I thank all the students involved in preparing this document for their support without whose help it would not have been possible to complete this document.

Q1. Explain the four main elements of a computer system in detail?

Ans: -> At the top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

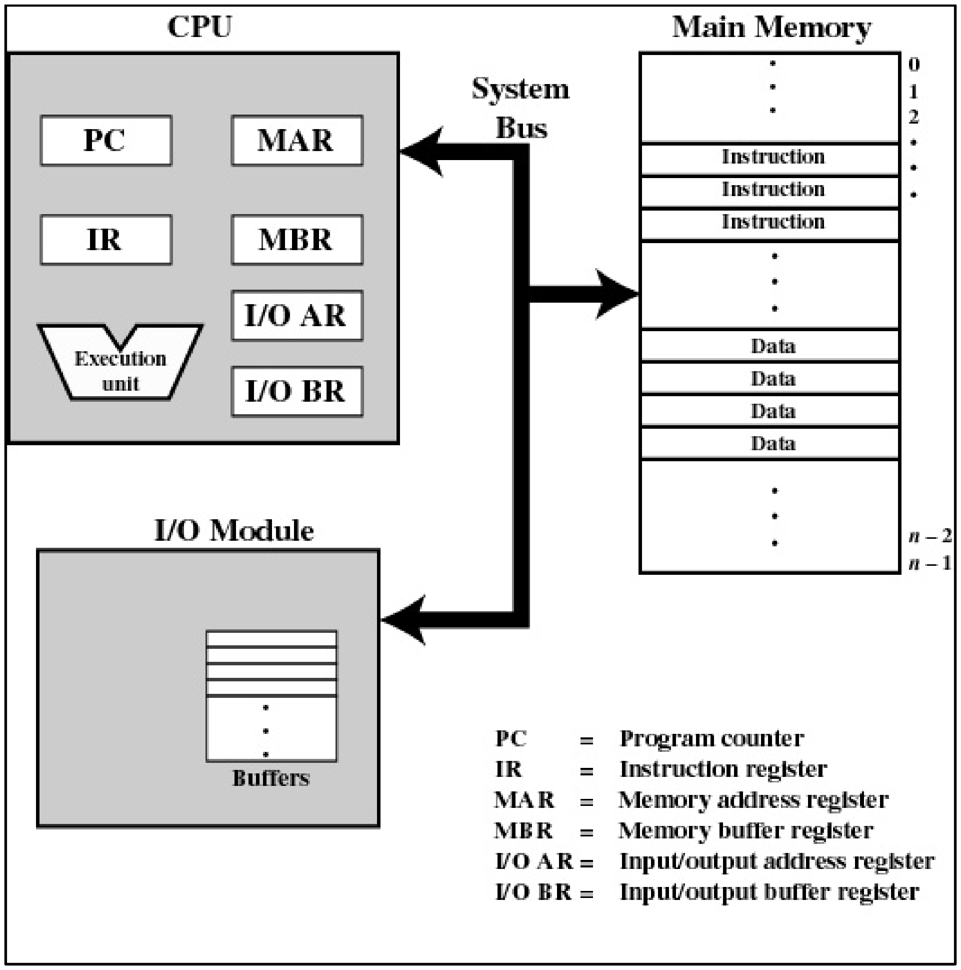
- Processor: Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit (CPU).

- Main memory: Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as real memory or primary memory.

- I/O modules: Move data between the computer and its external environment.The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.

- System bus: Provides for communication among processors, main memory, and I/O modules.

-> One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal (to the processor) registers: a memory address register (MAR), which specifies the address in memory for the next read or write; and a memory buffer register (MBR), which contains the data to be written into memory or which receives.



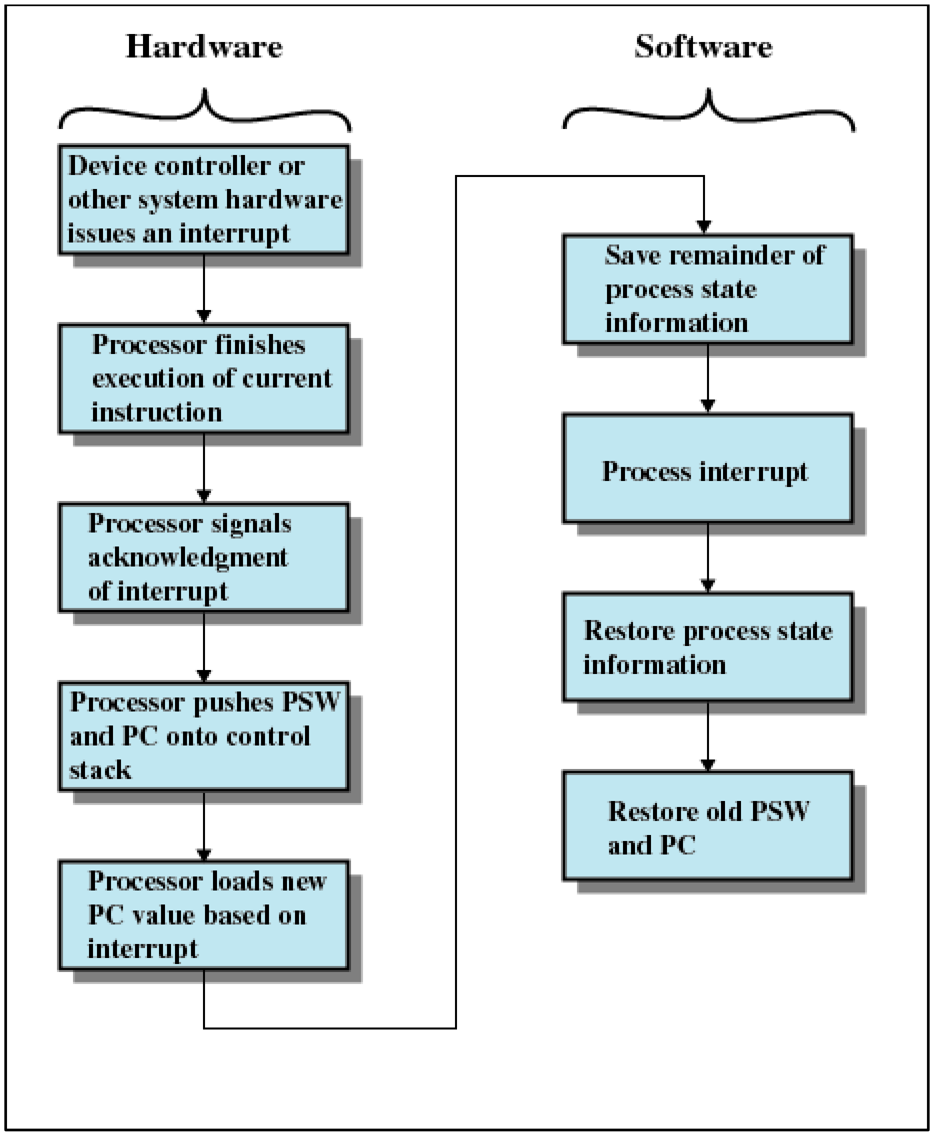
Q2. What is an interrupt? How are simple and multiple interrupts dealt with? Explain in detail?

Ans. A mechanism by which other modules (I/O, Memory) may interrupt the normal sequencing of the processor. Interrupts are provided primarily as a way to improve processor utilization.

Simple Interrupts:-

- In an operating system (OS), interrupt processing is a mechanism by which the normal execution of a program can be temporarily halted to handle an external event or condition.
- Interrupts are signals generated by hardware or software to gain the attention of the CPU and request it to stop its current activities and switch to a specific routine to handle the interrupt.

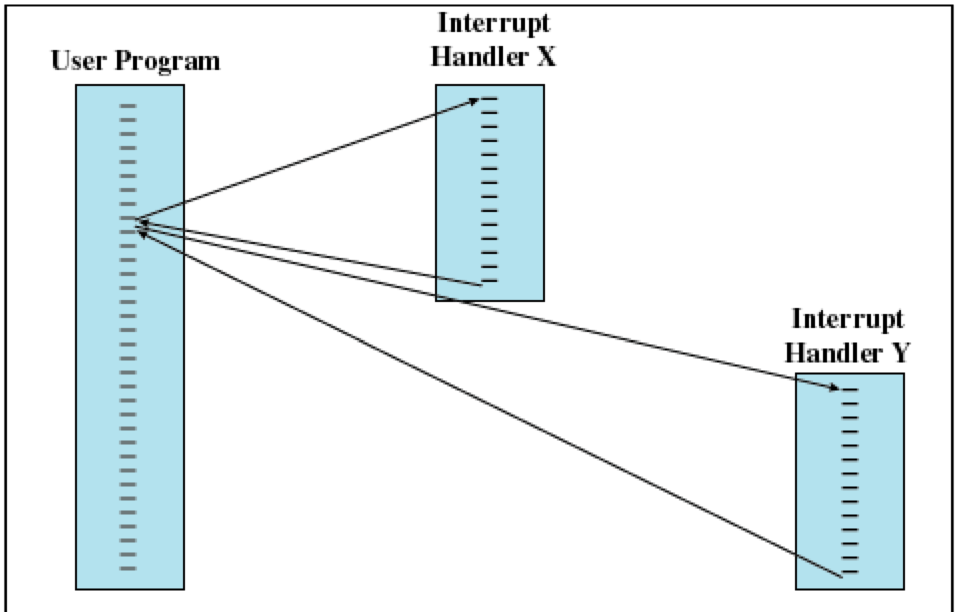
- Simple interrupt processing is straightforward and sufficient for handling basic interrupt scenarios.


Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

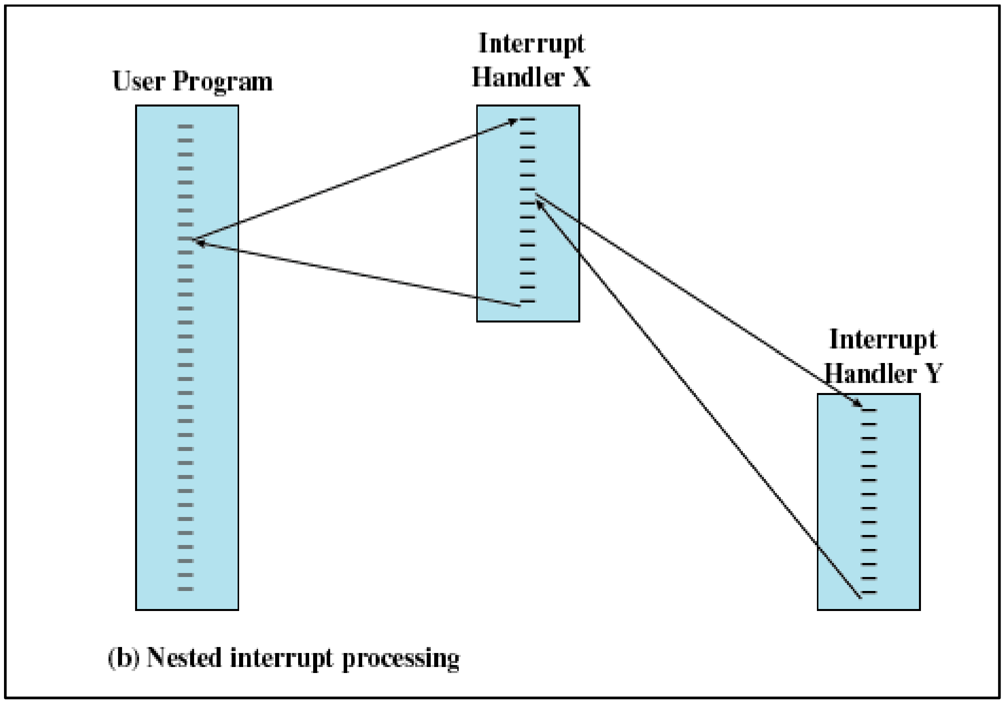
Multiple Interrupts:-

- If one or more interrupts can occur while an interrupt is being processed. For example, a program may be receiving data from a communications line and printing results at the same time. The printer will generate an interrupt every time that it completes a print operation.
- Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed.

- A disabled interrupt simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has re-enabled interrupts.
- Thus, if an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are re-enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred.



(a) Sequential interrupt processing



Q-3 What is an interrupt? Describe the steps that the system takes when interrupt occurs.

Ans. A mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Interrupts are provided primarily as a way to improve processor utilization. Most I/O devices are slower than the processor, it must pause to wait for device E.g processor-printer. There are 4 classes of interrupts:

1. **Program:** Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
2. **Timer:** Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
3. **I/O:** Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
4. **Hardware Failure:** Generated by a failure, such as power failure or memory parity error.

Steps taken by the system when interrupt occurs:

1. The device issues an interrupt signal to the processor.

2. The processor finishes execution of the current instruction before responding to the interrupt.

3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.

4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word 3 (PSW) and the location of the next instruction to be executed, which is contained in the program counter (PC). These can be pushed onto a control stack.

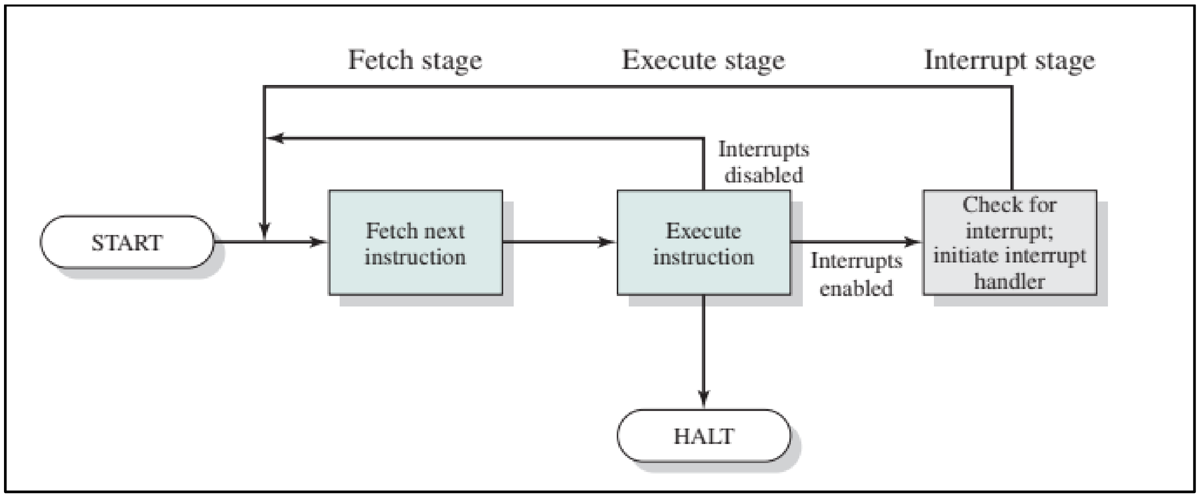
5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt. Depending on the computer architecture and OS design, there may be a single program,one for each type of interrupt, or one for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke.

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values,plus any other state information, need to be saved.

7. The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.

8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.

9. The final act is to restore the PSW and program counter values from the stack.As a result, the next instruction to be executed will be from the previously interrupted program.



Q4. Explain instruction cycle in detail.

Ans. A program to be executed by a processor consists of a set of instructions stored in memory. The processing required for a single instruction is called an instruction cycle.

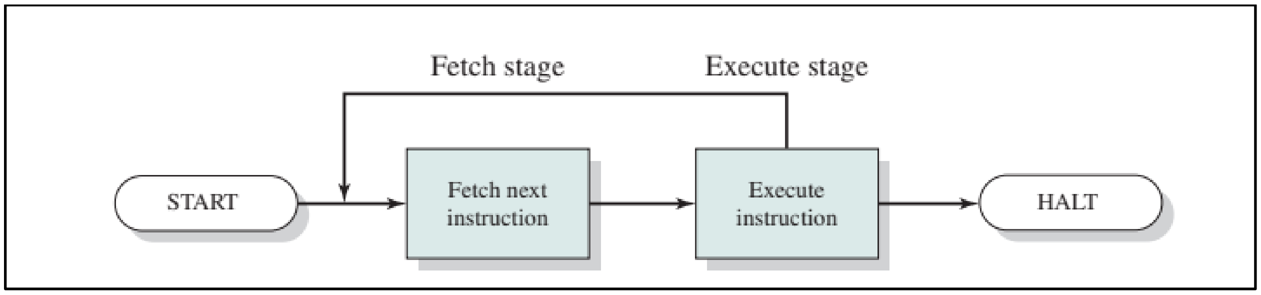
-> Instruction processing consists of two steps: the fetch stage and the execute stage.

-> Program execution consists of repeating the process of instruction fetch and instruction execution.

-> Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

-> At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Eg: the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained subsequently.

-> The fetched instruction is loaded into the instruction register (IR). The instruction contains
bits that specify the action the processor is to take. The processor interprets the instruction
and performs the required action. In general, these actions fall into four categories:
o **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
o **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
o **Data processing:** The processor may perform some arithmetic or logic operation on data.
o **Control:** An instruction may specify that the sequence of execution be altered.



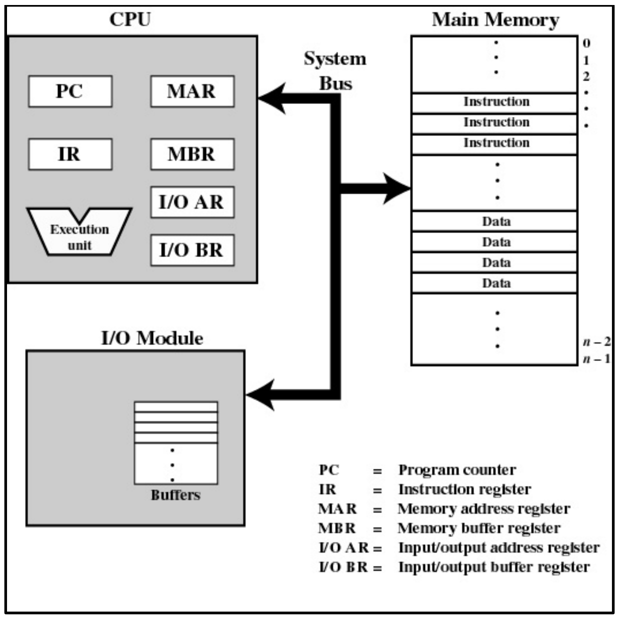
Q5. What is an operating system? Give any two examples (names) of operating systems. List the resources of a computer system that are managed by an operating system. Mention the main objectives of the operating system and explain them in brief.
Ans. An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O (input/output) devices on behalf of its users. To run other programmes, every computer must have at least one operating system installed.

-> Eg: Windows, Linux, and Android are examples of operating systems that enable the user to use programs like MS Office, Notepad, and games on the computer or mobile phone.

-> Resources that are managed by the OS are: -

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

- o **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit (CPU).
- o **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as real memory or primary memory.
- o **I/O modules:** Move data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.
- o **System bus:** Provides for communication among processors, main memory, and I/O modules.



-> Objectives of an OS are as follows : -

An operating system can be thought of as having three objectives:
• **Convenience:** An OS makes a computer more convenient to use.
• **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
• **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions with-
out interfering with service.

Q6. List and briefly explain five storage management responsibilities of a typical OS.
Ans. System managers need efficient and orderly control of storage allocation. The OS, to satisfy these requirements, has five principal storage management responsibilities:

1. **Process isolation:** The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
2. **Automatic allocation and management:** Programs should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the OS can achieve efficiency by assigning memory to jobs only as needed.
3. **Support of modular programming:** Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically.
4. **Protection and access control:** Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by applications. At other times, it threatens the integrity of programs and even of the OS itself. The OS must allow portions of memory to be accessible in various ways by various users.
5. **Long-term storage:** Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

-> Typically, operating systems meet these requirements with virtual memory and file system facilities. The file system implements a long-term store, with information stored in named objects,called files. The file is a convenient concept for the programmer and is a useful unit of access control and protection for the OS.

Q7. Explain various reasons for process creation and process termination.
Ans. Process creation and termination are fundamental aspects of operating systems and are crucial for the execution of programs. Here are various reasons for both process creation and process termination:

- **Process Creation:**
  - When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.
  - In a batch environment, a process is created in response to the submission of a job.
  - In an interactive environment, a process is created when a new user attempts to log on.
  - In both cases, the OS is responsible for the creation of the new process. An OS may also create a process on behalf of an application.
  - For example, if a user requests that a file be printed, the OS can create a process that will manage the printing. The requesting process can thus proceed independently of the time required to complete the printing task.

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, user program can dictate the creation of a number of processes.

  - Traditionally, operating systems created processes transparently for users or applications. Many modern systems still follow this approach. However, allowing one process to explicitly create another can be beneficial for enhanced control and flexibility.
  - For example, an application process may generate another process to receive data that the application is generating and to organize that data

into a form suitable for later analysis. The new process runs in parallel to the original process and is activated from time to time when new data is available. This arrangement can be very useful in structuring the application.

- As another example, a server process (e.g., print server, file server) may generate a new process for each request that it handles. When the OS creates a process at the explicit request of another process, the action is referred to as process spawning.

➢ **Process Termination:**

- Any computer system must provide a means for a process to indicate its completion.
- A batch job should include a Halt instruction or an explicit OS service call for termination.
- In the former case, the Halt instruction will generate an interrupt to alert the OS that a process has completed.
- For an interactive application, the action of the user will indicate when the process is completed.
- For example, in a time-sharing system, the process for a particular user is to be terminated when the user logs off or turns off his or her terminal. On a personal computer or workstation, a user may quit an application (e.g., word processing or spreadsheet). All of these actions ultimately result in a service request to the OS to terminate the requesting process.

