BCA 3rd Semester
Operating system
Course Code: BCAC0022

Presented by:
Jayati Krishna Goswami
Assistant Professor, Dept. Of CEA
GLA University, Mathura

BCAC 0022: OPERATING SYSTEM

Objective:

Module No.	Content	Teaching Hours
I	**Introduction:** Operating system and functions, Classification of Operating systems: Batch, Interactive, Time sharing, Real Time System, Multiprocessor Systems, Multiuser Systems, Multithreaded Systems, Operating System Structure, System Components, Operating System Services, Kernels, Monolithic and Microkernel Systems. **Process Management:** Process Concept, Process States, Process Synchronization, Critical Section, Mutual Exclusion, Classical Synchronization Problems, Process Scheduling, Process States, Process Transitions, Scheduling Algorithms Inter-process Communication, Threads and their management, Security Issues. **CPU Scheduling:** Scheduling Concepts, Techniques of Scheduling, Preemptive and Non-Preemptive Scheduling: First-Come-First-Serve, Shortest Request Next, Highest Response Ration Next, Round Robin, Least Complete Next, Shortest Time to Go, Long, Medium, Short Scheduling, Priority Scheduling. Deadlock: System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock. **Memory Management:** Memory allocation, Relocation, Protection, Sharing, Paging, Segmentation, Virtual Memory, Demand Paging, Page Replacement Algorithms, Thrashing.	30
II	**I/O Management and Disk Scheduling:** I/O devices, and I/O subsystems, I/O buffering, Disk storage and disk scheduling, RAID. **File System:** File concept, File organization and access mechanism, File directories, and File sharing, File system implementation issues, File system protection and security. **Shell introduction and Shell Scripting:** What is shell and various type of shell, Various editors present in linux, Different modes of operation in vi editor, What is shell script, Writing and executing the shell script, Shell variable (user defined and system variables) System calls, Using system calls, Pipes and Filters, Decision making in Shell Scripts (If else, switch), Loops in shell, Functions, Utility programs (cut, paste, join, tr , uniq utilities), Pattern matching utility (grep).	30

Text Book:

- Abraham Silberschatz, Greg Gagne, and Peter B. Galvin, "Operating System Concepts," Tenth Edition, Wiley, 2018.

Reference Books:

- Andrew S. Tanenbaum and Herbert Bos,"Modern Operating Systems," Fourth Edition, Pearson, 2014.
- William Stallings, "Operating Systems: Internals and Design Principles," Seventh Edition, Prentice Hall, 2011.
- Dhanjay Dhamdhere, "Operating Systems," First Edition, McGraw-Hill, 2008.
- Milan Milankovic "Operating systems, Concepts and Design" McGraw Hill.

Outcome: *A student who successfully completes the course will have the ability to:*

- CO1: Understand role, responsibilities, features, and design of operating system.
- CO2: Analyze memory management schemes and process scheduling algorithms.
- CO3: Apply process synchronization techniques to formulate solution for critical section problems.
- CO2: Illustrate concept of disk scheduling.
- CO5: Evaluate process deadlock handling techniques.

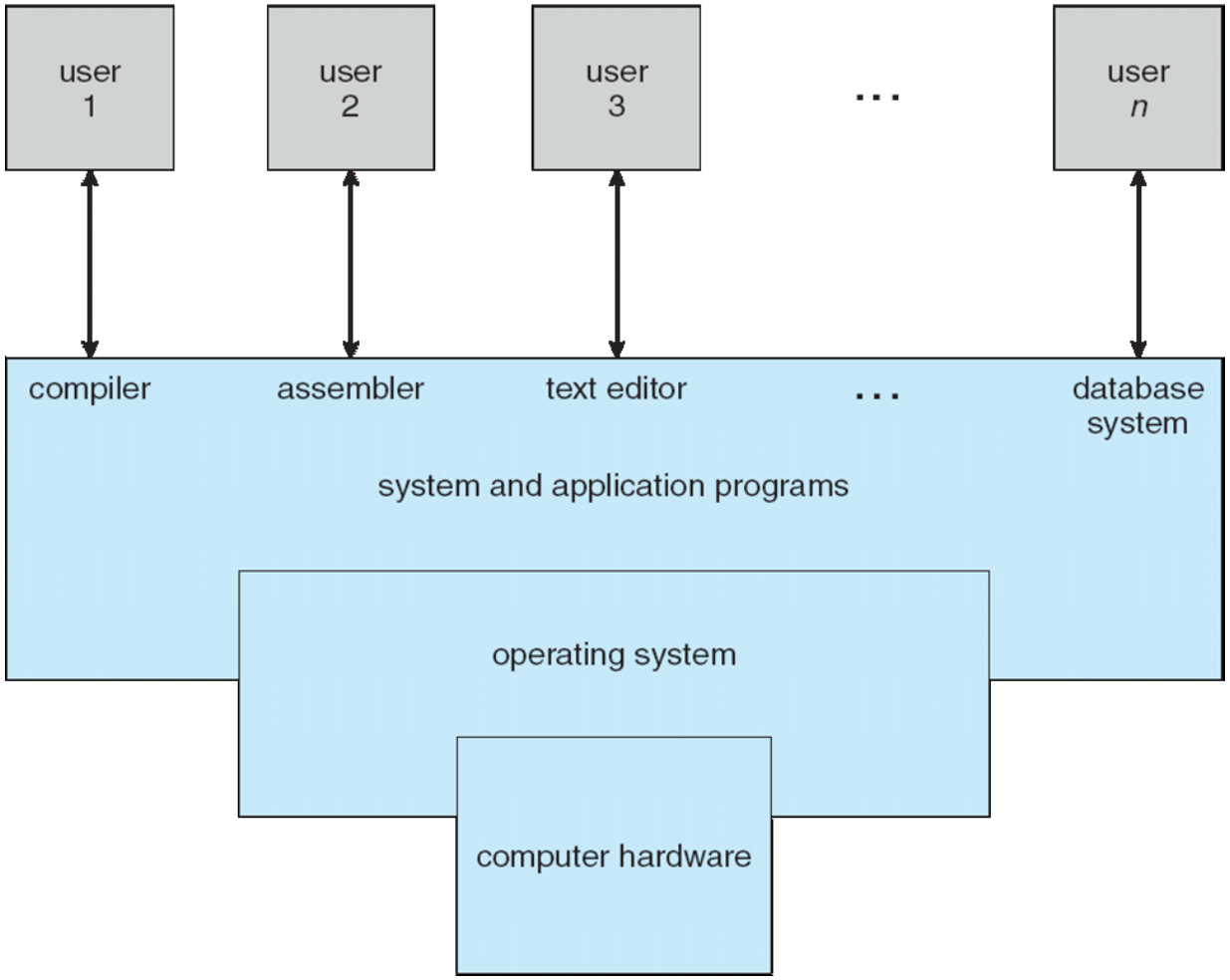
An Operating System

- An **Operating System** (**OS**) is a collection of program which provides an interface between a computer user and computer hardware.
- An **operating system** is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.
- A Collection program that acts as an intermediary between a user of a computer and the computer hardware.
- It is responsible for the management and coordination of activities and the sharing of the resources of the computer.

Computer System Components

- **Hardware** Provides basic computing resources (CPU, memory, I/O devices).
- **Operating System** Controls and coordinates the use of hardware among application programs.
- **Application Programs** Solve computing problems of users (compilers, database systems, video games, business programs such as banking software).
- **Users** People, machines, other computers

Four Components of a Computer System



Operating System Views

- Resource allocator :to allocate resources (software and hardware) of the computer system and manage them efficiently.

- Control program : Controls execution of user programs and operation of I/O devices.

- Kernel :The program that executes forever (everything else is an application with respect to the kernel).

Goals of an Operating System

- Simplify the execution of user programs and make solving user problems easier.
- Use computer hardware efficiently.
- Allow sharing of hardware and software resources.
- Make application software portable and versatile.
- Provide isolation, security and protection among user programs.
- Improve overall system reliability

Functions/Components of Operating System:

Process management

- A program in its execution state is known as **process.**
- A process needs certain resources including CPU time, memory, files and I/O devices to accomplish its task.
- These resources are either given to the process when it is created or allocated to it while it is running.
- A program is a passive *entity* such as contents of a file stored on the disk whereas a process is an active *entity*

The operating system is responsible for the following activities in process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling.

2. Memory management

Main memory is a collection of quickly accessible data shared by the CPU and I/O devices.

The central processor reads instructions from main memory (during instruction-fetch cycle) and both reads and writes data from main memory (during data-fetch cycle).

The operating system is responsible for the following activities in memory management:

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes and data to move into and out of memory
- Allocating and deallocating memory space as needed.

3. File-System Management

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

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Backing up files on stable (nonvolatile) storage media.

4. Secondary storage Management The operating system is responsible for the following activities with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

5. I/O system Management

The operating system is responsible for the following activities with I/O subsystem:

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

6. Protection and Security

- A mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have been allowed proper authorization from the operating system.
- For example, memory-addressing hardware ensures that a process can execute only within its own address space.
- Protection is a mechanism for controlling the access of processes or users to the resources defined by a computer system.

7. Networking

- A distributed system is a collection of physically separated computer systems that are networked to provide the users with access to the various resources that the system maintains.
- Access to a shared resource increases computation speed, functionality, data availability, and reliability.
- Network Operating System (NOS) provides remote access to the users.
- It also provides the sharing of h/w and s/w resources from remote machine to own systems.

8.Command Interpreter

- To interface with the operating System we use command-line interface or **command interpreter** that allows users to directly enter commands that are to be performed by the operating system.
- The main function of the command interpreter is to get and execute the user-specified commands. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. Eg: MS-DOS and UNIX shells.

Types of Operating System

- **Single user Operating system** This OS provides the environment for single user i.e. only one user can interact with the system at a time. Eg: MS-DOS, MS WINDOWS-XP, ME, 2000 etc.

- **Multi user Operating System** This OS provides the environment for multiple users i.e. many user can interact with the system at a time. These users are remotely connected to a system taking benefits of shared resources of master system through networking. Eg: UNIX, LINUX.

Serial Processing Operating System

- Early computer from late 1940 to the mid 1950.
- The programmer interacted directly with the computer hardware.
- These machine are called bare machine as they don't have OS.
- Every computer system is programmed in its machine language.
- Uses Punch Card, paper tapes and language translator

- In a typical sequence first the editor is been called to create a source code of user program then translator is been called to convert source code into its object code, finally the loader is been called to load its executable program into main memory for execution.

- If syntax errors are detected than the whole program must be restarted from the beginning.

Batch processing Operating System

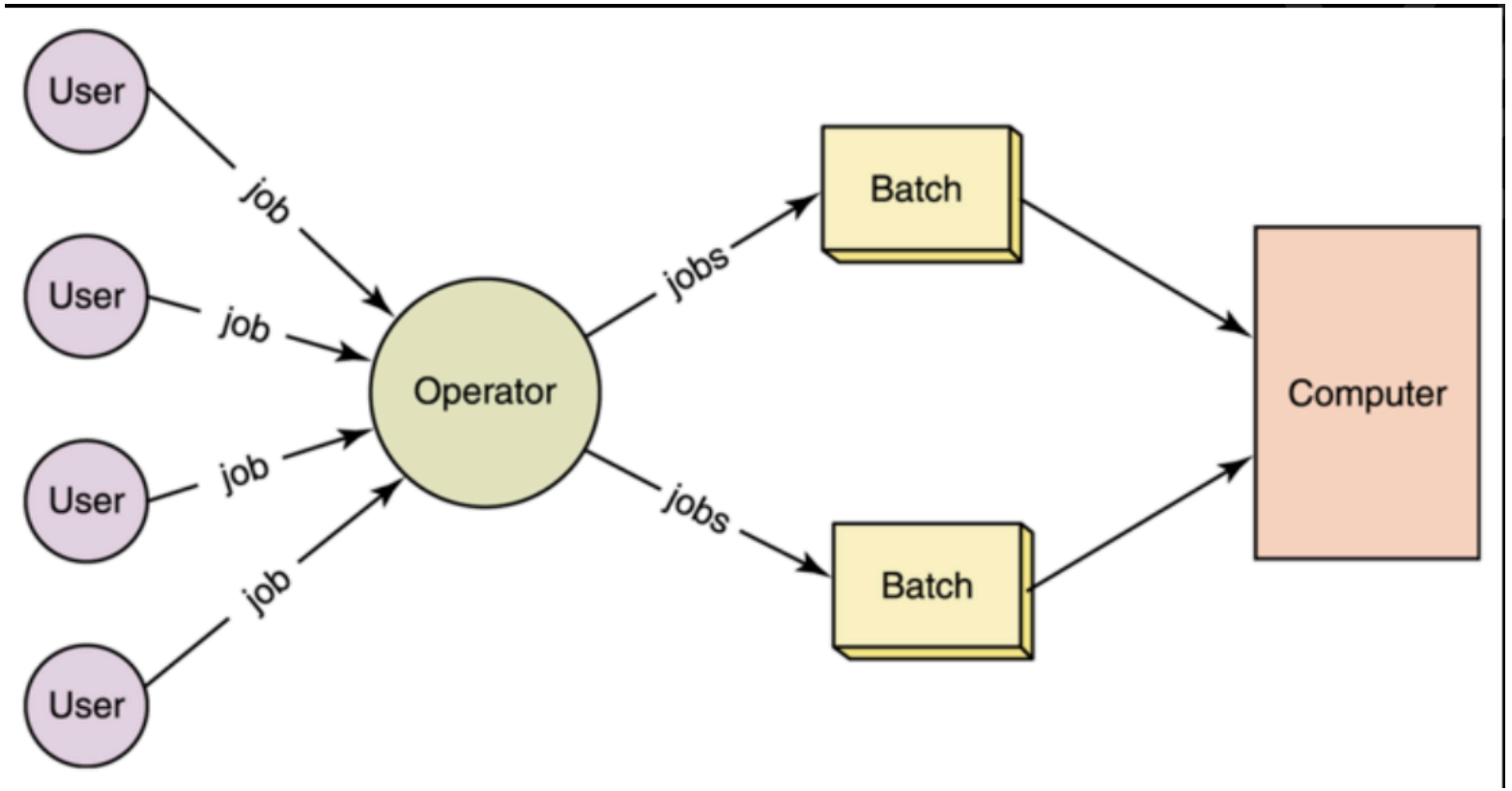
- Early computers were not interactive device, there user use to prepare a job which consist three parts
    1. Program
    2. Control information
    3. Input data
- Only one job is given input at a time as there was no memory, computer will take the input then process it and then generate output.
- Common input/output device were punch card or tape drives. So these devices were very slow, and processor remain ideal most of the time.

Batch processing Operating System

Batch processing Operating System

- To speed up the processing job with similar types (for e.g. FORTRAN jobs, COBOL jobs etc.) were batched together and were run through the processor as a group (batch).
- In some system grouping is done by the operator while in some systems it is performed by the 'Batch Monitor' resided in the low end of main memory)
- Then jobs (as a deck of punched cards) are bundled into batches with similar requirement.

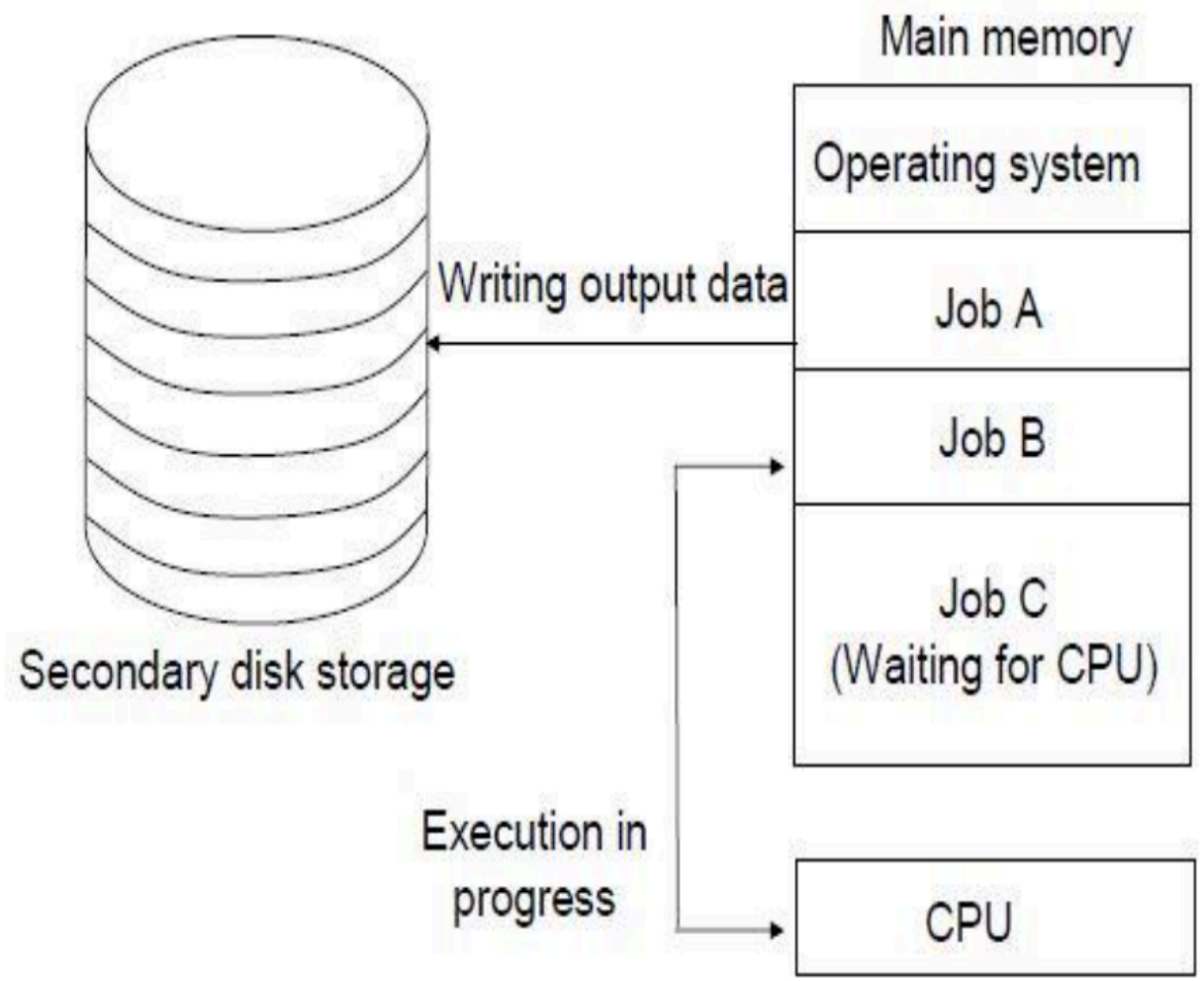
Batch processing Operating System



Multiprogramming

- Multiprogramming is a technique to execute number of programs simultaneously by a single processor.
- In Multiprogramming, number of processes reside in main memory at a time.
- The OS picks and begins to executes one of the jobs in the main memory.
- If any I/O wait happened in a process, then CPU switches from that job to another job.
- Hence CPU in not idle at any time.

