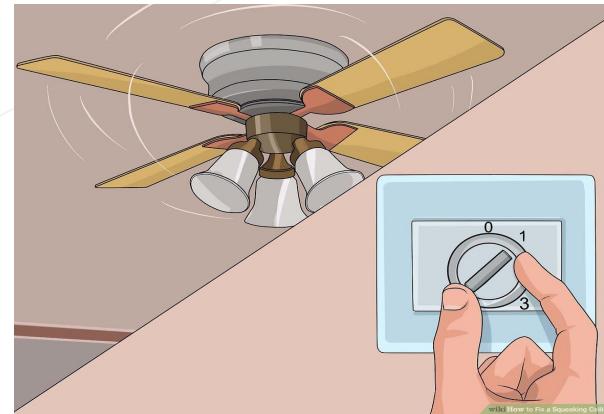


What is Operating System

- Whatever used as an interface between the user and the core machine is OS.
 - E.g. steering of car, switch of the fan etc., Buttons over electronic devices.



- Question comes why we need an operating system?
 - To enable everybody to use h/w In a convenient and efficient manner

Definition of Operating System

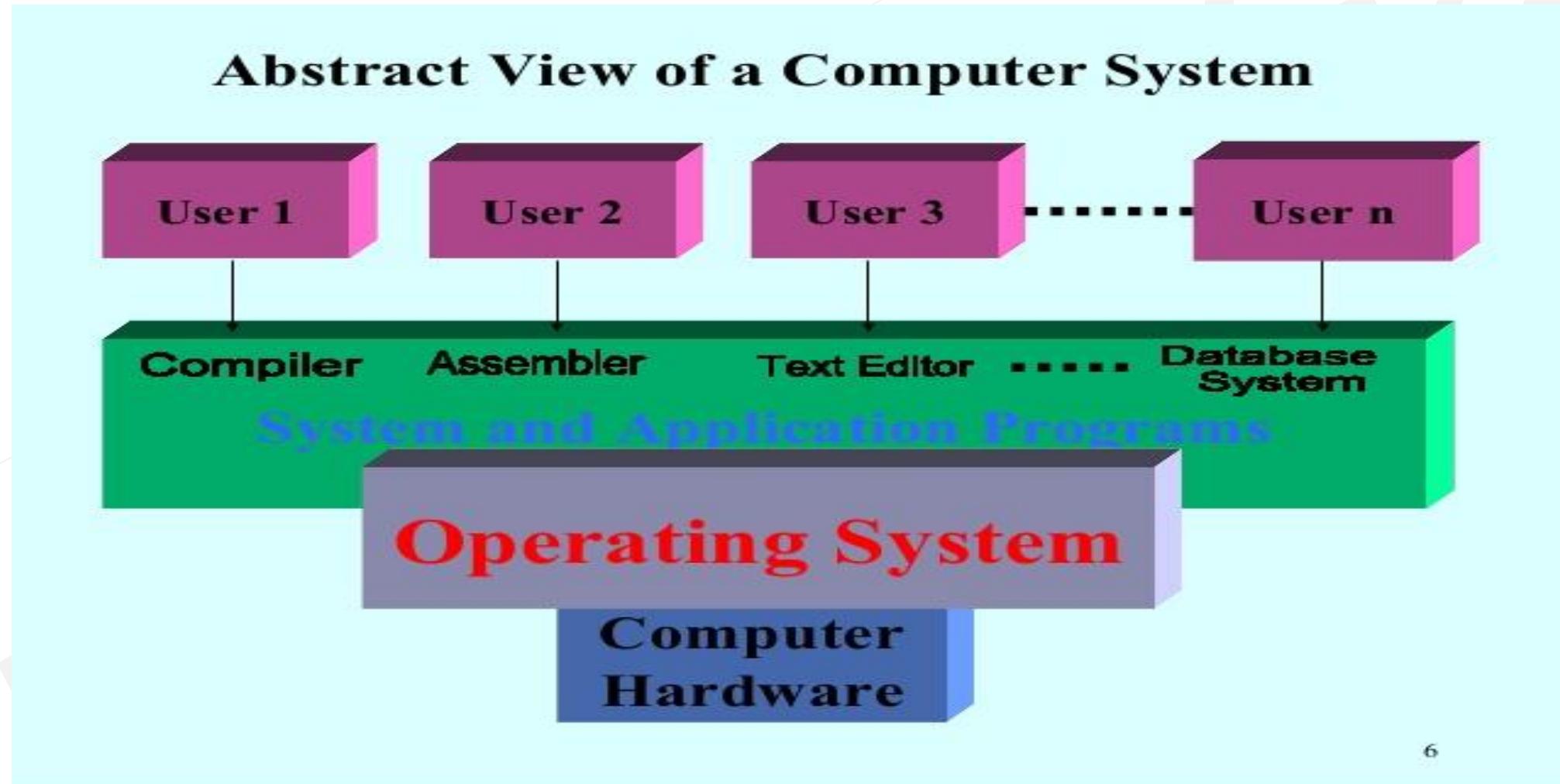
- There is no exact or precise definition for OS but we can say, “A program or System software”
 - Which Acts as an **intermediary** between user & h/w



- **Resource Manager/Allocator** - Manage system resources in an unbiased fashion both h/w (mainly CPU time, memory, system buses) & s/w (access, authorization, semaphores) and provide functionality to application programs.
- OS controls and coordinates the use of resources among various application programs.



- OS provides platform on which other application programs can be installed, provides the environment within which programs are executed.



What is Operating System



सत्यमेव जयते

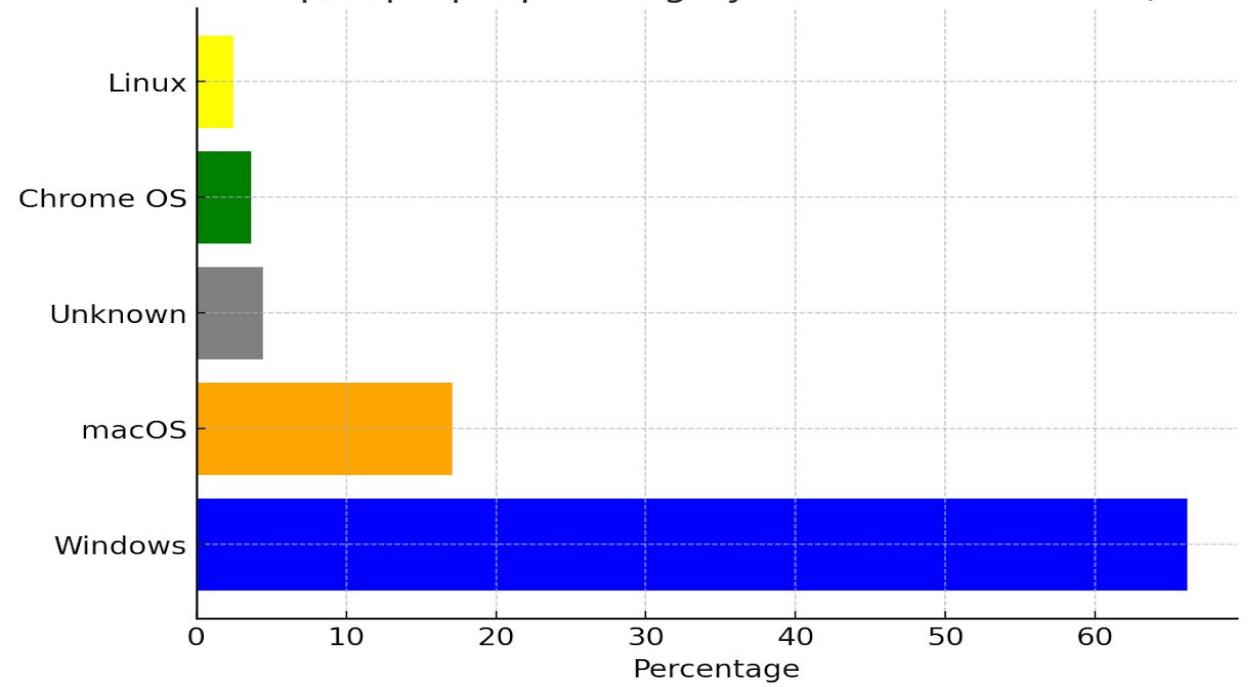
भारत सरकार
GOVERNMENT
OF INDIA

Map of India

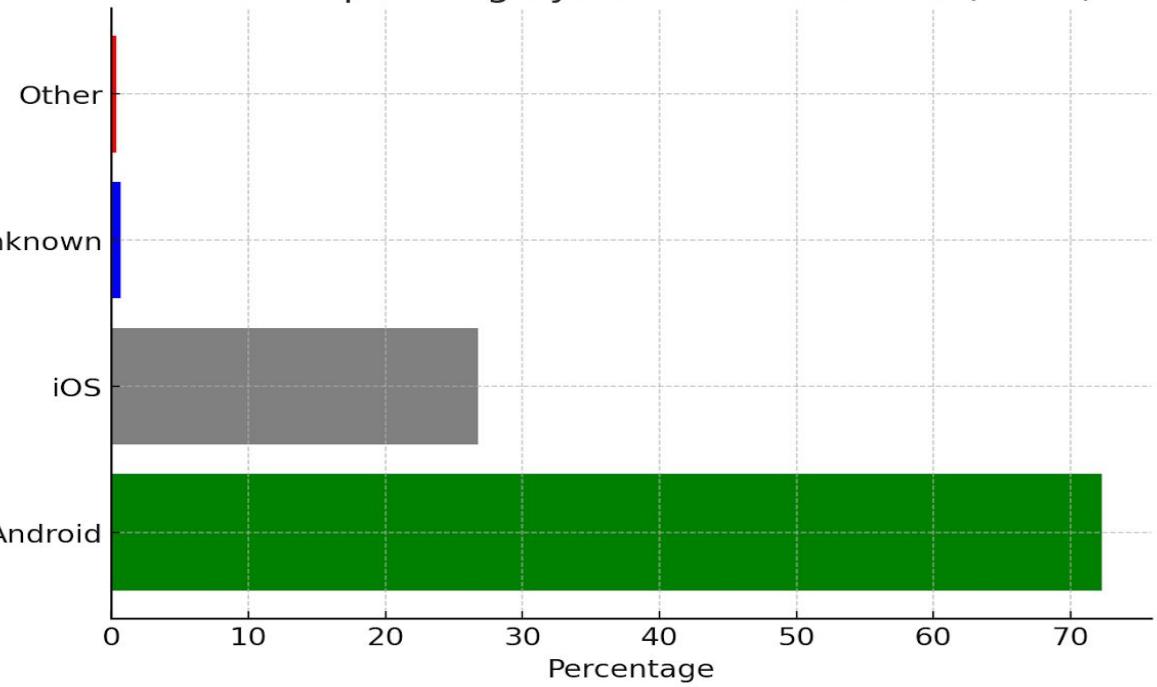


- **Windows OS:**
 - Windows 10: Popular version of Microsoft's OS.
 - Windows 11: Latest version with new features.
- **Mac OS:**
 - macOS Big Sur: Older version with improved performance.
 - macOS Monterey: Latest version with new features.
- **Linux OS:**
 - Ubuntu: Popular Linux distribution for desktops and servers.
 - Fedora: Known for cutting-edge features.
 - Debian: Highly stable Linux distribution.
- **Unix:**
 - AIX: IBM's version of UNIX.
 - HP-UX: Hewlett Packard's version of UNIX.
- **Mobile OS:**
 - Android: Developed by Google for smartphones and tablets.
 - iOS: Apple's OS for iPhones and iPads.
- **Real-Time Operating Systems (RTOS):**
 - VxWorks: Used in embedded systems and real-time applications.
 - RTEMS: Open-source RTOS for embedded systems.

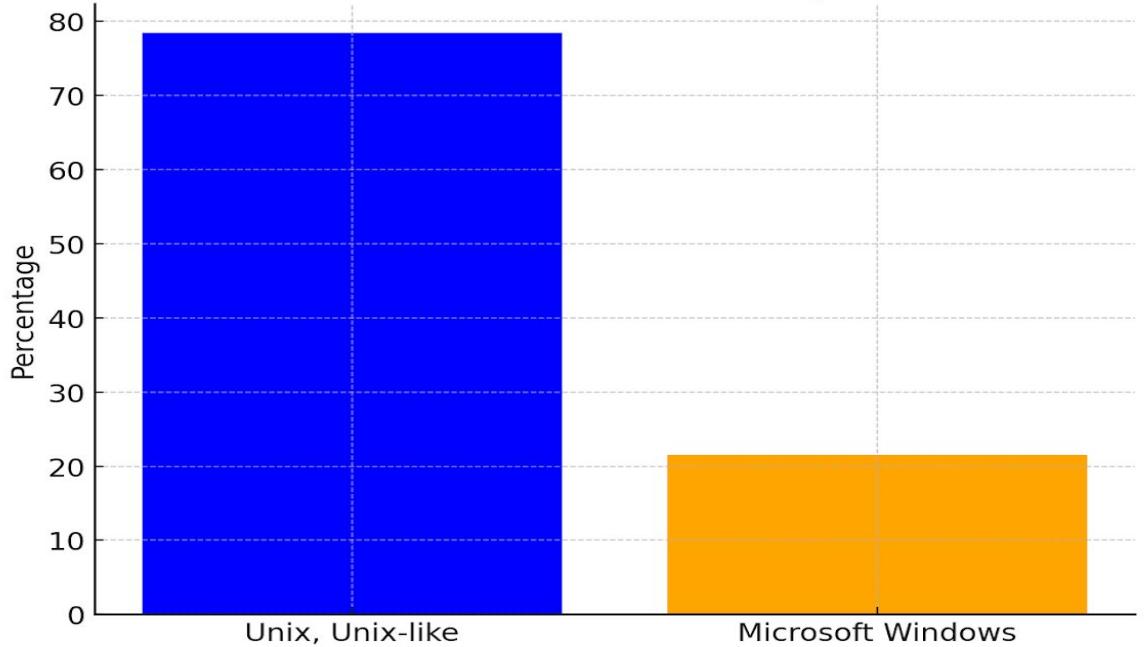
Desktop/Laptop Operating System Market Share (2024)



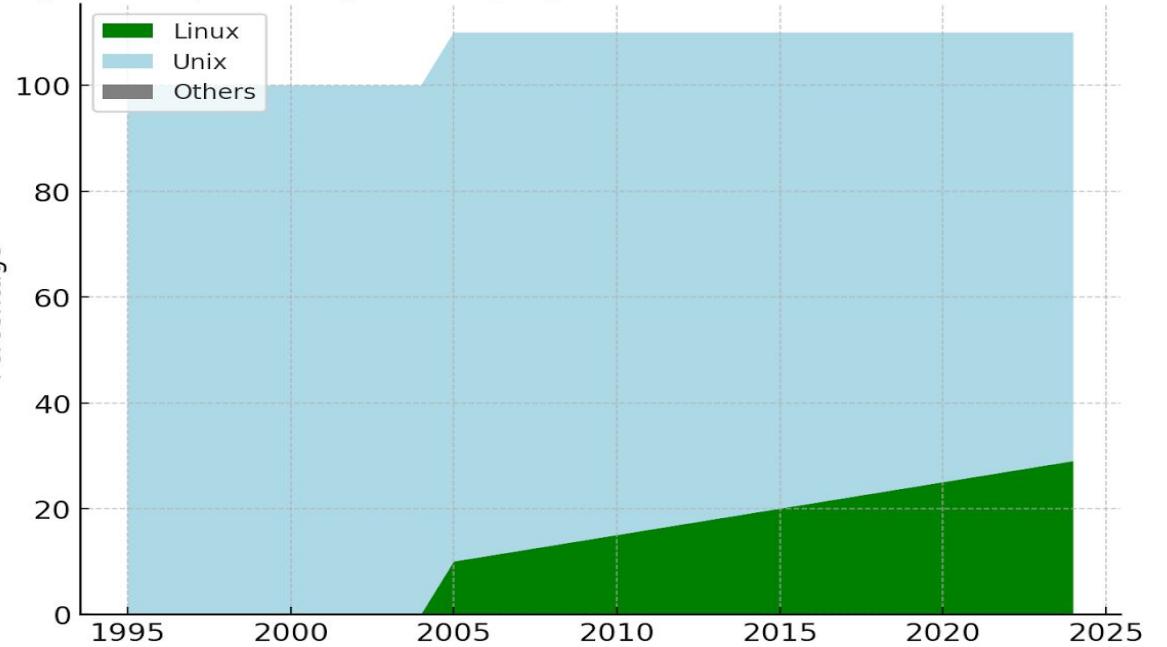
Mobile Operating System Market Share (2024)



Public Servers on the Internet by OS (2024)

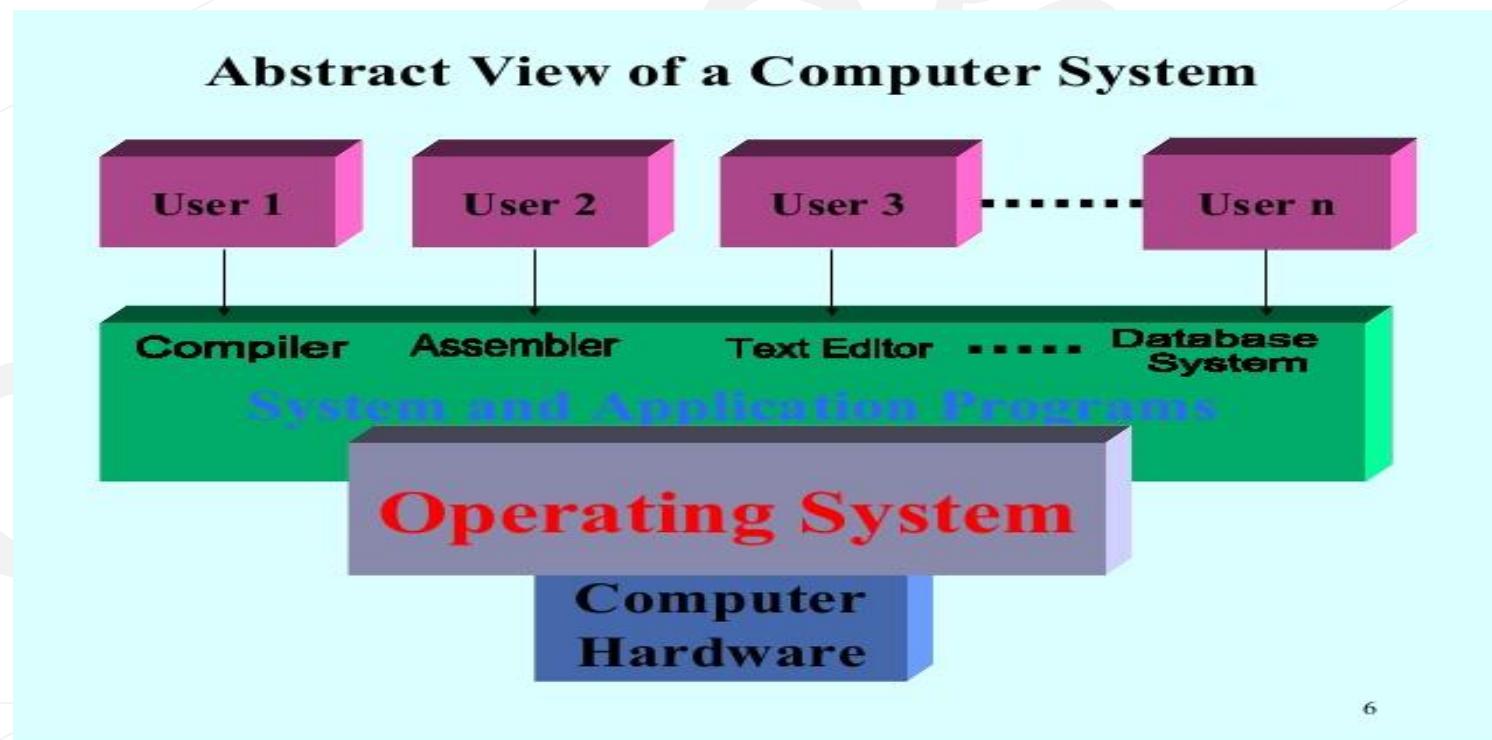


Supercomputer Operating System Market Share (1995-2024)



- **Computer hardware** – CPU, memory units, i/o devices, system bus, registers etc. provides the basic computing resources.
- **OS** - Control and coordinates the use of the hardware among the various applications programs.
- **System and Applications programs** - Defines the way in which these resources are used to solve the computing problems of the user.

- **User**



Goals and Functions of operating system

- Goals are the ultimate destination, but we follow functions to implement goals.
 - Primary goals (Convenience / user friendly)
 - Secondary goals (Efficiency (Using resources in efficient manner) / Reliability / maintainability)



सबका साथ सबका विकास



सत्यमेव जयते

Government Of India

| Ministries | Departments |
|------------|-------------|
| 58 | 93 |

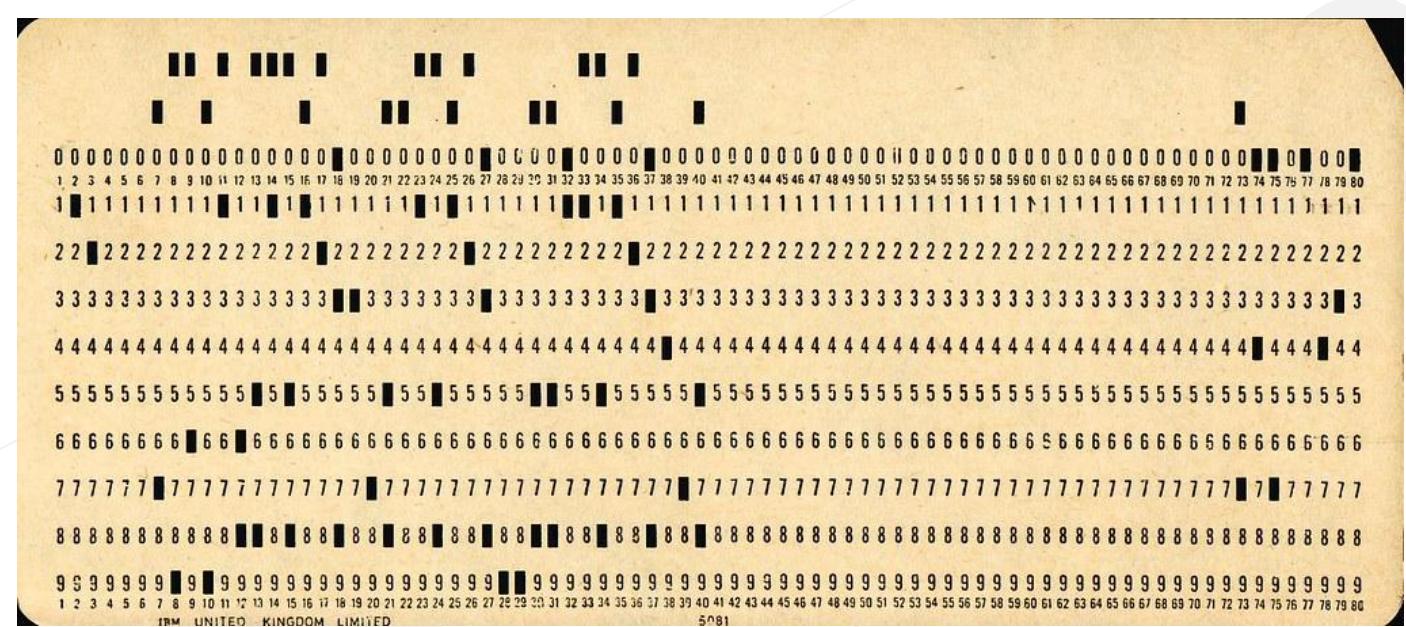
Primary goals (Convenience / user friendly)



Evolution of Operating System

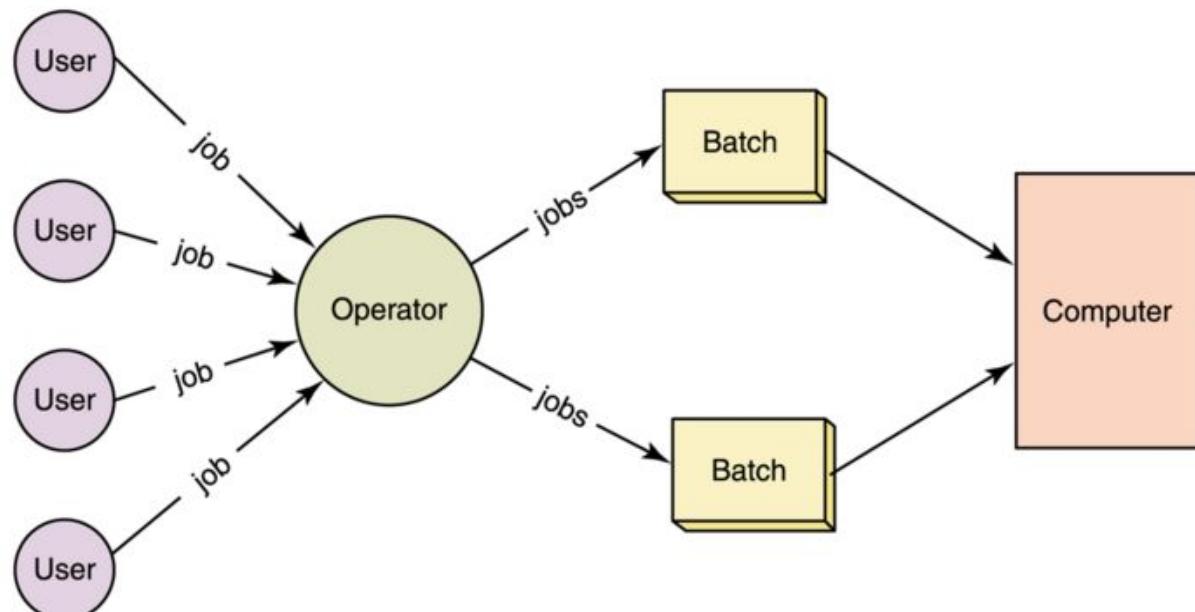
- Early computers were not interactive device, there user use to prepare a job which consist three parts
 - Program
 - Control information
 - Input data
- Only one job is given input at a time as there was no memory, computer will take the input then process it and then generate output.
- Common input/output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

Punch Card in Punch Card Machine



Batch Operating System

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

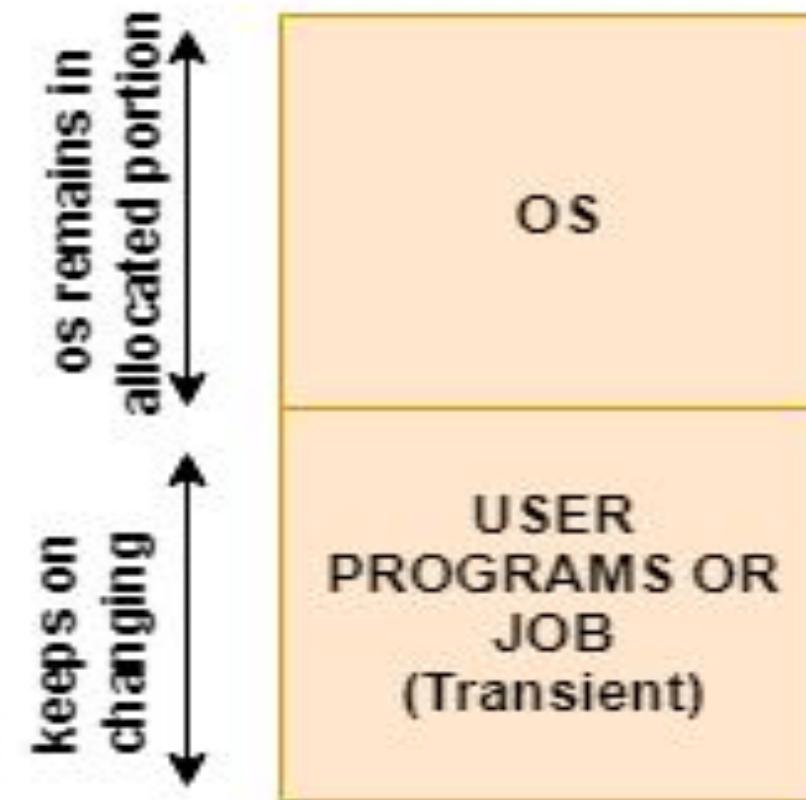


Advantage

- The batched jobs were executed automatically one after another saving its time by performing the activities (like loading of compiler) only for once. It resulted in improved system utilization due to reduced turnaround time.
- Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention

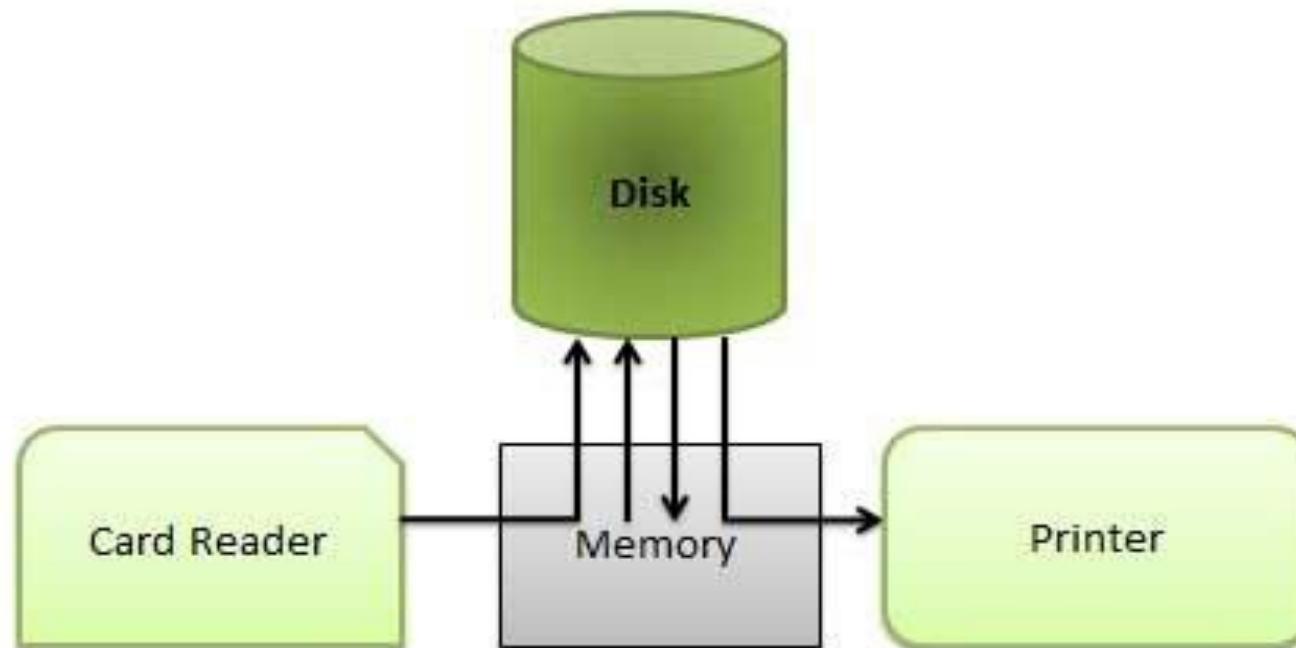
Disadvantage

- Memory limitation – memory was very limited, because of which interactive process or multiprogramming was not possible

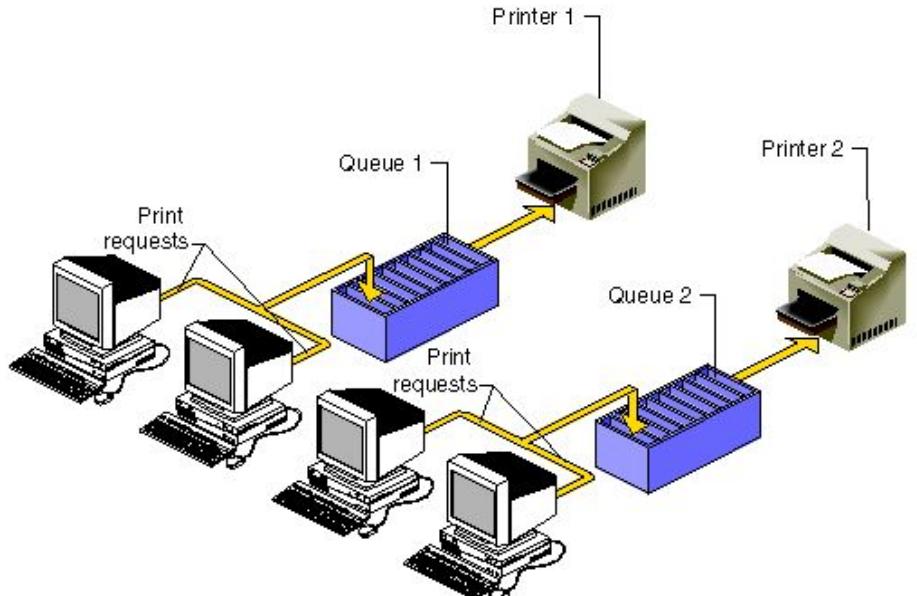


Spooling

- ("Spool" is technically an acronym for simultaneous peripheral operations online.)
- In a computer system peripheral equipment, such as printers and punch card readers etc, are very slow relative to the performance of the rest of the system. Spooling is useful because devices access data at different rates.
- Spooling is a process in which data is temporarily held to be used and executed by a device, program or the system. Data is sent to and stored in memory or other volatile storage until the program or computer requests it for execution.
- Generally, the spool is maintained on the computer's physical memory, buffers or the I/O device-specific interrupts



- The most common implementation of spooling can be found in typical input/output devices such as the keyboard, mouse and printer. For example, in printer spooling, the documents/files that are sent to the printer are first stored in the memory. Once the printer is ready, it fetches the data and prints it.
- A spooler works by intercepting the information going to the printer, parking it temporarily on disk or in memory. The computer can send the document information to the spooler at full speed, then immediately return control of the screen to you.
- The spooler, meanwhile, hangs onto the information and feeds it to the printer at the slow speed the printer needs to get it. So if your computer can **spool**, you can work while a document is being printed.



- Even experienced a situation when suddenly for some seconds your mouse or keyboard stops working? Meanwhile, we usually click again and again here and there on the screen to check if its working or not. When it actually starts working, what and wherever we pressed during its hang state gets executed very fast because all the instructions got stored in the respective device's spool.
- Spooling is capable of overlapping I/O operation for one job with processor operations for another job. i.e. multiple processes can write documents to a print queue without waiting and resume with their work.

Q. Which of the following is an example of spooled device? (GATE -1996, 1 Marks)

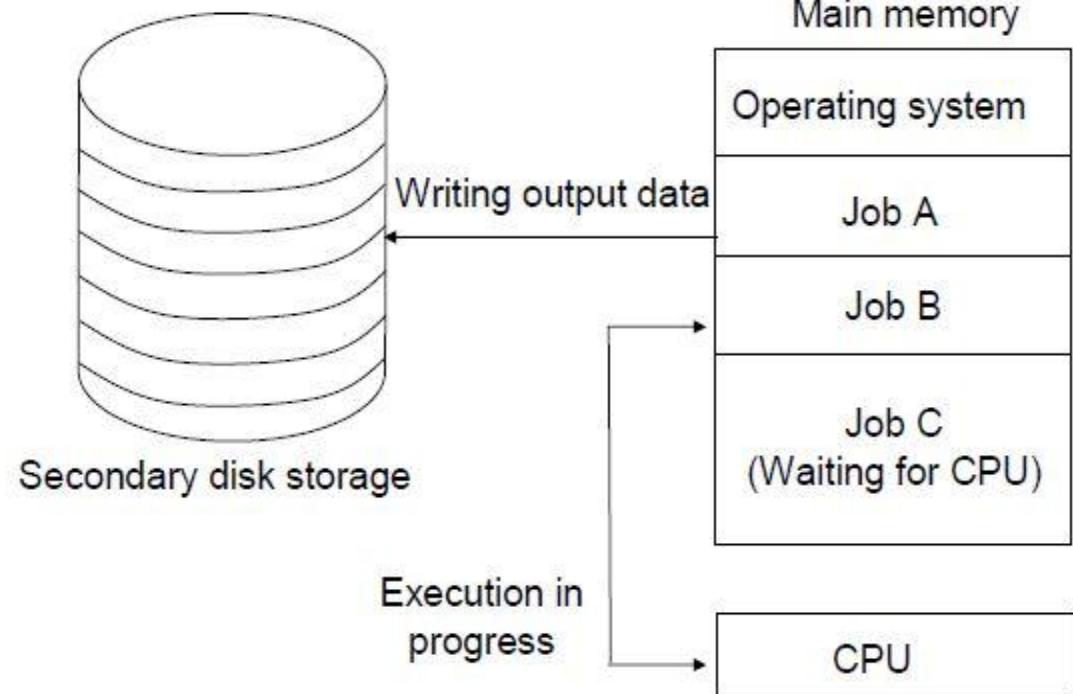
- a) A line printer used to print the output of a number of jobs
- b) A terminal used to enter data to a running program
- c) A secondary storage device in a virtual memory system
- d) A graphic display device

Q. Which of the following is an example of spooled device? (GATE -1998, 1 Marks)

- a) A terminal used to enter data for the C Program being executed
- b) An output device used to print the output of a number of jobs
- c) A secondary storage device in a virtual storage system
- d) The swapping area on a disk used by the swapper

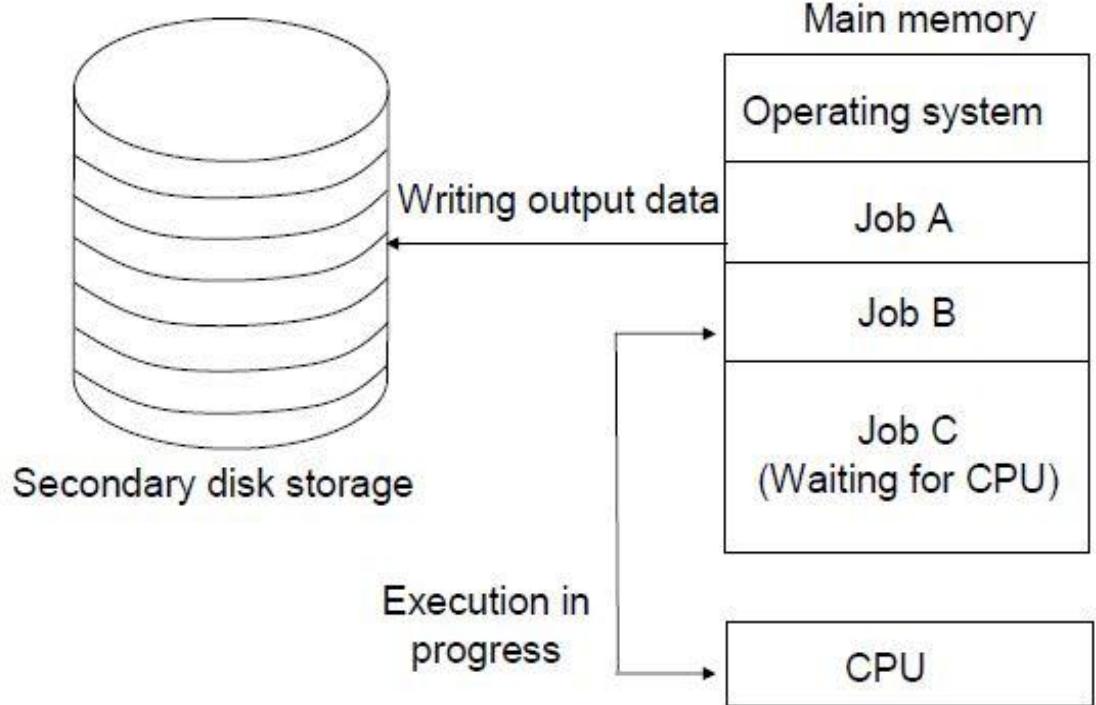
Multiprogramming Operating System

- A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. The basic idea of multiprogramming operating system is it keeps several jobs in main memory simultaneously.
- The jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.
- The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.



Show must go on

- In a non-multi-programmed system, the CPU would sit idle and wait but in a multi-programmed system, the operating system simply switches to, and executes, another job. When ***that*** job needs to wait the CPU switches to ***another*** job, and so on.
- Eventually, the first job finishes waiting and gets the CPU back. So, conclusion is as long as at least one job needs to execute, the CPU is never idle.



Processor किसी के लिए wait नहीं करेगा

Advantage

- High and efficient CPU utilization.
- Less response time or waiting time or turnaround time
- In most of the applications multiple tasks are running and multiprogramming systems better handle these type of applications
- Several processes share CPU time

Disadvantage

- It is difficult to program a system because of complicated schedule handling.
- To accommodate many jobs in main memory, complex memory management is required.

Q Which of the following features will characterize an OS as multi-programmed OS?
(NET-DEC-2012) (Gate-2002) (1 Marks)

- (a)** More than one program may be loaded into main memory at the same time.
 - (b)** If a program waits for certain event another program is immediately scheduled.
 - (c)** If the execution of a program terminates, another program is immediately scheduled.
- (A)** (a) only
- (B)** (a) and (b) only
- (C)** (a) and (c) only
- (D)** (a), (b) and (c) only

Multitasking Operating system/time sharing/Multiprogramming with Round Robin/ Fair Share

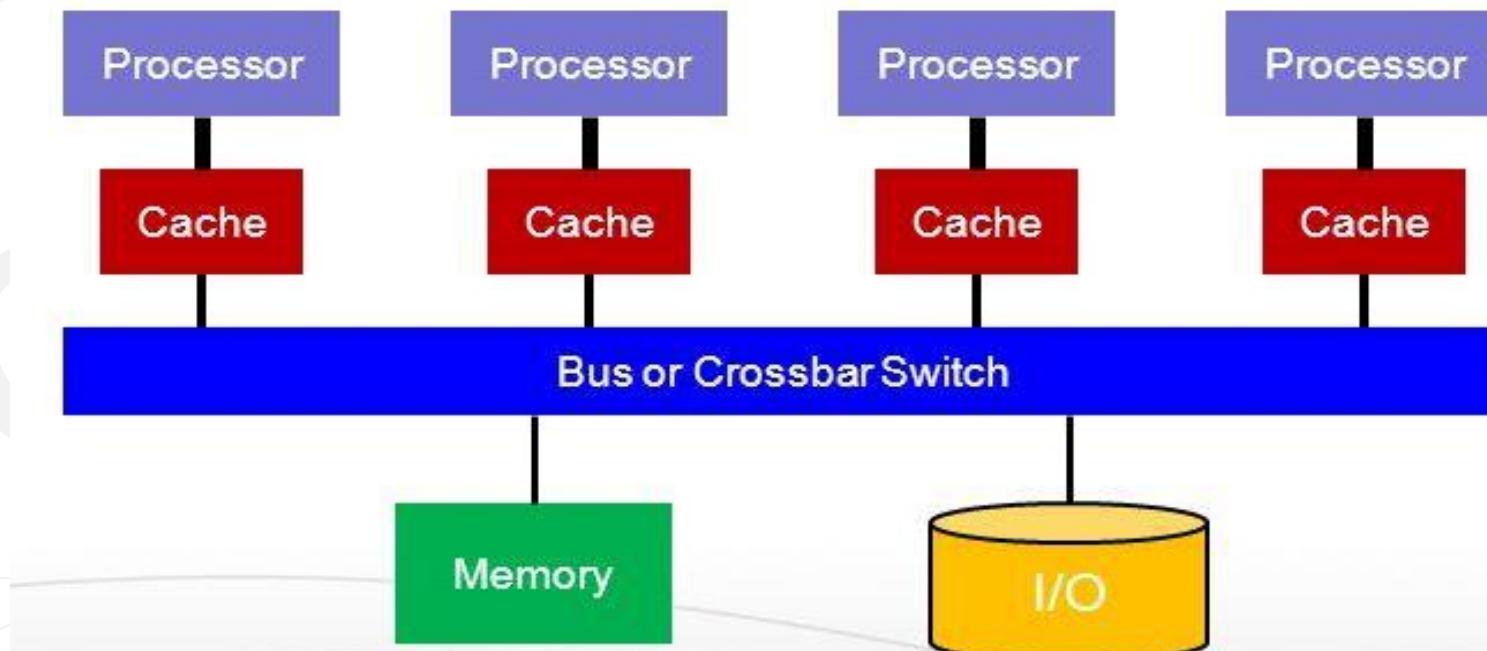
- In the modern operating systems, we are able to play MP3 music, edit documents in Microsoft Word, surf the Google Chrome all running at the same time. (by context switching, the illusion of parallelism is achieved)
- For multitasking to take place, firstly there should be multiprogramming i.e. presence of multiple programs ready for execution. And secondly the concept of time sharing.



- Time sharing (or multitasking) is a logical extension of multiprogramming, it allows many users to share the computer simultaneously. the CPU executes multiple jobs (May belong to different user) by switching among them, but the switches occur so frequently that, each user is given the impression that the entire computer system is dedicated to his/her use, even though it is being shared among many users.

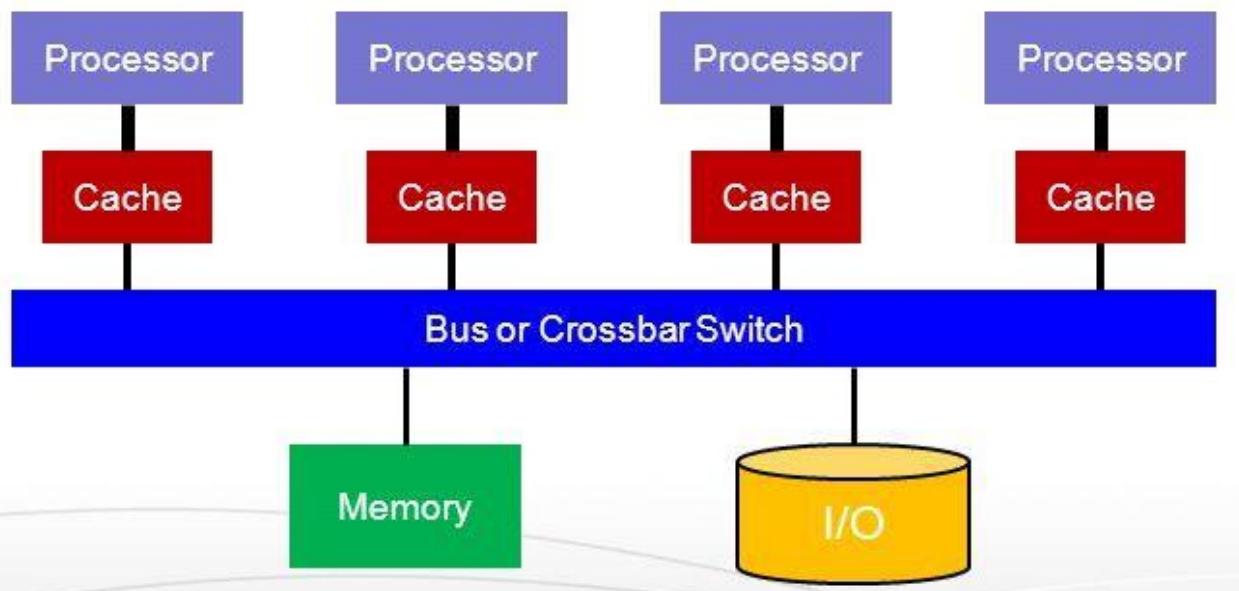
Multiprocessing Operating System/ tightly coupled system

- **Definition and Purpose:** A multiprocessing operating system supports the simultaneous execution of multiple processes by utilizing two or more CPUs within a single computer system. The goal is to enhance processing speed and reliability.
- **How It Works:** The system distributes processes across multiple CPUs, allowing tasks to be executed in parallel. This parallelism increases efficiency, reduces processing time, and improves system performance.
- **Advantages:**
 - **Increased Throughput:** More tasks can be processed in a shorter time due to parallel execution.
 - **Improved Reliability:** If one CPU fails, others can continue to work, maintaining system functionality.
 - **Better Resource Utilization:** Multiple CPUs ensure that system resources are used more effectively, reducing idle time.