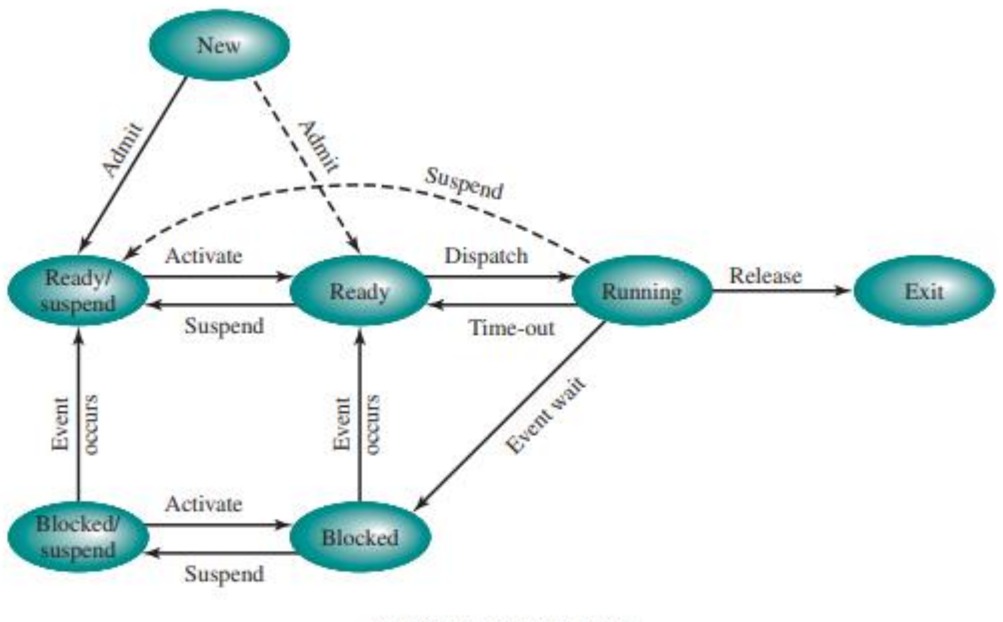
Normal completion	The process executes an OS service call to indicate that it has completed Running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries

	to use it in an improper fashion, such as writing to a read only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a non-existent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

-> Understanding the reasons for process creation and termination is crucial for effective system management and resource utilization. Operating systems need to balance the need for creating new processes with the necessity of terminating processes to maintain system stability and efficiency.

Q8. Design a seven-state process model using a proper state transition diagram.
Ans. A seven-state process model is a five state process model with two suspended states.

- When the OS performs a swapping-out operation, it faces a choice: admit a newly created process or bring in a previously suspended process. While the preference may be to bring in a suspended process to avoid increasing system load, a challenge arises. Suspended processes were in the Blocked state, and bringing a blocked process back isn't helpful as it's still not ready for execution. However, each process in the Suspend state was originally blocked on an event, and when that event occurs, the process is not blocked and potentially available for execution.
- To address this, two independent concepts are identified: whether a process is waiting on an event (blocked or not) and whether a process has been swapped out of main memory (suspended or not). This leads to the need for four states to accommodate the 2 * 2 combination of these concepts.
  - **Ready:** The process is in main memory and available for execution.
  - **Blocked:** The process is in main memory and awaiting an event.
  - **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
  - **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.



(b) With two suspend states

- There are old states (from five states model):
    - **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
    - **Ready:** A process that is prepared to execute when given the opportunity.
    - **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.

- In virtual memory systems, processes can be partially in main memory. While virtual memory eliminates the need for explicit swapping during address references, excessive active processes partially in main memory can lead to performance issues. Consequently, the operating system may still need to perform explicit and complete process swapping to maintain optimal performance.
- **Blocked Blocked/Suspend:** If no ready processes are available, the operating system may swap out at least one blocked process to make room for another unblocked process. This transition can also occur, even with ready processes available, if the OS determines that the currently running or a ready process needs more main memory for optimal performance.
- **Blocked/Suspend Ready/Suspend:** A process in Blocked/Suspend state transitions to Ready/Suspend when the awaited event occurs, necessitating accessible state information for suspended processes by the operating system.
- **Ready/Suspend Ready:** When no ready processes are in main memory, the OS must bring one in for execution. If a process in Ready/Suspend state has higher priority than those in the Ready state, the OS may prioritize fetching the higher-priority process over minimizing swapping as per the OS design.
- **Ready Ready/Suspend:** The OS typically prioritizes suspending blocked processes over ready ones. However, a ready process may be suspended if it's necessary to free up a significant block of main memory. The OS might also suspend a lower-priority ready process

instead of a higher-priority blocked process if it anticipates the blocked process becoming ready soon.

Several other transitions from five-states model they are use in seven-state model also:

- **Ready [] Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher.
- **Running [] Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; virtually all multiprogramming operating systems impose this type of time discipline. There are several other alternative causes for this transition, which are not implemented in all operating systems. Of particular importance is the case in which the OS assigns different levels of priority to different processes. Suppose, for example, that process A is running at a given priority level, and process B, at a higher priority level, is blocked. If the OS learns that the event upon which process B has been waiting has occurred, moving B to a ready state, then it can interrupt process A and dispatch process B. We say that the OS has preempted process A. Finally, a process may voluntarily release control of the processor. An example is a background process that performs some accounting or maintenance function periodically.
- **Running [] Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.
- **Running [] Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. For example, a process may request a service from the OS that the OS is not prepared to perform immediately. It can request a resource, such as a file or a shared section of virtual memory, that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data or waiting for a message from another process.

- **Blocked Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.

Several other transitions that are worth considering are the following:

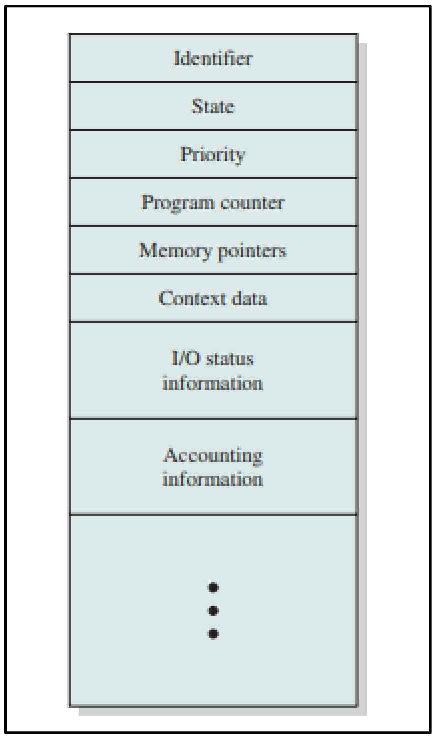
- **New Ready/Suspend and New Ready:** When a new process is created, it can be added to the Ready or Ready/Suspend queue. The OS must create a process control block and allocate an address space. Early housekeeping allows for a large pool of unblocked processes, potentially leading to insufficient main memory space. The (New: Ready/Suspend) transition is used to address this, while a just-in-time approach argues for creating processes as late as possible to reduce OS overhead, even during system congestion with blocked processes.
- **Blocked/Suspend Blocked:** The inclusion of the (New: Ready/Suspend) transition might initially appear poorly designed, as bringing in a non-ready process to main memory seems counterintuitive. However, it becomes relevant in scenarios where a higher-priority process in the (Blocked/Suspend) queue is expected to unblock soon after a process terminates, making it preferable to bring in the blocked process rather than a ready process to utilize the freed-up main memory.
- **Running Ready/Suspend:** Normally, a running process is moved to the Ready state when it's time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory.
- **Any State Exit:** Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state.

Q9. What is a Process Control Block (PCB)? Discuss its need and describe the fields of PCB.

- A Process Control Block is a crucial data structure that encapsulates the specifics about every process present within a computing system. When a process is birthed, the operating system instantaneously establishes a unique PCB in the OS for it. This is done to ensure that the system can actively monitor the state of the process, efficiently manage the resources it might require, and oversee its execution in a streamlined manner.
- Essentially, the PCB in the OS acts as the fingerprint of a process, providing the OS with all the pertinent details required to manage and control the process optimally. Without the PCB in the OS, the operating system would be blind to the processes it needs to govern, making the PCB indispensable for smooth OS functionality.

- **Need for PCB:**
  - **Process Management:**
    - The PCB allows the operating system to keep track of all the necessary information about a process, including its current state, program counter, registers, and other essential details.
    - It enables the operating system to manage multiple processes efficiently by providing a centralized repository of information.
  - **Context Switching:**
    - During multitasking or time-sharing, the operating system needs to switch between different processes quickly. The PCB facilitates this context switching by saving and restoring the state of a process.
  - **Resource Management:**
    - The PCB contains information about the resources allocated to a process, such as memory, file descriptors, and open files. This helps in effective resource management.
  - **Synchronization and Communication:**
    - PCBs include fields related to synchronization and communication between processes, such as flags, queues, and pointers, which are crucial for inter-process communication and synchronization.
  - **Security and Protection:**
    - Information in the PCB is used to enforce security and protection mechanisms. It includes details about the process's permissions, ownership, and other security-related attributes.

- **Fields of PCB:**



- **Identifier:**
    - A unique identifier associated with this process, to distinguish it from all other processes.
- **State:**
    - If the process is currently executing, it is in the running state.

- **Priority**:
    - Process priority is a numeric value that represents the priority of each process. The lesser the value, the greater the priority of that process. This priority is assigned at the time of the creation of the PCB and may depend on many factors like the age of that process, the resources consumed, and so on. The user can also externally assign a priority to the process.
- **Program counter:**
    - The address of the next instruction in the program to be executed.

- **Memory pointers:**
    - Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:**
    - These are data that are present in registers in the processor while the process is executing.
- **I/O status information:**
    - Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:**
    - May include the amount of processor time and clock time used, time limits, account numbers, and so on.

The information in the preceding list is stored in a data structure, typically called a process control block, that is created and managed by the OS. The significant point about the process control block is that it contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value, such as *blocked* or *ready* (described subsequently). The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.

Thus, we can say that a process consists of program code and associated data plus a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the *running* state.

Q.10) List reasons why a mode switch between threads may be cheaper than a mode switch between processes.

Ans. A mode switch between threads within the same process is generally considered to be cheaper than a mode switch between processes. Here are some reasons why:

Shared Address Space:

Threads within the same process share the same address space, which means they can easily access each other's data without the need for complex memory management or communication mechanisms. This reduces the overhead of switching between threads compared to processes that have separate address spaces and require inter-process communication (IPC).

Lightweight:

Threads are generally lighter weight than processes. They share resources such as memory and file descriptors, and the overhead of creating and managing a thread is typically lower than that of a process. This results in faster context switches between threads.

No Context Switching Overhead:

When switching between threads, there is no need for a full context switch. Context switching involves saving the state of the current execution context (registers, program counter, etc.) and loading the state of the new context. In the case of threads, this is a faster operation as they share the same address space.

Efficient Communication:

Communication between threads is more efficient since they can directly share data through shared variables. In contrast, inter-process communication between separate processes requires more overhead, often involving IPC mechanisms like message passing or shared memory with additional synchronization mechanisms.

Synchronization Overhead:

Threads within a process can synchronize more easily because they share the same memory space. Synchronization mechanisms such as locks or semaphores are more efficient when used between threads than when used between processes.

Resource Sharing:

Threads share the same resources, such as file descriptors and open sockets. In contrast, separate processes have their own independent copies of these resources, leading to additional overhead when switching between processes.

Faster Creation and Termination:

Creating and terminating threads is generally faster than creating and terminating processes. Threads can be spawned more quickly since they share resources with the parent process.

User-Level Threads:

Some threading implementations allow for user-level threads, where the operating system is not involved in the thread management. In such cases, thread context switches can be very efficient as they are handled entirely in user space.

Q11) Give four general examples of the use of threads in a single-user multiprocessing system.

Ans. **Multimedia Handling:**

In multimedia applications, separate threads can be dedicated to tasks like audio processing, video decoding, and rendering. This ensures smooth playback and responsiveness even when dealing with large multimedia files.

Background Processing:

Threads can be used for background tasks, such as file downloads, data synchronization, or periodic updates. This allows the main thread to focus on handling user interactions while background threads handle time-consuming tasks.

Asynchronous I/O Operations:

Threads can be employed to handle asynchronous I/O operations, allowing the system to continue processing other tasks while waiting for input/output operations to complete. This is common in network communication and file I/O.

Parallel Data Processing:

When performing data-intensive tasks, such as sorting, filtering, or searching, multiple threads can work in parallel to process different portions of the data. This can lead to significant performance improvements. This can include all divide-and-conquer algorithms.

Q.12) What is Thread? How is it different from process? What is the concept of multithreading? Is it the same as multiprocessing? Discuss.

Ans. <u>Thread:</u> A thread is the smallest unit of execution within a process. It consists of a program counter, a set of registers, and a stack. Threads within the same process share the same memory space and resources, such as file descriptors, while each thread has its own program counter and registers. Threads allow for concurrent execution of tasks within a single process.

Basis	**THREAD**	**PROCESS**
Definition	Smallest unit of execution within a process.	Independent program with its own resources.
Memory Sharing	Share memory space with other threads.	Has its own separate memory space.
Resource Sharing	Share resources (file descriptors, etc.).	Has its own independent set of resources
Overhead	Lower overhead, as threads share resources.	Higher overhead, as processes are independent.
Communication	Communicates through shared memory.	Requires Inter-process Communication.

Creation and Termination	Faster to create and terminate.	Slower to create and terminate.
Isolation	Less isolated; can directly access data of other threads in the same process.	Highly isolated; communication requires explicit mechanisms.
Example Use Case	UI responsiveness, parallel processing, background tasks within an application.	Running independent applications, parallel processing tasks that require strong isolation.

Multithreading: Multithreading is a programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the same resources, including memory space, but have their own execution context. Multithreading enables concurrent execution of tasks, which can improve performance and responsiveness in certain applications.

Basis	**Multi-Threading**	**Multi-Processing**

Memory Sharing	Threads within the same process share the same memory space. This means they can easily access each other's data, which simplifies communication but requires careful synchronization to avoid data races.	Processes have separate memory spaces. Communication between processes is achieved through inter-process communication (IPC) mechanisms, and they do not share memory by default.
Communication	Communication between threads is typically done through shared memory. Threads can exchange data by directly accessing shared variables, which can lead to efficient communication.	Communication between processes requires explicit IPC mechanisms, such as message passing, pipes, or shared memory with synchronization mechanisms. This adds overhead but ensures better isolation.
Resource Overhead	Threads within the same process share resources such as file descriptors, which reduces resource overhead compared to separate processes.	Processes have their own independent set of resources. While this provides stronger isolation, it also introduces higher resource overhead as each process needs its own resources.

Creation and Termination	Creating and terminating threads is generally faster than creating and terminating processes. Threads can be spawned more quickly since they share resources with the parent process.	Creating and terminating processes involve more overhead, including memory allocation and initialization. Processes are heavier and take more time to start and stop.

Q.13 Design a solution to the Producer Consumer problem using monitors. (Write pseudocode)

Ans. A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data.

➔ By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility. The data variables in the monitor can be accessed by only one process at a time. Thus, a shared data structure can be protected by placing it in a monitor. If the data in a monitor represent some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

➔ A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

• cwait(c) : Suspend execution of the calling process on condition c . The monitor is now available for use by another process.

• csignal(c) : Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

➔ **Pseudocode:**

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                          /* space for N items */
int nextin, nextout;                                        /* buffer pointers */
int count;                                        /* number of items in buffer */
cond notfull, notempty;               /* condition variables for synchronization */

void append (char x)
{
      if (count == N) cwait(notfull);         /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (nonempty);                       /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);      /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                    /* one fewer item in buffer */
      csignal (notfull);                         /* resume any waiting producer */
}
{                                                              /* monitor body */
      nextin = 0; nextout = 0; count = 0;            /* buffer initially empty */
}
```

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Q14. What are the advantages of monitor over semaphore in inter process communication? Give a solution to the Producer Consumer problem using monitors.

Ans. A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. It is an integer variable (S), and it is initialized with the number of resources in the system. The wait() and signal() methods are the only methods that may modify the semaphore (S) value. When one process modifies the semaphore value, other processes can't modify the semaphore value simultaneously.

-> A monitor is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

-> Advantages of Monitors over Semaphores:

1. A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. On the other hand, a monitor is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true.
2. When a process uses shared resources in semaphore, it calls the **wait()** method and blocks the resources. When it wants to release the resources, it executes the **signal()** In contrast, when a process uses shared resources in the monitor, it has to access them via procedures.
3. Semaphore is an integer variable, whereas monitor is an abstract data type.
4. In semaphore, an integer variable shows the number of resources available in the system. In contrast, a monitor is an abstract data type that permits only a process to execute in the crucial section at a time.
5. Semaphores have no concept of condition variables, while monitor has condition variables.

6. A semaphore's value can only be changed using the **wait()** and **signal()** In contrast, the monitor has the shared variables and the tool that enables the processes to access them.

-> Solution using Monitors:

A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to buffer . The procedure first checks the condition notfull to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition. Some other process (producer or consumer) may now enter the monitor. Later, when the buffer is no longer full, the blocked process may be removed from the queue, reactivated, and resume processing. After placing a character in the buffer, the process signals the notempty condition. A similar description can be made of the consumer function.

This solution points out the division of responsibility with monitors compared to semaphores. In the case of monitors, the monitor construct itself enforces mutual exclusion: It is not possible for both a producer and a consumer simultaneously to access the buffer. However, the programmer must place the appropriate cwait and csignal primitives inside the monitor to prevent processes from depositing items in a full buffer or removing them from an empty one. In the case of semaphores, both mutual exclusion and synchronization are the responsibility of the programmer.

Q15. Write the Solution to the readers/writers Problem Using Semaphore where readers have priority.

Ans. The **readers/writers problem** is defined as follows:

There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

-> Readers Have Priority

The writer process is simple. The semaphore *wsem* is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on *wsem*. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable readcount is used to keep track of the number of readers, and the semaphore x is used to assure that readcount is updated properly.

-> **Pseudocode:**

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

Q16. Write a solution to the Readers/Writers Problem Using Semaphore where Writers have priority.

Ans. The **readers/writers problem** is defined as follows:

There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

-> Writers Have Priority:

Writers Have Priority solution that guarantees that no new readers are allowed access to the data area once at least one writer has declared a desire to write. For writers, the following semaphores and variables are added to the ones already defined:

- A semaphore rsem that inhibits all readers while there is at least one writer desiring access to the data area.
- A variable writecount that controls the setting of rsem
- A semaphore y that controls the updating of writecount.

For readers, one additional semaphore is needed. A long queue must not be allowed to build up on rsem ; otherwise writers will not be able to jump the queue. Therefore, only one reader is allowed to queue on rsem , with any additional readers queueing on semaphore z , immediately before waiting on rsem.

```
/* program readersandwriters */
int readcount,writecount;
void reader()
{
    while (true){
      semWait (z);
          semWait (rsem);
              semWait (x);
                   readcount++;
                   if (readcount == 1)
                        semWait (wsem);
                   semSignal (x);
              semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
              readcount--;
              if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
void writer ()
{
    while (true){
      semWait (y);
          writecount++;
          if (writecount == 1)
              semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
          writecount;
          if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

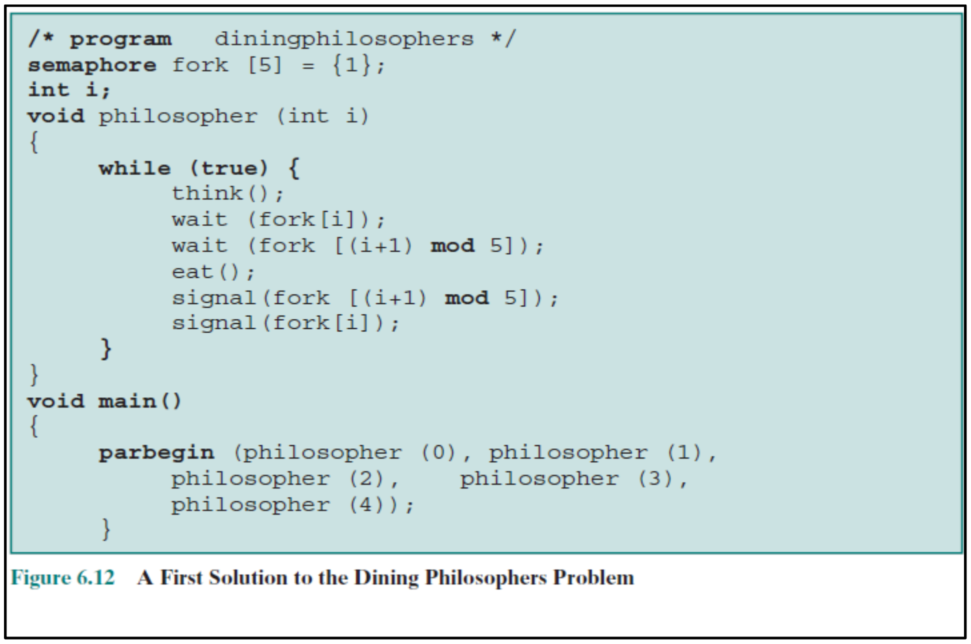
Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

Q17. Design a solution to the Dining Philosophers Problem using semaphore. (Write pseudocode).

Ans.

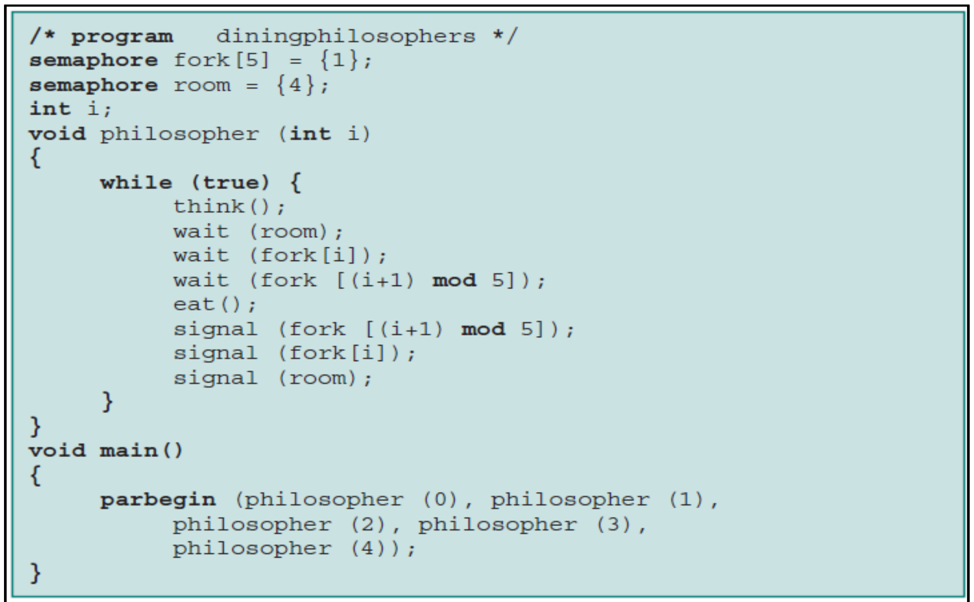
Solution Using Semaphores: Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table. This solution, alas, leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this undignified position, all philosophers starve. To overcome the risk of deadlock, we could buy five additional forks (a more sanitary solution!) or teach the philosophers to eat spaghetti with just one fork.