Multiprogramming

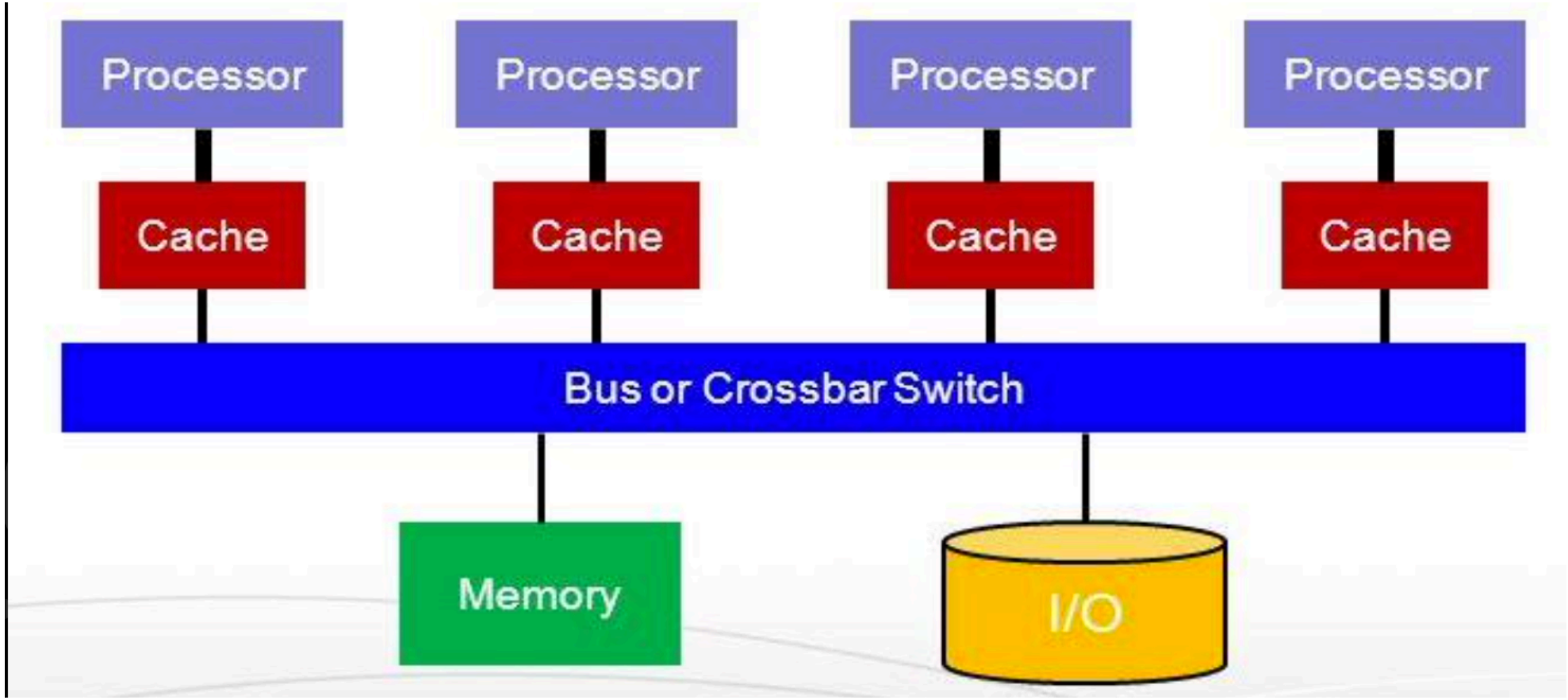


- Figure dipicts the layout of multiprogramming system.
- The main memory consists of 3jobs at a time, the CPU executes one by one.
- **Advantages & Disadvantages:**
- Efficient memory utilization
- Throughput increases
- CPU is never idle, so performance increases.
- Complex Scheduling: Difficult to program.
- Complex Memory Management: Intricate handling of memory is required.

Multiprocessor system :-This system has more than one processor which share common bus, clock, peripheral devices and sometimes memory.

These systems have following advantages:

- **Increased throughput** By increasing the number of processor, we get more work done in less time.
- **Economy of scale** Multiprocessor system are cost effective as several processors share same resources of the system(memory, peripherals etc.)
- **Increased reliability** each processor is been fairly allotted with different job, failure of one processor will not halt the system, only it will slower down the performance. for example if we have ten processor and one fails, then each of the remaining nine processor share the work of failed processor. Thus the system will be 10% slower rather than failing altogether



This system can be categorized into:

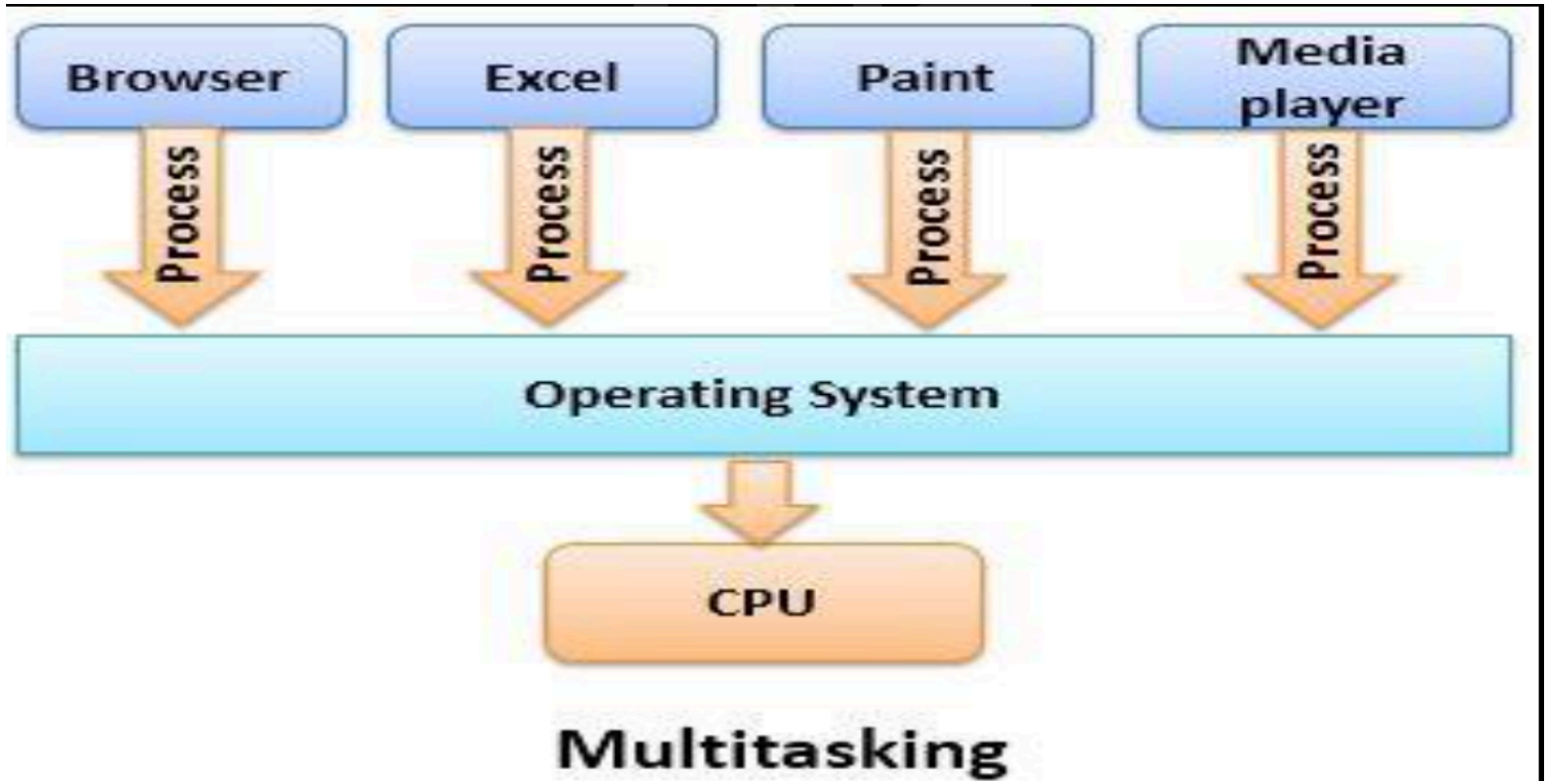
- **i) SMP (Symmetric multiprocessing)**In this system each processor runs an identical copy of the task and however these processor communicate with each other whenever needed. Eg Windows NT, Solaris, Unix/Linux.

- **ii) ASMP(Asymmetric multiprocessing)** In asymmetric multiprocessing each processor runs a specific task. As there is a master processor which controls the whole system,and other processor looks to the master for instructions. Eg Sun OS ver. 4

Time sharing System(Multitasking)

- Time sharing, or multitasking, is a logical extension of multiprogramming.
- Multiple jobs are executed by switching the CPU between them.
- In this, the CPU time is shared by different processes, so it is called as "Time sharing Systems".
- A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.
- Time slice is defined by the OS, for sharing CPU time between processes.
- Examples: Multics, Unix, etc.,

Time sharing System(Multitasking)

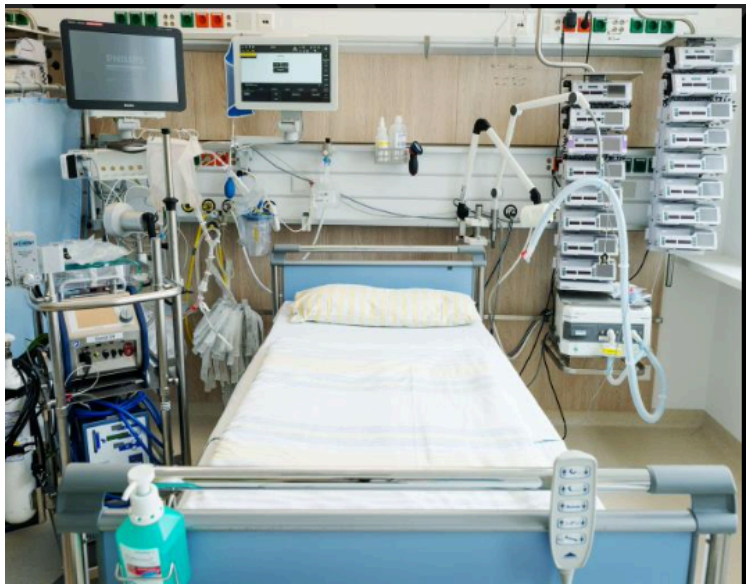


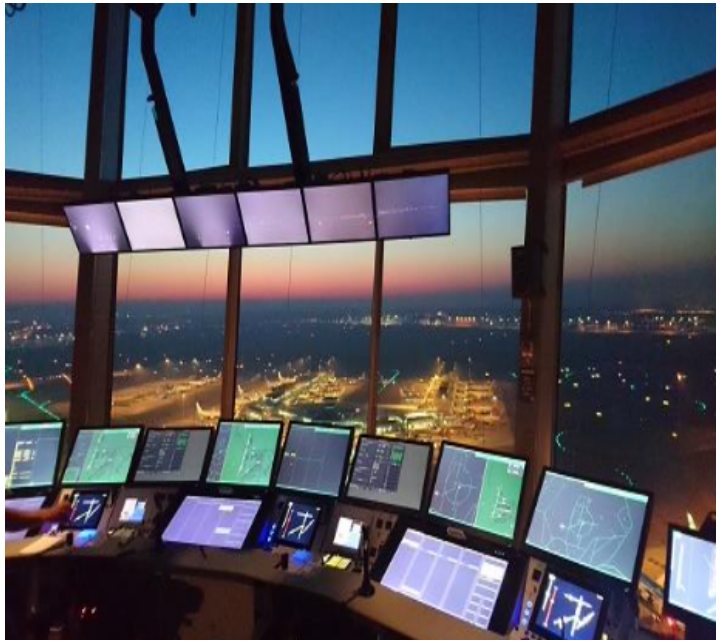
Time sharing System(Multitasking)

1. It allows many users to share the computer simultaneously. the CPU executes multiple jobs (May belong to different user) by switching among them, but the switches occur so frequently that, each user is given the impression that the entire computer system is dedicated to his/her use, even though it is being shared among many users.
2. In the modern operating systems, we are able to play MP3 music, edit documents in Microsoft Word, surf the Google Chrome all running at the same time. (by context switching, the illusion of parallelism is achieved)
3. For multitasking to take place, firstly there should be multiprogramming i.e. presence of multiple programs ready for execution. And secondly the concept of time sharing.

Real time Operating System

- A real time system has well defined fixed time constraints, processing must be done within defined constraints or system will get failed.
- System that controls scientific system, experimenting medical system, industrial control system and certain display systems are real time system.
- They are also applicable to automobile engine fuel system, home appliance controller and weapon systems.
- There are two types of real system:





- **i) <u>Hard real time system</u>** This system guarantees that critical tasks be completed on time. For this all the delays in the system should be bounded, from the retrieval of stored data to the time it takes operating system to finish any request made to it.

- **ii<u>) Soft real time system</u>** This is less restrictive type of system defined as **not hard real-time**, simply providing that a critical real-time task will receive priority over other tasks and that it will retain the priority until it completes.

Network Operating System :-An Os that includes special functions for connecting computers and devices into a LAN. Some OS such as UNIX and the Mac OS, having networking functions built in.

Some popular NOS's for DOS and Windows systems include Novell Netware, Microsoft LAN Manager and Windows NT.

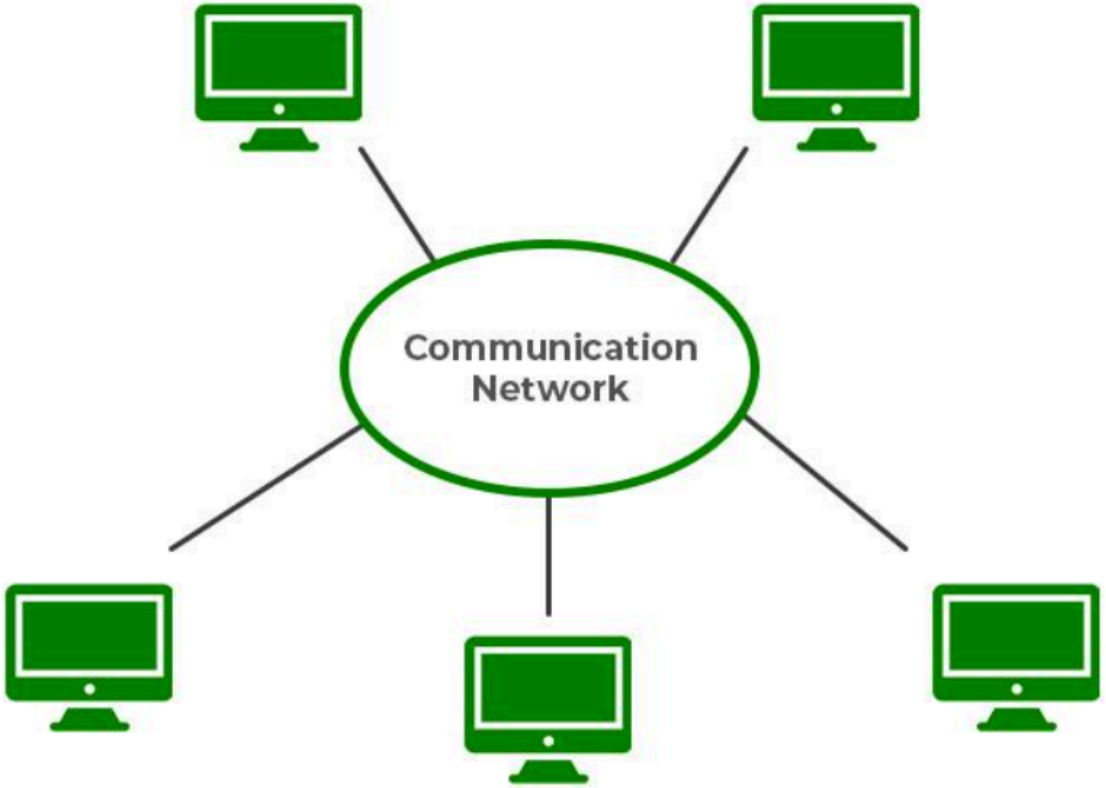
- **Some characteristics**
- Each computer has its own private OS, instead of running part of a global system wide operating system.
- Each user normally works on his/her own system.

Distributed Operating System :-It hides the existence of multiple computers from the user i.e. the user doesn't know that many computers are being used to process the data.

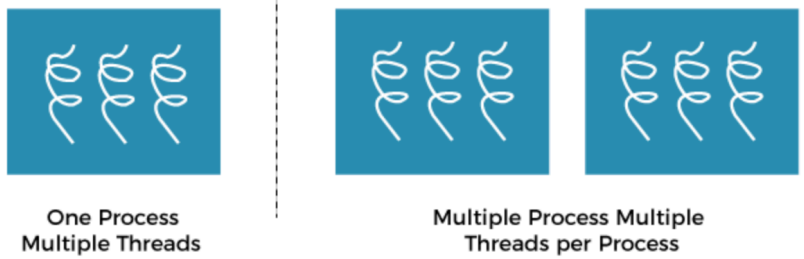
These computers may be located at many places around the globe. This OS provides provide single-system image to its users. All these computers work in close coordination with each other.

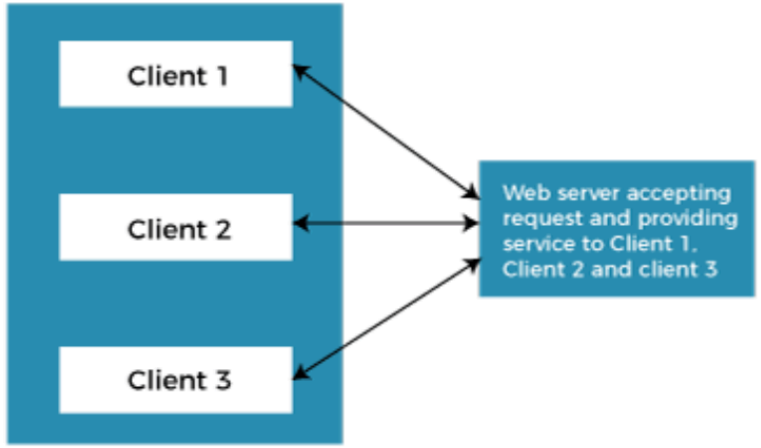
- In this OS, each processor has its own memory and clock. The processor communicates with each other through various communication lines such as high speed buses and telephone lines.
- There are four major reasons for building distributed systems: resource sharing, computation speedup, reliability, and communication.

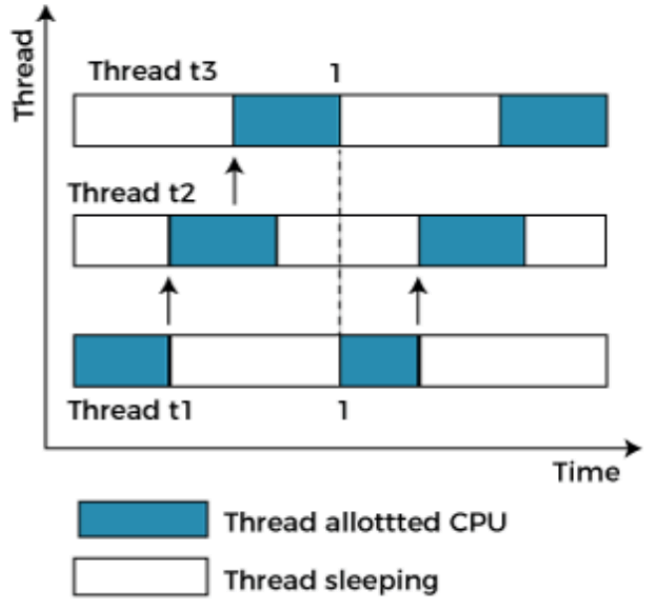
Distributed Operating System



Multithreading Operating System







Multithreading Operating System

- Multithreading allows the application to divide its task into individual threads.
- In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading.
- With the use of multithreading, multitasking can be achieved.
- The main drawback of single threading systems is that only one task can be performed at a time, so to overcome the drawback of this single threading, there is multithreading that allows multiple tasks to be performed.

Operating-System Services

1) **User interface** Almost all operating systems have a **user interface** (UI).This interface can take several forms.

- **command-line interface (CLI)** which uses text commands.
- **graphical user interface (GUI)** the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and keyboard to enter text.

2) Program execution:- The system should to load a program into memory and to run that program. The program must be able to end its execution.

3) I/O operations:- A running program may require I/O operation(such as recording to a CD or DVD drive or blanking a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore the operating system must provide a means to do I/O.

4)File-system manipulation:- Programs need to read and write files. They also need to create and delete. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.

5**) Input / Output Operations**:-A program which is currently executing may require I/O, which may involve file or other I/O device. For efficiency and protection, users cannot directly govern the I/O devices. So, the OS provide a means to do I/O Input / Output operation which means read or write operation with any file

6) **Communications.** There are many circumstances in which one process needs to exchange information with another process.

Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network.