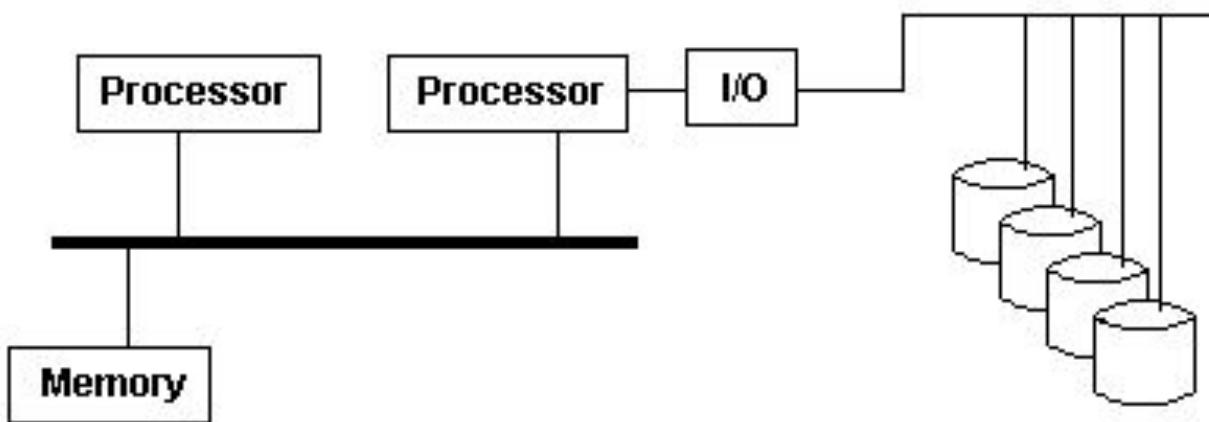


Multiprocessing can be of two types

- **Symmetric multiprocessing** - In a symmetric multi-processing, a single OS instance controls two or more identical processors connected to a single shared main memory.
- Most of the multi-processing PC motherboards utilize symmetric multiprocessing. Here each processor runs an identical copy of operating system and these copies communicate with each other.
- SMP means that all processors are peers; no boss–worker relationship exists between processors. Windows, Mac OSX and Linux.



- **Asymmetric** - This scheme defines a master-slave relationship, where one processor behaves as a master and control other processor which behaves as slaves.
- For e.g. It may require that only one particular CPU respond to all hardware interrupts, whereas all other work in the system may be distributed equally among CPUs;
- Or execution of kernel-mode code may be restricted to only one particular CPU, whereas user-mode code may be executed in any combination of processors.
- Multiprocessing systems are often easier to design if such restrictions are imposed, but they tend to be less efficient than systems in which all CPUs are utilized.
- A boss processor controls the system, other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss-worker relationship. The boss processor schedules and allocates work to the worker processors.



Advantage of multiprocessing

- **Increased Throughput** - By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N , however; rather, it is less than N . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. These types of systems are used when very high speed is required to process a large volume of data.
- **Economy of Scale** - Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.
- **Increased Reliability (fault tolerance)** - If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fail, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.
- **Less Battery consumption and heat generation**

Disadvantages

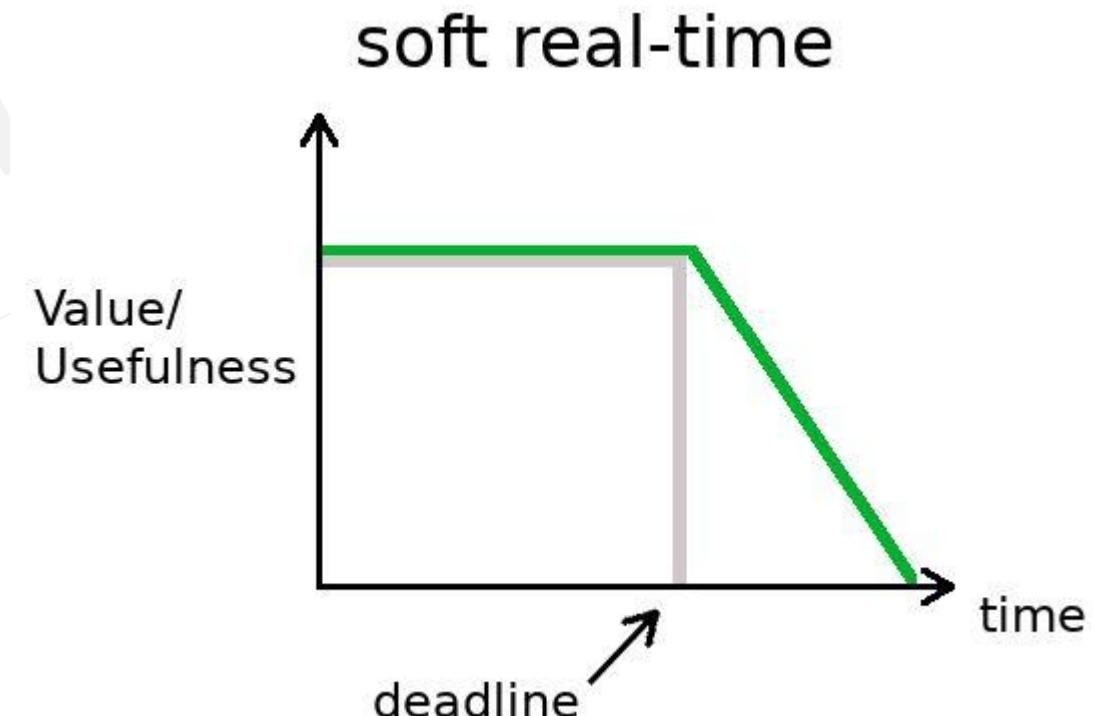
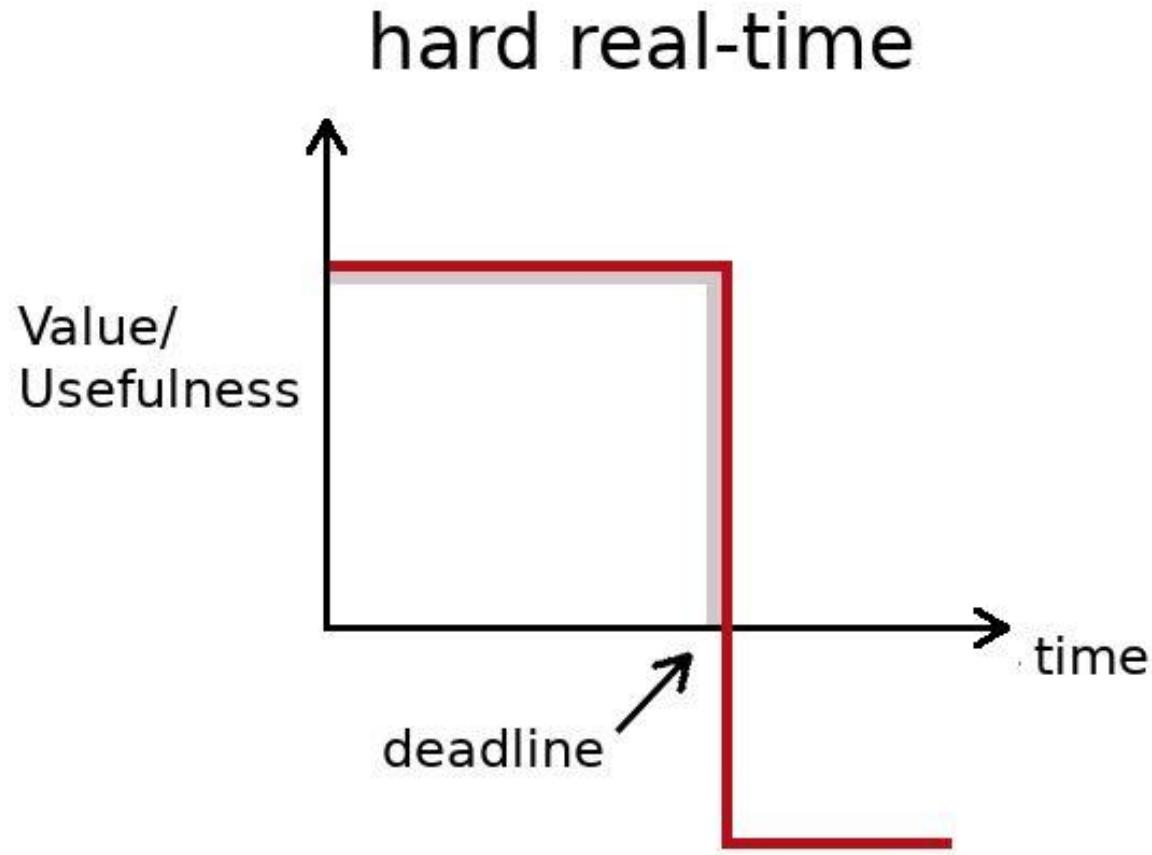
- It's more complex than simple operating systems.
- It requires context switching which may impacts performance.
- If one processor fails then it will affect in the speed
- large main memory required

Real time Operating system

- **Definition and Purpose:** An RTOS is designed to process data and execute tasks within a strict time constraint, providing predictable and deterministic responses to real-time events. It is essential for applications requiring precise timing and reliability, such as embedded systems and industrial control.
- **Mechanism:** RTOS prioritizes tasks based on their urgency and importance, using real-time scheduling algorithms. It ensures that high-priority tasks are executed immediately, preempting lower-priority tasks if necessary, to meet critical deadlines.
- **Advantages:**
 - **Predictability:** Ensures consistent, timely responses to real-time events.
 - **Reliability:** Provides stable and dependable performance in time-sensitive applications.
 - **Resource Management:** Efficiently manages system resources to meet the demands of real-time applications, minimizing latency and maximizing throughput.

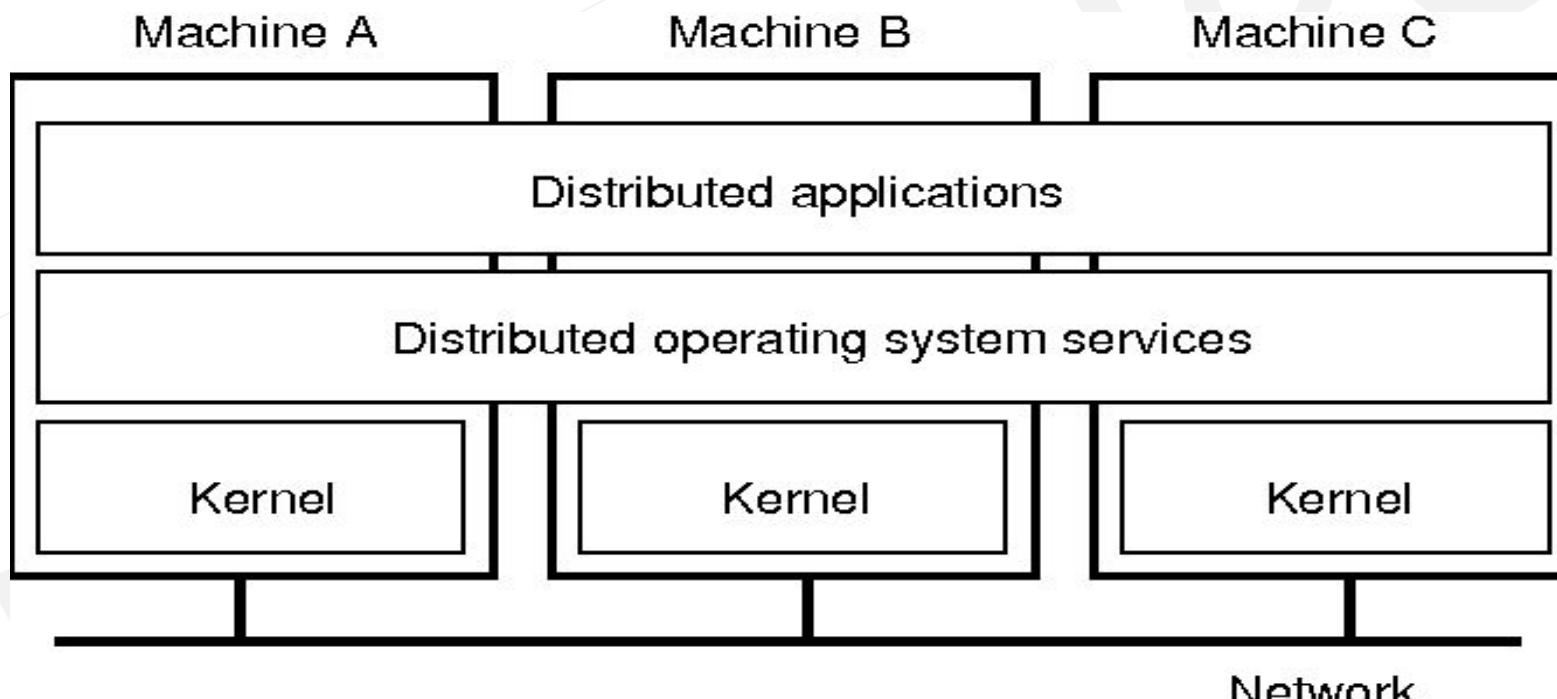


- **Hard Real-Time OS:** Ensures tasks are completed within strict deadlines, reacting at $t=0$. Used in critical systems like airbag control in cars, anti-lock brakes, and engine control systems where missing a deadline can cause catastrophic failure.
- **Soft Real-Time OS:** Allows for some flexibility in deadlines, reacting at $t=0+t = 0+t=0+$. Delays are acceptable but kept to a minimum. Common in systems like digital cameras, mobile phones, and online data processing where delays are not critical.

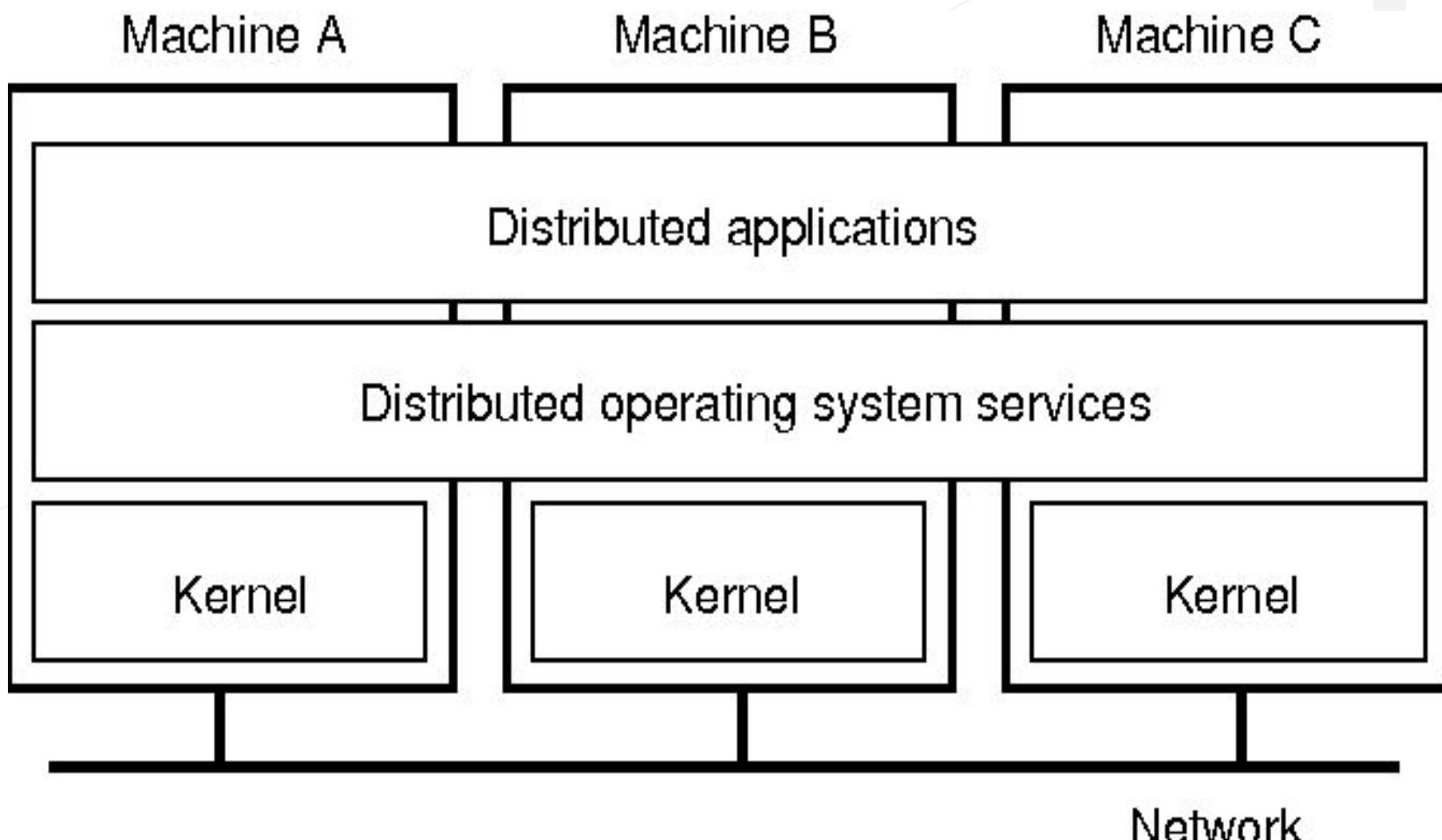


Distributed OS

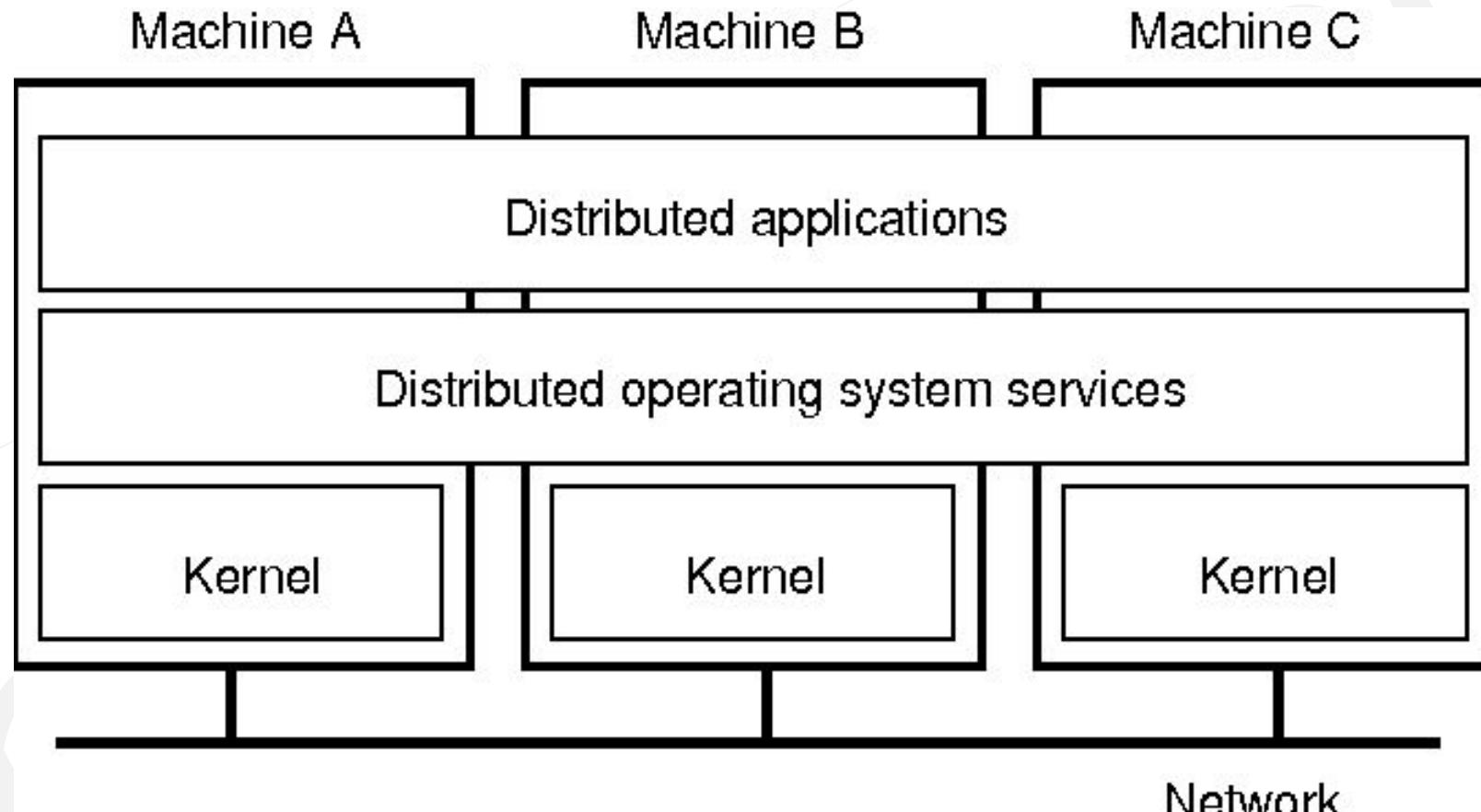
- A **distributed operating system** is a software over a collection of independent, networked, communicating, loosely coupled nodes and physically separate computational nodes.
- These nodes do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate with one another through various networks, such as high-speed buses and the Internet.
- They handle jobs which are serviced by multiple CPUs. Each individual node holds a specific software subset of the global aggregate operating system.



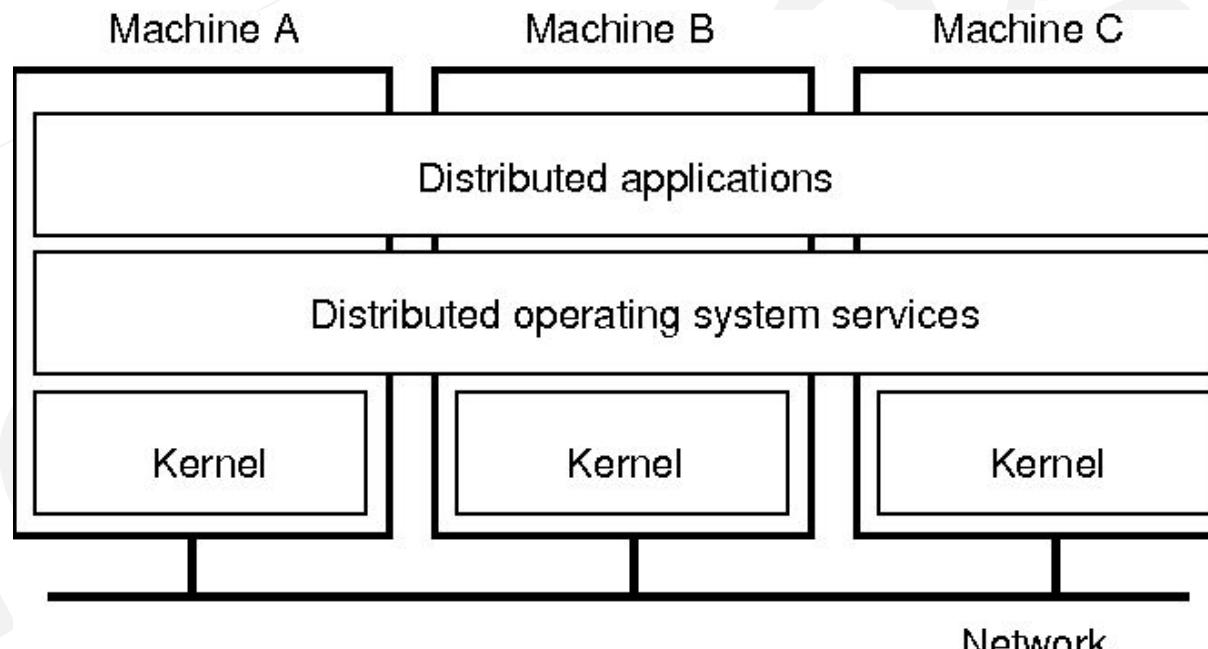
- Each subset is a composite of two distinct service provisioners. The first is a ubiquitous minimal **kernel**, or **microkernel**, that directly controls that node's hardware.
- Second is a higher-level collection of *system management components* that coordinate the node's individual and collaborative activities. These components abstract microkernel functions and support user applications.



- The microkernel and the management components collection work together. They support the system's goal of integrating multiple resources and processing functionality into an efficient and stable system.
- This seamless integration of individual nodes into a global system is referred to as *transparency*, or *single system image*; describing the illusion provided to users of the global system's appearance as a single computational entity.



- To a user, a distributed OS works in a manner similar to a single-node, monolithic operating system. That is, although it consists of multiple nodes, it appears to users and applications as a single-node.
- There are four major reasons for building distributed systems: resource sharing, computation speedup, reliability, and communication.
- E.g. Plan 9 from Bell Labs, **Inferno**



User and Operating-System Interface

- There are several ways for users to interface with the operating system.

```
Command Prompt
12/29/2017 03:42 PM <DIR> .
12/29/2017 03:42 PM <DIR> ..
12/29/2017 03:42 PM <DIR> CameraRaw
12/29/2017 03:42 PM 0 file(s) 0 bytes

Directory of C:\adobeTemp\ETR551A.tmp\A\SharedApplicationData\Adobe\CameraRaw

12/29/2017 03:42 PM <DIR> .
12/29/2017 03:42 PM <DIR> ..
12/29/2017 03:42 PM <DIR> CameraProfiles
12/29/2017 03:42 PM 0 file(s) 0 bytes

Directory of C:\adobeTemp\ETR551A.tmp\A\SharedApplicationData\Adobe\CameraRaw\CameraProfiles

12/29/2017 03:42 PM <DIR> .
12/29/2017 03:42 PM <DIR> ..
12/29/2017 03:40 PM <DIR> Adobe Standard
12/29/2017 03:42 PM <DIR> Camera
11/23/2017 09:03 AM 376,919 Index.dat
1 File(s) 376,919 bytes

Directory of C:\adobeTemp\ETR551A.tmp\A\SharedApplicationData\Adobe\CameraRaw\CameraProfiles\Adobe Standard

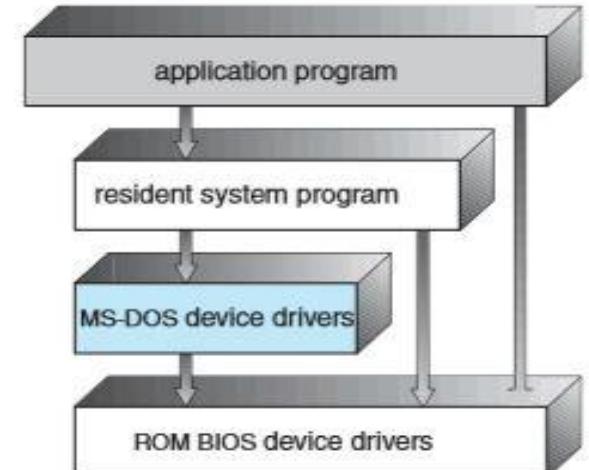
12/29/2017 03:40 PM <DIR> .
12/29/2017 03:40 PM <DIR> ..
11/23/2017 09:03 AM 120,696 Apple iPad6,3 back camera Adobe Standard.dcp
11/23/2017 09:03 AM 121,228 Apple iPad6,3 back camera Camera Default.dcp
11/23/2017 09:03 AM 121,228 Apple iPad6,3 front camera Camera Default.dcp
11/23/2017 09:03 AM 121,228 Apple iPad6,4 back camera Camera Default.dcp
11/23/2017 09:03 AM 120,700 Apple iPhone10,1 back camera Camera Adobe Standard.dcp
11/23/2017 09:03 AM 121,232 Apple iPhone10,1 back camera Camera Default.dcp
11/23/2017 09:03 AM 120,700 Apple iPhone10,2 back camera Camera Default.dcp
11/23/2017 09:03 AM 121,232 Apple iPhone10,2 back camera Camera Default.dcp
11/23/2017 09:03 AM 120,710 Apple iPhone10,2 back telephoto camera Camera Adobe Standard.dcp
11/23/2017 09:03 AM 121,242 Apple iPhone10,2 back telephoto camera Camera Default.dcp
11/23/2017 09:03 AM 120,700 Apple iPhone10,3 back camera Camera Default.dcp
11/23/2017 09:03 AM 121,232 Apple iPhone10,3 back camera Camera Default.dcp
```



| Feature | Command-Line Interface (CLI) | Graphical User Interface (GUI) | Touchscreen Interface |
|------------------|---|---|---|
| Interface Method | Command Interpreters | Mouse-based window and menu system | Touch-based interaction |
| User Interaction | Users enter text commands | Users click on icons and menus | Users make gestures on the screen |
| Typical Users | System administrators, power users | General users, beginners | Mobile users, tablet users |
| Advantages | Efficient for experienced users, faster access to tasks | User-friendly, visually intuitive | Intuitive for mobile use, supports gestures |
| Disadvantages | Steep learning curve, less user-friendly | Can be slower for advanced users, limited functionality compared to CLI | May lack precision, limited to touch-compatible devices |

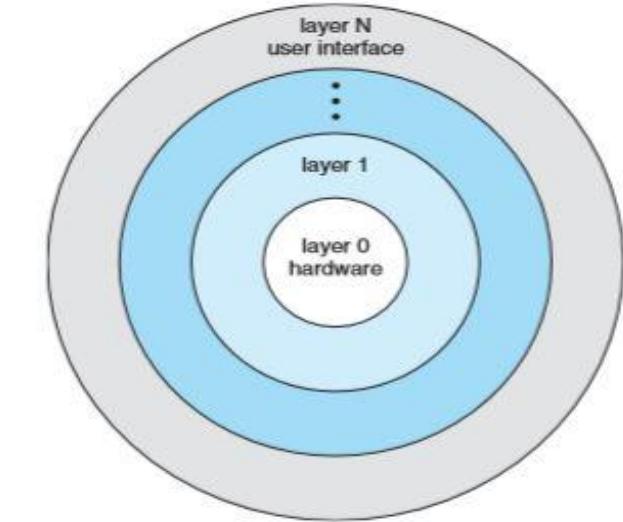
Structure of Operating System

- A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.
- Simple Structure - Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system.



MS-DOS layer structure.

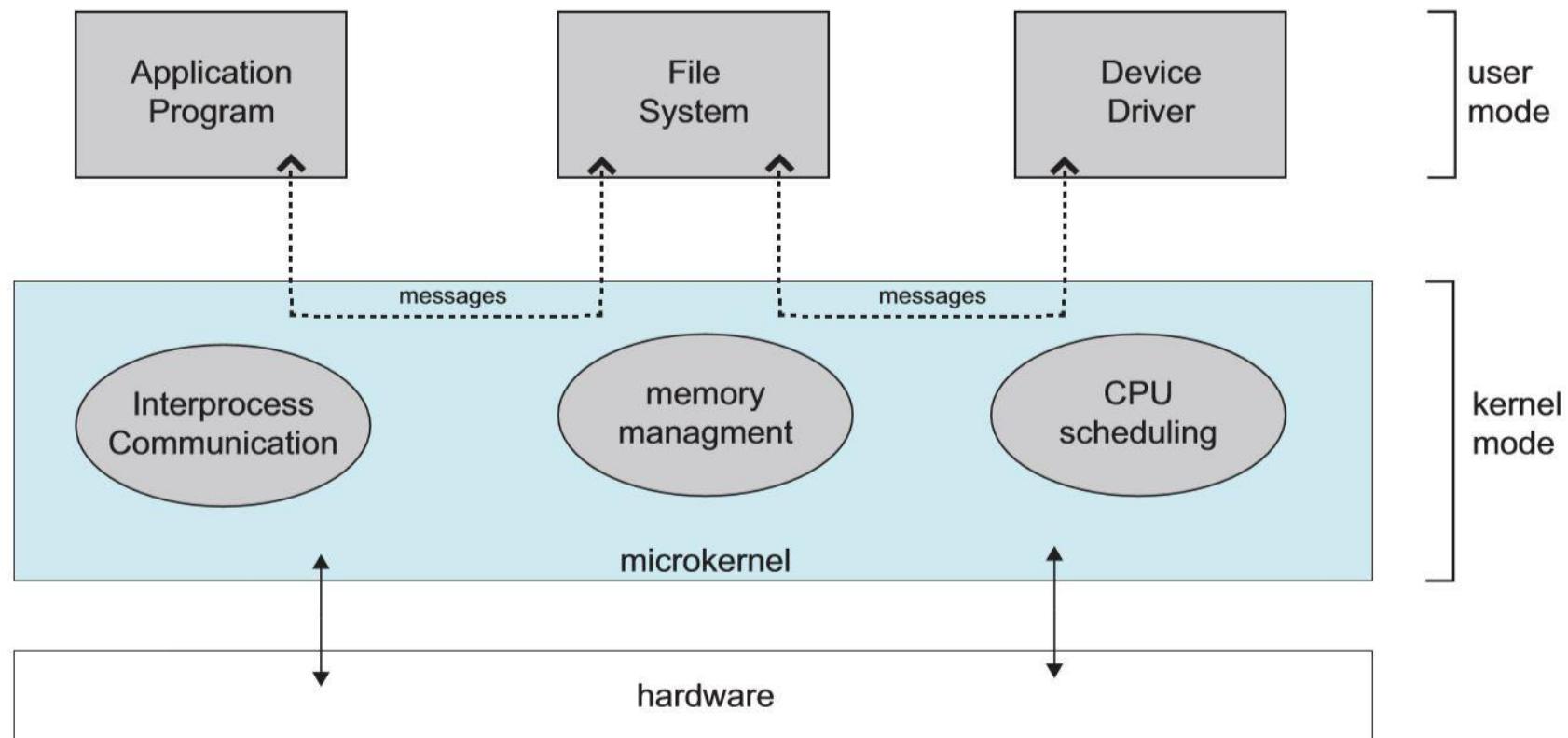
- **Layered Approach** - With proper hardware support, operating systems can be broken into pieces. The operating system can then retain much greater control over the computer and over the applications that make use of that computer.
- Implementers have more freedom in changing the inner workings of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



A layered operating system.

Micro-Kernel approach

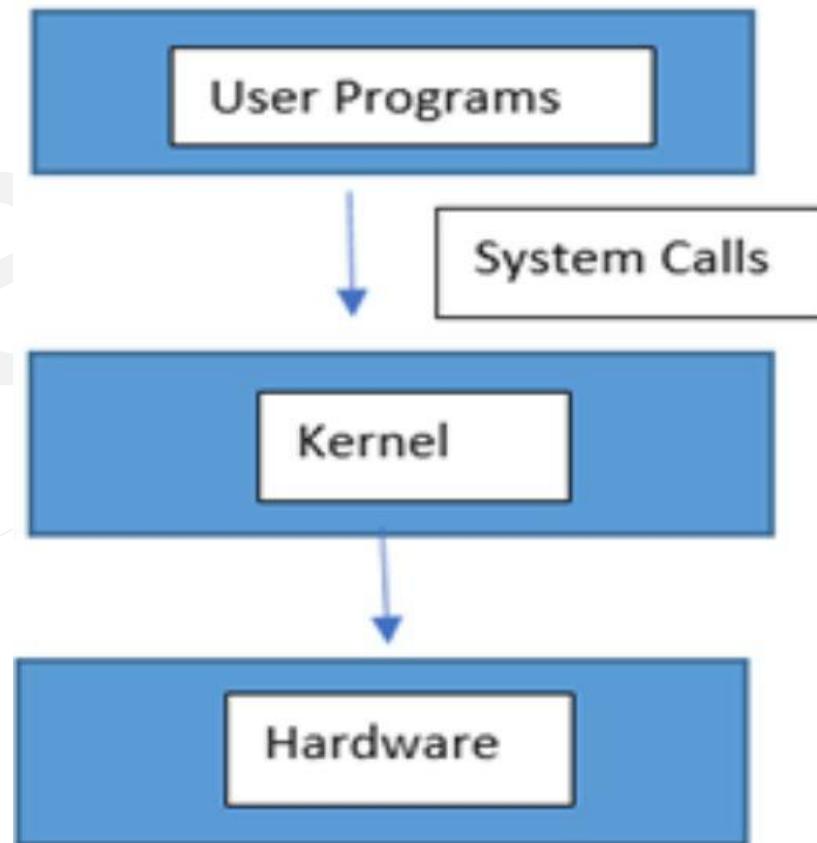
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.



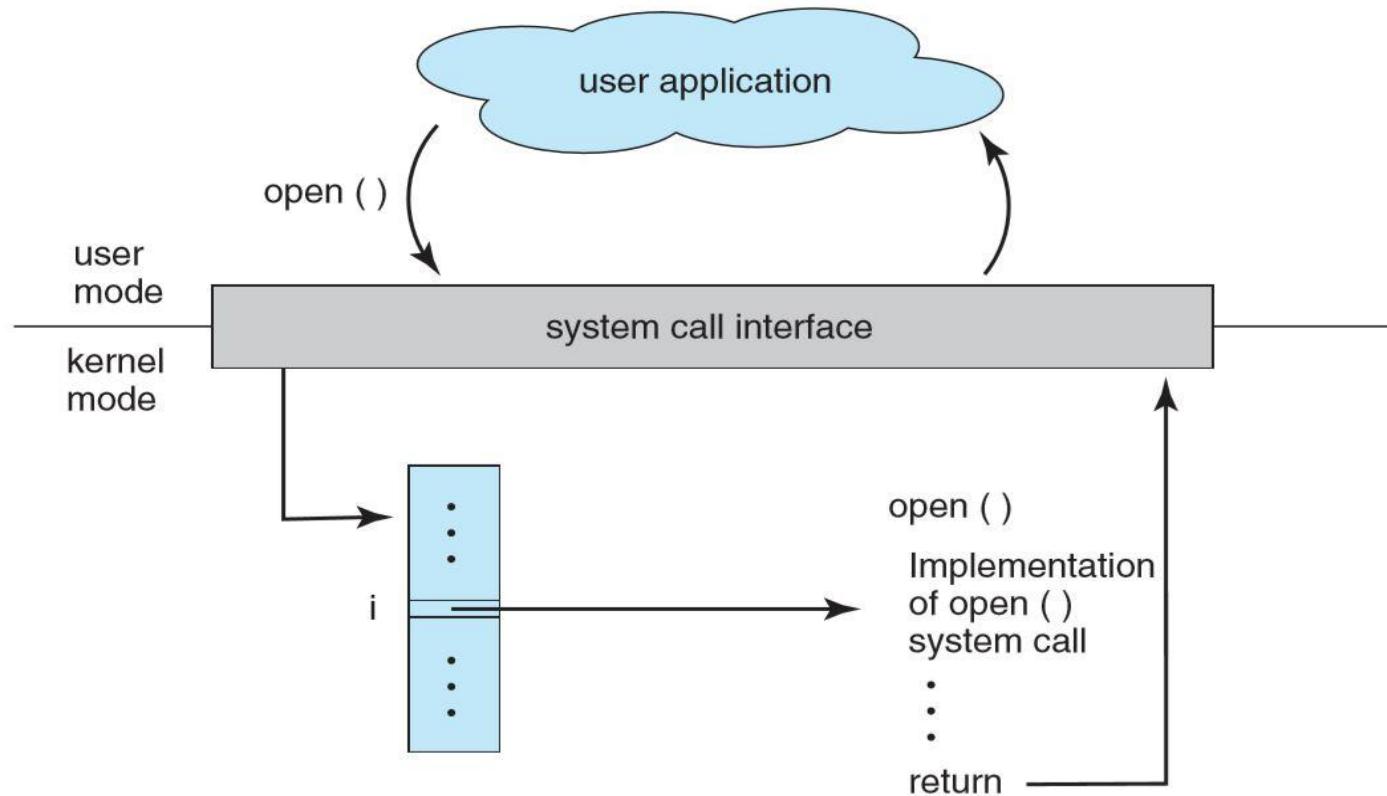
- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The MINIX 3 microkernel, for example, has only approximately 12,000 lines of code.
- Developer Andrew S. Tanenbaum

System call

- System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
- System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++.
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

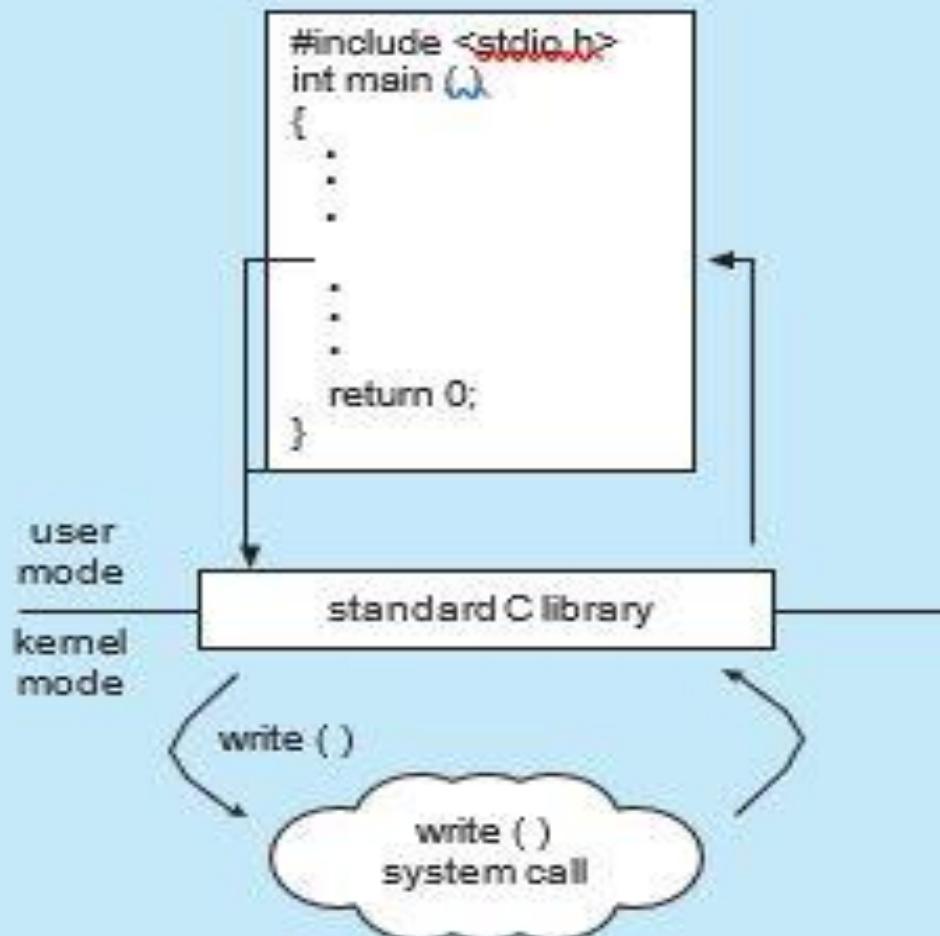


- Three of the most common APIs available to application programmers are the
 - Windows API for Windows systems
 - POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OS X)
 - The Java API for programs that run on the Java virtual machine.
- The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.



EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:

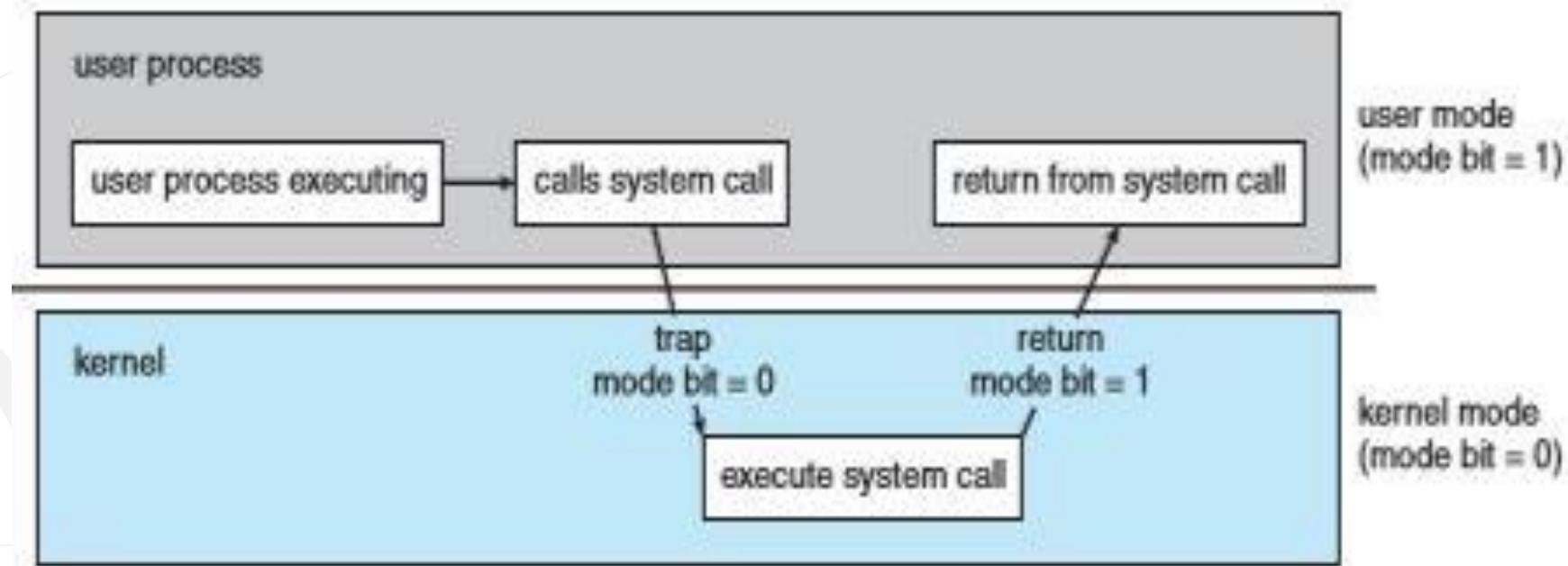


- **Types of System Calls** - System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**.
- **Process control**
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- **File management**
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes

- **Device management**
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- **Information maintenance**
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes
- **Communications**
 - create, delete communication connection
 - send, receive messages transfer status information

Mode bit

- In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code.
- At the very least, we need two separate *modes* of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).
- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.





