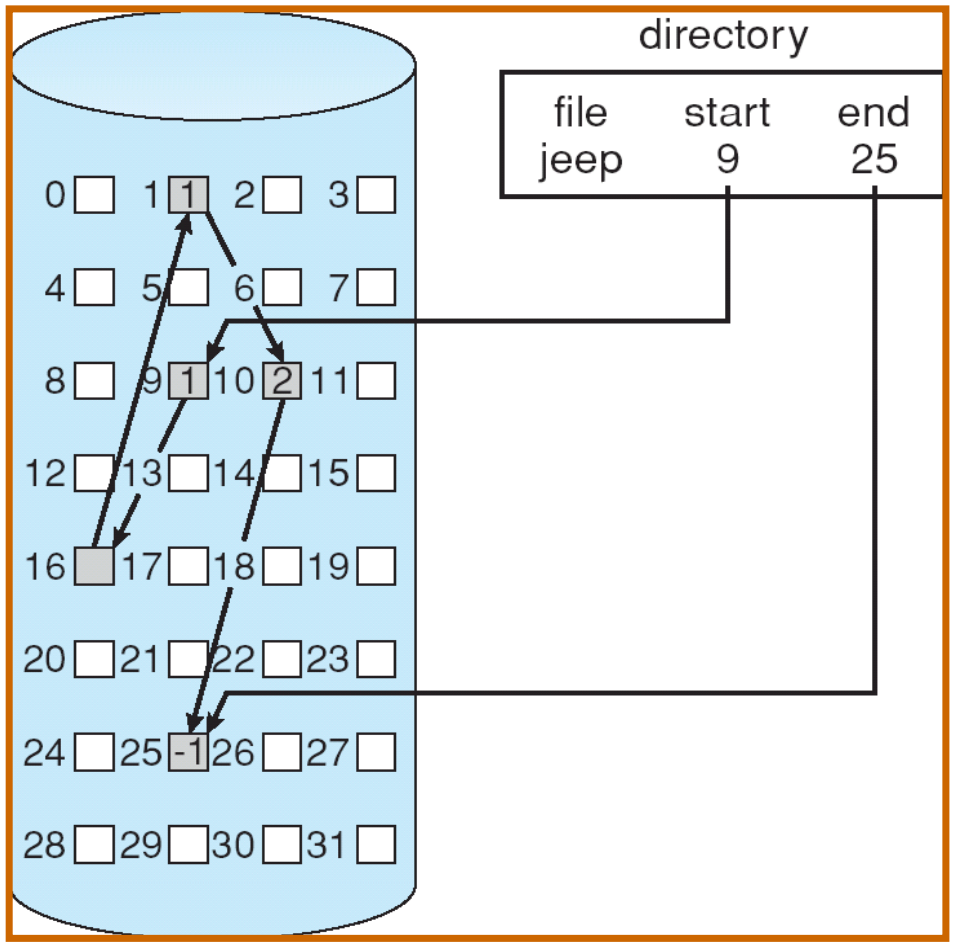
Contiguous Allocation (continued)



Linked Allocation

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block
  - This new block is written to and is linked to the end of the file
- Reading from a file requires only reading blocks by following the pointers from block to block
- Advantages
  - There is no external fragmentation
  - Any free blocks on the free list can be used to satisfy a request for disk space
  - The size of a file need not be declared when the file is created
  - A file can continue to grow as long as free blocks are available
    - It is never necessary to compact disk space for the sake of linked allocation (however, file access efficiency may require it)

Linked Allocation (continued)



Linked Allocation (continued)

Disadvantages

- Can only be used effectively for sequential access of files
  - Each access to a file block requires a disk access, and some may also require a disk seek
  - It is inefficient to support direct access capability