-> As another approach, we could consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers,at least one philosopher will have access to two forks. This solution is free of deadlock and starvation.



```
/* program   diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1),
          philosopher (2),    philosopher (3),
          philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

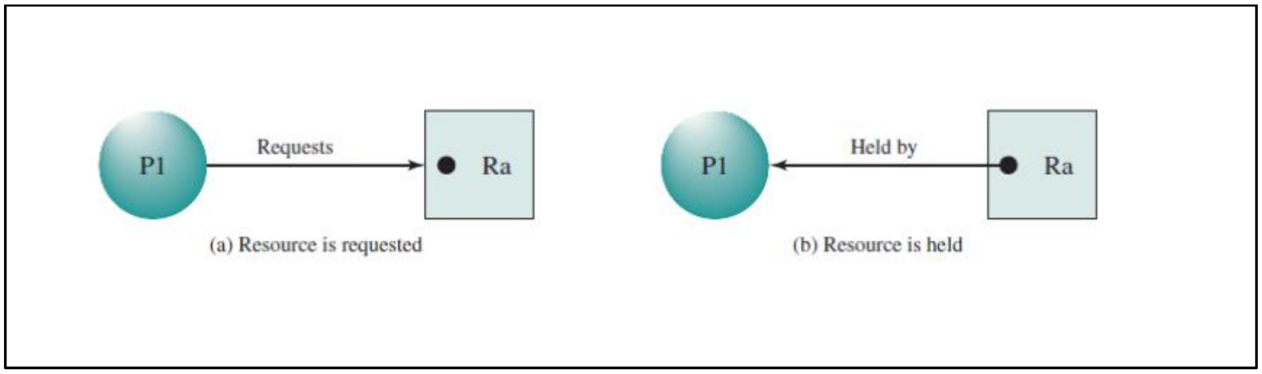


```
/* program   diningphilosophers */
semaphore fork[5]  = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (room);
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal (fork [(i+1) mod 5]);
          signal (fork[i]);
          signal (room);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1),
          philosopher (2), philosopher (3),
          philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

Q18. Define deadlock. How does it differ from livelock? Also what is starvation? List and explain conditions for deadlock in the system.

Ans. **Deadlock** can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. Livelock is a deadlock-like situation in which processes block each other with a repeated state change yet make no progress.



Livelock: A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

-> Deadlock occurs when a set of processes wishes to enter their critical sections but no process can succeed. With livelock, there are possible sequences of executions that succeed, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section.

Starvation: A situation in which a runnable process is overlooked indefinitely by the scheduler although it is able to proceed, it is never chosen.

Eg: Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

-> **List of Conditions of Deadlock:**

1. Mutual exclusion

2. Hold and wait

3. No preemption

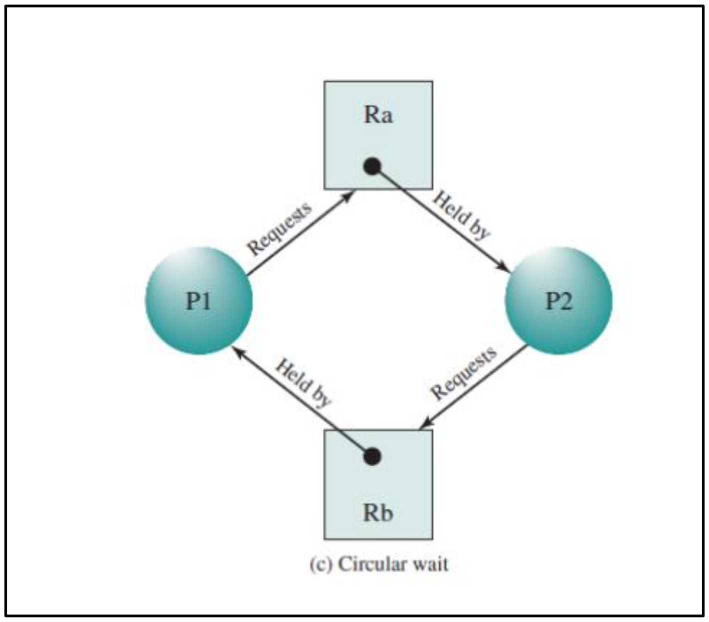
4. Circular wait

Mutual exclusion: Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

Hold and wait: A process may hold allocated resources while awaiting assignment of other resources.

No preemption: No resource can be forcibly removed from a process holding it. For example, mutual exclusion is needed to ensure consistency of results and the integrity of a database. Similarly, preemption should not be done arbitrarily. For example, when data resources are involved, preemption must be supported by a rollback recovery mechanism, which restores a process and its resources to a suitable previous state from which the process can eventually repeat its actions. The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required:

Circular wait: A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



(c) Circular wait

Q19. Give the variants of Dining Philosophers problem projecting deadlock and starvation. Given an efficient solution to solve the given problem.

Ans. The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A spaghetti is placed at the center of the table along with five chopsticks for each of the philosophers.

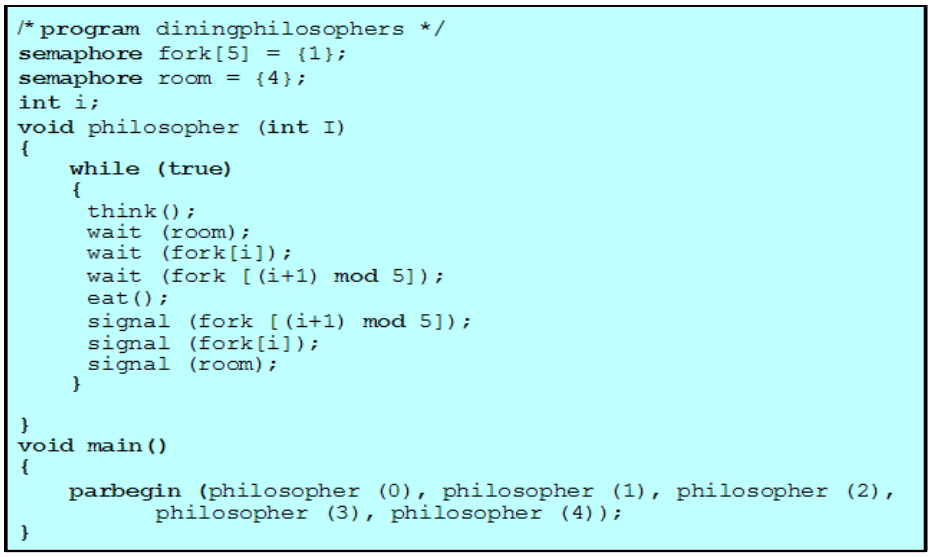
- To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both the immediate left and right chopsticks of the philosopher are available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.
- The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

Variants of Dining philosophers problem:

- Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table. This solution leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this undignified position, all philosophers starve.

- To overcome the risk of deadlock, we could buy five additional forks or teach the philosophers to eat spaghetti with just one fork.

- As another approach, we could consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks.



```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

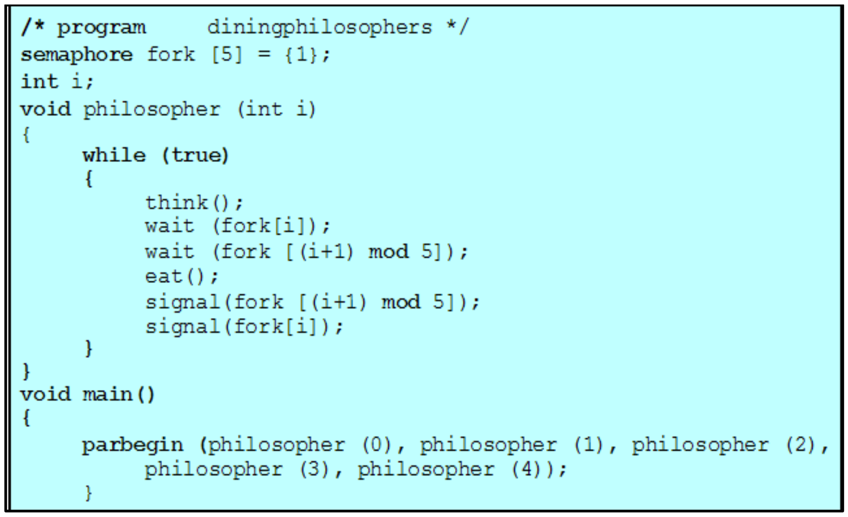
Solution Using Semaphores:

The shared resources, the forks, are protected using semaphores, which act like tokens that a philosopher needs to pick up to eat. When a philosopher wants to eat, they check if both neighboring forks are available (by checking semaphore values). If so, they pick up the forks, they enable wait semaphore to show that they are taken. After eating, they release the forks, incrementing the semaphore values and indicating they are now available. The use of semaphores ensures that philosophers coordinate their actions, avoiding conflicts over forks and ensuring fairness. This solution avoids deadlock by carefully managing the availability of forks through semaphores.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal. whose definitions are as follows:

(1) wait (s){ while (s<=0){ s--;}}

(2) signal (s) { s++; }



