

OPERATING SYSTEMS

Chapter 1 Introduction

What is an Operating System?

An **operating system (OS)** is a program that manages a computer's hardware.

It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

As a manager, the operating system has two basic functions:

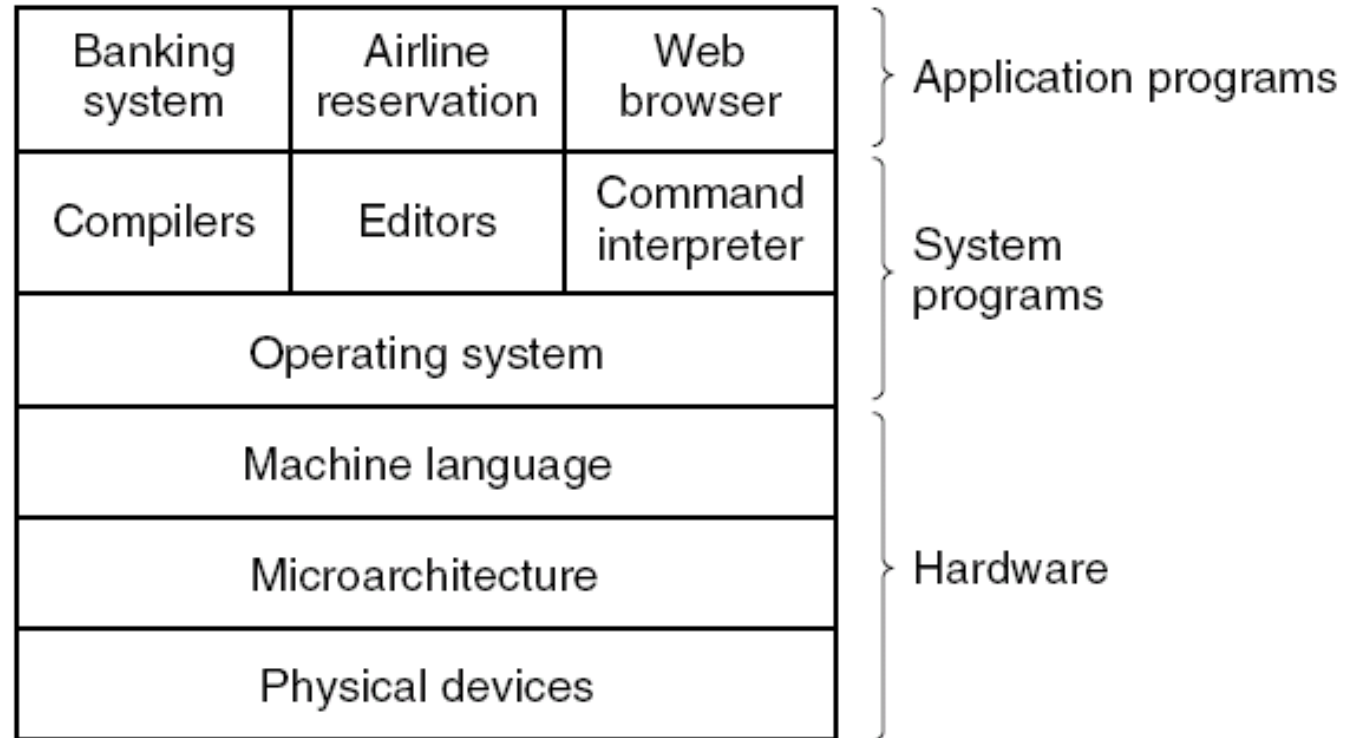
OS oversees all hardware resources and allocates them to user and applications as needed.

Performs many low-level tasks on behalf of users and application programs

Operating Systems Types

- Mainframe OS
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Real-Time OS
- Smart Card OS

The Modern Computer System

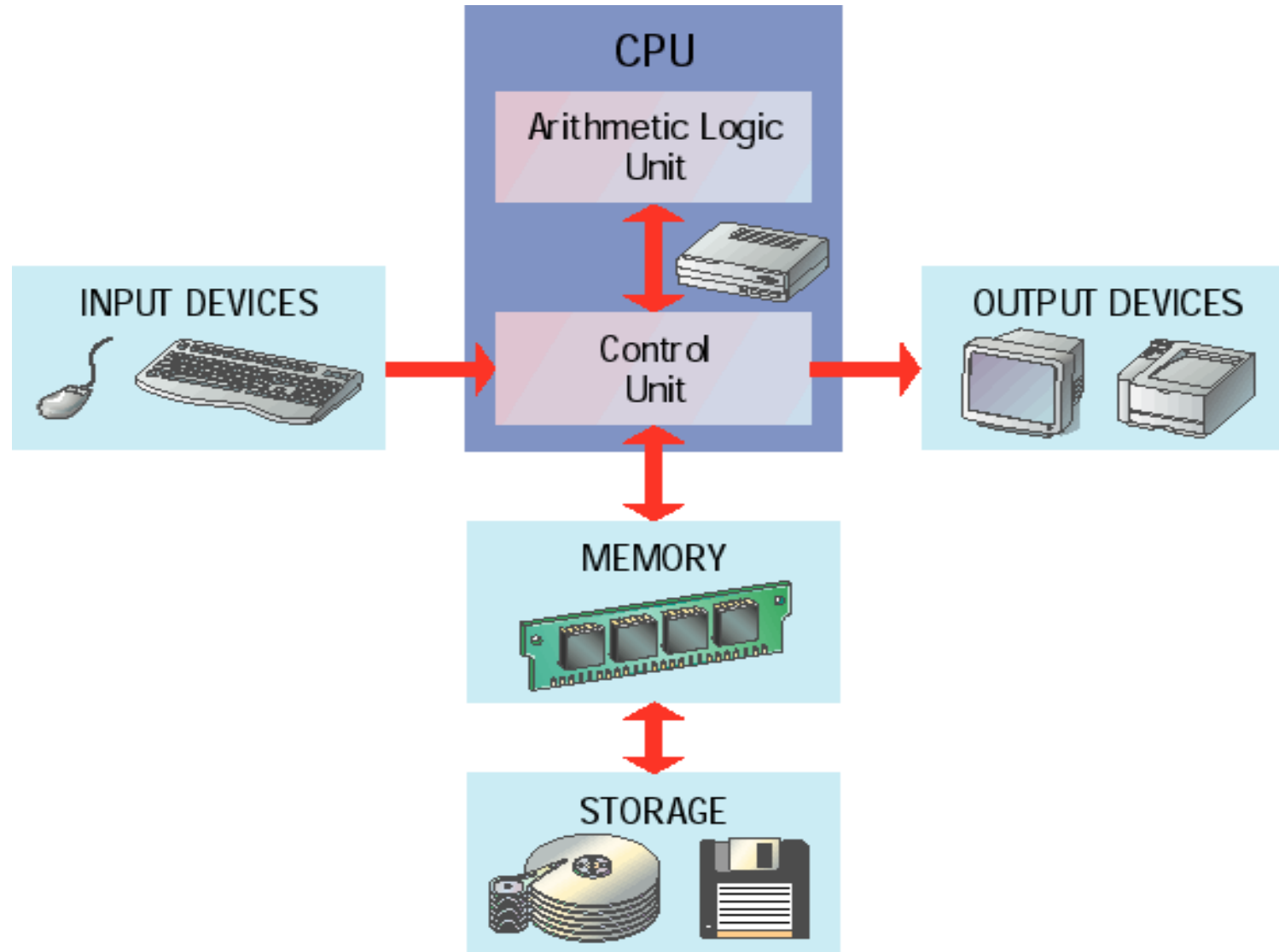


A computer system consists of hardware, system programs, and application programs.

Computer System Components

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

Computer Hardware



Computer Hardware

- I/O devices and the CPU can execute concurrently
- A number of device controllers connected through a common bus that provides access to shared memory
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- Device controller communicates with CPU by using an interrupt

Computer-System Operation

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run.

This initial program, or **bootstrap program**, is stored within the computer hardware in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term **firmware**.

It initializes all aspects of the system, from CPU registers to device controllers to memory contents.

Computer-System Operation

The bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users.

Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

Interrupts

- If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen.
- 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 that an operating-system service be performed.
- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.
- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
- Software may trigger an interrupt by executing a special operation called a system call

Interrupts

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location (**interrupt vector**) usually contains the starting address where the **service routine** for the interrupt is located.

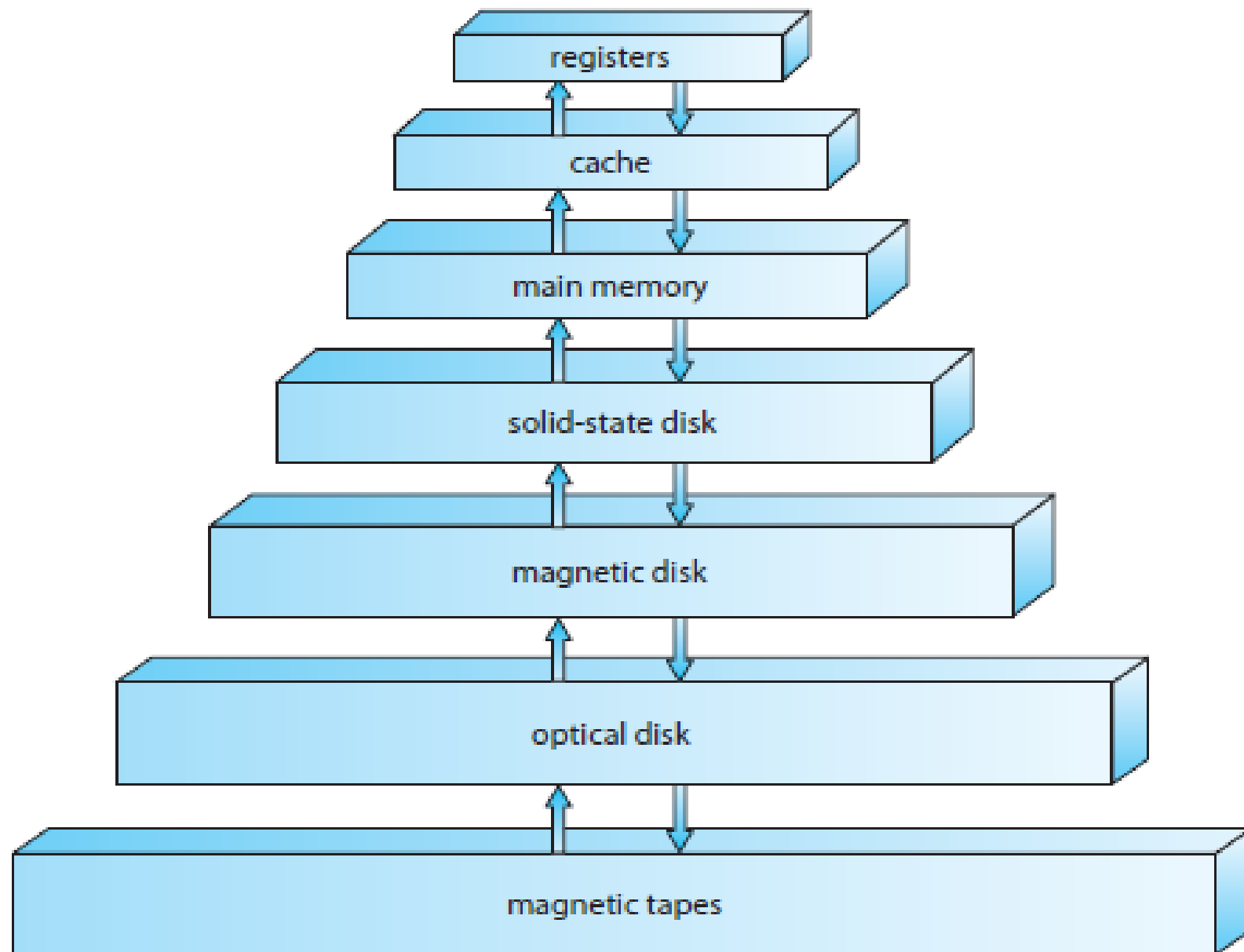
The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Interrupt architecture must save the address of the interrupted Instruction.