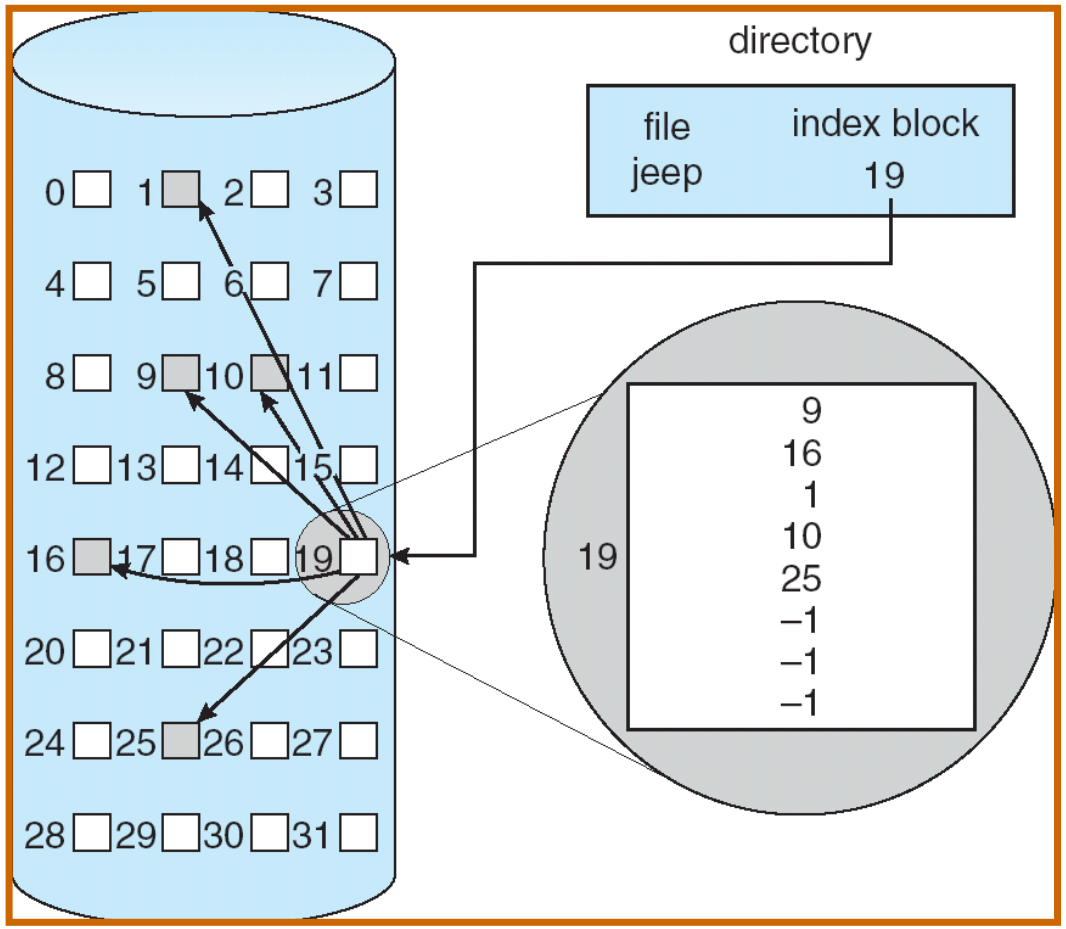
Indexed Allocation

- Solves the problems of linked allocation by bringing all the pointers (for a file's blocks) together into one location called the *index block*
- Each file has its own index block, which is an array of disk-block addresses
- Each entry in the index block points to the corresponding block of the file
- The directory contains the address of the index block
- Finding and reading a specific block in a file only requires the use of the pointer in the index block

Indexed Allocation

- When a file is created
  - The pointer to the index block is set to nil
  - When a new block is first written, it is obtained from the free-space management system and its address is put in the index block
- Supports direct access without suffering from external fragmentation
- Requires the additional space of an index block for each file
- Disadvantages
  - Suffers from some of the same performance problems as linked allocation
    - Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume

Example of Indexed Allocation



Operations Performed on Directory

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

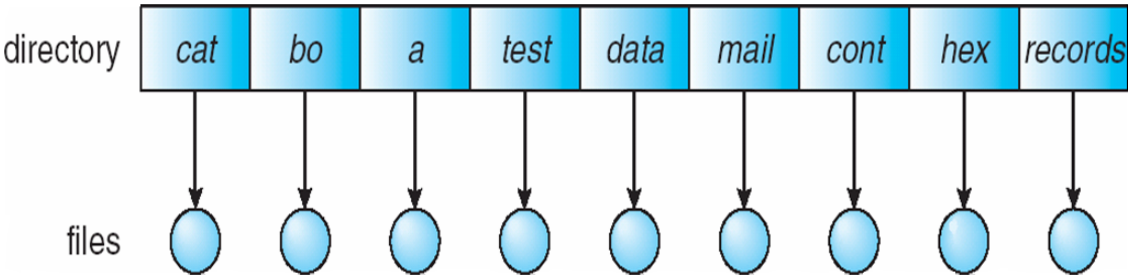
Directory Organization

The directory is organized logically to obtain

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

Single-Level Directory

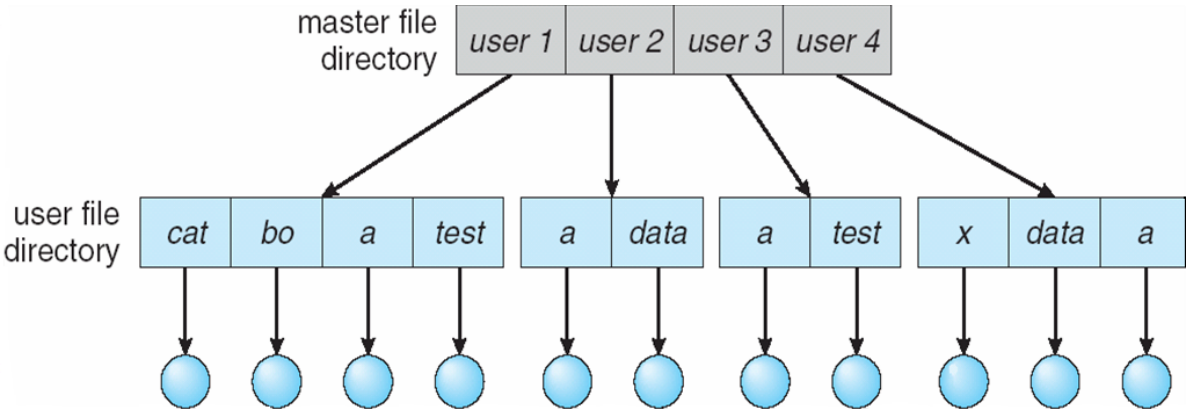
- A single directory for all users



- Naming problem
- Grouping problem

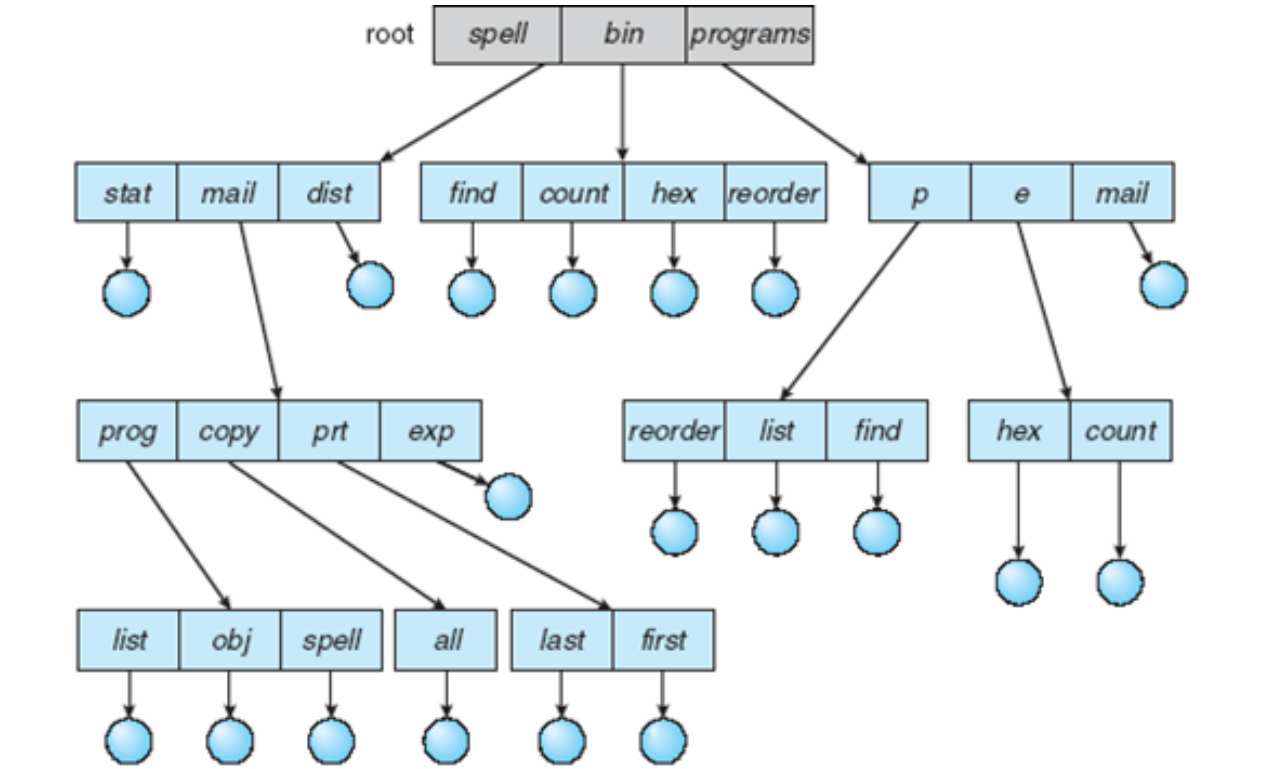
Two-Level Directory