7) Error detection. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (a network failure, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or too-great use of CPU time).

For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

8. Resource allocation. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system trough various methods such as CPU-scheduling.

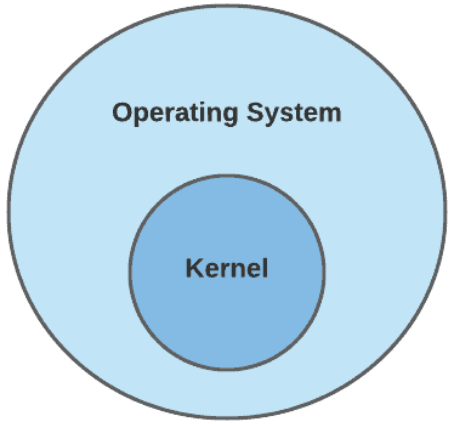
9. Accounting. OS keeps track of which users use how much and what kind of computer resources. This record keeping may be used for accounting

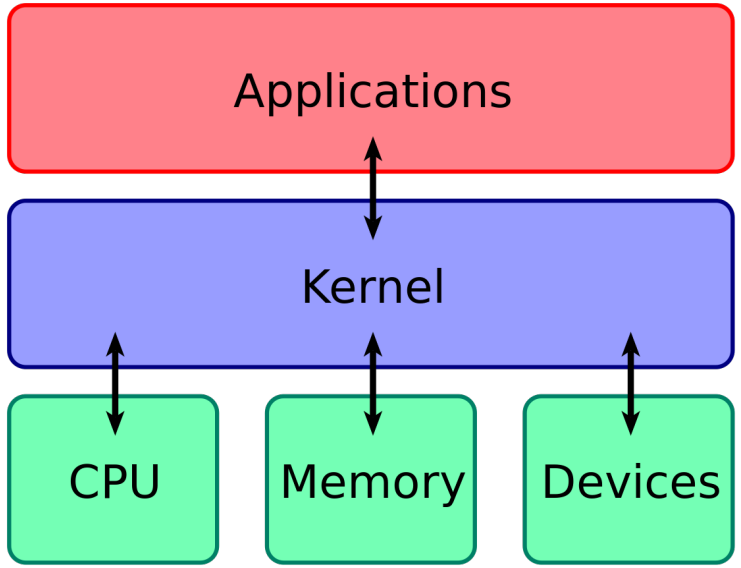
10. Protection and security. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself.

Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important by means of a password, to gain access to system resources.

Kernel

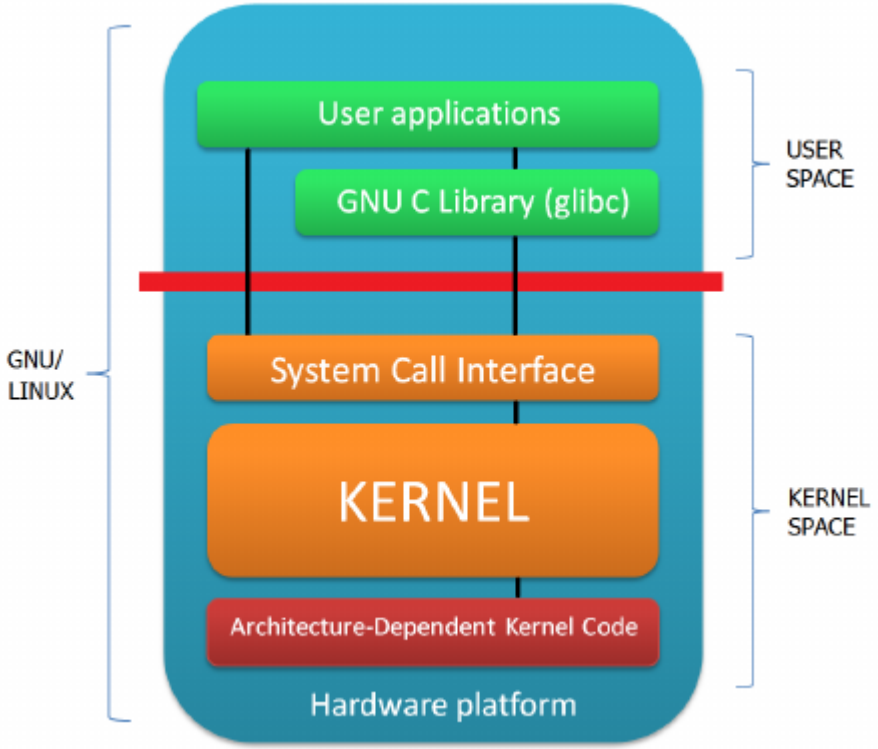
- Kernel is a part & lowest layer of a operating system.
- It provides service to all other parts of OS.
- It acts as interface between hardware and processes of a computer.
- It loads first in memory after OS has been loaded and remains in memory until shutdown.

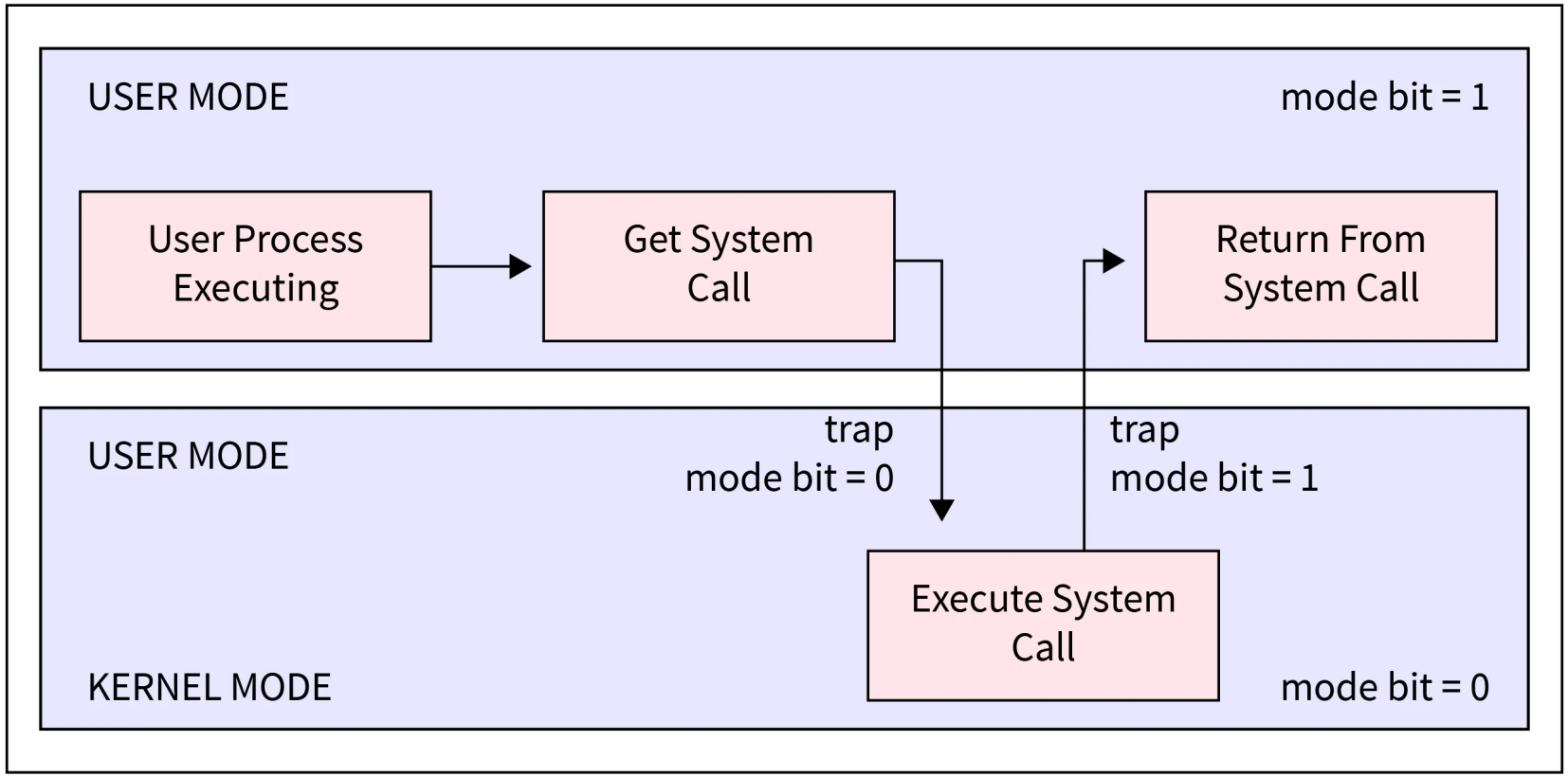




Kernel

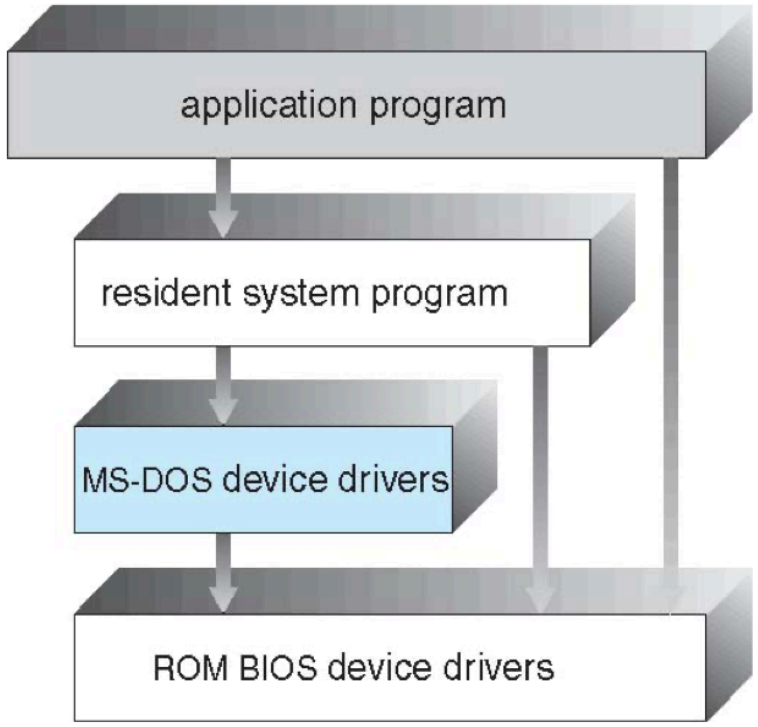
- Functions are device, task and memory management.
- Memory has user space and protected kernel space.

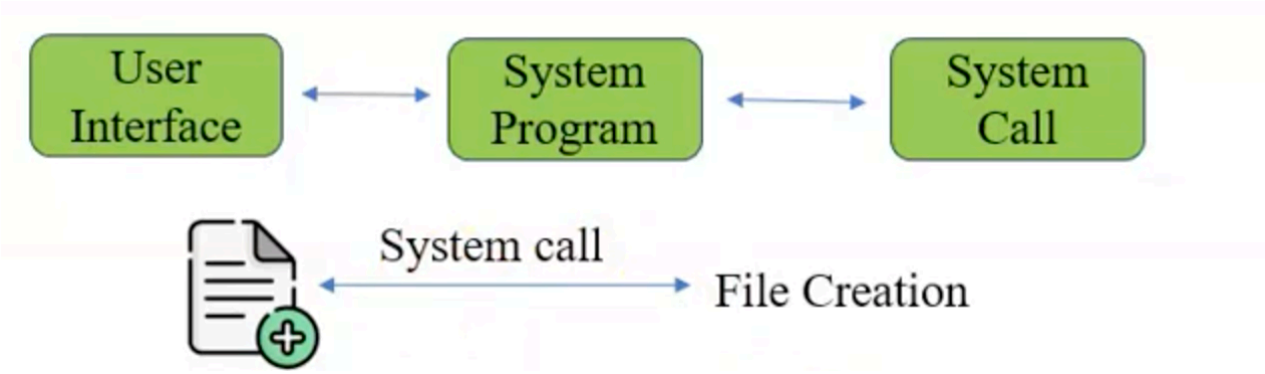




OS Structures

- **Simple Structure- MS-DOS**

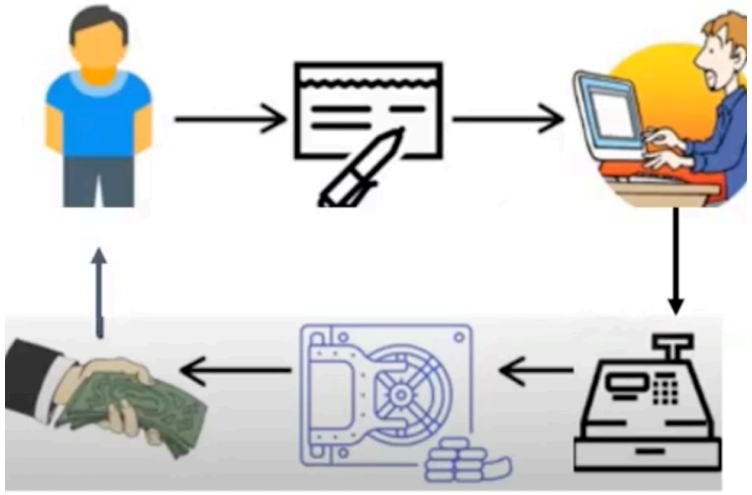


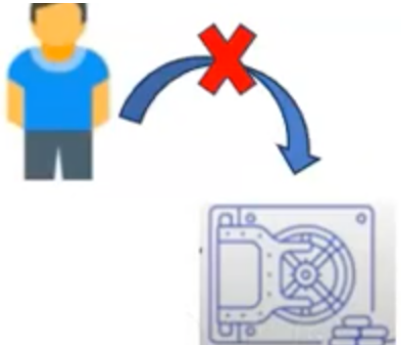


System becomes **vulnerable** and malicious programs can access base H/W

- Basic Input Output System stored in ROM
- Drivers to start the system

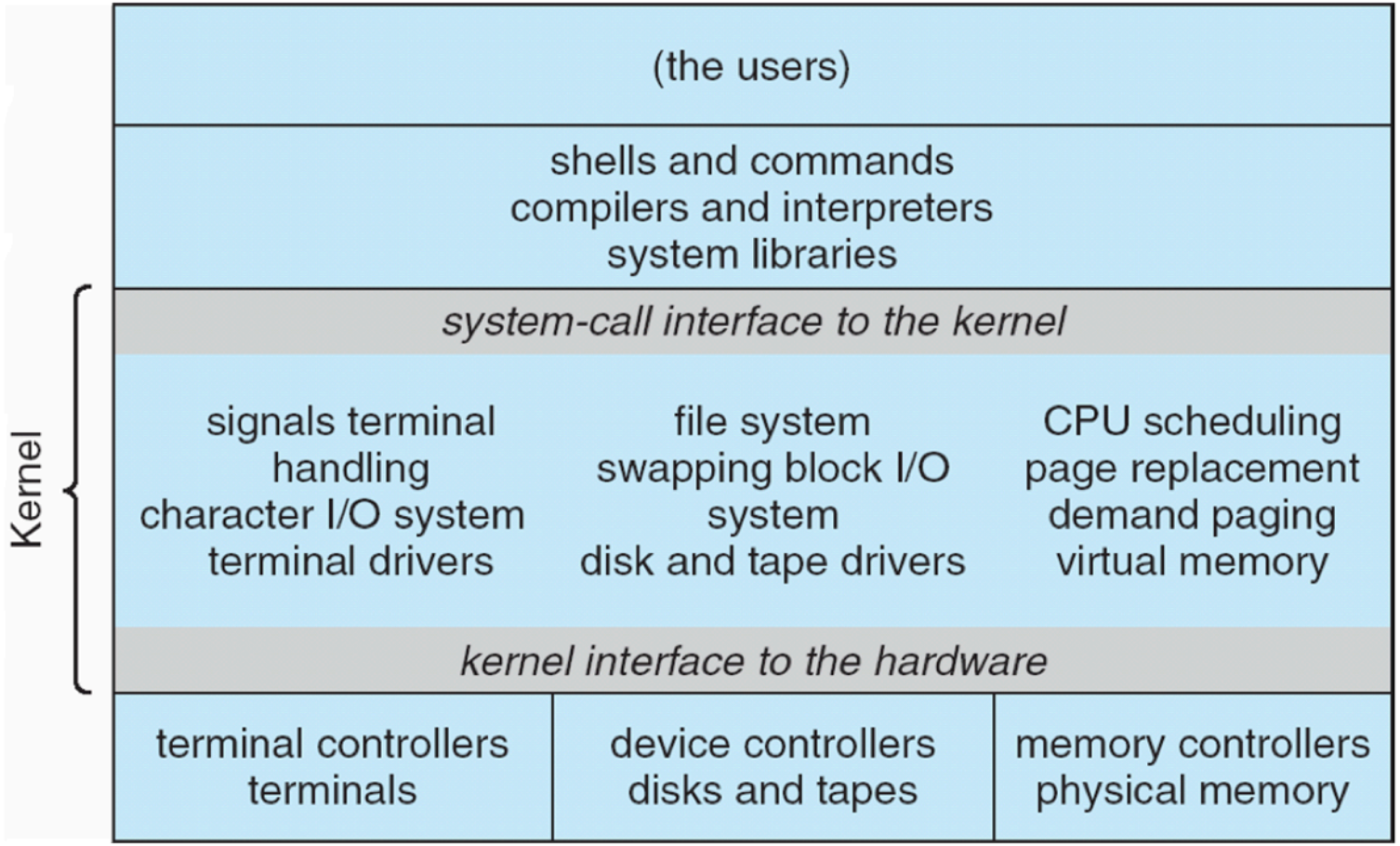
- Easy to Develop, maintain and provides good performance as layers are limited.
- Vulnerable to to unauthorised access and BIOS and Device drivers is accessible to all layers.





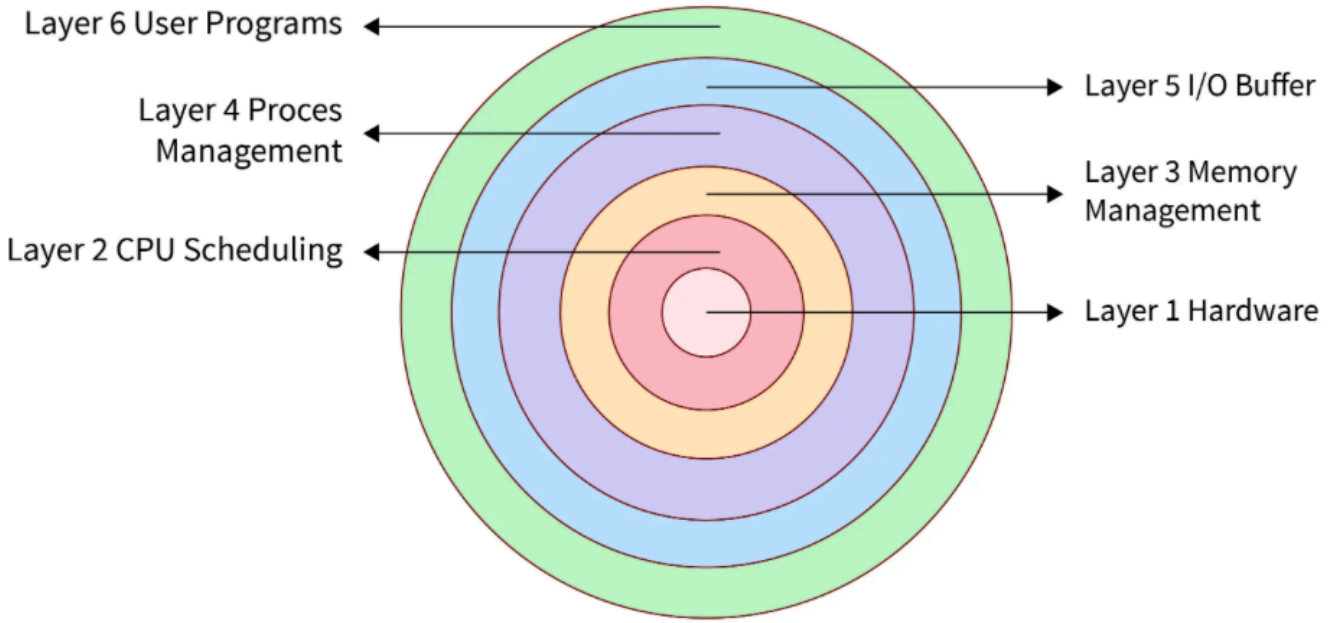
- **Monolithic Structure - UNIX**

Too Many functions packed into one level. maintenance and implementation difficult.