Thus, the operating system is interrupt driven!

Storage Structure

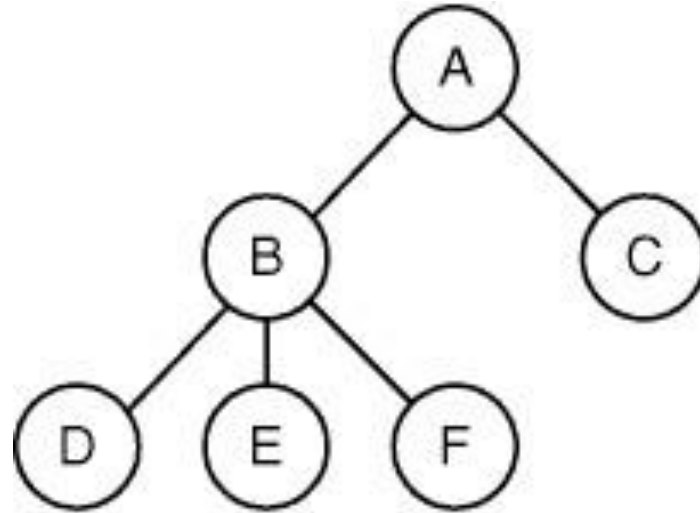
- **Main memory** – only large storage media that the CPU can access directly. It is **Random access** and **volatile**!
- **Secondary storage** – an extension of main memory that provides large **nonvolatile** storage capacity.
- The disk surface is logically divided into **tracks**, which are subdivided into **sectors**!
- The **disk controller** determines the logical interaction between the device and the computer



OS Concepts

Process is a running program, for each process there is **address space**, which is a list of memory locations. Also, associated with each process is a set of **resources**.

Processes



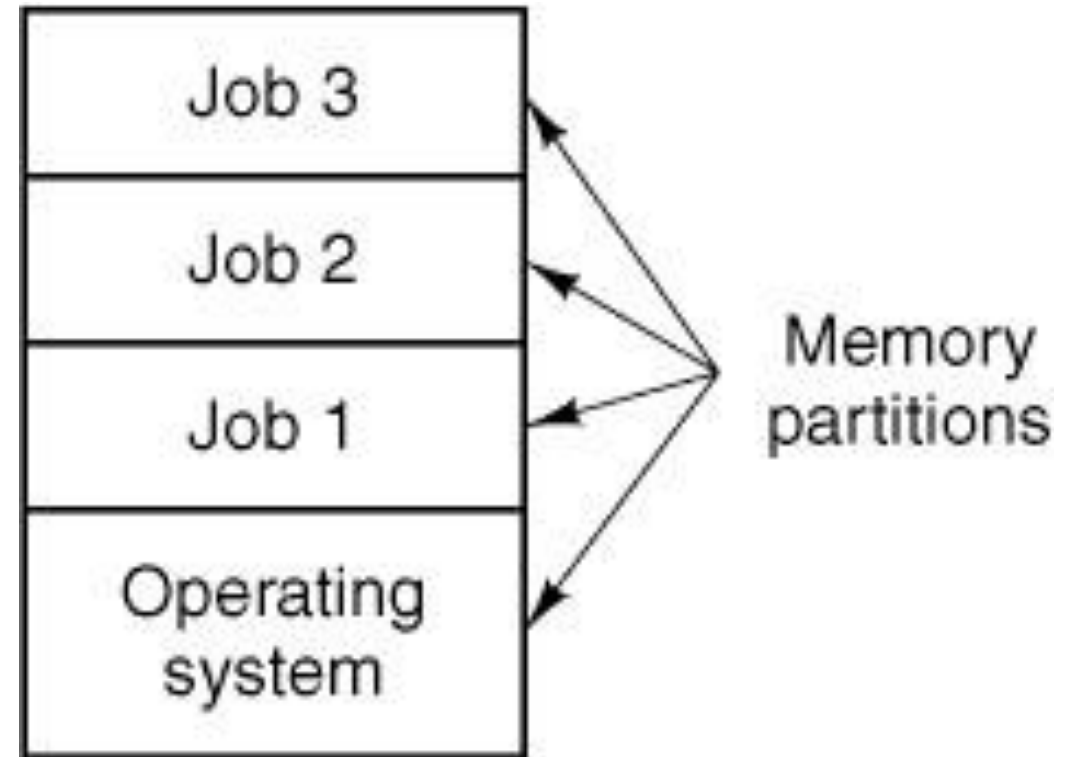
A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

Address Space

- Each computer has RAM to store executing programs.
- In a very simple OS, only one program at a time is in memory.
- More sophisticated OS allows multiple programs to be in RAM,

RAM for Multiprogramming

A multiprogramming system with three jobs in memory.



Operating System Structure

Multiprogramming is needed for efficiency, Single user cannot keep CPU and I/O devices busy at all times.

Multiprogramming organizes jobs (code and data) so the CPU always has one to execute.

One job selected and run via **job scheduling**!

When it has to wait (for I/O for example), OS switches to another job.

Timesharing (multitasking) is a logical extension in which the CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing.

Overview

Scheduling of this kind is a fundamental operating-system function.

All computer resources are scheduled before use.

The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU Scheduling

CPU Scheduling

When more than one process runs on the system the OS decides how and when a process will use the CPU. Hence, the name is also **CPU Scheduling**. The OS:

- Allocates and deallocates processor to the processes.
- Keeps record of CPU status.

Certain algorithms used for CPU scheduling are as follows:

- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Round-Robin Scheduling
- Priority-based scheduling.

Scheduling Algorithms

1) First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm**.

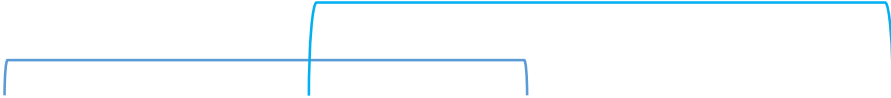
With this scheme, the process that requests the CPU first is allocated the CPU first.

The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

On the negative side, the average waiting time under the FCFS policy is often quite long.

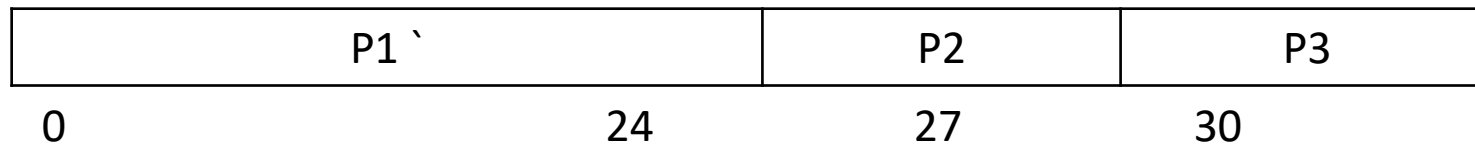
EX: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
<i>P1</i>	24
<i>P2</i>	3
<i>P3</i>	3



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	24	24	24	0
P2	0	3	27	27	24
P3	0	3	30	30	27

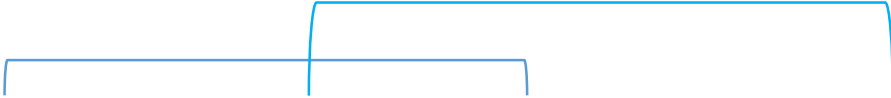
$$AV = 51/3 = 17$$



<P1, P2, P3>

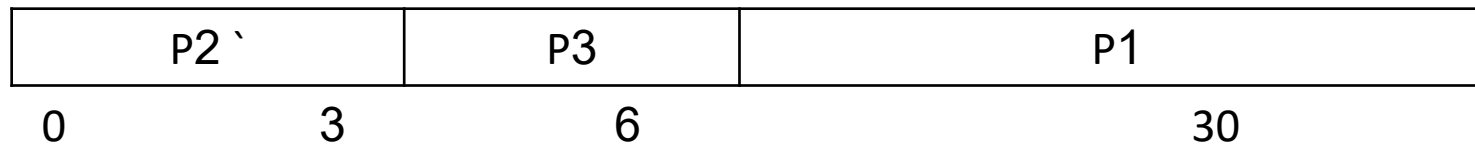
EX: Consider the following set of processes that arrive at 0 times, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
<i>P2</i>	3
<i>P3</i>	3
<i>P1</i>	24



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P2	0	3	3	3	0
P3	0	3	6	6	3
P1	0	24	30	30	6

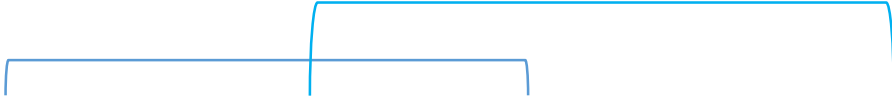
$$AV = 9/3 = 3$$



<P2, P3, P1>

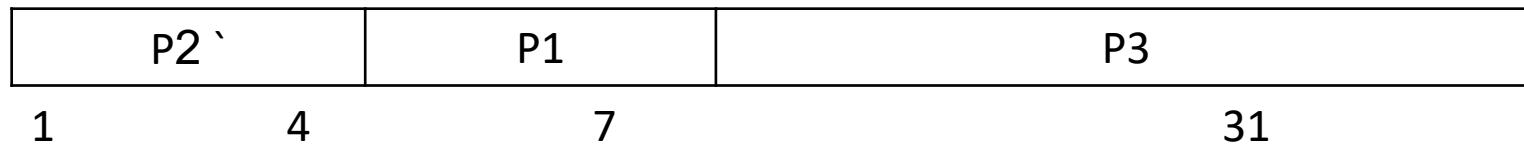
EX: Consider the following set of processes that arrive at different times, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Arrival time</u>	<u>Burst Time</u>
<i>P1</i>	<i>3</i>	<i>3</i>
<i>P2</i>	<i>1</i>	<i>3</i>
<i>P3</i>	<i>5</i>	<i>24</i>



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	3	3	7	4	1
P2	1	3	4	3	0
P3	5	24	31	26	2

$$AV = 3/3 = 1$$



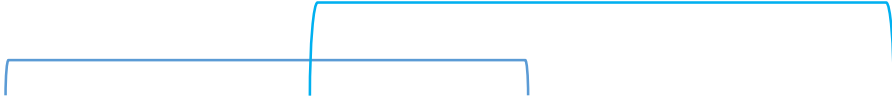
<P2, P1, P3>

2) Shortest-Job-First Scheduling

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

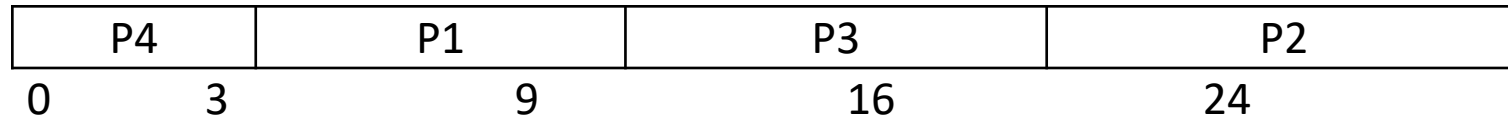
EX: Consider the following set of processes that arrive at 0 times, with the length of the CPU burst given in milliseconds:

Process	Arrival time	Burst Time
<i>P1</i>	<i>0</i>	<i>6</i>
<i>P2</i>	<i>0</i>	<i>8</i>
<i>P3</i>	<i>0</i>	<i>7</i>
<i>P4</i>	<i>0</i>	<i>3</i>



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	6	9	9	3
P2	0	8	24	24	16
P3	0	7	16	16	9
P4	0	3	3	3	0

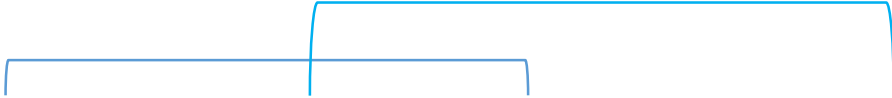
$$AV = 28/4 = 7$$



<P4, P1, P3, P2>

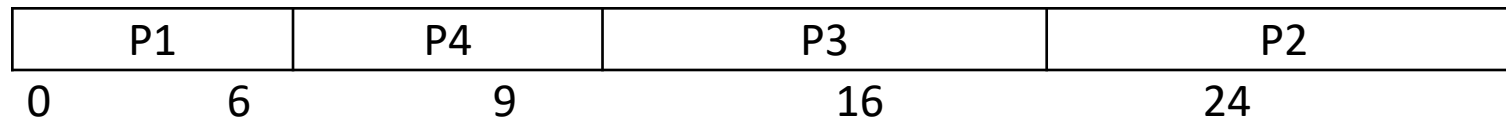
EX: Consider the following set of processes that arrive at different times, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Arrival time</u>	<u>Burst Time</u>
<i>P1</i>	<i>0</i>	<i>6</i>
<i>P2</i>	<i>5</i>	<i>8</i>
<i>P3</i>	<i>4</i>	<i>7</i>
<i>P4</i>	<i>2</i>	<i>3</i>



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	6	6	6	0
P2	5	8	24	19	11
P3	4	7	16	12	5
P4	2	3	9	7	4

$$AV = 20/4 = 5$$



<P1, P4, P3, P2>

The SJF scheduling algorithm is **provably optimal**, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The **real difficulty** with the SJF algorithm is knowing the length of the next CPU request. It cannot be implemented, there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling.

3) Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

Priority scheduling can be either **preemptive or non-preemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

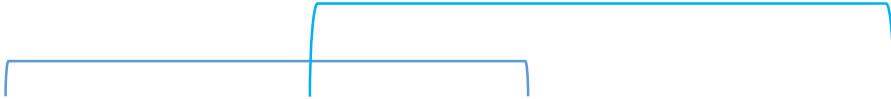
A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

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

Example

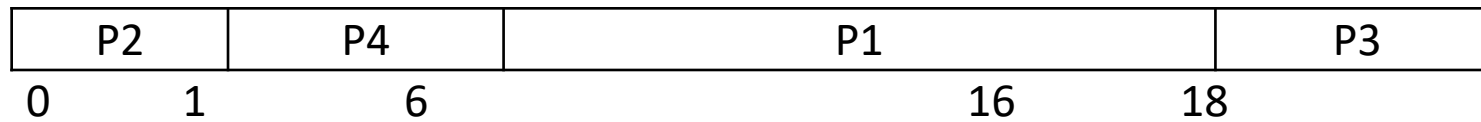
Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds: If the processes arrive in the order P1, . . . , P4, schedule these processes using priority scheduling:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P1	10	3
P2	1	1
P3	2	4
P4	5	2



Processes	Priority	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	3	0	10	16	16	6
P2	1	0	1	1	1	0
P3	4	0	2	18	18	16
P4	2	0	5	6	6	1

$$AV = 23/4 = 5.75$$

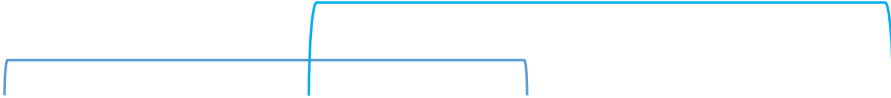


<P2, P4, P1, P3>

Example

Consider the following set of processes that arrive at different time, with the length of the CPU-burst time given in milliseconds: If the processes arrive in the order P1, . . . , P4, schedule these processes using priority scheduling:

<u>Process</u>	<u>Priority</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P1	7	2	10
P2	1	1	1
P3	11	4	2
P4	3	3	5



Process	Priority	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	7	2	10	12	10	0
P2	1	1	1	2	1	0
P3	11	4	2	19	15	13
P4	3	3	5	17	14	9

$$AV = 22/4 = 5.5$$

P2	P1		P4	P3
1	2	12	17	19

<P2, P1, P4, P3>

A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.

A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting for a long time. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blocking of low-priority processes is **aging**.

Aging involves gradually increasing the priority of processes that wait in the system for a long time.

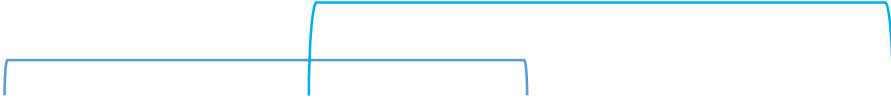
4) Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time called a **time quantum or time slice**, is defined.

Example

Consider the set of 3 processes that arrived at time 0 and burst time are given below-If the CPU scheduling policy is Round Robin with time quantum = 4 unit, calculate the average waiting time.

<u>Process</u>	<u>Burst Time</u>
P1	24
P2	3
P3	3



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	24	30	30	6
P2	0	3	7	7	4
P3	0	3	10	10	7

$$AV = 17/3 = 5.6$$

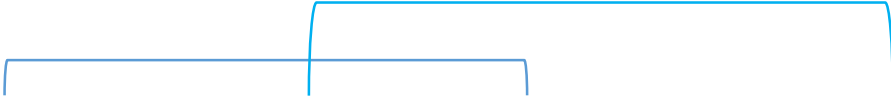
P1	P2	P3	P1	P1	P1	P1	P1	
0	4	7	10	14	18	22	26	30

<P1, P2, P3>

Example

Consider the set of 5 processes whose arrival time and burst time are given in table below. If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time.

Process	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	5/ 3 1 0	14	14	9
P2	1	3/ 1 0	12	11	8
P3	2	1/ 0	5	3	2
P4	3	2/ 0	7	4	2
P5	4	3/ 1 0	13	9	6

$$AV = 27/5 = 5.4$$


P1	P2	P3	P4	P5	P1	P2	P5	P1	
0	2	4	5	7	9	11	12	13	14

<P1, P2, P3, P3, P4, P5, P1, P2, P5, P1 >

Example

Consider the set of 5 processes whose arrival time and burst time are given in table below. If the CPU scheduling policy is Round Robin with time quantum = 4 unit, calculate the average waiting time.

Process	Arrival time	Burst time
P1	0	4
P2	5	5
P3	2	2
P4	3	1
P5	4	6



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	4/ 0	4	4	0
P2	5	5/ 1 0	18	13	6
P3	2	2/ 0	6	4	2
P4	3	1/ 0	7	4	3
P5	4	6/ 0	17	13	7

$$AV = \frac{18}{5} = 3.6$$

P1	P3	P4	P5	P2	P5	P2
0	4	6	7	11	15	17
18						

Preemptive Scheduling

CPU-scheduling decisions may take place under the following four conditions:

1. When a process switches from the **running** state to the **waiting** state (for example, as the result of an I/O request).
2. When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs)
3. When a process switches from the **waiting** state to the **ready** state (for example, at completion of I/O)
4. When a process terminates.

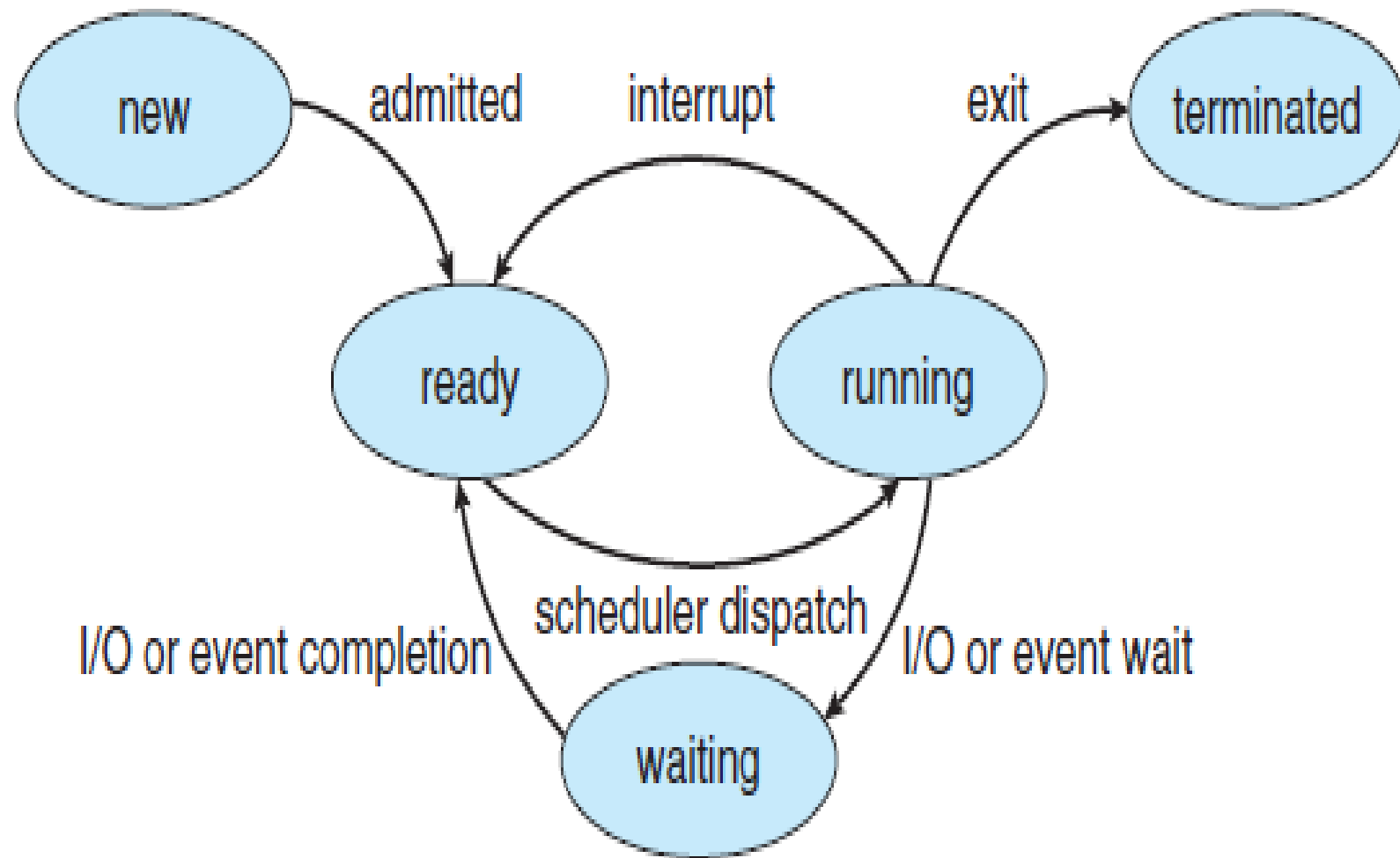


Figure 3.2 Diagram of process state.


When scheduling takes place only under conditions 1 and 4, we say that the scheduling scheme is **nonpreemptive**. Otherwise, it is **preemptive**.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Example

Consider the set of 5 processes whose arrival time and burst time are given below- If the CPU scheduling policy is SJF **preemptive**, calculate the average waiting time and average turn around time.

Process	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	3	1/ 0	4	1	0
P2	1	4/ 2 0	6	5	1
P3	4	2/ 0	8	4	2
P4	0	6/ 5 0	16	16	10
P5	2	3/ 0	11	9	6


$$AV = 19/5 = 3.8$$

P4	P2	P1	P2	P3	P5	P4	
0	1	3	4	6	8	11	16

Example

Consider the set of 6 processes whose arrival time and burst time are given below- If the CPU scheduling policy is **preemptive SJF**, calculate the average waiting time and average turn around time.