- Separate directory for each user



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

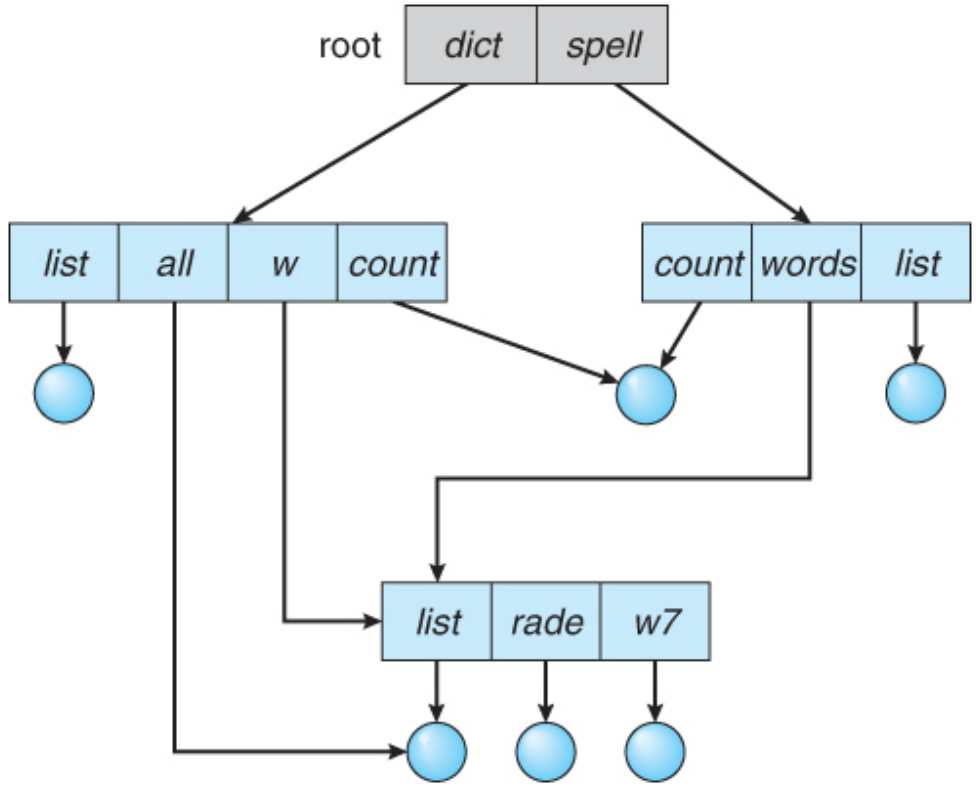
Tree-Structured Directories



- Path name
- Can have the same file name for different user
- Efficient searching
- Grouping capability

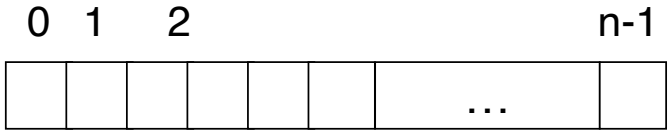
Acyclic Directory Structure

- Files can be shared by two or more users.
- UNIX provides two types of links for implementing the acyclic-graph structure.
  - Hard link
  - Symbolic link