```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true)
     {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
     }
```

Solution using Monitor:

Imagine each philosopher as a person sitting at a table, sometimes thinking and sometimes eating spaghetti using two forks. The monitor keeps track of who is thinking, hungry, or eating and makes sure no one fights over forks. When a philosopher wants to eat, they check if their neighbors are also eating. If not, they pick up the forks and start eating. After eating, they put the forks back and let their neighbors know they are available. This way, we prevent conflicts and make sure everyone gets a turn to eat without any issues.

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};         /* availability status of each fork */

void get_forks(int pid)           /* pid is the philosopher id number */
{
   int left = pid;
   int right = (pid++) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);          /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (pid++) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]           /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

Q20. Consider the following snapshot of a system

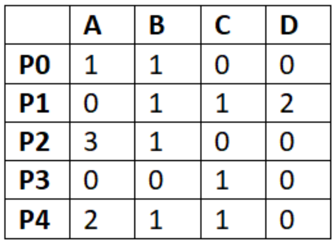
Process Name	Allocated				Maximum Demand				Available			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	3	0	1	1	4	1	1	1	2	0	2	0
P1	0	1	0	0	0	2	1	2				
P2	1	1	1	0	4	2	1	0				
P3	1	1	0	1	1	1	1	1				
P4	0	0	0	0	2	1	1	0				

Answer the following question:

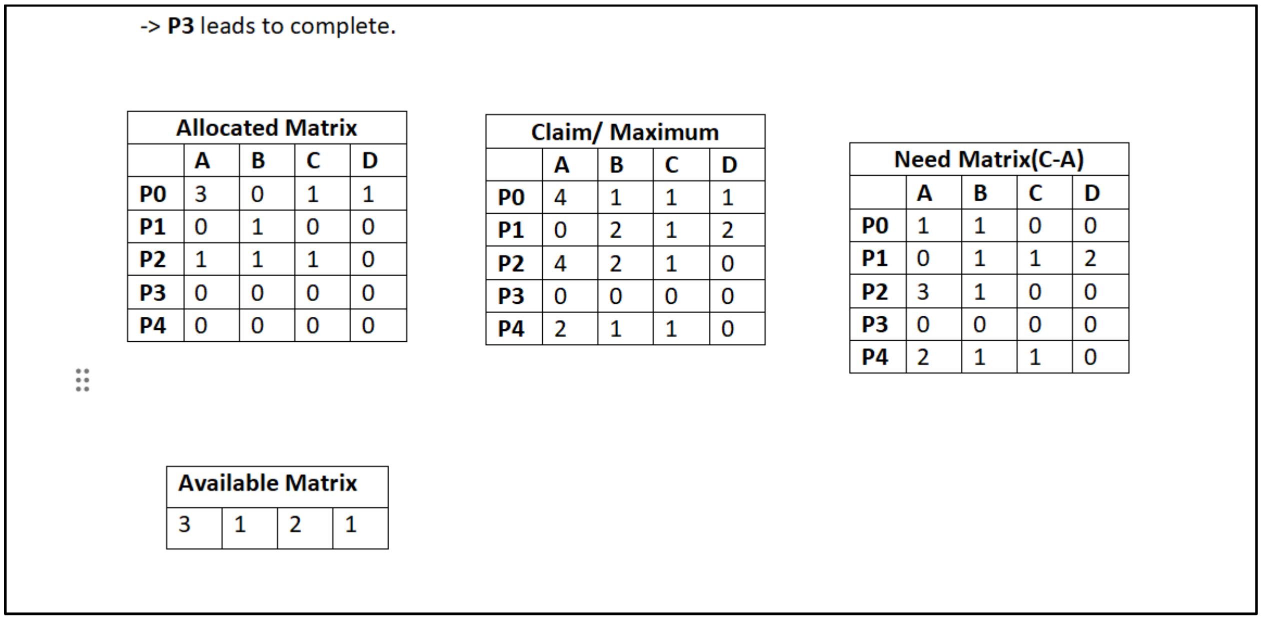
1. What is the content of need matrix?
2. Is the system in safe state? If yes, give a safe sequence else justify.
3. If a request comes from P1 of (0,0,1,0) can we immediately grant the request? If yes, then check whether the system is still in safe state?

Ans.

Need matrix:



	A	B	C	D
P0	1	1	0	0
P1	0	1	1	2
P2	3	1	0	0
P3	0	0	1	0
P4	2	1	1	0

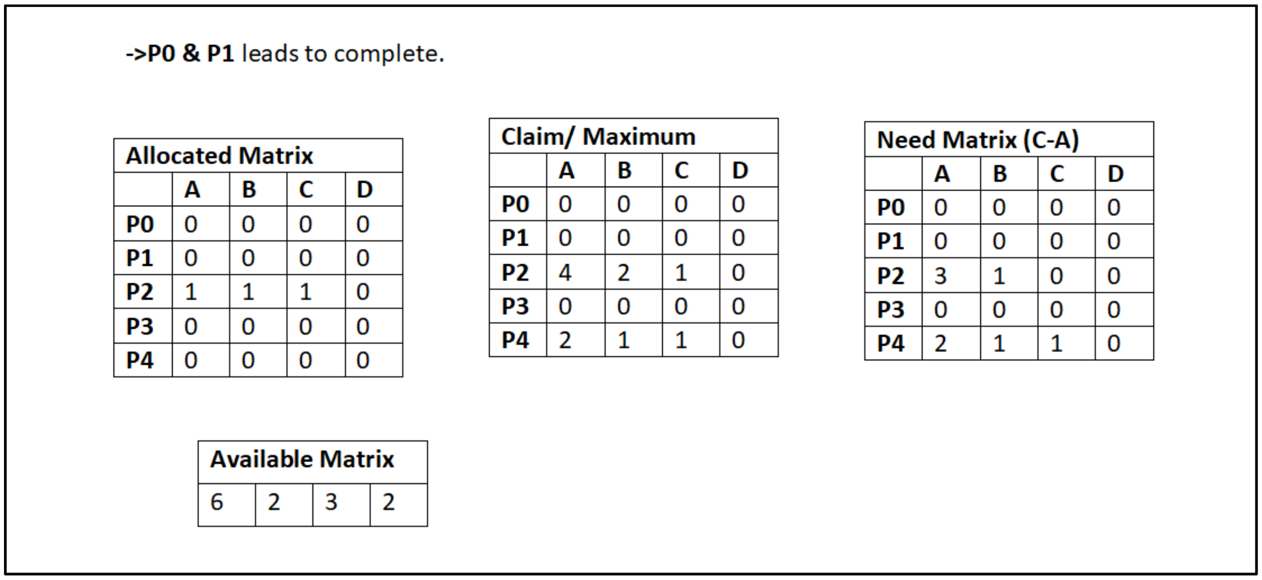

-> **P3** leads to complete.

Allocated Matrix				
	A	**B**	**C**	**D**
P0	3	0	1	1
P1	0	1	0	0
P2	1	1	1	0
P3	0	0	0	0
P4	0	0	0	0

Claim/ Maximum				
	A	**B**	**C**	**D**
P0	4	1	1	1
P1	0	2	1	2
P2	4	2	1	0
P3	0	0	0	0
P4	2	1	1	0

Need Matrix(C-A)				
	A	**B**	**C**	**D**
P0	1	1	0	0
P1	0	1	1	2
P2	3	1	0	0
P3	0	0	0	0
P4	2	1	1	0

Available Matrix			
3	1	2	1

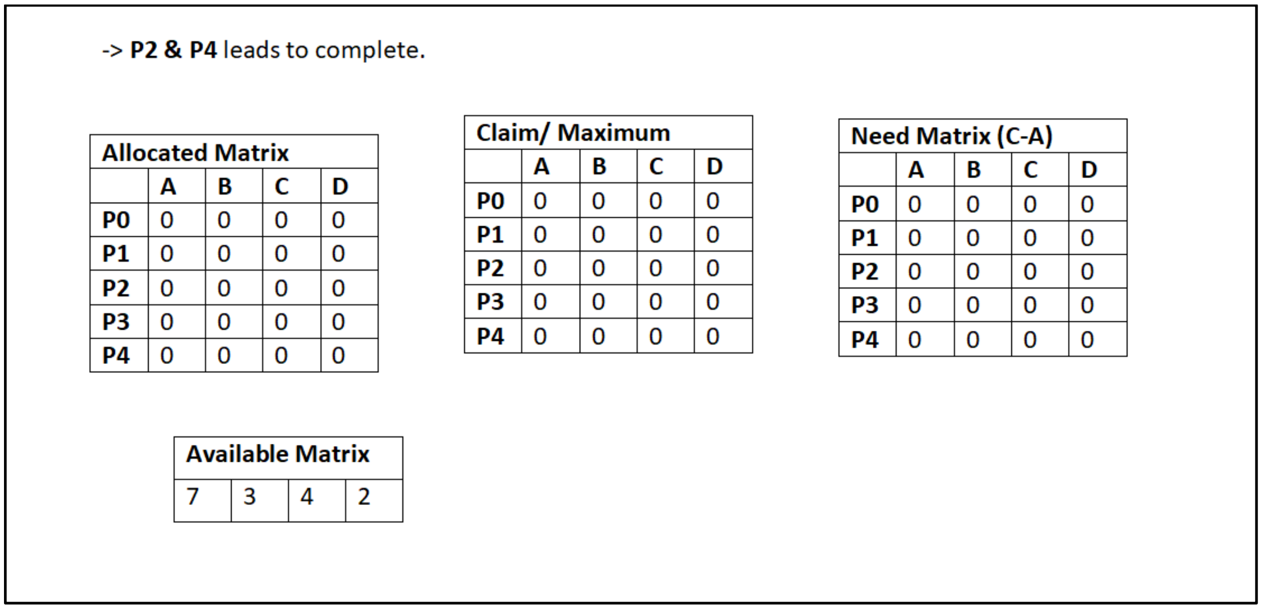

->**P0 & P1** leads to complete.

Allocated Matrix				
	A	**B**	**C**	**D**
P0	0	0	0	0
P1	0	0	0	0
P2	1	1	1	0
P3	0	0	0	0
P4	0	0	0	0

Claim/ Maximum				
	A	**B**	**C**	**D**
P0	0	0	0	0
P1	0	0	0	0
P2	4	2	1	0
P3	0	0	0	0
P4	2	1	1	0

Need Matrix (C-A)				
	A	**B**	**C**	**D**
P0	0	0	0	0
P1	0	0	0	0
P2	3	1	0	0
P3	0	0	0	0
P4	2	1	1	0

Available Matrix			
6	2	3	2



-> **P2 & P4** leads to complete.

Allocated Matrix				
	A	B	C	D
P0	0	0	0	0
P1	0	0	0	0
P2	0	0	0	0
P3	0	0	0	0
P4	0	0	0	0

Claim/ Maximum				
	A	B	C	D
P0	0	0	0	0
P1	0	0	0	0
P2	0	0	0	0
P3	0	0	0	0
P4	0	0	0	0

Need Matrix (C-A)				
	A	B	C	D
P0	0	0	0	0
P1	0	0	0	0
P2	0	0	0	0
P3	0	0	0	0
P4	0	0	0	0

Available Matrix			
7	3	4	2

=> Yes, The system is in Safe State. Safe sequence is: P3->P0->P1->P2->P4.

=> Yes, If a request comes from P1 of (0,0,1,0) we can immediately grant the request. And after granting this process system is still in safe state and the Safe Sequence is: P1->P0->P3->P2->P4

Q21. Consider the following system. Apply Bankers Algorithm.

Process Name	**Allocated**			**Maximum**			**Available**		
	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			

Answer the following questions:

1. What is the content of the Need matrix?
2. Is the system in safe state? If yes, kindly find the safe sequence.
3. If a request came from P1 of (1,0,2), can we immediately grant the request? If yes then still the system is safe?

Ans.

Need Matrix

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

->**P1** leads to complete

Allocated Matrix

	A	B	C
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Claim/Maximum

	A	B	C
P0	7	5	3
P1	0	0	0
P2	9	0	2
P3	2	2	2
P4	4	3	3

Need Matrix(C-A)

	A	B	C
P0	7	4	3
P1	0	0	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Available

5	3	2

-> **P4** leads to complete

Allocated Matrix

	A	B	C
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	0

Claim/ Maximum

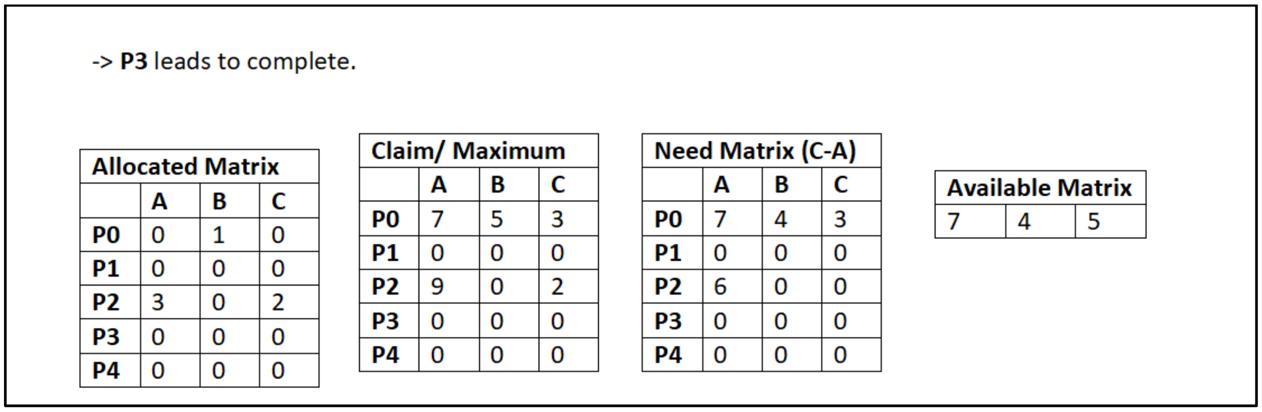
	A	B	C
P0	7	5	3
P1	0	0	0
P2	9	0	2
P3	2	2	2
P4	0	0	0

Need Matrix (C-A)

	A	B	C
P0	7	4	3
P1	0	0	0
P2	6	0	0
P3	0	1	1
P4	0	0	0

Available Matrix

5	3	4



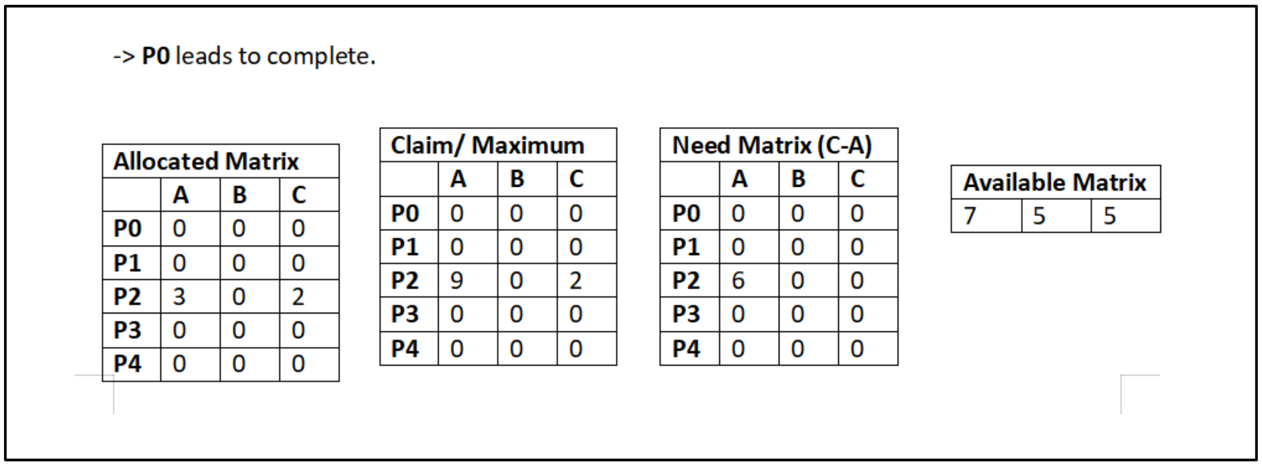
-> **P3** leads to complete.

Allocated Matrix			
	A	B	C
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	0	0	0
P4	0	0	0

Claim/ Maximum			
	A	B	C
P0	7	5	3
P1	0	0	0
P2	9	0	2
P3	0	0	0
P4	0	0	0

Need Matrix (C-A)			
	A	B	C
P0	7	4	3
P1	0	0	0
P2	6	0	0
P3	0	0	0
P4	0	0	0

Available Matrix		
7	4	5



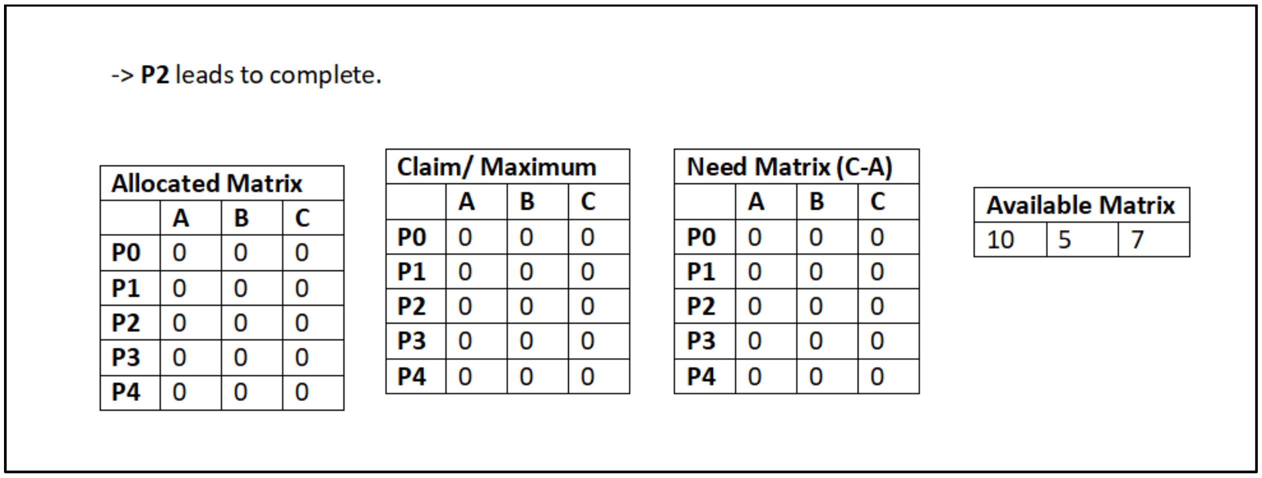
-> **P0** leads to complete.

Allocated Matrix			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	3	0	2
P3	0	0	0
P4	0	0	0

Claim/ Maximum			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	9	0	2
P3	0	0	0
P4	0	0	0

Need Matrix (C-A)			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	6	0	0
P3	0	0	0
P4	0	0	0

Available Matrix		
7	5	5



-> **P2** leads to complete.

Allocated Matrix			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

Claim/ Maximum			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

Need Matrix (C-A)			
	A	B	C
P0	0	0	0
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

Available Matrix		
10	5	7

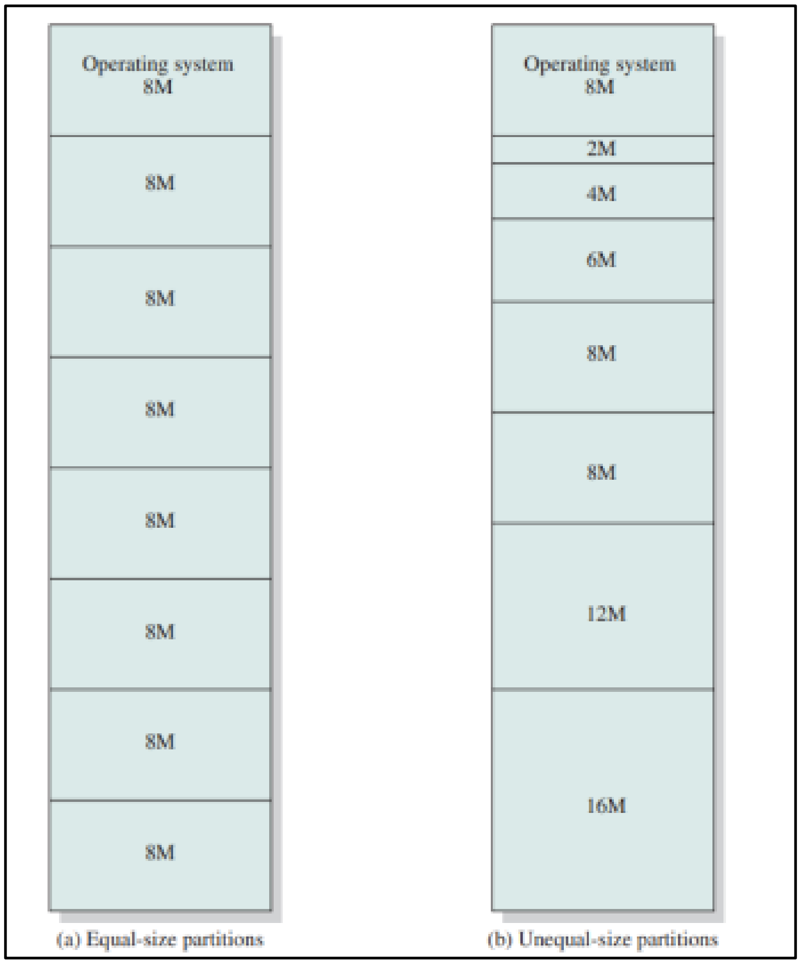
=> Yes, The system is in Safe State. Safe sequence is: P1->P4->P3->P0->P2.

=> Yes, If a request comes from P1 of (1,0,2) we can immediately grant the request. And after granting this process, the system is still in safe state and the Safe Sequence is: P1->P4-> P3->P0->P2.

Q22. What is a fixed partitioning and dynamic partitioning scheme in the context of memory management? Also discuss the advantages and disadvantages of both in detail.

Ans. **Fixed partitioning:** OS occupies some fixed portion of main memory and that the rest of main memory is available for use by multiple processes.

• The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries.
• In this partitioning there are two ways to partition memory, one is Equal-size partitions and second is Unequal-size partitions.



- Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. In our example, there may be a program whose length is less than 2 Mb; yet it occupies an 8-Mb partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.
- With equal-size partitions, the placement of processes in memory is trivial. As long as there is any available partition, a process can be loaded into that partition. Because all partitions are of equal size, it does not matter which partition is used. If all partitions are occupied with processes that are not ready to run, then one of these processes must be swapped out to make room for a new process. Which one to swap out is a scheduling decision; this topic is explored in Part Four.
- With unequal-size partitions, there are two possible ways to assign processes to partitions. The simplest way is to assign each process to the smallest partition within which it will fit. 1 In this case, a scheduling queue is needed for each partition, to hold swapped-out processes destined for that partition. The advantage of this approach is that processes are always assigned in such a way as to minimize wasted memory within a partition.

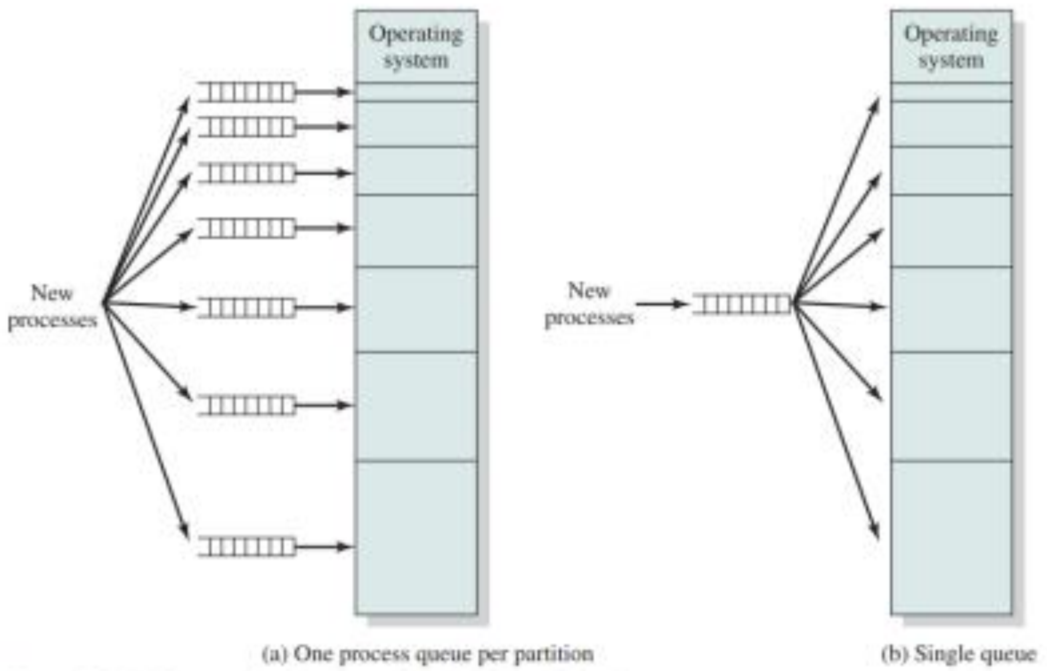


Figure 7.3 **Memory Assignment for Fixed Partitioning**