User Mode



Kernel Mode

Farmer



Instruction set programmer

Industry(Gov / Private)



OS designer(API)

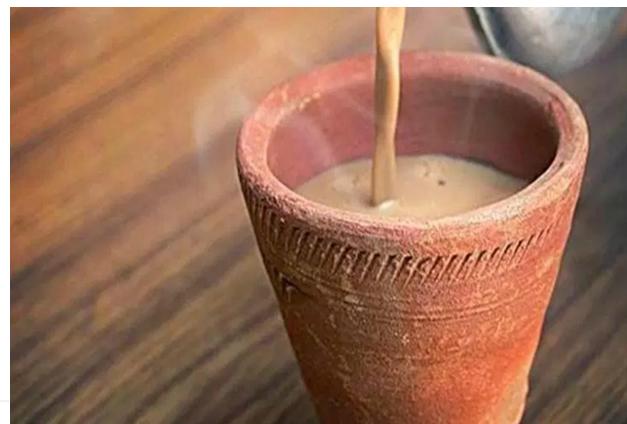




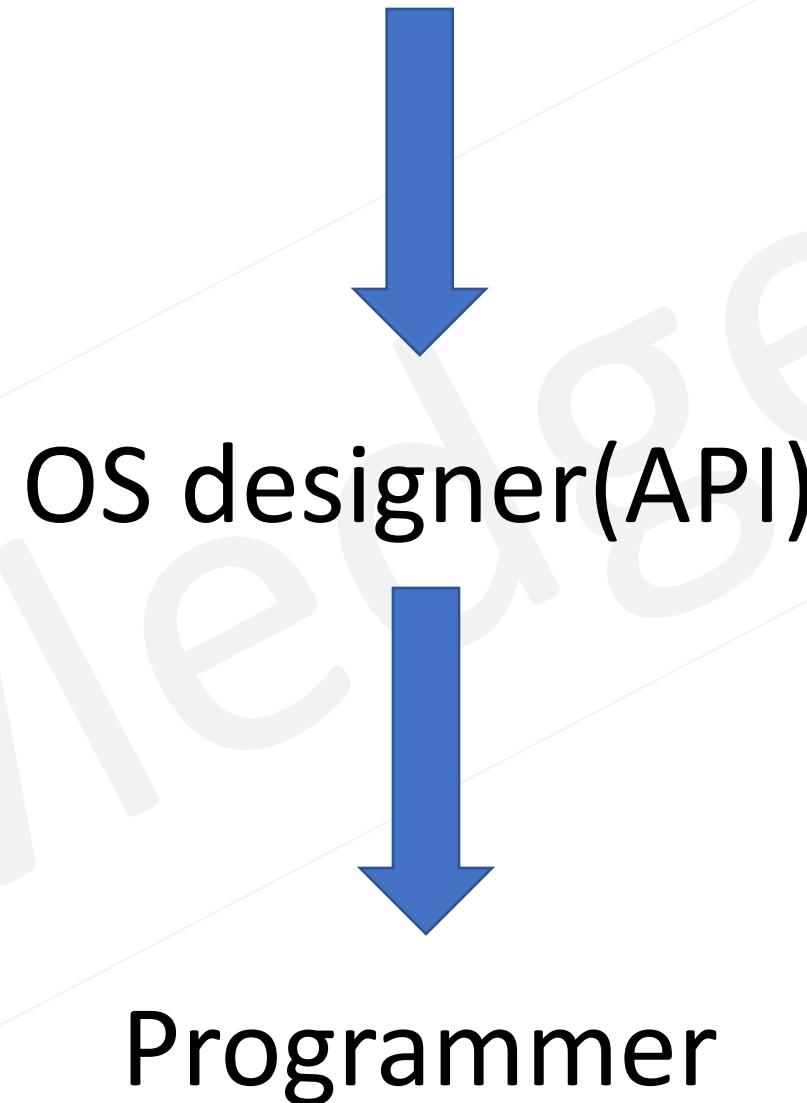
Chef



Programmer



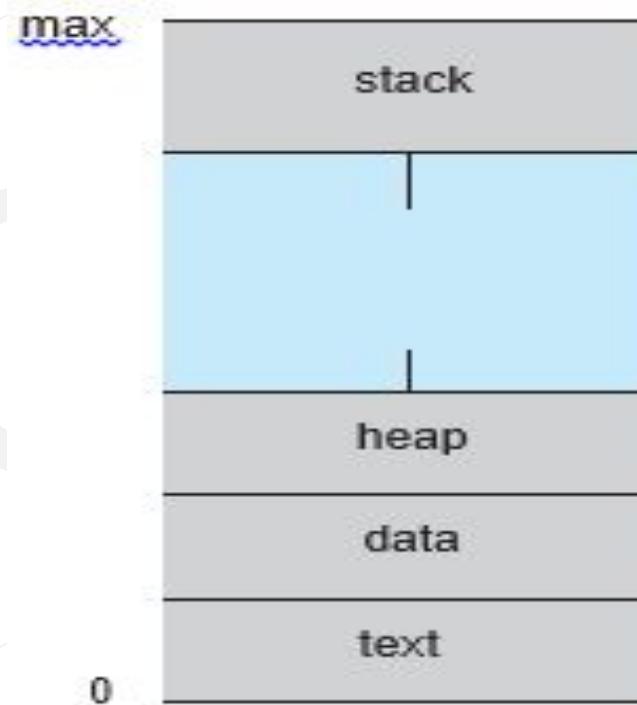
Instruction set programmer



Process

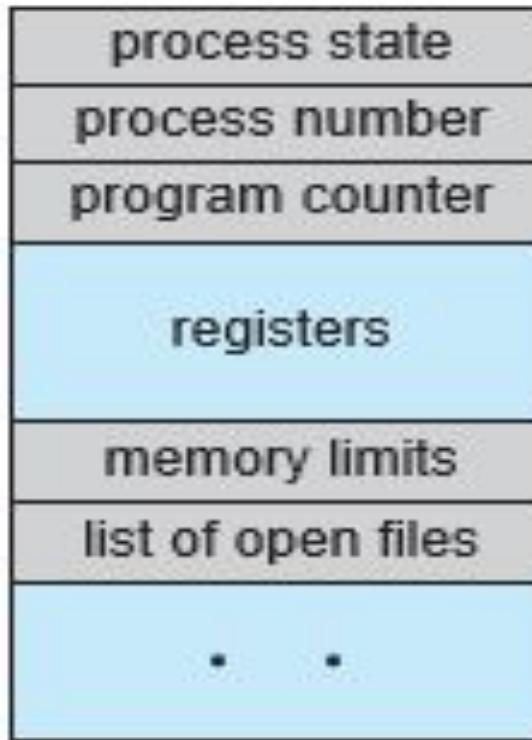
- In general, a process is a program in execution.
- A program is not inherently a process. A program is a passive entity, meaning it is a file containing a list of instructions stored on disk (secondary memory) and is often referred to as an executable file.
- A program becomes a process when the executable file is loaded into main memory, and its Process Control Block (PCB) is created.
- Conversely, a process is an active entity that requires resources like main memory, CPU time, registers, system buses, etc. Even if two processes are associated with the same program, they are considered separate execution sequences and are entirely different processes.
- For instance, if a user has multiple copies of a web browser program running, each copy will be treated as a separate process. Although the text section is the same, the data, heap, and stack sections can vary.

- A Process consists of following sections:
 - **Text section:** also known as Program Code.
 - **Stack:** which contains the temporary data (Function Parameters, return addresses and local variables).
 - **Data Section:** Containing global variables.
 - **Heap:** which is memory dynamically allocated during process runtime.



Process Control Block (PCB)

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



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

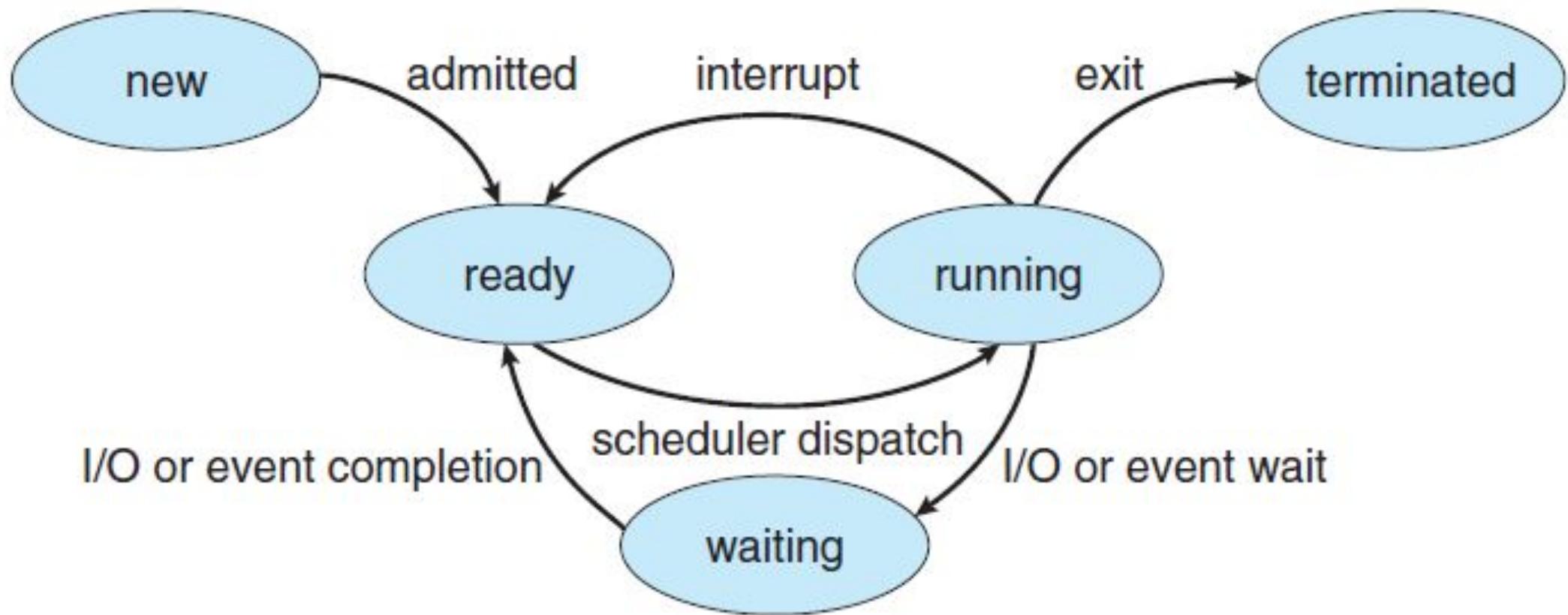


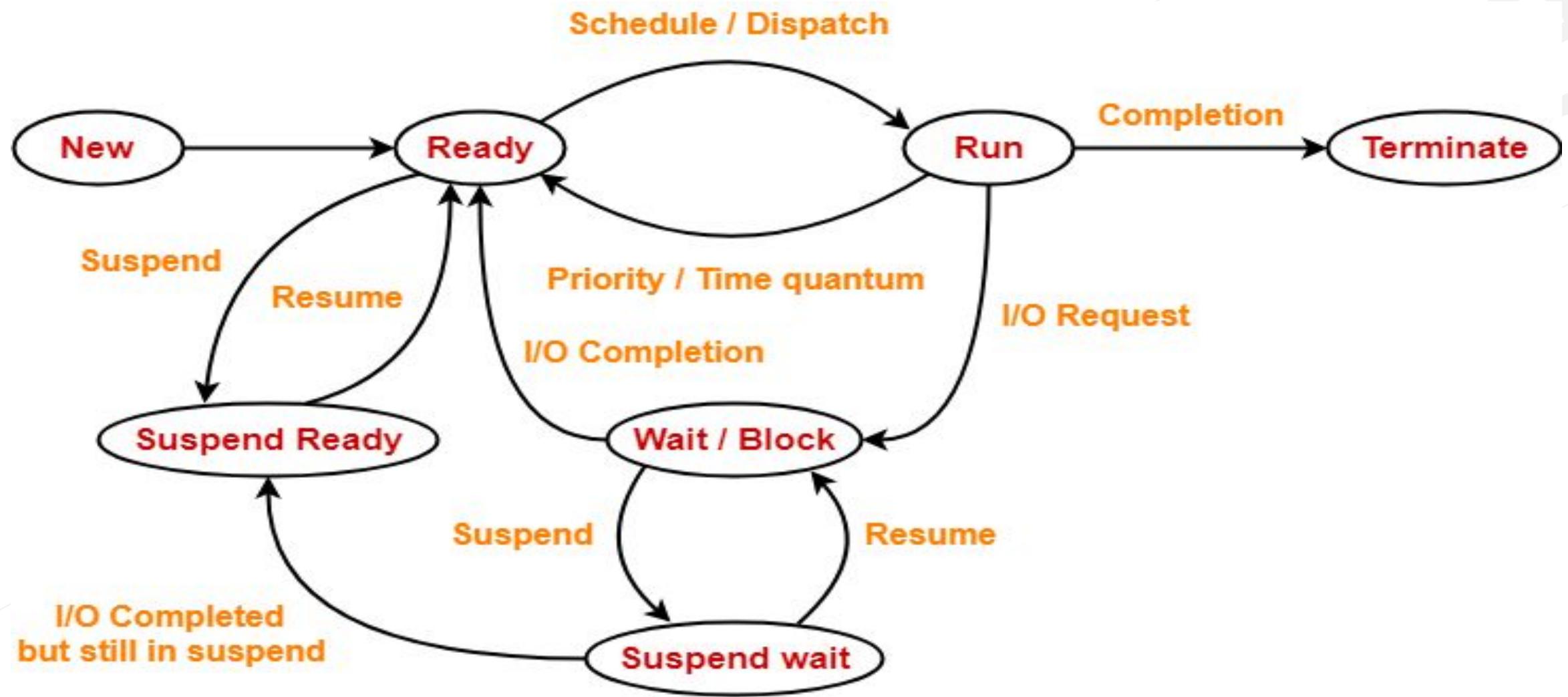
The Human Life Cycle



Process States

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

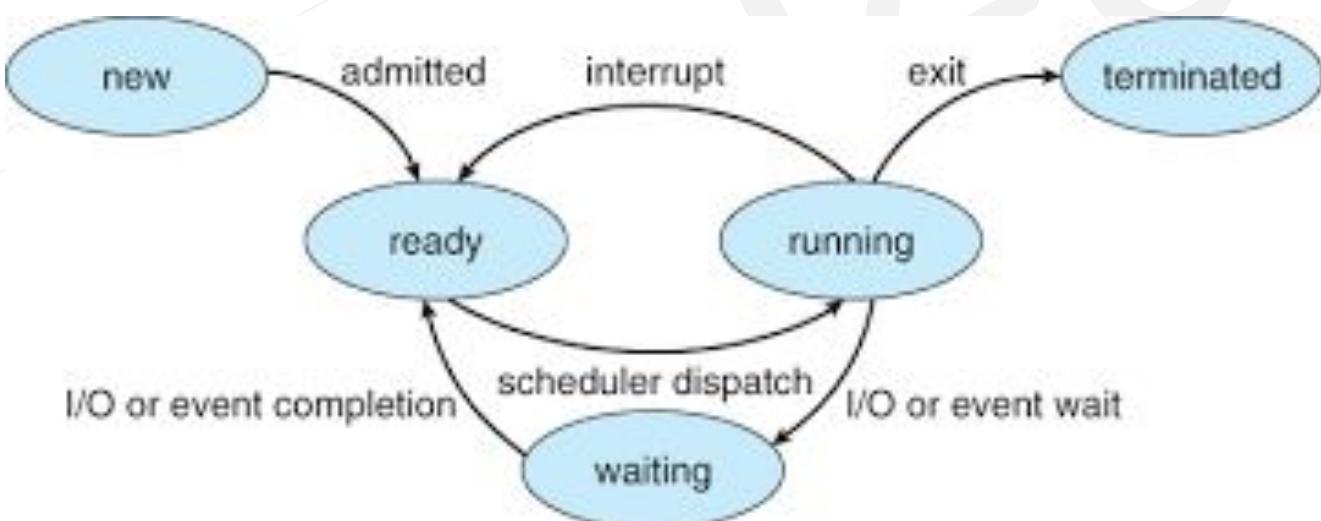




Process State Diagram

Q What is the minimum and maximum number of processes that can be in the ready, run, and blocked states, if total number of process is n?

| | Min | Max |
|-------|-----|-----|
| Ready | | |
| Run | | |
| Block | | |



Q On a system with P CPUs and n processes, what is the minimum and maximum number of processes that can be in the ready, run, and blocked states, assuming $p << n$?

| | Min | Max |
|-------|-----|-----|
| Ready | | |
| Run | | |
| Block | | |

Q What is the ready state of a process?

- a) When process is scheduled to run after some execution
- b) When process is using the CPU
- c) When process is unable to run until some task has been completed
- d) None of the mentioned

Q A task in a blocked state

- a) Is executable
- b) Is running
- c) Must still be placed in the run queries
- d) Is waiting for some temporarily unavailable resources

Q Which combination of the following features will suffice to characterize an OS as a multi-programmed OS? **(GATE-2002) (2 Marks)**

- (a)** More than one program may be loaded into main memory at the same time for execution.
 - (b)** If a program waits for certain events such as I/O, another program is immediately scheduled for execution.
 - (c)** If the execution of program terminates, another program is immediately scheduled for execution.
- (A)** a
- (B)** a and b
- (C)** a and c
- (D)** a, b and c

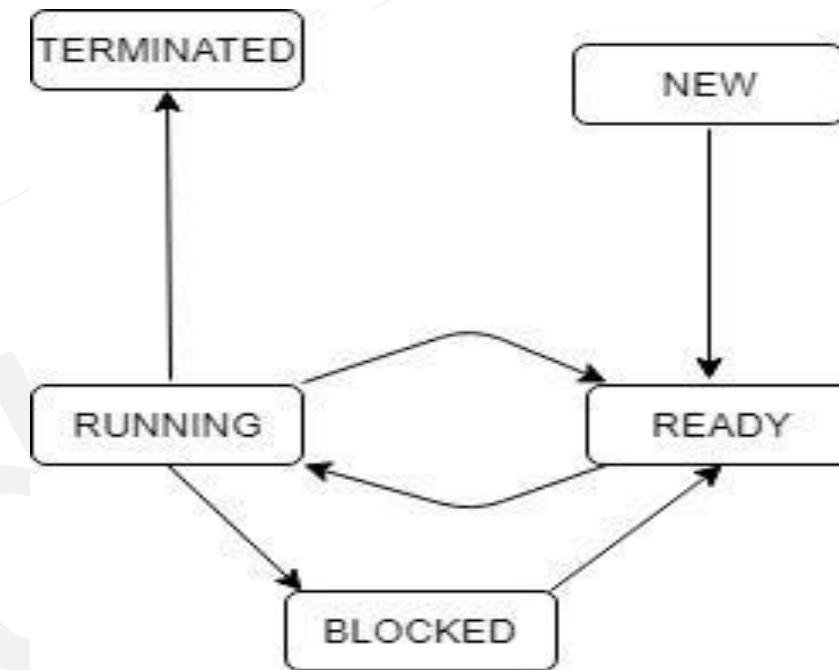
Q The process state transition diagram in below figure is representative of Untitled Diagram (**GATE-1996**) (1 Marks)

a) a batch operating system

b) an operating system with a preemptive scheduler

c) an operating system with a non-preemptive scheduler

d) a uni-programmed operating system



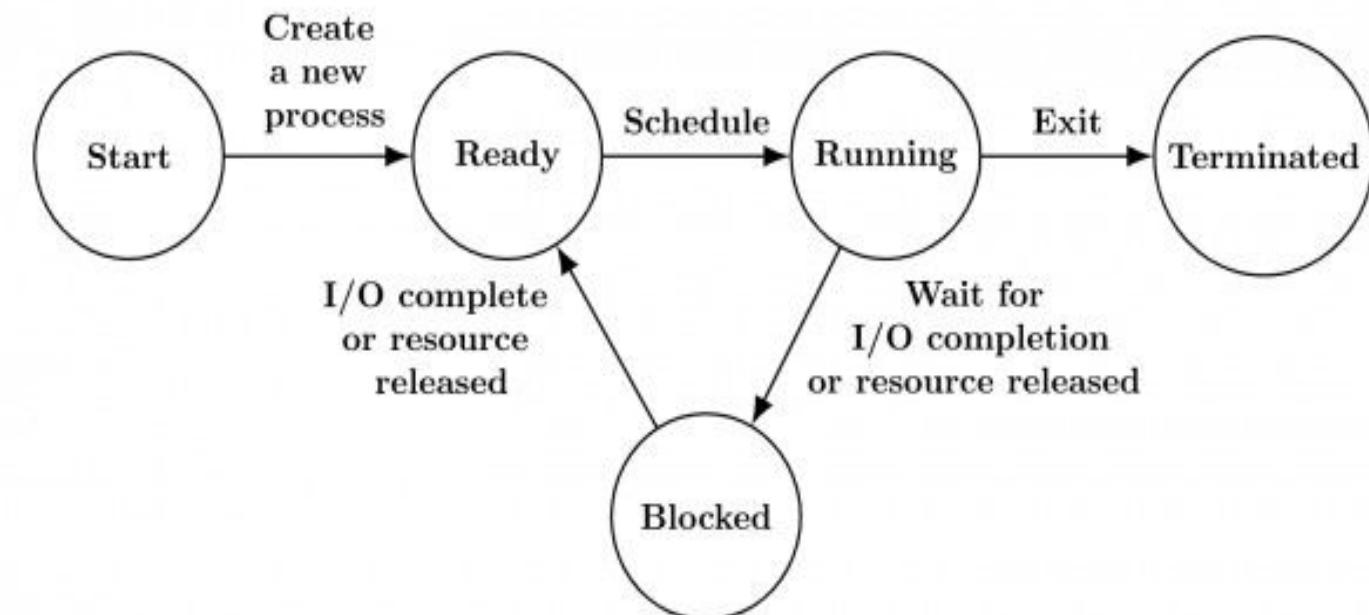
Q The process state transition diagram of an operating system is as given below. Which of the following must be FALSE about the above operating system?
(GATE-2006) (1 Marks)

a) It is a multiprogram operating system

b) It uses preemptive scheduling

c) It uses non-preemptive scheduling

d) It is a multi-user operating system



Q In the following process state transition diagram for a uniprocessor system, assume that there are always some processes in the ready state:

Now consider the following statements:

- I) If a process makes a transition D, it would result in another process making transition A immediately.
- II) A process P2 in blocked state can make transition E while another process P1 is in running state.
- III) The OS uses preemptive scheduling.
- IV) The OS uses non-preemptive scheduling.

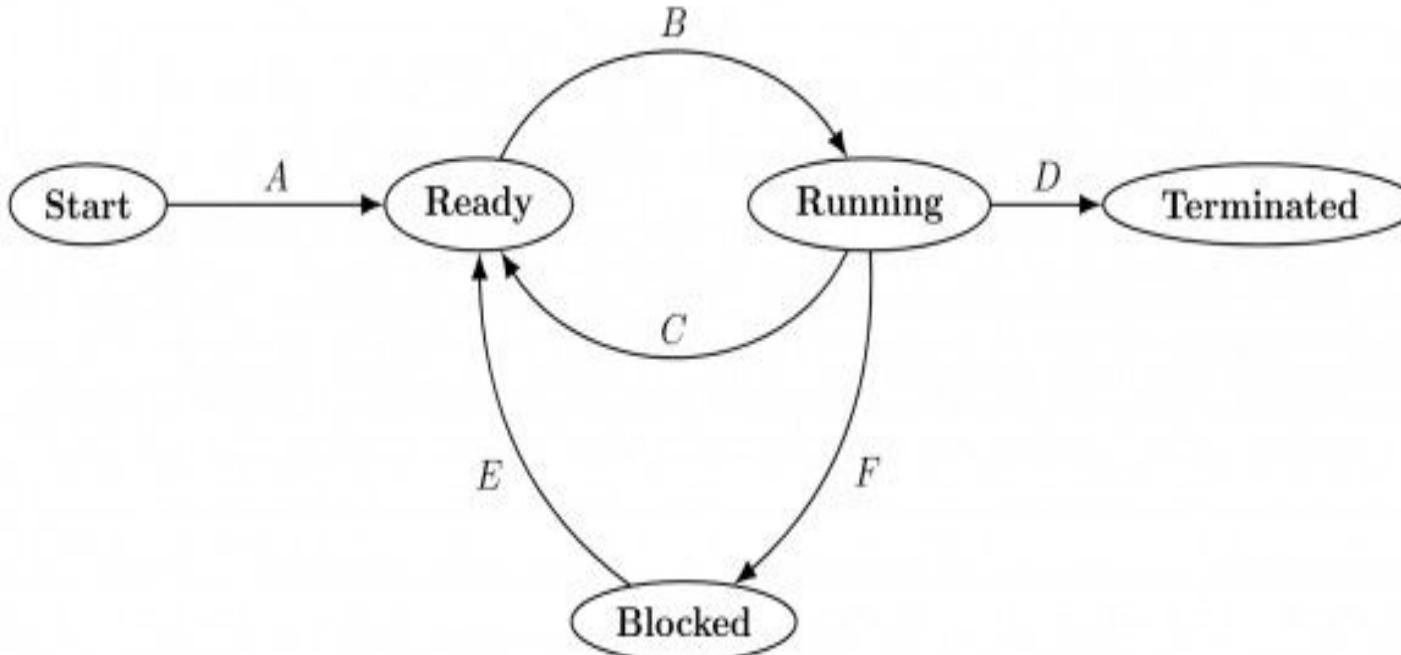
Which of the above statements are TRUE? **(GATE-2009) (2 Marks)**

a) I and II

b) I and III

c) II and III

d) II and IV



Q A computer handles several interrupt sources of which the following are relevant for this question.

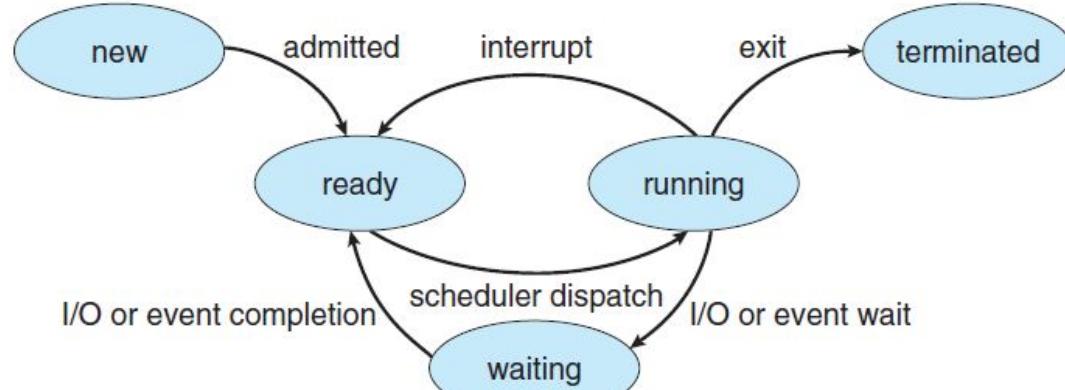
- Interrupt from CPU temperature sensor (raises interrupt if CPU temperature is too high)
- Interrupt from Mouse (raises interrupt if the mouse is moved or a button is pressed)
- Interrupt from Keyboard (raises interrupt when a key is pressed or released)
- Interrupt from Hard Disk (raises interrupt when a disk read is completed)

Which one of these will be handled at the HIGHEST priority? **(GATE - 2011) (1 Marks)**

- (A) Interrupt from Hard Disk
- (B) Interrupt from Mouse
- (C) Interrupt from Keyboard
- (D) Interrupt from CPU temperature sensor

Q.Which of the following process state transitions is /are NOT possible? (Gate 2024,CS) (1 Marks) (MSQ)

- (a) Running to Ready
- (b) Waiting to Running
- (c) Ready to Waiting
- (d) Running to terminated



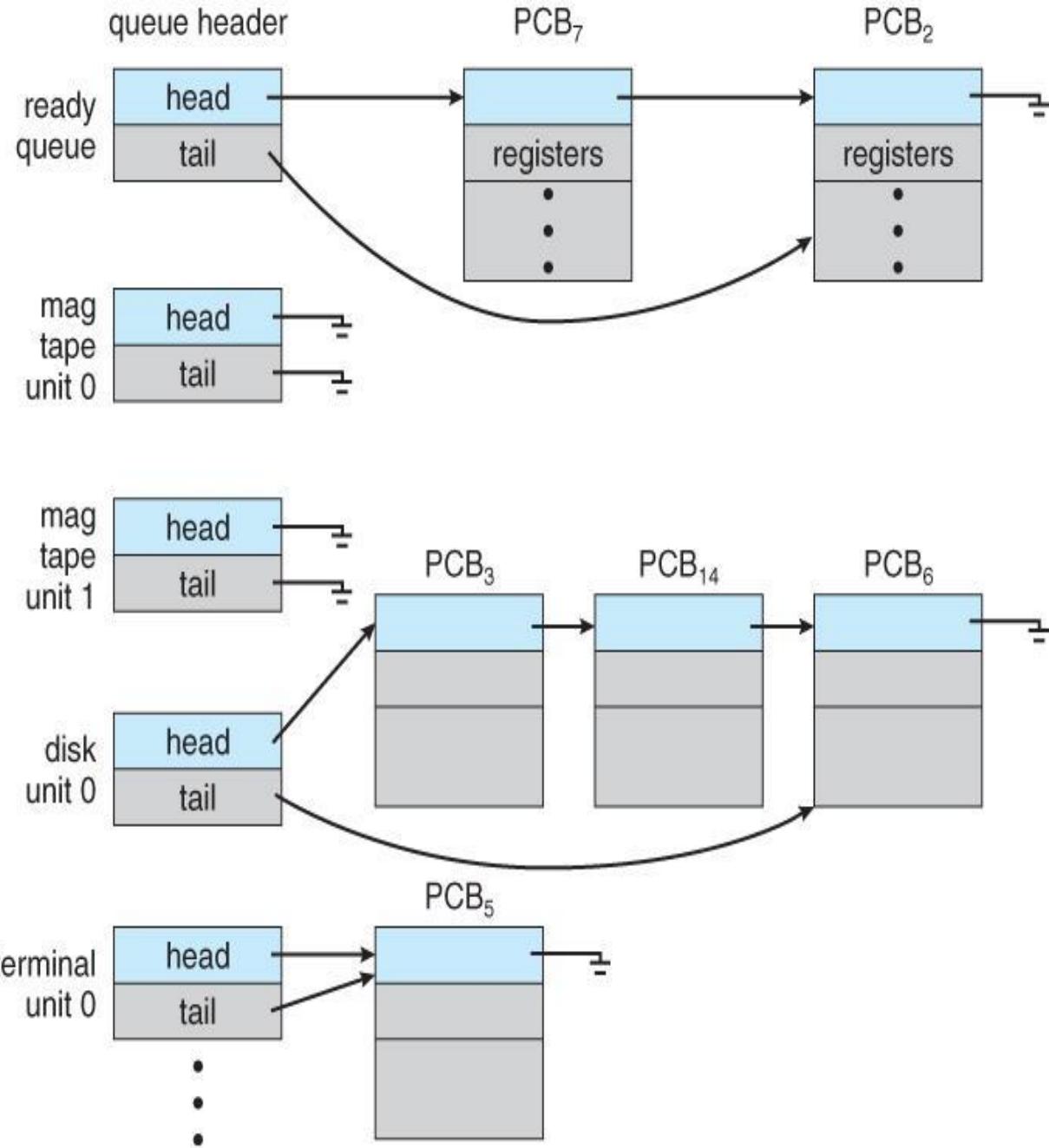
Q. Consider a process P running on a CPU. Which one or more of the following events will always trigger a context switch by the OS that results in process P moving to a non-running state (e.g., ready, blocked)? **(Gate 2024 CS)**

- (a) P makes a blocking system call to read a block of data from the disk
- (b) P tries to access a page that is in the swap space, triggering a page fault
- (c) An interrupt is raised by the disk to deliver data requested by some other process
- (d) A timer interrupt is raised by the hardware

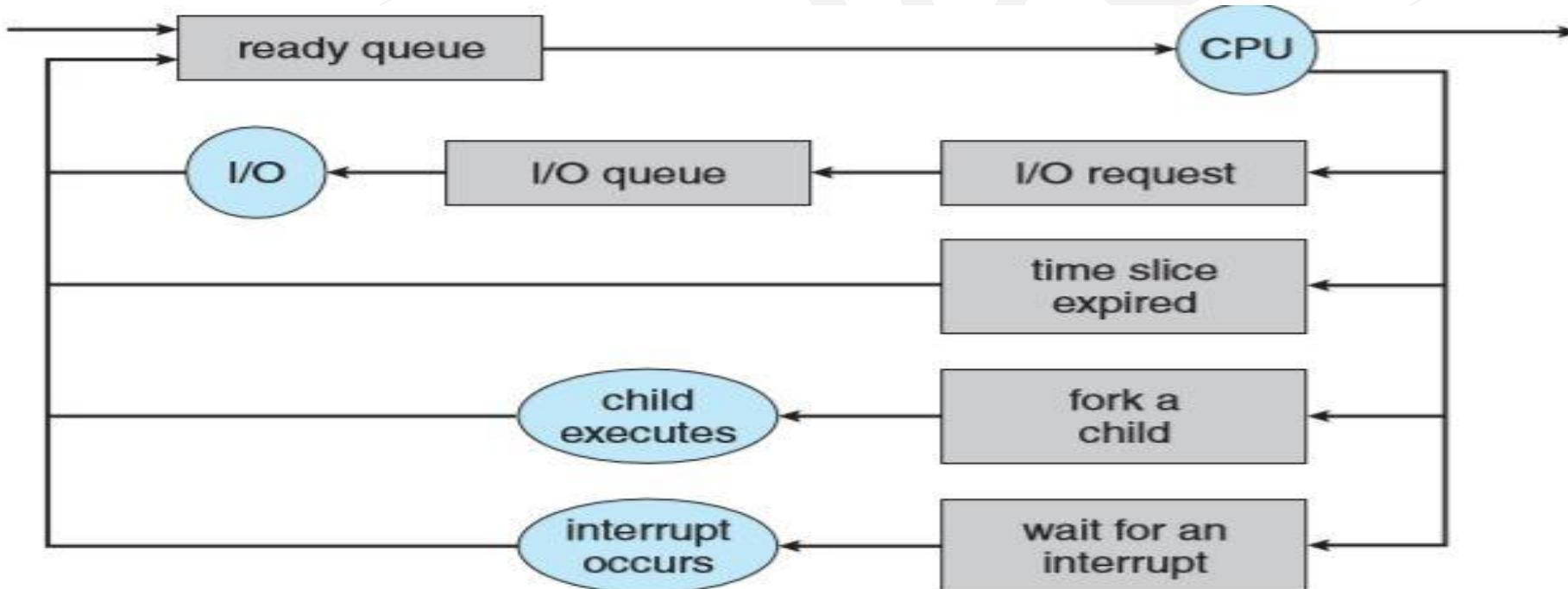
Q. Suppose in a multiprogramming environment, the following C program segment is executed. A process goes into I/O queue whenever an I/O related operation is performed. Assume that there will always be a context switch whenever a process requests for an I/O, and also whenever the process returns from an I/O. The number of times the process will enter the ready queue during its lifetime (not counting the time the process enters the ready queue when it is run initially) is _____. (Answer in integer)? **(Gate 2025)**

```
int main() {  
    int x=0,i=0;  
    scanf("%d",&x);  
    for(i=0; i<20; i++)  
    {  x = x+20;  
       printf("%d\n",x);  
    } return 0;  
}
```

- **Job Queue:** Contains all processes in the system. It is the initial queue where processes are placed as they enter the system.
- **Ready Queue:** Holds processes that are in main memory and ready to execute. This queue is typically implemented as a linked list, with a ready-queue header containing pointers to the first and last process control blocks (PCBs) in the list. Each PCB has a pointer to the next PCB in the ready queue.
- **Device Queues:** These are used when processes require I/O operations and the requested I/O device is busy. Each device (like a disk) has its own queue holding processes that are waiting for it to become available.

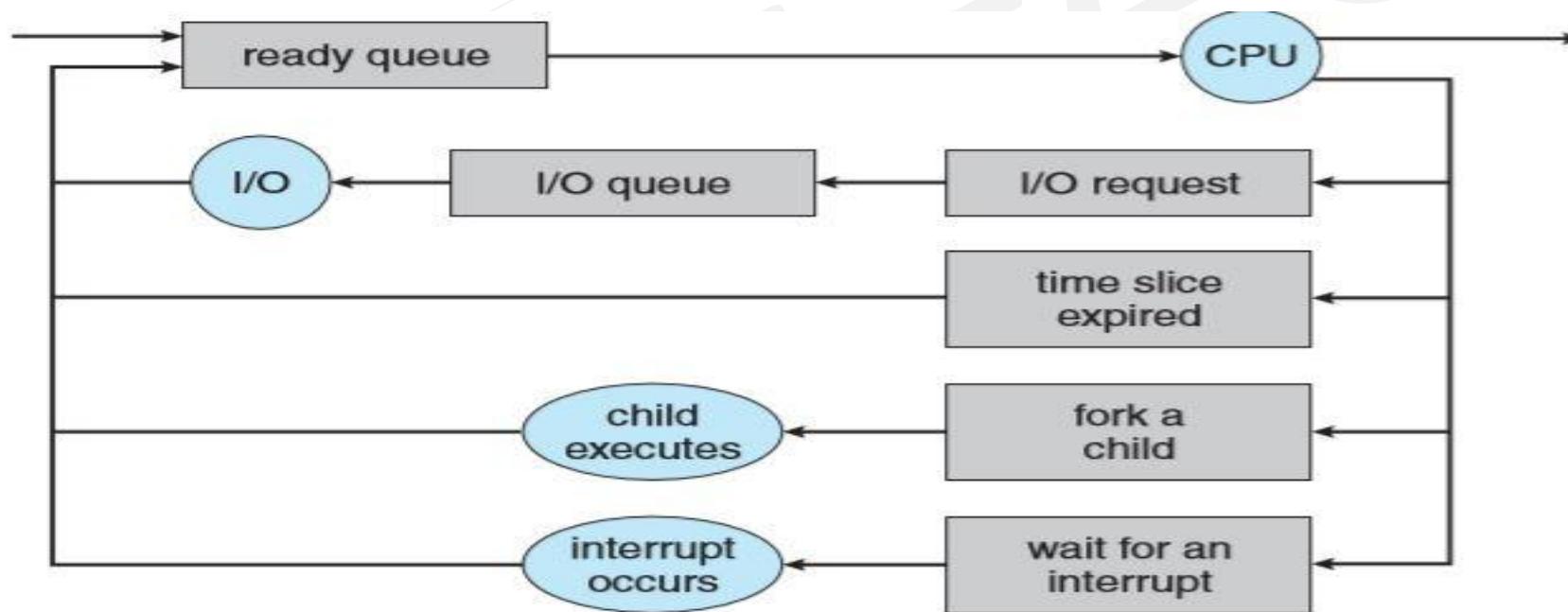


- Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues. A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched.
- Once the process is allocated the CPU and is executing, one of several events could occur:
 - The process could issue an I/O request and then be placed in an I/O queue.
 - The process could create a new child process and wait for the child's termination.
 - Time slice expired, the process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



Queueing-diagram representation of process scheduling.

- In the first two cases the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.



Queueing-diagram representation of process scheduling.

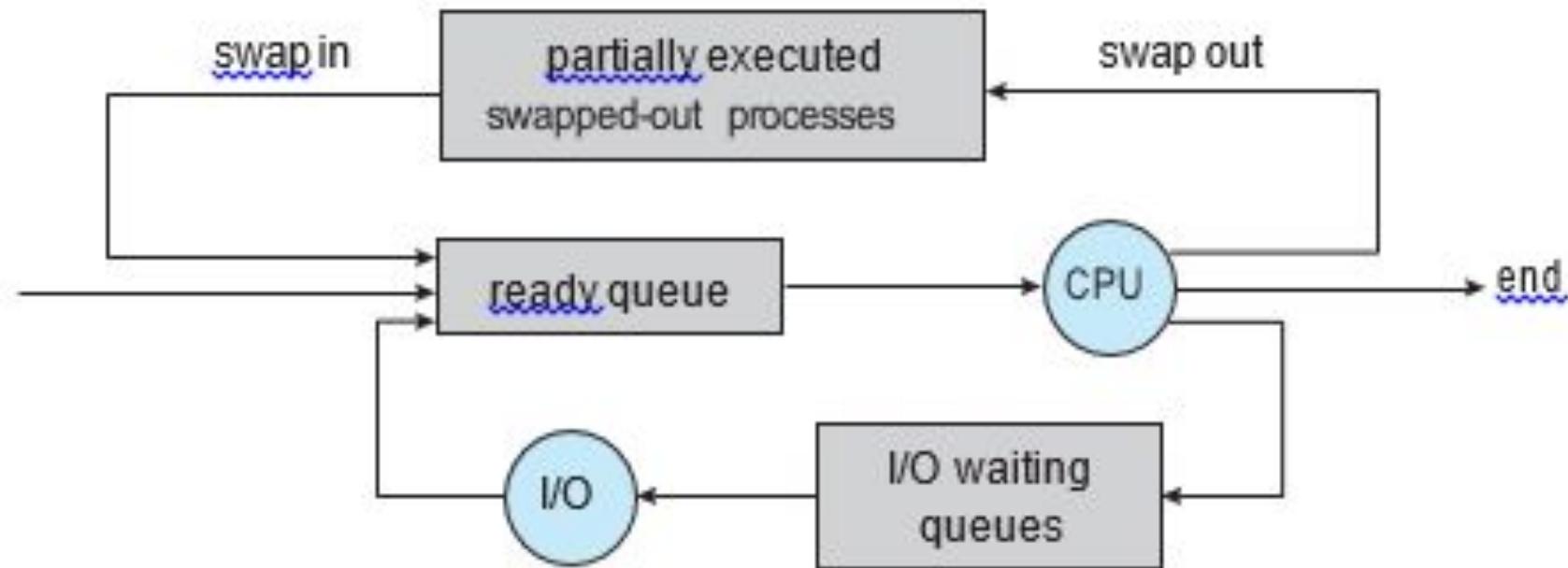
Schedulers

- **Schedulers**: A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.
- **Types of Schedulers**
 - **Long Term Schedulers (LTS)/Spooler**: In multiprogramming os, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.
 - **Short Term Scheduler (STS)**: The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

- **Difference between LTS and STS** - The primary distinction between these two schedulers lies in frequency of execution.
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds.
- Because of the short time between executions, the short-term scheduler must be fast. The long-term scheduler on the other hand executes much less frequently; minutes may separate the creation of one new process and the next.

Degree of Multiprogramming - The number of processes in memory is known as Degree of Multiprogramming.

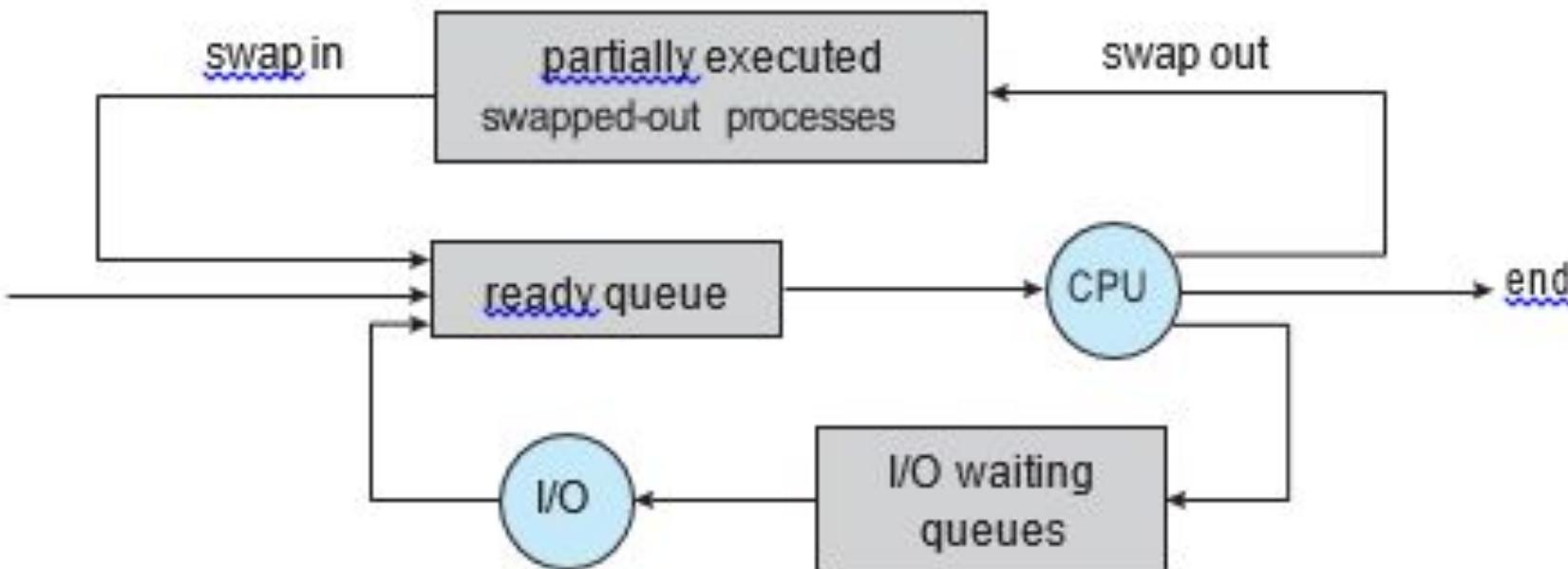
- The long-term scheduler controls the degree of multiprogramming as it is responsible for bringing in the processes to main memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. So, this means the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.