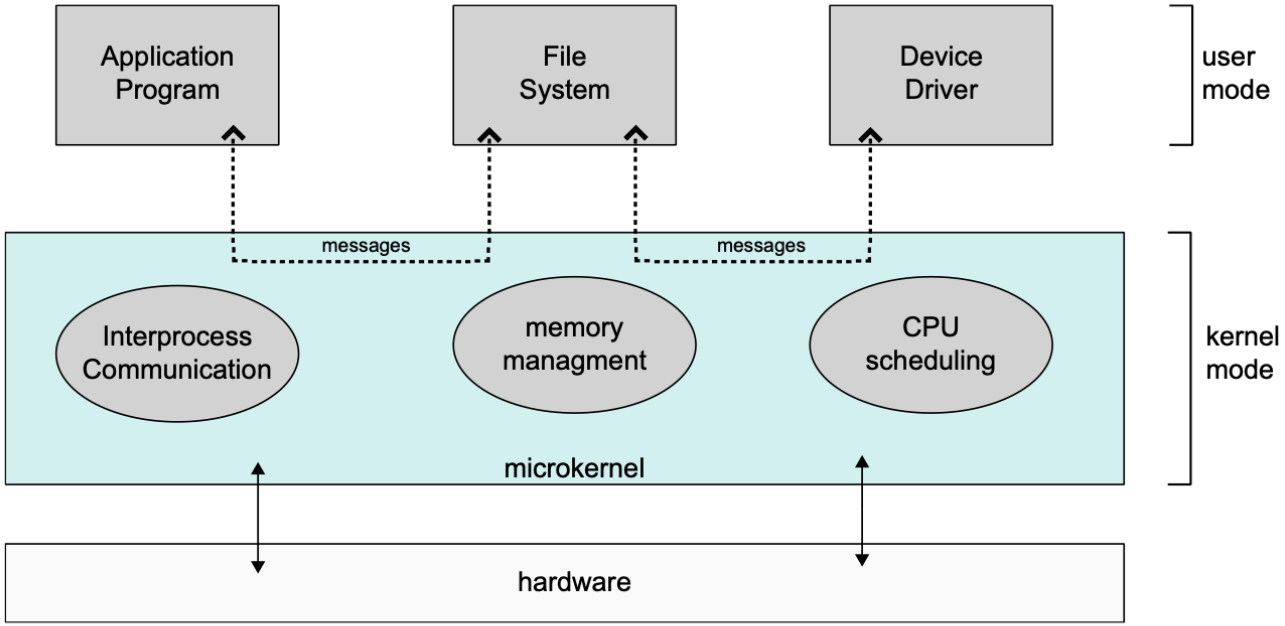
- **Layered structure**

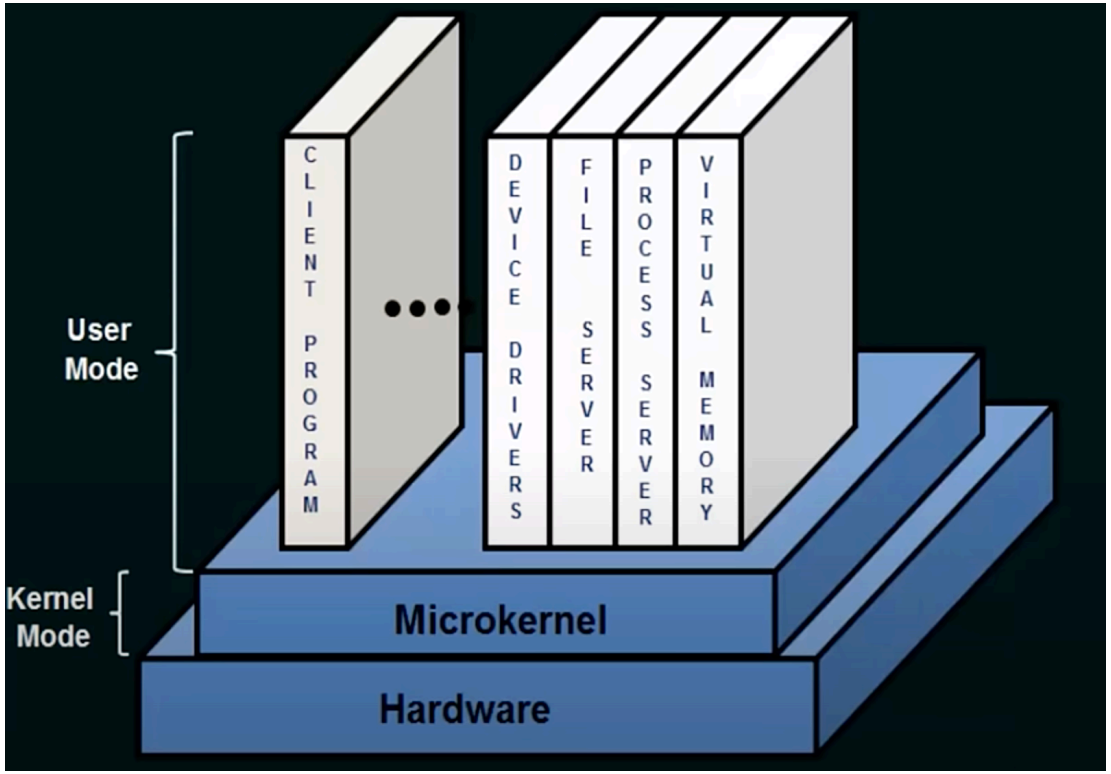
Easy to implement and debug.
But designing & deciding of layers are difficult.
also response to top layer from bottom layer may be slow.



- Layered Approach - With proper hardware support, operating systems can be broken into pieces. The operating system can then retain much greater control over the computer and over the applications that make use of that computer.
- Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems.
- Under a top-down approach, the overall functionality and features are determined and are separated into components.
- A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

- **Microkernel structure- Mach**





- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.
- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The MINIX 3 microkernel, for example, has only approximately 12,000 lines of code. Developer Andrew S. Tanenbaum.

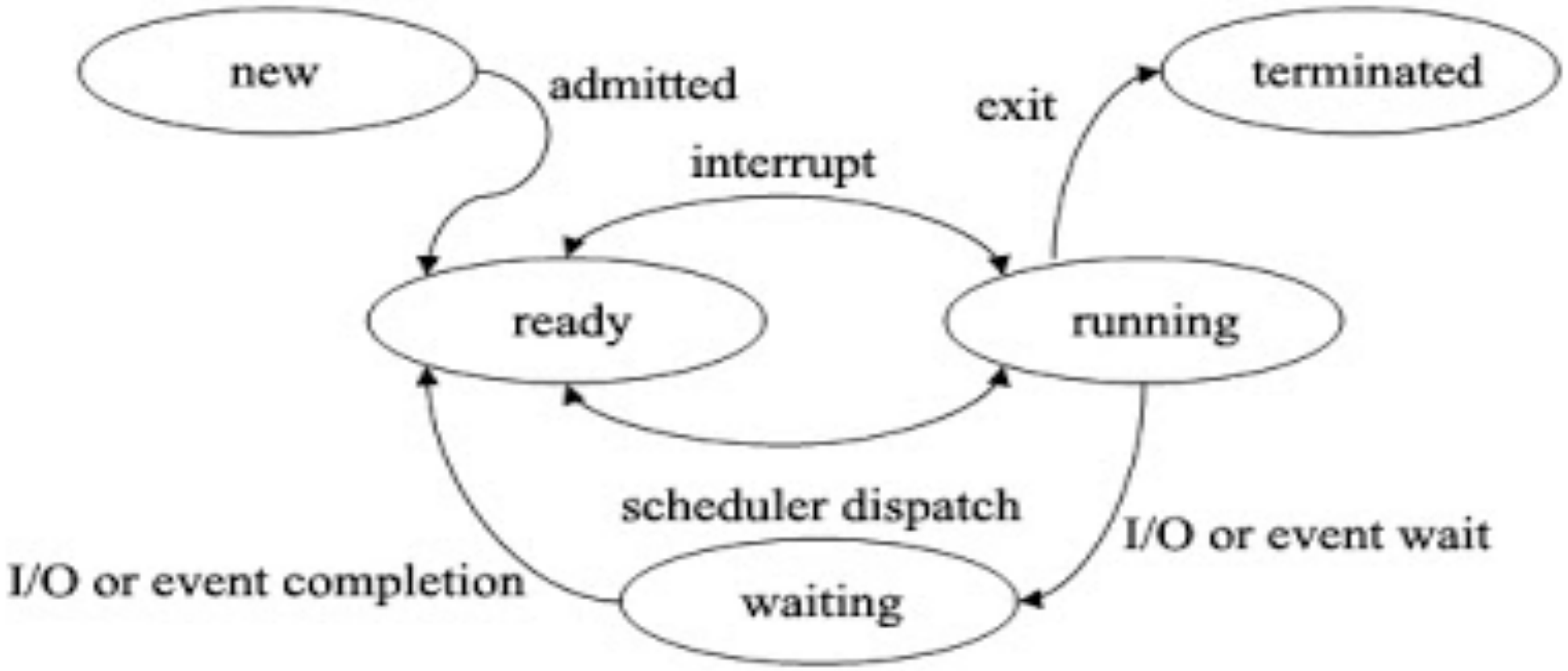
Definition of Process

- A program in execution
  - A process has its own address space consisting of:
    - Text region
      - Stores the code that the processor executes
    - Data region
      - Stores variables and dynamically allocated memory
    - Stack region
      - Stores instructions and local variables for active procedure calls

Process States: Life Cycle of a Process

- **Process State**
- As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:
    - **New.** The process is being created.
    - **Running.** Instructions are being executed.
    - **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
    - **Ready.** The process is waiting to be assigned to a processor
    - **Terminated.** The process has finished execution.

Process state transition diagram



Process Management

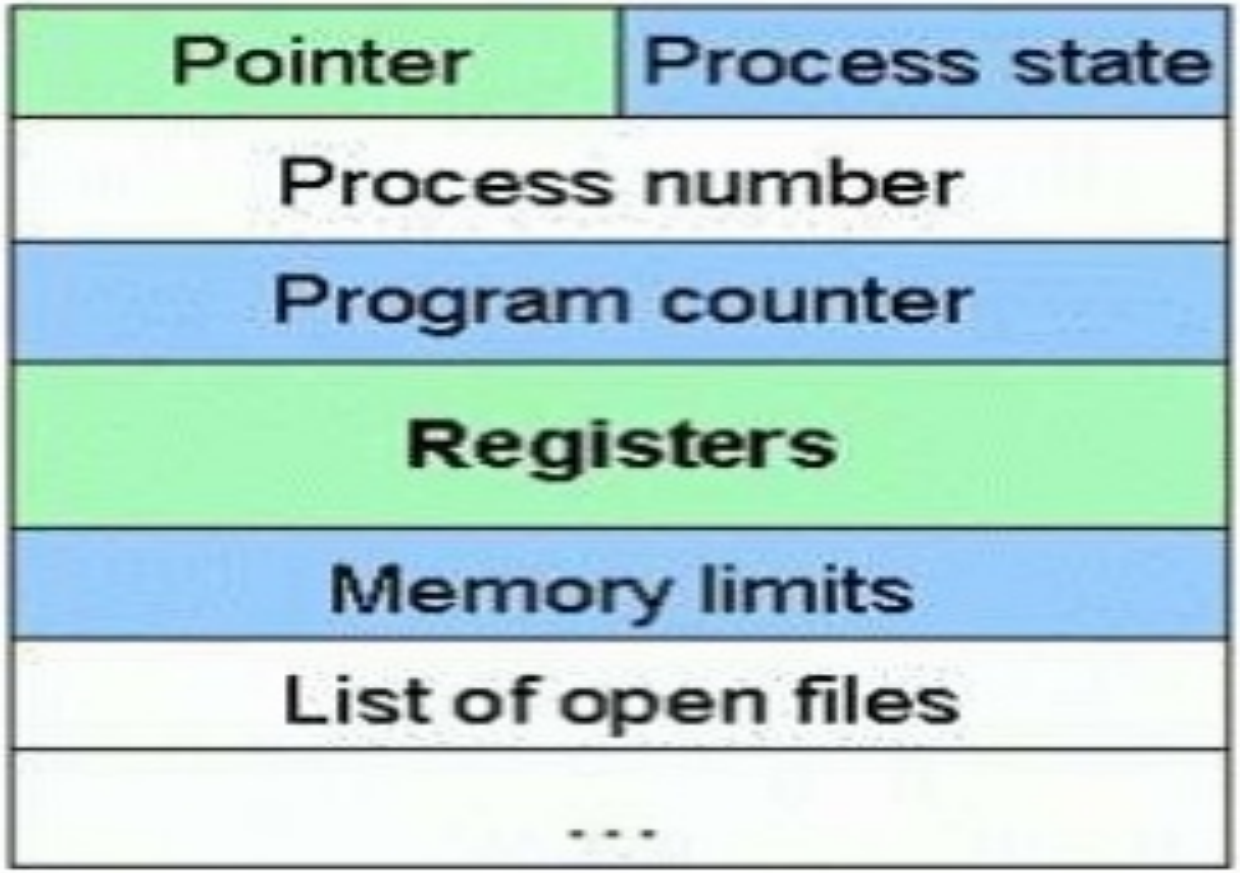
- Operating systems provide fundamental services to processes including:
  - Creating processes
  - Destroying processes
  - Suspending processes
  - Resuming processes
  - Changing a process's priority
  - Blocking processes
  - Waking up processes
  - Dispatching processes
  - Interprocess communication (IPC)

Process States and State Transitions

- Process states
  - The act of assigning a processor to the first process on the ready list is called dispatching
  - The OS may use an interval timer to allow a process to run for a specific time interval or quantum
  - Cooperative multitasking lets each process run to completion
- State Transitions
  - At this point, there are four possible state transitions
    - When a process is dispatched, it transitions from *ready* to *running*
    - When the quantum expires, it transitions from *running* to *ready*
    - When a process blocks, it transitions from *running* to *blocked*
    - When the event occurs, it transitions from *blocked* to *ready*

Process Control Blocks (PCBs)

- Each process is represented in the operating system by a **process control block (PCB)** also called a *task control block.* A PCB contains many pieces of information associated with a specific process, including these:
  - **Process state.** The state may be new, ready, running, waiting, halted, and so on.
  - **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
  - **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.



Block diagram of PCB

- • **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- • **Memory-management information.** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- • **Accounting information.** This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.
- • **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.
- In brief, the PCB simply serves as the repository for any information that may vary from process to process.
-

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

- As processes enter the system, they are put into a **job queue,** which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** A ready-queue header contains pointers to the first and final PCBs in the list.
- The list of processes waiting for a particular I/O device is called a device **queue.**
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

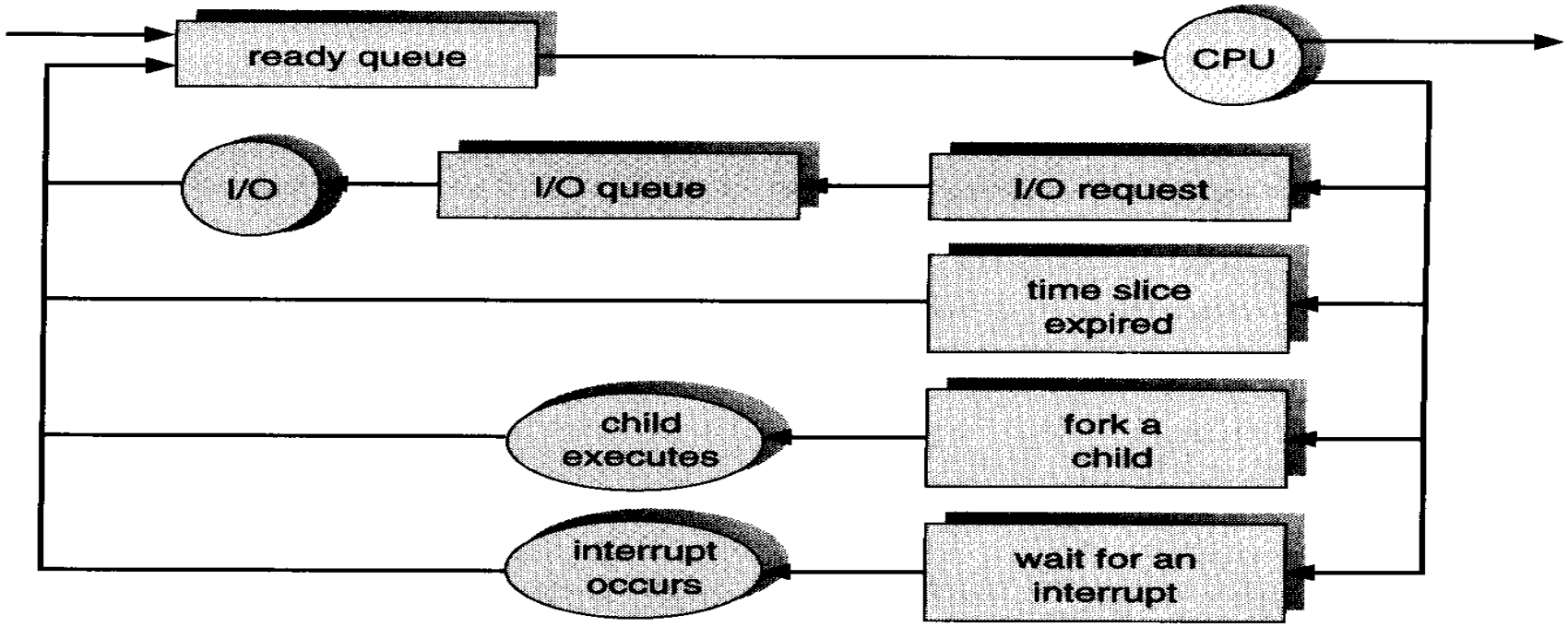


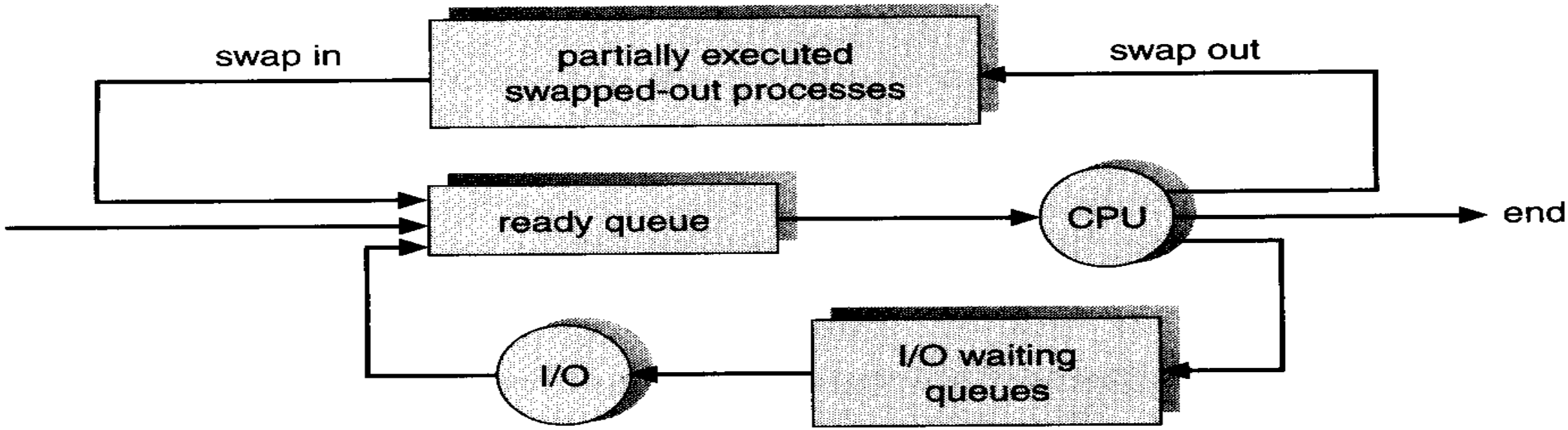
Fig: **Queuing diagram representation of process scheduling**

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler.**
- Types of schedulers:

- **i) Long term Scheduler** In batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming(**the number of processes in memory**).** It is important that the long-term scheduler make a careful selection.
- **ii) Short term Scheduler** The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them. Because of the short time between executions, the short-term scheduler must be fast.