Process	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	7 6 0	19	19	12
P2	1	5 4 0	13	12	7
P3	2	3 2 0	6	4	1
P4	3	1 0	4	1	0
P5	4	2 0	9	5	3
P6	5	1 0	7	2	1

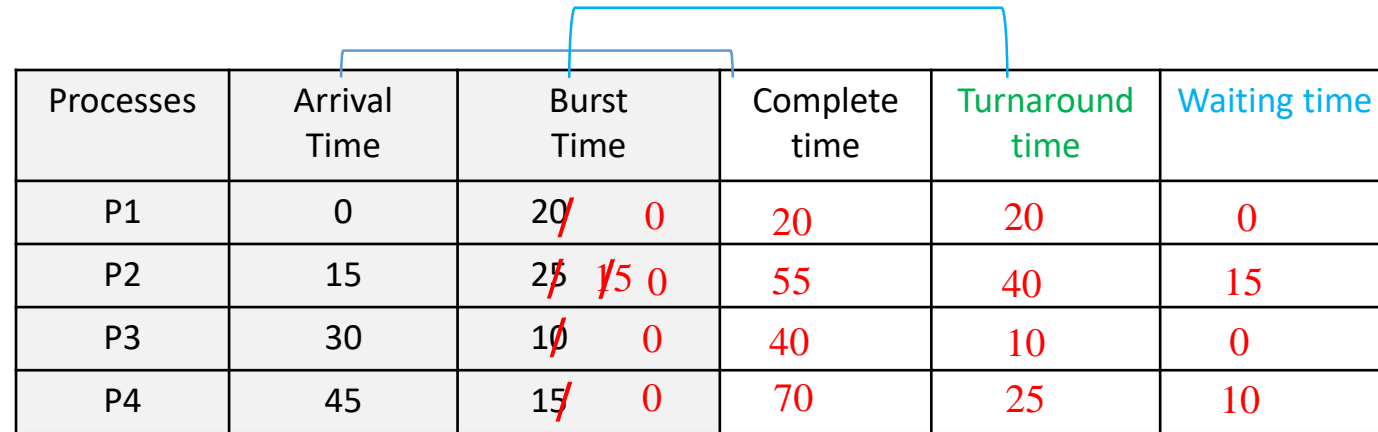
$$AV = 24/6 = 4$$

P1	P2	P3	P4	P3	P6	P5	P2	P1	
0	1	2	3	4	6	7	9	13	19

Example

Consider the following set of processes that arrive at different time with the length of the CPU burst given in milliseconds: Count the average waiting time under the **Preemptive SJF** policy

Process ID	Arrival time	Burst time
P1	0	20
P2	15	25
P3	30	10
P4	45	15



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	20 0	20	20	0
P2	15	25 15 0	55	40	15
P3	30	10 0	40	10	0
P4	45	15 0	70	25	10

$$AV = 25/4 = 6.2$$


P1	P2	P3	P2	P4	
0	20	30	40	55	70

Example

Consider the following set of processes that arrive at different time with the length of the CPU burst given in milliseconds: Count the average waiting time under the **Preemptive Priority** policy

.

Process	Arrival time	Burst time	
P1	0	<u>8</u>	
P2	1	2	
P3	2	4	
P4	3	1	
P5	4	6	
P6	6	5	
P7	10	1	



Processes	Priority	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	3	0	8 / 3 0	15	15	7
P2	4	1	2 / 0	17	16	14
P3	4	3	4 / 0	21	18	14
P4	5	4	1 / 0	22	18	17
P5	2	5	6 / 1 0	12	7	1
P6	6	6	5 / 0	27	21	16
P7	1	10	1 / 0	11	1	0

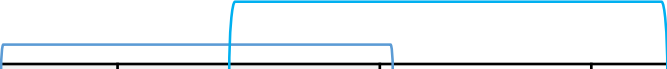
$$AV = 69/7 = 9.8$$

P1	P5	P7	P5	P1	P2	P3	P4	P6	
0	5	10	11	12	15	17	21	22	27

Example

Consider the following set of processes that arrive at different time with the length of the CPU burst given in milliseconds: Count the average waiting time under the **Preemptive SJF** policy

Process	Arrival time	Burst time
P1	0	9
P2	1	4
P3	2	9



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	0	9/ 8 0	13	13	4
P2	1	4/ 0	5	4	0
P3	2	9/ 0	22	20	11

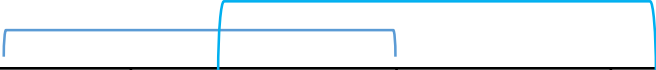
$AV = 15/3 = 5$

P1	P2	P1	P3	
0	1	5	13	22

Example

Consider the following set of processes that arrive at different times, with the length of the CPU burst given in milliseconds: Count the average waiting time using SJF scheduling.

Process ID	Arrival Time	Burst Time
P1	15	2
P2	13	8
P3	0	9
P4	1	2



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P0	15	2	23	8	6
P1	13	8	21	8	0
P2	0	9	9	9	0
P3	1	2	11	10	8

$$AV = 14/4 = 3.5$$


P2	P3		P1	P0	
0	9	11	13	21	23

Example (Interrupts)

Count the average waiting time (A.W.T) for executing the following processes using:

Round Robin (RR) scheduling when the time quantum is 5 and an interrupt is occurred at time 6 for 3 milliseconds.

Process ID	Arrival Time	Burst Time
P1	2	5
P2	3	13
P3	0	8
P4	5	4



Processes	Arrival Time	Burst Time	Complete time	Turnaround time	Waiting time
P1	2	5/ 4 0	13	11	6
P2	3	13 8/ 0	33	30	17
P3	0	8/ 13 0	25	25	17
P4	5	4/ 22 0	22	17	14

$$AV = 54/4 = 13.5$$

P3	P1		P1	P2	P4	P3	P2	
0	5	6	9	13	18	22	25	33

Enhancing performance of OS

Caching, Buffering, and Spooling

Differences between Caching, Buffering and Spooling:

Spooling :

- It's a process of placing data in a temporary working area for another program to process. E.g.: Print spooling and Mail spools etc.
- When there is a resource (like a printer) to be accessed by two or more processes(or devices), there spooling comes in handy to schedule the tasks. Data from each process is put on the spool (print queue) and processed in FIFO(first in first out) manner.
- After writing the data on the spool, the process can perform other tasks. And printing process operates separately.
- Without spooling, the process would be tied up until the printing is finished.

Buffering :

- Preloading data into a reserved area of memory (the buffer).
- It temporarily stores input or output data in an attempt to better match the speeds of two devices such as a fast CPU and a slow disk drive.
- The buffer may be used in between when moving data between two processes within a computer. Data is stored in a buffer as it is retrieved from one process or just before being sent to another process.

Caching :

- Caching transparently stores data in the component called Cache, so that future request for that data can be served faster.
- A special high-speed storage mechanism. It can be either a reserved section of main memory or an independent high-speed storage device.
- The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere.
- E.g: Memory Caching, Disk Caching, Web Caching(used in browser), Database Caching etc.

OS Operation and Functions

Process Management

- A program is an *inactive entity*, like the contents of a file stored on disk, whereas a process is an *active entity*.
- A process needs certain resources—including CPU time, memory, files, and I/O devices—to complete its task.

- The operating system is responsible for the following activities in connection with **process management**:

- 1.Scheduling processes and threads on the CPUs
- 2.Creating and deleting both user and system processes
- 3.Suspending and resuming processes
- 4.Providing mechanisms for process organization
- 5.Providing mechanisms for process communication

Memory Management

- The operating system is responsible for the following activities in connection with **memory management**:
 1. Keeping track of which parts of memory are currently being used and who is using them
 2. Deciding which processes (or parts of processes) and data to move into and out of memory
 3. Allocating and deallocating memory space as needed

File-System Management

The operating system is responsible for the following activities in connection with **file management**:

1. Creating and deleting files
2. Creating and deleting directories to organize files
3. Supporting primitives for manipulating files and directories
4. Mapping files onto secondary storage
5. Backing up files on stable (nonvolatile) storage media

Mass-Storage Management

The operating system is responsible for the following activities in connection with **disk management**:

1. Free-space management
2. Storage allocation
3. Disk scheduling

I/O Systems

The OS I/O subsystem consists of **several components**:

- 1.A memory-management component that includes buffering, caching, and spooling
- 2.A general device-driver interface
- 3.Drivers for specific hardware devices

Protection and Security

If a computer system has multiple users and allows the parallel execution of multiple processes, then access to data must be regulated (controlled).

For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

- **Protection** is any mechanism for controlling the access of processes or users to the resources.
- A protection- oriented system provides a means to distinguish between authorized and unauthorized usage.
- **Security** is used to defend a system from external and internal attacks.
- Such attacks spread across a huge range and include viruses and worms, denial-of service attacks.

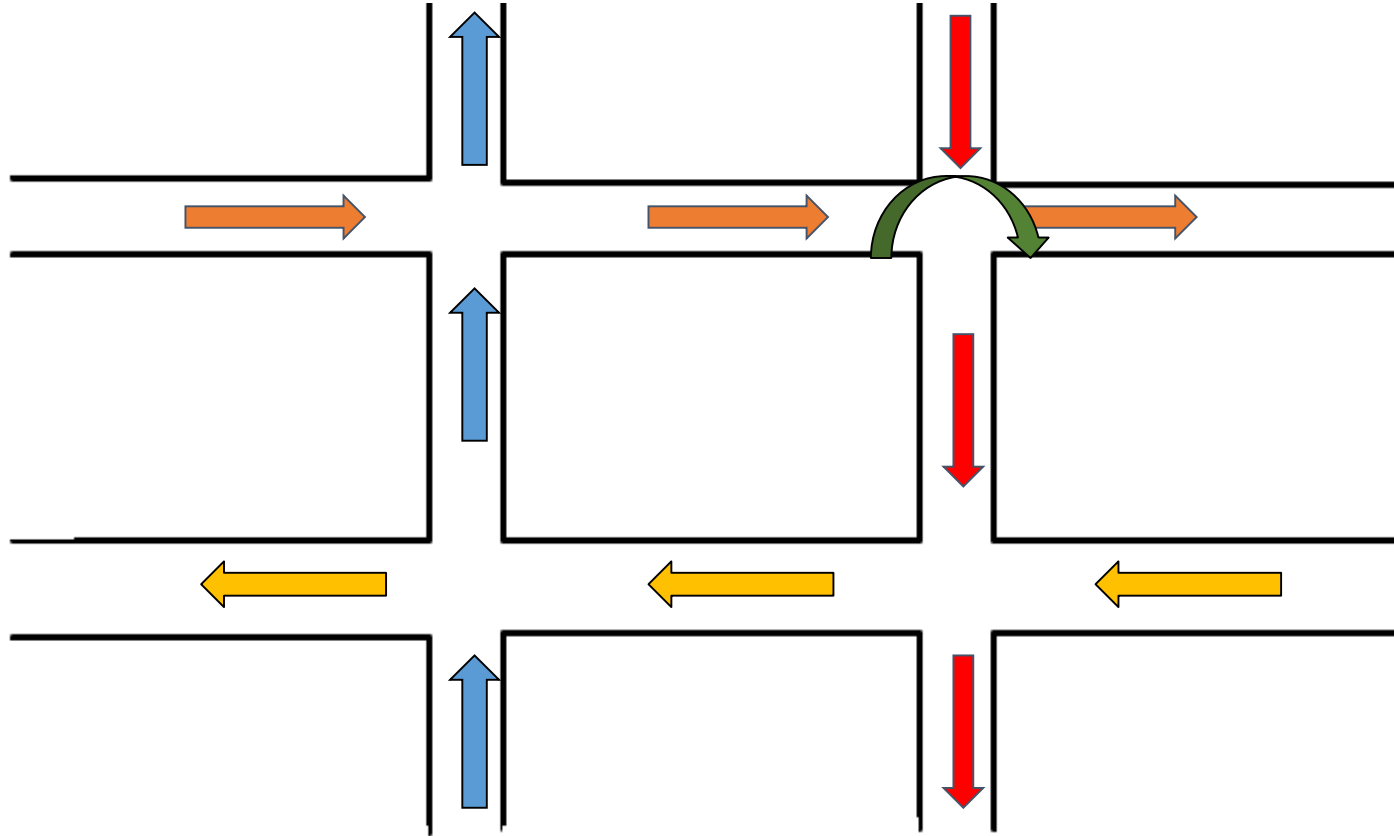
Deadlocks

DEADLOCK

Deadlock is a situation in OS where a set of processes are blocked because of multi processes compete for a limit number of resources.