- This technique seems optimum from the point of view of an individual partition, it is not optimum from the point of view of the system as a whole. For example, consider a case in which there are no processes with a size between 12 and 16M at a certain point in time. In that case, the 16M partition will remain unused, even though some smaller process could have been assigned to it. Thus, a preferable approach would be to employ a single queue for all processes. When it is time to load a process into main memory, the smallest available partition that will hold the process is selected. If all partitions are occupied, then a swapping decision must be made.

- The use of fixed partitioning is almost unknown today. One example of a successful operating system that did use this technique was an early IBM mainframe operating system, OS/MFT (Multiprogramming with a Fixed Number of Tasks). a partition (internal fragmentation).

➢ **Advantages :-**

- **Simplicity :-** Fixed-size partitioning is a straightforward and easy-to-implement memory management scheme.

- **Prevention of fragmentation :-** Fixed-size partitions help prevent external fragmentation, as each partition has a fixed size.

- **Predictability :-** The fixed-size nature of partitions makes it easier to predict and allocate memory requirements for processes.

- **Efficient Allocation :-** Allocating memory to processes is generally faster in fixed-size partitioning systems because the operating system doesn't need to search for appropriate-sized holes in memory.

➢ **Disadvantages :-**

- **Internal Fragmentation :-** Each partition is of a fixed size, if a process doesn't perfectly fit into a partition, there will be wasted space within that partition. This inefficient use of memory is known as internal fragmentation.

- **Inflexibility :-** Fixed-size partitioning can be inflexible when dealing with processes of varying sizes. Small processes may waste space in larger partitions, and larger processes may not fit into smaller partitions, leading to inefficient use of memory.

- **Limited Utilization :-** The fixed-size nature might lead to underutilization of memory, especially if processes are smaller than the allocated partition size. This could result in a waste of memory resources.

- **Not suitable for multiprogramming environments :-** In multiprogramming environments where multiple processes run concurrently, fixed-size partitioning may not be the most efficient method. ✓ It may lead to suboptimal utilization of resources and may not adapt well to the dynamic memory requirements of different processes.

❖ **Dynamic Partitioning :-**

- To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. Again, this approach has been supplanted by more sophisticated memory management techniques. An important operating system that used this technique was IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks).
- With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

• An example, using 64 Mb of main memory, is shown in Figure 7.4 . Initially, main memory is empty, except for the OS (a). The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The operating system swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (g) and swaps process 2 back in (h).

- As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation.
- One technique for overcoming external fragmentation is compaction: From time to time, the OS shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure 7.4h, compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process.
- The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability.

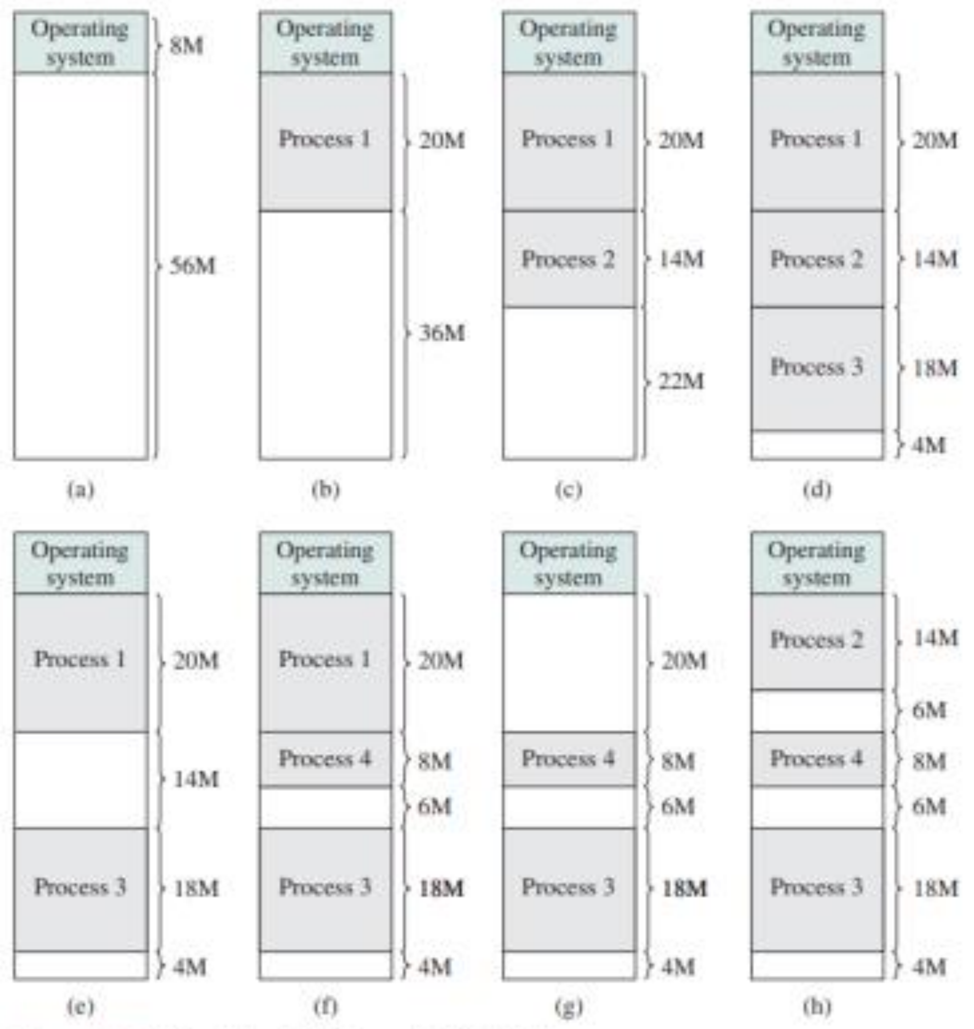


Figure 7.4 The Effect of Dynamic Partitioning

➢ **Advantages :-**

• **Flexibility :-** Dynamic partitions allow for more efficient use of memory as compared to fixed partitions.

• **Optimal memory utilization :-** Dynamic partitioning allows for better memory utilization since the size of allocated memory can match the actual needs of a process, minimizing wasted memory.

• **Support for varying process sizes :-** Processes in a system often have varying memory requirements. Dynamic partitions can accommodate processes of different sizes, adapting to their specific needs.

• **Reduced fragmentation :-** Dynamic partitioning can help reduce fragmentation compared to fixed partitioning schemes. It enables more efficient use of memory by allocating only the necessary amount of memory for each process.

➢ **Disadvantages :-**

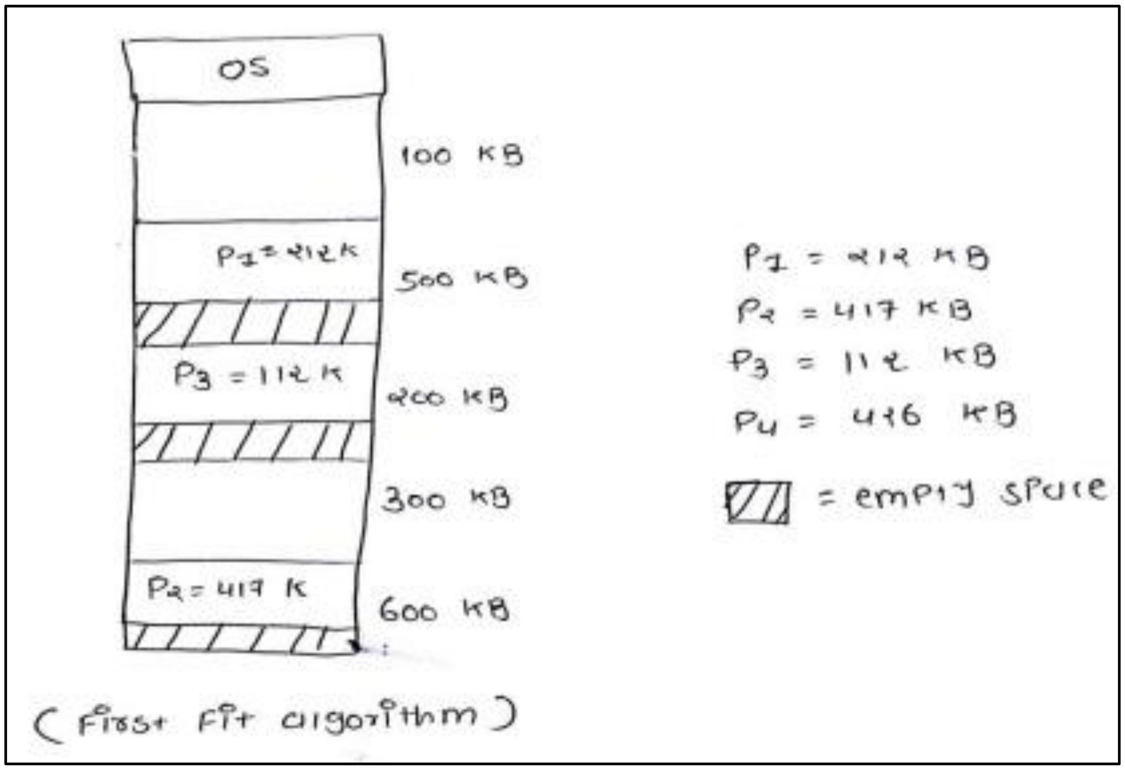
• **Fragmentation :-** External fragmentation can still occur in dynamic partitioning systems. This happens when the free memory is scattered in small blocks, making it challenging to allocate contiguous space for a larger process.

• **Compaction overhead :-** To deal with external fragmentation, compaction may be necessary to rearrange memory and create larger contiguous blocks. Compaction introduces additional overhead and may require temporarily stopping processes.

• **Complexity :-** Dynamic partitioning algorithms can be more complex than fixed partitioning schemes, leading to increased system complexity and potential for bugs or inefficiencies in memory management.

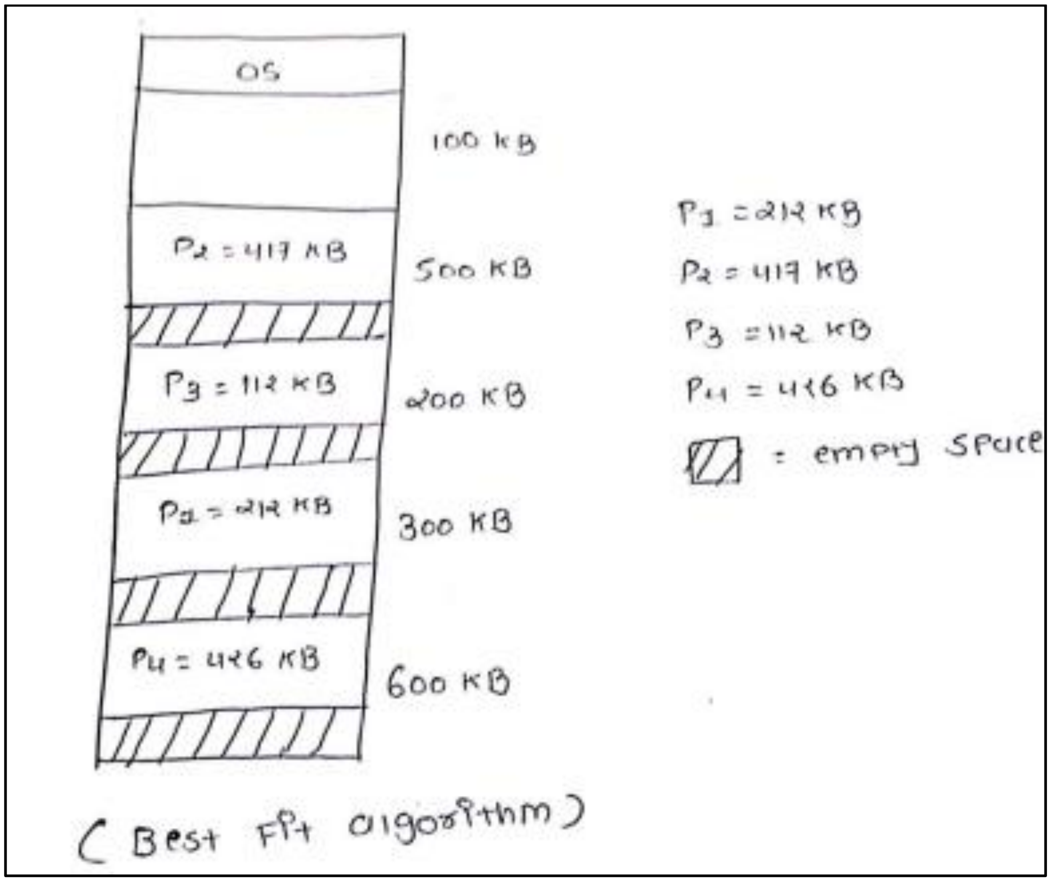
Q23. Given Memory partitions of 100KB, 500KB, 200KB, 300KB and 600KB size (in that order), how would each of the first fit, best fit and worst fit algorithms places processes of 212KB, 417KB, 112KB and 426KB (in that order)? Which algorithm makes the most efficient use of memory? Justify your answer.

Ans. **First fit :-** In the first fit, approach is to allocate the first free partition or block large enough which can accommodate the process.
• It finishes after finding the first suitable free partition.



-> In the First Fit algorithm, Process 4 is not assigned to memory because although there are available empty blocks, contiguous memory is not available for Process 4. As a result, despite the presence of space, Process 4 remains in a waiting state. Fastest algorithm because it searches as little as possible.

Best fit : Best fit deals with allocating the smallest free partition which meets the requirement of the requesting process.
• This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate.
• It then tries to find a hole which is close to the actual process size needed.



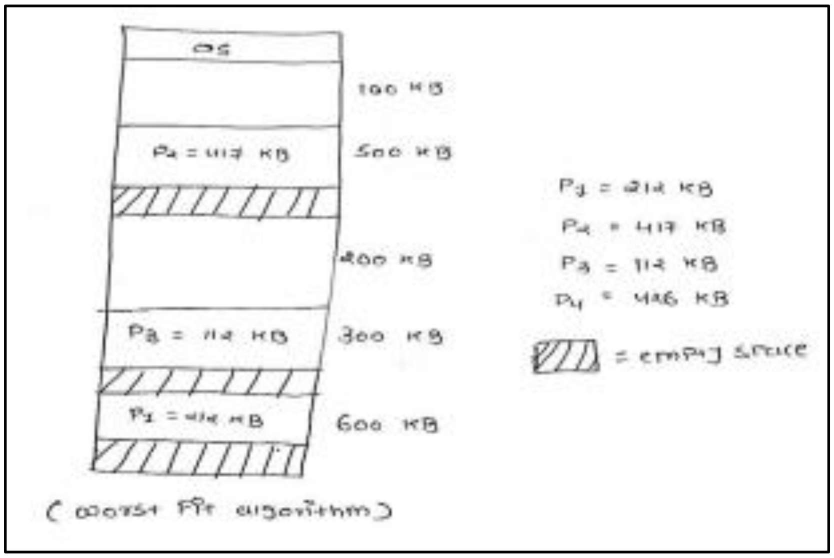
- The Best Fit algorithm evaluates all the available memory partitions by considering both the size of the processes and their proximity to the dimensions of each partition. In the above example, each of the processes Process 1, Process 2, Process 3, and Process 4 is efficiently assigned to memory.
- This approach focuses on efficiency by choosing the smallest partition that can accommodate each process, ultimately optimizing memory usage and reducing fragmentation.
- The Best Fit algorithm ensures a thoughtful allocation strategy, making the most effective use of available memory.

- Memory utilization is much better than first fit as it searches the smallest free partition first available.

Worst fit: The Worst Fit algorithm is a memory allocation algorithm that selects the largest available partition to allocate a process.

• In other words, when a new process needs to be placed into memory, the Worst Fit algorithm searches for the largest available block of memory and assigns the process to that block.

• In the Worst Fit algorithm, Process 4 cannot be assigned to memory because there isn't a continuous block of memory that fits its requirements. This algorithm gives priority to allocating processes to the largest available memory partitions. Unfortunately, in the case of Process 4, the largest partition doesn't offer a contiguous space.



-> In terms of efficiency, the Best Fit algorithm generally makes the most efficient use of memory. This is because it selects the partition that is closest in size to the process, minimizing wasted memory. In this example, the Best Fit algorithm results in the least amount of remaining memory after all processes are allocated. The First Fit and Worst Fit algorithms may lead to more wasted space, especially if larger processes are allocated first, leaving smaller partitions that cannot be efficiently used later.