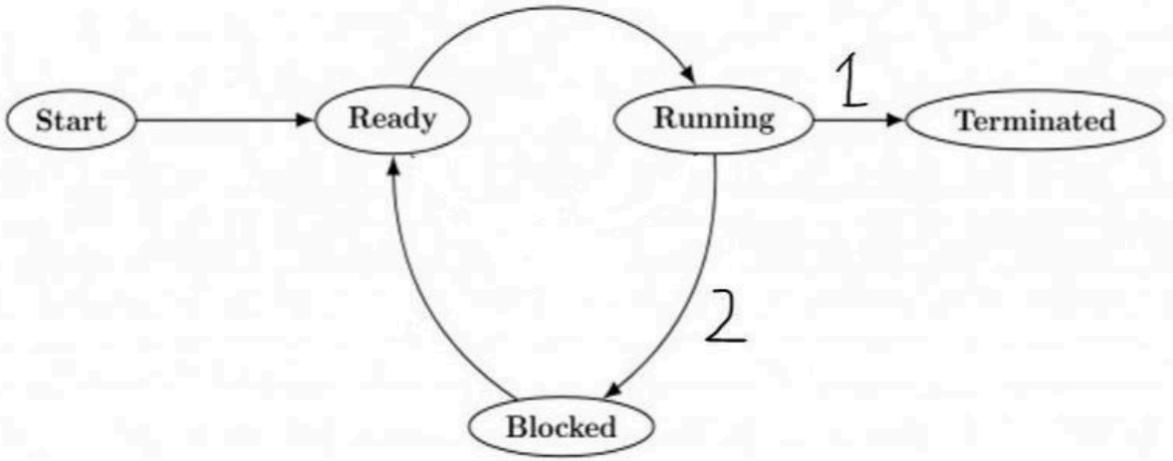
- **medium-term scheduler** The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory for I/O operation and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



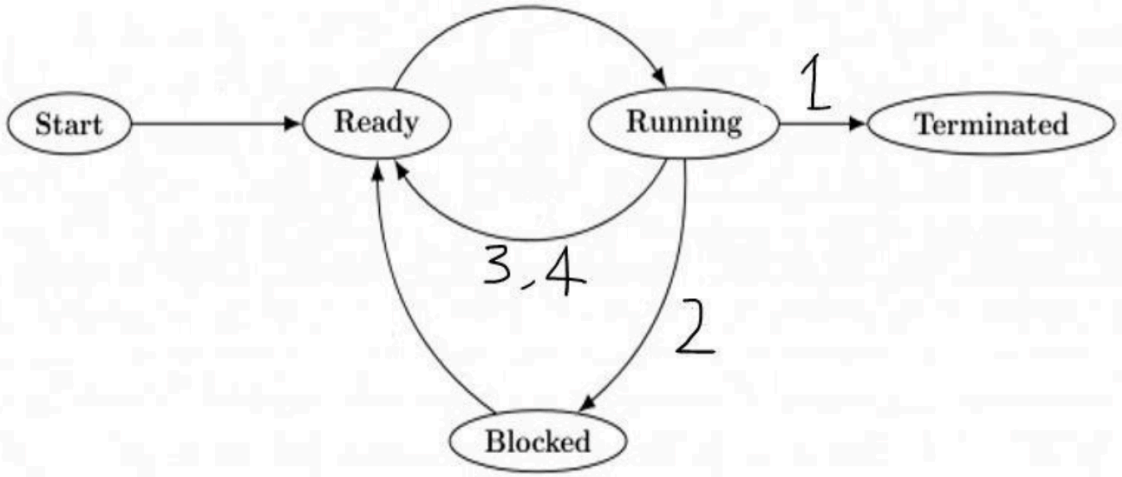
Scheduling Algorithms

- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

- These algorithms are either **non-preemptive or preemptive**.
- Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time
- preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.



Non - Preemptive Scheduling



Preemptive Scheduling

Scheduling Criteria

- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.
  Turn Around Time = Completion Time – Arrival Time

- **Waiting Time(W.T):** Time Difference between turn around time and burst time.
  Waiting Time = Turn Around Time – Burst Time
- **ResponseTime(R.T) :** The time at which a process got CPU first time - Arrival time

First-Come, First-Served (FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high

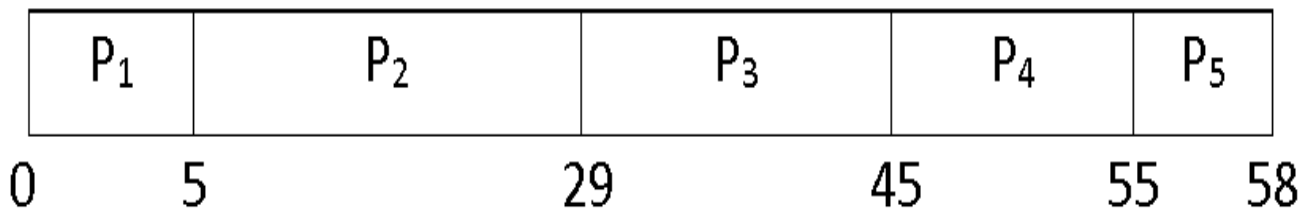
FCFS- Convoy Effect

If the smaller process have to wait more for the CPU because of Larger process then this effect is called **Convoy Effect,** it result into more average waiting time.

Solution, smaller process have to be executed before longer process, to achieve less average waiting time.

FCFS Non Preemptive Example

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3



Average Waiting Time

Waiting Time = Starting Time - Arrival Time

Waiting time of

P1 = 0

P2 = 5 - 0 = 5 ms

P3 = 29 - 0 = 29 ms

P4 = 45 - 0 = 45 ms

P5 = 55 - 0 = 55 ms

Average Waiting Time = Waiting Time of all Processes / Total Number of Process

Therefore, average waiting time = (0 + 5 + 29 + 45 + 55) / 5 = 26.8 ms

Average Turnaround Time

- *Turnaround Time = Waiting time in the ready queue + executing time + waiting time in waiting-queue for I/O or [Completion - arrival time]*

  Turnaround time of

  P1 = 0 + 5 + 0 = 5ms

  P2 = 5 + 24 + 0 = 29ms

  P3 = 29 + 16 + 0 = 45ms

  P4 = 45 + 10 + 0 = 55ms

  P5 = 55 + 3 + 0 = 58ms

  Total Turnaround Time = (5 + 29 + 45 + 55 + 58)ms = 192ms

  Average Turnaround Time = (Total Turnaround Time / Total Number of Process) = (192 / 5)ms = 38.4ms

Practice

Solve following using FCFS:

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

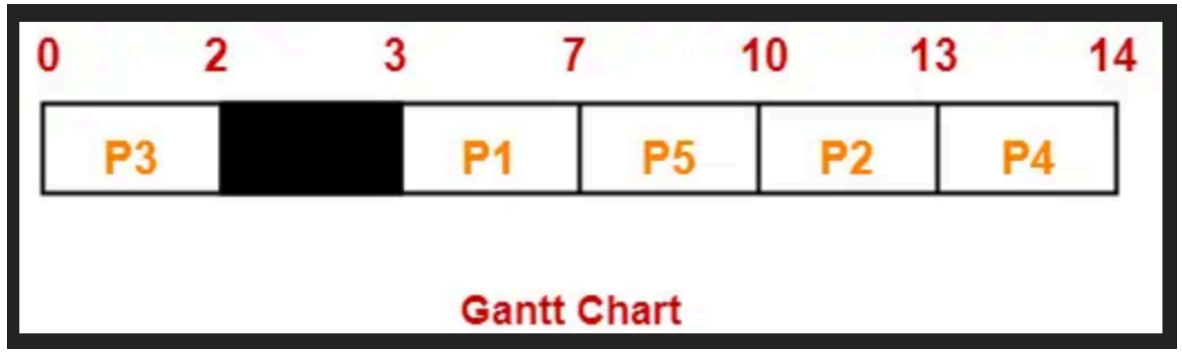
(A)

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

(B)

Solution

(A)



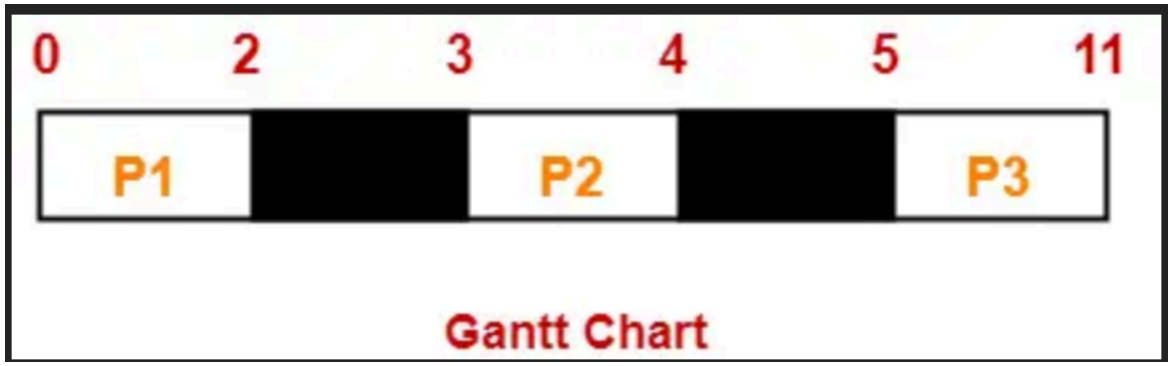
Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit

Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

Solution

(B)



Process Id	Exit time	Turn Around time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

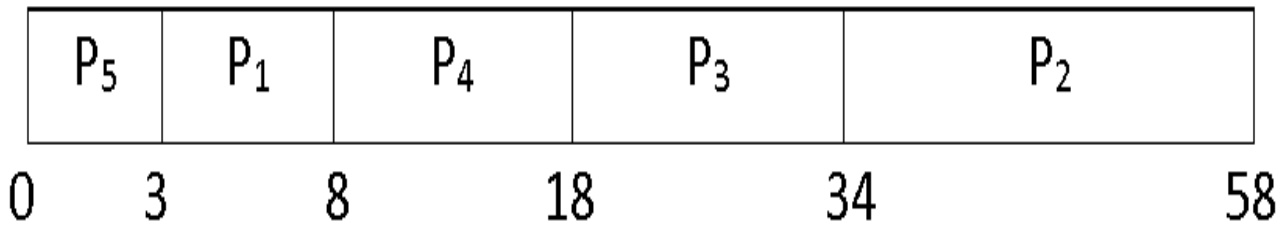
Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit
Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Shortest Job First (SJF)

- This is a non-preemptive, pre-emptive scheduling algorithm.
- pre-emptive SJF is also known as SRTF(Shortest remaining time first) algorithm
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Example of Non Preemptive SJF

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3



- **Average Waiting Time :** arrival time is common to all processes(i.e., zero).
  Waiting Time for
  P1 = 3 - 0 = 3ms
  P2 = 34 - 0 = 34ms
  P3 = 18 - 0 = 18ms
  P4 = 8 - 0 = 8ms
  P5 = 0ms
  Now, Average Waiting Time = (3 + 34 + 18 + 8 + 0) / 5 = 12.6ms

- **Average Turnaround Time**
- According to the SJF Gantt chart and the turnaround time formulae, Turnaround Time of
  P1 = 3 + 5 = 8ms
  P2 = 34 + 24 = 58ms
  P3 = 18 + 16 = 34ms
  P4 = 8 + 10 = 18ms
  P5 = 0 + 3 = 3ms
  Therefore, Average Turnaround Time = (8 + 58 + 34 + 18 + 3) / 5 = 24.2ms

Practice

Solve following using SJF:

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

(A)

Process ID	Arrival Time	Burst Time
P1	1	3
P2	2	4
P3	1	2
P4	4	4

(B)

Solution

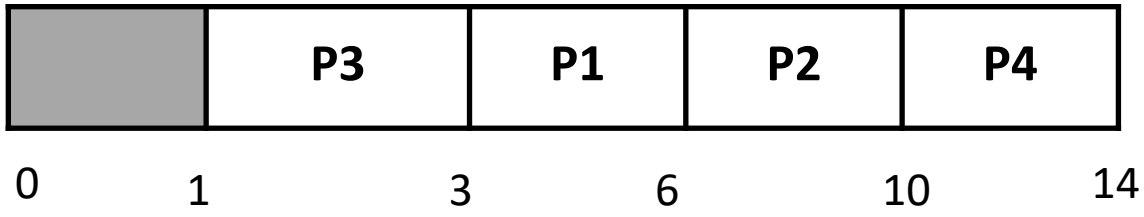
(A)

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Solution

(B)

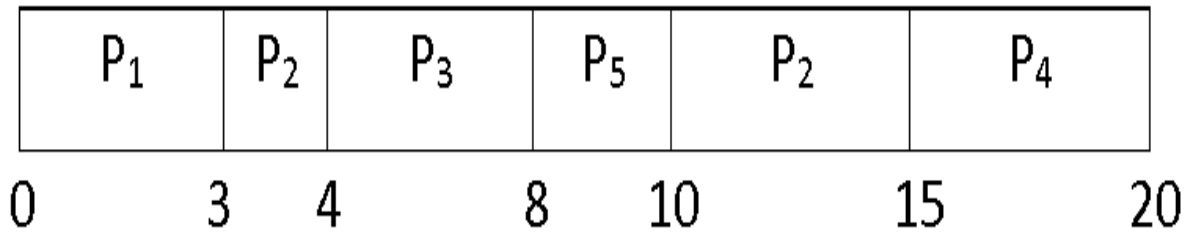


Avg. TAT=6.25

Avg. WT= 3

Example of Preemptive SJF/SRTF
Shortest Remaining Time First

Process	Burst Time(CPU)	Arrival Time(ms)
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8



- **Average Waiting Time**
- First of all, we have to find the waiting time for each process.
  Waiting Time of process
  P1 = 0ms
  P2 = (3 - 2) + (10 - 4) = 7ms
  P3 = (4 - 4) = 0ms
  P4 = (15 - 6) = 9ms
  P5 = (8 - 8) = 0ms
  Therefore, Average Waiting Time = (0 + 7 + 0 + 9 + 0) / 5 = 3.2ms

- **Average Turnaround Time**
- First of all, we have to find the turnaround time of each process.
  Turnaround Time of process
  P1 = (0 + 3) = 3ms
  P2 = (7 + 6) = 13ms
  P3 = (0 + 4) = 4ms
  P4 = (9 + 5) = 14ms
  P5 = (0 + 2) = 2ms
  Therefore, Average Turnaround Time = (3 + 13 + 4 + 14 + 2) / 5 = 7.2ms

Practice

Solve following using SRTF/SJF Preemption:

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	4
P4	4	1

(A)

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

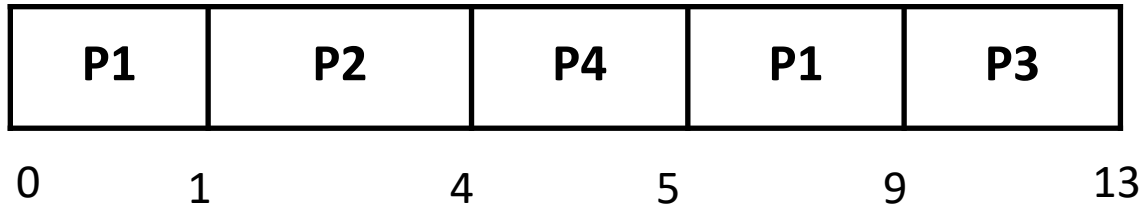
(B)

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

(C)

Solution

(A)



Avg. TAT= 6

Avg. WT= 2.75

(B)

Avg. TAT=7

Avg. WT=3.8

(C)

Avg. TAT=7.17

Avg. WT=4

Priority Scheduling

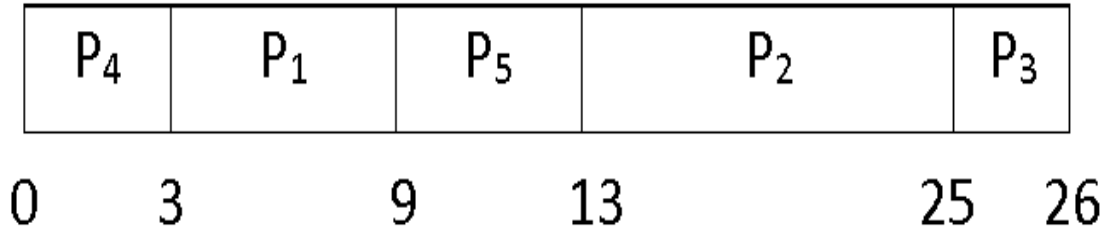
- Priority scheduling can be non-preemptive or preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement

Priority Scheduling Example

Here Priority Scheduling(Lower Number Higher Priority):

Process	CPU Burst Time	Priority
P_1	6	2
P_2	12	4
P_3	1	5
P_4	3	1
P_5	4	3

Gantt Chart



- **Average Waiting Time**
- First of all, we have to find out the waiting time of each process.
  Waiting Time of process
  P1 = 3ms
  P2 = 13ms
  P3 = 25ms
  P4 = 0ms
  P5 = 9ms
  Therefore, Average Waiting Time = (3 + 13 + 25 + 0 + 9) / 5 = 10ms

- **Average Turnaround Time**
- First finding Turnaround Time of each process.
  Turnaround Time of process
  P1 = (3 + 6) = 9ms
  P2 = (13 + 12) = 25ms
  P3 = (25 + 1) = 26ms
  P4 = (0 + 3) = 3ms
  P5 = (9 + 4) = 13ms
  Therefore, Average Turnaround Time = (9 + 25 + 26 + 3 + 13) / 5 = 15.2ms

Practice

Solve following using Priority Scheduling(Higher Number Higher Priority):

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

(A) Non Preemptive

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

(B)Preemptive

Solution

(A)

Gantt Chart

Avg. TAT= 8.2

Avg. WT= 5.2

(B)

Gantt Chart

Avg. TAT=7.6

Avg. WT=4.6

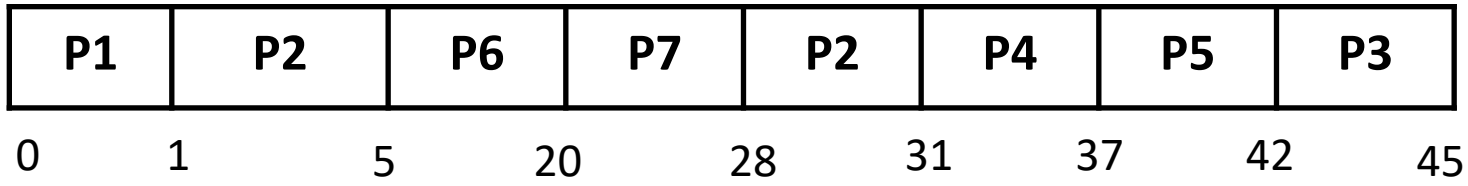
Practice

Solve following using Priority Scheduling in both the scenarios (A) Preemptive and (B) Non preemptive

Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

Solution

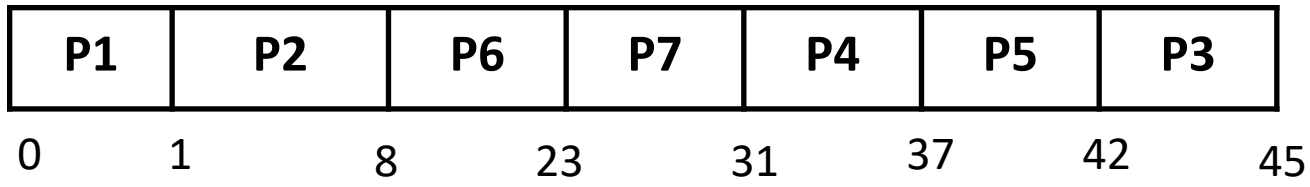
(A)



Avg. TAT= 24.71 UNITS

Avg. WT=18.28 UNITS

(B)



Avg. TAT=21 UNITS

Avg. WT= 16 UNITS

Round Robin (RR)

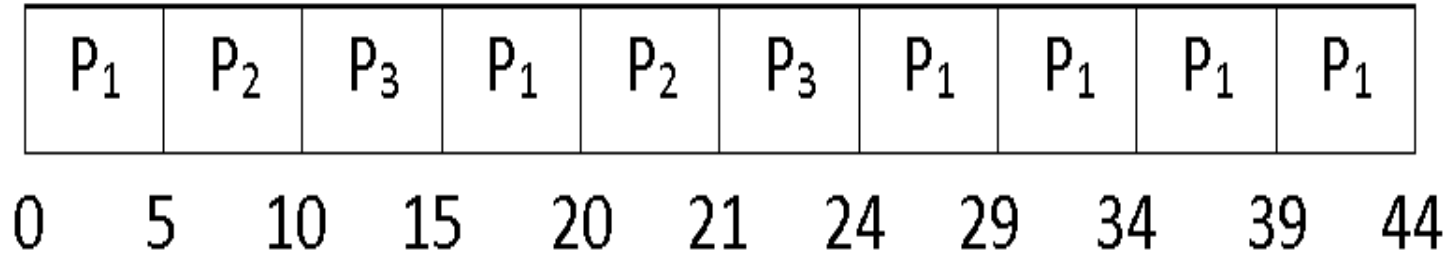
- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Round Robin (RR) Example

Time quantum is 5

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Gantt Chart



- **Average Waiting Time**
- For finding Average Waiting Time, we have to find out the waiting time of each process.
-

  Waiting Time of
  P1 = 0 + (15 - 5) + (24 - 20) = 14ms
  P2 = 5 + (20 - 10) = 15ms
  P3 = 10 + (21 - 15) = 16ms
  Therefore, Average Waiting Time = (14 + 15 + 16) / 3 = 15ms

- **Average Turnaround Time**

- Same concept for finding the Turnaround Time.
  Turnaround Time of
-
  P1 = 14 + 30 = 44ms
  P2 = 15 + 6 = 21ms
  P3 = 16 + 8 = 24ms
-
  Therefore, Average Turnaround Time = (44 + 21 + 24) / 3 = 29.66ms

Practice

Solve following using Round Robin-

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

(A)Time Quantum=2

Process Id	Arrival time	Burst time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

(B)Time Quantum=2

Process Id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

(C)Time Quantum=3

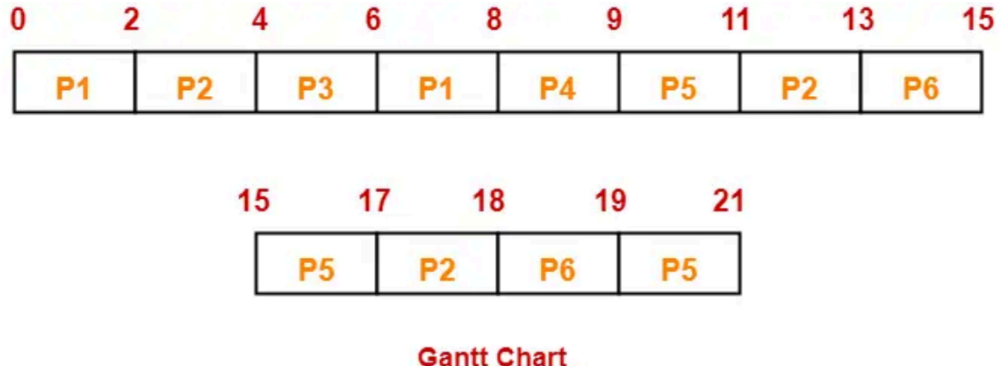
Solution

(A)

Avg. TAT= 8.6 units

Avg. WT= 5.8 units

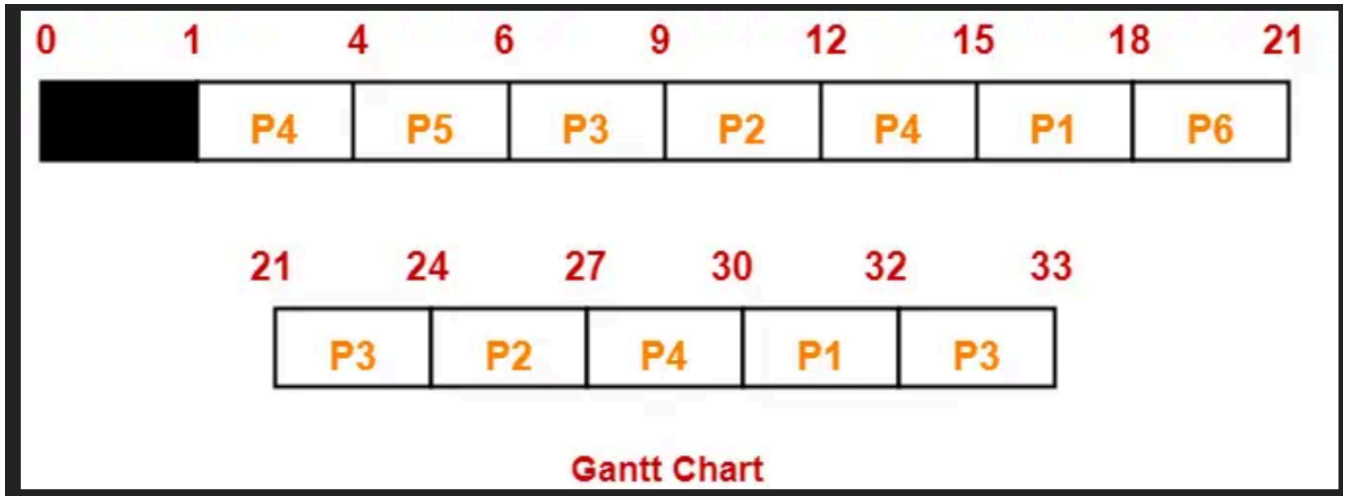
(B)



Avg. TAT=10.84 units

Avg. WT=7.33 units

(C)



Avg. WT=21.33 units

Avg. TAT=16 units

Process Synchronization

➢It means sharing system resources by processes in a such a way that, concurrent access to shared data is handled and thus allow minimum chance of inconsistency (inconsistent data).

➢ Maintaining data consistency demands mechanisms by ensuring synchronized execution among cooperating/sharing/concurrently executing processes.

➢Process Synchronization was introduced to handle problems that used to get arose while multiple process executes concurrently.

➢Some of the problems are discussed below.

```
P()
{
   read(i)
    i=i+1
    write(i)
}
```

Race condition is a situation in which the output of a process depends on the execution sequence of process.

Critical Section Problem

➢When two or more processes access shared data,

➢Typically, a process that reads data from a shared queue cannot read it while the data is currently being written or its value being changed.

➢A process is considered that it cannot be interrupted while performing a critical function such as updating data, it is prevented from being interrupted by the operating system till it has completed the update.

➢ During this time, the process is said to be in its **critical section**. Once the process has written the data, it can then be interrupted and other processes can also run.

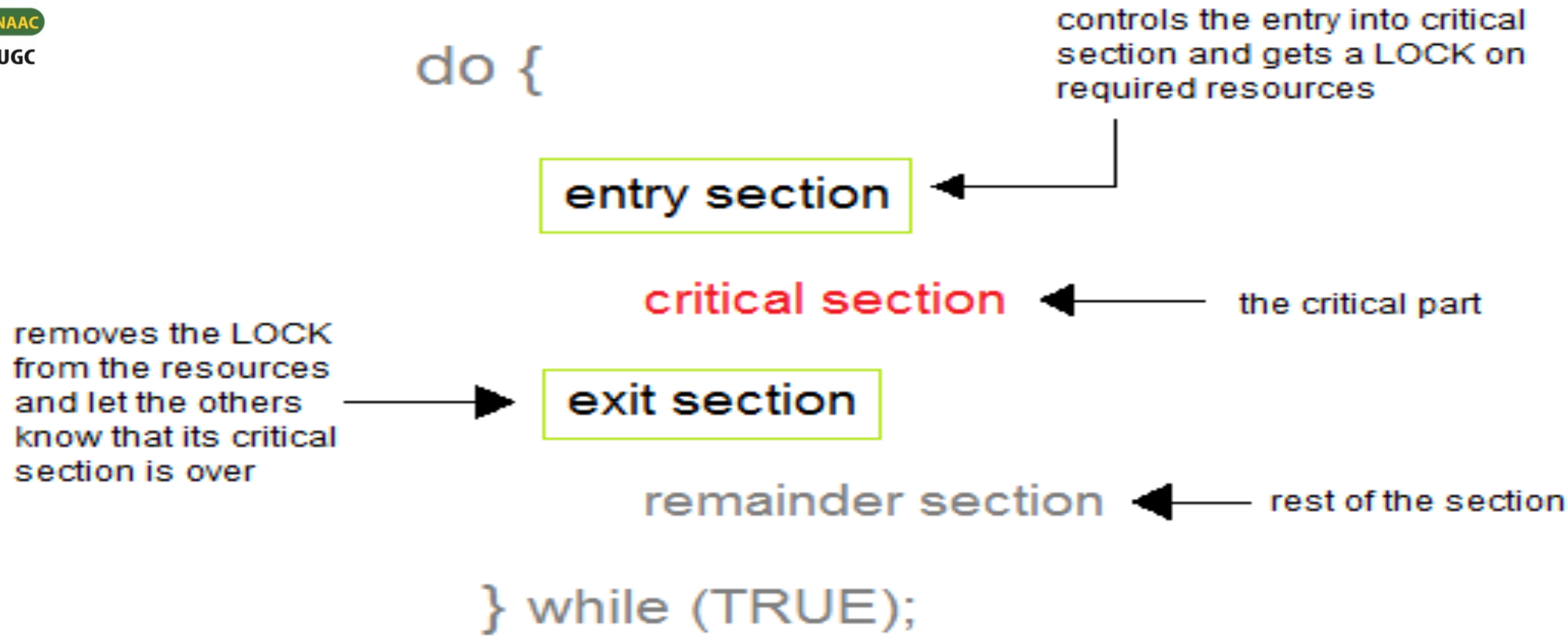
Consider a system consisting of n processes {P0,P1,.....Pn-1}.

➢*Each process has a segment of code, called a* ***critical section,*** *in which the process may be changing common variables, updating a table etc.*

➢*The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.,*

➢*no two processes are executing in their critical sections at the same time.*

➢*Each* process *must request permission to enter its critical section in entry section, may follow critical section with exit section.*



The part of the program (process) that is changing and accessing the shared data is called its critical section

A solution to the critical section problem must satisfy the following three conditions :

1)Mutual Exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2)Progress: If no process is executing in its critical section, and if one or more process want to execute their critical section then any one of these process must be allowed to get into its critical section.

3) Bounded Waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section Problem: Algorithm 1

Use the common variable *turn* to determine which process may enter the critical section. If *turn* = *i* then process P_i is allowed to execute it's critical section.

```
repeat
    while turn != i do no-op;
        critical section
    turn := j;
        remainder section
until false;
```

P_0	P_1

while (1)
{
while (turn! = 0);
Critical Section
turn = 1;
Remainder section
}
	while (1)
	{
	while (turn! = 1);
	Critical Section
	turn = 0;
	Remainder Section
	}

This does not satisfy the **progress requirement** because there is strict alternation between the processes.

Critical Section Problem: Algorithm 2

Algorithm 1 does not remember the *state* of each process. Instead of a single integer variable, try an array of boolean values:

var *flag*: **array** [0..1] **of** *boolean*;

The array elements are initialized to *false*

If *flag*[*i*] is *true* then process P_i is executing its critical section.

```
repeat
        flag[i] := true;
        while flag[j] do no-op;
            critical section
        flag[i] := false;
            remainder section
until false;
```

P_0	P_1

while (1)
{
flag [0] = T;
while (flag [1]);
Critical Section
flag [0] = F;
Remainder section
}
	while (1)
	{
	flag [1] = T;
	while (flag [0]);
	Critical Section
	flag [1] = F;
	Remainder Section
	}

This algorithm can fail the **progress requirement** if both processes set their flags to *true* and then both execute the **while** loop.

Swapping the **while** and the assignment *flag*[*i*] := *true* doesn't help either -- in this case we can't guarantee mutual exclusion

Peterson's algorithm (or **Peterson's solution**)

It allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

Assumption

•assume that a variable (memory location) can only have one value; never ``between values" .

•if processes A and B write a value to the same memory location at the ``same time," either the value from A or the value from B will be written rather than some scrambling of bits.

Peterson's Algorithm

Shared variables are created and initialized before either process starts. The shared variables flag[0] and flag[1] are initialized to FALSE because neither process is yet interested in the critical section. The shared variable turn is set to either 0 or 1 randomly (or it can always be set to say 0).

var flag: **array** [0 or 1] **of** boolean;
turn: 0 or1; //*flag[k] means that process[k] is interested in the critical section*

flag[0] := FALSE;
flag[1] := FALSE;
turn := random(0..1)

After initialization, each process, which is called **process *i*** in the code (the other process is process *j*), runs the following code:

repeat
flag[i] := TRUE;
turn := j;
while (flag[j] **and** turn=j) **do** no-op;
CRITICAL SECTION
flag[i] := FALSE;
REMAINDER SECTION
until FALSE;

Information common to both processes:
turn = 0
flag[0] = FALSE
flag[1] = FALSE

P_0	P_1

while (1)
{
flag [0] = T;
turn = 1;
while (turn = = 1 && flag [1] = = T);
Critical Section
flag [0] = F;
Remainder section
}
	while (1)
	{
	flag [1] = T;
	turn = 0;
	while (turn = = 0 && flag [0] = =T);
	Critical Section
	flag [1] = F;
	Remainder Section
	}

EXAMPLE 1	
Process 0	**Process 1**
i = 0, j = 1	i = 1, j = 0
flag[0] := TRUE turn := 1 check (flag[1] = TRUE and turn = 1) - *Condition is false because flag[1] = FALSE* - *Since condition is false, no waiting in while loop* - *Enter the critical section* - *Process 0 happens to lose the processor*	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Since condition is true, it keeps busy waiting until it loses the processor

- Process 0 resumes and continues until it finishes in the critical section *- Leave critical section* flag[0] := FALSE *- Start executing the remainder (anything else a process does besides using the critical section)* *- Process 0 happens to lose the processor*	
	check (flag[0] = TRUE and turn = 0) *- This condition fails because flag[0] = FALSE* *- No more busy waiting* *- Enter the critical section*

Synchronization Hardware

➢The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

➢We could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, **this solution is not feasible in a multiprocessor environment.**

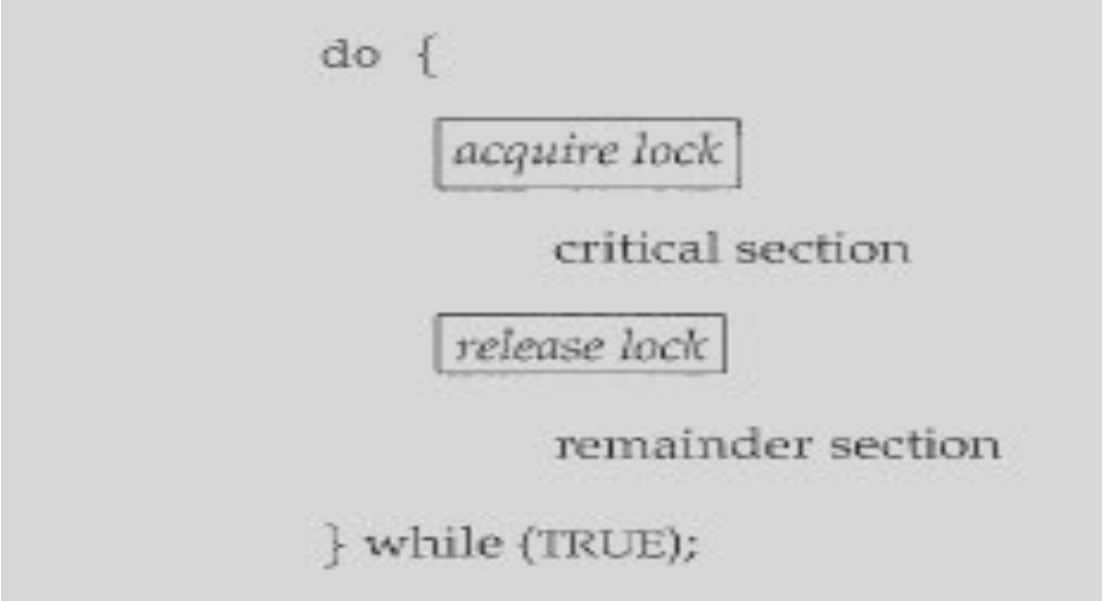
➢Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

➢This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks(*mutual exclusion object*)

➢ A strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a **LOCK** is acquired over the critical resources modified and used inside critical section, and in the exit section **that LOCK** is released.

➢As the resource is **locked** while a process executes its critical section hence no other process can access it. The ***acquire()*** function acquires the lock and the ***release()*** function releases the lock.



```
acquire() {
        while(!available)
                ; /* busy wait */
        available=false;;
}


release() {
        available=true;
}
```

Definition of acquire and release function

Semaphores

➢In 1965, Dijkstra proposed a technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.

➢This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait()** and **signal()** designated by P() and V() respectively.

➢The classical definition of wait and signal are :

Wait()

➢It also called ***P ("Proberen" meaning to test)***

➢ is called when a process wants access to a resource.

➢ This would be equivalent to the arriving customer trying to get an open table.

➢If there is an open table, or the semaphore is greater than zero, then he can take that resource and sit at the table.

➢*decrement the value of its argument S as soon as it would become non-negative.*

Signal()

➢It also called ***V (for Dutch "Verhogen" meaning to increment)***

➢ is called when a process is done using a resource, or when the guest is finished with his meal.

➢The following is an implementation of this counting semaphore (where the value can be greater than 1)

➢*increment the value of its argument, S as an individual operation.*

The two most common kinds of semaphores are:

i) Counting semaphores : represent multiple resources.

ii) Binary semaphores : represents two binary states ***(generally 0 or 1; locked or unlocked).***

```
do
{
    wait(s);
    // critical section
    signal(s);
    // remainder section
} while(1);
```

(Fig-1)

The definition of wait () is as follows:

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

The definition of signal () is as follows:

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

(Fig-2)

For example

- we have semaphore s, and two processes, P1 and P2 that want to enter their critical sections at the same time.

- P1 first calls wait(s). The value of s is decremented to 0 and P1 enters its critical section. While P1 is in its critical section, P2 calls wait(s), but because the value of s is zero, it must wait until P1 finishes its critical section and executes signal(s).

- When P1 calls signal, the value of s is incremented to 1, and P2 can then proceed to execute in its critical section (after decrementing the semaphore again).

- Mutual exclusion is achieved because only one process can be in its critical section at any time.