A process request the resources, the resources are not available at that time, so the process enter into the waiting state. The requesting resources are held by another waiting process, both are in waiting state, this situation is called a **deadlock**.

4. Circular wait all the process are waiting



A deadlock situation can arise if the following four **conditions** hold at the same time in a system:

- 1. Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, **only one process at a time can use the resource**. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- 2. Hold and wait.** **A process must be holding at least one resource and waiting to get additional resources** that are currently being held by other processes.
- 3. No preemption.** **Resources cannot be preempted**; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- 4. Circular wait.** **all the processes are waiting**

A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_0 .

Resource-Allocation Graph

Resource allocation Graph (RAG) used for discovering system's status, is there any deadlock or not.

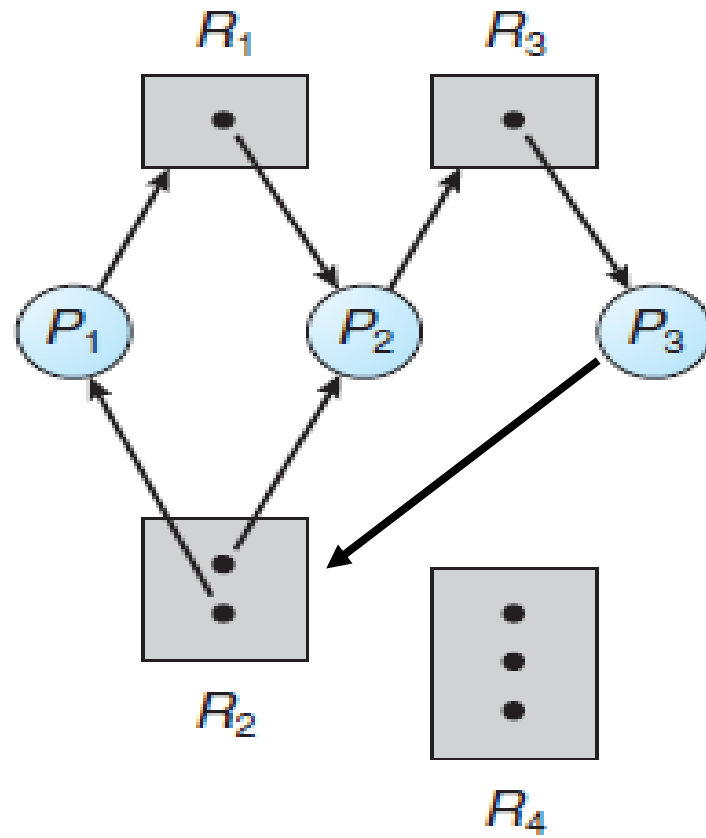
RAG components:

➤ (P) Process 

➤ (R) Resource 

➤ Instance (Copies)  

➤ (E) Request  and Allocate arrows 



◦ $R = \{ R_1, R_2, R_3, R_4 \}$

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

Q1. How many Resources?

Ans. R1,R2,R3,R4

Q.2 How many processes?

Ans. P1,P2,P3

Q.3 How many instances?

Ans. 7 instances

Q4. Which instances are free? And which are allocated?

Ans. all instances of R1,R2,R3 are allocated, instances of R4 are free

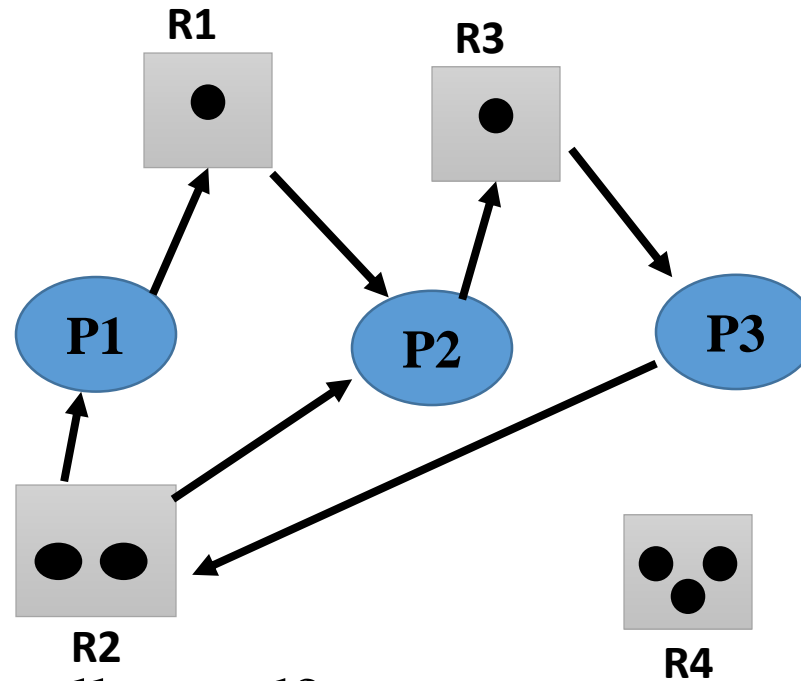
Q5.Is there any cycle?

Cycle=<P1→R1→P2→R3→P3→R2→P1>

(Start and stop at the same point by following one direction)

Q6.Is there any Deadlock?

Ans. YES



H.W

Q1. How many Resources?

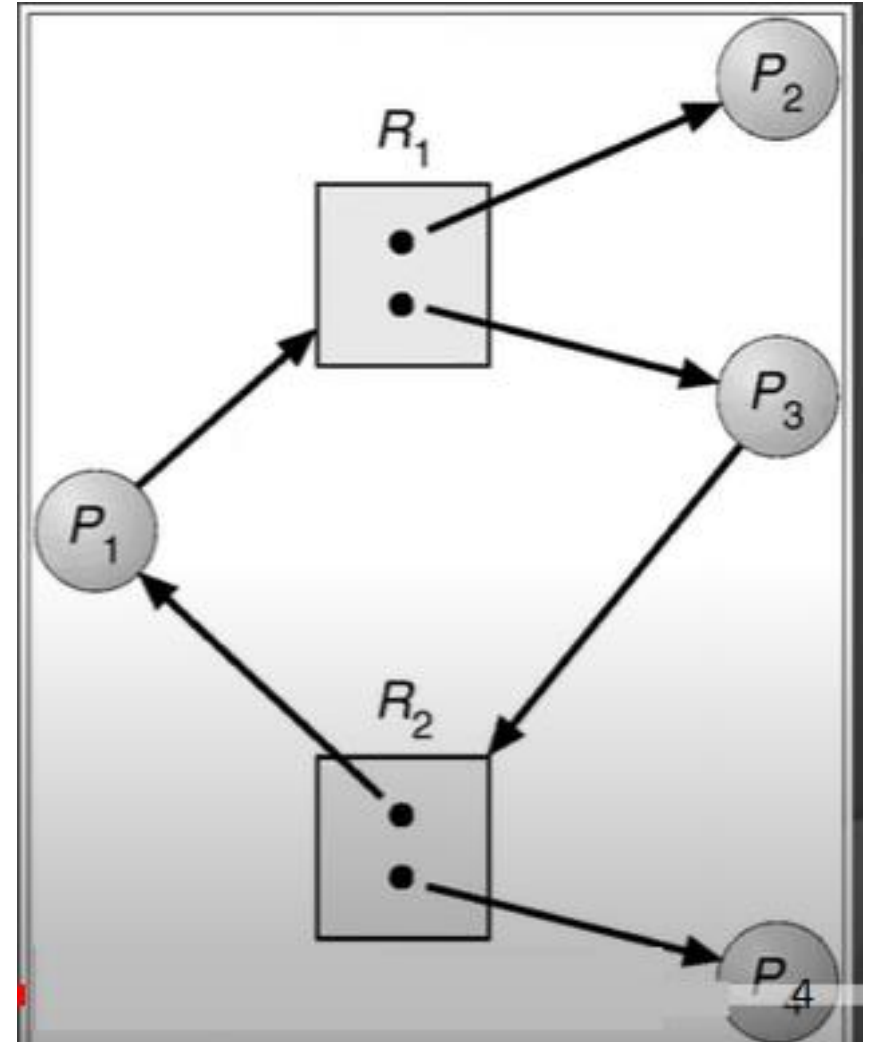
Q.2 How many processes?

Q.3 How many instances?

Q4. Which instances are free? And which are allocated?

Q5. Is there any cycle?

Q6. Is there any Deadlock?



- Answer the followings:

For a system with the following set of processes and resources information:

$P = \{P1, P2, P3\}$ $R = \{A, B, C, D\}$

$E = \{P1 \rightarrow A, A \rightarrow P2, P2 \rightarrow C, C \rightarrow P1\}$

Resource instances:

2 instance of resource type A

2 instance of resource type B.

2 instance of resource type C.

2 instance of resource type D.

Draw the resource allocation graph for the system to show the state of the system safe or unsafe

Methods for Handling Deadlocks

1. Ensure that the system will never enter a deadlock state:
 - Prevent deadlocks
 - Avoid deadlocks
2. We can allow the system to enter a deadlocked state, detect it, and recover (kill the process).
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent the occurrence of a deadlock*.

Mutual Exclusion: The mutual exclusion condition must hold. That is, at least one resource must be non sharable. **Sharable resources, do not require mutually exclusive** access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.

Hold and Wait: To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

No Preemption: The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.

If a process is holding some resources and requests another resource It must release the holding resources . The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait: The another condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds **is to arrange a total ordering of all resource types and to require that each process requests resources in an increasing order of counting.**

Deadlock Avoidance (Banker's Algorithm)

Data Structures for the Banker's Algorithm

There are four types of data structures used to implement 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: $n \times m$ matrix. If Max $[i, j] = k$, then process P_i may request at most k instances of resource type R_j
- Allocation: $n \times m$ matrix. If Allocation $[i, j] = k$ then P_i is currently allocated k instances of R_j
- Need $n \times m$ matrix If Need $[i, j] = k$, then P_i may need k more instances of R to complete its task
$$Need [i, j] = Max[i, j] - Allocation [i, j]$$
- **Banker's algorithm comprises of two algorithms:**
 - Safety algorithm
 - Resource request algorithm

Example of Banker's Algorithm

- 5 processes P₀ through P₄
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T₀:

Need = Max - Allocation

ABC	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	5	3	3	3	2	7	4	3
P ₁	2	0	0	3	2	2				1	2	2
P ₂	3	0	2	9	0	2				6	0	0
P ₃	2	1	1	2	2	2				0	1	1
P ₄	0	0	2	4	3	3				4	3	1

Example of Banker's Algorithm

Check whether the system works in the safe state or not?

	<u>Need</u>			<u>Available</u>			<u>Allocation</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P0	7	4	3	3	3	2	0	1	0
				+ 2	0	0	2 0 0		
P1	1	2	2	5	3	2			
P2	6	0	0	+ 2	1	1	3 0 2		
				7	4	3			
P3	0	1	1	+ 0	0	2	2 1 1		
				7	4	5			
P4	4	3	1	+ 0	1	0	0 0 2		
				7	5	5			
				+ 3	0	2			
				10	5	7			

The system is in a **safe state with** the sequence (*P1*, *P3*, *P4*, *P0*, *P2*)

H.W

Consider the following snapshot of a system:

	<u>Allocation</u>				<u>Max</u>				<u>Available</u>				<u>Need</u>			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	1	3	3	2	1				
P1	3	1	2	1	5	2	5	2								
P2	2	1	0	3	2	3	1	6								
P3	1	3	1	2	1	4	2	4								
P4	1	4	3	2	3	6	6	5								

Use the banker's algorithm to answer the following questions:

1. What is the content of the matrix ***Need***?
2. Is the system in a safe state?

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

	<u>Allocation</u>			
	A	B	C	D
P0	2	0	0	1
P1	3	1	2	1
P2	2	1	0	3
P3	1	3	1	2
P4	1	4	3	2

	<u>Available</u>			
	A	B	C	D
	3	3	2	1
	<hr/>			
	5	3	2	2
	<hr/>			
	6	6	3	4
	+ 1	4	3	2
	<hr/>			
	7	10	6	6
	+ 3	1	2	1
	<hr/>			
	10	11	8	7
	+ 2	1	0	3
	<hr/>			
	12	12	8	10

	<u>Need</u>			
	A	B	C	D
	2	2	1	0
	2	1	3	1
	0	2	1	3
	0	1	1	2
	2	2	3	3

The system is in a **safe state with** the sequence (*P0, P3, P4, P1, P2*)

Safety Algorithm

- 1. Let **Work** and **Finish** be vectors of length m and n , respectively.