Free-Space Management

Bit Vector Approach

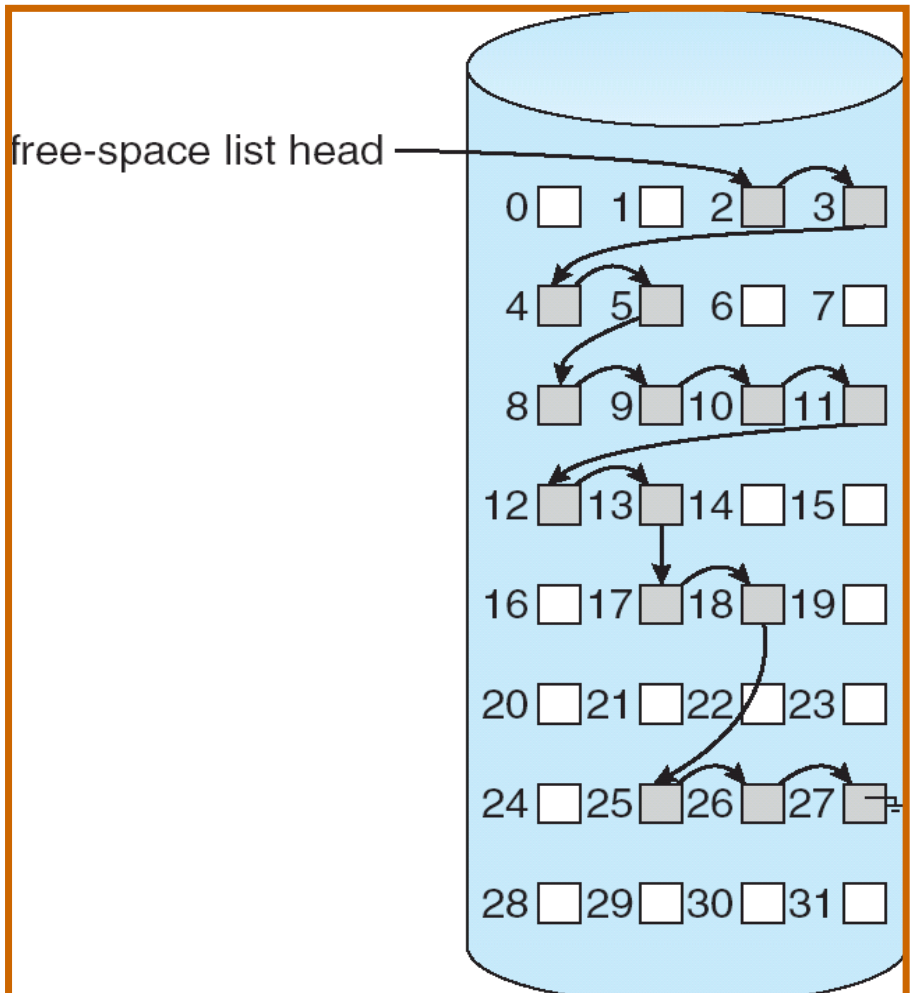


- Free-space blocks are tracked via a bit vector (where *n* = #blocks)

bit[*i*] = 0 ⇒ block[*i*] allocated

1 ⇒ block[*i*] free

Linked List Approach



Grouping

- Stores the addresses of n free blocks in the first free block.
- The first n-1 blocks are actually free; the last block contains the addresses of another n free blocks.
- The addresses of large number of free blocks can be found quickly using the approach.

Counting

- Storing several contiguous blocks.
- Storing the address of the first free block and the number of free contiguous blocks that follow the first block.
- Each entry in the free space list consists of a disk address and a count.