Q24. How does external fragmentation and internal fragmentation differ?

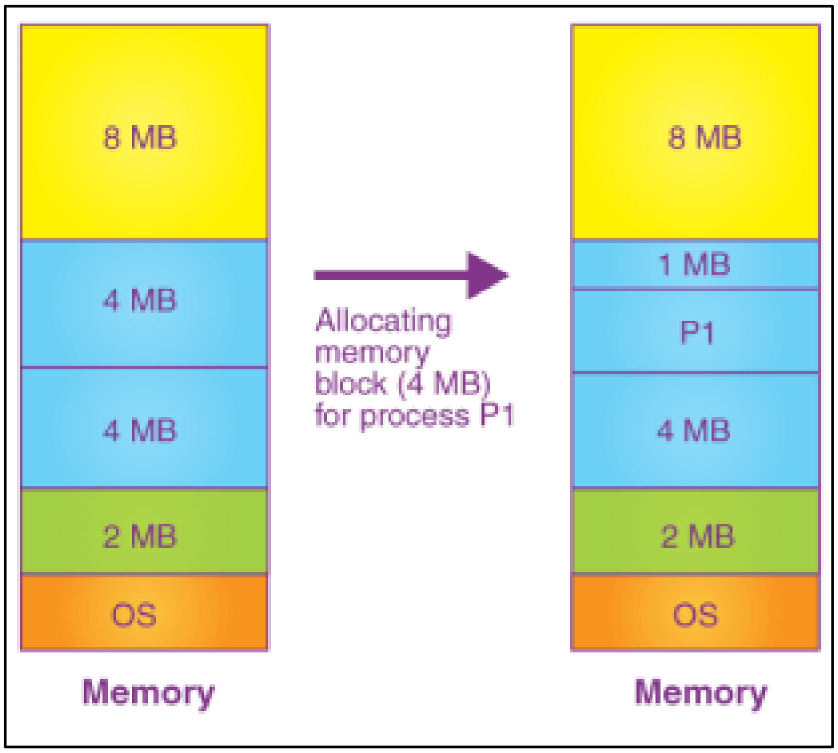
Explain the difference between both of them with an example.

Ans.

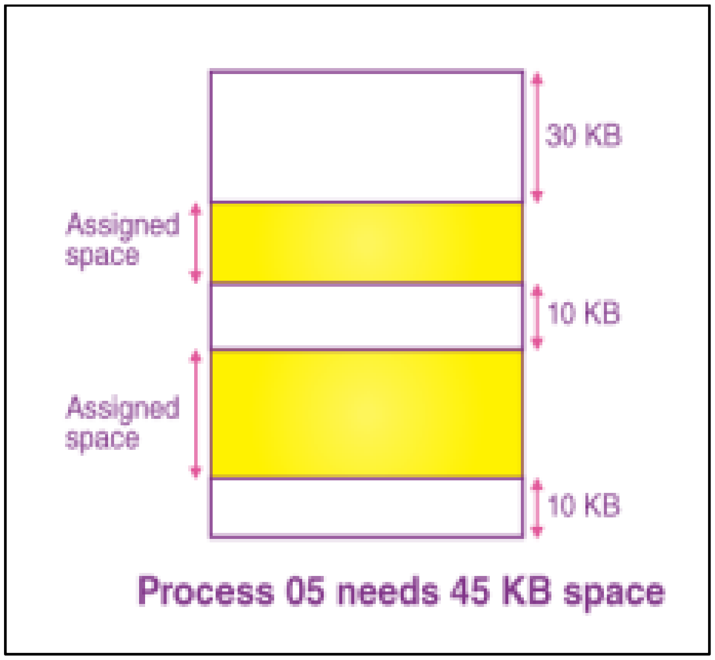
S.NO	Internal Fragmentation	External Fragmentation
1	When there is a difference between required memory space vs allocated memory space, the problem is termed as Internal Fragmentation.	When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as External Fragmentation.
2	Internal fragmentation happens when the method or process is smaller than the memory.	External fragmentation happens when the method or process is removed.
3	The solution of internal fragmentation is the best-fit block.	The solution to external fragmentation is compaction and paging.
4	Internal fragmentation occurs when memory is divided into fixed-sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
5	Internal Fragmentation occurs when **Paging** is employed.	External Fragmentation occurs when **Segmentation** is employed.

Example:

Internal Fragmentation: Suppose a P1 process of 3MB size arrives, and it is given a 4MB memory block. Thus, as a result, the 1MB free space in this very block stays unused, and it can't be used for the allocation of memory to another process. It is called internal fragmentation.

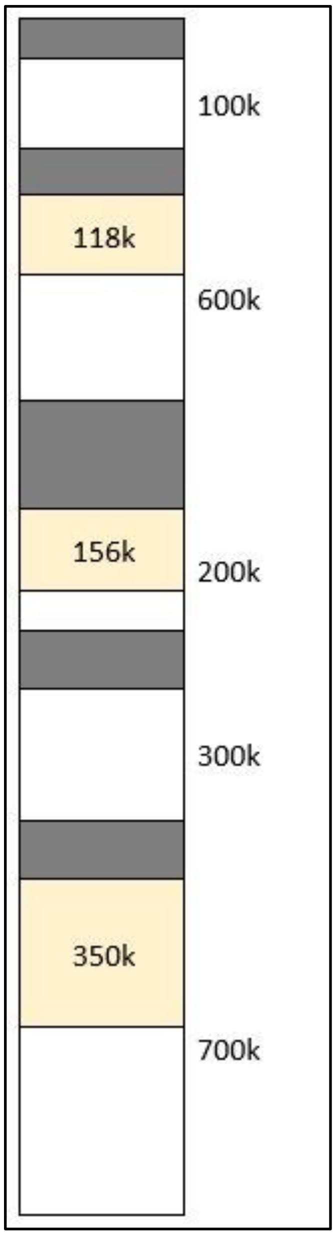


External Fragmentation: In the diagram, there is sufficient space (of 50 KB) to run a process 05 (requirement of 45 KB), but the memory here is not contiguous. Thus, one can use compaction, segmentation, and paging in order to use the free space for the execution of a process. We can not assign the process 05 because memory is not contiguous.

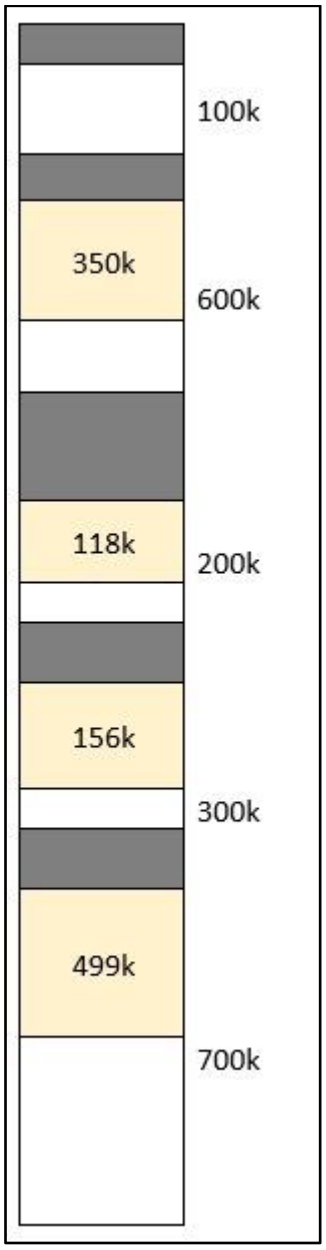


Q25. Given a dynamic memory partition setup with current holes of (A) 100K (B) 600K (C) 200K (D) 300K and (E) 700K (in order), show how each of First fit, Best Fit and Worst Fit algorithms would place processes of size 118K, 350K, 156K and 499K (in order)? Indicate for each algorithm the total space left, the smallest available hole, the largest available hole and processes (if any) are unable to place. Which algorithm makes the most efficient use of the memory?

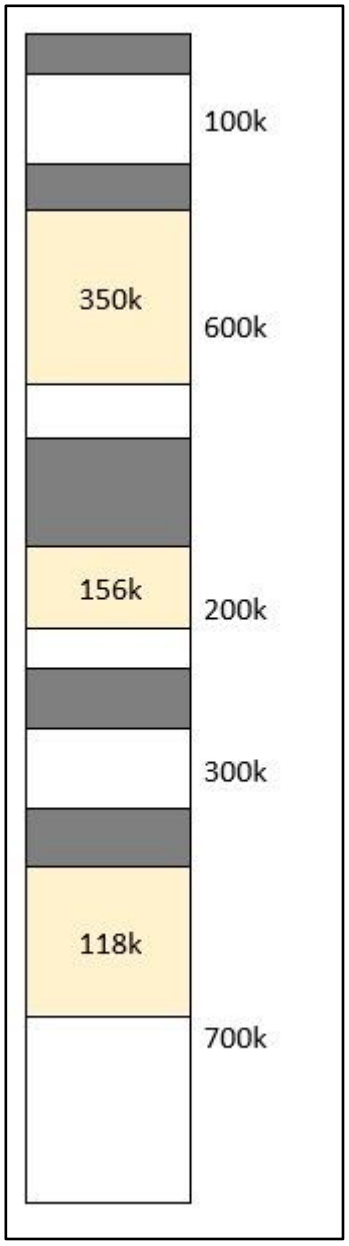
Ans.



First Fit



Best Fit



Worst Fit

->Analysis

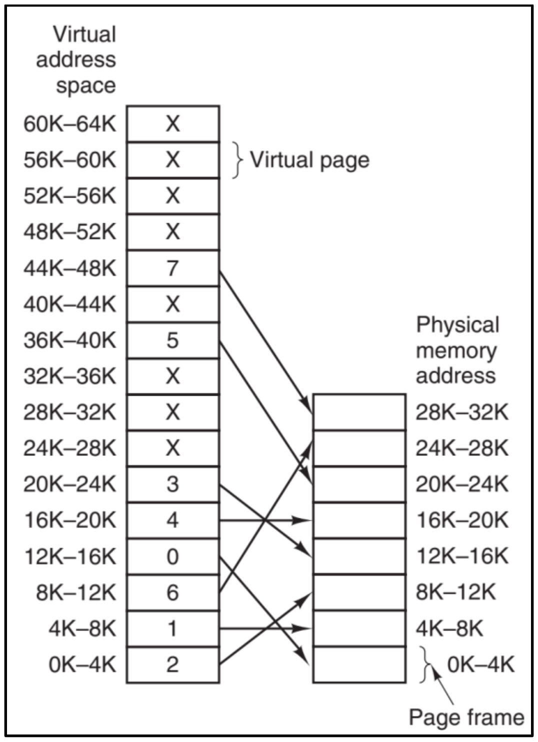
- **First Fit:** In the first fit ,the process is fit into the first available block. From a total of 4 processes,the first fit was able to only fit the first three processes.So,the fourth process did not get the memory space.In total,the fragmentation caused by the first fit was 876k. The total unused space was 100k+300k=400k.

- **Best Fit:** The best fit finds the smallest block which can fit the process. From a total of 4 processes,best fit was able to fit all the processes.The best fit has the lowest amount of fragmentation.In total,the fragmentation caused by best fit was 677k.The total unused space was 100k.

- **Worst Fit:** The worst fit finds the largest block which can fit the process. From total 4 processes, worst fit was able to only fit three processes.So,the fourth process did not get the memory space.The worst fit has the highest amount of fragmentation.In total,the fragmentation caused by best fit was 876k.The total unused space was 100k+300k=400k.

Q26. How is the logical address converted to the physical address in the basic method of paging? Justify it with an appropriate diagram?

Ans. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU (Memory Management Unit)** that maps the virtual addresses onto the physical memory addresses.



-> The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

-> The notation in Figure is as follows. The range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095. The range 4K–8K refers to addresses 4096 to 8191, and so on. Each page contains exactly 4096 addresses starting at a multiple of 4096 and ending one shy of a multiple of 4096.

->When the program tries to access address 0, for example, using the instruction MOV REG,0 virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

->Similarly, the instruction MOV REG,8192 is effectively transformed into MOV REG,24576 because virtual address 8192 (in virtual page 2) is mapped onto 24576 (in physical page frame 6). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

Q27. Consider a process has 8 virtual pages and is assigned a fixed allocation of three-page frames in main memory. The pages are referenced in the following order.
0 3 2 1 4 5 3 1 6 7 2 4 6 5 1 3 2 0 3 7 5 3 1 2.
If the main memory frames are initially empty, determine the number of page faults that will occur with the following page replacement algorithms.

-> FIFO (First in First out)
-> LRU (Least recently used)
-> Optimal page replacement algorithm.

Illustrate the procedure using a suitable diagram.

Ans.

<u>FIFO (First In First Out):</u>

0	3	2	1	4	5	3	1	6	7	2	4	6	5	1	3	2	0	3	7	5	3	1	2
0	0	0	1	1	1	3	3	3	7	7	7	6	6	6	3	3	3	3	7	7	7	1	1
	3	3	3	4	4	4	1	1	1	2	2	2	5	5	5	2	2	2	2	5	5	5	2
		2	2	2	5	5	5	6	6	6	4	4	4	1	1	1	0	0	0	0	3	3	3
f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f		f	f	f	f	f

Page faults: 23

<u>LRU (Least Recently used):</u>

0	3	2	1	4	5	3	1	6	7	2	4	6	5	1	3	2	0	3	7	5	3	1	2
0	0	0	1	1	1	3	3	3	7	7	7	6	6	6	3	3	3	3	3	3	3	3	3
	3	3	3	4	4	4	1	1	1	2	2	2	5	5	5	2	2	2	7	7	7	1	1
		2	2	2	5	5	5	6	6	6	4	4	4	1	1	1	0	0	0	5	5	5	2
f	f	f	f	f	f	f	f	f	F	f	f	f	f	f	f	f	f		f	f		f	f

Page faults: 22

<u>OPT (Optimal Page Replacement):</u>

0	3	2	1	4	5	3	1	6	7	2	4	6	5	1	3	2	0	3	7	5	3	1	2
0	0	0	1	1	1	1	1	1	7	2	4	4	4	4	3	3	3	3	3	3	3	3	3
	3	3	3	3	3	3	3	6	6	6	6	6	6	1	1	2	0	0	7	7	7	7	7
		2	2	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1	2
f	f	f	f	f	f			f	f	f	f			f	f	f	f		f			f	f

Page faults: 17

Q28: Consider the following page trace.
1,2,3,4,2,1,5,6,2,1,2,3,4,6,3,2,1,2,3,4,5,6
Calculate the number of page faults that would occur for First in First Out, Optimal and Least recently used page replacement algorithm with three and four page frames. Initially all the frames are empty.

Ans.

FIFO (First In First Out) with 3 page frames:

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	4	4	4	4	6	6	6	6	3	3	3	3	2	2	2	2	4	4	4
	2	2	2	2	1	1	1	2	2	2	2	4	4	4	4	1	1	1	1	5	5
		3	3	3	3	5	5	5	1	1	1	1	6	6	6	6	6	3	3	3	6
f	f	f	f		f	f	f	f	f		f	f	f		f	f		f	f	f	f

Page faults: 18

FIFO (First In First Out) with 4 page frames:

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	1	1	1	5	5	5	5	5	3	3	3	3	3	1	1	1	1	1	6
	2	2	2	2	2	2	6	6	6	6	6	4	4	4	4	4	4	3	3	3	3
		3	3	3	3	3	3	2	2	2	2	2	6	6	6	6	6	6	4	4	4
			4	4	4	4	4	4	1	1	1	1	1	1	2	2	2	2	2	5	5
f	f	f	f			f	f	f	f		f	f	f		f	f		f	f	f	f

Page faults: 17

OPT (Optimal Page Replacement): with 3 page frames:

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3	6
	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	1	1	1	4	4	4
		3	4	4	4	5	6	6	6	6	6	6	6	6	2	2	2	2	2	5	5
f	f	f	f			f	f				f	f			f	f			f	f	f

Page faults: 13

OPT (Optimal Page Replacement): with 4 page frames:

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	1	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4	4	4	4
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	6
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
			4	4	4	5	6	6	6	6	6	6	6	6	6	1	1	1	1	5	5
f	f	f	f			f	f					f				f				f	f

Page faults: 10

LRU (Least Recently used): with 3 page frames:

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	4	4	4	5	5	5	1	1	1	4	4	4	2	2	2	2	2	5	5
	2	2	2	2	2	2	6	6	6	6	3	3	3	3	3	3	3	3	3	3	6
		3	3	3	1	1	1	2	2	2	2	2	6	6	6	1	1	1	4	4	4
f	f	f	f		f	f	f	f	f		f	f	f		f	f			f	f	f

Page faults: 17

LRU (Least Recently used): with 4 page frames

1	2	3	4	2	1	5	6	2	1	2	3	4	6	3	2	1	2	3	4	5	6
1	1	1	1	1	1	1	1	1	1	1	1	1	6	6	6	6	6	6	4	4	4