Initialize:

Work = **Available**

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

- 2. Find an i such that both:

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

(b) **Need** i < **Work**

If no such i exists, go to step 4

- 3. **Work** = **Work** + **Allocation**,

Finish[i] = **true**

go to step 2

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

Resource-Request Algorithm for Process P_i

$\mathbf{Request}_i$ = request vector for process P_i , If $\mathbf{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j ,

1. If $\mathbf{Request}_i \leq \mathbf{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\mathbf{Request}_i \leq \mathbf{Available}$, 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:

$$\mathbf{Available} = \mathbf{Available} - \mathbf{Request}_i;$$

$$\mathbf{Allocation}_i = \mathbf{Allocation}_i + \mathbf{Request}_i;$$

$$\mathbf{Need}_i = \mathbf{Need}_i - \mathbf{Request}_i;$$

If safe \rightarrow the resources are allocated to P_i

If unsafe $\rightarrow P_i$, must wait, and the old resource-allocation state is restored

Example Resource-Request of Banker's Algorithm

5 processes P0 through P4

3 resource types: A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T0:

Use the banker's algorithm to answer the following questions:

1. What is the content of the matrix Need?
2. Is the system in a safe state?
3. Can request for (1,0,2) by P1 granted, Is the system in a safe state?

Answer

1. What is the content of the matrix Need?

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

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
ABC	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Example of Banker's Algorithm

2. Check the system works with the safe state or not?

<u>Process</u>	<u>Need</u>			<u>Available</u>			<u>Allocation</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P0	7	4	3	3	3	2	0	1	0
P1	1	2	2	+ 2	0	0	2	0	0
P2	6	0	0	5	3	2	3	0	2
P3	0	1	1	+ 2	1	1	2	1	1
P4	4	3	1	7	4	3	0	0	2
				+ 0	0	2			
				7	4	5			
				+ 0	1	0			
				7	5	5			
				+ 3	0	2			
				10	5	7			

The system is in a **safe state with** the sequence (*P1*, *P3*, *P4*, *P0*, *P2*)

3. Can request for (1,0,2) by P1 granted ?

$Request \leq Need$, $(1,0,2) \leq (1,2,2)$, which is true

$Request \leq Available$, $(1,0,2) \leq (3,3,2)$, which is true.

$Available = Available - Request_i$

$$3 \ 3 \ 2 - 1 \ 0 \ 2 = 2 \ 3 \ 0$$

$Allocation_i = Allocation_i + Request_i$

$$2 \ 0 \ 0 + 1 \ 0 \ 2 = 3 \ 0 \ 2$$

$Need_i = Need_i - Request_i$

$$1 \ 2 \ 2 - 1 \ 0 \ 2 = 0 \ 2 \ 0$$

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P0	0	1	0	7	5	3	<u>2</u>	<u>3</u>	<u>0</u>	7	4	3
P1	<u>3</u>	<u>0</u>	<u>2</u>	3	2	2				<u>0</u>	<u>2</u>	<u>0</u>
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Check the system works with the safe state or not?

<u>Process</u>	<u>Need</u>			<u>Available</u>			<u>Allocation</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P0	7	4	3	2	3	0	0	1	0
P1	0	2	0	+ 3	0	2	3	0	2
P2	6	0	0	5	3	2	3	0	2
P3	0	1	1	+ 2	1	1	2	1	1
P4	4	3	1	7	4	3	0	0	2
				+ 0	0	2			
				7	4	5			
				+ 0	1	0			
				7	5	5			
				+ 3	0	2			
				10	5	7			

The system is in a **safe state with** the sequence (*P1, P3, P4, P0, P2*)

Example 2- Resource-Request of Banker's Algorithm

Consider the following snapshot of a system:

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Use the banker's algorithm to answer the following questions:

1. What is the content of the matrix Need?
2. Is the system in a safe state?
3. Can request for (4,3,2) by P1 granted ?
4. Can request for (3,3,0) by P4 granted ?
5. Can request for (0,2,0) by P0 granted ?

1. What is the content of the matrix **Need**?

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

2. Is the system in a safe state?

<u>Process</u>	<u>Need</u>	<u>Available</u>	<u>Allocation</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P0	7 4 3	3 3 2	0 1 0
P1	1 2 2	+ 2 0 0	2 0 0
P2	6 0 0	5 3 2	3 0 2
P3	0 1 1	+ 2 1 1	2 1 1
P4	4 3 1	7 4 3	0 0 2
		+ 0 0 2	
		7 4 5	
		+ 0 1 0	
		7 5 5	
		+ 3 0 2	
		10 5 7	

The system is in a **safe state with** the sequence (*P1, P3, P4, P0, P2*)

3. Can request for (4,3,2) by P1 granted

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

$$\text{Request} \leq \text{Need}$$

$$4 \ 3 \ 2 \leq 1 \ 2 \ 2$$

$$\text{Request} \leq \text{Available}$$

$$4 \ 3 \ 2 \leq 3 \ 3 \ 2 \text{ the system must wait, since } \text{Available} \leq \text{Request}$$

4. Can request for (3,3,0) by P4 granted

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Request \leq Need

$$3 \ 3 \ 0 \leq 4 \ 3 \ 1$$

Request \leq Available

$$3 \ 3 \ 0 \leq 3 \ 3 \ 2$$

Available = Available - Request

$$3 \ 3 \ 2 - 3 \ 3 \ 0 = 0 \ 0 \ 2$$

Allocation = Allocation + Request

$$0 \ 0 \ 2 + 3 \ 3 \ 0 = 3 \ 3 \ 2$$

Need = Need - Request

$$4 \ 3 \ 1 - 3 \ 3 \ 0 = 1 \ 0 \ 1$$

Is the system in a safe state?

<u>Process</u>	<u>Need</u>			<u>Available</u>			<u>Allocation</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P0	7	4	3	0	0	2	0	1	0
P1	1	2	2	+2	0	0	2	0	0
P2	6	0	0	5	3	2	3	0	2
P3	0	1	1	+2	1	1	2	1	1
P4	4	3	1	7	4	3	0	0	2
				+0	0	2			
				7	4	5			
				+0	1	0			
				7	5	5			
				+3	0	2			
				10	5	7			

The system is in a **safe state with** the sequence (*P1*, *P3*, *P4*, *P0*, *P2*)

5. Can request for (0,2,0) by P0 granted ?

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Request \leq Need

$$0 \ 2 \ 0 \leq 7 \ 4 \ 3$$

Request \leq Available

$$0 \ 2 \ 0 \leq 3 \ 3 \ 2$$

Available = Available - Request

$$3 \ 3 \ 2 - 0 \ 2 \ 0 = 3 \ 1 \ 2$$

Allocation = Allocation + Request

$$0 \ 1 \ 0 + 0 \ 2 \ 0 = 0 \ 3 \ 0$$

Need = Need - Request

$$7 \ 4 \ 3 - 0 \ 2 \ 0 = 7 \ 2 \ 3$$

Is the system in a safe state?

<u>Process</u>	<u>Need</u>	<u>Available</u>	<u>Allocation</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P0	7 2 3	3 1 2	0 3 0
P1	1 2 2	+ 2 1 1	2 0 0
P2	6 0 0	5 2 3	3 0 2
P3	0 1 1	+ 2 0 0	2 1 1
P4	4 3 1	7 2 3	0 0 2
	+ 3 0 2	10 2 5	
	+ 0 0 2	10 2 7	
	+ 0 3 0	10 5 7	

The system is in a **safe state with** the sequence (*P3, P1, P2, P4, P0*)

Consider the following snapshot of a system: Use the banker's algorithm to answer the following questions:

1. What is the content of the matrix Need?
2. Is the system in a safe state?
3. Can request for (3,3,2) by P4 granted ?
4. Can request for (0,2,0) by P0 granted ?

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P0	1	3	0	6	3	0	0	1	2
P1	2	1	1	4	3	1			
P2	5	1	2	7	1	2			
P3	2	2	1	3	2	1			
P4	0	2	2	1	3	3			

Answer

1. What is the content of the matrix Need?

<u>Process</u>	<u>Allocation</u>			<u>Available</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C
P0	1	3	0	0	1	2	5	0	0
P1	2	1	1				2	2	0
P2	5	1	2				2	0	0
P3	2	2	1				1	0	0
P4	0	2	2				1	1	1

2. The system NOT in a safe state

Use the banker's algorithm to answer the following questions:

1. What is the content of the matrix *Need*?
2. Is the system in a safe state?
3. Can request for (3,2,1) by *P2* be granted?

Process	Max			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P1	8	2	3	3	1	0	2	1	2
P2	4	8	5	3	8	4			
P3	1	7	1	0	6	0			
P4	7	0	4	0	0	4			

	<u>Need</u>	<u>Available</u>	<u>Allocation</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P1	5 1 3	2 1 2	3 1 0
P2	1 0 1	+ 3 8 4	3 8 4
P3	1 1 1	5 9 6	0 6 0
P4	7 0 0	+ 0 6 0	0 0 4
		5 15 6	
		+ 3 1 0	
		+ 0 0 4	
		8 16 6	
		8 16 10	

The system is in a **safe state with** the sequence (*P2, P3, P1, P4*)

3. Can request for (3,2,1) by *P2* be granted?

Request \leq Need

3 2 1 \leq 1 0 1 the system raise an error, since Need \leq Request

Memory Management

Address Binding

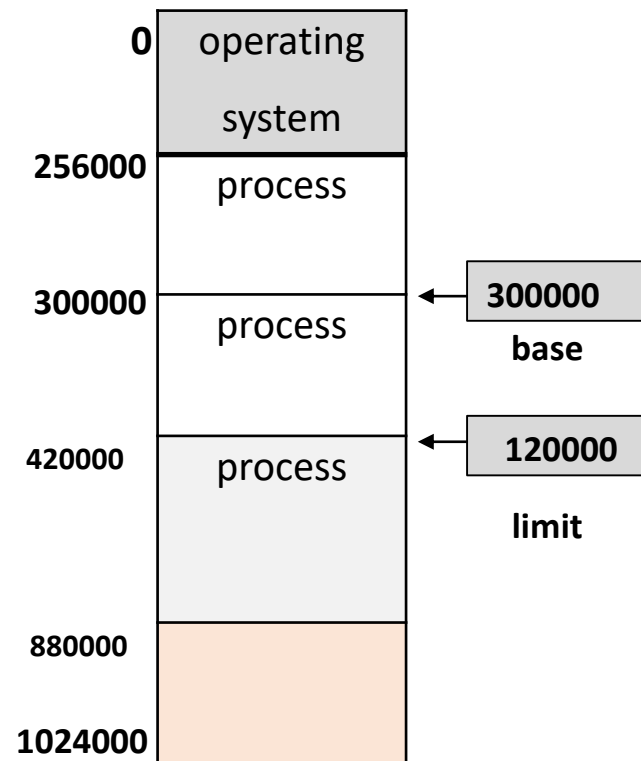
Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.

Depending on the memory management in use, the process may be moved between disk and memory during its execution.