Free-Space Management Approaches

- Bit vector
  - Advantage: easy to get contiguous blocks for a file
  - Disadvantage: requires extra space to store the bit vector
- Linked list (free list)
  - Link together all the free disk blocks and keep a pointer to the first free block
  - Advantage: no waste of space
  - Disadvantage: cannot get contiguous blocks easily for a file
- Grouping (modification of free list approach)
  - Store the addresses of n free blocks in the first free block
  - The first $n - 1$ of these blocks are actually free
  - The last block contains the addresses of another n free blocks
- Counting
  - Take advantage of the fact that several contiguous blocks may be allocated or freed simultaneously
  - Keep the address of the first free block and the number n of free contiguous blocks that follow the first block

End of Chapter

DISK MANAGEMENT

Disk Structure

Disk provide bulk of secondary storage of computer system. The disk can be considered the one I/O device that is common to each and every computer. Disks come in many size and speeds, and information may be stored optically or magnetically. Magnetic tape was used as an early secondary storage medium, but the access time is much slower than for disks. For backup, tapes are currently used.

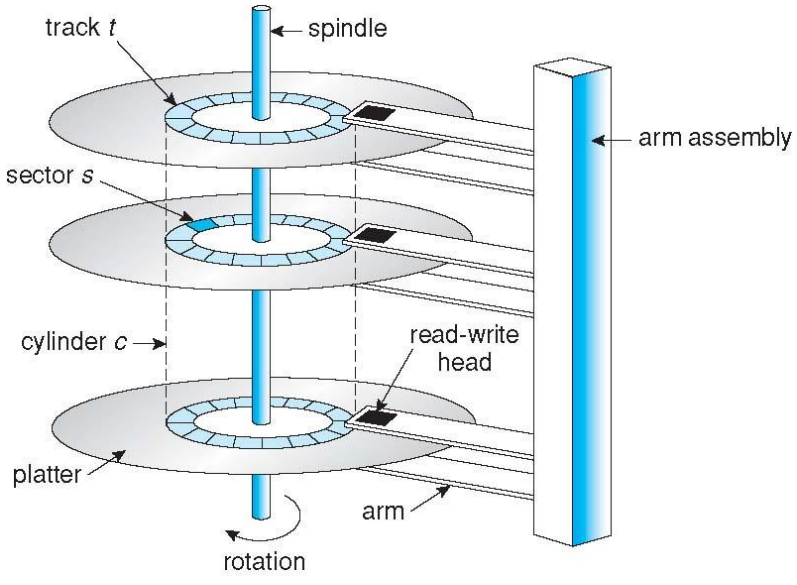
Modern disk drives are addressed as large one dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The actual details of disk I/O operation depends on the computer system, the operating system and the nature of the I/O channel and disk controller hardware.

The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The basic unit of information storage is a sector. The sectors are stored on a flat, circular, media disk. This media spins close to one or more read/write heads. The heads can move from the inner portion of the disk to the outer portion.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable head system or electronically selecting one head on a fixed head system. These characteristics are common to floppy disks, hard disks, CD-ROM and DVD.

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.



Disk Performance Parameters

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track.

Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time.**

When once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay,** or rotational latency. The sum of the seek time, if any, and the rotational delay equals the **access time,** which is the time it takes to get into position to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time.**

Seek Time Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components:

the initial startup time and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed.

$\mathbf{T_s = m \times n + s}$

Rotational Delay Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm up to, as of this writing, 15,000 rpm; at this latter speed, there is one revolution per 4 ms. Thus, on the average, the rotational delay will be 2 ms. Floppy disks typically rotate at between 300 and 600 rpm. Thus the average delay will be between 100 and 50 ms.

Transfer Time The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$\mathbf{T = b/rN}$

where

T = transfer time

b = number of bytes to be transferred

N = number of bytes on a track

r = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

$Ta = Ts + T$

where Ts is the average seek time.

Disk Scheduling

The amount of head needed to satisfy a series of I/O request can affect the performance. If desired disk drive and controller are available, the request can be serviced immediately. If a device or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. When one request is completed, the operating system chooses which pending request to service next. Any one of several disk-scheduling algorithms can be used.

Different types of scheduling algorithms are as follows.