Medium-term scheduler: The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



- **Dispatcher** - The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves the following: Switching context, switching to user mode, jumping to the proper location in the user program to restart that program.
- The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

CPU Bound and I/O Bound Processes

- A process execution consists of a cycle of CPU execution or wait and i/o execution or wait. Normally a process alternates between two states.
- Process execution begin with the CPU burst that may be followed by a i/o burst, then another CPU and i/o burst and so on. Eventually in the last will end up on CPU burst. So, process keep switching between the CPU and i/o during execution.

- **I/O Bound Processes:** An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- **CPU Bound Processes:** A CPU-bound process, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- Similarly, if all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. So, to have the best system performance LTS needs to select a good combination of I/O and CPU Bound processes.

Q Which of the following does not interrupt a running process? (GATE-2001) (2 Marks)

(A) A device

(B) Timer

(C) Scheduler process

(D) Power failure

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching.
- Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

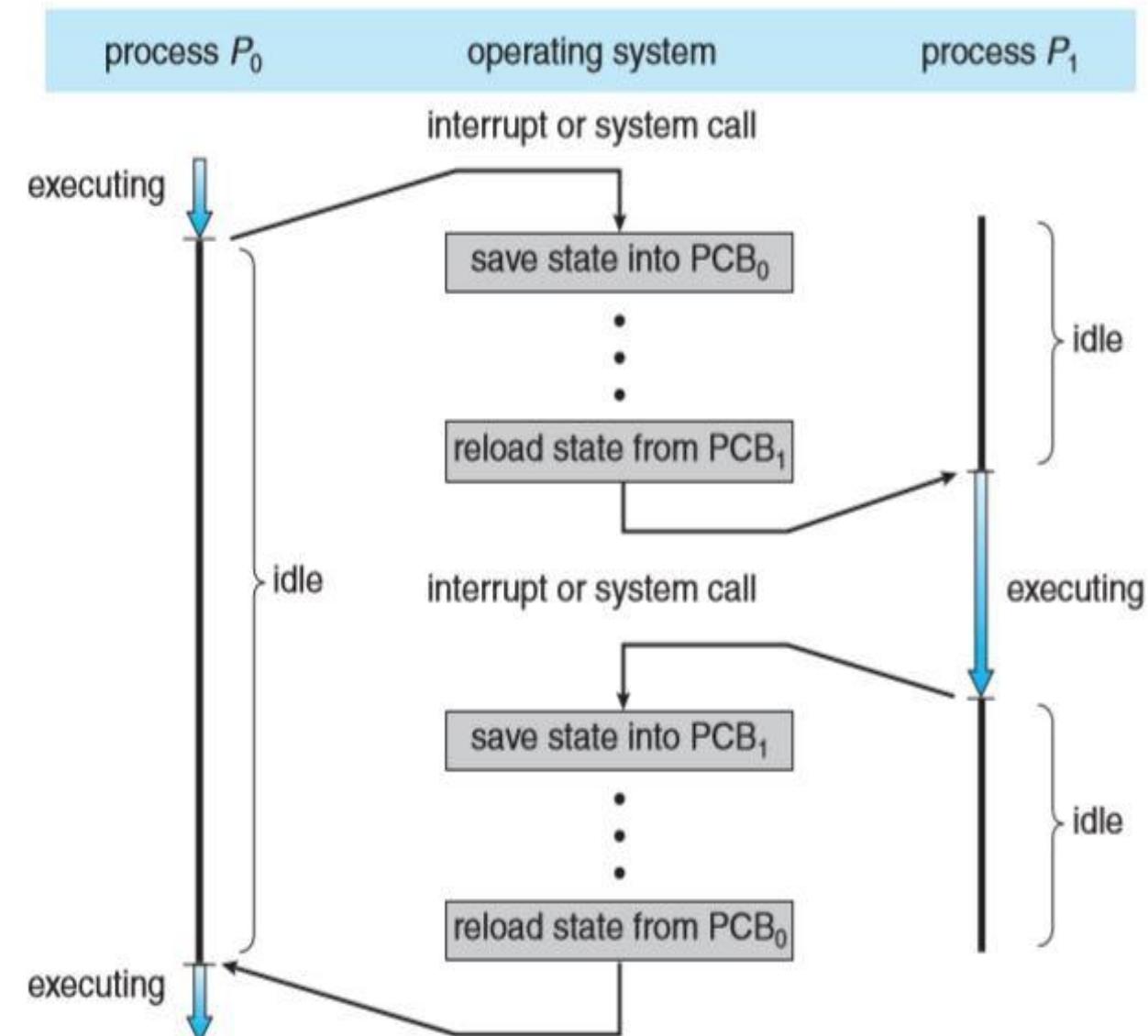
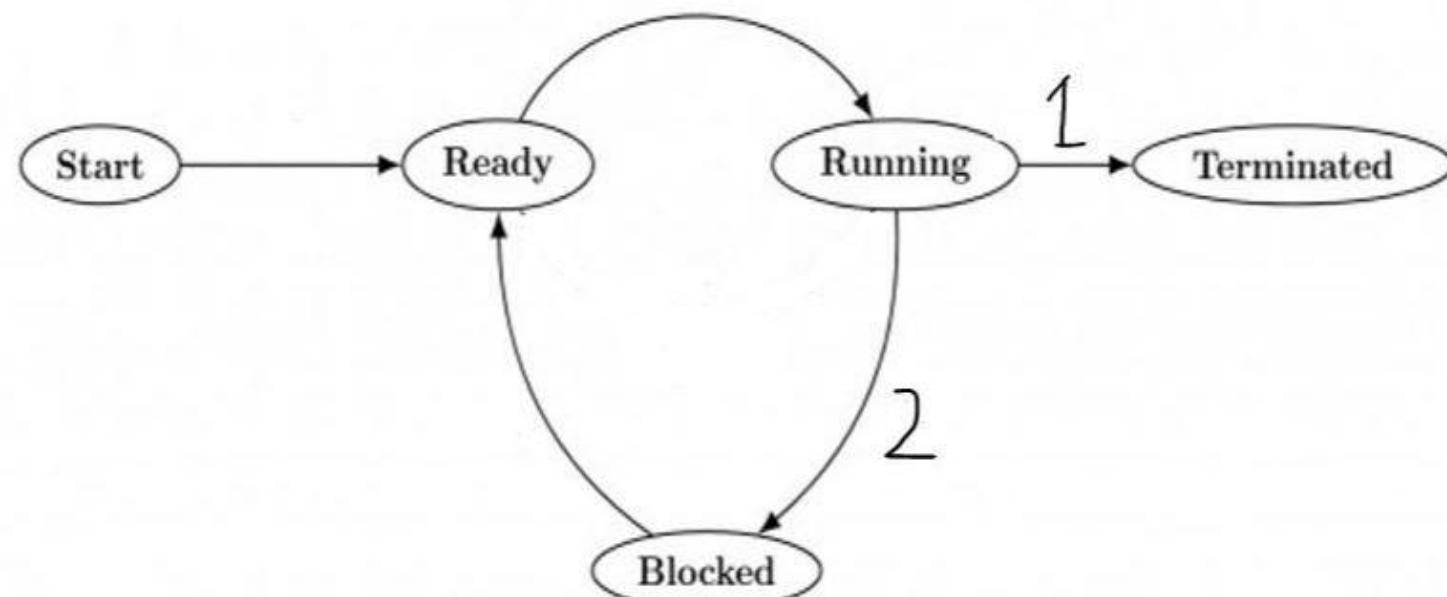


Diagram showing CPU switch from process to process.

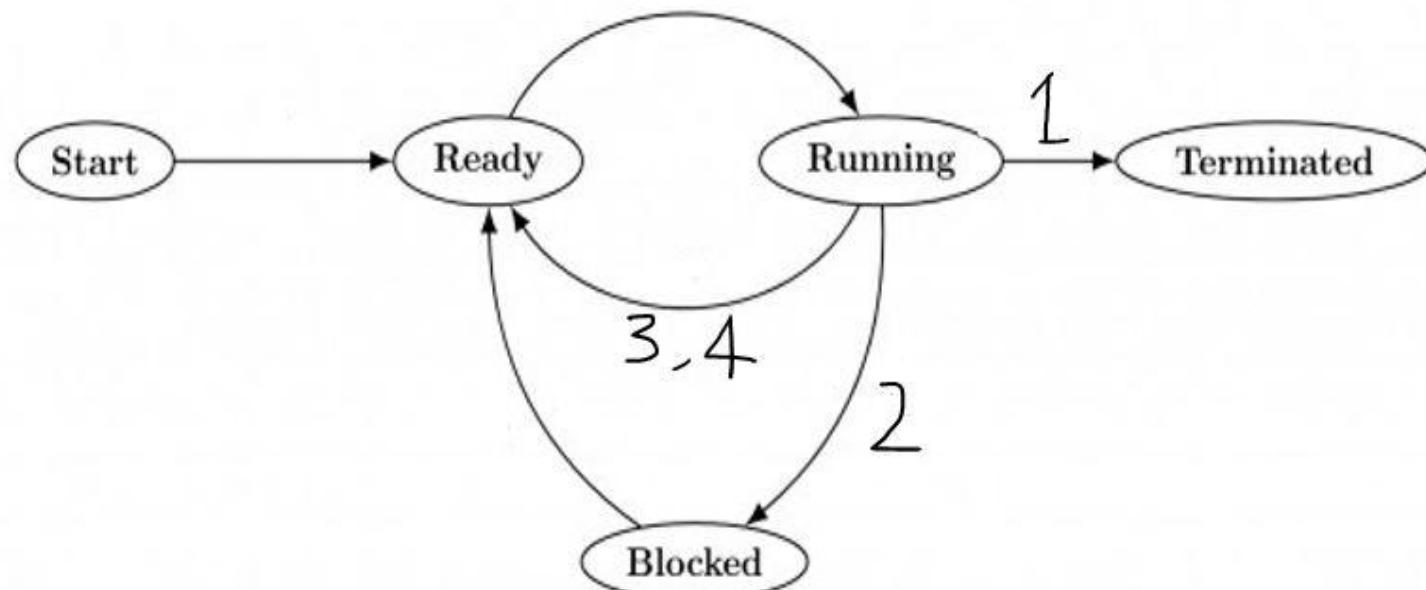
Type of scheduling

- **Non-Pre-emptive**: Under Non-Pre-emptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU willingly.
- A process will leave the CPU only
 1. When a process completes its execution (Termination state)
 2. When a process wants to perform some i/o operations(Blocked state)



Pre-emptive

- Under Pre-emptive scheduling, once the CPU has been allocated to a process, A process will leave the CPU willingly or it can be forced out. So it will leave the CPU
 - When a process completes its execution
 - When a process leaves CPU voluntarily to perform some i/o operations
 - If a new process enters in the ready states (new, waiting), in case of high priority
 - When process switches from running to ready state because of time quantum expire.



- **Scheduling criteria** - Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. So, in order to efficiently select the scheduling algorithms following criteria should be taken into consideration:

- **CPU utilization:** Keeping the CPU as busy as possible.



- **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput.



- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.



- **Response Time**: Is the time it takes to start responding, not the time it takes to output the response.



- Note: The CPU-scheduling algorithm does not affect the amount of time during which a process executes I/O; it affects only the amount of time that a process spends waiting in the ready queue.
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Terminology

- **Arrival Time (AT):** Time at which process enters a ready state.
- **Burst Time (BT):** Amount of CPU time required by the process to finish its execution.
- **Completion Time (CT):** Time at which process finishes its execution.
- **Turn Around Time (TAT):** Completion Time (CT) – Arrival Time (AT), Waiting Time + Burst Time (BT)
- **Waiting Time:** Turn Around Time (TAT) – Burst Time (BT)

FCFS (FISRT COME FIRST SERVE)

- FCFS is the simplest scheduling algorithm, as the name suggest, the process that requests the CPU first is allocated the CPU first.
- Implementation is managed by FIFO Queue.
- It is always non pre-emptive in nature.



| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 2 | 4 | | | |
| P ₁ | 1 | 2 | | | |
| P ₂ | 0 | 3 | | | |
| P ₃ | 4 | 2 | | | |
| P ₄ | 3 | 1 | | | |
| Average | | | | | |

| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 0 | 3 | | | |
| P ₁ | 2 | 2 | | | |
| P ₂ | 6 | 4 | | | |
| Average | | | | | |

Advantage

- Easy to understand, and can easily be implemented using Queue data structure.
- Can be used for Background processes where execution is not urgent.

| P. No | AT | BT | TAT=CT-AT | WT=TAT -BT |
|---------|----|-----|-----------|------------|
| P_0 | 0 | 100 | | |
| P_1 | 1 | 2 | | |
| Average | | | | |

| P. No | AT | BT | TAT=CT-AT | WT=TAT -BT |
|---------|----|-----|-----------|------------|
| P_0 | 1 | 100 | | |
| P_1 | 0 | 2 | | |
| Average | | | | |

Convoy Effect

- If the smaller process have to wait more for the CPU because of Larger process then this effect is called Convoy Effect, it result into more average waiting time.
- Solution, smaller process have to be executed before longer process, to achieve less average waiting time.



Disadvantage

- FCFS suffers from convoy which means smaller process have to wait larger process, which result into large average waiting time.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems (due to its non-pre-emptive nature), where it is important that each user get a share of the CPU at regular intervals.
- Higher average waiting time and TAT compared to other algorithms.

Shortest Job First (SJF)(non-pre-emptive)

Shortest Remaining Time First (SRTF)/ (Shortest Next CPU Burst) (Pre-emptive)

- Whenever we make a decision of selecting the next process for CPU execution, out of all available process, CPU is assigned to the process having smallest burst time requirement. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If there is a tie, FCFS is used to break tie.
- It supports both version non-pre-emptive and pre-emptive (purely greedy approach)

- In Shortest Job First (SJF)(non-pre-emptive) once a decision is made and among the available process, the process with the smallest CPU burst is scheduled on the CPU, it cannot be pre-empted even if a new process with the smaller CPU burst requirement then the remaining CPU burst of the running process enter in the system.
- In Shortest Remaining Time First (SRTF) (Pre-emptive) whenever a process enters in ready state, again we make a scheduling decision weather, this new process with the smaller CPU burst requirement then the remaining CPU burst of the running process and if it is the case then the running process is pre-empted and new process is scheduled on the CPU.
- This version (SRTF) is also called optimal is it guarantee minimal average waiting time.

| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 1 | 7 | | | |
| P ₁ | 2 | 5 | | | |
| P ₂ | 3 | 1 | | | |
| P ₃ | 4 | 2 | | | |
| P ₄ | 5 | 8 | | | |
| Average | | | | | |

| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 0 | 6 | | | |
| P ₁ | 1 | 4 | | | |
| P ₂ | 2 | 3 | | | |
| P ₃ | 3 | 1 | | | |
| P ₄ | 4 | 2 | | | |
| P ₅ | 5 | 1 | | | |
| Average | | | | | |

In Shortest Remaining Time First (SRTF) (Pre-emptive) whenever a process enters in ready state, again we make a scheduling decision whether, this new process with the smaller CPU burst requirement then the remaining CPU burst of the running process and if it is the case then the running process is pre-empted and new process is scheduled on the CPU.

This version (SRTF) is also called Optimal is it guarantee minimal average waiting time.

- **Advantage**

- Pre-emptive version guarantees minimal average waiting time so some time also referred as optimal algorithm.
- Provide a standard for other algo in terms of average waiting time
- Provide better average response time compare to FCFS

- **Disadvantage**

- This algo cannot be implemented as there is no way to know the length of the next CPU burst.
- Here process with the longer CPU burst requirement goes into starvation.
- No idea of priority, longer process has poor response time.



- As SJF is not implementable, we can use the one technique where we try to predict the CPU burst of the next coming process.
- The method is used as exponential averaging technique, where we consider the previous value and previous prediction.

| Process | t | tau |
|----------------|----|-----|
| P ₁ | 10 | 20 |
| P ₂ | 12 | |
| P ₃ | 14 | |
| P ₄ | | |

$$\text{Tau}_{(n+1)} = \alpha t_n + (1 - \alpha) \text{tau}_n$$

- This idea is also more of theoretical importance as most of the time the burst requirement of the coming process may vary by a large extent and if the burst time requirement of all the process is approximately same then there is no advantage of using this scheme.

Q Consider an arbitrary set of CPU-bound processes with unequal CPU burst lengths submitted at the same time to a computer system. Which one of the following process scheduling algorithms would minimize the average waiting time in the ready queue? **(GATE - 2016) (1 Marks)**

- (a)** Shortest remaining time first
- (b)** Round-robin with time quantum less than the shortest CPU burst
- (c)** Uniform random
- (d)** Highest priority first with priority proportional to CPU burst length

Q For the processes listed in the following table, which of the following scheduling schemes will give the lowest average turnaround time? (GATE-2015) (2 Marks)

| Process | Arrival Time | Processing Time |
|---------|--------------|-----------------|
| A | 0 | 3 |
| B | 1 | 6 |
| C | 4 | 4 |
| D | 6 | 2 |

- (A) First Come First Serve
- (B) Non-pre-emptive Shortest Job First
- (C) Shortest Remaining Time
- (D) Round Robin with Quantum value two

Q Three processes arrive at time zero with CPU bursts of 16, 20 and 10 milliseconds. If the scheduler has prior knowledge about the length of the CPU bursts, the minimum achievable average waiting time for these three processes in a non-preemptive scheduler (rounded to nearest integer) is _____ milliseconds. **(GATE 2021) (1 MARKS)**

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 16 | | | |
| P ₂ | 0 | 20 | | | |
| P ₃ | 0 | 10 | | | |

Q Consider the following four processes with arrival times (in milliseconds) and their length of CPU bursts (in milliseconds) as shown below: (GATE - 2019) (2 Marks)

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 3 | | | |
| P ₂ | 1 | 1 | | | |
| P ₃ | 3 | 3 | | | |
| P ₄ | 4 | Z | | | |

These processes are run on a single processor using pre-emptive Shortest Remaining Time First scheduling algorithm. If the average waiting time of the processes is 1 millisecond, then the value of Z is ____.

Q Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds (GATE - 2017) (2 Marks)

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 5 | | | |
| P ₂ | 1 | 3 | | | |
| P ₃ | 2 | 3 | | | |
| P ₄ | 4 | 1 | | | |

What is the average turnaround time for these processes with the pre-emptive shortest remaining processing time first (SRPT) algorithm?

- (A) 5.50 (B) 5.75 (C) 6.00 (D) 6.25

Q Consider the following CPU processes with arrival times (in milliseconds) and length of CPU bursts (in milliseconds) as given below: (GATE - 2017) (1 Marks)

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 7 | | | |
| P ₂ | 3 | 3 | | | |
| P ₃ | 5 | 5 | | | |
| P ₄ | 6 | 2 | | | |

If the pre-emptive shortest remaining time first scheduling algorithm is used to schedule the processes, then the average waiting time across all processes is _____ milliseconds.

Q Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is pre-emptive shortest remaining-time first.

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 10 | | | |
| P ₂ | 3 | 6 | | | |
| P ₃ | 7 | 1 | | | |
| P ₄ | 8 | 3 | | | |

The average turnaround time of these processes is _____ milliseconds. (GATE - 2016) (2 Marks)

Q Consider the following set of processes that need to be scheduled on a single CPU. All the times are given in milliseconds? **(GATE-2014) (2 Marks)**

| Process Name | Arrival Time | Execution Time | CT | TAT | WT |
|--------------|--------------|----------------|----|-----|----|
| A | 0 | 6 | | | |
| B | 3 | 2 | | | |
| C | 5 | 4 | | | |
| D | 7 | 6 | | | |
| E | 10 | 3 | | | |

Using the *shortest remaining time first* scheduling algorithm, the average process turnaround time (in msec) is _____.

Q An operating system uses shortest remaining time first scheduling algorithm for pre-emptive scheduling of processes. Consider the following set of processes with their arrival times and CPU burst times (in milliseconds) (GATE-2014) (2 Marks)

| Process | Arrival Time | Burst Time | CT | TAT | WT |
|----------------|--------------|------------|----|-----|----|
| P ₁ | 0 | 12 | | | |
| P ₂ | 2 | 4 | | | |
| P ₃ | 3 | 6 | | | |
| P ₄ | 8 | 5 | | | |

The average waiting time (in milliseconds) of the processes is _____.

Q Consider the following table of arrival time and burst time for three processes P_0 , P_1 and P_2 .

| Process | Arrival Time | Burst Time | CT | TAT | WT |
|---------|--------------|------------|----|-----|----|
| P_0 | 0 | 9 | | | |
| P_1 | 1 | 4 | | | |
| P_2 | 2 | 9 | | | |

The pre-emptive shortest job first scheduling algorithm is used. Scheduling is carried out only at arrival or completion of processes. What is the average waiting time for the three processes?

(GATE-2011) (2 Marks)

- (A) 5.0 ms (B) 4.33 ms (C) 6.33 (D) 7.33

Q An operating system uses Shortest Remaining Time first (SRT) process scheduling algorithm. Consider the arrival times and execution times for the following processes:

| Process | Arrival Time | CPU Time | CT | TAT | WT |
|----------------|--------------|----------|----|-----|----|
| P ₁ | 0 | 20 | | | |
| P ₂ | 15 | 25 | | | |
| P ₃ | 30 | 10 | | | |
| P ₄ | 45 | 15 | | | |

What is the total waiting time for process P₂? (GATE - 2007) (2 Marks)

- (A) 5 (B) 15 (C) 40 (D) 55

Q Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements a shortest remaining time first scheduling algorithm? Do not count the context switches at time zero and at the end? **(GATE-2006) (1 Marks)**

(A) 1

(B) 2

(C) 3

(D) 4

| Process | Arrival Time | CPU Time |
|---------|--------------|----------|
| P_1 | 0 | 10 |
| P_2 | 2 | 20 |
| P_3 | 6 | 30 |

Q. Consider a single processor system with four processes A, B, C, and D, represented as given below, where for each process the first value is its arrival time, and the second value is its CPU burst time. **(Gate 2024 CS)**

A (0, 10), B (2, 6), C (4, 3), and D (6, 7).

Which one of the following options gives the average waiting times when preemptive Shortest Remaining Time First (SRTF) and Non-Preemptive Shortest Job First (NP-SJF) CPU scheduling algorithms are applied to the processes?

- (a) SRTF = 6, NP-SJF = 7
- (b) SRTF = 6, NP-SJF = 7.5
- (c) SRTF = 7, NP-SJF = 7.5
- (d) SRTF = 7, NP-SJF = 8.5

Q. Processes P_1, P_2, P_3, P_4 arrive in that order at times 0, 1, 2, and 8 milliseconds respectively, and have execution times of 10, 13, 6, and 9 milliseconds respectively. Shortest Remaining Time First (SRTF) algorithm is used as the CPU scheduling policy. Ignore context switching times.
Which ONE of the following correctly gives the average turnaround time of the four processes in milliseconds? **(Gate 2025)**

- A) 22
- B) 15
- C) 37
- D) 19

Priority scheduling



Bangalore में आपका स्वागत है

Priority scheduling

- Here a priority is associated with each process. At any instance of time out of all available process, CPU is allocated to the process which possess highest priority (may be higher or lower number).
- Tie is broken using FCFS order. No importance to senior or burst time. It supports both non-pre-emptive and pre-emptive versions.
- In Priority (non-pre-emptive) once a decision is made and among the available process, the process with the highest priority is scheduled on the CPU, it cannot be pre-empted even if a new process with higher priority more than the priority of the running process enter in the system.

| P. No | AT | BT | Priority | CT | TAT = CT - AT | WT = TAT - BT |
|----------------|----|----|----------|----|---------------|---------------|
| P ₀ | 1 | 4 | 4 | | | |
| P ₁ | 2 | 2 | 5 | | | |
| P ₂ | 2 | 3 | 7 | | | |
| P ₃ | 3 | 5 | 8(H) | | | |
| P ₄ | 3 | 1 | 5 | | | |
| P ₅ | 4 | 2 | 6 | | | |
| Average | | | | | | |

Q. A computer has two processors, M_1 and M_2 . Four processes P_1, P_2, P_3, P_4 with CPU bursts of 20, 16, 25, and 10 milliseconds, respectively, arrive at the same time and these are the only processes in the system. The scheduler uses non-preemptive priority scheduling, with priorities decided as follows:

- M_1 uses priority of execution for the processes as, $P_1 > P_3 > P_2 > P_4$, i.e., P_1 and P_4 have highest and lowest priorities, respectively.
- M_2 uses priority of execution for the processes as, $P_2 > P_3 > P_4 > P_1$, i.e., P_2 and P_1 have highest and lowest priorities, respectively.

A process P_i is scheduled to a processor M_k , if the processor is free and no other process P_j is waiting with higher priority. At any given point of time, a process can be allocated to any one of the free processors without violating the execution priority rules. Ignore the context switch time. What will be the average waiting time of the processes in milliseconds? **(Gate 2025)**

- A) 9.00
- B) 8.75
- C) 6.50
- D) 7.50

- In Priority (pre-emptive) once a decision is made and among the available process, the process with the highest priority is scheduled on the CPU.
- if it a new process with priority more than the priority of the running process enter in the system, then we do a context switch and the processor is provided to the new process with higher priority.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. There is no general agreement on whether 0 is the highest or lowest priority, it can vary from systems to systems.

| P. No | AT | BT | Priority | CT | TAT = CT - AT | WT = TAT - BT |
|----------------|----|-----|----------|----|---------------|---------------|
| P ₀ | 0 | 50 | 4 | | | |
| P ₁ | 20 | 20 | 1(h) | | | |
| P ₂ | 40 | 100 | 3 | | | |
| P ₃ | 60 | 40 | 2 | | | |
| Average | | | | | | |

- **Advantage**
 - Gives a facility specially to system process.
 - Allow us to run important process even if it is a user process.
- **Disadvantage**
 - Here process with the smaller priority may starve for the CPU
 - No idea of response time or waiting time.
- Note: - Specially use to support system process or important user process
- **Ageing**: - a technique of gradually increasing the priority of processes that wait in the system for long time. E.g. priority will increase after every 10 mins

Q Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

| Process | Arrival Time | Burst Time | Priority | CT | TAT | WT |
|----------------|--------------|------------|----------|----|-----|----|
| P ₁ | 0 | 11 | 2 | | | |
| P ₂ | 5 | 28 | 0 | | | |
| P ₃ | 12 | 2 | 3 | | | |
| P ₄ | 2 | 10 | 1 | | | |
| P ₅ | 9 | 16 | 4 | | | |

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is ____. (GATE-2017) (2 Marks)

Q Consider a uniprocessor system executing three tasks T_1 , T_2 and T_3 , each of which is composed of an infinite sequence of jobs (or instances) which arrive periodically at intervals of 3, 7 and 20 milliseconds, respectively. The priority of each task is the inverse of its period and the available tasks are scheduled in order of priority, with the highest priority task scheduled first. Each instance of T_1 , T_2 and T_3 requires an execution time of 1, 2 and 4 milliseconds, respectively. Given that all tasks initially arrive at the beginning of the 1st milliseconds and task preemptions are allowed, the first instance of T_3 completes its execution at the end of _____ milliseconds. (GATE-2015) (2 Marks)

[Asked in Amcat 2021]

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Q We wish to schedule three processes P_1 , P_2 and P_3 on a uniprocessor system. The priorities, CPU time requirements and arrival times of the processes are as shown below.

We have a choice of pre-emptive or non-pre-emptive scheduling. In pre-emptive scheduling, a late-arriving higher priority process can pre-empt a currently running process with lower priority. In non-pre-emptive scheduling, a late-arriving higher priority process must wait for the currently executing process to complete before it can be scheduled on the processor.

What are the turnaround times (time from arrival till completion) of P_2 using pre-emptive and non-pre-emptive scheduling respectively. **(GATE-2005) (2 Marks)**

- (A) 30 sec, 30 sec (B) 30 sec, 10 sec
(C) 42 sec, 42 sec (D) 30 sec, 42 sec

| Process | Priority | CPU time | Arrival time |
|---------|-------------|----------|--------------|
| P_1 | 10(highest) | 20 sec | 00:00:05 |
| P_2 | 9 | 10 sec | 00:00:03 |
| P_3 | 8 (lowest) | 15 sec | 00:00:00 |

Round robin

- This algo is designed for time sharing systems, where it is not necessary to complete one process and then start another, but to be responsive and divide time of CPU among the processes in the ready state. Here ready queue is treated as a circular queue (FIFO).
- It is similar to FCFS scheduling, but pre-emption is added to enable the system to switch between processes. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval equivalent to 1 Time quantum (where value of TQ can be anything).



- We fix a time quantum, up to which a process can hold the CPU in one go, with in which either a process terminates or process must release the CPU and enter the ready queue and wait for the next chance. The process may have a CPU burst of less than given time quantum. In this case, the process itself will release the CPU voluntarily.
- CPU Scheduler will select the next process for execution. OR The CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units.
- Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum.

| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 0 | 4 | | | |
| P ₁ | 1 | 5 | | | |
| P ₂ | 2 | 2 | | | |
| P ₃ | 3 | 1 | | | |
| P ₄ | 4 | 6 | | | |
| P ₅ | 6 | 3 | | | |
| Average | | | | | |

| P. No | Arrival Time (AT) | Burst Time (BT) | Completion Time (CT) | Turn Around Time (TAT) = CT - AT | Waiting Time (WT) = TAT - BT |
|----------------|----------------------|--------------------|-------------------------|-------------------------------------|---------------------------------|
| P ₀ | 5 | 5 | | | |
| P ₁ | 4 | 6 | | | |
| P ₂ | 3 | 7 | | | |
| P ₃ | 1 | 9 | | | |
| P ₄ | 2 | 2 | | | |
| P ₅ | 6 | 3 | | | |
| Average | | | | | |

- If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at $1/n$ the speed of the real processor. We also need also to consider the effect of context switching on the performance of RR scheduling.

- **Advantage**

- Perform best in terms of average response time
- Works well in case of time-sharing systems, client server architecture and interactive system
- kind of SJF implementation

- **Disadvantage**

- Longer process may starve
- Performance depends heavily on time quantum - If value of the time quantum is very less, then it will give lesser average response time (good but total no of context switches will be more, so CPU utilization will be less), If time quantum is very large then average response time will be more bad, but no of context switches will be less, so CPU utilization will be good.
- No idea of priority

Q Consider four processes P, Q, R, and S scheduled on a CPU as per round robin algorithm with a time quantum of 4 units. The processes arrive in the order P, Q, R, S, all at time $t = 0$. There is exactly one context switch from S to Q, exactly one context switch from R to Q, and exactly two context switches from Q to R. There is no context switch from S to P. Switching to a ready process after the termination of another process is also considered a context switch. Which one of the following is NOT possible as CPU burst time (in time units) of these processes? **(GATE 2022) (2 MARKS)**

- (A) P = 4, Q = 10, R = 6, S = 2
- (B) P = 2, Q = 9, R = 5, S = 1
- (C) P = 4, Q = 12, R = 5, S = 4
- (D) P = 3, Q = 7, R = 7, S = 3

Q Which of the following statement(s) is/are correct in the context of CPU scheduling?(GATE 2021) (1 MARKS)

- (A) Turnaround time includes waiting time**
- (B) The goal is to only maximize CPU utilization and minimize throughput**
- (C) Round-robin policy can be used even when the CPU time required by each of the processes is not known apriori**
- (D) Implementing preemptive scheduling needs hardware support**

Q A scheduling algorithm assigns priority proportional to the waiting time of a process. Every process starts with priority zero (the lowest priority). The scheduler re-evaluates the process priorities every T time units and decides the next process to schedule. Which one of the following is TRUE if the processes have no I/O operations and all arrive at time zero? **(GATE-2013) (1 Marks)**

- (A)** This algorithm is equivalent to the first-come-first-serve algorithm
- (B)** This algorithm is equivalent to the round-robin algorithm.
- (C)** This algorithm is equivalent to the shortest-job-first algorithm..
- (D)** This algorithm is equivalent to the shortest-remaining-time-first algorithm

Q Consider n processes sharing the CPU in a round-robin fashion. Assuming that each process switch takes s seconds, what must be the quantum size q such that the overhead resulting from process switching is minimized but at the same time each process is guaranteed to get its turn at the CPU at least every t seconds? **(GATE-1998) (1 Marks) (NET-Dec-2012)**

a) $q \leq (t-ns)/(n-1)$

b) $q \geq (t-ns)/(n-1)$

c) $q \leq (t-ns)/(n+1)$

d) $q \geq (t-ns)/(n+1)$

Q If the time-slice used in the round-robin scheduling policy is more than the maximum time required to execute any process, then the policy will (**GATE-2008**)
(1 Marks)

- (A) degenerate to shortest job first
- (B) degenerate to priority scheduling
- (C) degenerate to first come first serve
- (D) none of the above

Q Four jobs to be executed on a single processor system arrive at time 0^+ in the order A, B, C, D. their burst CPU time requirements are 4, 1, 8, 1 time units respectively. The completion time of A under round robin scheduling with time slice of one-time unit is. **(GATE-1996) (2 Marks)** **(ISRO-2008)**

(a) 10



(b) 4

(c) 8

(d) 9

**Q Which scheduling policy is most suitable for a time-shared operating system?
(GATE-1995) (1 Marks)**

(a) Shortest Job First

(b) Round Robin

(c) First Come First Serve

(d) Elevator

Q Assume that the following jobs are to be executed on a single processor system

The jobs are assumed to have arrived at time 0^+ and in the order p, q, r, s, t. calculate the departure time (completion time) for job p if scheduling is round robin with time slice 1.

(GATE-1993) (2 Marks)

(a) 4

(b) 10

(c) 11

(d) 12

| Job Id | CPU Burst Time |
|--------|----------------|
| P | 4 |
| Q | 1 |
| R | 8 |
| S | 1 |
| T | 2 |

Q Three processes A, B and C each execute a loop of 100 iterations. In each iteration of the loop, a process performs a single computation that requires t_c CPU milliseconds and then initiates a single I/O operation that lasts for $t_{i/o}$ milliseconds. It is assumed that the computer where the processes execute has sufficient number of I/O devices and the OS of the computer assigns different I/O devices to each process. Also, the scheduling overhead of the OS is negligible. The processes have the following characteristics:

The processes A, B, and C are started at times 0, 5 and 10 milliseconds respectively, in a pure time-sharing system (round robin scheduling) that uses a time slice of 50 milliseconds. The time in milliseconds at which process C would complete its first I/O operation is _____. **(GATE-2014) (2 Mark)**

| Process Id | T_c | $T_{i/o}$ |
|------------|-------|-----------|
| A | 100 | 500 |
| B | 350 | 500 |
| C | 200 | 500 |

Q Consider the 3 processes, P_1 , P_2 and P_3 shown in the table. (GATE-2012) (2 Marks)

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2-time units) are

- (A) FCFS: P_1, P_2, P_3 RR: P_1, P_2, P_3
- (B) FCFS: P_1, P_3, P_2 RR: P_1, P_3, P_2
- (C) FCFS: P_1, P_2, P_3 RR: P_1, P_3, P_2
- (D) FCFS: P_1, P_3, P_2 RR: P_1, P_2, P_3

| Process | Arrival Time | Time Unit Required |
|---------|--------------|--------------------|
| P_1 | 0 | 5 |
| P_2 | 1 | 7 |
| P_3 | 3 | 4 |

Q Which of the following statements are true? (GATE-2010) (1 Marks)

1) Shortest remaining time first scheduling may cause starvation

2) Pre-emptive scheduling may cause starvation

3) Round robin is better than FCFS in terms of response time

(A) 1 only

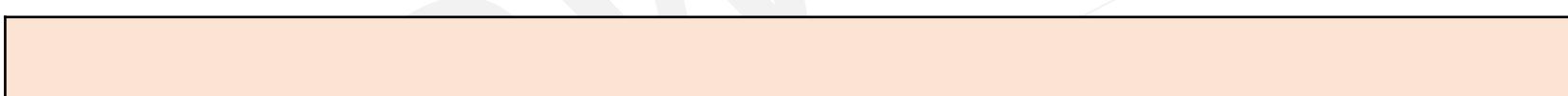
(B) 1 and 3 only

(C) 2 and 3 only

(D) 1, 2 and 3

Q Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle? (GATE-2006) (2 Marks)

- (A) 0% (B) 10.6% (C) 30.0% (D) 89.4%



| Process | AT | ET | I/O | CPU | I/O |
|----------------|----|----|-----|-----|-----|
| P ₁ | 0 | 10 | | | |
| P ₂ | 0 | 20 | | | |
| P ₃ | 0 | 30 | | | |

Q The arrival time, priority, and duration of the CPU and I/O bursts for each of three processes P_1 , P_2 and P_3 are given in the table below. Each process has a CPU burst followed by an I/O burst followed by another CPU burst. Assume that each process has its own I/O resource. **(GATE-2006) (2 Marks)**

| Process | Arrival Time | Priority | Burst duration, CPU, I/O CPU |
|---------|--------------|----------|------------------------------|
| P_1 | 0 | 2 | 1,5,3 |
| P_2 | 2 | 3(L) | 3,3,1 |
| P_3 | 3 | 1(H) | 2,3,1 |

The programmed operating system uses preemptive priority scheduling. What are the finish times of the processes P_1 , P_2 and P_3 ?

- (A) 11, 15, 9 (B) 10, 15, 9 (C) 11, 16, 10 (D) 12, 17, 11**

Q A uniprocessor computer system only has two processes, both of which alternate 10 ms CPU bursts with 90 ms I/O bursts. Both the processes were created at nearly the same time. The I/O of both processes can proceed in parallel. Which of the following scheduling strategies will result in the least CPU utilization (over a long period of time) for this system? **(GATE-2003) (2 Marks)**

- (A)** First come first served scheduling
- (B)** Shortest remaining time first scheduling
- (C)** Static priority scheduling with different priorities for the two processes
- (D)** Round robin scheduling with a time quantum of 5 ms

Longest Job First

- Process having longest Burst Time will get scheduled first.
- It can be both pre-emptive and non-pre-emptive in nature.
- The pre-emptive version is referred to as Longest Remaining Time First (LRTF) Scheduling Algorithm.

| P. No | AT | BT |
|----------------|----|----|
| P ₁ | 0 | 3 |
| P ₂ | 1 | 2 |
| P ₃ | 2 | 4 |
| P ₄ | 3 | 5 |
| P ₅ | 4 | 6 |

Longest Remaining Time First (LRTF)

- It is pre-emptive in nature.
- Longest Burst time process will get scheduled first.

| P. No | AT | BT |
|----------------|----|----|
| P ₁ | 0 | 2 |
| P ₂ | 0 | 4 |
| P ₃ | 0 | 8 |

Q Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (LRTF) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. The average turnaround time is? **(GATE-2006) (2 Marks)**

(A) 13 units

| P. No | AT | BT | CT | TAT |
|-------|----|----|----|-----|
| P_0 | 0 | 2 | | |
| P_1 | 0 | 4 | | |
| P_2 | 0 | 8 | | |

(B) 14 units

(C) 15 units



(D) 16 units

Highest response ratio next (HRRN)

- Scheduling is a non-pre-emptive discipline, similar to shortest job next (SJN), in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting.
- Jobs gain higher priority the longer they wait, which prevents indefinite waiting or in other words what we say starvation. In fact, the jobs that have spent a long time waiting compete against those estimated to have short run times.
- Response Ratio = $(W + S)/S$
- Here, **W** is the waiting time of the process so far and **S** is the Burst time of the process.
- So, the conclusion is it gives priority to those processes which have less burst time (or execution time) but also takes care of the waiting time of longer processes, thus preventing starvation.

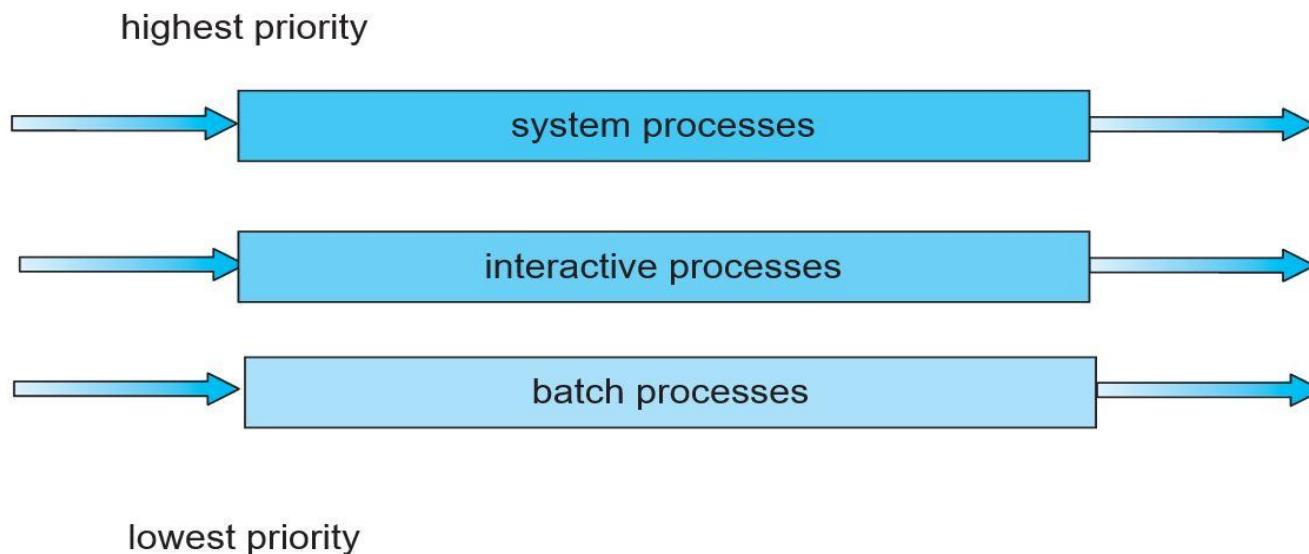
Q Consider a set of n tasks with known runtimes r_1, r_2, \dots, r_n to be run on a uniprocessor machine. Which of the following processor scheduling algorithms will result in the maximum throughput? **(GATE-2001) (1 Marks)**

- (A) Round-Robin
- (B) Shortest-Job-First
- (C) Highest-Response-Ratio-Next
- (D) First-Come-First-Served

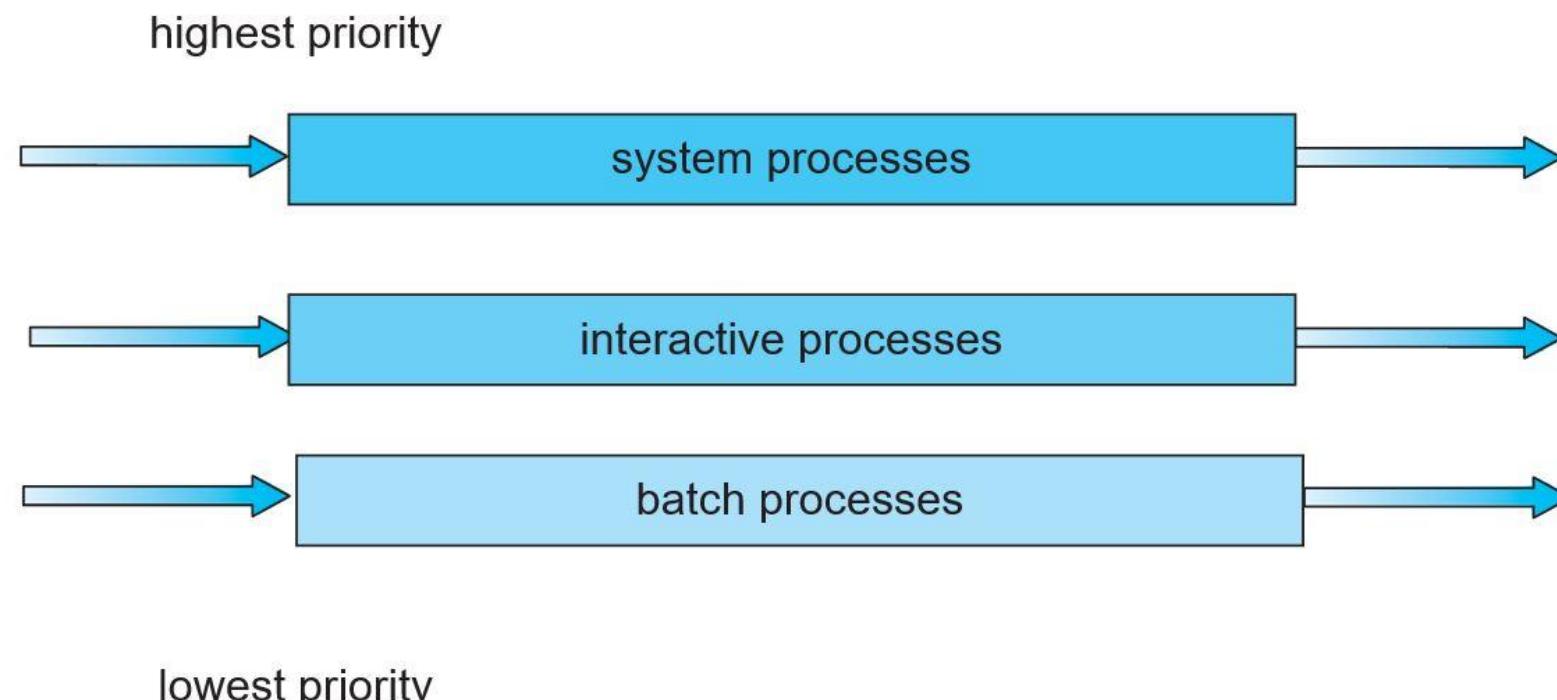
Q highest response ratio next scheduling policy favours ----- jobs, but it also limits the waiting time of ----- jobs. **(GATE-1990) (1 Marks)**

Multi Level-Queue Scheduling

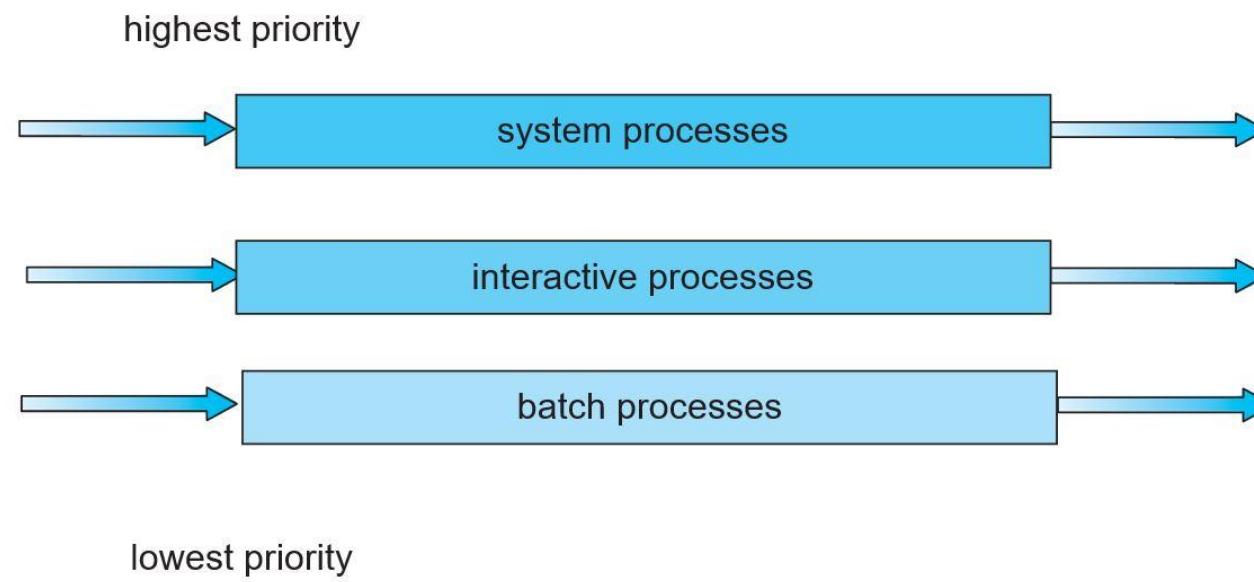
- Another class of scheduling algorithm has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs.
- A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.



- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.



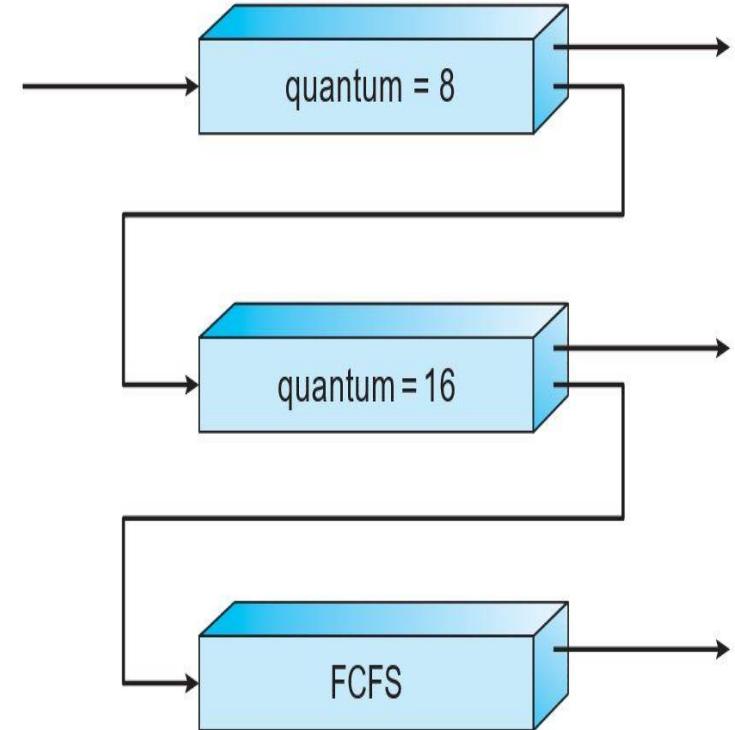
- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground– background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.



Multi-level Feedback Queue Scheduling

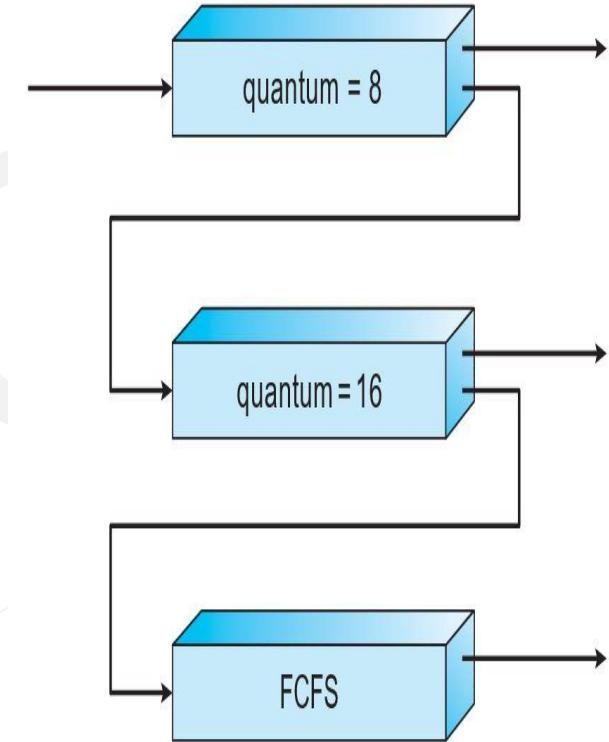
- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.
- The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.



In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service
- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm,
- since defining the best scheduler requires some means by which to select values for all the parameters.



Q Which of the following scheduling algorithms is non-preemptive? (GATE-2002) (1 Marks)

- (A) Round Robin
- (B) First-In First-Out
- (C) Multilevel Queue Scheduling
- (D) Multilevel Queue Scheduling with Feedback

Process Synchronization

- As we understand in a multiprogramming environment a good number of processes compete for limited number of resources.
- Concurrent access to shared data at some time may result in data inconsistency for e.g.

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

Race Condition

- The condition in which the output of a process depends on the execution sequence of process. i.e. if we change the order of execution of different process with respect to other process the output may change.
- That is why we need some kind of synchronization to eliminate the possibility of data inconsistency.

Q The following two functions P_1 and P_2 that share a variable B with an initial value of 2 execute concurrently.

| $P_1()$ | $P_2()$ |
|--------------|--------------|
| { | { |
| $C = B - 1;$ | $D = 2 * B;$ |
| $B = 2 * C;$ | $B = D - 1;$ |
| } | } |

The number of distinct values that B can possibly take after the execution is **(GATE-2015) (1 Mark)**

| $P_1()$ | $P_2()$ |
|-----------------------|-----------------------|
| { | { |
| $(I_{11}) C = B - 1;$ | $(I_{21}) D = 2 * B;$ |
| $(I_{12}) B = 2 * C;$ | $(I_{22}) B = D - 1;$ |
| } | } |

| Case ₁ | Case ₂ | Case ₃ | Case ₄ | Case ₅ | Case ₆ |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| $I_{11} C=1$ | $I_{21} D=4$ | $I_{11} C=1$ | $I_{21} D=4$ | $I_{11} C=1$ | $I_{21} D=4$ |
| $I_{12} B=2$ | $I_{22} B=3$ | $I_{21} D=4$ | $I_{11} C=1$ | $I_{21} D=4$ | $I_{11} C=1$ |
| $I_{21} D=4$ | $I_{11} C=2$ | $I_{22} B=3$ | $I_{12} B=2$ | $I_{12} B=2$ | $I_{22} B=3$ |
| $I_{22} B=3$ | $I_{12} B=4$ | $I_{12} B=2$ | $I_{22} B=3$ | $I_{22} B=3$ | $I_{12} B=2$ |

Q Consider three concurrent processes P_1 and P_2 and P_3 as shown below, which access a shared variable D that has been initialized to 100. (GATE-2019) (2 Marks)

The process are executed on a uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of $Y-X$ is _____.

| P_1 | P_2 | P_3 |
|--------------|--------------|--------------|
| . | . | . |
| . | . | . |
| $D = D + 20$ | $D = D - 50$ | $D = D + 10$ |
| . | . | . |
| . | . | . |

Q When the result of a computation depends on the order of the processes execution, there is said to be **(GATE-1998) (1 Marks)**

a) cycle stealing

b) race condition

c) a time lock

d) a deadlock

Q. Consider the following two threads T1 and T2 that update two shared variables a and b .Assume that initially $a=b=1$. Though context switching between threads can happen at any time, each statement of T1 or T2 is executed automatically without interruption. (Gate 2024,CS) (2 Marks) (MCQ)

| T1 | T2 |
|----------|----------|
| $a=a+1;$ | $b=2*b;$ |
| $b=b+1;$ | $a=2*a;$ |

Which one of the following options lists all the possible combinations of values of a and b after both T1 and T2 finish execution?

- (a) ($a = 4, b = 4$); ($a = 3, b = 3$); ($a = 4, b = 3$)
- (b) ($a = 3, b = 4$); ($a = 4, b = 3$); ($a = 3, b = 3$)
- (c) ($a = 4, b = 4$); ($a = 4, b = 3$); ($a = 3, b = 4$)
- (d) ($a = 2, b = 2$); ($a = 2, b = 3$); ($a = 3, b = 4$)

General Structure of a process

- **Initial Section:** Where process is accessing private resources.
- **Entry Section:** Entry Section is that part of code where, each process request for permission to enter its critical section.
- **Critical Section:** Where process is access shared resources.
- **Exit Section:** It is the section where a process will exit from its critical section.
- **Remainder Section:** Remaining Code.

```
P()
{
    While(T)
    {
        Initial Section
        Entry Section
        Critical Section
        Exit Section
        Remainder Section
    }
}
```

Criterion to Solve Critical Section Problem

- **Mutual Exclusion:** No two processes should be present inside the critical section at the same time, i.e. only one process is allowed in the critical section at an instant of time.
- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next(means other process will participate which actually wish to enter). there should be no deadlock.
- **Bounded Waiting:** There exists a bound or a limit on the number of times a process is allowed to enter its critical section and no process should wait indefinitely to enter the CS.

Some Points to Remember

- Mutual Exclusion and Progress are mandatory requirements that needs to be followed in order to write a valid solution for critical section problem.
- Bounded waiting is optional criteria, if not satisfied then it may lead to starvation.

Solutions to Critical Section Problem

We generally have the following solutions to a Critical Section Problems:

1. Two Process Solution
 1. Using Boolean variable turn
 2. Using Boolean array flag
 3. Peterson's Solution
2. Operating System Solution
 1. Counting Semaphore
 2. Binary Semaphore
3. Hardware Solution
 1. Test and Set Lock
 2. Disable interrupt

Two Process Solution

- In general it will be difficult to write a valid solution in the first go to solve critical section problem among multiple processes, so it will be better to first attempt two process solution and then generalize it to N-Process solution.
- There are 3 Different idea to achieve valid solution, in which some are invalid while some are valid.
- **1- Using Boolean variable turn**
- **2- Using Boolean array flag**
- **3- Peterson's Solution**

- Here we will use a Boolean variable turn, which is initialize randomly(0/1).

| P ₀ | P ₁ |
|--|--|
| while (1) { while (turn! = 0); CriticalSection turn = 1; Remainder section } | while (1) { while (turn! = 1); CriticalSection turn = 0; Remainder Section } |

- The solution follows Mutual Exclusion as the two processes cannot enter the CS at the same time.
- The solution does not follow the Progress, as it is suffering from the strict alternation. Because we never asked the process whether it wants to enter the CS or not?

- Here we will use a Boolean array flag with two cells, where each cell is initialized to F

| P_0 | P_1 |
|---|---|
| <pre>while (1) { flag [0] = T; while (flag [1]); Critical Section flag [0] = F; Remainder section }</pre> | <pre>while (1) { flag [1] = T; while (flag [0]); Critical Section flag [1] = F; Remainder Section }</pre> |

- This solution follows the Mutual Exclusion Criteria.
- But in order to achieve the progress the system ended up being in a deadlock state.

- Peterson's solution is a classic Software-based solution to the critical-section problem for two processes. This solution ensures **Mutual Exclusion, Progress and Bounded Wait**.

| P_0 | P_1 |
|---|---|
| <pre>while (1) { flag [0] = T; turn = 1; while (turn == 1 && flag [1] == T); Critical Section flag [0] = F; Remainder section }</pre> | <pre>while (1) { flag [1] = T; turn = 0; while (turn == 0 && flag [0] == T); Critical Section flag [1] = F; Remainder Section }</pre> |

Q Consider the methods used by processes P_1 and P_2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S_1 and S_2 are randomly assigned. **(GATE-2010) (1 Marks) (NET-JUNE-2012)**

| $P_1()$ | $P_2()$ |
|-----------------------------|-----------------------------|
| $\text{While}(S_1 == S_2);$ | $\text{While}(S_1 != S_2);$ |
| Critical section | Critical section |
| $S_1 = S_2;$ | $S_1 = \text{not}(S_2);$ |

Which one of the following statements describes the properties achieved?

- (A) Mutual exclusion but not progress
- (B) Progress but not mutual exclusion
- (C) Neither mutual exclusion nor progress
- (D) Both mutual exclusion and progress

Q Consider the following two-process synchronization solution. (GATE-2016) (2 Marks)

The shared variable turn is initialized to zero.

Which one of the following is TRUE?

(a) This is a correct two-process synchronization solution.

(b) This solution violates mutual exclusion requirement

(c) This solution violates progress requirement.

(d) This solution violates bounded wait requirement

| Process 0 | Process 1 |
|---|---|
| Entry: loop while ($\text{turn} == 1$); | Entry: loop while ($\text{turn} == 0$); |
| (critical section) | (critical section) |
| Exit: $\text{turn} = 1$; | Exit: $\text{turn} = 0$; |

Q Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes. Here, varP and varQ are shared variables and both are initialized to false. Which one of the following statements is true? (GATE-2015) (1 Mark)

- (A) The proposed solution prevents deadlock but fails to guarantee mutual exclusion
- (B) The proposed solution guarantees mutual exclusion but fails to prevent deadlock
- (C) The proposed solution guarantees mutual exclusion and prevents deadlock
- (D) The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion

| Process X | Process Y |
|-------------------|-------------------|
| While(t) | While(t) |
| { | { |
| varP = T; | varQ = T; |
| While (varQ == T) | While (varP == T) |
| { | { |
| Critical section | Critical section |
| varP = F; | varQ = F; |
| } | } |
| } | } |

Q Two processes, P_1 and P_2 , need to access a critical section of code. Consider the following synchronization construct used by the processes: Here, $wants_1$ and $wants_2$ are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct? **(GATE-2007) (2 Mark)**

- (A)** It does not ensure mutual exclusion.
- (B)** It does not ensure bounded waiting.
- (C)** It requires that processes enter the critical section in strict alternation.
- (D)** It does not prevent deadlocks, but ensures mutual exclusion.

| $P_1()$ | $P_2()$ |
|---------------------------|---------------------------|
| While(t) | While(t) |
| { | { |
| $wants_1 = T$ | $wants_2 = T$ |
| While ($wants_2 == T$); | While ($wants_1 == T$); |
| Critical section | Critical section |
| $wants_1 = F$ | $wants_2 = F$ |
| Remainder section | Remainder section |
| } | } |

Q Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j. The program executed by process is shown below. (GATE-2001)
(2 Mark)

For the program to guarantee mutual exclusion, the predicate P in the while loop should be.

- (A) flag[j] = true and turn = i**
- (B) flag[j] = true and turn = j**
- (C) flag[i] = true and turn = j**
- (D) flag[i] = true and turn = i**

Repeat

flag[i] = T;

turn = j;

while(P) do no-op;

Enter critical section, perform actions, then critical section

flag[i] = f;

Perform other non-critical section actions

Until false;

Operation System Solution

- Semaphores are synchronization tools using which we will attempt n-process solution.
- A semaphore S is a simple integer variable that, but apart from initialization it can be accessed only through two standard atomic operations: wait(S) and signal(S).
- The wait(S) operation was originally termed as P(S) and signal(S) was originally called V(S).
- The definition of wait () is as follows:

| |
|----------------|
| Wait(S) |
| { |
| while(s<=0); |
| S--; |
| } |

| |
|------------------|
| Signal(S) |
| { |
| S++; |
| } |

- Peterson's Solution was confined to just two processes, and since in a general system can have n processes, Semaphores provides n-processes solution.
- While solving Critical Section Problem only we initialize semaphore S = 1.
- Semaphores are going to ensure Mutual Exclusion and Progress but does not ensures bounded waiting.

| |
|------------------------|
| P_i() |
| { |
| While(T) |
| { |
| Initial Section |
| wait(s) |
| Critical Section |
| signal(s) |
| Remainder Section |
| } |
| } |

| |
|----------------|
| Wait(S) |
| { |
| while(s<=0); |
| s--; |
| } |

| |
|------------------|
| Signal(S) |
| { |
| s++; |
| } |

Q A critical section is a program segment? (GATE-1996) (1 Mark)

- (a) which should run in a certain specified amount of time**
- (b) which avoids deadlocks**
- (c) where shared resources are accessed**
- (d) which must be enclosed by a pair of semaphore operations, P and V**

Q A critical region is (GATE-1987) (1 Marks)

- a) One which is enclosed by a pair of P and V operations on semaphores.
- b) A program segment that has not been proved bug-free.
- c) A program segment that often causes unexpected system crashes.
- d) A program segment where shared resources are accessed.

Q Using Semaphores ensure that the order of execution of there concurrent process p_1 , p_2 and p_3 must be $p_2 \rightarrow p_3 \rightarrow p_1$?

| $P_1()$ | $P_2()$ | $P_3()$ |
|---------|---------|---------|
| | | |
| code | code | code |
| | | |

Q Consider the following threads, T_1 , T_2 , and T_3 executing on a single processor, synchronized using three binary semaphore variables, S_1 , S_2 , and S_3 , operated upon using standard wait() and signal(). The threads can be context switched in any order and at any time. Which initialization of the semaphores would print the sequence BCABCABC...? (GATE 2022) (1 MARKS)

(A) $S_1 = 1; S_2 = 1; S_3 = 1$

(B) $S_1 = 1; S_2 = 1; S_3 = 0$

(C) $S_1 = 1; S_2 = 0; S_3 = 0$

(D) $S_1 = 0; S_2 = 1; S_3 = 1$

| T_1 | T_2 | T_3 |
|---|---|---|
| <pre>while(true) { wait(S₃) ; print("C") ; signal(S₂) ; }</pre> | <pre>while(true) { wait(S₁) ; print("B") ; signal(S₃) ; }</pre> | <pre>while(true) { wait(S₂) ; print("A") ; signal(S₁) ; }</pre> |

Q A certain computation generates two arrays a and b such that $a[i]=f(i)$ for $0 \leq i < n$ and $b[i]=g(a[i])$ for $0 \leq i < n$. Suppose this computation is decomposed into two concurrent processes X and Y such that X computes the array a and Y computes the array b. The processes employ two binary semaphores R and S, both initialized to zero. The array a is shared by the two processes. The structures of the processes are shown below. (GATE-2013) (2 Marks)

- a)**
- ```
ExitX(R, S) {
 P(R);
 V(S);
}

EntryY(R, S) {
 P(S);
 V(R);
}
```
- b)**
- ```
ExitX(R, S) {  
    V(R);  
    V(S);  
}  
  
EntryY(R, S) {  
    P(R);  
    P(S);  
}
```
- c)**
- ```
ExitX(R, S) {
 P(S);
 V(R);
}

EntryY(R, S) {
 V(S);
 P(R);
}
```
- d)**
- ```
ExitX(R, S) {  
    V(R);  
    P(S);  
}  
  
EntryY(R, S) {  
    V(S);  
    P(R);  
}
```

| Process X: | Process Y: |
|---|---|
| private i; for ($i=0; i < n; i++$) { $a[i] = f(i);$ ExitX(R, S); } } | private i; for ($i=0; i < n; i++$) { EntryY(R, S); $b[i]=g(a[i]);$ } } |

Q Two concurrent processes P_1 and P_2 use four shared resources R_1 , R_2 , R_3 and R_4 , as shown below.

Both processes are started at the same time, and each resource can be accessed by only one process at a time. The following scheduling constraints exist between the access of resources by the processes:

- i) P_2 must complete use of R_1 before P_1 gets access to R_1 .
- ii) P_1 must complete use of R_2 before P_2 gets access to R_2 .
- iii) P_2 must complete use of R_3 before P_1 gets access to R_3 .
- iv) P_1 must complete use of R_4 before P_2 gets access to R_4 .

There are no other scheduling constraints between the processes. If only binary semaphores are used to enforce the above scheduling constraints, what is the minimum number of binary semaphores needed? (GATE-2005) (2 Marks)

- (A) 1
- (B) 2
- (C) 3
- (D) 4

| P_1 | P_2 |
|--|--|
| Compute: Use R_1 ; Use R_2 ; Use R_3 ; Use R_4 ; | Compute; Use R_1 ; Use R_2 ; Use R_3 ; Use R_4 ; |

P_2 must complete use of R_1 before P_1 gets access to R_1 .
 P_1 must complete use of R_2 before P_2 gets access to R_2 .
 P_2 must complete use of R_3 before P_1 gets access to R_3 .
 P_1 must complete use of R_4 before P_2 gets access to R_4 .

| P_1 | P_2 |
|------------------------------|------------------------------|
| Compute: | Compute; |
| Use R_1; | Use R_1; |
| Use R_2; | Use R_2; |
| Use R_3; | Use R_3; |
| Use R_4; | Use R_4; |

| P_1 | P_2 |
|-------|-------|
| | R_1 |
| R_1 | |
| R_2 | |
| | R_2 |
| | R_3 |
| R_3 | |
| R_4 | |
| | R_4 |

| P_1 | P_2 |
|--------|--------|
| $P(X)$ | R_1 |
| R_1 | $V(X)$ |
| R_2 | $P(Y)$ |
| $V(Y)$ | R_2 |
| $P(X)$ | R_3 |
| R_3 | $V(X)$ |
| R_4 | $P(Y)$ |
| $V(Y)$ | R_4 |

Q Semaphores S and T. The code for the processes P and Q is shown below. Synchronization statements can be inserted only at points W, X, Y and Z.

Which of the following will always lead to an output starting with '001100110011' ? (GATE-2004) (2 Marks)

- (A) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S and T initially 1
- (B) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S initially 1, and T initially 0
- (C) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S and T initially 1
- (D) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S initially 1, and T initially 0

| Process P | Process Q |
|------------|------------|
| While(t) | While(t) |
| { | { |
| W: | Y: |
| print '0'; | print '1'; |
| print '0'; | print '1'; |
| X: | Z: |
| } | } |

Q Semaphores S and T. The code for the processes P and Q is shown below. Synchronization statements can be inserted only at points W, X, Y and Z.

Which of the following will ensure that the output string never contains a substring of the form 01^n0 or 10^n1 where n is odd?
(GATE-2004) (2 Marks)

- (A) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S and T initially 1
- (B) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S and T initially 1
- (C) P(S) at W, V(S) at X, P(S) at Y, V(S) at Z, S initially 1
- (D) V(S) at W, V(T) at X, P(S) at Y, P(T) at Z, S and T initially 1

| Process P | Process Q |
|------------|------------|
| While(t) | While(t) |
| { | { |
| W: | Y: |
| print '0'; | print '1'; |
| print '0'; | print '1'; |
| X: | Z: |
| } | } |

Q which of the following will ensure deadlock free execution, if both s_1 and s_2 both are initialized to 1?

| Process P | Process Q |
|---------------|---------------|
| While(t) | While(t) |
| { | { |
| wait(s_1) | wait(s_2) |
| wait(s_2) | wait(s_1) |
| CS | CS |
| } | } |

| Process P | Process Q |
|---------------|---------------|
| While(t) | While(t) |
| { | { |
| wait(s_1) | wait(s_1) |
| wait(s_2) | wait(s_2) |
| CS | CS |
| } | } |

Conclusion: if we want to avoid deadlock the order to apply wait operation on different semaphores among different must be same.

Q Three concurrent processes X, Y, and Z execute three different code segments that access and update certain shared variables. Process X executes the P operation (i.e., wait) on semaphores a, b and c; process Y executes the P operation on semaphores b, c and d; process Z executes the P operation on semaphores c, d, and a before entering the respective code segments. After completing the execution of its code segment, each process invokes the V operation (i.e., signal) on its three semaphores. All semaphores are binary semaphores initialized to one. Which one of the following represents a deadlock-free order of invoking the P operations by the processes? (GATE-2013) (1 Marks)

- (A) X: P(a)P(b)P(c) Y: P(b)P(c)P(d) Z: P(c)P(d)P(a)
- (B) X: P(b)P(a)P(c) Y: P(b)P(c)P(d) Z: P(a)P(c)P(d)
- (C) X: P(b)P(a)P(c) Y: P(c)P(b)P(d) Z: P(a)P(c)P(d)
- (D) X: P(a)P(b)P(c) Y: P(c)P(b)P(d) Z: P(c)P(d)P(a)

| X | Y | Z |
|----|----|----|
| | | |
| | | |
| | | |
| CS | CS | CS |

Q Consider two processes P_1 and P_2 accessing the shared variables X and Y protected by two binary semaphores S_X and S_Y respectively, both initialized to 1. P and V denote the usual semaphore operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code of P_1 and P_2 is as follows: (GATE-2004) (2 Marks)

In order to avoid deadlock, the correct operators at L_1 , L_2 , L_3 and L_4 are respectively

- (A) $P(S_Y), P(S_X); P(S_X), P(S_Y)$
- (B) $P(S_X), P(S_Y); P(S_Y), P(S_X)$
- (C) $P(S_X), P(S_X); P(S_Y), P(S_Y)$
- (D) $P(S_X), P(S_Y); P(S_X), P(S_Y)$

| P_1 | P_2 |
|-------------------------|-------------------------|
| While true do { | While true do { |
| $L_1 : \dots\dots\dots$ | $L_3 : \dots\dots\dots$ |
| $L_2 : \dots\dots\dots$ | $L_4 : \dots\dots\dots$ |
| $X = X + 1;$ | $Y = Y + 1;$ |
| $Y = Y - 1;$ | $X = Y - 1;$ |
| $V(S_X);$ | $V(S_X);$ |
| $V(S_Y);$ | $V(S_Y);$ |
| } | } |

Q A shared variable x , initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution? (GATE-2013) (2 Marks)

- (A) -2 (B) -1 (C) 1 (D) 2

| W | X | Y | Z |
|---------------|---------------|---------------|---------------|
| Wait(S) | Wait(S) | Wait(S) | Wait(S) |
| $R(x)$ | $R(x)$ | $R(x)$ | $R(x)$ |
| $X = x + 1$ | $X = x + 1$ | $X = x - 2$ | $X = x - 2$ |
| $W(x)$ | $W(x)$ | $W(x)$ | $W(x)$ |
| Signal(S) | Signal(S) | Signal(S) | Signal(S) |

Barrier is a synchronization construct where a set of processes synchronizes globally i.e. each process in the set arrives at the barrier and waits for all others to arrive and then all processes leave the barrier. Let the number of processes in the set be three and S be a binary semaphore with the usual P and V functions. Consider the following C implementation of a barrier with line numbers shown on left.

The variables process_arrived and process_left are shared among all processes and are initialized to zero. In a concurrent program all the three processes call the barrier function when they need to synchronize globally.

Q The above implementation of barrier is incorrect. Which one of the following is true?
(GATE-2006) (2 Marks)

- a)** The barrier implementation is wrong due to the use of binary semaphore S.
- b)** The barrier implementation may lead to a deadlock if two barrier invocations are used in immediate succession.
- c)** Lines 6 to 10 need not be inside a critical section.
- d)** The barrier implementation is correct if there are only two processes instead of three.

```
void barrier (void) {  
1: P(S);  
2: process_arrived++;  
3: V(S);  
4: while (process_arrived !=3);  
5: P(S);  
6: process_left++;  
7: if (process_left==3) {  
8:   process_arrived = 0;  
9:   process_left = 0;  
10: }  
11: V(S);  
}
```

Barrier is a synchronization construct where a set of processes synchronizes globally i.e. each process in the set arrives at the barrier and waits for all others to arrive and then all processes leave the barrier. Let the number of processes in the set be three and S be a binary semaphore with the usual P and V functions. Consider the following C implementation of a barrier with line numbers shown on left.

The variables process_arrived and process_left are shared among all processes and are initialized to zero. In a concurrent program all the three processes call the barrier function when they need to synchronize globally.

Q Which one of the following rectifies the problem in the implementation? **(GATE-2006) (2 Marks)**

- a) Lines 6 to 10 are simply replaced by process_arrived--.
- b) At the beginning of the barrier the first process to enter the barrier waits until process_arrived becomes zero before proceeding to execute P(S).
- c) Context switch is disabled at the beginning of the barrier and re-enabled at the end.
- d) The variable process_left is made private instead of shared.

```
void barrier (void) {  
1: P(S);  
2: process_arrived++;  
3: V(S);  
4: while (process_arrived !=3);  
5: P(S);  
6: process_left++;  
7: if (process_left==3) {  
8:   process_arrived = 0;  
9:   process_left = 0;  
10: }  
11: V(S);  
}
```

Q Given below is a program which when executed spawns two concurrent processes:

```
semaphore X := 0 ;  
/* Process now forks into concurrent processes P1 & P2 */
```

Consider the following statements about processes P₁ and P₂:

- i) It is possible for process P₁ to starve.
- ii) It is possible for process P₂ to starve.

Which of the following holds? **(GATE-2005) (2 Marks)**

- (A) Both I and II are true
- (B) I is true but II is false
- (C) II is true but I is false
- (D) Both I and II are false

| P ₁ | P ₂ |
|--|---|
| repeat forever V (X) ; Compute ; P(X) ; | repeat forever P(X) ; Compute ; V(X) ; |

Classical Problems on Synchronization

- There are number of actual industrial problem we try to solve in order to improve our understand of Semaphores and their power of solving problems.
- Here in this section we will discuss a number of problems like
 - Producer consumer problem/ Bounder Buffer Problem
 - Reader-Writer problem
 - Dining Philosopher problem

Producer-Consumer Problem

- **Problem Definition** – There are two process Producer and Consumers, producer produces information and put it into a buffer which have n cell, that is consumed by a consumer.
- Both Producer and Consumer can produce and consume only one article at a time.



Producer-Consumer Problem needs to sort out three major issues

- A producer needs to check whether the buffer is overflowed or not after producing an item before accessing the buffer.
- Similarly, a consumer needs to check for an underflow before accessing the buffer and then consume an item.
- Also, *the producer and consumer must be synchronized, so that once a producer and consumer it accessing the buffer the other must wait.*



Solution Using Semaphores

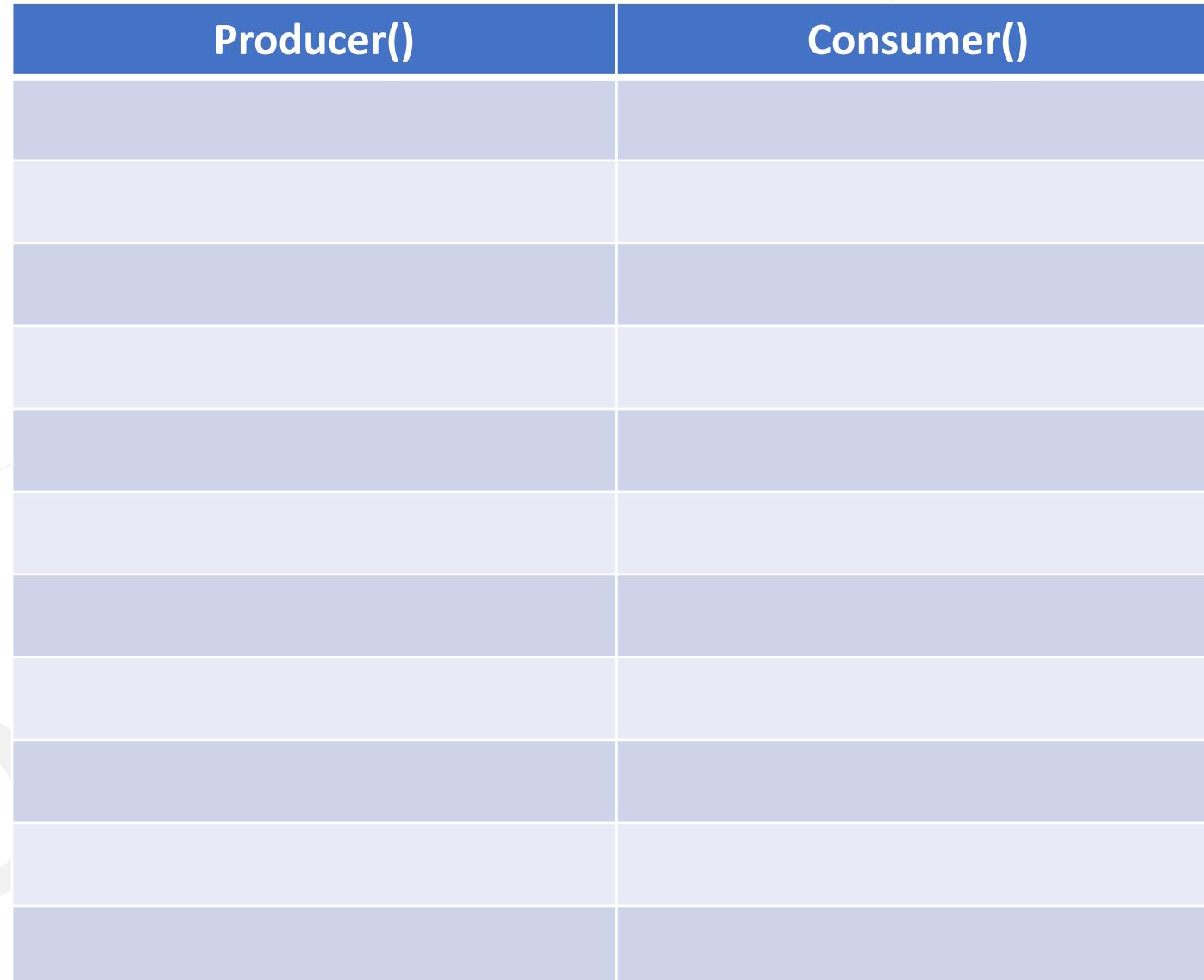
Now to solve the problem we will be using three semaphores:

Semaphore S = 1 // CS

Semaphore E = n // Count Empty cells

Semaphore F = 0 // Count Filled cells





Total three resources
are used

- semaphore E take count of empty cells and over flow
- semaphore F take count of filled cells and under flow
- Semaphore S take care of buffer

| Producer() | Consumer() |
|-----------------------|--------------------------|
| { | { |
| while(T) | while(T) |
| { | { |
| // Produce an item | wait(F)//UnderFlow |
| wait(E)//OverFlow | wait(S) |
| wait(S) | // Pick item from buffer |
| // Add item to buffer | signal(S) |
| signal(S) | signal(E) |
| signal(F) | Consume item |
| } | } |
| } | } |

Q Consider the following solution to the producer-consumer synchronization problem. The shared buffer size is N. Three semaphores *empty*, *full* and *mutex* are defined with respective initial values of 0, N and 1. Semaphore *empty* denotes the number of available slots in the buffer, for the consumer to read from. Semaphore *full* denotes the number of available slots in the buffer, for the producer to write to. The placeholder variables, denoted by P, Q, R and S, in the code below can be assigned either *empty* or *full*. The valid semaphore operations are: *wait()* and *signal()*.

(GATE-2018) (2 Marks)

Which one of the following assignments to P, Q, R and S will yield the correct solution?

- (A) P: *full*, Q: *full*, R: *empty*, S: *empty*
- (B) P: *empty*, Q: *empty*, R: *full*, S: *full*
- (C) P: *full*, Q: *empty*, R: *empty*, S: *full*
- (D) P: *empty*, Q: *full*, R: *full*, S: *empty*

| Producer: | Consumer: |
|----------------------|----------------------------|
| Do{ | Do{ |
| Wait(P); | Wait(R); |
| Wait(mutex); | Wait(mutex); |
| //Add item to buffer | //Consume item from buffer |
| Signal(mutex); | Signal(mutex); |
| Signal(Q); | Signal(S); |
| } while(1); | } while(1); |

Q Consider the procedure below for the Producer-Consumer problem which uses semaphores:

Semaphore n = 0;

Semaphore s = 1;

Which one of the following is TRUE? (GATE-2014) (2 Marks)

(A) The producer will be able to add an item to the buffer, but the consumer can never consume it.

(B) The consumer will remove no more than one item from the buffer.

(C) Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.

(D) The starting value for the semaphore n must be 1 and not 0 for deadlock-free operation.

| Void Producer () | Void Consumer () |
|-----------------------|---------------------|
| { | { |
| While(true) | While(true) |
| { | { |
| Produce (); | semWait(s); |
| SemWait(s); | semWait(n); |
| addToBuffer(); | RemovefromBuffer(); |
| semSignal(s); | semSignal(s); |
| SemSignal(n); | consume(); |
| } | } |
| } | } |

Q Process P_1 repeatedly adds one item at a time to a buffer of size n , and process P_2 repeatedly removes one item at a time from the same buffer using the programs given below. In the programs, K, L, M and N are unspecified statements.

(GATE-2004) (2 Marks)

The statements K, L, M and N are respectively

(A) P(full), V(empty), P(full), V(empty)

(B) P(full), V(empty), P(empty), V(full)

(C) P(empty), V(full), P(empty), V(full)

(D) P(empty), V(full), P(full), V(empty)

| $P_1()$ | $P_2()$ |
|----------------------------|-------------------------------|
| while(T) { | while(T) { |
| K; | M; |
| P(mutex); | P(mutex); |
| Add an item to the buffer; | Remove an item to the buffer; |
| V(mutex); | V(mutex); |
| L; | N; |
| } | } |

Reader-Writer Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers), whereas others may want to update (that is, to read and write) the database(writers). The former are referred to as readers and to the latter as writers.
- If two readers access the shared data simultaneously, no adverse effects will result. But, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

| title | release_year | length | replacement_cost |
|--|--------------|--------|------------------|
| dmlentile select title, release_year, length, replacement_cost from film | | | |
| dmlentile where length > 120 and replacement_cost > 20.99 | | | |
| dmlentile order by title desc; | | | |
| West Side | 2006 | 139 | 29.99 |
| Virgin Baby | 2006 | 179 | 29.99 |
| West Suicide | 2006 | 173 | 29.99 |
| Tracy Cider | 2006 | 143 | 29.99 |
| Song Hiding | 2006 | 161 | 29.99 |
| Stalker Lataan | 2006 | 179 | 29.99 |
| Serry Packer | 2006 | 154 | 29.99 |
| River Outlaw | 2006 | 149 | 29.99 |
| Night Crimes | 2006 | 133 | 29.99 |
| Quint Musallam | 2006 | 177 | 29.99 |
| Position Forever | 2006 | 159 | 29.99 |
| Laufting Legality | 2006 | 148 | 29.99 |
| Lawless Nation | 2006 | 181 | 29.99 |
| Dingle Begebrook | 2006 | 124 | 29.99 |
| Derrick Nolen | 2006 | 171 | 29.99 |
| Japanese Run | 2006 | 139 | 29.99 |
| Baltimore Rollied | 2006 | 163 | 29.99 |
| Floata Barlow | 2006 | 145 | 29.99 |
| Fantastic Park | 2006 | 131 | 29.99 |
| Extraordinary Computer | 2006 | 172 | 29.99 |
| Everyone Craft | 2006 | 183 | 29.99 |
| Thirty Ace | 2006 | 147 | 29.99 |
| Clyde Theory | 2006 | 139 | 29.99 |
| Clockwork Paradise | 2006 | 141 | 29.99 |
| Ballroom Rockingbird (25 rows) | 2006 | 173 | 29.99 |

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- **Points that needs to be taken care for generating a Solutions:**
 - The solution may allow more than one reader at a time, but should not allow any writer.
 - The solution should strictly not allow any reader or writer, while a writer is performing a write operation.
- **Solution using Semaphores**
 - The reader processes share the following data structures:
 - semaphore mutex = 1, wrt =1; // Two semaphores
 - int readcount = 0; // Variable

Mutex =

Wrt =

Readcount =

| Writer() | Reader() |
|-----------------|-----------------|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

- Three resources are used
- Semaphore Wrt is used for synchronization between WW, WR, RW
 - Semaphore reader is used to synchronize between RR
 - Readcount is simple int variable which keep counts of number of readers

| Writer() | Reader() |
|--------------------|----------------------|
| | Wait(mutex) |
| | Readcount++ |
| | |
| Wait(wrt) | signal(mutex) |
| CS //Write | CS //Read |
| Signal(wrt) | Wait(mutex) |
| | Readcount-- |
| | |
| | signal(mutex) |

- Three resources are used
- Semaphore Wrt is used for synchronization between WW, WR, RW
 - Semaphore reader is used to synchronize between RR
 - Readcount is simple int variable which keep counts of number of readers

| Writer() | Reader() |
|-------------|----------------------------|
| | Wait(mutex) |
| | Readcount++ |
| | If(readcount ==1) |
| | wait(wrt) // first |
| Wait(wrt) | signal(mutex) |
| CS //Write | CS //Read |
| Signal(wrt) | Wait(mutex) |
| | Readcount-- |
| | If(readcount ==0) |
| | signal(wrt) // last |
| | signal(mutex) |

Q Synchronization in the classical readers and writers problem can be achieved through use of semaphores. In the following incomplete code for readers-writers problem, two binary semaphores mutex and wrt are used to obtain synchronization **(GATE-2007) (2 Marks)**

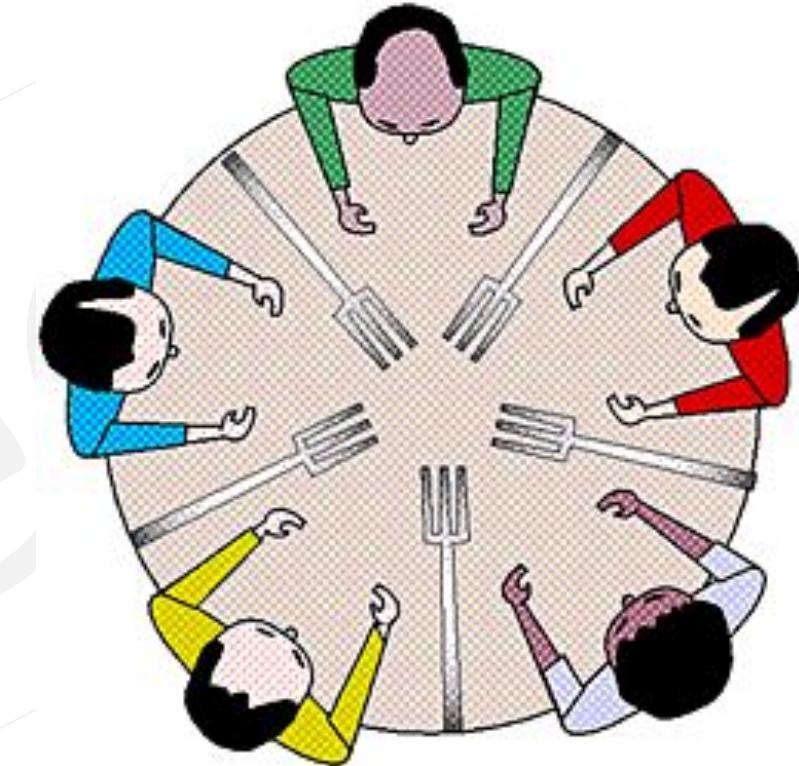
The values of S_1, S_2, S_3, S_4 , (in that order) are

- (A) signal (mutex), wait (wrt), signal (wrt), wait (mutex)
- (B) signal (wrt), signal (mutex), wait (mutex), wait (wrt)
- (C) wait (wrt), signal (mutex), wait (mutex), signal (wrt)
- (D) signal (mutex), wait (mutex), signal (mutex), wait (mutex)

| Writer() | Reader() |
|----------------------|-------------------|
| | Wait(mutex) |
| | Readcount++ |
| | If(readcount ==1) |
| | S_1 |
| Wait(wrt) | S_2 |
| Writing is performed | CS //Read |
| Signal(wrt) | S_3 |
| | Readcount-- |
| | If(readcount ==0) |
| | S_4 |
| | signal(mutex) |

Dining Philosopher Problem

- **Scenario Setup:** Five philosophers are seated around a circular table, each with a bowl of rice in the center. The table has five chairs and five single chopsticks placed between each pair of philosophers.
- **Activity Cycle:** Philosophers alternate between thinking and eating. They do not interact with each other while thinking.
- **Eating Process:**
 - A philosopher becomes hungry and attempts to pick up the two closest chopsticks — one between her and the philosopher on her left, and one between her and the philosopher on her right.
 - Each philosopher can pick up only one chopstick at a time.
 - A philosopher cannot pick up a chopstick if it is already being held by a neighbor.
 - Once a philosopher has both chopsticks, she eats without releasing the chopsticks.
- **Post-Eating:** After eating, the philosopher puts both chopsticks back on the table and resumes thinking.





Indian Chopsticks

Solution for Dining Philosophers

Void Philosopher (void)

{

 while (T)

 {

Thinking () ;

 wait(chopstick [i]);

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

Eat();

 signal(chopstick [i]);

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

 }

}



- Here we have used an array of semaphores called chopstick[]

- Solution is not valid because there is a possibility of deadlock.
- The proposed solution for deadlock problem is
 - Allow at most four philosophers to be sitting simultaneously at the table.
 - Allow six chopstick to be used simultaneously at the table.
 - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).



- One philosopher picks up her right chopstick first and then left chop stick, i.e. reverse the sequence of any philosopher.
- Odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.



Q Let $m[0] \dots m[4]$ be mutexes (binary semaphores) and $P[0] \dots P[4]$ be processes. Suppose each process $P[i]$ executes the following:

`wait (m[i]); wait(m[(i+1) mode 4]);`

`release (m[i]); release (m[(i+1)mod 4]);`

This could cause: **(GATE-2000) (1 Marks)**

- (A) Thrashing**
- (B) Deadlock**
- (C) Starvation, but not deadlock**
- (D) None of the above**

Q A solution to the Dining Philosophers Problem which avoids deadlock is:

(GATE-1996) (2 Marks)

- (A)** ensure that all philosophers pick up the left fork before the right fork
- (B)** ensure that all philosophers pick up the right fork before the left fork
- (C)** ensure that one particular philosopher picks up the left fork before the right fork, and that all other philosophers pick up the right fork before the left fork
- (D)** None of the above

Types of Semaphore

- **Binary Semaphores:** The value of a binary semaphore can range only between 0 and 1.
- **Counting Semaphores:** can range over an unrestricted domain Counting semaphore can range over an unrestricted domain. i.e. $-\infty$ to $+\infty$.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- **Implementation of semaphore** This simple implementation of semaphore with this wait(S) and signal(S) function suffer from busy waiting.
- **Modified Wait**
 - To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not >0, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

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

- **Modified Signal**
 - A process that is blocked, waiting on a semaphore S , should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

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

To implement semaphores under this definition, we define a semaphore as follows:

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

Each semaphore has an integer value and a list of processes list.