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	6
		3	3	3	3	5	5	5	5	5	3	3	3	3	3	3	3	3	3	3	3
			4	4	4	4	6	6	6	6	6	4	4	4	4	1	1	1	1	5	5
f	f	f	f			f	f				f	f	f			f			f	f	f

Page faults: 13

Q29. Consider the following reference string with four page frames.

1,2,3,4,2,1,5,6,2,1,5,3,4,6,1,2,7,4,8,6,4,2,1,3,6

How many page faults occur for FIFO, optimal and LRU page replacement algorithms?

Ans.

FIFO (First In First Out):

1	2	3	4	2	1	5	6	2	1	5	3	4	6	1	2	7	4	8	6	4	2	1	3	6
1	1	1	1	1	1	5	5	5	5	5	3	3	3	3	3	7	7	7	7	7	7	7	3	3
	2	2	2	2	2	2	6	6	6	6	6	4	4	4	4	4	4	8	8	8	8	8	8	6
		3	3	3	3	3	3	2	2	2	2	2	6	6	6	6	6	6	6	4	4	4	4	4
			4	4	4	4	4	4	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1
f	f	f	f			f	f	f	f		f	f	f		f	f		f		f		f	f	f

Page faults: 18

OPT (Optimal Page Replacement):

1	2	3	4	2	1	5	6	2	1	5	3	4	6	1	2	7	4	8	6	4	2	1	3	6
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7	7	8	8	8	8	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	4	6
			4	4	4	5	5	5	5	5	3	4	4	4	4	4	4	4	4	4	4	4	3	3
f	f	f	f			f	f				f	f				f		f				f	f	f

Page faults: 13

LRU (Least Recently used):

1	2	3	4	2	1	5	6	2	1	5	3	4	6	1	2	7	4	8	6	4	2	1	3	6
1	1	1	1	1	1	1	1	1	1	1	1	1	6	6	6	6	4	4	4	4	4	4	4	6
	2	2	2	2	2	2	2	2	2	2	2	4	4	4	4	7	7	7	7	7	2	2	2	2
		3	3	3	3	5	5	5	5	5	5	5	5	1	1	1	1	8	8	8	8	1	1	1
			4	4	4	4	6	6	6	6	3	3	3	3	2	2	2	2	6	6	6	6	3	3
f	f	f	f			f	f				f	f	f	f	f	f	f	f	f		f	f	f	f

Page faults: 19

Q30. Consider the following set of processes with given CPU burst time and arrival time.

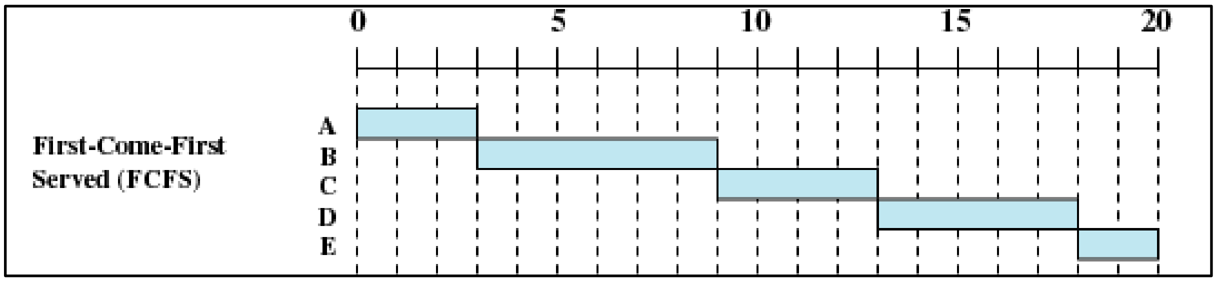
Process	Arrival time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Draw the Gantt charts that illustrate the execution of processes and compute the average waiting time and average turnaround time for the following scheduling algorithms:

1. First Come First Serve (FCFS)
2. Round Robin (Time quantum = 2)
3. Shortest Process Next (SPN)
4. Shortest Remaining Time (SRT)

Ans.

First Come First Serve (FCFS):



Queue	A 1/3	A 2/3	A 3/3	B 1/6	B 2/6	B 3/6	B 4/6	B 5/6	B 6/6	C 1/4	C 2/4	C 3/4	C 4/4	D 1/5	D 2/5	D 3/5	D 4/5	D 5/5	E 1/2	E 2/2

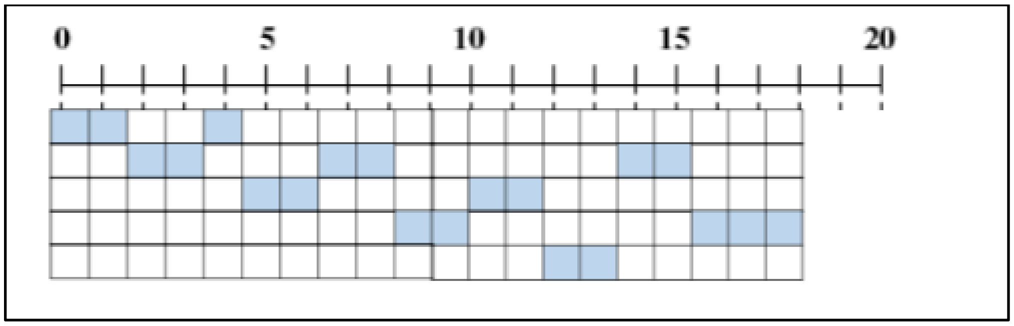
Gantt Chart

A	B	C	D	E

0 3 9 13 18 20

Job	Arrival Time	Service Time	Completion Time	Turnaround Time (CT-AT)	Waiting Time (TAT-ST)
A	0	3	3	3	0
B	2	6	9	7	1
C	4	4	13	9	5
D	6	5	18	12	7
E	8	2	20	12	10
			Average	43 / 5 = 8.6	23 / 5 = 4.6

Round Robin: (Quantum 2)



Queue	A 1/3	A 2/3	B 1/6	B 2/6	A 3/3	C 1/4	C 2/4	B 3/6	B 4/6	D 1/5	D 2/5	C 3/4	C 4/4	E 1/2	E 2/2	B 5/6	B 6/6	D 3/5	D 4/5	D 5/5

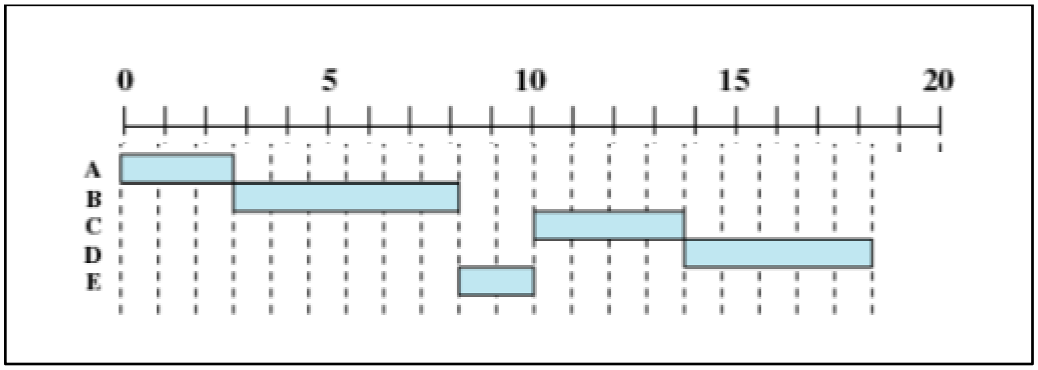
Gantt Chart

A	B	A	C	B	D	C	E	B	D	D	
0	2	4	5	7	9	11	13	15	17	19	20

Job	Arrival Time	Service Time	Completion Time	Turnaround Time (CT-AT)	Waiting Time(TAT-ST)
A	0	3	5	5	2
B	2	6	17	15	9
C	4	4	13	9	5
D	6	5	20	14	9

E	8	2	15	7	5
			Average	50 / 5 = 10	30 / 5 = 6

Shortest Process Next (SPN):



Queue	A 1/3	A 2/3	A 3/3	B 1/6	B 2/6	B 3/6	B 4/6	B 5/6	B 6/6	E 1/2	E 2/2	C 1/4	C 2/4	C 3/4	C 4/4	D 1/5	D 2/5	D 3/5	D 4/5	D 5/5

Gantt Chart

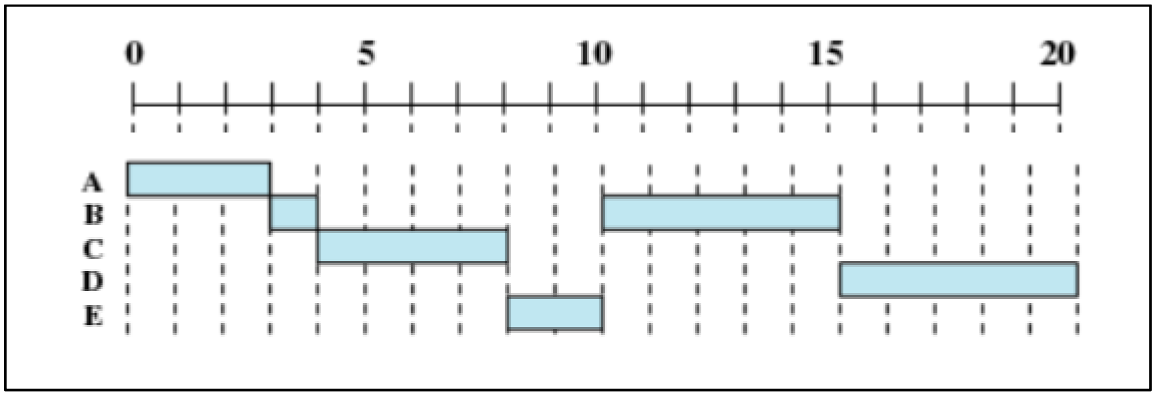
A	B	E	C	D

0 3 9 11 15 20

Job	Arrival Time	Service Time	Completion Time	Turnaround Time (CT-AT)	Waiting Time(TAT-ST)
A	0	3	3	3	0
B	2	6	9	7	1

C	4	4	15	11	7
D	6	5	20	14	9
E	8	2	11	3	1
			Average	38 / 5 = 7.6	18 / 5 = 3.6

Shortest Remaining Time (SRT):



Gantt Chart

A	B	C	E	B	D

0 3 4 8 10 15 20

QUEUE	A 1/3	A 2/3	A 3/3	B 1/6	C 1/4	C 2/4	C 3/4	C 4/4	E 1/2	E 2/2	B 2/6	B 3/6	B 4/6	B 5/6	B 6/6	D 1/5	D 2/5	D 3/5	D 4/5	D 5/5

Job	Arrival Time	Service Time	Completion Time	Turnaround Time (CT-AT)	Waiting Time(TAT-ST)
A	0	3	3	3	0
B	2	6	15	13	7
C	4	4	8	4	0
D	6	5	20	14	9
E	8	2	10	2	0
Average				36 / 5 = 7.2	16 / 5 = 3.2

Q31. Consider the following set of processes with given CPU burst time and arrival time.

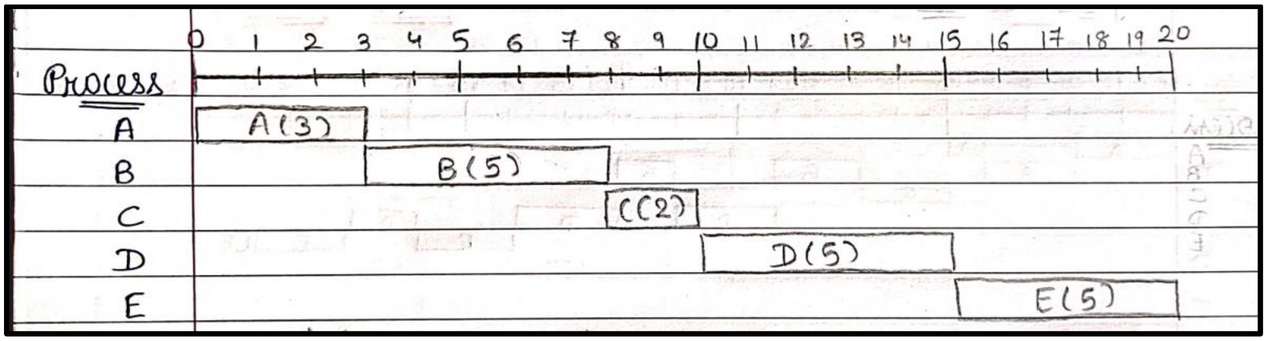
Process Name	**Arrival time**	**Burst Time**
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

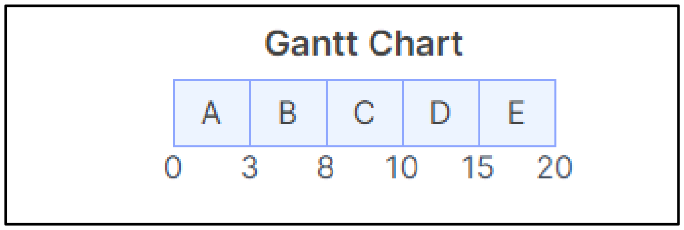
Draw the Gantt charts that illustrate the execution of processes and compute the average waiting time and average turnaround time for the following scheduling algorithms:

1. First Come First Serve (FCFS)
2. Round Robin (Time quantum = 2)
3. Shortest Process Next(SPN)
4. Shortest Remaining Time (SRT)

ANS:

First Come First Serve (FCFS):

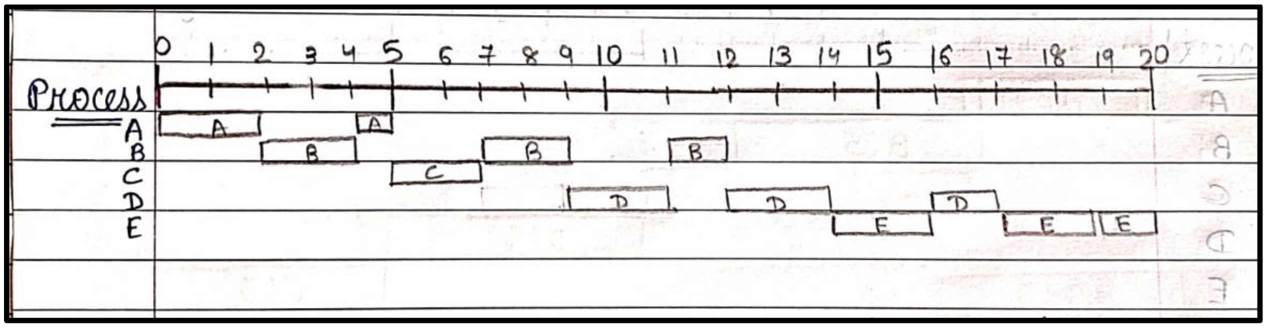


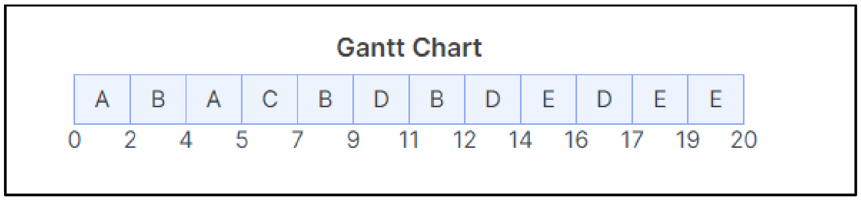


Job	Arrival Time	Service Time	Completion Time	Turnaround Time (CT-AT)	Waiting Time(TAT-ST)
A	0	3	3	3	0

B	1	5	8	7	2
C	3	2	10	7	5
D	9	5	15	6	1
E	12	5	20	8	3
			Average	31 / 5 = 6.2	11 / 5 = 2.3

Round Robin (Time quantum = 2)

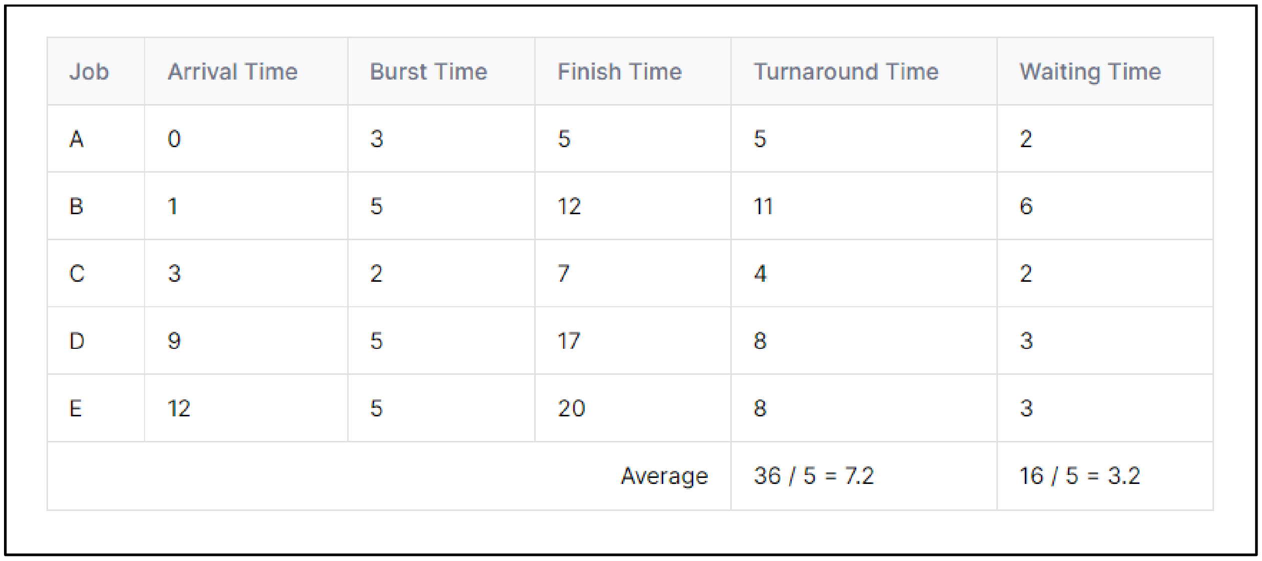




Gantt Chart

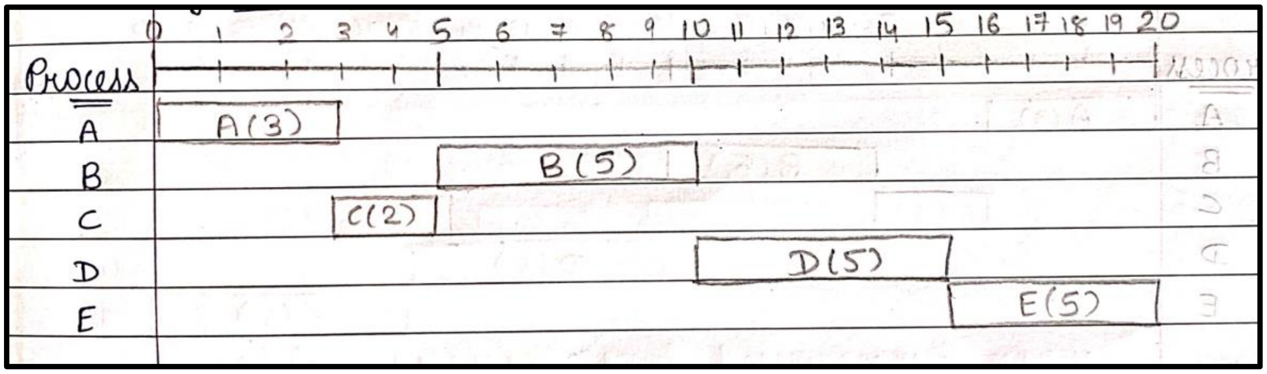
A	B	A	C	B	D	B	D	E	D	E	E

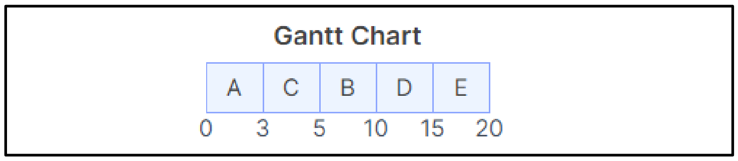
0 2 4 5 7 9 11 12 14 16 17 19 20



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	3	5	5	2
B	1	5	12	11	6
C	3	2	7	4	2
D	9	5	17	8	3
E	12	5	20	8	3
			Average	36 / 5 = 7.2	16 / 5 = 3.2

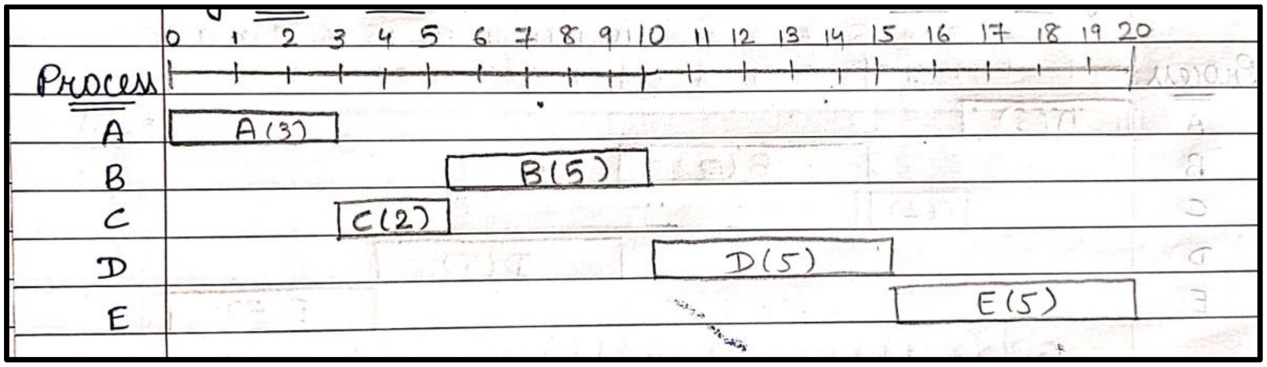
Shortest Process Next(SPN):

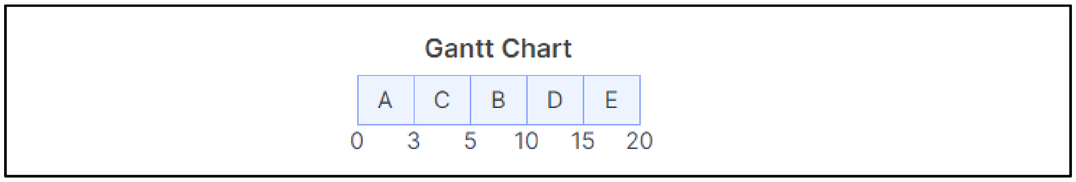


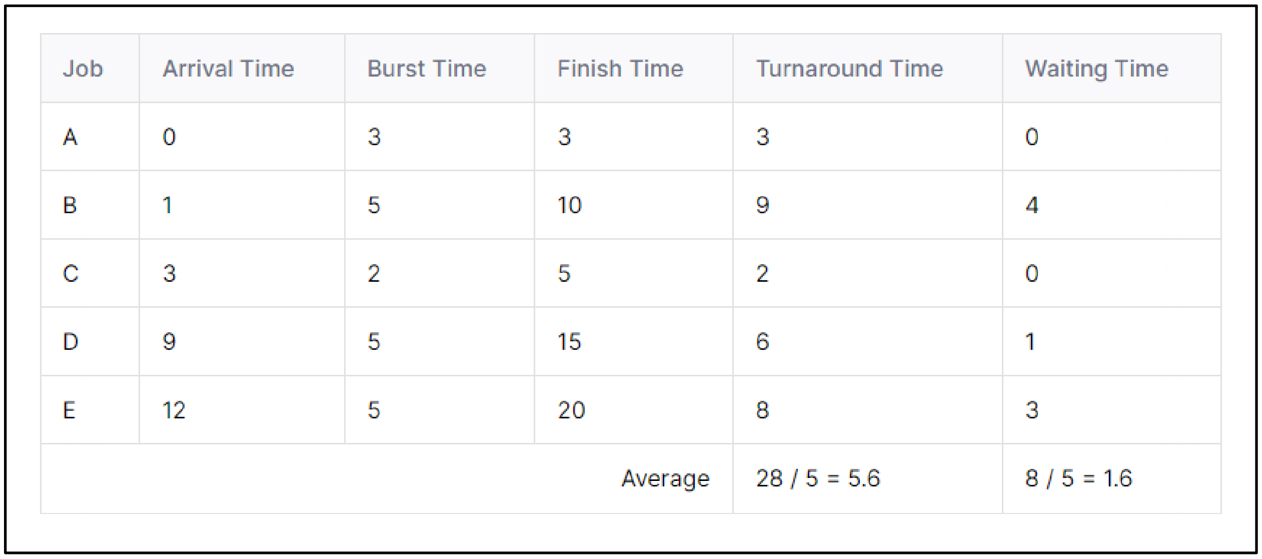


Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	3	3	3	0
B	1	5	10	9	4
C	3	2	5	2	0
D	9	5	15	6	1
E	12	5	20	8	3
			Average	28 / 5 = 5.6	8 / 5 = 1.6

Shortest Remaining Time (SRT):







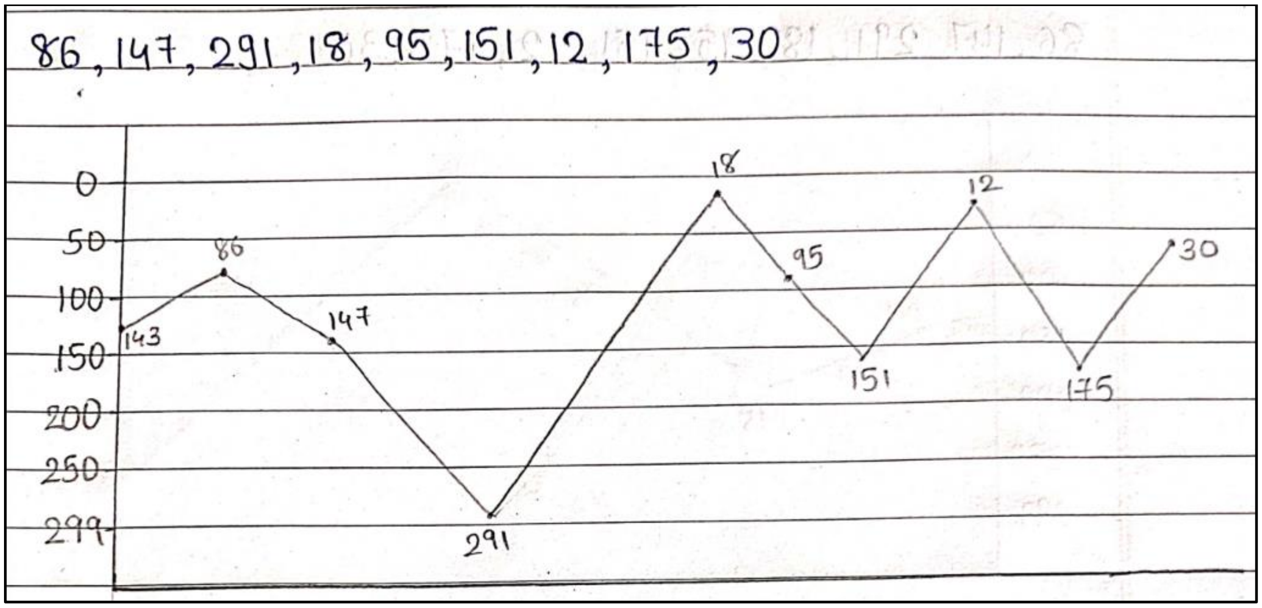
Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	3	3	3	0
B	1	5	10	9	4
C	3	2	5	2	0
D	9	5	15	6	1
E	12	5	20	8	3
			Average	28 / 5 = 5.6	8 / 5 = 1.6

Q32. Suppose that the disk drive has 300 cylinders, numbered 0 to 299. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 15. The queue of pending requests, in FIFO order is 86, 147, 291, 18, 95, 151, 12, 175, and 30. Starting from the current head position, what is the total distance that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms.

1. First Come First Serve.
2. Shortest Seek time First (SSTF)
3. SCAN

Ans.

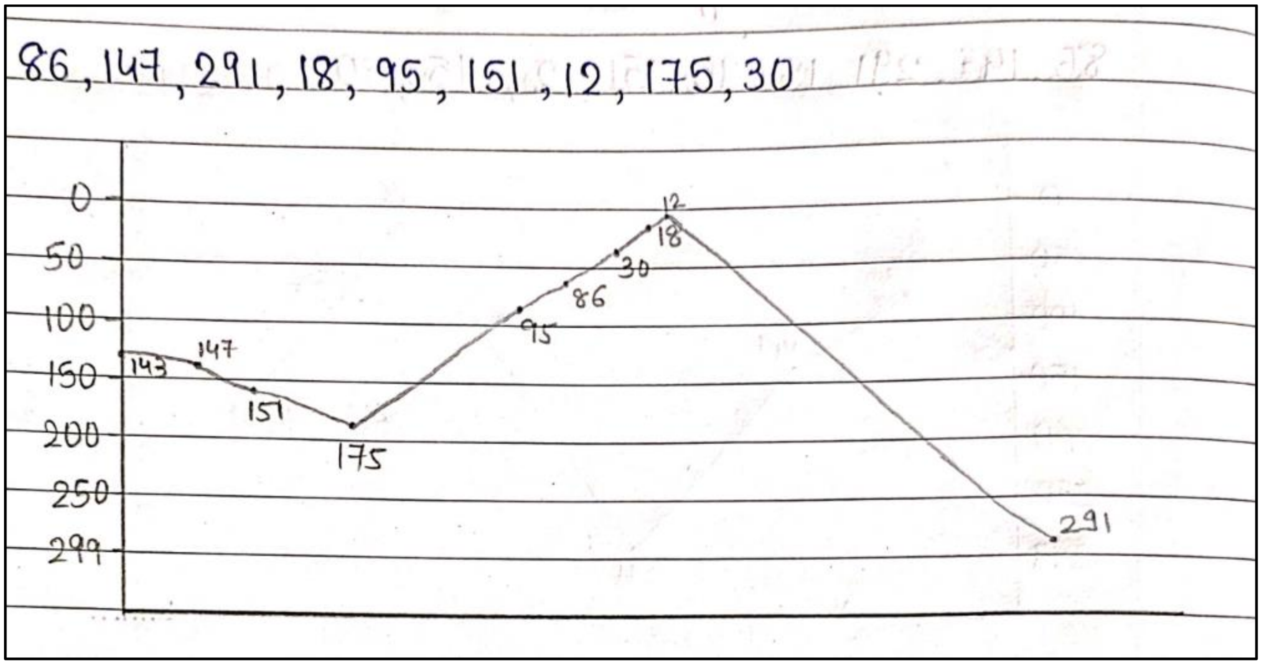
<u>First Come First Serve:</u>



Disk	Traversed

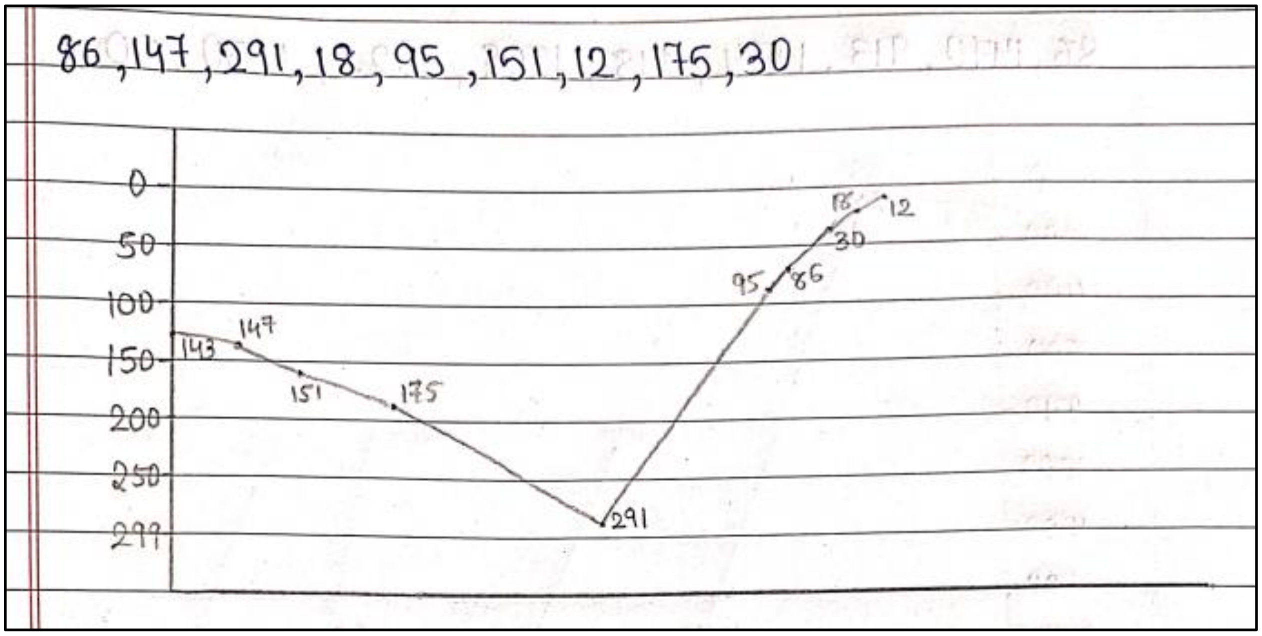
143	
86	57
147	61
291	144
18	273
95	77
151	56
12	139
175	163
30	145
Avg. Seek Length	1115/9 =123.88

Shortest Seek Time First:



Disk	Traversed
147	4
151	4
175	24
95	80
86	9
30	56
18	12
12	6
291	279
Avg. Seek Length	474/9 = 52.67

SCAN:



Disk	Traversed

147	4
151	4
175	24
291	116
95	195
86	9
30	56
18	12
12	6
Avg. Seek Length	426/9 = 47.33

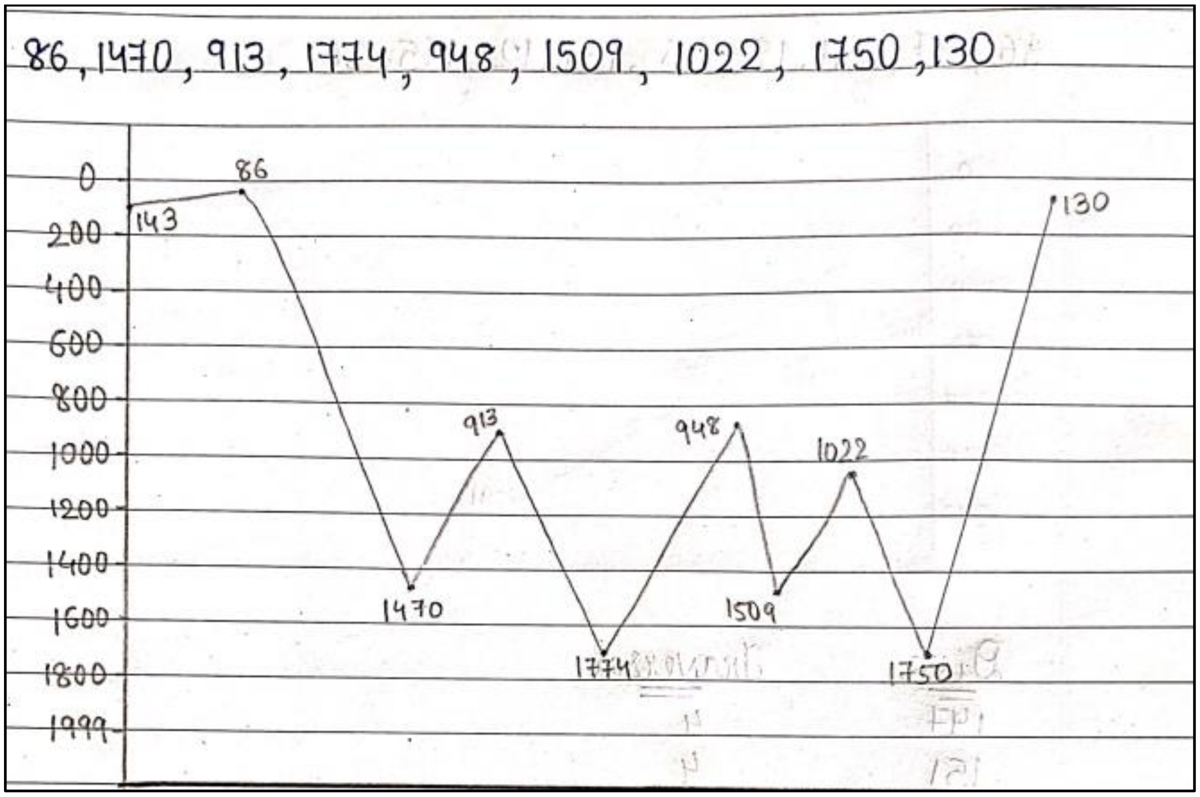
Q33. Suppose that the disk drive has 2000 cylinders, numbered 0 to 1999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder

125. The queue of pending requests in FIFO order is : 86, 1470, 913, 1774, 948, 1509, 1022, 1750 and 130. Starting from the current head position, what is the total distance that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms:

1. FCFS 2. SSTF 3. SCAN

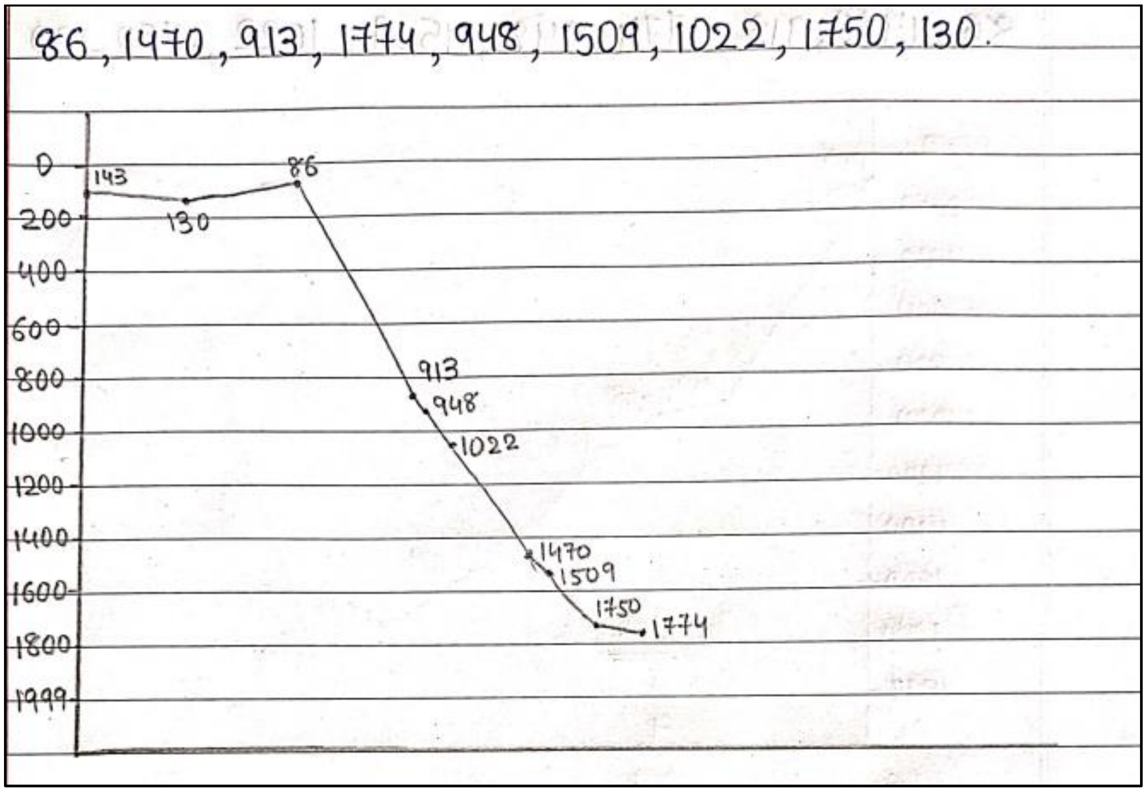
Ans.

First Come First Serve:



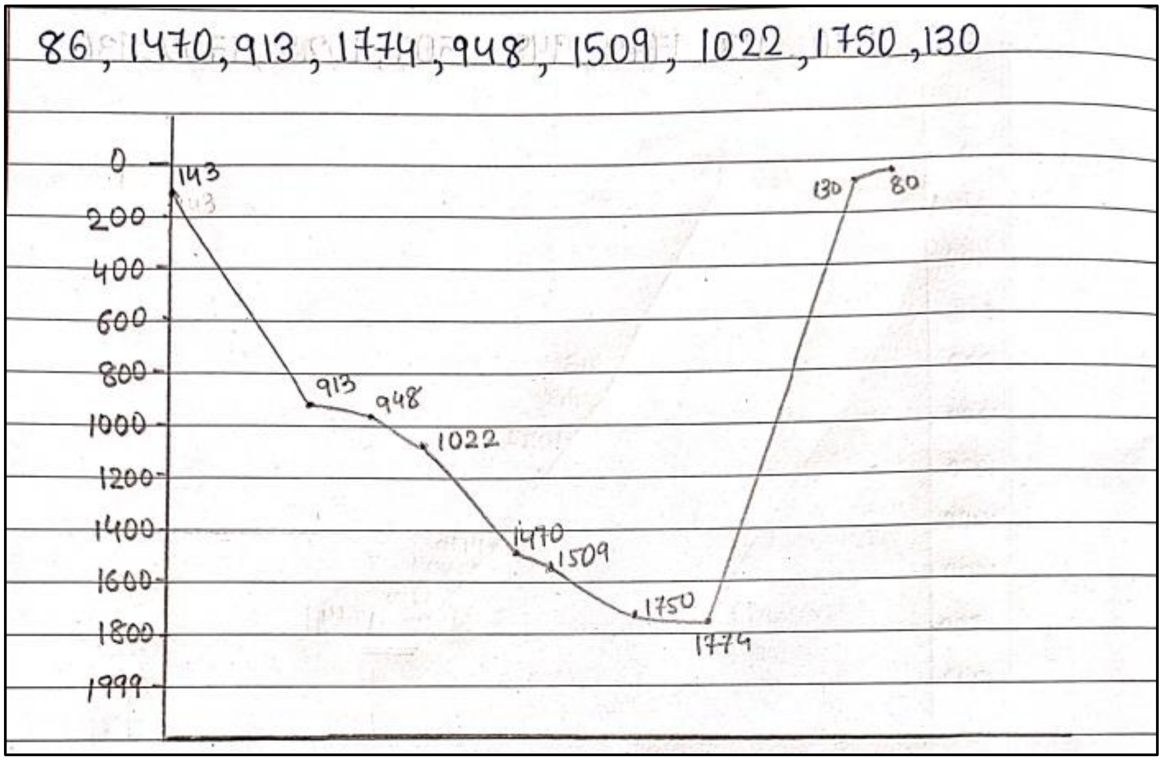
Disk	Traversed
86	57
1470	1384
913	557
1774	861
948	826
1509	561
1022	487
1750	728
130	1620
Avg. Seek Length	7081/9 = 786.78

Shortest Seek Time First:



Disk	Traversed
130	13
86	44
913	827
948	35
1022	74
1470	448
1509	39
1750	241
1774	24
Avg. Seek Length	1745/9 = 193.89

SCAN:

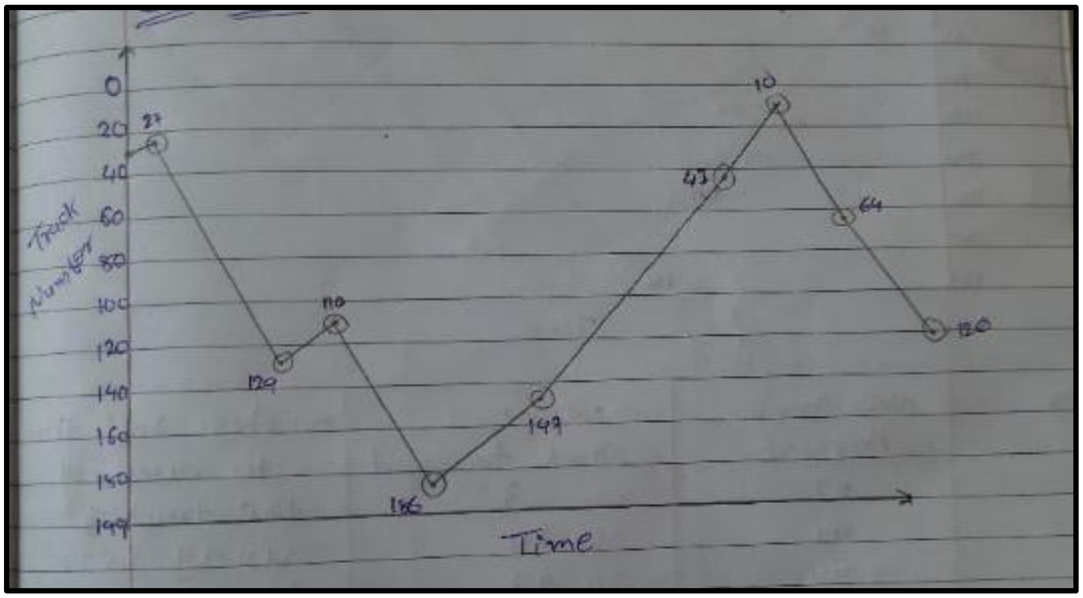


Disk	Traversed
913	770
948	35
1022	74
1470	448
1509	39
1750	241
1774	24
130	1644
80	50
Avg. Seek Length	3325/9 = 369.44

Q34. Consider the following sequence of disk track requests: -27, 129, 110, 186, 147, 41, 10, 64 and 120. Assume that head is initially at track 30. Compute the number of tracks the head traverses using FCFS, SSTF and elevator algorithms.

Ans.

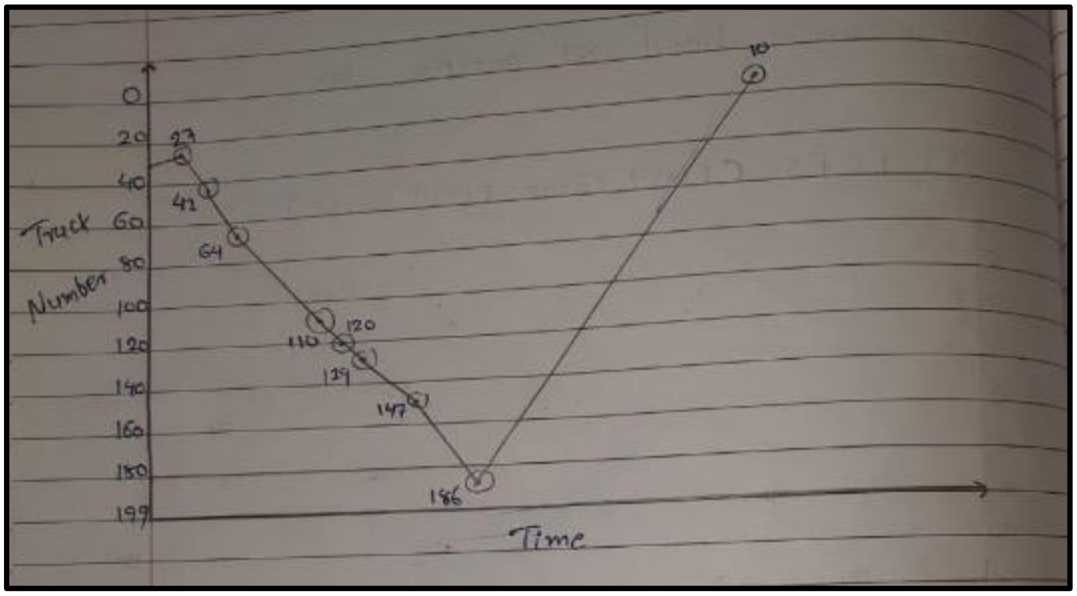
First Come First Serve: 27, 129, 110, 186, 147, 41, 10, 64, 120



Disk	Traversed

27	3
129	102
110	19
186	76
147	39
41	106
10	31
64	54
120	56
Avg. Seek Length	486/9 = 54

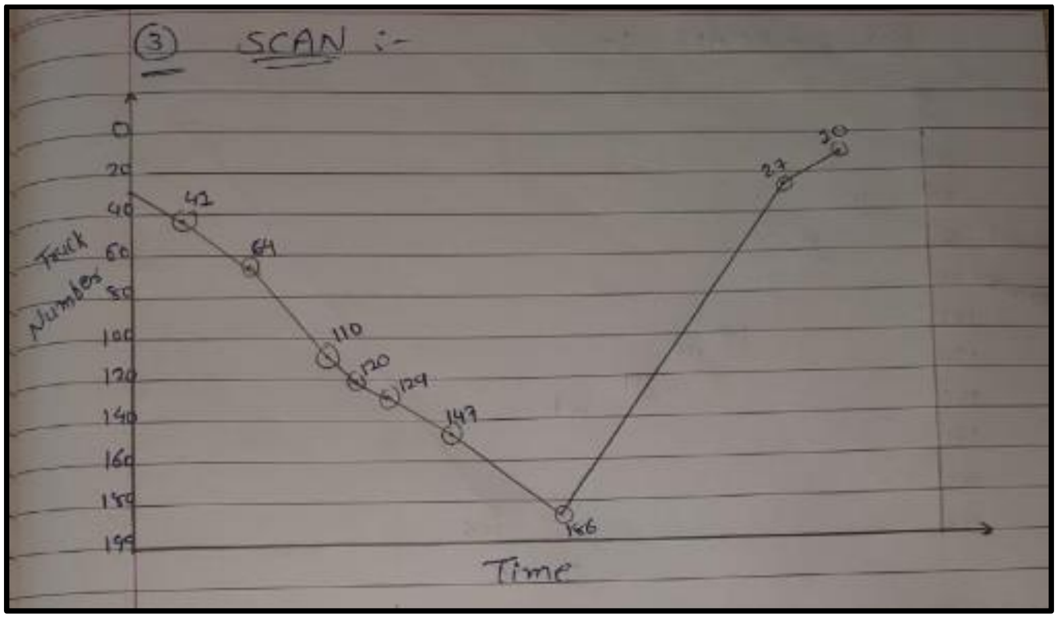