1. First Come, First Served scheduling algorithm(FCFS).

2. Shortest Seek Time First (SSTF) algorithm

3. SCAN algorithm

4. Circular SCAN (C-SCAN) algorithm

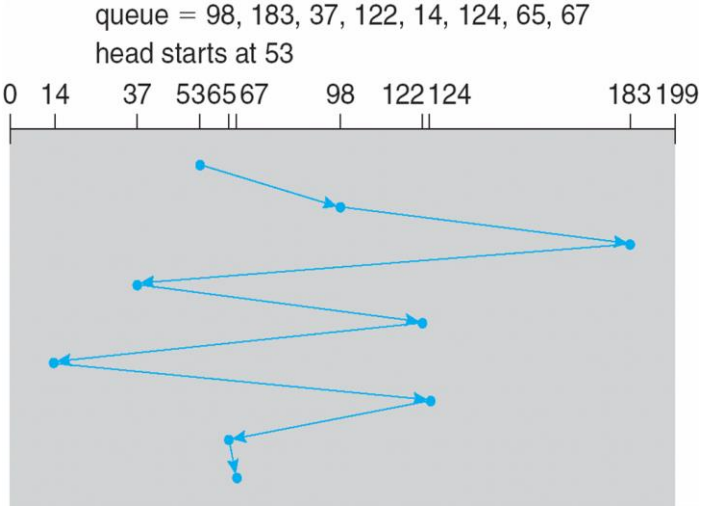
5. Look Scheduling Algorithm

First Come, First Served scheduling algorithm(FCFS).
The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.

Priority With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software.

Last In First Out ln transaction processing systems, giving the device to the most recent user should result. In little or no arm movement for moving through a sequential file. Taking advantage of this locality improves throughput and reduces queue length.

Example: Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67 in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders.

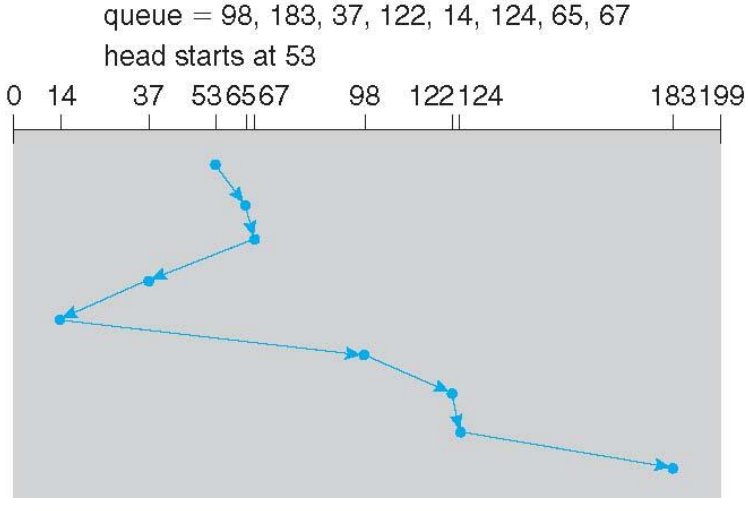


Shortest Seek Time First (SSTF) algorithm
The SSTF policy is to select the disk I/O request that requires the least movement of the disk arm from its current position. **Scan** With the exception of FIFO, all of the policies described

so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.

The choice should provide better performance than FCFS algorithm.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.



Under heavy load, SSTF can prevent distant request from ever being serviced. This phenomenon is known as starvation. SSTF scheduling is essentially a form of shortest job first scheduling. SSTF scheduling algorithm are not very popular because of two reasons.
1. Starvation possibly exists.

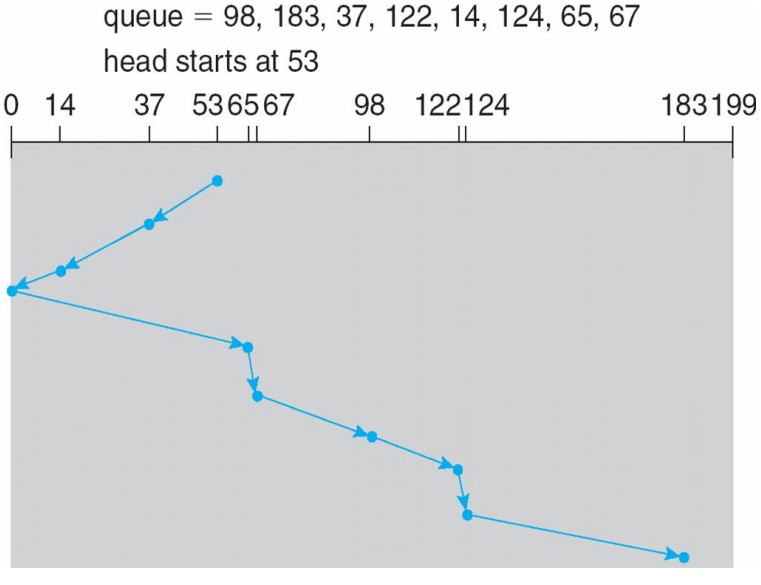
2. It increases higher overheads.