The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.

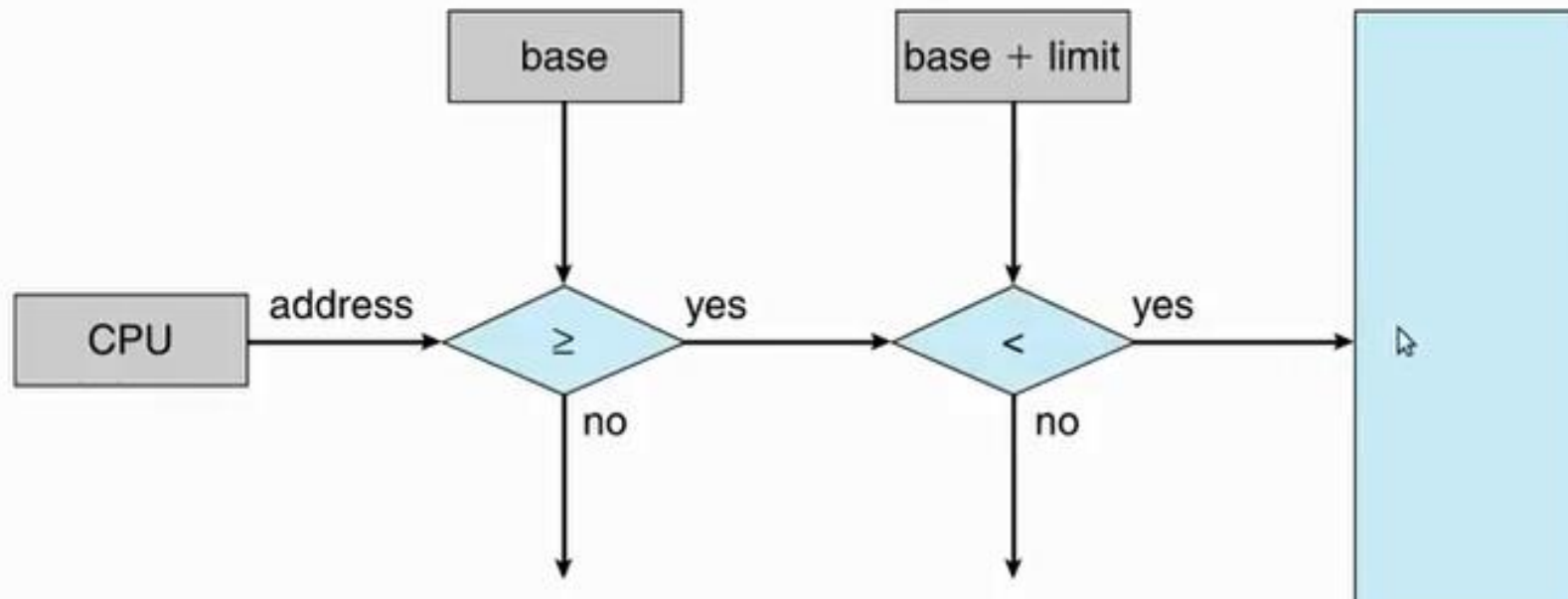
Base and Limit Registers

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





Hardware Address Protection



Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**,

An address seen by the memory unit that referred to as a **physical address**.

The set of all logical addresses generated by a program is a **logical (virtual) address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The mapping from **virtual** to **physical** addresses is done by a hardware device called the **memory-management unit (MMU)**.

The base register is called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 8.4).

The user program never sees the real physical addresses.

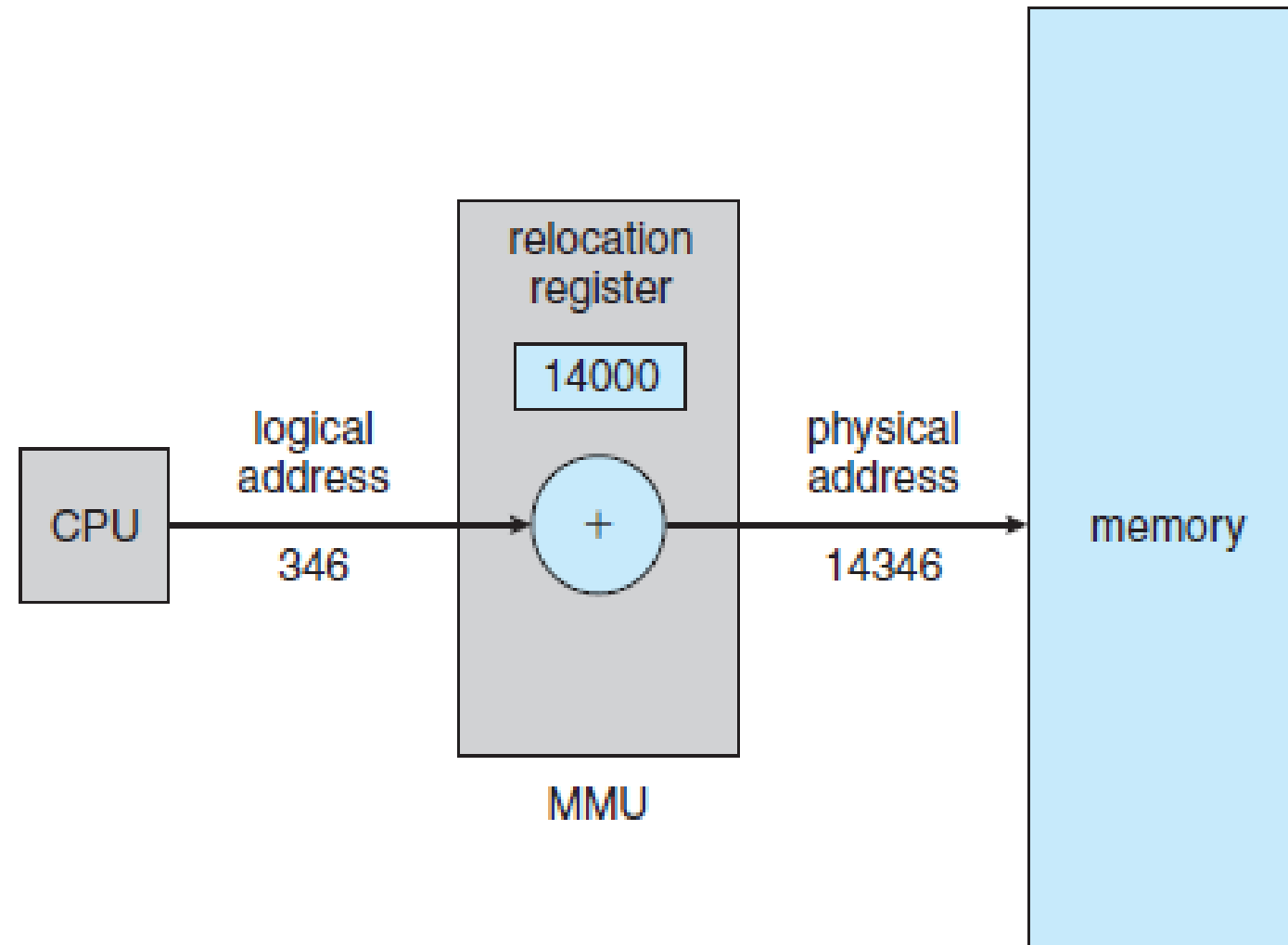


Figure 8.4 Dynamic relocation using a relocation register.

Quiz

Calculate the physical address if the logical address is 375 and the relocation address (MMU) is 15000.

What are the differences between the Logical addresses and Physical addresses in the Operating System?



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

1. Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

Processes	Memory
	100
	500
	200
	300
	600

In this example, best-fit turns out to be the best.

Example

Consider a swapping system in which memory consists of the following whole sizes in memory order: 10K, 4k, 20k, 18k, 7k, 9k, 12k, and 15k. Which hole is taken for successive segment request of i) 12k, ii) 10k, iii) 9k for first fit? Now repeat the question for best fit and worst fit.

Assignment

let us assume the jobs and the memory requirements as the following:

Job1 90k

Job2 20k

Job3 50k

Job4 10k

The free pace memory allocation blocks be:

Block1 50k

Block2 100k

Block3 90k

Block4 200k

Block5 50k

How would First Fit, Best Fit, and Worst Fit algorithms place processes?

Quiz

Regarding to the memory management. Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 210 Kb, 400 Kb, 100 Kb, and 420 Kb (in order)? Which algorithm makes the most efficient use of memory?

Dynamic Loading

The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**.

With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.

Contiguous Memory Allocation

In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.

Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Another allocation method is the **variable-partition scheme**. OS keeps a table indicating which parts of memory are available and which are occupied.

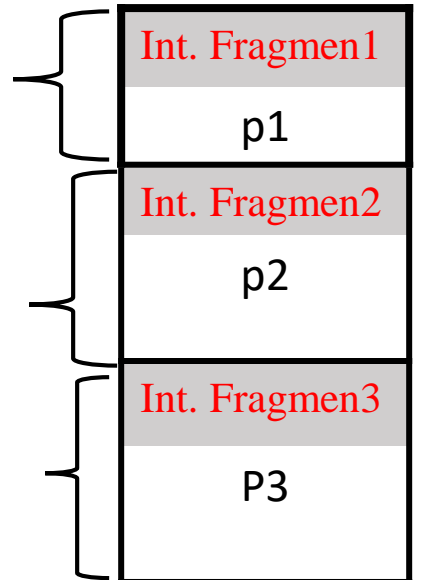
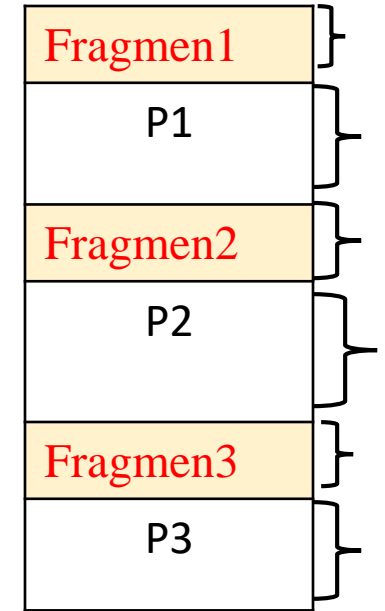
Initially, all memory is available for user processes and is considered one large block of available memory.

The problem of Contiguous Memory Allocation: Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

External Fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

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



Fragmentation (Cont.)

Reduce external fragmentation by **compaction**.

Compaction is a method for removing external fragmentation. External fragmentation may be decreased when dynamic partitioning is used for memory allocation by combining all free memory into a single large block. The larger memory block is used to allocate space based on the requirements of the new processes. This method is also known as **defragmentation**.

Segmentation

- **Segmentation** is a memory management technique to **avoid external fragmentation**. In Segmentation, a process is divided into multiple segments. The size of each segment is not necessarily the same which is different from paging. The module contained in a segment decides the size of the segment.

Each segment has a:

- **Segment Number(s)** - It is the number of bits required to represent a Segment. It is used as an index in the Segment Table.
- **Segment Offset(d)** - It is the number of bits required to represent the size of a Segment.
- The details about each segment are stored in a table called a **segment table**. It helps in the mapping of the two-dimensional logical addresses to the physical addresses.

There are two entries in the Segment Table.

- **Base Address** - It is the starting physical address of the particular segment inside the main memory.
- **Limit** - It denotes the size of a particular segment.

Segmentation (Cont...)

Segment Number from the Segment Table gives the base address of a Segment in the main memory and Segment Offset gives the size of that Segment.

These two together give the Physical Address of a Segment in the memory.
If $\text{Offset}(d) < \text{Limit}$ then the segment is fetched from the memory else there is an error.