Classical Problem of Synchronization

we present a number of synchronization problems as examples

1)Bounded Buffer Problem/Producers and Consumers problem

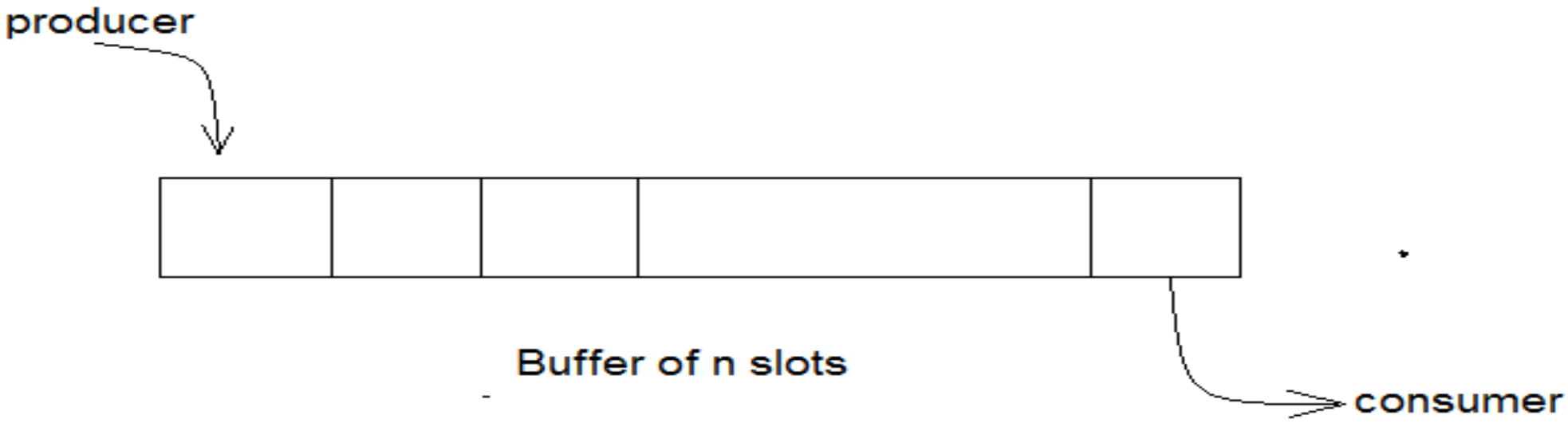
2) The Dining Philosophers Problem

3)The Reader Writer Problem

In This we use semaphores for synchronization.

1)Bounded Buffer Problem/Producers and Consumers problem

➢The bounded buffer/ producers and consumers assumes that there is a fixed buffer size i.e., a finite numbers of slots are available.



Statement

➢A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the common fixed-size (bounded) buffer.

➢To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

➢As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as **bounded buffer problem**.

➢Sleep It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

➢Wakeup It is a system call that wakes up the process.

Trouble arises when :

The producer wants to put a new data in the buffer, but buffer is already full.

Solution: Producer goes to sleep and to be awakened when the consumer has removed data.

The consumer wants to remove data the buffer but buffer is already empty.

Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
int itemCount = 0;

procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

Structure of the producer process

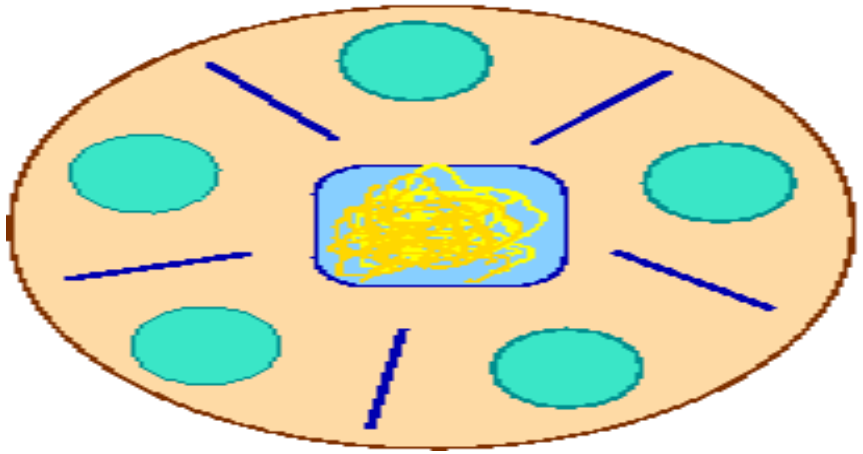
```
do {
//produce an item in nextp
wait(empty);
wait(mutex);
//add nextp to buffer
signal(mutex);
signal(full);
} while (TRUE);
```

Structure of the consumer process

```
do {
wait (full);
wait (mutex) ;
//remove an item from buffer to nextc
signal(mutex);
signal(empty);
//consume the item in nextc
} while (TRUE);
```

2).The Dining Philosophers Problem:

The *Dining Philosophers* problems is a classic synchronization problem (E. W. Dijkstra), introducing semaphores as a conceptual synchronization mechanism.



Statement:

➢There is a dining room containing a circular table with five chairs.

➢ At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti.

➢Near the room are five philosophers who spend most of their time **thinking,**

➢ A philosopher needs both their right and left chopstick to eat.

➢A hungry philosopher may only eat if there are both chopsticks available.

➢Otherwise a philosopher puts down their chopstick and begin thinking again.

➢A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick.

➢A **Chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.**

Thus, each philosopher is represented by the following pseudo code:


```
process P[i]
while true do
{
        THINK;

        WAIT(CHOPSTICK[i]);
        WAIT(CHOPSTICK[(i+1 )mod 5]);  //pickup

        EAT;
        SIGNAL(CHOPSTICK[i]);
    SIGNAL(CHOPSTICK[(i+1) mod 5]); //putdown
 }
```

➢A philosopher may THINK indefinitely.

➢Every philosopher who EATS will eventually finish.

➢Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or non deterministically, but these are atomic actions

➢two philosophers cannot use a single CHOPSTICK at the same time.

Difficulty with the solution

➢If all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows:

➢There should be at most four philosophers on the table.

➢An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.

➢A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Readers-Writers problem

- The readers-writers problem relates to an object such as a file that is shared between multiple processes.
- Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.
- The readers-writers problem is used to manage synchronization so that there are no problems with the object data.
- For example - If two readers access the object at the same time there is no problem.
- However if two writers or a reader and writer access the object at the same time, there may be problems.

- To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it.
- However, multiple readers can access the object at the same time.
- This can be implemented using semaphores.
- The codes for the reader and writer process in the reader-writer problem are given as follows −

Reader Process

- Int rc = 0 (rc= Read Count)
- Semaphore Mutex and db initially are 1
- wait (mutex);
- rc ++;
- if (rc == 1) then wait (db);
- signal(mutex);
- . READ THE OBJECT .(Data Base)
- wait(mutex);
- rc --;
- if (rc == 0) then signal (db);
- signal(mutex);

Writer Code

wait(db); .

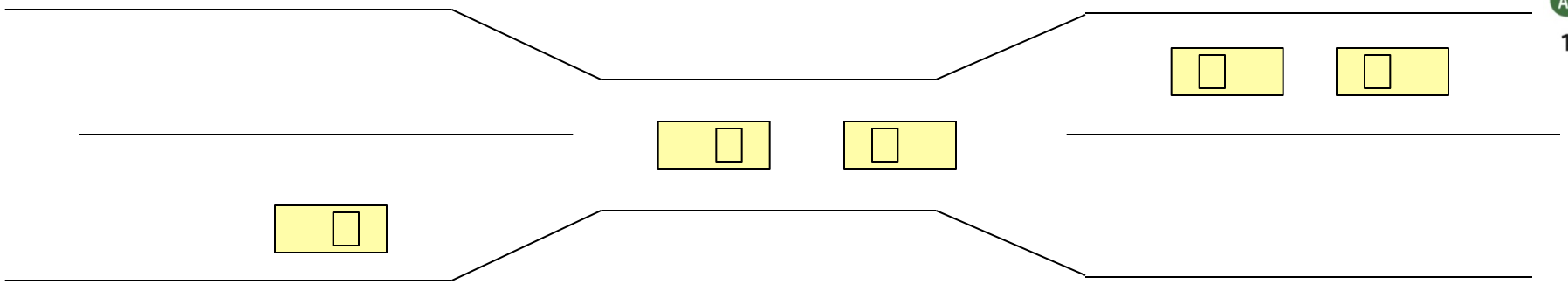
. WRITE INTO THE OBJECT .(Data Base)

signal(db);

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- A process requests resources; and if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This situation is called a **deadlock.**

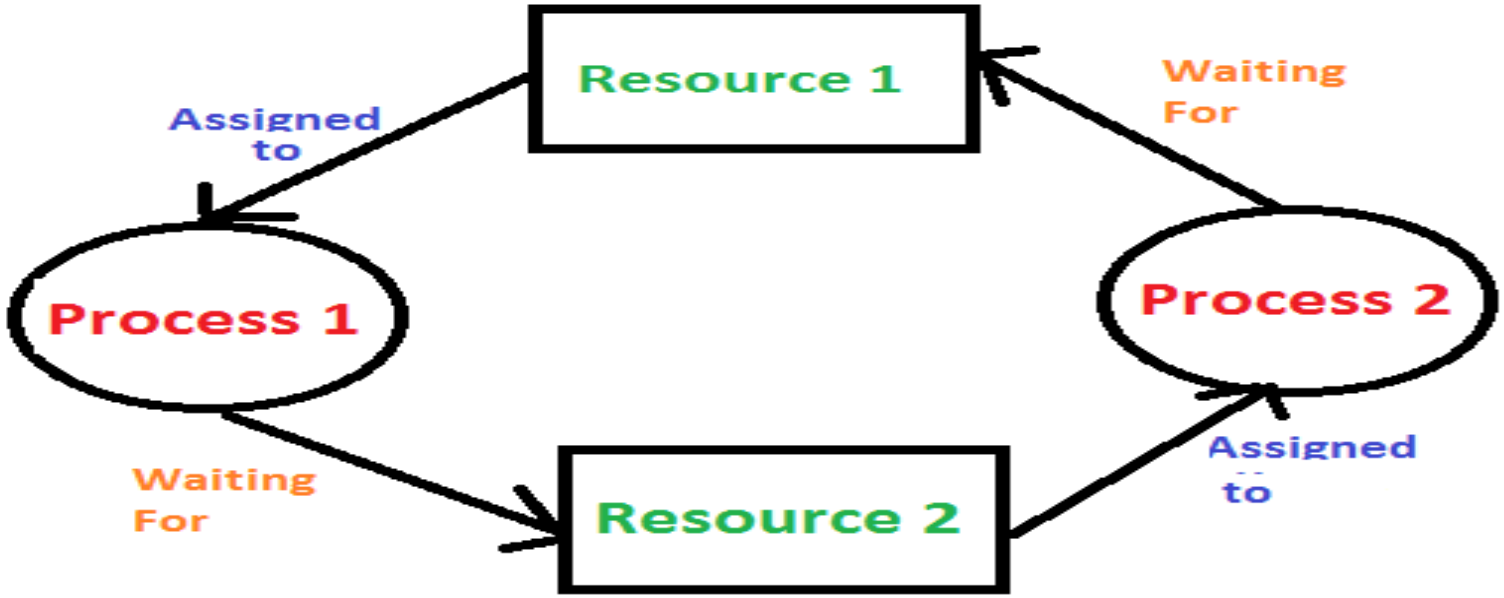
Bridge Crossing Example



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

Here is the simplest example

Process 1 requests resource 1 and receive it.
Process 2 requests resource 2 and receive it.
Process 1 requests resource 2 and is queued up, pending the release of resource 2 by process 2.
Process 2 requests resource 1 and is queued up, pending the release of Resource 1 by process 1.



System Model

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

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.

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

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

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

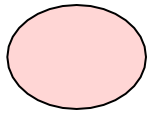
Resource-Allocation Graph

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

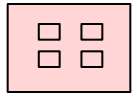
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

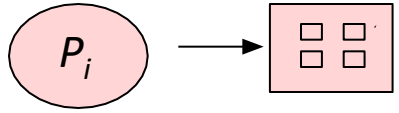
- Process



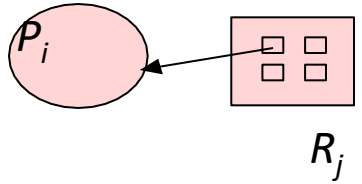
- Resource Type with 4 instances



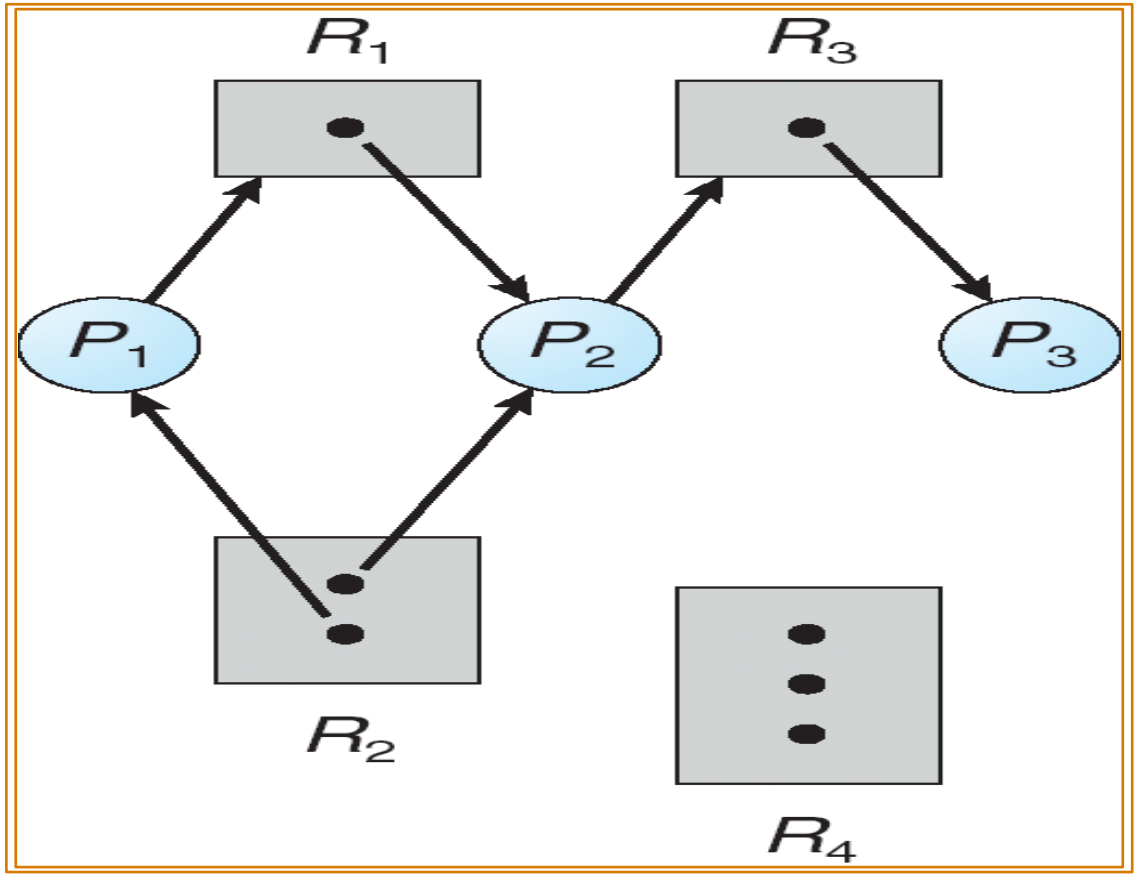
- P_i requests instance of R_j



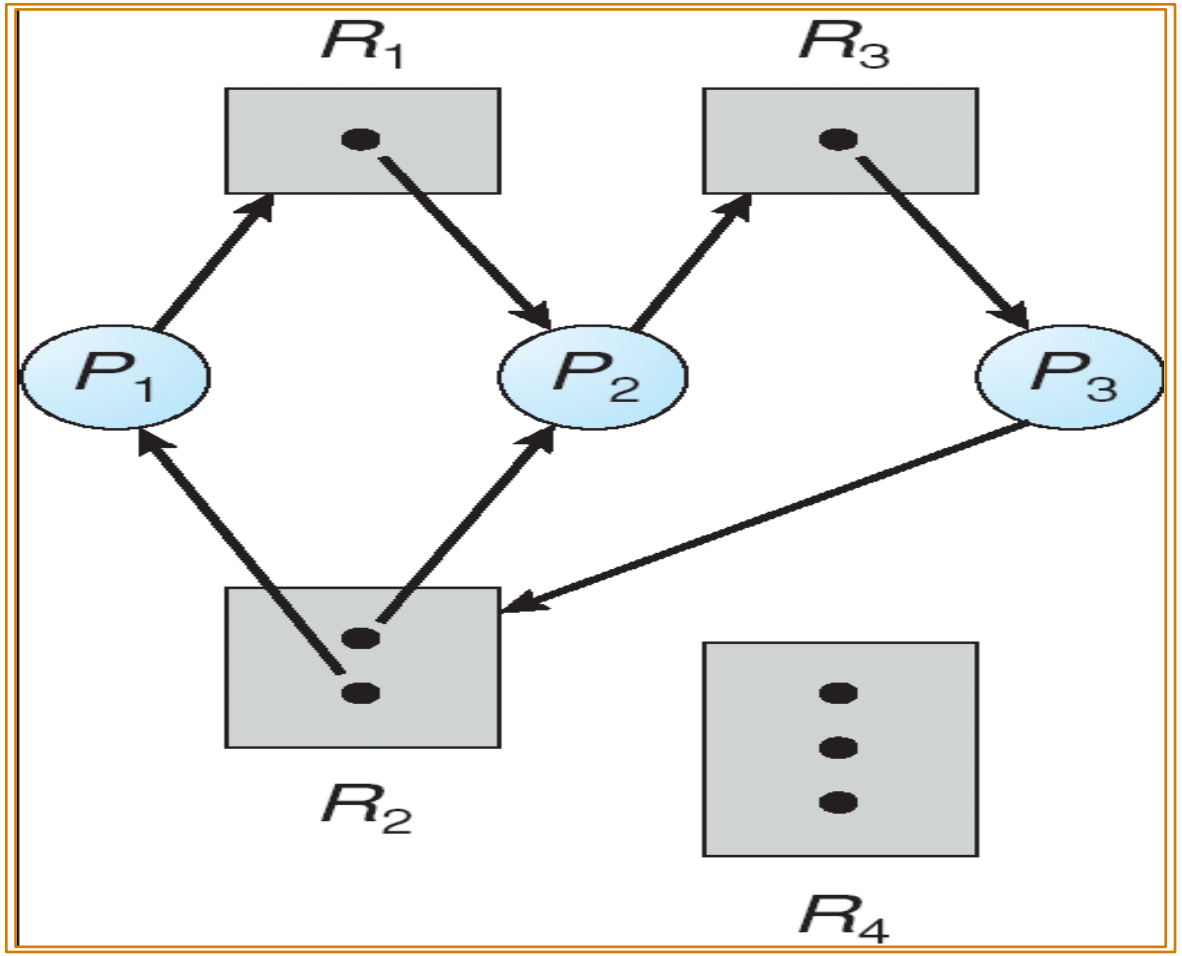
- P_i is holding an instance of R_j



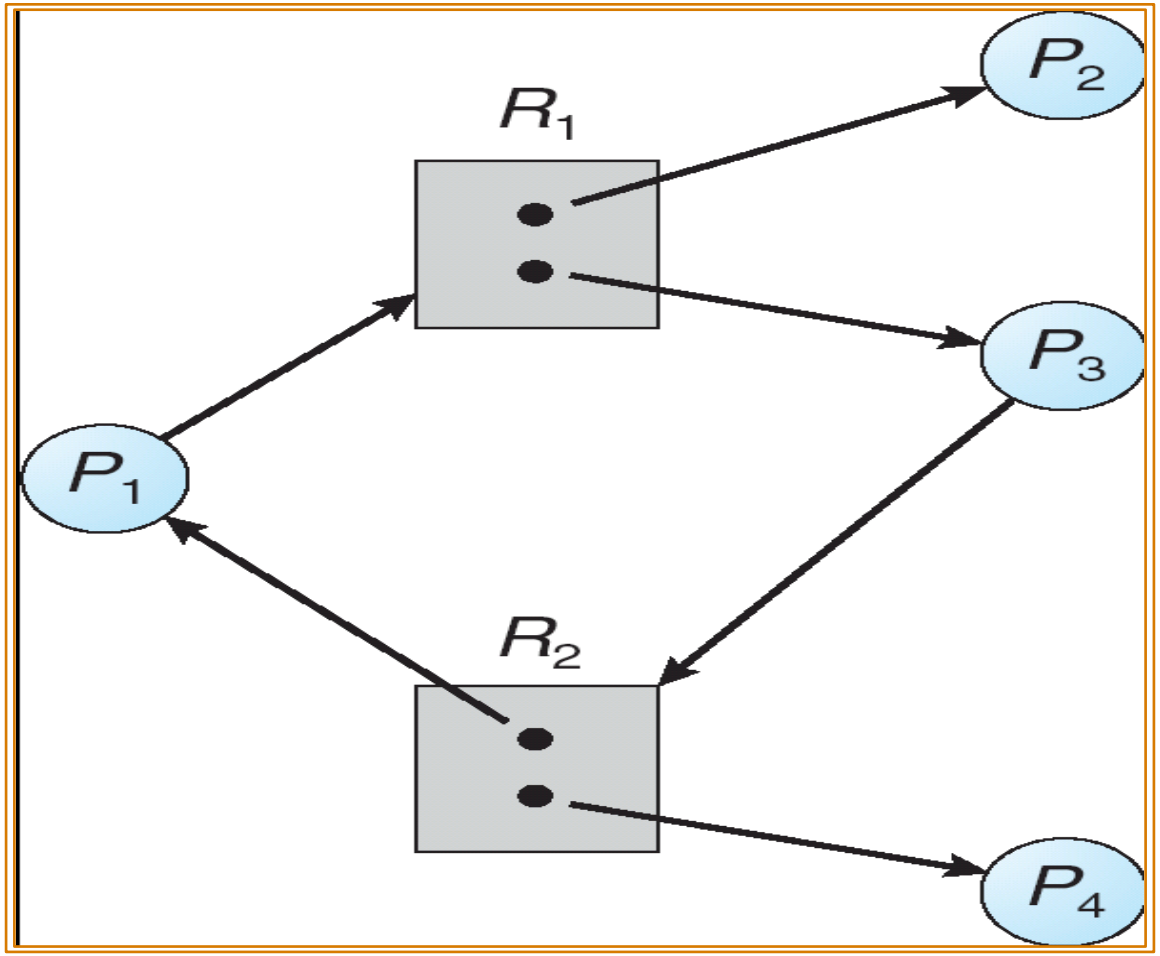
Example of a Resource Allocation Graph



Will there be a deadlock here?