SCAN scheduling algorithm

In the **SCAN algorithm,** the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. Thus, the scan algorithm has the head start at track 0 and move towards the highest numbered track, servicing all requests for a track as it passes the track. At the other end, the direction of

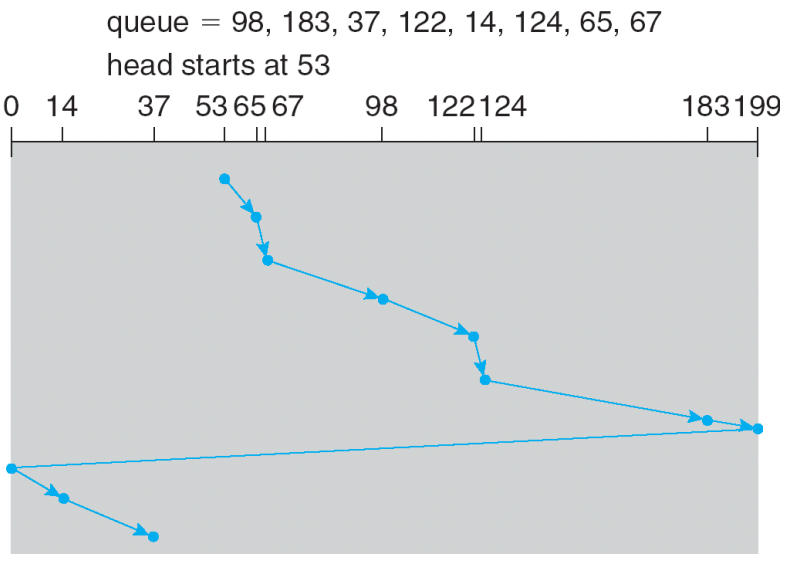
head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm,** since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way. SCAN algorithm is guaranteed to service every request in one complete pass through the disk. SCAN algorithm behaves almost identically with the SSTF algorithm.

Before applying SCAN to schedule the requests on cylinders 98,183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183. If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.



C SCAN Scheduling Algorithm

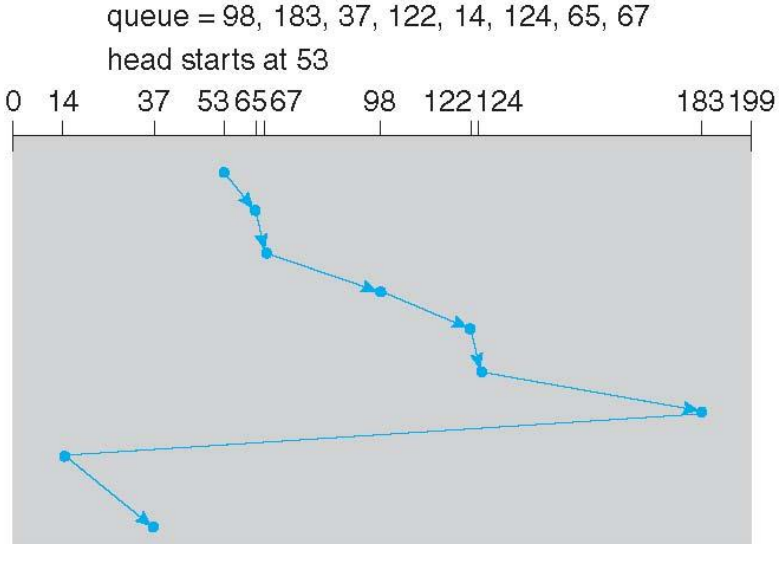
Circular SCAN (C-SCAN) **scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.



This reduces the maximum delay experienced by new requests.

LOOK Scheduling Algorithm

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling,** because they *look* for a request before continuing to move in a given direction.



Start the head moving in one direction. Satisfy the request for the closest track in that direction when there is no more request in the direction, the head is traveling, reverse direction and repeat. This algorithm is similar to innermost and outermost track on each circuit.