Shortest Seek Time First: 27, 129, 110, 186, 147, 41, 10, 64, 120



Disk	Traversed

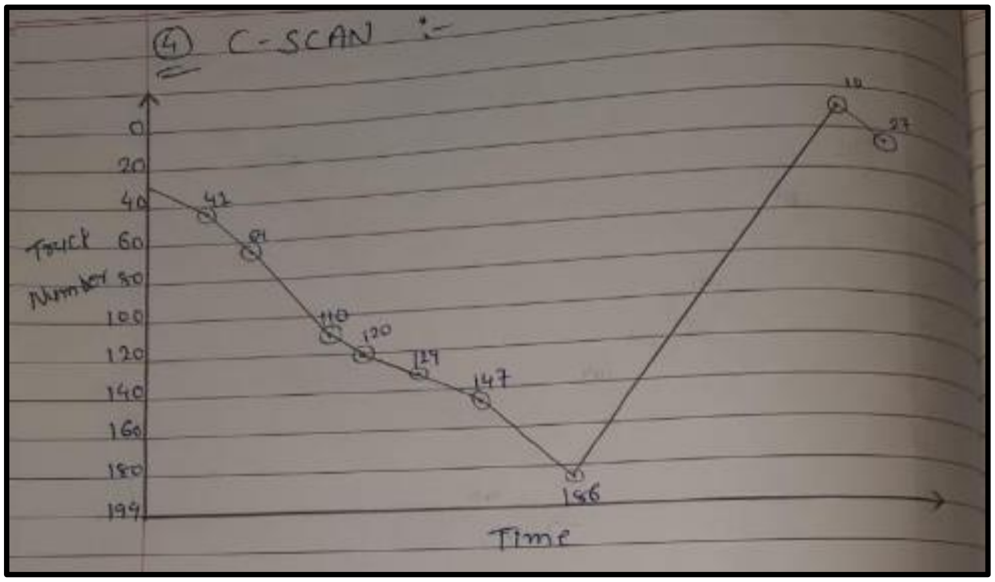
27	3
41	14
64	23
110	46
120	10
129	9
147	18
186	39
10	176
Avg. Seek Length	338/9 = 37.6

SCAN: 27, 129, 110, 186, 147, 41, 10, 64, 120



Disk	Traversed
41	11
64	23
110	46
120	10
129	9
147	18
186	39
27	159
10	17
Avg. Seek Length	332/9 = 36.9

C- SCAN: 27, 129, 110, 186, 147, 41, 10, 64, 120



Disk	Traversed
41	11
64	23
110	46
120	10
129	9
147	18
186	39
10	176
27	17
Avg. Seek Length	349/9 = 38.8

Q35. Briefly explain any four different file organizations.

Ans. The term file organization refers to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy. In choosing a file organization, several criteria are important:

- Short access time
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability

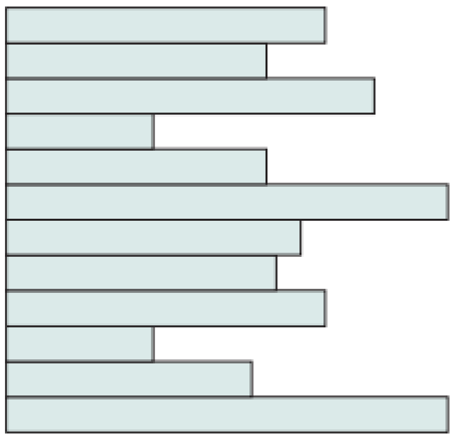
There are 5 File Organization methods to organize files in an operating system..

1) The Pile
2) The Sequential file
3) The Indexed sequential file
4) The Indexed file
5) The Direct or Hashed file

Here, we discuss only any 4 File Organization Methods:

-> **The Pile:** The least-complicated form of file organization may be termed the pile. Data is collected in the order in which they arrive. Each record consists of one burst of data. The purpose of the pile is simply to accumulate the mass of data and save it. Records may have different fields, or similar fields in different orders. Thus, each field should be self-describing, including a field name as well as a value.

-> Because there is no structure to the pile file, record access is by exhaustive search. That is, if we wish to find a record that contains a particular field with a particular value, it is necessary to examine each record in the pile until the desired record is found or the entire file has been searched.



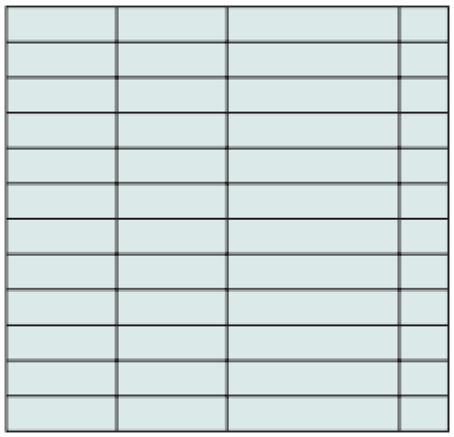
(a) Pile file

->**The Sequential file:** The most common form of file structure is the sequential file. In this type of file, a fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length and position of each field are known, only the values of fields need to be stored; the field name and length for each field are attributes of the file structure.

-> One particular field, usually the first field in each record, is referred to as the key field. The key field uniquely identifies the record; Thus, key values for different records are always different. The records are stored in key sequence.

-> Sequential files are typically used in batch applications and are generally optimum for such applications if they involve the processing of all the records. Efficient for reading all records sequentially but can be inefficient for random access or modifications.

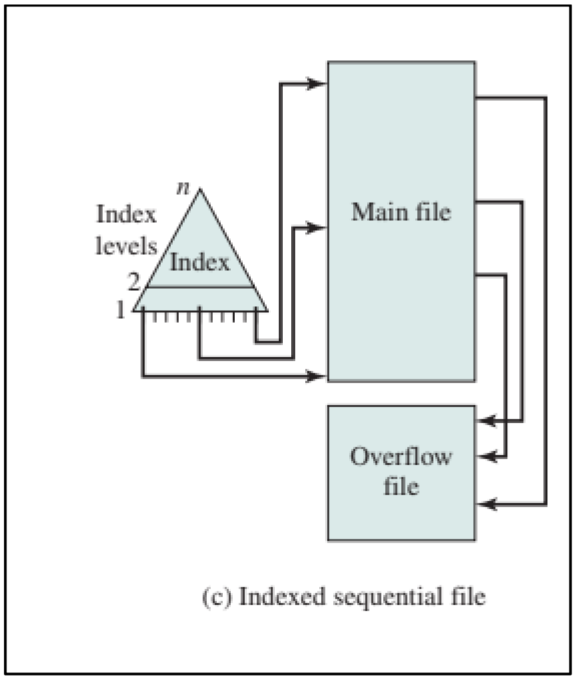
->A sequential file is stored in simple sequential ordering of the records within blocks. That is, the physical organization of the file on tape or disk directly matches the logical organization of the file.


Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential file

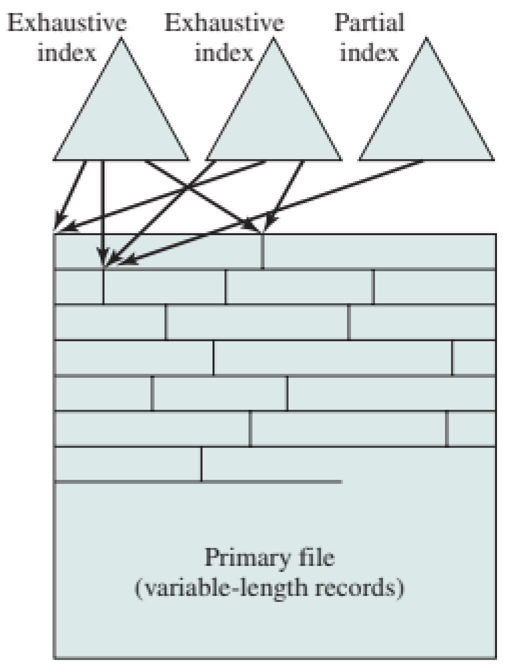
-> The Indexed sequential file: A popular approach to overcoming the disadvantages of the sequential file is the indexed sequential file. The indexed sequential file maintains a key characteristic of the sequential file and two more features are added: an index to the file to support random access, and an overflow file.

->The index provides a lookup capability to quickly reach the vicinity of a desired record. The overflow file is similar to the log file used with a sequential file but is integrated so that a record in the overflow file is located by following a pointer from its predecessor record.



(c) Indexed sequential file

-> The Indexed file: In the general indexed file, the concept of sequentially and a single key are abandoned. Records are accessed only through their indexes. The result is that there is now no restriction on the placement of records as long as a pointer in at least one index refers to that record.
-> Furthermore, variable-length records can be employed. Two types of indexes are used. An exhaustive index contains one entry for every record in the main file. The index itself is organized as a sequential file for ease of searching. A partial index contains entries to records where the field of interest exists. With variable-length records, some records will not contain all fields.
-> When a new record is added to the main file, all of the index files must be updated. Indexed files are used mostly in applications where timeliness of information is critical and where data is rarely processed exhaustively. **Examples** are airline reservation systems and inventory control systems.



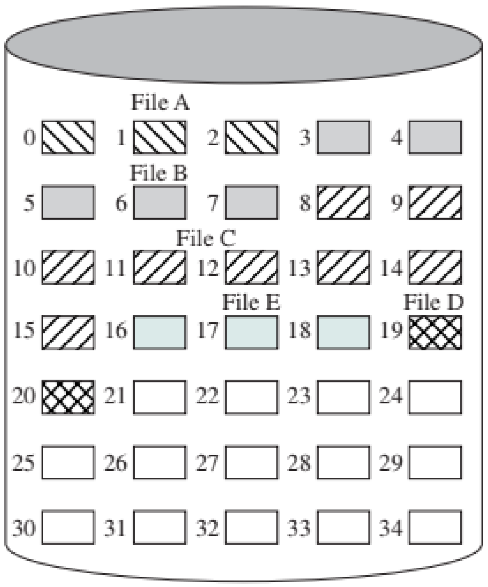
(d) Indexed file

Q36. Briefly explain different file allocation methods.
Ans. There are 3 file allocation methods that is commonly used:

1) Contiguous Allocation
2) Chained Allocation
3) Indexed Allocation

1. Contiguous Allocation: In contiguous allocation, a single contiguous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable-size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file.

-> Contiguous allocation is the best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block.



File allocation table

File name	Start block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.10 Contiguous File Allocation (After Compaction)

For example, If a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b + i – 1.
In this file allocation system External fragmentation will occur ,making it difficult to find contiguous blocks of space of sufficient length. After some specific time-interval, it will be necessary to perform a compaction algorithm to free up additional space on the disk.

2. Chained Allocation: At the opposite extreme from contiguous allocation is chained allocation. Typically, allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed.

-> The selection of blocks is now a simple matter: Any free block can be added to a chain. There is no external fragmentation to worry about because only one block at a time is needed. This type of physical organization is best suited to sequential files that are to be processed sequentially. To select an individual block of a file requires tracing through the chain to the desired block.

->One consequence of chaining, as described so far, is that there is no accommodation of the principle of locality. Thus, if it is necessary to bring in several blocks of a file at a time, as in sequential processing, then a series of accesses to different parts of the disk are required.

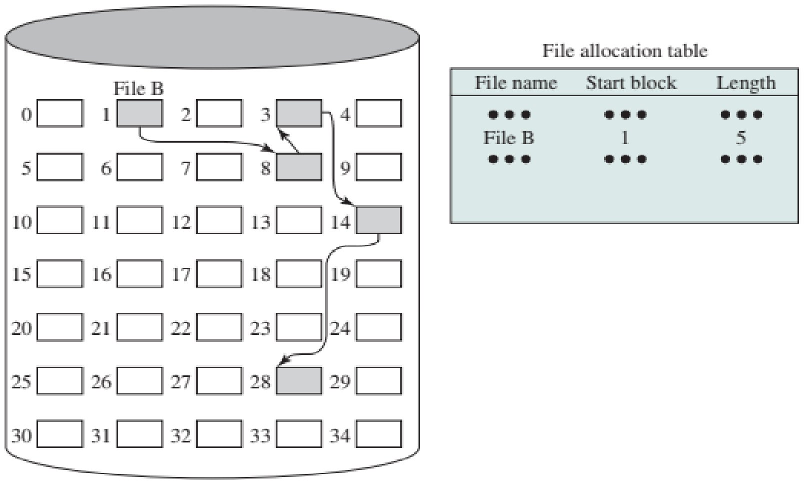


Figure 12.11 Chained Allocation

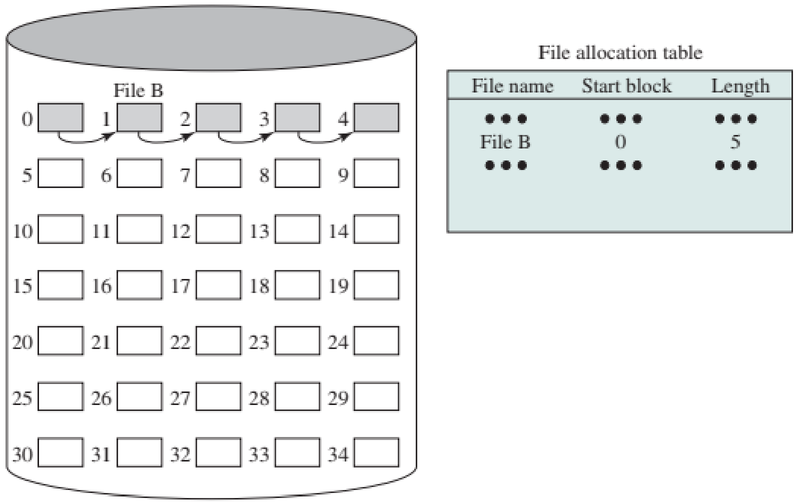


Figure 12.12 Chained Allocation (After Consolidation)

3. Indexed Allocation: Indexed allocation addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file; the index has one entry for each portion allocated to the file.

-> Typically, the file indexes are not physically stored as part of the file allocation table. Rather, the file index for a file is kept in a separate block, and the entry for the file in the file allocation table points to that block.

->Allocation may be on the basis of either fixed-size blocks or variable-size portions. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size portions improves locality. In either case, file consolidation may be done from time to time. File consolidation reduces the size of the index in the case of Variable-size portions, but not in the case of block allocation.

-> Indexed allocation supports both sequential and direct access to the file and thus is the most popular form of file allocation.

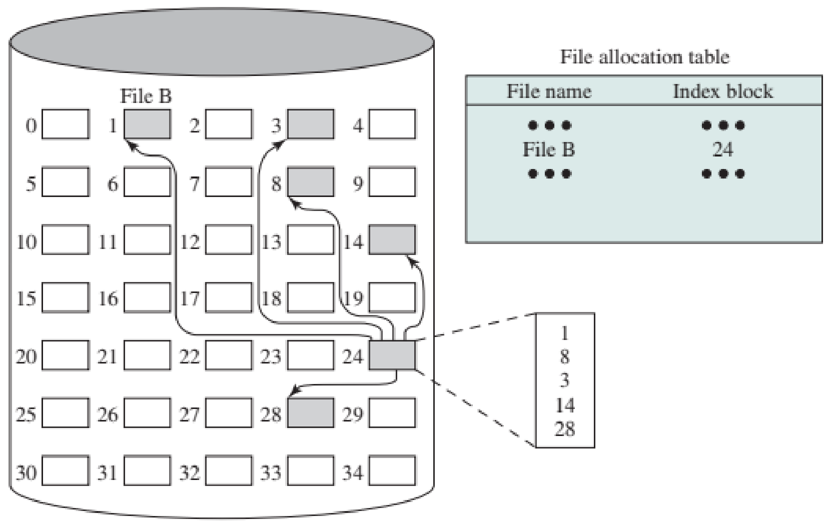


Figure 12.13 **Indexed Allocation with Block Portions**