Resource Allocation Graph With A Cycle But No Deadloc



Basic Facts

- If graph contains no cycles means no deadlock.

- If graph contains a cycle means
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

The Methods (continued

- Deadlock Prevention - Do not let the deadlock occur.
- Deadlock Avoidance -Avoid the Deadlock
- Deadlock Detection -Let the deadlock occur in the system and then attempt to recover the system from deadlock.

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 the deadlock.

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

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

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

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

Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

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

Deadlock Detection

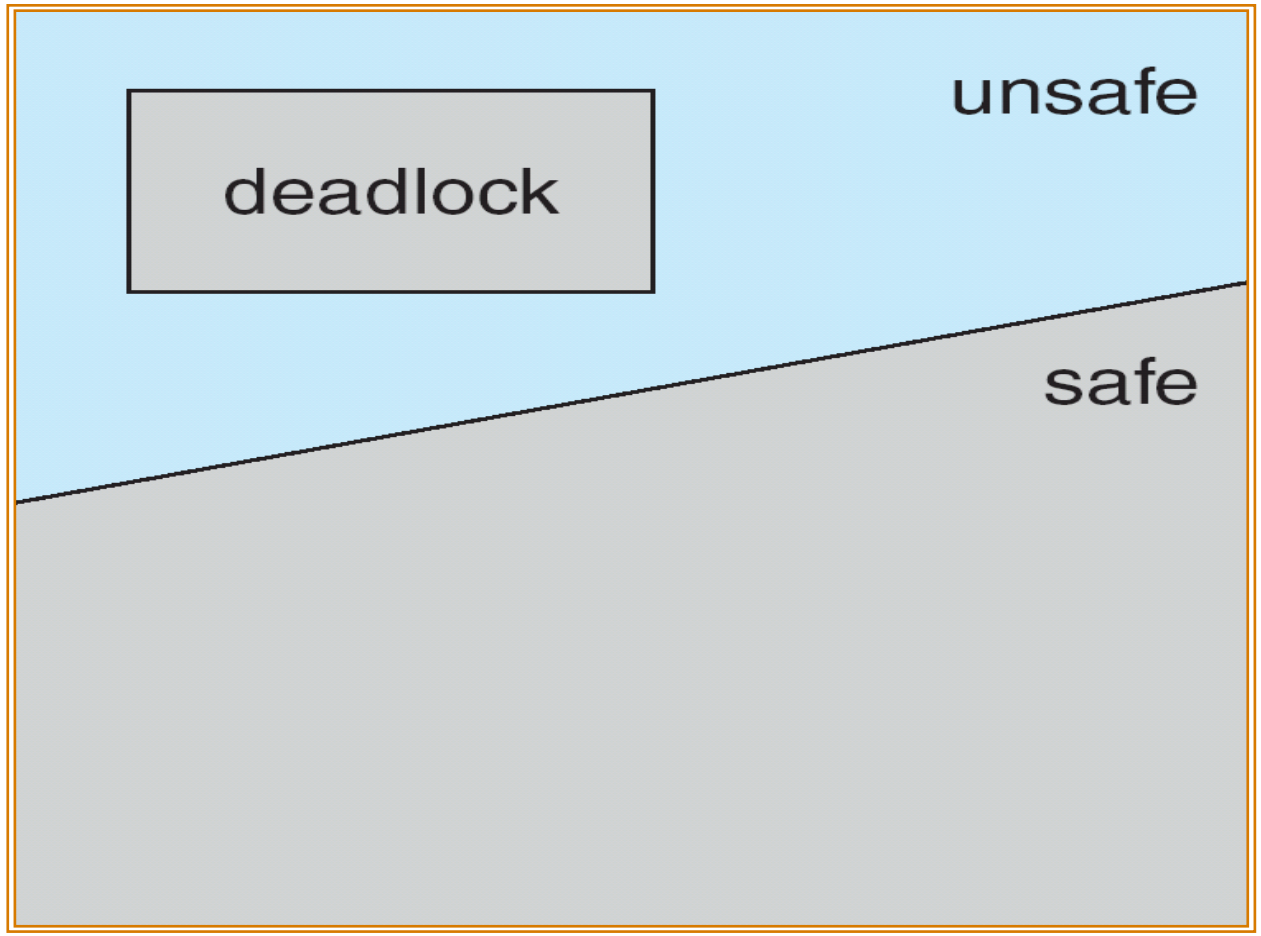
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Safe State

- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.

- A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with $j < i$.

Safe, Unsafe , Deadlock State



- If a system is in safe state, there is no deadlock.
- If the system is deadlocked, it is in an unsafe state.
- If a system is in unsafe state, there is a possibility for a deadlock.
- Avoidance: making sure the system will not enter an unsafe state.

Resource-Allocation Graph Algorithm

- *Claim edge* P_i - -> R_j indicated that process P_i may request resource R_j; represented by a dashed line.

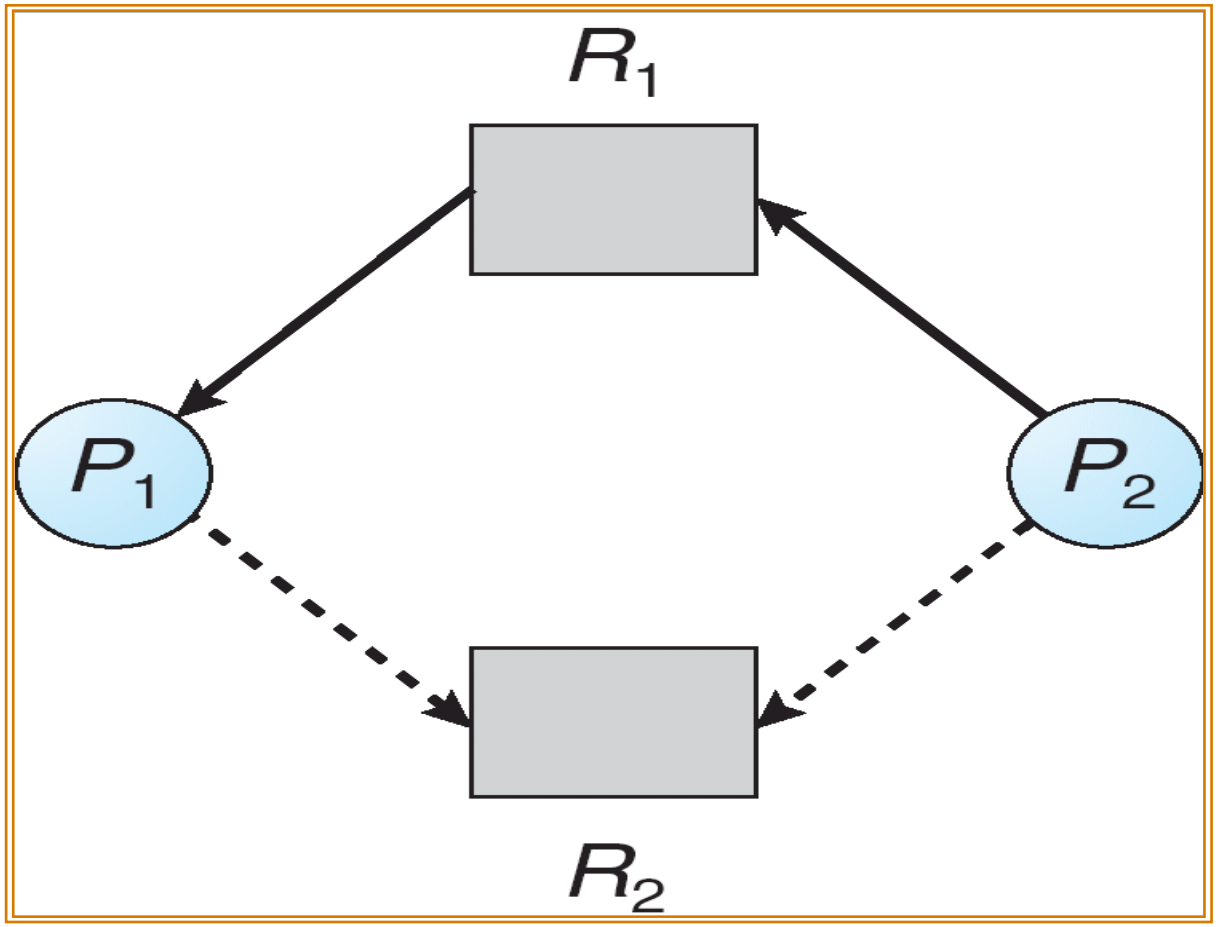
- Claim edge converts to request edge when a process requests a resource.

- When a resource is released by a process, assignment edge reconverts to a claim edge.

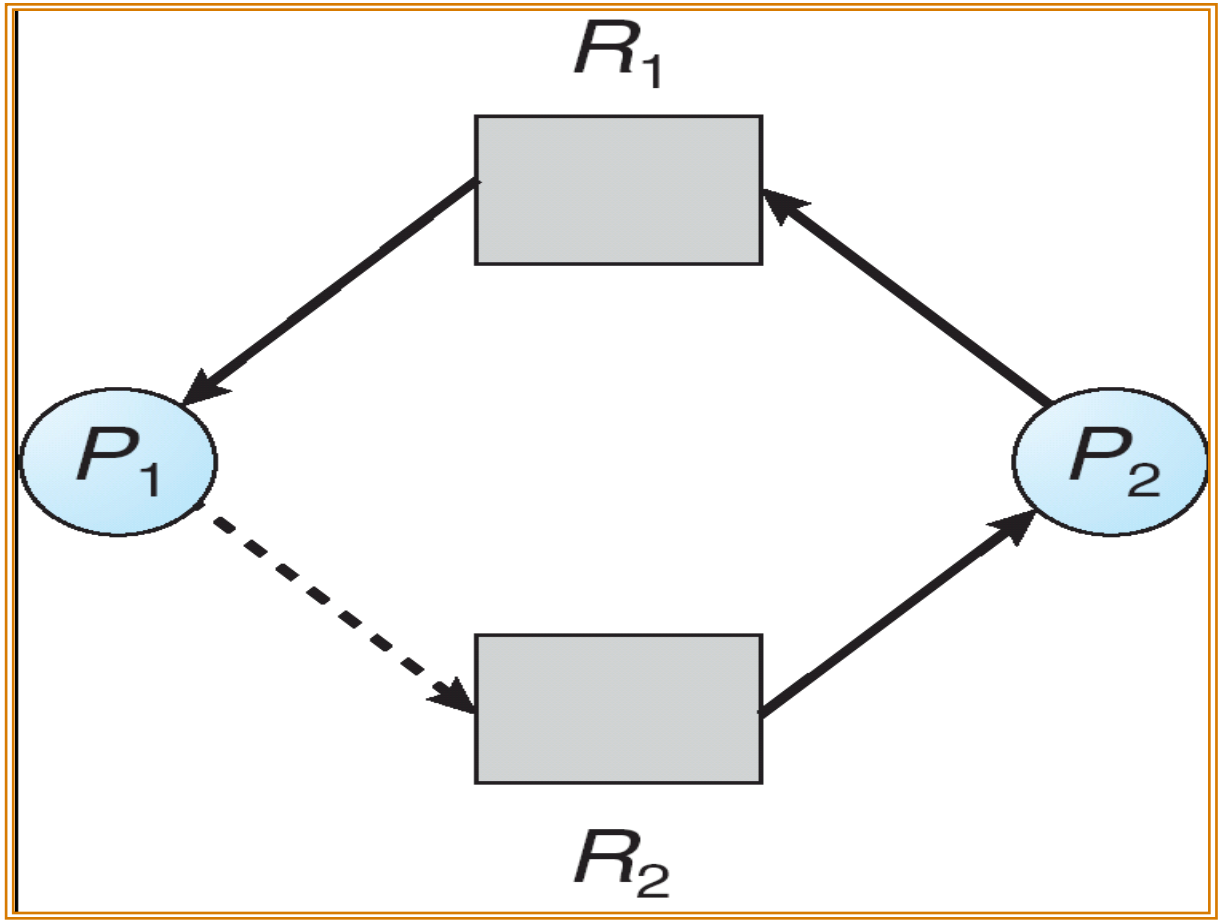
- Resources must be claimed *a priori* in the system.

- A claim edge denotes that a request may be made in future. Based on claim edges we can see if there is a chance for a cycle and then grant requests if the system will again be in a safe state.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



•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

▪**The resource allocation graph is not much useful if there are multiple instances for a resource**

Example formal algorithms

- Banker's Algorithm
- Resource-Request Algorithm
- Safety Algorithm

Banker's Algorithm

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

Data Structures

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

- **Available**: **vector** of length ***m*.**

If **available [*j*] = *k***, there are ***k*** instances of resource type R_j available

- **Max**: *n x m* **matrix**.

If ***Max* [*i,j*] = *k***, then process P_i may request ***at most k*** instances of resource type R_j

- **Allocation**: *n* x *m* **matrix**.

If **Allocation [*i,j*] = *k*** then P_i is ***currently allocated k*** instances of R_j

- **Need**: *n* x *m* **matrix**.

If ***Need* [*i,j*] = *k***, then P_i ***may need k*** more instances of R_j to complete its task

***Need* [*i,j*] = *Max* [*i,j*] – *Allocation* [*i,j*]**

Safety Algorithm

Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively.
Initialize:
Work = ***Available***
Finish **[*i*]** = *false* for $i = 0, 1, \ldots, n-1$

2. Find an index *i* such that:
Finish [*i*] = *false* AND $Need_i \leq Work$

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

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

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

The Algorithm requires *m* x n^2 operations to detect whether the system is in deadlocked state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

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

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

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

- *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.

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

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

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

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) = 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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.

- Can request for (3,3,0) by P4 be granted?

- Can request for (0,2,0) by P0 be granted?

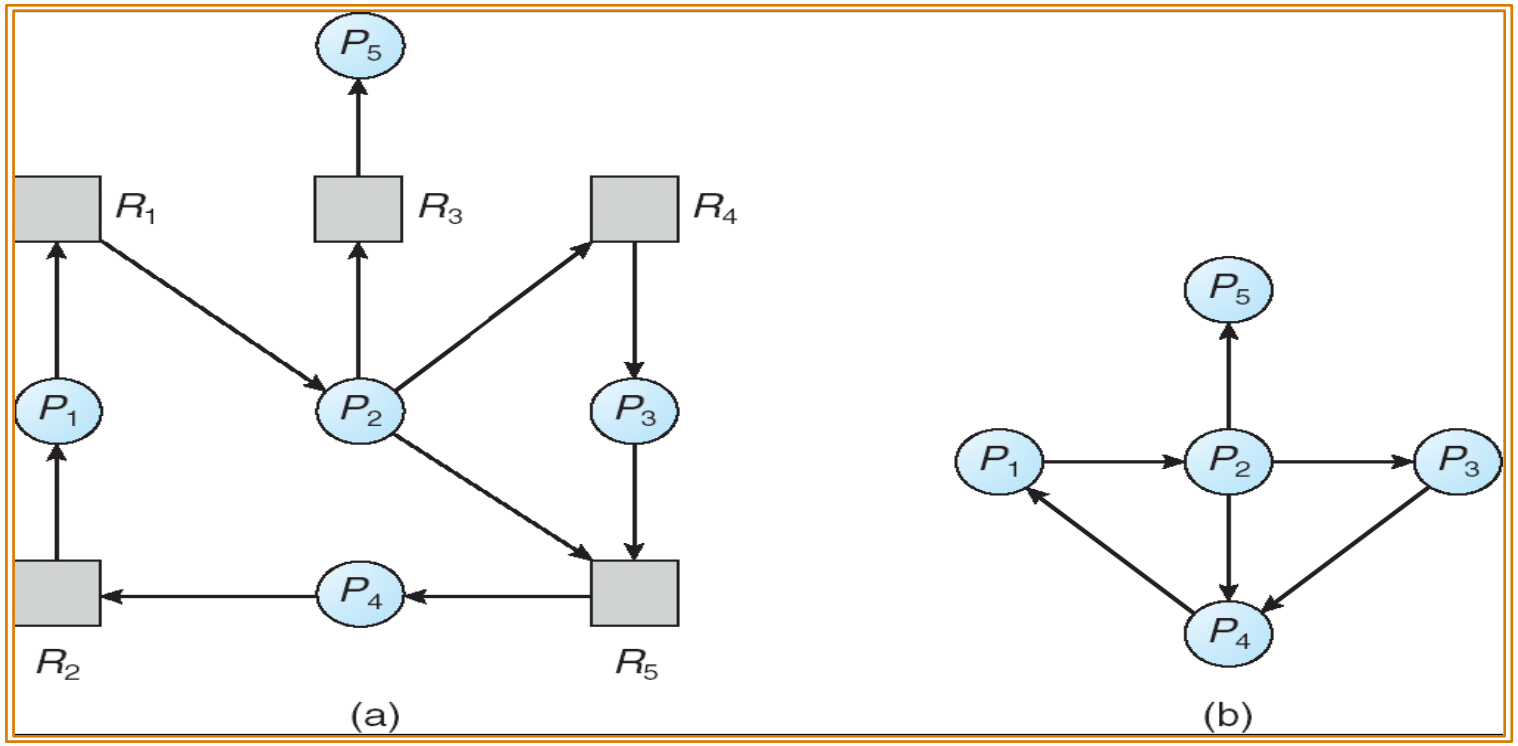
Deadlock Detection

- 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 -> P_j if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.
- *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

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

   (a)Work = Available

   (b)For $i = 1,2, \ldots, n$, if $Allocation_i$ Not equal to 0, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.

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

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

   (b)$Request_i \leq Work$

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

Detection Algorithm (Cont.)

3. Work = Work + $Allocation_i$

   Finish[i] = true

go to step 2.

4. If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state.
   Moreover,
   if Finish[i] == false, then P_i is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	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.

Recovery from Deadlock: Process Termination

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

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Thank
You