- The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

Q Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12 V(S) operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _____. (GATE-2016) (1 Marks)

Q The P and V operations on counting semaphores, where s is a counting semaphore, are defined as follows:

P(s) : $s = s - 1;$

if ($s < 0$) then wait;

V(s) : $s = s + 1;$

if ($s \leq 0$) then wakeup a process waiting on s;

Assume that P_b and V_b the wait and signal operations on binary semaphores are provided. Two binary semaphores X_b and Y_b are used to implement the semaphore operations $P(s)$ and $V(s)$ as follows:

| | |
|-------------|--|
| P(s) | $P_b(X_b);$ $s = s - 1;$ if ($s < 0$) { $V_b(X_b);$ $P_b(Y_b);$ } else $V_b(X_b);$ |
| V(s) | $P_b(X_b);$ $s = s + 1;$ if ($s \leq 0$) $V_b(Y_b);$ $V_b(X_b);$ |

The initial values of X_b and Y_b are respectively (GATE-2008)

(2 Marks)

(A) 0 and 0

(B) 0 and 1

(C) 1 and 0

(D) 1 and 1

Q A counting semaphore was initialized to 10. Then 6P (wait) operations and 4V (signal) operations were completed on this semaphore. The resulting value of the semaphore is **(GATE-1998) (1 Marks)**

a) 0

b) 8

c) 10

d) 12

Q At a particular time of computation the value of a counting semaphore is 7. Then 20 P operations and 15V operations were completed on this semaphore. If the new value of semaphore is will be **(GATE-1992) (1 Marks)**

(A) 42

(B) 2

(C) 7

(D) 12

Q.46 Consider a multi-threaded program with two threads T1 and T2. The threads share two semaphores: s1 (initialized to 1) and s2 (initialized to 0). The threads also share a global variable x (initialized to 0). The threads execute the code shown below. **(Gate 2024 CS)**

```
// code of T1  
wait(s1);  
x = x+1;  
print(x);  
wait(s2);  
signal(s1);
```

```
// code of T2  
wait(s1);  
x = x+1;  
print(x);  
signal(s2);  
signal(s1);
```

Which of the following outcomes is/are possible when threads T1 and T2 execute concurrently?

- (a) T1 runs first and prints 1, T2 runs next and prints 2
- (b) T2 runs first and prints 1, T1 runs next and prints 2
- (c) T1 runs first and prints 1, T2 does not print anything (deadlock)
- (d) T2 runs first and prints 1, T1 does not print anything (deadlock)

Disable Interrupt

- This could be a hardware solution where process have a privilege instruction, i.e. before entering into critical section, process will disable all the interrupts and at the time of exit, it again enable interrupts.
- This solution is only used by OS, as if some user process enter into critical section, then can block the entire system.
- Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

```
Pi()  
{  
    While(T)  
    {  
        Initial Section  
        Entry Section//Disable interrupt  
        Critical Section  
        Exit Section//Enable interrupt  
        Remainder Section  
    }  
}
```

Hardware Type Solution Test and Set

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software, all these solutions are based on the premise of locking —that is, protecting critical regions through the use of locks.
- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

```
Boolean test and set (Boolean *target)    While(1)
{
    Boolean rv = *target;
    *target = true;
    return rv;
}
```

- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word atomically —that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner.
- The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Q Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available.

| | |
|--|-------------------------------------|
| AcquireLock(L) { while (Fetch_And_Add(L,1)) L = 1; } | ReleaseLock (L) { L = 0; } |
|--|-------------------------------------|

This implementation (**GATE-2012**) (2 Marks)

- (A) fails as L can overflow
- (B) fails as L can take on a non-zero value when the lock is actually available
- (C) works correctly but may starve some processes
- (D) works correctly without starvation

Q The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
    while test-and-set(X) ;
}
```

```
void leave_CS(X)
{
    X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0. Now consider the following statements: **(GATE-2009) (2 Marks)**

- I. The above solution to CS problem is deadlock-free
- II. The solution is starvation free.
- III. The processes enter CS in FIFO order.
- IV. More than one process can enter CS at the same time.

Which of the above statements is TRUE?

- (A) I only
- (B) I and II
- (C) II and III
- (D) IV only

Q The atomic fetch-and-set x, y instruction unconditionally sets the memory location x to 1 and fetches the old value of x in y without allowing any intervening access to the memory location x. consider the following implementation of P and V functions on a binary semaphore S. Which one of the following is true? **(GATE-2006) (2 Marks)**

- (A)** The implementation may not work if context switching is disabled in P
- (B)** Instead of using fetch-and –set, a pair of normal load/store can be used
- (C)** The implementation of V is wrong
- (D)** The code does not implement a binary semaphore

```
void P (binary_semaphore *s)
{
    unsigned y;
    unsigned *x = &(s->value);
    do
    {
        fetch-and-set x, y;
    }
    while (y);
}
```

```
void V (binary_semaphore *s)
{
    S->value = 0;
}
```

Q The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as $S_0 = 1$, $S_1 = 0$, $S_2 = 0$. (GATE-2010) (2 Marks)

How many times will process P_0 print '0'?

(A) At least twice

(B) Exactly twice

(C) Exactly thrice

(D) Exactly once

| Process P_0 | Process P_1 | Process P_2 |
|-------------------|-------------------|-------------------|
| While(true){ | Wait(S_1); | Wait(S_2); |
| Wait(S_0); | Release(S_0); | Release(S_0); |
| Print'0' | | |
| Release(S_1); | | |
| Release(S_2); | | |
| } | | |

Q Suppose a processor does not have any stack pointer register. Which of the following statements is true? **(GATE-2001) (1 Marks)**

- (A)** It cannot have subroutine call instruction
- (B)** It can have subroutine call instruction, but no nested subroutine calls
- (C)** Nested subroutine calls are possible, but interrupts are not
- (D)** All sequences of subroutine calls and also interrupts are possible

Q Each Process P_i , $i = 1 \dots 9$ is coded as follows (**GATE-1997**) (1 Marks)

repeat

 P(mutex)

 {Critical section}

 V(mutex)

forever

The code for P_{10} is identical except it uses V(mutex) in place of P(mutex). What is the largest number of processes that can be inside the critical section at any moment?

- (A) 1
- (B) 2
- (C) 3
- (D) None of above

Q Consider the following proposed solution for the critical section problem. There are n processes: $P_0 \dots P_{(n-1)}$. In the code, function pmax returns an integer not smaller than any of its arguments. For all i, $t[i]$ is initialized to zero.

Code for P_i :

```
do {
    c[i]=1; t[i] = pmax(t[0],...,t[n-1])+1; c[i]=0;
    for every j ≠ i in {0,...,n-1} {
        while (c[j]);
        while (t[j] != 0 && t[j]<=t[i]);
    }
    Critical Section;
    t[i]=0;
    Remainder Section;
} while (true);
```

Which one of the following is TRUE about the above solution? **(GATE-2016) (2 Marks)**

- (a)** At most one process can be in the critical section at any time
- (b)** The bounded wait condition is satisfied
- (c)** The progress condition is satisfied
- (d)** It cannot cause a deadlock

Q Consider the following pseudocode, where S is a semaphore initialized to 5 in line #2 and counter is a shared variable initialized to 0 in line #1. Assume that the increment operation in line #7 is not atomic.

```
int counter =0;  
Semaphore S= init(5);  
void parop(void)  
{  
    wait(S);  
    wait(S);  
    counter++;  
    signal(S)  
    signal(S);  
}
```

If five threads execute the function parop concurrently, which of the following program behavior(s) is/are possible? **(GATE 2021)**
(2 MARKS)

- (a)** The value of counter is 5 after all the threads successfully complete the execution of parop
- (b)** The value of counter is 1 after all the threads successfully complete the execution of parop
- (c)** The value of counter is 0 after all the threads successfully complete the execution of parop
- (d)** There is a deadlock involving all the threads

Q Consider a computer system with multiple shared resource types, with one instance per resource type. Each instance can be owned by only one process at a time. Owning and freeing of resources are done by holding a global lock (L). The following scheme is used to own a resource instance:

function OWNRESOURCE(Resource R)

 Acquire lock L // a global lock

 if R is available then

 Acquire R

 Release lock L

 else

 if R is owned by another process P then

 Terminate P, after releasing all resources owned by P

 Acquire R

 Restart P

 Release lock L

 end if

 end if

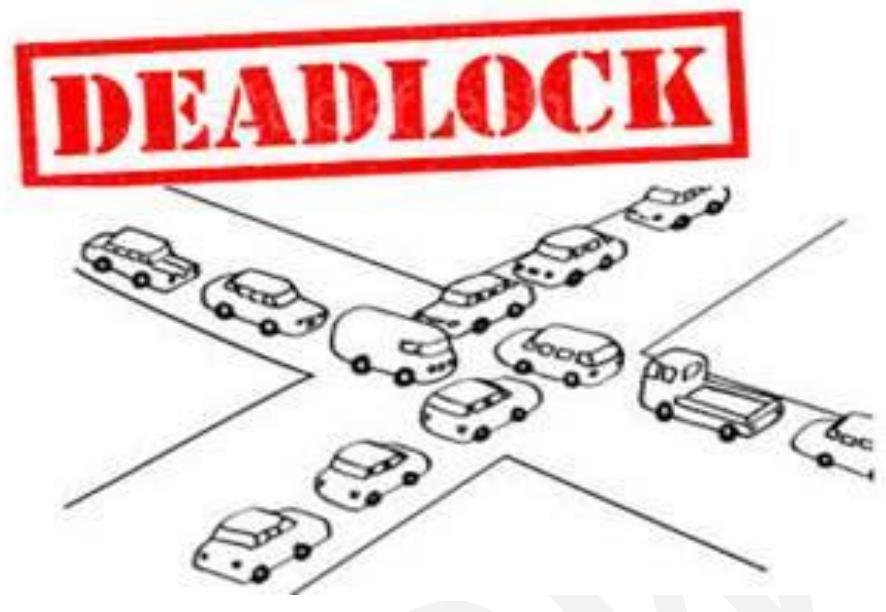
end function

Which of the following choice(s) about the above scheme is/are correct? **(GATE 2021) (2 MARKS)**

- a) The scheme ensures that deadlocks will not occur
- b) The scheme may lead to live-lock
- c) The scheme may lead to starvation
- d) The scheme violates the mutual exclusion property

Basics of Dead-Lock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- In these scenario there is a possibility of Deadlock



P₁

P₂

R₁

www.knowledgegate.in

R₂



Tax



Services



- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. Starvation is long waiting but deadlock is infinite waiting

Pehele aap



Pehele aap

System model

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

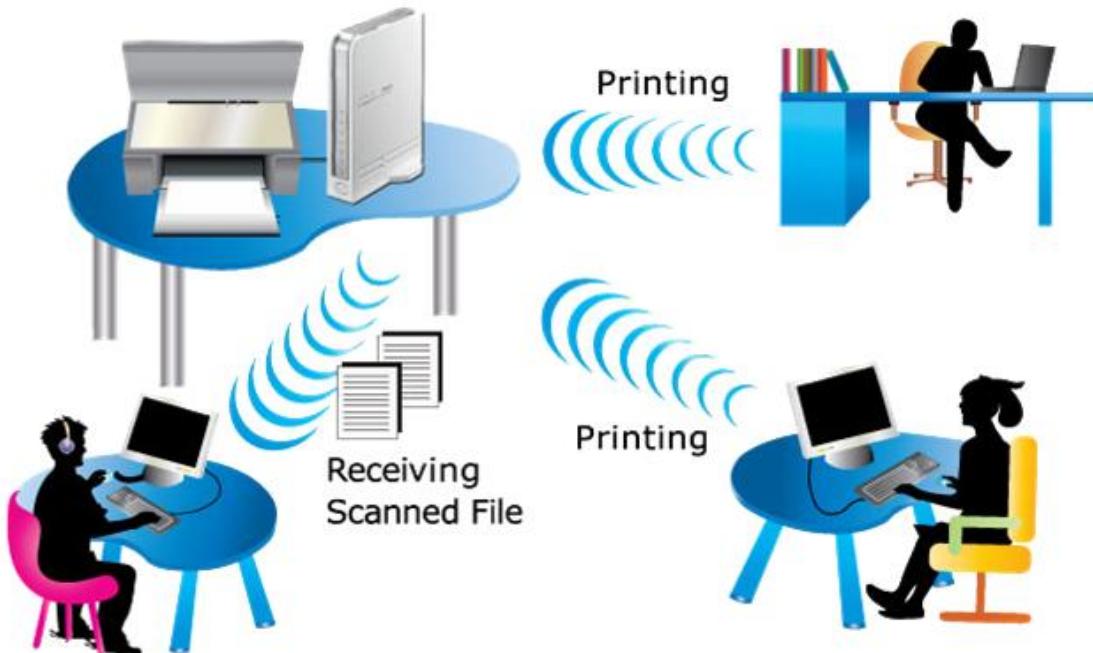
- **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release**. The process releases the resource.

Necessary conditions for deadlock

A deadlock can occur if all these 4 conditions occur in the system simultaneously.

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

- **Mutual exclusion**: - At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.
- If another process requests that resource, the requesting process must be delayed until the resource has been released. And the resource Must be desired by more than one process.



- **Hold and wait**: - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes. E.g. Plate and spoon



- **No pre-emption:** - Resources cannot be pre-empted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.



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

Deadlock Handling methods

- **Prevention**: - Design such protocols that there is no possibility of deadlock.
- **Avoidance**: - Try to avoid deadlock in run time so ensuring that the system will never enter a deadlocked state.
- **Detection**: - We can allow the system to enter a deadlocked state, then detect it, and recover.
- **Ignorance**: - We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Prevention

- It means designing such systems where there is no possibility of existence of deadlock. For that we have to remove one of the four necessary condition of deadlock.

Polio vaccine



- **Mutual exclusion**: -In prevention approach, there is no solution for mutual exclusion as resource can't be made sharable as it is a hardware property and process also can't be convinced to do some other task.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold & wait

- In conservative approach, process is allowed to run if & only if it has acquired all the resources.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Wait time outs we place a max time outs up to which a process can wait. After which process must release all the holding resources & exit.

No pre-emption

- if a process requests some resources, we first check whether they are available. If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be pre-empted, but only if another process requests them.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting.

Circular wait

- We can eliminate circular wait problem by giving a natural number mapping to every resource and then any process can request only in the increasing order and if a process wants a lower number, then process must first release all the resource larger than that number and then give a fresh request.

Q Consider the following policies for preventing deadlock in a system with mutually exclusive resources.

- I)** Process should acquire all their resources at the beginning of execution. If any resource is not available, all resources acquired so far are released.
- II)** The resources are numbered uniquely, and processes are allowed to request for resources only in increasing resource numbers
- III)** The resources are numbered uniquely, and processes are allowed to request for resources only in decreasing resource numbers
- IV)** The resources are numbered uniquely. A process is allowed to request for resources only for a resource with resource number larger than its currently held resources

Which of the above policies can be used for preventing deadlock? **(GATE-2015) (2 Marks)**

- (A)** Any one of I and III but not II or IV
- (B)** Any one of I, III and IV but not II
- (C)** Any one of II and III but not I or IV
- (D)** Any one of I, II, III and IV

Q An operating system implements a policy that requires a process to release all resources before making a request for another resource. Select the TRUE statement from the following: **(GATE-2008) (2 Marks)**

- (A)** Both starvation and deadlock can occur
- (B)** Starvation can occur but deadlock cannot occur
- (C)** Starvation cannot occur but deadlock can occur
- (D)** Neither starvation nor deadlock can occur

**Q Which of the following is NOT a valid deadlock prevention scheme? (GATE-2000)
(2 Marks)**

- (A) Release all resources before requesting a new resource**
- (B) Number the resources uniquely and never request a lower numbered resource than the last one requested.**
- (C) Never request a resource after releasing any resource**
- (D) Request all required resources be allocated before execution.**

Q Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of K instances. Resource instances can be requested and released only one at a time. The largest value of K that will always avoid deadlock is _____. **(GATE-2018) (1 Marks)**

Q A system has 6 identical resources and N processes competing for them. Each process can request at most 2 resources. Which one of the following values of N could lead to a deadlock? **(GATE-2015) (1 Marks)**

- a) 1
- b) 2
- c) 3
- d) 6

Q A system contains three programs and each requires three tape units for its operation. The minimum number of tape units which the system must have such that deadlocks never arise is **(GATE-2014) (2 Marks)**

- (A) 6
- (B) 7
- (C) 8
- (D) 9

Q. Suppose n processes, P_1, \dots, P_n share m identical resource units, which can be reserved and released one at a time. The maximum resource requirement of process P_i is S_i , where $S_i > 0$. Which one of the following is a sufficient condition for ensuring that deadlock does not occur? (GATE 2005, 2 Marks)

- a) $\forall i, S_i < m$
- b) $\forall i, S_i < n$
- c) $\sum_{i=1}^n S_i < (m + n)$
- d) $\sum_{i=1}^n S_i < (m * n)$

Q A computer has six tape drives, with n processes competing for them. Each process may need two drives. What is the maximum value of n for the system to be deadlock free? **(GATE-1998) (2 Marks)**

- (A) 6
- (B) 5
- (C) 4
- (D) 3

Q An operating system contains 3 user processes each requiring 2 units of resource R . The minimum number of units of R such that no deadlocks will ever arise is **(GATE-1997) (2 Marks)**

- (A) 3
- (B) 5
- (C) 4
- (D) 6

Q Consider a system having m resources of the same type. These resources are shared by 3 processes A, B and C, which have peak demands of 3, 4 and 6 respectively. For what value of m deadlock will not occur? (GATE-1993) (2 Marks)

- (a) 7
- (b) 9
- (c) 10
- (d) 13
- (e) 15

Q A computer system has 6 tape drives, with n process completing for them. Each process may need 3 tape drives. The maximum value of n for which the system is guaranteed to be deadlock free is **(GATE-1992) (2 Marks)**

(a) 2

(b) 3

(c) 4

(d) 1

- **Problem with Prevention:** - Different deadlock Prevention approach put different type of restrictions or conditions on the processes and resources Because of which system becomes slow and resource utilization and reduced system throughput.
- So, in order to avoid deadlock in run time, System try to maintain some books like a banker, whenever someone ask for a loan(resource), it is granted only when the books allow.



Avoidance

- To avoid deadlocks we require additional information about how resources are to be requested. which resources a process will request and use during its lifetime i.e. maximum number of resources of each type that it may need.
- With this additional knowledge, the operating system can decide for each request whether process should wait or not.

| | Max Need | | |
|----------------|----------|---|---|
| | E | F | G |
| P ₀ | 4 | 3 | 1 |
| P ₁ | 2 | 1 | 4 |
| P ₂ | 1 | 3 | 3 |
| P ₃ | 5 | 4 | 1 |

| | Allocation | | |
|----------------|------------|---|---|
| | E | F | G |
| P ₀ | 1 | 0 | 1 |
| P ₁ | 1 | 1 | 2 |
| P ₂ | 1 | 0 | 3 |
| P ₃ | 2 | 0 | 0 |

| | Current Need | | |
|----------------|--------------|---|---|
| | E | F | G |
| P ₀ | 3 | 3 | 0 |
| P ₁ | 1 | 0 | 2 |
| P ₂ | 0 | 3 | 0 |
| P ₃ | 3 | 4 | 1 |

| System Max | | |
|------------|---|---|
| E | F | G |
| 8 | 4 | 6 |

| Available | | |
|-----------|---|---|
| E | F | G |
| 3 | 3 | 0 |

Q A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST? (GATE-2007) (2 Marks)

(A) P_0

(B) P_1

(C) P_2

| | Allocation | | | request | | |
|-------|------------|---|---|---------|---|---|
| | X | Y | Z | X | Y | Z |
| P_0 | 1 | 2 | 1 | 1 | 0 | 3 |
| P_1 | 2 | 0 | 1 | 0 | 1 | 2 |
| P_2 | 2 | 2 | 1 | 1 | 2 | 0 |

(D) None of the above, since the system is in a deadlock

Banker's Algorithm

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type R_j are available.

| Available | | |
|-----------|---|---|
| E | F | G |
| 3 | 3 | 0 |

Max: An n*m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process P_i may request at most k instances of resource type R_j.

| | Max Need | | |
|----------------|----------|---|---|
| | E | F | G |
| P ₀ | 4 | 3 | 1 |
| P ₁ | 2 | 1 | 4 |
| P ₂ | 1 | 3 | 3 |
| P ₃ | 5 | 4 | 1 |

Allocation: An $n*m$ matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process P_i is currently allocated k instances of resource type R_j .

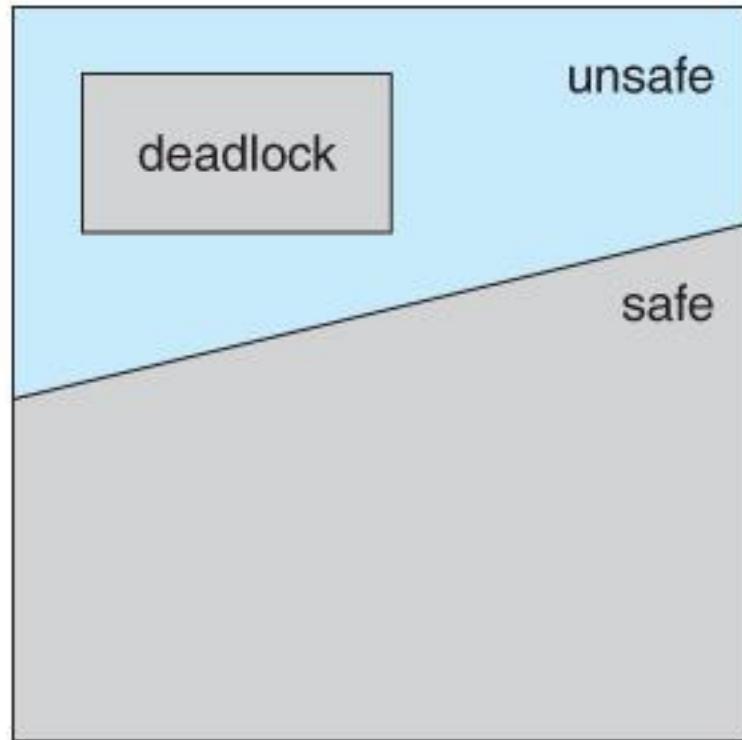
| | Allocation | | |
|-------|------------|---|---|
| | E | F | G |
| P_0 | 1 | 0 | 1 |
| P_1 | 1 | 1 | 2 |
| P_2 | 1 | 0 | 3 |
| P_3 | 2 | 0 | 0 |

Need/Demand/Requirement: An $n*m$ matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$. These data structures vary over time in both size and value.

| | Current Need | | |
|-------|--------------|---|---|
| | E | F | G |
| P_0 | 3 | 3 | 0 |
| P_1 | 1 | 0 | 2 |
| P_2 | 0 | 3 | 0 |
| P_3 | 3 | 4 | 1 |

- A deadlock-avoidance algorithm dynamically examines the resource-allocation state.
- The resource- allocation state is defined by the number of available and allocated resources and the maximum demands of the processes before allowing that request first.
- We check, if there exist “some sequence in which we can satisfies demand of every process without going into deadlock, if yes, this sequence is called safe sequence” and request can be allowed. Otherwise there is a possibility of going into deadlock.

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.



Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

- 1- Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for $i = 0, 1, \dots, n - 1$.
- 2- Find an index i such that both
 - Finish[i] == false
 - $\text{Need}_i \leq \text{Work}$If no such i exists, go to step 4.
- 3- $\text{Work} = \text{Work} + \text{Allocation}_i$
 - Finish[i] = true
 - Go to step 2.
- 4- If $\text{Finish}[i] == \text{true}$ for all i, then the system is in a safe state.

This algorithm may require an order of $m * n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let Request_i be the request vector for process P_i . If $\text{Request}_i[j] = k$, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1- If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2- If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3- Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

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

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

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

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for Request_i , and the old resource-allocation state is restored.

Q Consider the following snapshot of a system running n concurrent processes. Process i is holding X_i instances of a resource R , $1 \leq i \leq n$. Assume that all instances of R are currently in use. Further, for all i , process i can place a request for at most Y_i additional instances of R while holding the X_i instances it already has. Of the n processes, there are exactly two processes p and q such that $Y_p = Y_q = 0$. Which one of the following conditions guarantees that no other process apart from p and q can complete execution? **(GATE-2019) (2 Marks)**

- a) $X_p + X_q < \text{Min}\{Y_k \mid 1 \leq k \leq n, k \neq p, k \neq q\}$
- b) $X_p + X_q < \text{Max}\{Y_k \mid 1 \leq k \leq n, k \neq p, k \neq q\}$
- c) $\text{Min}(X_p, X_q) \geq \text{Min}\{Y_k \mid 1 \leq k \leq n, k \neq p, k \neq q\}$
- d) $\text{Min}(X_p, X_q) \leq \text{Max}\{Y_k \mid 1 \leq k \leq n, k \neq p, k \neq q\}$

Q In a system, there are three types of resources: E, F and G. Four processes P_0 , P_1 , P_2 and P_3 execute concurrently. At the outset, the processes have declared their maximum resource requirements using a matrix named Max as given below. For example, $\text{Max}[P_2, F]$ is the maximum number of instances of F that P_2 would require. The number of instances of the resources allocated to the various processes at any given state is given by a matrix named Allocation.

| | Allocation | | | Max | | |
|-------|------------|---|---|-----|---|---|
| | E | F | G | E | F | G |
| P_0 | 1 | 0 | 1 | 4 | 3 | 1 |
| P_1 | 1 | 1 | 2 | 2 | 1 | 4 |
| P_2 | 1 | 0 | 3 | 1 | 3 | 3 |
| P_3 | 2 | 0 | 0 | 5 | 4 | 1 |

Consider a state of the system with the Allocation matrix as shown below, and in which 3 instances of E and 3 instances of F are the only resources available.

From the perspective of deadlock avoidance, which one of the following is true? **(GATE-2018) (2 Marks)**

- (A) The system is in *safe* state
- (B) The system is not in *safe* state, but would be *safe* if one more instance of E were available
- (C) The system is not in *safe* state, but would be *safe* if one more instance of F were available
- (D) The system is not in *safe* state, but would be *safe* if one more instance of G were available

Q A system shares 9 tape drives. The current allocation and maximum requirement of tape drives for 3 processes are shown below: Which of the following best describes the current state of the system? **(GATE-2014) (2 Marks)**

(A) Safe, Deadlocked

(B) Safe, Not Deadlocked

(C) Not Safe, Deadlocked

(D) Not Safe, Not Deadlocked

| Process | Current Allocation | Maximum Requirement | Current Requirement | Current Available |
|----------------|--------------------|---------------------|---------------------|-------------------|
| P ₁ | 3 | 7 | | |
| P ₂ | 1 | 6 | | |
| P ₃ | 3 | 5 | | |

Q An operating system uses the Banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y, and Z to three processes P_0 , P_1 , and P_2 . The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution. There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state. Consider the following independent requests for additional resources in the current state:

REQ₁: P_0 requests 0 units of X, 0 units of Y and 2 units of Z

REQ₂: P_1 requests 2 units of X, 0 units of Y and 0 units of Z

Which one of the following is TRUE? (GATE-2014) (2 Marks)

- (A) Only **REQ₁** can be permitted.
- (B) Only **REQ₂** can be permitted.
- (C) Both **REQ₁** and **REQ₂** can be permitted.
- (D) Neither **REQ₁** nor **REQ₂** can be permitted

| | Allocation | | | Max | | | | | |
|-------|------------|---|---|-----|---|---|--|--|--|
| | X | Y | Z | X | Y | Z | | | |
| P_0 | 0 | 0 | 1 | 8 | 4 | 3 | | | |
| P_1 | 3 | 2 | 0 | 6 | 2 | 0 | | | |
| P_2 | 2 | 1 | 1 | 3 | 3 | 3 | | | |

Q Which of the following is NOT true of deadlock prevention and deadlock avoidance schemes? **(GATE-2008) (2 Marks)**

- (A)** In deadlock prevention, the request for resources is always granted if the resulting state is safe
- (B)** In deadlock avoidance, the request for resources is always granted if the result state is safe
- (C)** Deadlock avoidance is less restrictive than deadlock prevention
- (D)** Deadlock avoidance requires knowledge of resource requirements a priori

Q Consider the following snapshot of a system running n processes. Process P_i is holding X_i instances of a resource R , $1 \leq i \leq n$. currently, all instances of R are occupied. Further, for all i , process i has placed a request for an additional Y_i instances while holding the X_i instances it already has. There are exactly two processes p and q such that $Y_p = Y_q = 0$. Which one of the following can serve as a necessary condition to guarantee that the system is not approaching a deadlock? **(GATE-2006) (2 Mark)**

(A) $\min(X_p, X_q) < \max(Y_k)$ where $k \neq p$ and $k \neq q$

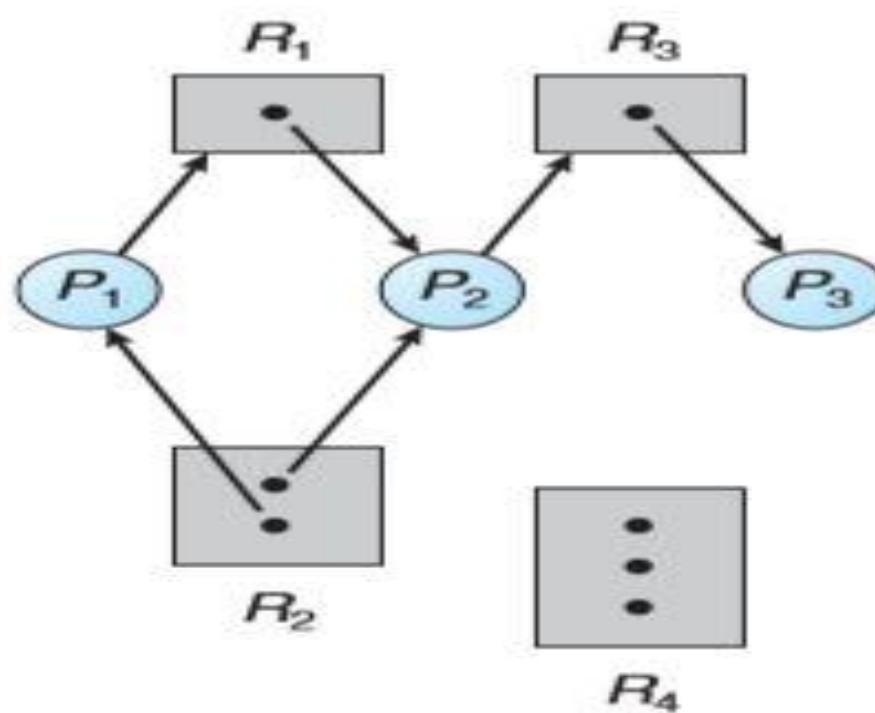
(B) $X_p + X_q \geq \min(Y_k)$ where $k \neq p$ and $k \neq q$

(C) $\max(X_p, X_q) > 1$

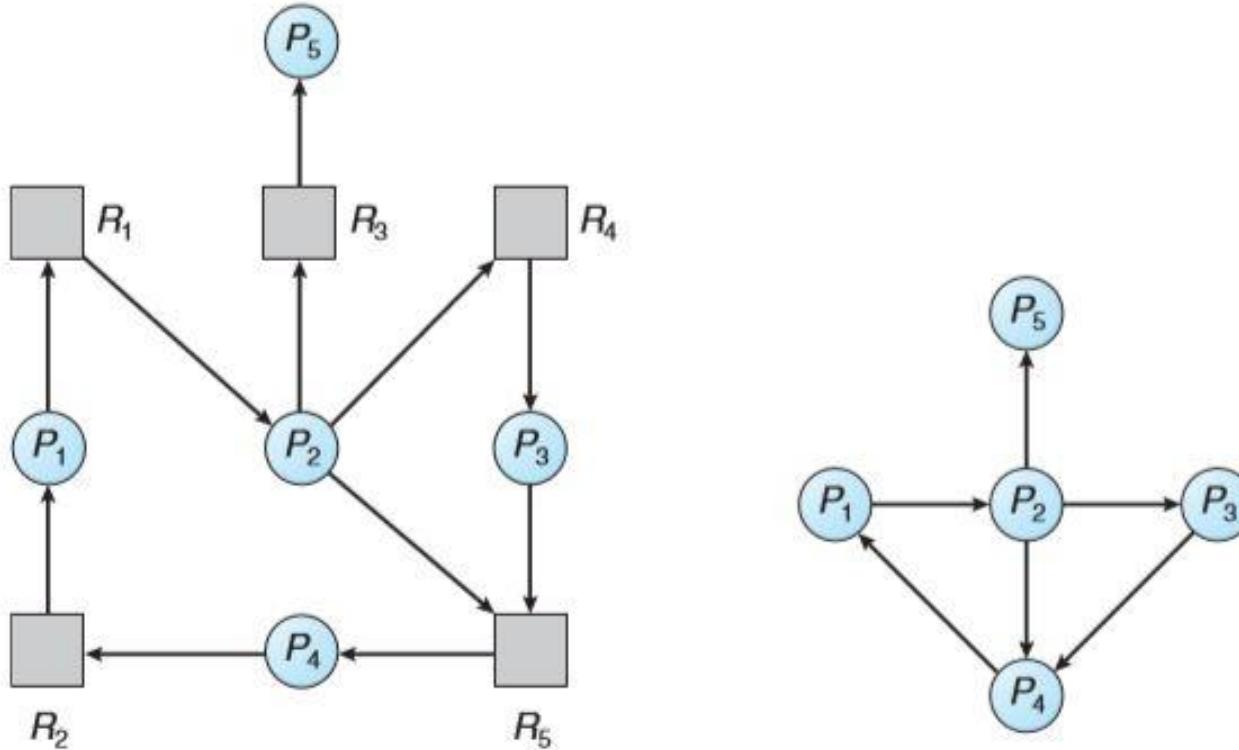
(D) $\min(X_p, X_q) > 1$

Resource Allocation Graph

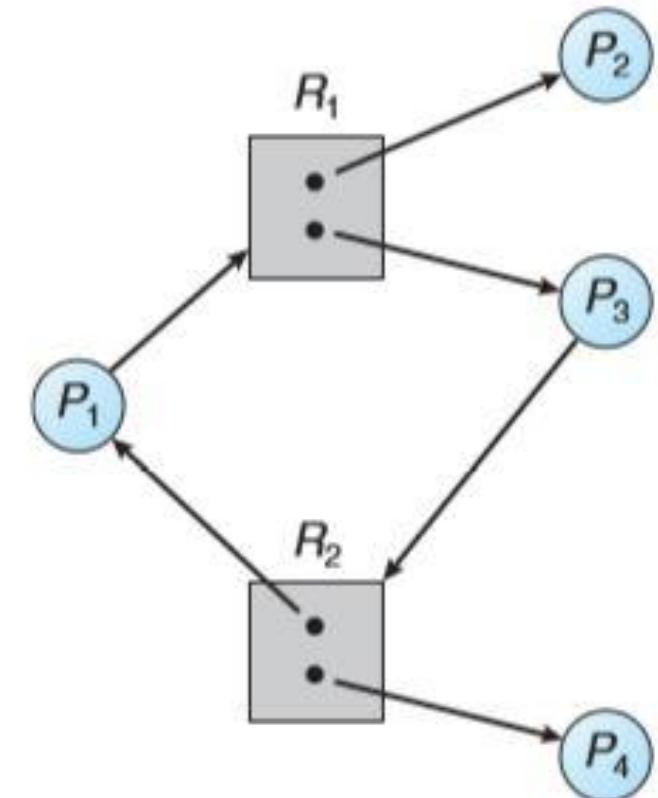
- Deadlock can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.



- A directed edge from process P_i to resource type R_j is denoted by $P_i \square R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- A directed edge from resource type R_j to process P_i is denoted by $R_j \square P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i .
- A directed edge $P_i \square R_j$ is called a request edge; a directed edge $R_j \square P_i$ is called an assignment edge.



- Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j , whereas an assignment edge must also designate one of the dots in the rectangle.
- When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.



- Cycle in resource allocation graph is necessary but not sufficient condition for detection of deadlock.
- If every resource have only one resource in the resource allocation graph than detection of cycle is necessary and sufficient condition for deadlock detection.

Q Which of the following statements is/are TRUE with respect to deadlocks? (GATE 2022) (1 MARKS)

- (A) Circular wait is a necessary condition for the formation of deadlock.
- (B) In a system where each resource has more than one instance, a cycle in its wait-for graph indicates the presence of a deadlock.
- (C) If the current allocation of resources to processes leads the system to unsafe state, then deadlock will necessarily occur.
- (D) In the resource-allocation graph of a system, if every edge is an assignment edge, then the system is not in deadlock state.

Deadlock detection and recovery

- Here we do not check safety and where any process request for some resources then these resources are allocated immediately, if available.
- Here there is a possibility of deadlock, which must be detected using different approaches.
- **Active approach:** Here we simply invoke the algorithm at defined intervals—for example, once per hour. or whenever CPU utilization drops below 40 percent.
- **Lazy approach:** whenever CPU utilization drops below 40 percent or some unusual performance is there, we go for Detection.

Resource-Request Algorithm

- Next, we describe the algorithm for determining whether requests can be safely granted. Let Request_i be the request vector for process P_i . If $\text{Request}_i[j] = k$, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1- If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2- If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3- Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

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

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

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

- If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for Request_i , and the old resource-allocation state is restored.

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock.

Process Termination

- **Abort All Deadlocked Processes:** This direct method guarantees the deadlock will be broken by terminating all involved processes. However, it is costly since all the computation done by these processes is lost and must be repeated later.
- **Abort One Process at a Time:** This more gradual approach involves aborting one deadlocked process at a time and then running a deadlock-detection algorithm to check if the deadlock still persists. This method can be resource-intensive due to the repeated checks needed after each process termination.
- **Challenges with Process Abortion:** Terminating a process can be problematic, especially if the process was engaged in critical operations like updating files or printing. Abrupt termination could leave resources in an inconsistent state, requiring additional steps to restore them to a correct state.

How to choose a victim

- **Decision Criteria:** Choosing which processes to terminate in a deadlock situation is an economic decision, similar to CPU scheduling. The aim is to minimize the cost associated with terminating a process.
- **Factors to Consider:**
 - **Process Priority:** Higher priority processes might be less likely to be terminated.
 - **Computation Time:** Consideration of how long the process has run and how much longer it needs to complete its tasks.
 - **Resource Usage:** The types and quantities of resources the process has used, and whether these resources are easily preemptible.
 - **Resource Needs:** The additional resources needed by the process to complete its task.
 - **Process Type:** Whether the process is part of an interactive system or a batch-processing system.
 - **Number of Processes Affected:** The total number of processes that might need to be terminated to resolve the deadlock.

Resource Pre-emption

- **Resource Pre-emption**: To resolve deadlocks, resources are preemptively taken from certain processes and reassigned to others until the deadlock is broken.
- **Selecting a Victim**: Deciding which resources and processes to preempt involves considering cost factors such as the number of resources held and the time the process has consumed.
- **Rollback**: Processes from which resources are preempted must be rolled back to a safe state before they can continue. This often involves aborting and restarting the process, although rolling back to the minimal necessary point is more efficient but requires detailed tracking of process states.
- **Preventing Starvation**: Ensuring no process is repeatedly chosen as a victim to prevent starvation involves including the number of rollbacks a process has undergone in the cost calculations, limiting the number of times a process can be picked as a victim.

Q Consider a system with 4 types of resources R1 (3 units), R2 (2 units), R3 (3 units), R4 (2 units). A non-pre-emptive resource allocation policy is used. At any given instance, a request is not entertained if it cannot be completely satisfied. Three processes P1, P2, P3 request the sources as follows if executed independently. (GATE-2009) (2 Marks)

| Process P1: | Process P2: | Process P3: |
|--|-----------------------------|-----------------------------|
| t=0: requests 2 units of R2 | t=0: requests 2 units of R3 | t=0: requests 1 unit of R4 |
| t=1: requests 1 unit of R3 | t=2: requests 1 unit of R4 | t=2: requests 2 units of R1 |
| t=3: requests 2 units of R1 | t=4: requests 1 unit of R1 | t=5: releases 2 units of R1 |
| t=5: releases 1 unit of R2 and 1 unit of R1. | t=6: releases 1 unit of R3 | t=7: requests 1 unit of R2 |
| t=7: releases 1 unit of R3 | t=8: Finishes | t=8: requests 1 unit of R3 |
| t=8: requests 2 units of R4 | | t=9: Finishes |
| t=10: Finishes | | |

Which one of the following statements is TRUE if all three processes run concurrently starting at time t=0?

- (A) All processes will finish without any deadlock
- (B) Only P1 and P2 will be in deadlock.
- (C) Only P1 and P3 will be in a deadlock.
- (D) All three processes will be in deadlock

| | R1 | R2 | R3 | R4 |
|-----------|----|----|----|----|
| P1 | | | | |
| P2 | | | | |
| P3 | | | | |
| Available | | | | |
| R1 | | R2 | R3 | R4 |
| | | | | |

Ignorance

- In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened.
- Operating System behaves like there is no concept of deadlock.

Ostrich Algorithm



- **Impact of Ignored Deadlocks:** Undetected deadlocks can severely degrade system performance as they monopolize resources while preventing processes from running. Over time, this can lead to more processes becoming deadlocked, ultimately causing the system to freeze and require a manual restart.
- **Reasons for Ignoring Deadlocks:** Despite the potential issues, many operating systems choose to ignore deadlocks due to cost considerations. Implementing and maintaining deadlock prevention or detection systems can be expensive. In environments where deadlocks are rare (e.g., occurring once a year), the cost of advanced deadlock management might not be justified.
- **Utilizing Existing Recovery Methods:** Operating systems that ignore deadlocks often rely on general recovery methods used for other system errors. For example, manual recovery techniques designed for other types of system freezes can also be applied to resolve deadlocks.

Q Consider the solution to the bounded buffer producer/consumer problem by using general semaphores S,F and E. The semaphore S is the mutual exclusion semaphore initialized to 1. The semaphore F corresponds to the number of free slots in the buffer and is initialized to N. The semaphore E corresponds to the number of elements in the buffer and is initialized to 0. **(GATE-2006) (2 Mark)**

Which of the following interchange operations may result in a deadlock?

- I) Interchanging Wait (F) and Wait (S) in the Producer process
- II) Interchanging Signal (S) and Signal (F) in the Consumer process

- a) (I) only
- b) (II) only
- c) Neither (I) nor (II)
- d) Both (I) and (II)

| Producer() | Consumer() |
|-------------------|--------------------|
| { | { |
| while(T) | while(T) |
| { | { |
| Produce() | wait(E)//UnderFlow |
| wait(F)//OverFlow | wait(S) |
| Wait(S) | pick() |
| append() | signal(S) |
| signal(S) | signal(F) |
| signal(E) | consume() |
| } | } |
| } | } |

Q In a certain operating system, deadlock prevention is attempted using the following scheme. Each process is assigned a unique timestamp, and is restarted with the same timestamp if killed. Let P_h be the process holding a resource R , P_r be a process requesting for the same resource R , and $T(P_h)$ and $T(P_r)$ be their timestamps respectively. The decision to wait or preempt one of the processes is based on the following algorithm.

```
if T(Pr) < T(Ph)
    then kill Pr
else wait
```

Which one of the following is TRUE? **(GATE-2004) (2 Marks)**

- (A) The scheme is deadlock-free, but not starvation-free
- (B) The scheme is not deadlock-free, but starvation-free
- (C) The scheme is neither deadlock-free nor starvation-free
- (D) The scheme is both deadlock-free and starvation-free

Q The following is a code with two threads, producer and consumer, that can run in parallel. Further, S and Q are binary semaphores equipped with the standard P and V operations. Which of the following is TRUE about the program above? **(GATE-2008) (2 Marks)**

- (A)** The process can deadlock
- (B)** One of the threads can starve
- (C)** Some of the items produced by the producer may be lost
- (D)** Values generated and stored in 'x' by the producer will always be consumed before the producer can generate a new value

| | |
|-------------------------|-----------------|
| semaphore S = 1, Q = 0; | |
| integer x; | |
| producer: | consumer: |
| while (true) do | while (true) do |
| P(S); | P(Q); |
| x = produce (); | consume (x); |
| V(Q); | V(S); |
| done | done |

Q. $P = \{P_1, P_2, P_3, P_4\}$ consists of all active processes in an operating system.

$R = \{R_1, R_2, R_3, R_4\}$ consists of single instances of distinct types of resources in the system.

The resource allocation graph has the following assignment and claim edges.