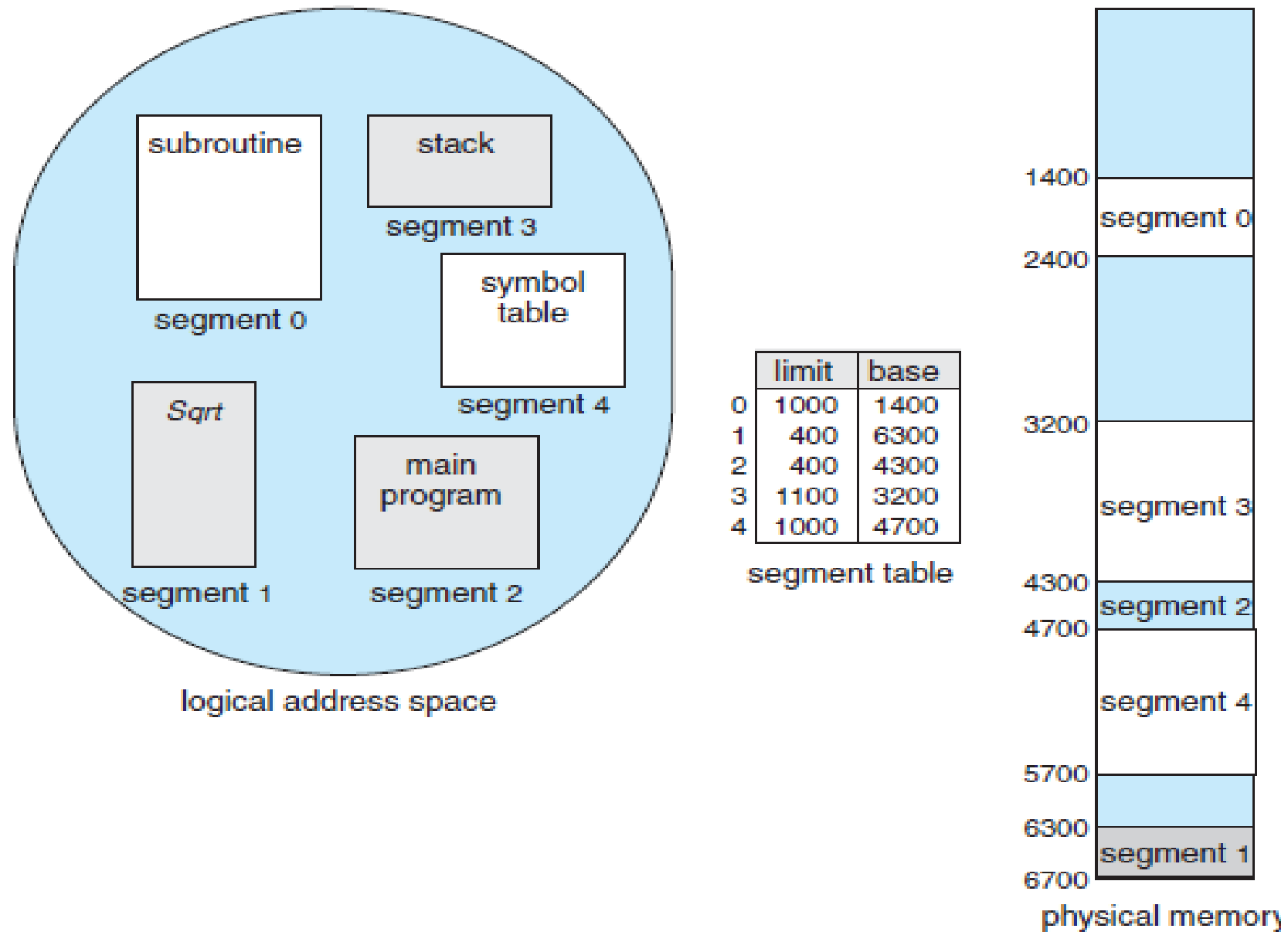


Figure 8.9 Example of segmentation.

Paging

- **Paging** is another memory-management technique in which the system stores and retrieves data from secondary storage. Paging **avoids external fragmentation**
- Paging helps in retrieving the processes from the secondary memory in the form of **pages**. In paging, processes are divided into equal parts called pages, and main memory is also divided into equal parts and each part is called a **frame**.
- Each page gets stored in one of the frames of the main memory whenever required. So, the size of a frame is equal to the size of a page. Pages of a process can be stored in the non-contiguous locations in the main memory.
- The **Page Table** contains the base address of each page inside the Physical Memory. It is then combined with Page Offset to get the actual address of the required data in the main memory.

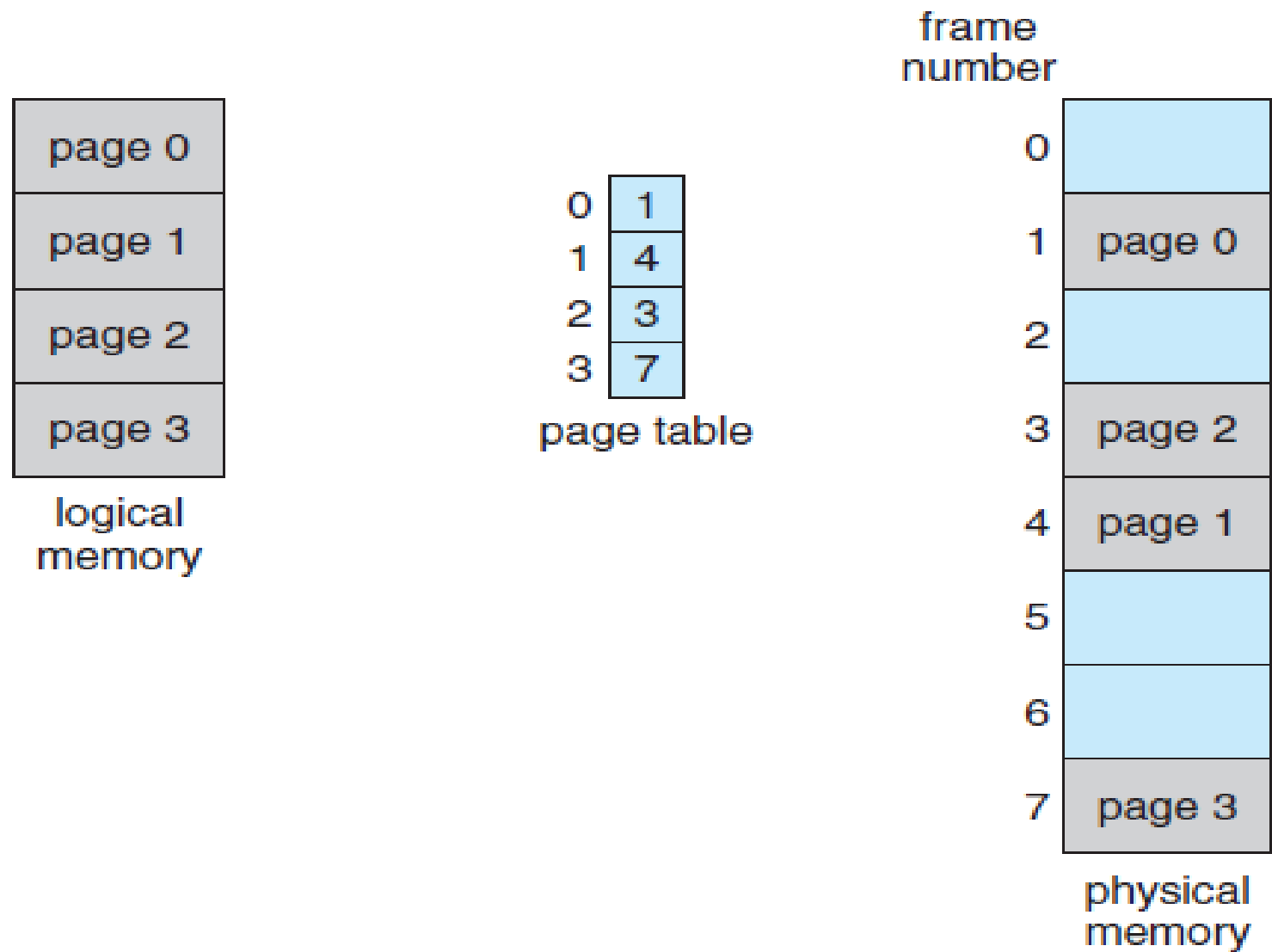


Figure 8.11 Paging model of logical and physical memory.

Virtual Memory

A virtual memory system attempts to optimize the use of the main memory (the higher speed portion) with the hard disk (the lower speed portion). In effect, virtual memory is a technique for using the secondary storage to extend the limited size of the physical memory beyond its actual physical size.

It is usually the case that the available physical memory space will not be enough to host all the parts of a given active program. Those parts of the program that are currently active are brought to the main memory while those parts that are not active will be stored on the magnetic disk.

Virtual Memory....

If the segment of the program requested by the processor is **not in the main memory at the time of the request**, then such **segment will have to be brought from the disk to the main memory**.

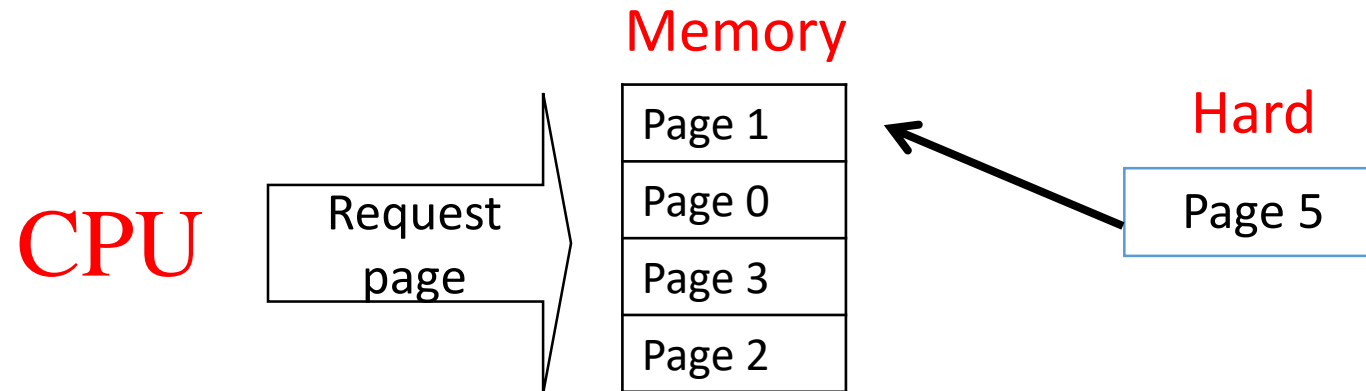
Movement of data between the disk and the main memory takes the form of **pages**. **A page is a collection of memory words**, which can be moved from the disk to the RAM when the processor requests accessing a word on that page.

A **page fault** occurs when the page required by the processor does not **exist in the RAM** and has to be brought from the disk.

Replacement Algorithms (Policies)

When a processor needs a page, not in memory, the operating system must decide which resident page is to be replaced by the requested page. The technique used in the virtual memory that makes this decision is called the **replacement policy**.

There exists a number of possible replacement mechanisms:



Replacement Algorithms (Policies)

- Random Replacement According to this replacement policy, a page is selected randomly for replacement.
- First-In-First-Out (FIFO) Replacement According to this replacement policy, the page that was loaded before all the others in the main memory is selected for replacement.
- Least Recently Used (LRU) Replacement According to this technique, page replacement is based on the pattern of usage of a given page residing in the main memory regardless of the time spent in the main memory. The page that has not been referenced for the longest time while residing in the main memory is selected for replacement.



First-In-First-Out (FIFO) Algorithm

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

reference string

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



page frames

15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - ▶ **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue

Quiz:

Consider the following page reference using four frames that are initially empty. Find the page faults using FIFO algorithm, where the page reference sequence:

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

Assignment

Q1. Consider the following page reference sequence:

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

How many page faults would occur for the FIFO replacement algorithms, assuming there are four available frames? Remember all frames are initially empty.

Q2. What are the differences between paging and Segmentation?

Q3. What is the difference between internal and external fragmentation, paging, and Segmentation?