Assignment edges: $R_1 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3, R_4 \rightarrow P_4$ (the assignment edge $R_1 \rightarrow P_1$ means resource R_1 is assigned to process P_1 , and so on for others)

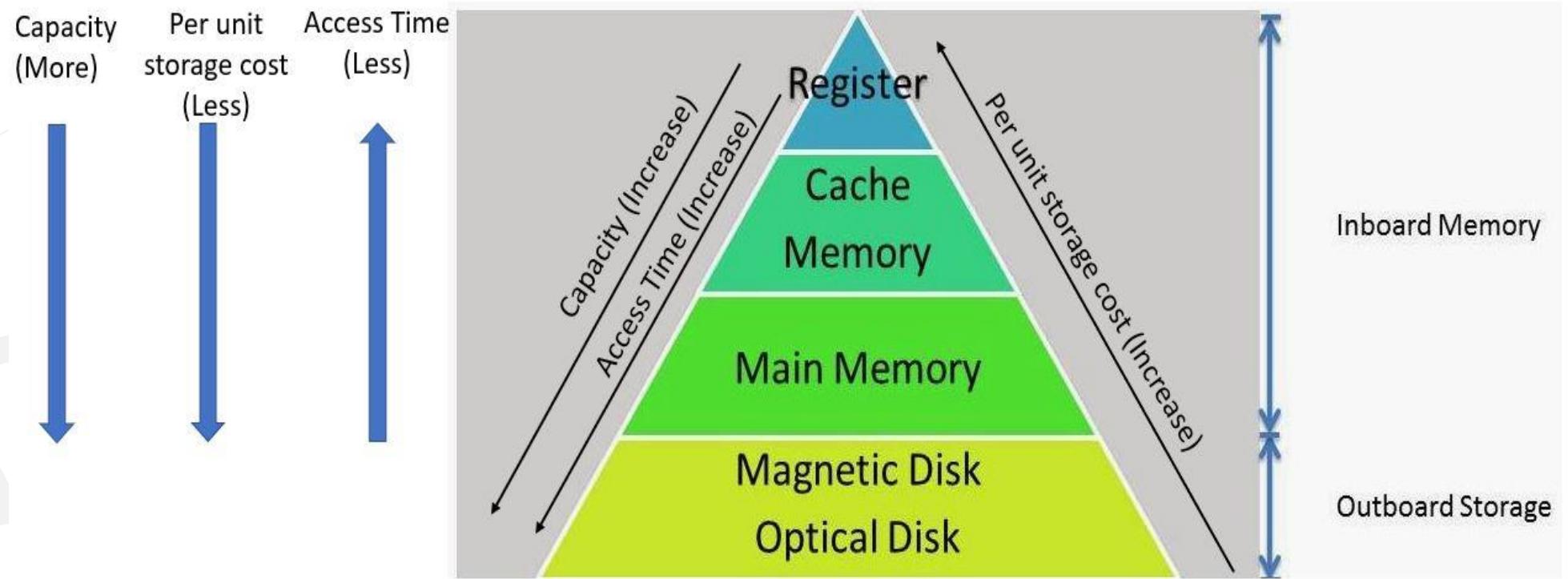
Claim edges: $P_1 \rightarrow R_2, P_2 \rightarrow R_3, P_3 \rightarrow R_1, P_2 \rightarrow R_4, P_4 \rightarrow R_2$ (the claim edge $P_1 \rightarrow R_2$ means process P_1 is waiting for resource R_2 , and so on for others)

Which of the following statement(s) is/are CORRECT? **(Gate 2025)**

- A) Aborting P_1 makes the system deadlock free.
- B) Aborting P_3 makes the system deadlock free.
- C) Aborting P_2 makes the system deadlock free.
- D) Aborting P_1 and P_4 makes the system deadlock free.

Memory Hierarchy

- The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic.
- Let first understand what we need from a memory
 - Large capacity
 - Less per unit cost
 - Less access time(fast access)





Cycle



Car



Airbus



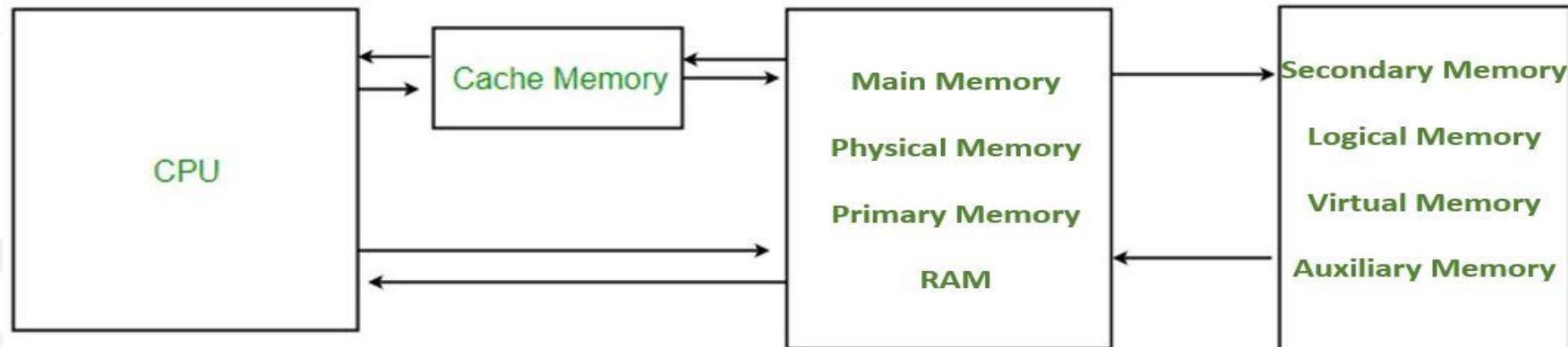
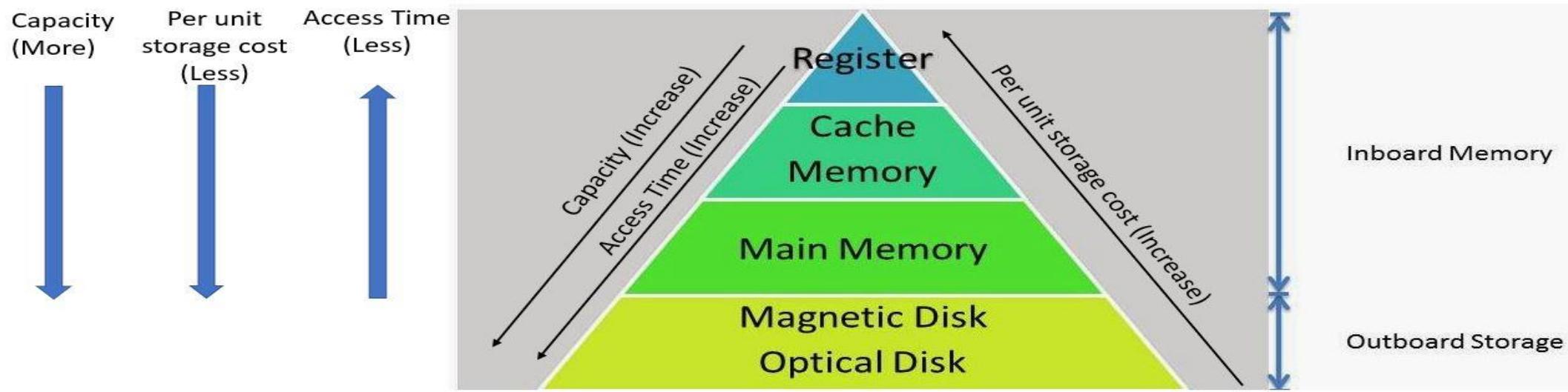
Lathi



303



AK-47





Showroom

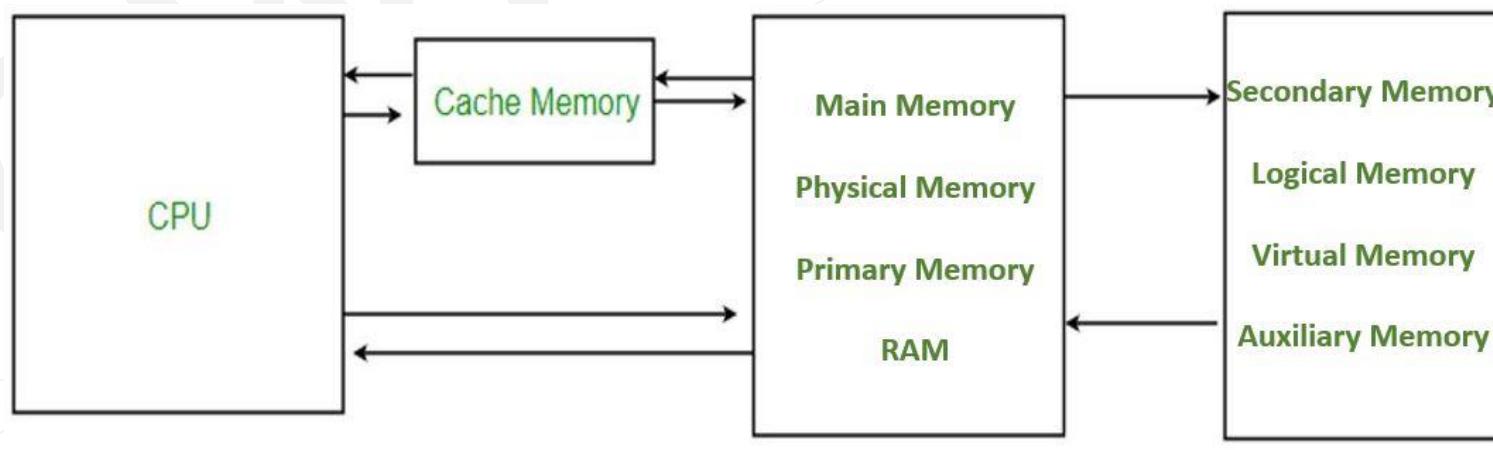


Go down



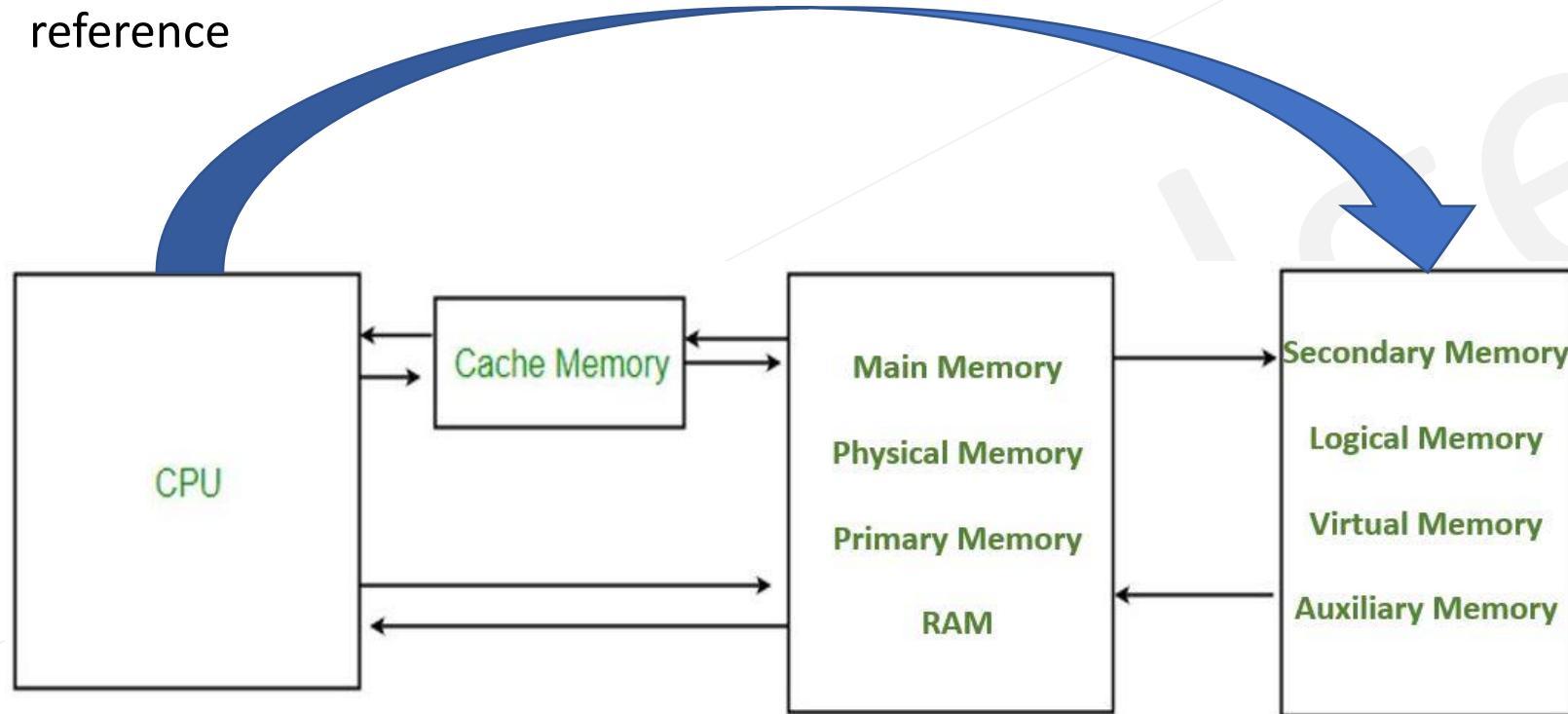
Factory

- If we want this hierarchy to work fine, then most of the time when CPU needs a data it must be present in Cache, if not possible main memory, worst case Secondary memory. But this is a difficult task to do as my computer has, 8 TB of Secondary Memory, 32 GB of Main Memory, but only 768KB, 4MB, 16 MB of L_1 , L_2 , L_3 cache respectively. If somehow we can estimate what data CPU will require in future we can prefetch in Cache and Main Memory. Locality of reference Helps us to perform this estimation.
- The main memory occupies a central position by being able to communicate directly with the CPU. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- A special very-high speed memory called a Cache is used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic.

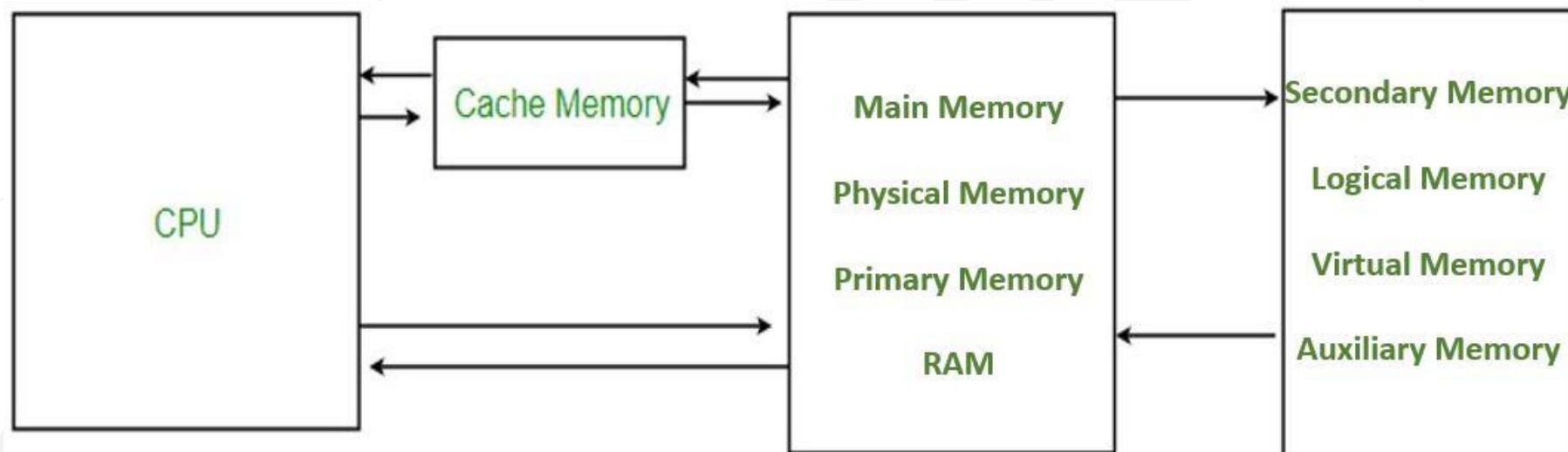


Locality of Reference

- The references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of locality of reference. There are two types of locality of reference



- **Spatial Locality:** Spatial locality refers to the use of data elements in the nearby locations.
- **Temporal Locality:** Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small-time duration. Or, the most frequently used items will be needed soon. (LRU is used for temporal locality)



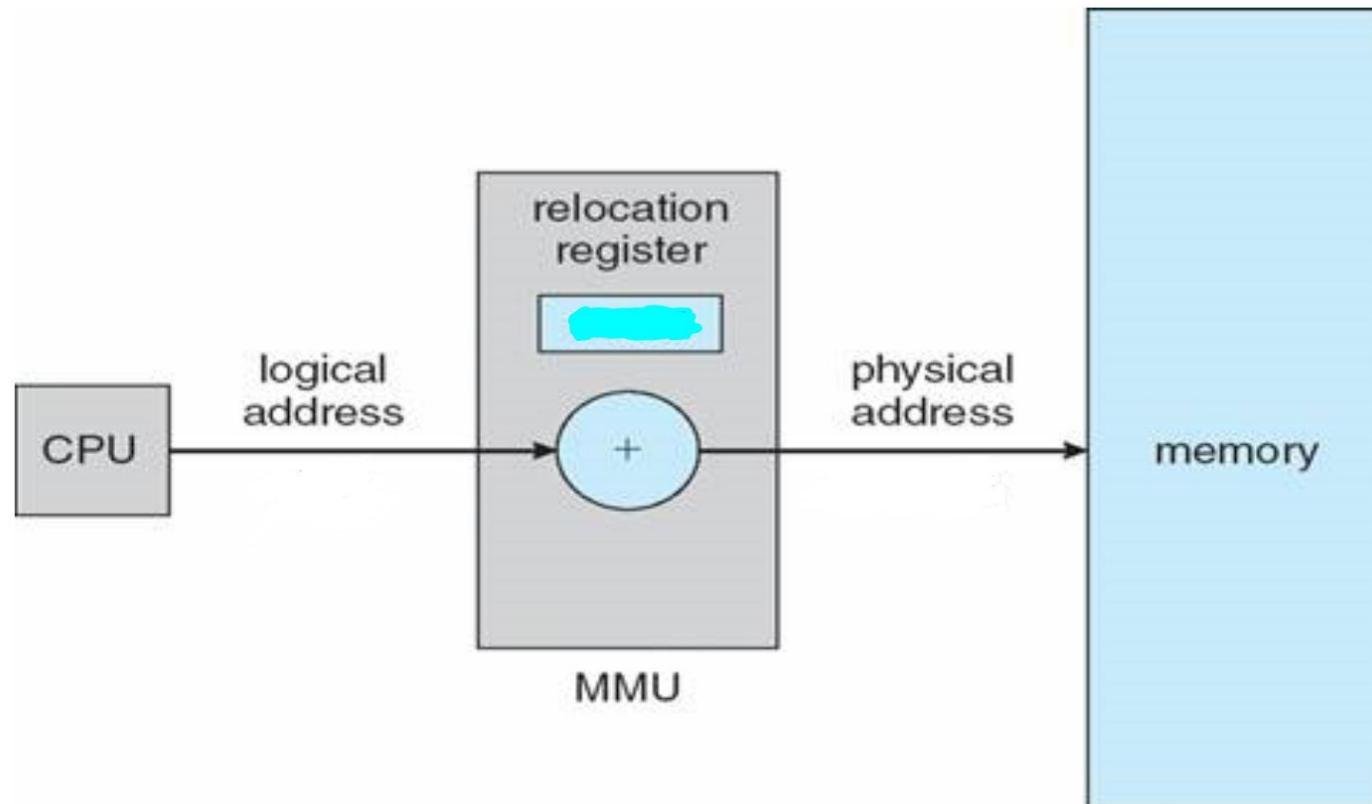
Duty of Operating System

- Operating system is responsible for the following activities in connection with memory management:
 - Address translation from Logical address to Physical address.
 - Deciding which processes (or parts of processes) and data to move into and out of memory. Allocating and deallocating memory space as needed.
 - Keeping track of which parts of memory are currently being used and who is using them.
 - The first duty of OS is to translate this logical address into physical address. There can be two approaches for storing a process in main memory.
 - Contiguous allocation policy
 - Non-contiguous allocation policy

Contiguous allocation policy

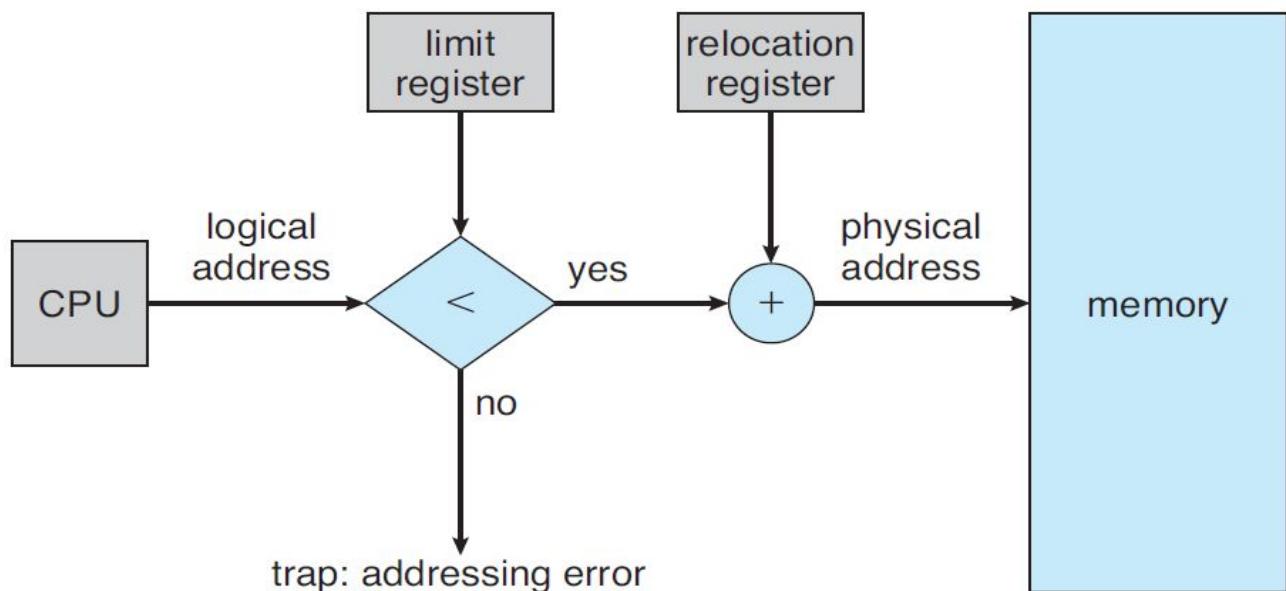
- We know that when a process is required to be executed it must be loaded to main memory, by policy has two implications.
 - It must be loaded to main memory completely for execution.
 - Must be stored in main memory in contiguous fashion.

- Here we use a memory management unit(OS) whose duty is to take logical address and translate it into physical address.
- MMU contains a relocation register, which contains the base address of the process in the main memory and it is added in the logical address every time



- **Advantage**: - easy strategy, simple to use, simple to understand.
- **Disadvantage**: - if a process generates logical address greater than its maximum limit, then it can easily access the data of some other process.

- In order to check whether address generated to CPU is valid(with in range) or invalid, we compare it with the value of limit register, which contains the max no of instructions in the process.
- So, if the value of logical address is less than limit, then it means it's a valid request and we can continue with translation otherwise, it is a illegal request which is immediately trapped by OS.



| LA | PA |
|-----|---------|
| 0 | 0 + r |
| Max | Max + r |
| Min | Min + r |

Q Consider the following tables of relocation and limit registers and find the illegal attempts and if valid find the physical address?

| | Limit Register | Relocation Register |
|----------------|----------------|---------------------|
| P ₀ | 500 | 1200 |
| P ₁ | 275 | 550 |
| P ₂ | 212 | 880 |
| P ₃ | 420 | 1400 |
| P ₄ | 118 | 200 |

Following are the request by process

P₀—450
P₁—300
P₂—210
P₃—450
P₄---80

Space Allocation Method in Contiguous Allocation

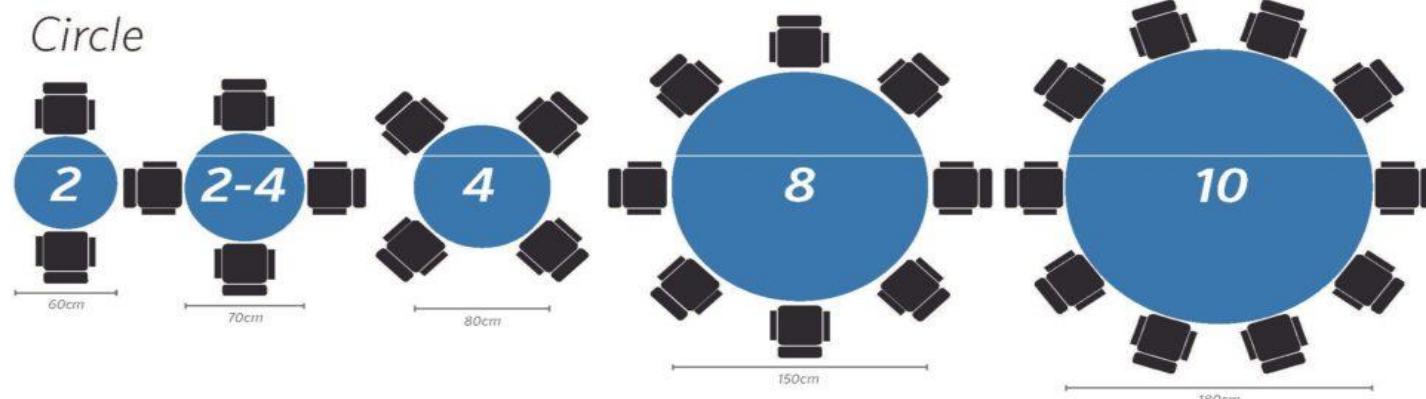
- **Variable size partitioning:** -In this policy, in starting, we treat the memory as a whole or a single chunk & whenever a process request for some space, exactly same space is allocated if possible and the remaining space can be reused again.



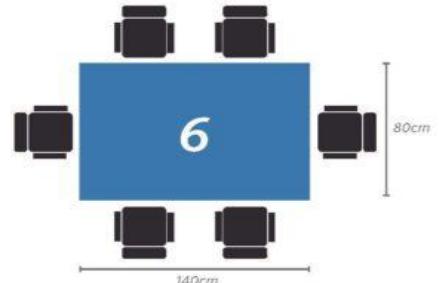
- **Fixed size partitioning**: - here, we divide memory into fixed size partitions, which may be of different sizes, but here if a process request for some space, then a partition is allocated entirely if possible, and the remaining space will be wasted internally.



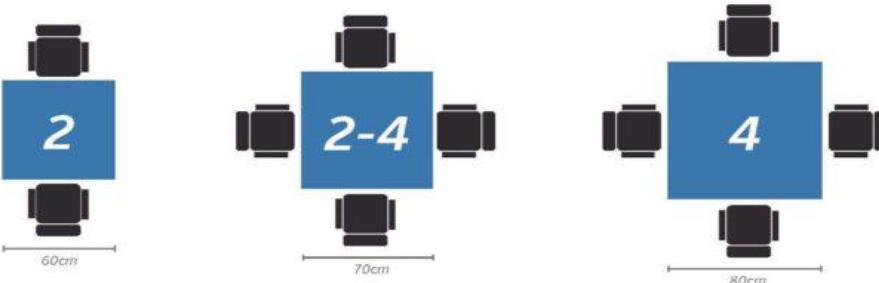
Circle



Rectangle



Square



- **First fit policy**: - as the name implies, it states searching the memory from the base and will allocate first partition which is capable enough.

- **Advantage**: - simple, easy to use, easy to understand
- **Disadvantage**: -poor performance, both in terms of time and space

| |
|--|
| |
|--|

| | | | | |
|--|--|--|--|--|
| | | | | |
|--|--|--|--|--|

- **Best fit policy**: - Here, we search the entire memory and will allocate the smallest partition which is capable enough.
 - **Advantage**: - perform best in fix size partitioning scheme.
 - **Disadvantage**: - difficult to implement, perform worst in variable size partitioning as the remaining spaces which are of very small size.

| | | | | |
|--|--|--|--|--|
| | | | | |
|--|--|--|--|--|

| | | | | |
|--|--|--|--|--|
| | | | | |
|--|--|--|--|--|

- **Worst fit policy**: - it also searches the entire memory and allocate the largest partition possible.
 - **Advantage**: - perform best in variable size partitioning
 - **Disadvantage**: - perform worst in fix size partitioning, resulting into large internal fragmentation.



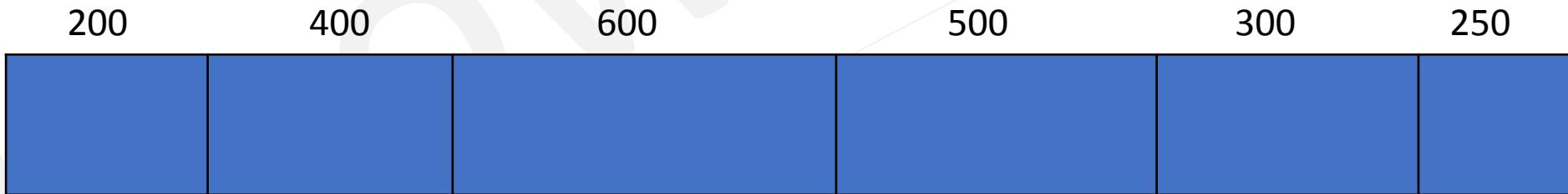
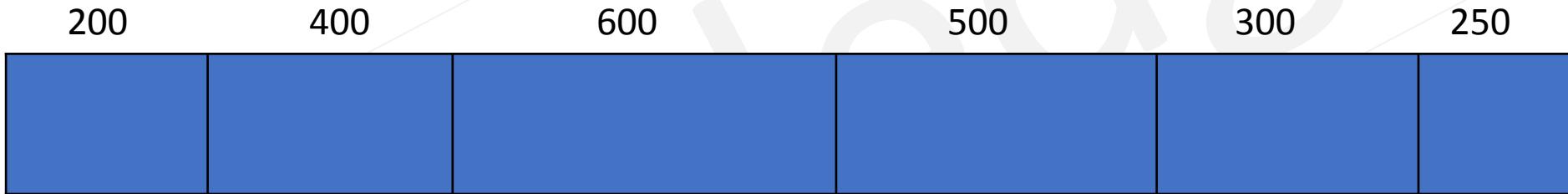
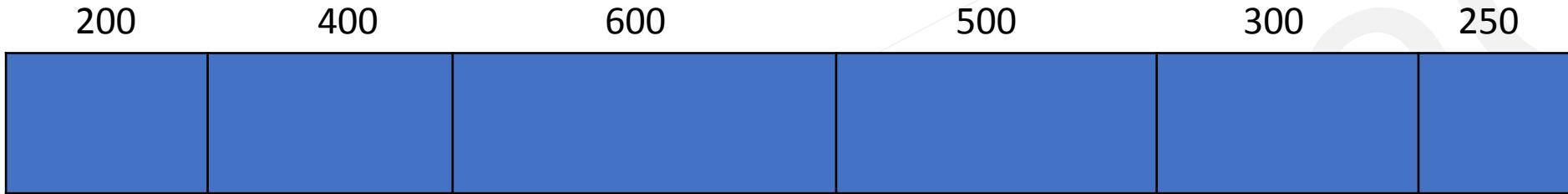
- **Next fit policy:** - next fit is the modification in the best fit where, after satisfying a request, we start satisfying next request from the current position.

| |
|--|
| |
|--|

| | | | | |
|--|--|--|--|--|
| | | | | |
|--|--|--|--|--|

Q Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB, and 250 KB, where KB refers to kilobyte. These partitions need to be allotted to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order. If the best fit algorithm is used, which partitions are NOT allotted to any process? (GATE-2015) (2 Marks)

- (A) 200 KB and 300 KB (B) 200 KB and 250 KB (C) 250 KB and 300 KB (D) 300 KB and 400 KB



Q Consider the following heap (figure) in which blank regions are not in use and hatched region are in use. **(GATE-1994) (2 Marks)**

The sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use

- (a)** either first fit or best fit policy (any one)
- (b)** first fit but not best fit policy
- (c)** best fit but first fit policy
- (d)** None of the above



Q A 1000 Kbyte memory is managed using variable partitions but No compaction. It currently has two partitions of sizes 200 Kbytes and 260 Kbytes respectively. The smallest allocation request in Kbytes that could be denied is for? **(GATE-1996) (1 Marks)**

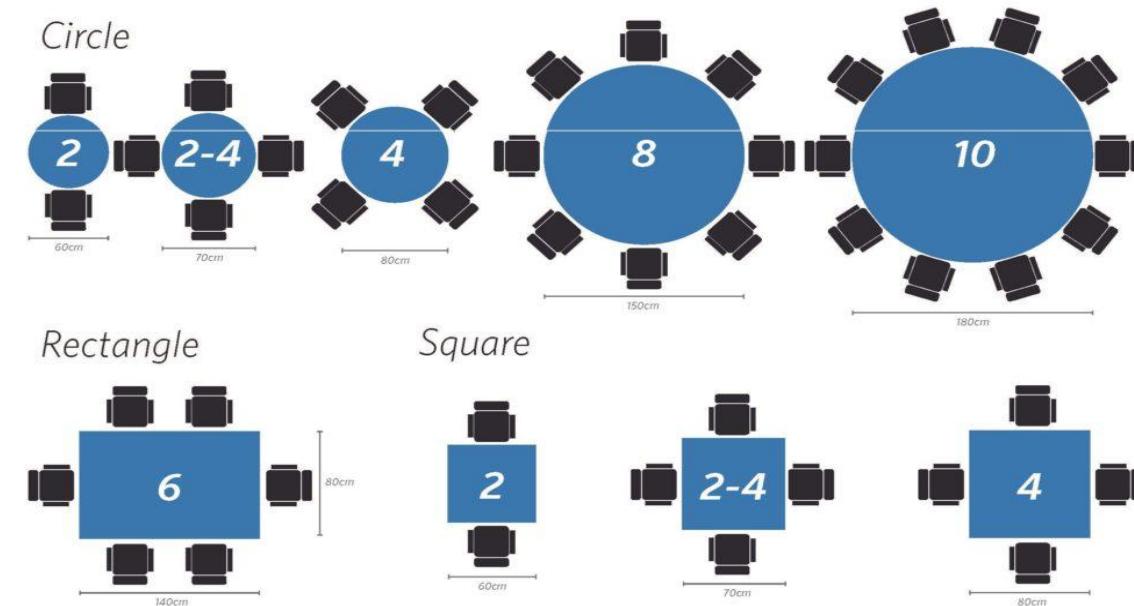
(a) 151

(b) 181

(c) 231

(d) 541

- **External fragmentation:** - External fragmentation is a function of contiguous allocation policy. The space requested by the process is available in memory but, as it is not being contiguous, cannot be allocated this wastage is called external fragmentation.



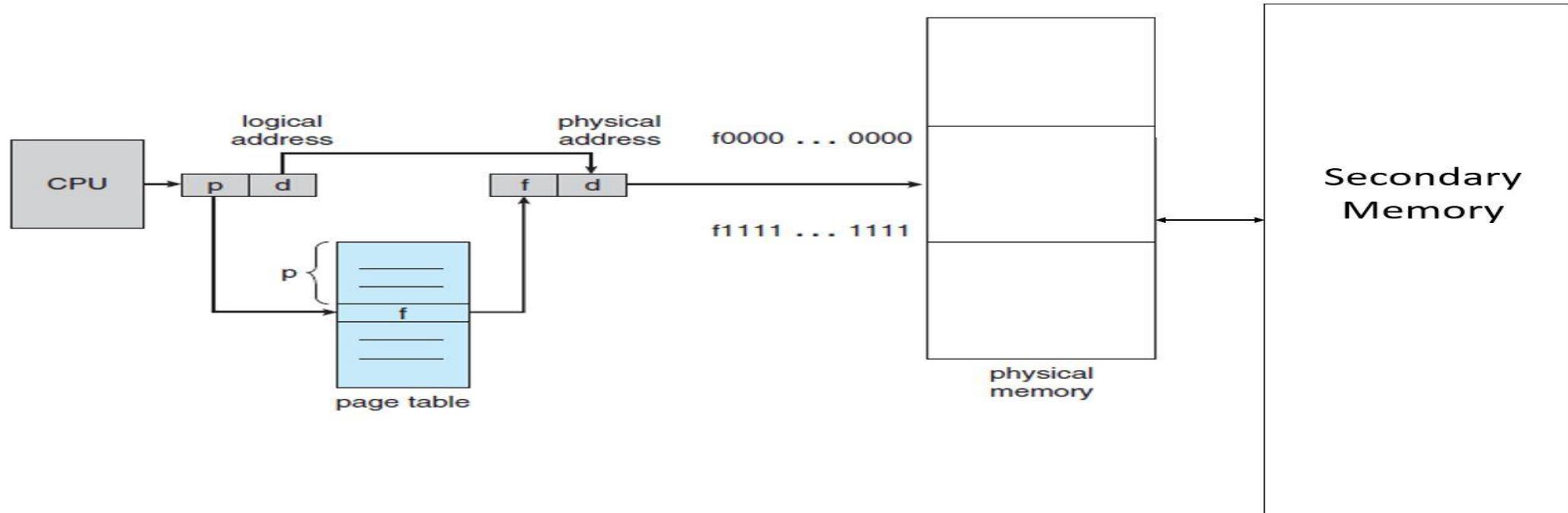
- **Internal fragmentation:** - Internal fragmentation is a function of fixed size partition which means, when a partition is allocated to a process. Which is either the same size or larger than the request then, the unused space by the process in the partition Is called as internal fragmentation



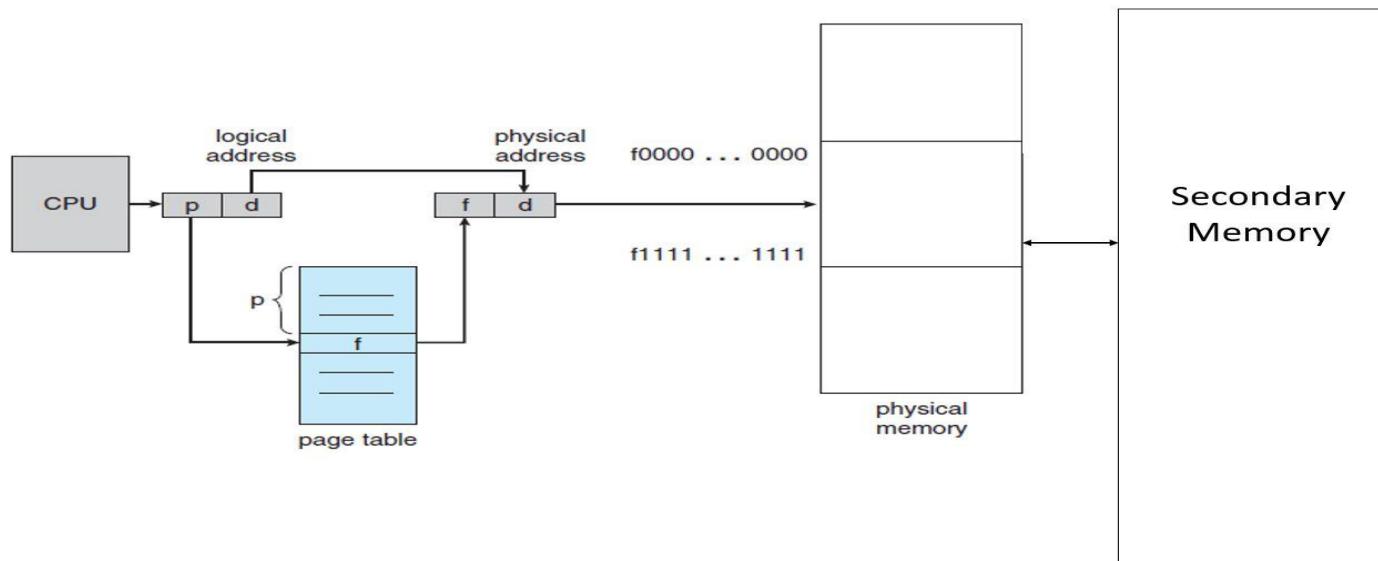
- How can we solve external fragmentation?
- We can also swap processes in the main memory after fixed intervals of time & they can be swapped in one part of the memory and the other part become empty(Compaction, defragmentation). This solution is very costly in respect to time as it will take a lot of time to swap process when system is in running state.
- Either we should go for non-contiguous allocation, which means process can be divided into parts and different parts can be allocated in different areas.

Non-Contiguous Memory allocation(Paging)

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids external fragmentation



- Secondary memory is divides into fixed size partition(because management is easy) all of them of same size called pages(easy swapping and no external fragmentation).
- Main memory is divided into fix size partitions (because management is easy), each of them having same size called frames(easy swapping and no external fragmentation).
- Size of frame = size of page
- In general number of pages are much more than number of frames (approx. 128 time)



Translation process

1. CPU generate a logical address is divided into two parts - **p** and **d**
 1. where p stands for page no and d stands for instruction offset.
2. The **page number(p)** is used as an index into a **Page table**
3. **Page table base register(PTBR)** provides the base of the page table and then the corresponding page no is accessed using p.
4. Here we will finds the corresponding frame no (the base address of that frame in main memory in which the page is stored)
5. Combine corresponding frame no with the instruction offset and get the physical address. Which is used to access main memory.

Page Table

- Page table is a data structure not hardware.
- Every process have a separate page table.
- Number of entries a process have in the page table is the number of pages a process have in the secondary memory.
- Size of each entry in the page table is same it is corresponding frame number.
- Page table is a data structure which is it self stored in main memory.

- **Advantage**
 - Removal of External Fragmentation
- **Disadvantage**
 - Translation process is slow as Main Memory is accessed two times(one for page table and other for actual access).
 - A considerable amount of space is wasted in storing page table(meta data).
 - System suffers from internal fragmentation(as paging is an example of fixed size partition).

Q The essential content(s) in each entry of a page table is / are? **(GATE-2009) (1 Marks)**

(A) Virtual page number

(B) Page frame number

(C) Both virtual page number and page frame number

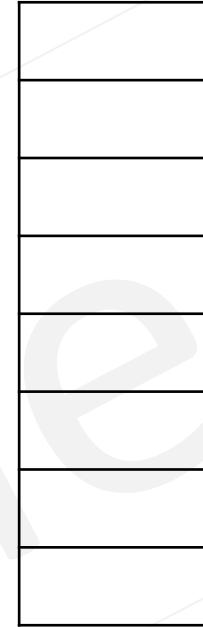
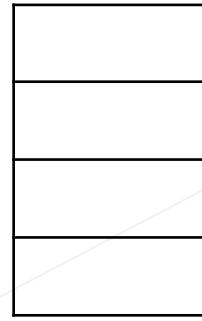
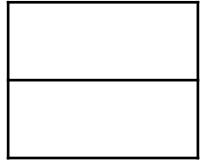
(D) Access right information

Q.24 Which of the following tasks is/are the responsibility/responsibilities of the memory management unit (MMU) in a system with paging-based memory management? (Gate 2024 CS)

- (a) Allocate a new page table for a newly created process
- (b) Translate a virtual address to a physical address using the page table
- (c) Raise a trap when a virtual address is not found in the page table
- (d) Raise a trap when a process tries to write to a page marked with read-only permission in the page table

| | | | | | |
|-----------|------------|-----------|---------|----------|---------|
| 10^3 | 1 Thousand | 10^3 | 1 kilo | 2^{10} | 1 kilo |
| 10^6 | 1 Million | 10^6 | 1 Mega | 2^{20} | 1 Mega |
| 10^9 | 1 Billion | 10^9 | 1 Giga | 2^{30} | 1 Giga |
| 10^{12} | 1 Trillion | 10^{12} | 1 Tera | 2^{40} | 1 Tera |
| | | 10^{15} | 1 Peta | 2^{50} | 1 Peta |
| | | 10^{18} | 1 Exa | 2^{60} | 1 Exa |
| | | 10^{21} | 1 Zetta | 2^{70} | 1 Zetta |
| | | 10^{24} | 1 Yotta | 2^{80} | 1 Yotta |

| | |
|-------------------------------|-------|
| Address Length in bits | n |
| No of Locations | 2^n |



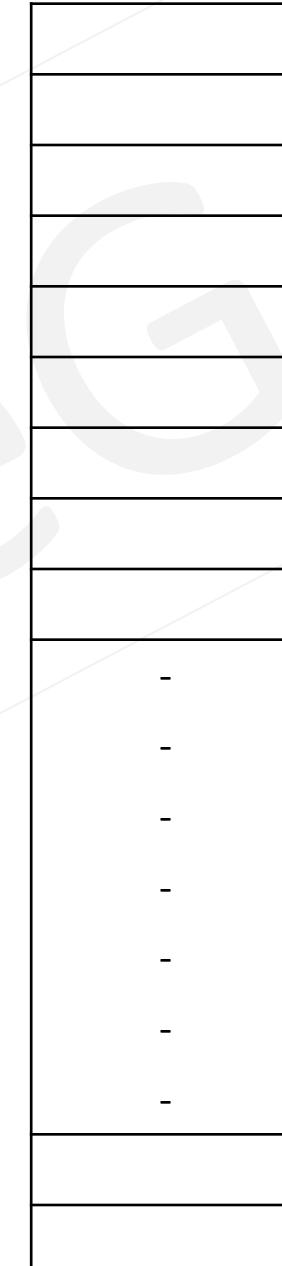
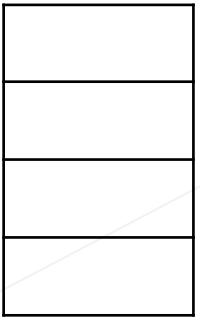
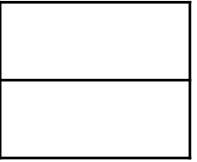
Memory Size = Number of Locations * Size of each Location
www.knowledgemantra.in

Address Length in bits

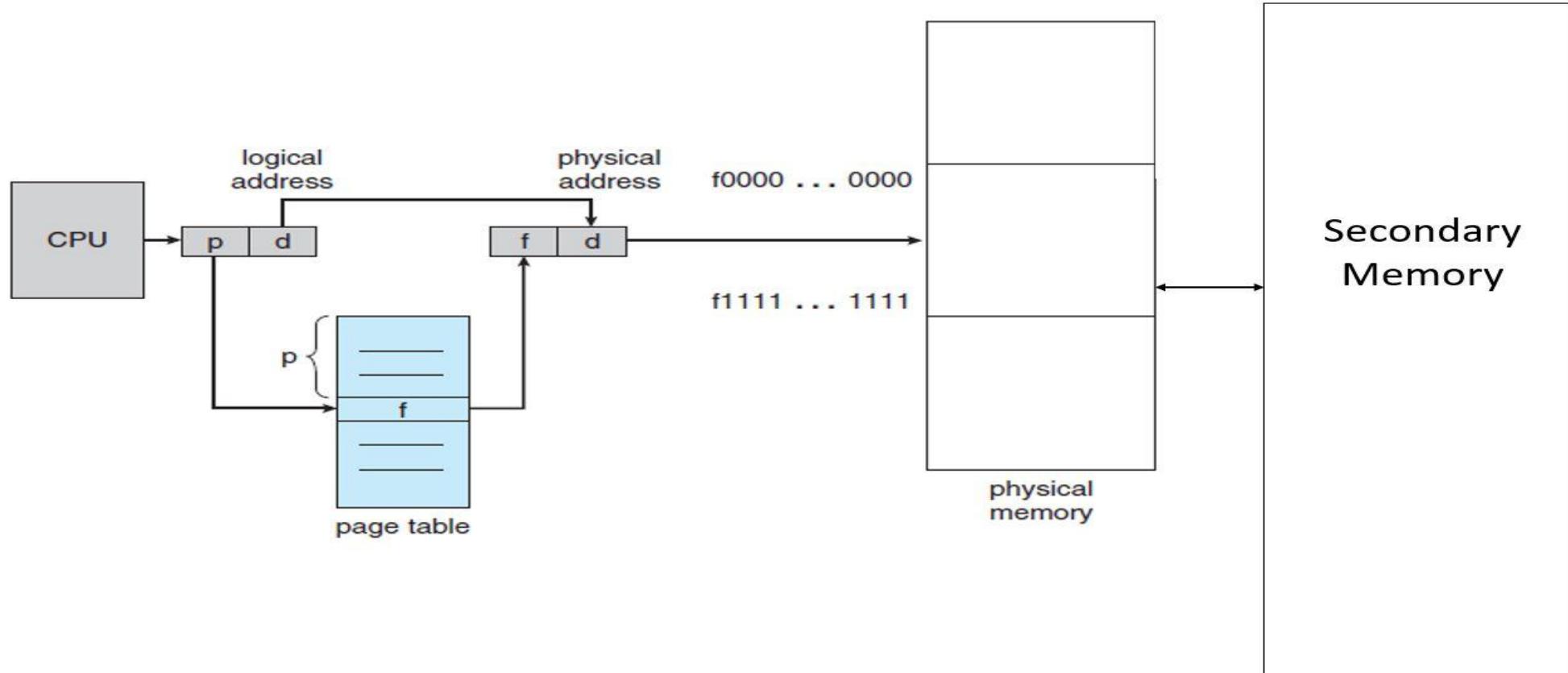
Upper Bound($\log_2 n$)

No of Locations

n



- Page Table Size = No of entries in Page table * Size of each entry(f)
- Process Size = No of Pages * Size of each page



| S No | SM | LA | MM | PA | p | f | d | addressable | Page Size |
|------|-------|----|--------|----|----|----|----|-------------|-----------|
| 1 | 32 GB | | 128 MB | | | | | 1B | 1KB |
| 2 | | 42 | | 33 | | | 11 | 1B | |
| 3 | 512GB | | | 31 | | | | 1B | 512B |
| 4 | 128GB | | 32GB | | 30 | | | 1B | |
| 5 | | | | | 28 | 14 | | | 4096B |

p

d

f

d

Secondary Memory

Main Memory

Q Consider a system with byte-addressable memory, 32-bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is _____ (GATE-2015) (2 Marks)

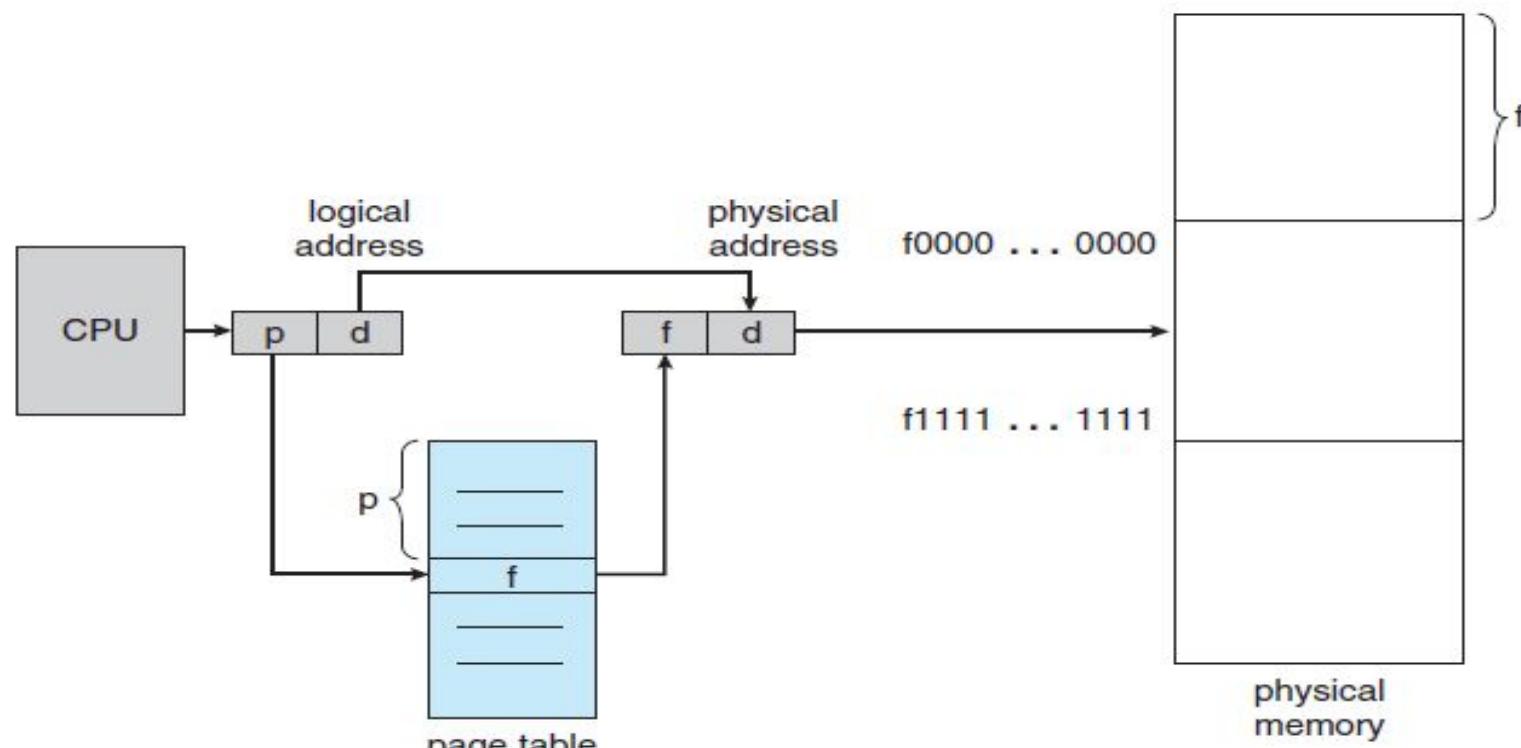


Figure 8.10 Paging hardware.

Q A Computer system implements 8 kilobyte pages and a 32-bit physical address space. Each page table entry contains a valid bit, a dirty bit three permission bits, and the translation. If the maximum size of the page table of a process is 24 megabytes, the length of the virtual address supported by the system is _____ bits? **(GATE-2015) (2 Marks)**

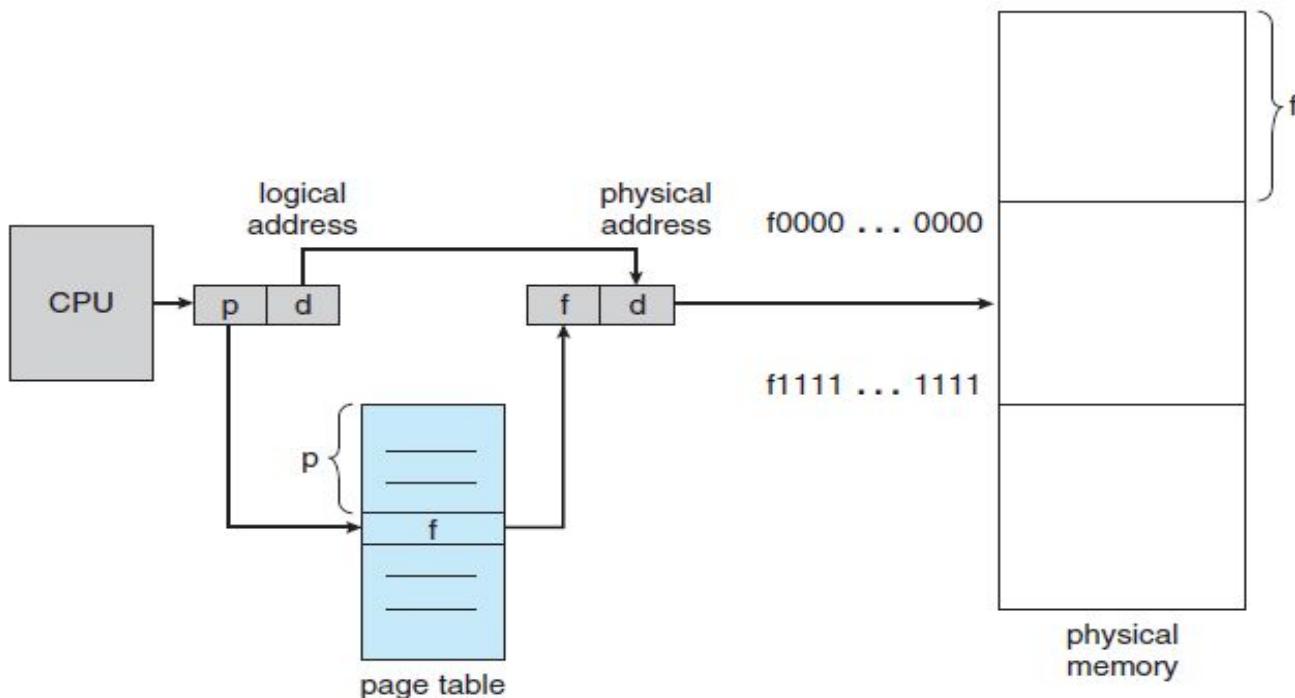


Figure 8.10 Paging hardware.

Q Consider a machine with 64 MB physical memory and a 32-bit virtual address space. If the page size is 4KB, what is the approximate size of the page table? **(GATE-2001) (2 Mark)**

(a) 16 MB

(b) 8 MB

(c) 2 MB

(d) 24 MB

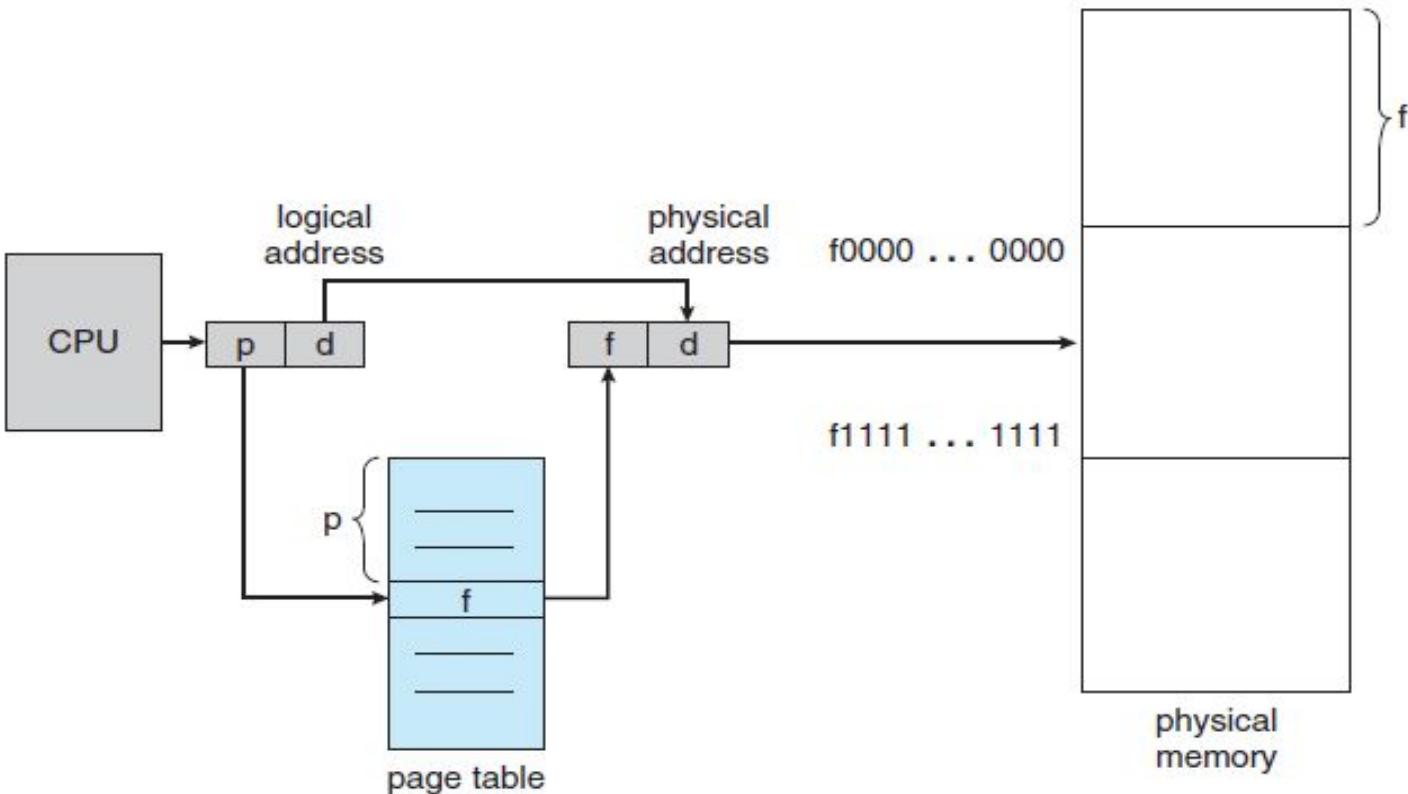
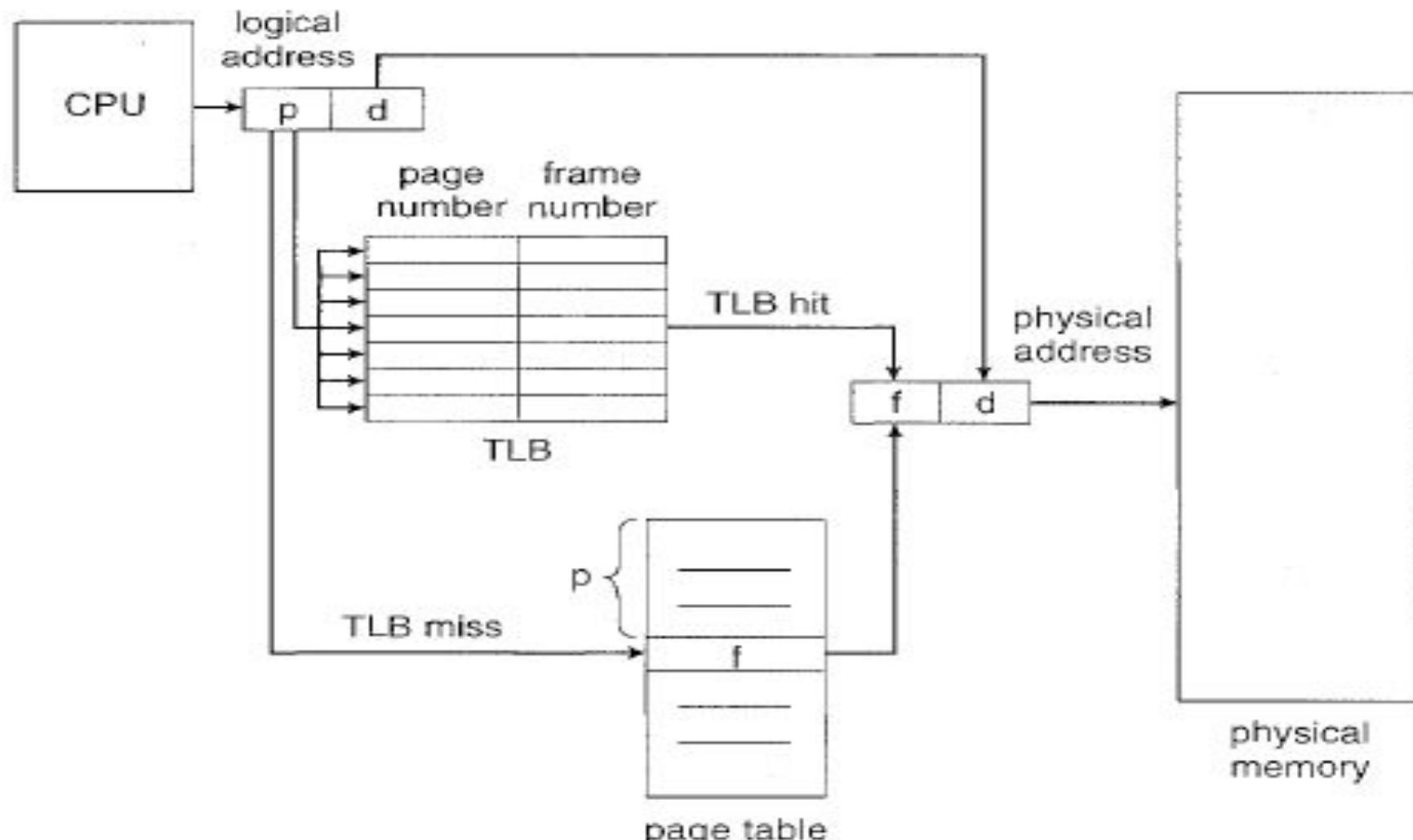


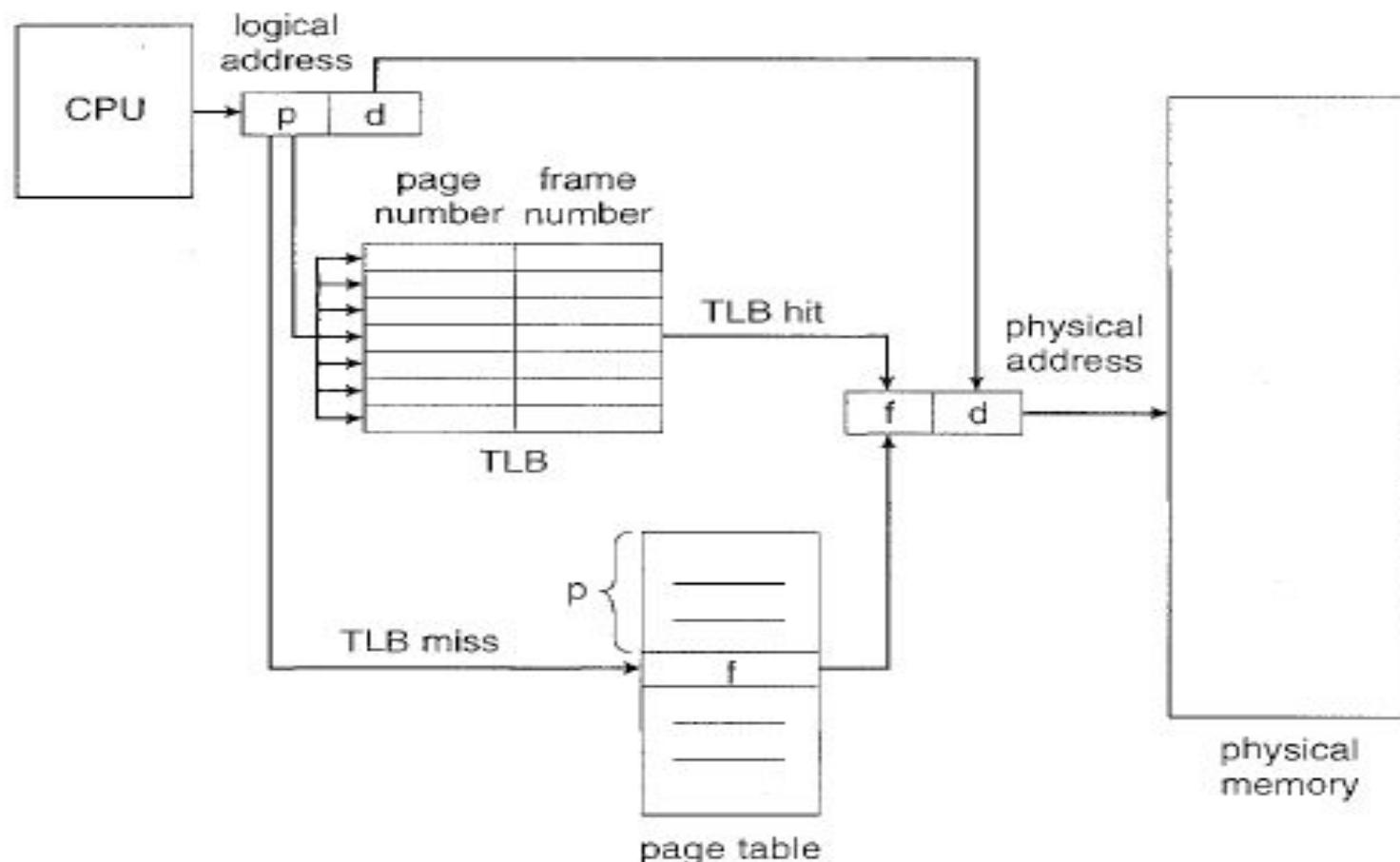
Figure 8.10 Paging hardware.

- A serious problem with page is, translation process is slow as page table is accessed two times (one for page table and other for actual access). To solve the problems in paging we take the help of TLB. The TLB is associative, high-speed memory.



- **TLB Structure:** Each entry in the TLB consists of a key (Page Number) and a value (Frame Number). The TLB operates as associative memory, allowing for simultaneous comparison of a searched page number against all stored page numbers.
- **Operation:** If a page number is found in the TLB, the corresponding frame number is quickly returned, facilitating fast memory access. This process is due to the hardware design, which is efficient but costly.
- **Integration with Page Tables:** The TLB stores only a select few page-table entries. When a logical address is generated, the CPU presents the page number to the TLB. If the page is found in the TLB, the frame number is used immediately for memory access. If not found (a TLB miss), the system must reference the page table.
- **Updating the TLB:** New page numbers and their corresponding frame numbers are added to the TLB for faster access in future references.
- **Management:** When the TLB is full, the operating system must employ page replacement policies to manage its entries.
- **Hit Ratio:** This is the percentage of times that a page number is found in the TLB, indicative of the TLB's effectiveness.

- **Effective Memory Access Time:**
 - Hit [TLB + Main Memory] + 1-Hit [TLB + 2 Main Memory]
- TLB removes the problem of slow access.

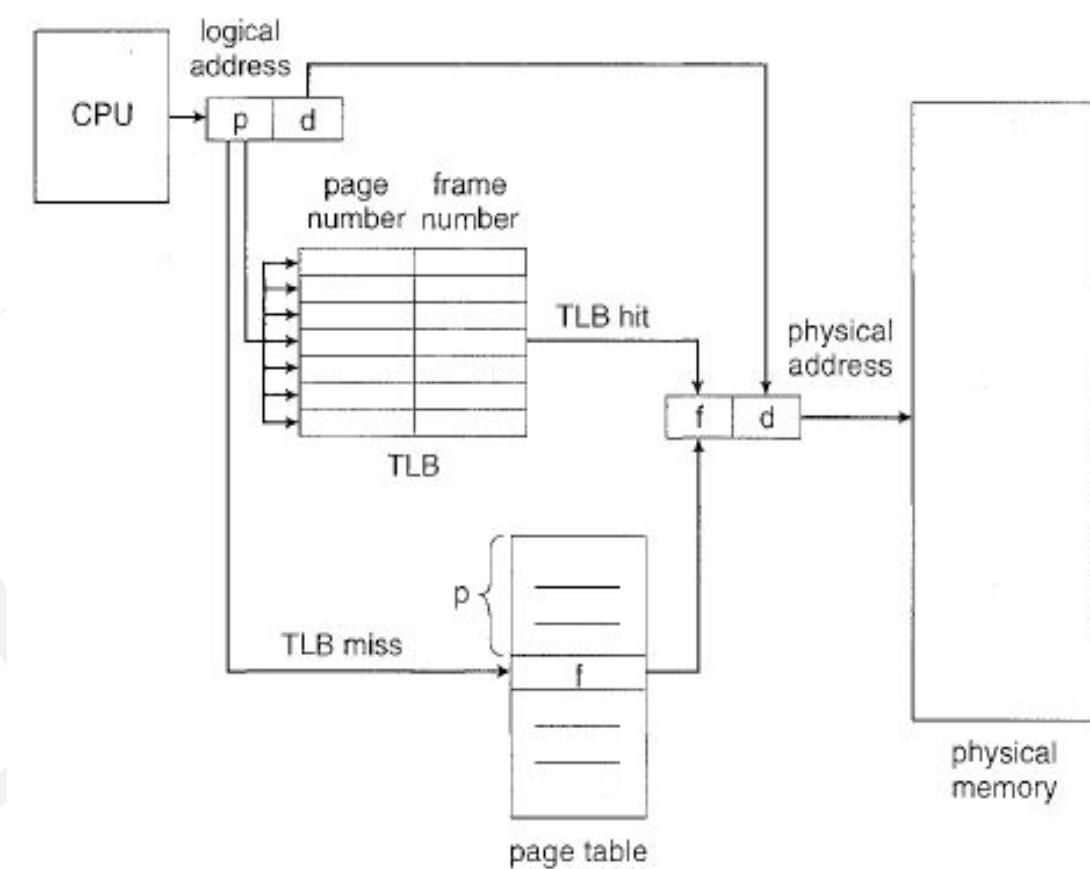


- **Disadvantage of TLB:**

- TLB can hold the data of one process at a time and in case of multiple context switches TLB will be required to flush frequently.

- **Solution:**

- Use multiple TLB's but it will be costly.
- Some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.



Q Which one of the following statements is FALSE? (GATE 2022) (2 MARKS)

- (A)** The TLB performs an associative search in parallel on all its valid entries using page number of incoming virtual address.
- (B)** If the virtual address of a word given by CPU has a TLB hit, but the subsequent search for the word results in a cache miss, then the word will always be present in the main memory.
- (C)** The memory access time using a given inverted page table is always same for all incoming virtual addresses.
- (D)** In a system that uses hashed page tables, if two distinct virtual addresses V1 and V2 map to the same value while hashing, then the memory access time of these addresses will not be the same.

Q Assume that in a certain computer, the virtual addresses are 64 bits long and the physical addresses are 48 bits long. The memory is word addressable. The page size is 8kB and the word size is 4 bytes. The Translation Look-aside Buffer (TLB) in the address translation path has 128 valid entries. At most how many distinct virtual addresses can be translated without any TLB miss? **(GATE-2019) (2 Marks)**

(A) 16×2^{10}

(B) 8×2^{20}

(C) 4×2^{20}

(D) 256×2^{10}

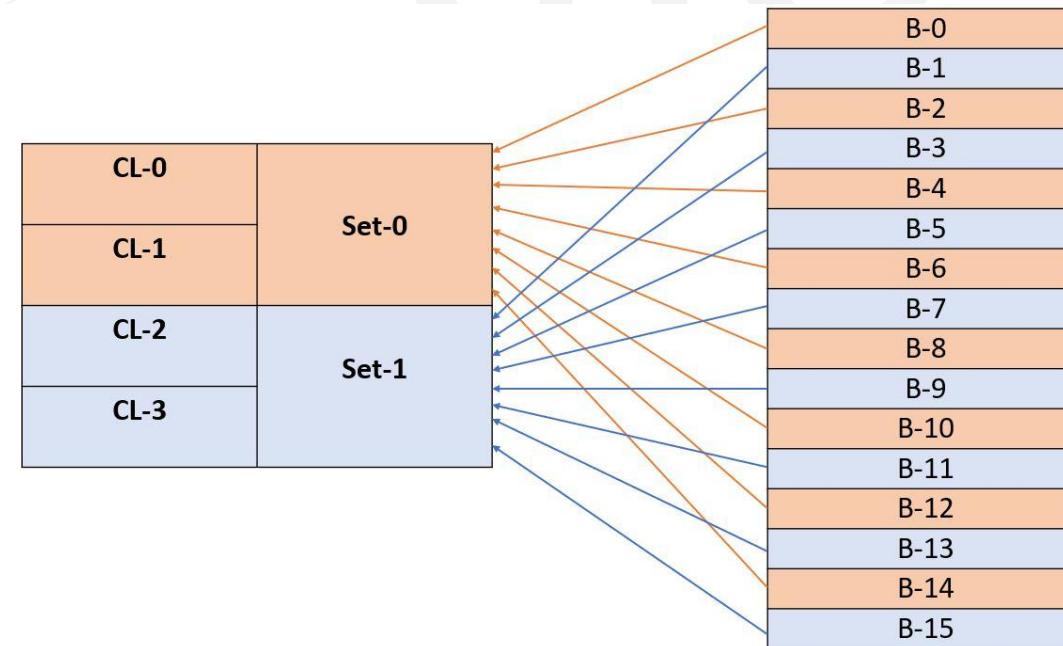
Q A computer system implements a 40-bit virtual address, page size of 8 kilobytes, and a 128-entry translation look-aside buffer (TLB) organized into 32 sets each having four ways. Assume that the TLB tag does not store any process id. The minimum length of the TLB tag in bits is _____ (GATE-2015) (2 Marks)

(A) 20

(B) 10

(C) 11

(D) 22



Q Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, the effective memory access time (in milliseconds) is _____. **(CS-2014) (2 Marks)**

Q A paging system tlb takes 10ns main memory takes 50ns what is effective memory access time if tlb hit ratio is 90%? **(GATE-2008) (1 Marks)**

a) 54

b) 60

c) 65

d) 75

Q A CPU generates 32-bit virtual addresses. The page size is 4 KB. The processor has a translation look-aside buffer (TLB) which can hold a total of 128-page table entries and is 4-way set associative. The minimum size of the TLB tag is?

(GATE-2006) (2 Marks)

- (A) 11 bits**
- (B) 13 bits**
- (C) 15 bits**
- (D) 20 bits**

Q Which of the following is not a form of memory? (GATE-2002) (1 Marks)

(A) instruction cache

(B) instruction register

(C) instruction opcode

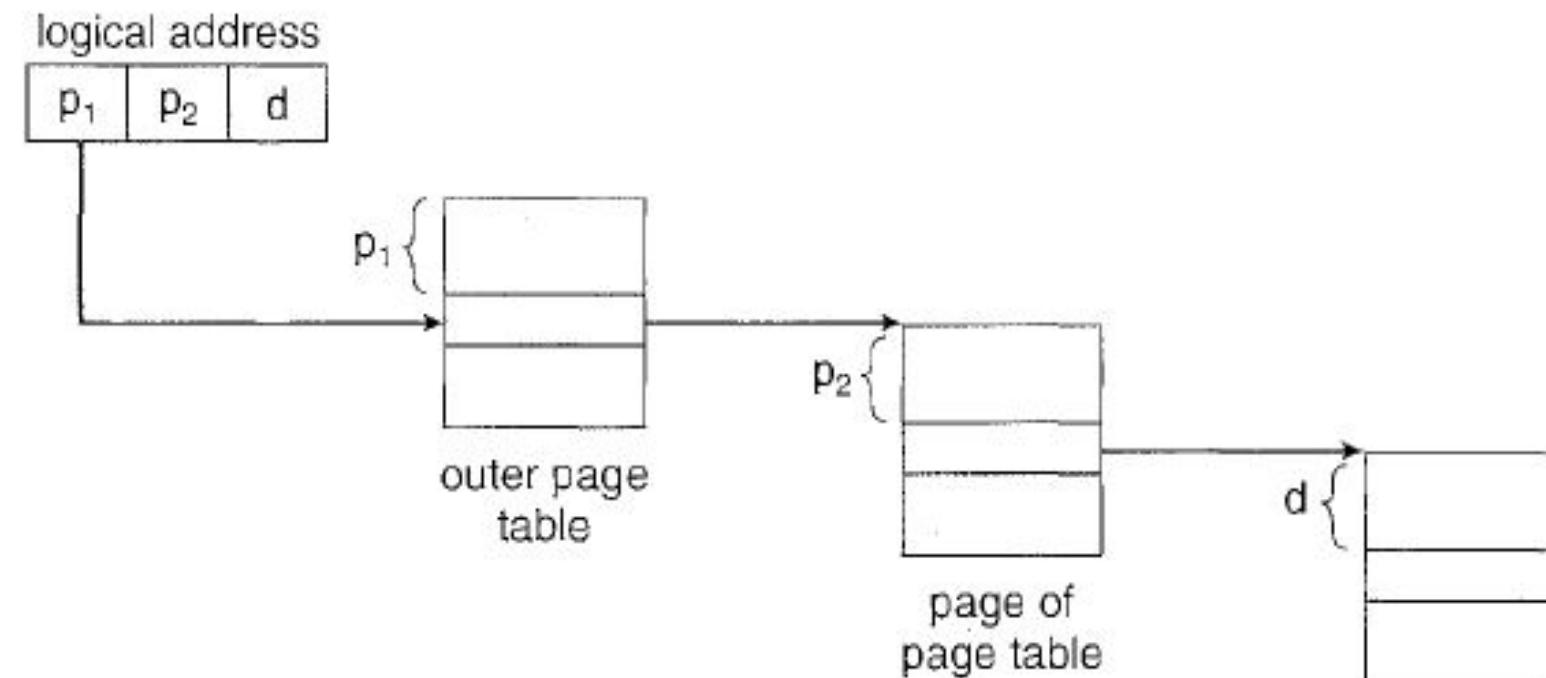
(D) translation lookaside buffer

Size of Page

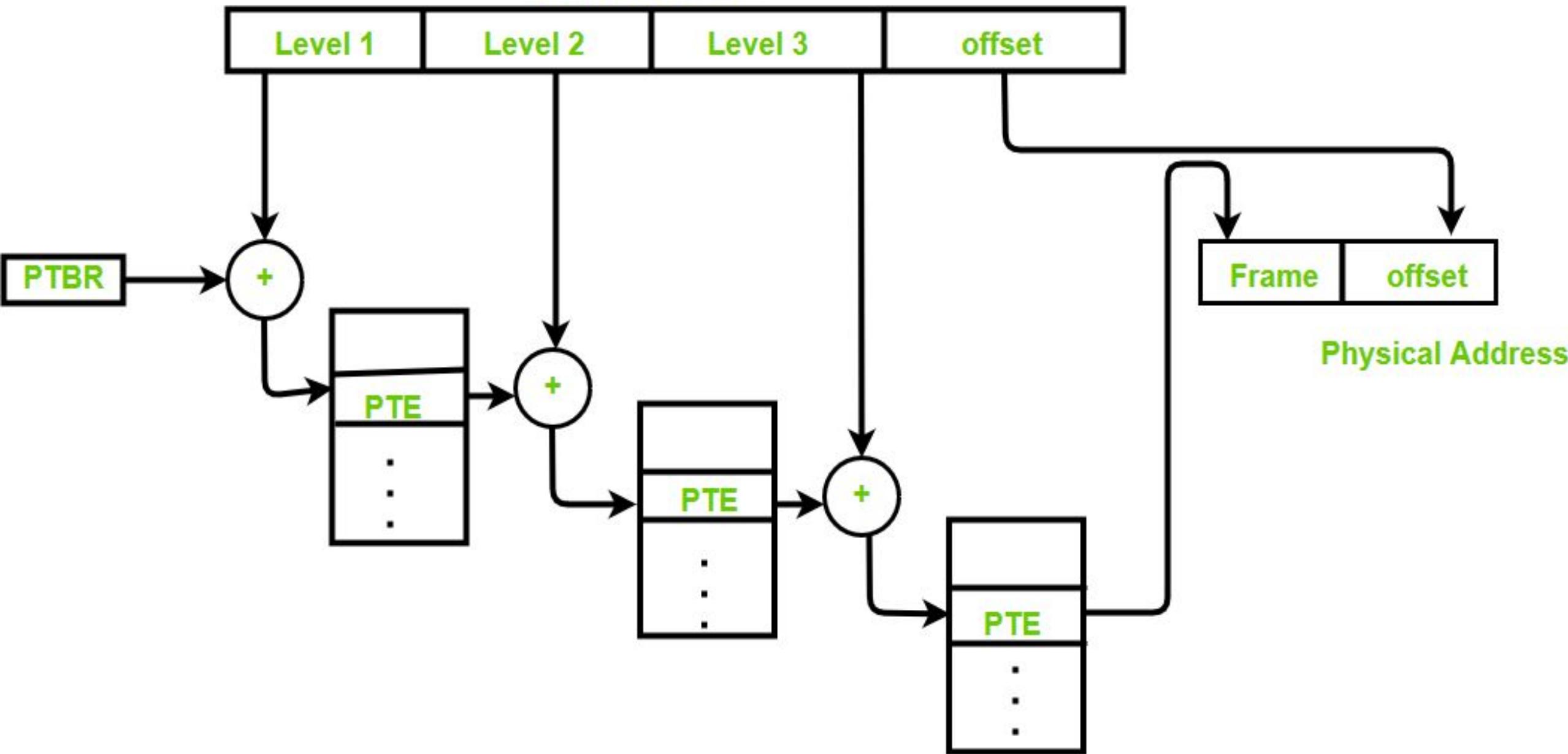
- If we increase the size of page table then internal fragmentation increase but size of page table decreases.
- If we decrease the size of page then internal fragmentation decrease but size of page table increases.
- So we have to find what should be the size of the page, where both cost are minimal.

Multilevel Paging / Hierarchical Paging

- Modern systems support a large logical address space (2^{32} to 2^{64}).
- In such cases, the page table itself becomes excessively large and can contain millions of entries and can take a lot of space in memory, so cannot be accommodated into a single frame.
- A simple solution to this is to divide page table into smaller pieces. One way is to use a **two-level paging algorithm**, in which the page table itself is also paged.



Virtual Address



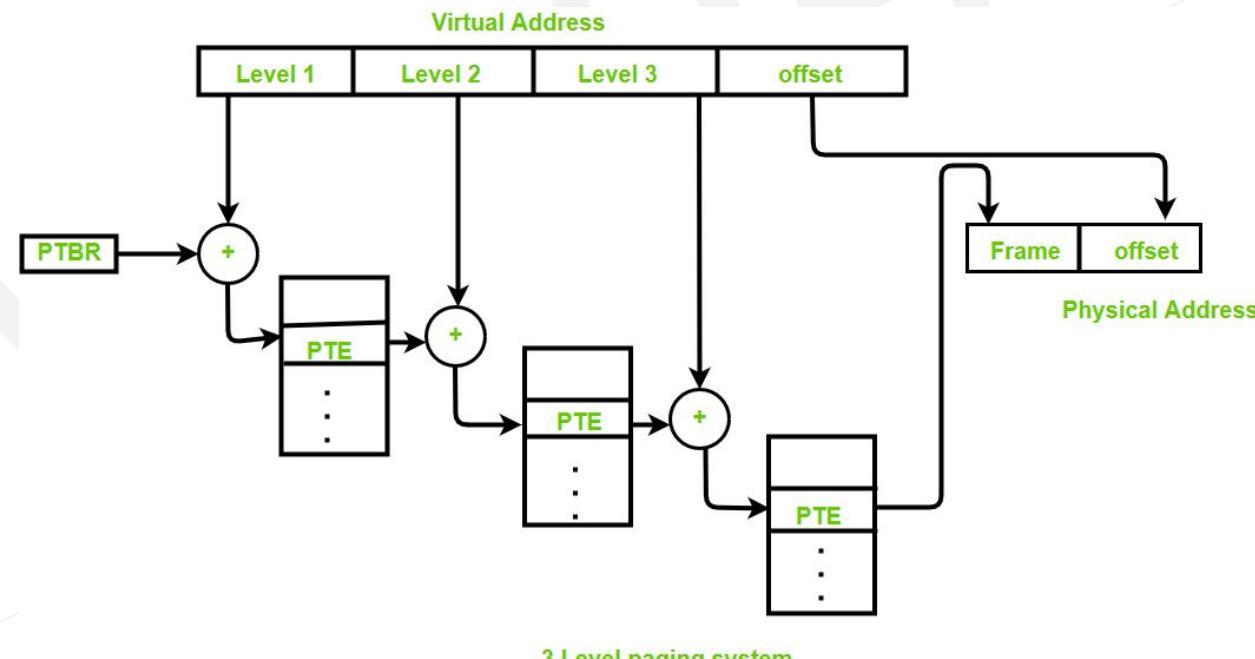
3 Level paging system

Q A multilevel page table is preferred in comparison to a single level page table for translating virtual address to physical address because **(GATE-2009) (1 Marks)**

- (A)** It reduces the memory access time to read or write a memory location.
- (B)** It helps to reduce the size of page table needed to implement the virtual address space of a process.
- (C)** It is required by the translation lookaside buffer.
- (D)** It helps to reduce the number of page faults in page replacement algorithms.

Q A processor uses 36-bit physical addresses and 32-bit virtual addresses, with a page frame size of 4 Kbytes. Each page table entry is of size 4 bytes. A three level page table is used for virtual to physical address translation, where the virtual address is used as follows • Bits 30-31 are used to index into the first level page table • Bits 21-29 are used to index into the second level page table • Bits 12-20 are used to index into the third level page table, and • Bits 0-11 are used as offset within the page. The number of bits required for addressing the next level page table (or page frame) in the page table entry of the first, second and third level page tables are respectively (GATE-2008) (2 Marks)

- (A) 20, 20 and 20 (B) 24, 24 and 24 (C) 24, 24 and 20 (D) 25, 25 and 24



Q In a system with 32 bit virtual addresses and 1 KB page size, use of one-level page tables for virtual to physical address translation is not practical because of
(GATE-2003) (1 Marks)

- (A) the large amount of internal fragmentation
- (B) the large amount of external fragmentation
- (C) the large memory overhead in maintaining page tables
- (D) the large computation overhead in the translation process

Q A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE). The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

What is the size of a page in KB in this computer? **(GATE-2013) (2 Marks)**

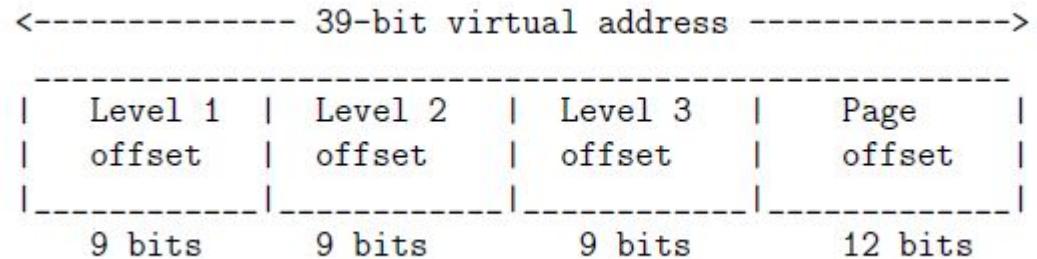
- (A) 2
- (B) 4
- (C) 8
- (D) 16

Q Consider a computer system with 40-bit virtual addressing and page size of sixteen kilobytes. If the computer system has a one-level page table per process and each page table entry requires 48 bits, then the size of the per-process page table is _____ megabytes. **(GATE-2016) (2 Marks)**

Q In the context of operating systems, which of the following statements is/are correct with respect to paging? **(GATE 2021)**

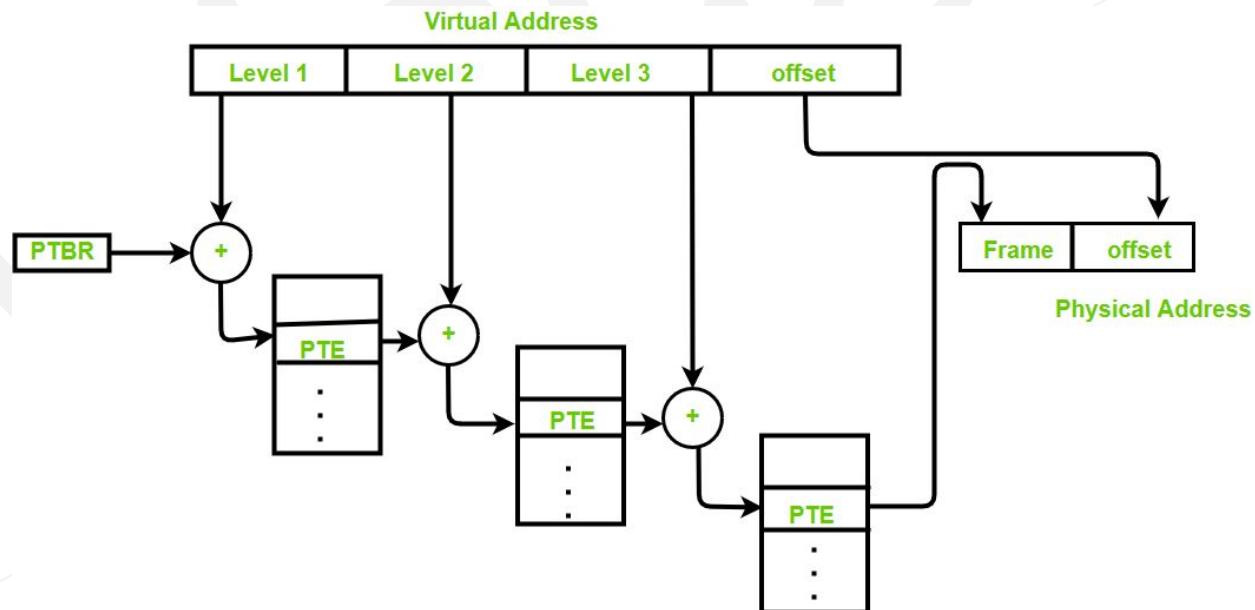
- (a)** Paging helps solve the issue of external fragmentation
- (b)** Page size has no impact on internal fragmentation
- (c)** Paging incurs memory overheads
- (d)** Multi-level paging is necessary to support pages of different sizes

Q Consider a three-level page table to translate a 39-bit virtual address to a physical address as shown below:



The page size is 4 KB = (1KB = 2^{10} bytes) and page table entry size at every level is 8 bytes. A process P is currently using 2 GB (1 GB = 2^{30} bytes) virtual memory which OS mapped to 2 GB of physical memory. The minimum amount of memory required for the page table of P across all levels is _____ KB (GATE 2021) (2 MARKS)

- (a) 4108
 - (b) 1027
 - (c) 3081
 - (d) 4698



Q. Consider a memory management system that uses a page size of 2 KB. Assume that both the physical and virtual addresses start from 0. Assume that the pages 0,1,2, and 3 are stored in the page frames 1,3,2, and 0 respectively. The Physical address (in decimal format) corresponding to the virtual address 2500 (in decimal format) is _____(Gate 2024,CS) (2 Marks) (NAT)

Q. Consider a 32-bit system with 4 KB page size and page table entries of size 4 bytes each. Assume 1 KB = 2^{10} bytes.(Gate 2024 CS)

The OS uses a 2-level page table for memory management, with the page table containing an outer page directory and an inner page table. The OS allocates a page for the outer page directory upon process creation. The OS uses demand paging when allocating memory for the inner page table, i.e., a page of the inner page table is allocated only if it contains at least one valid page table entry.

An active process in this system accesses 2000 unique pages during its execution, and none of the pages are swapped out to disk. After it completes the page accesses, let X denote the minimum and Y denote the maximum number of pages across the two levels of the page table of the process.

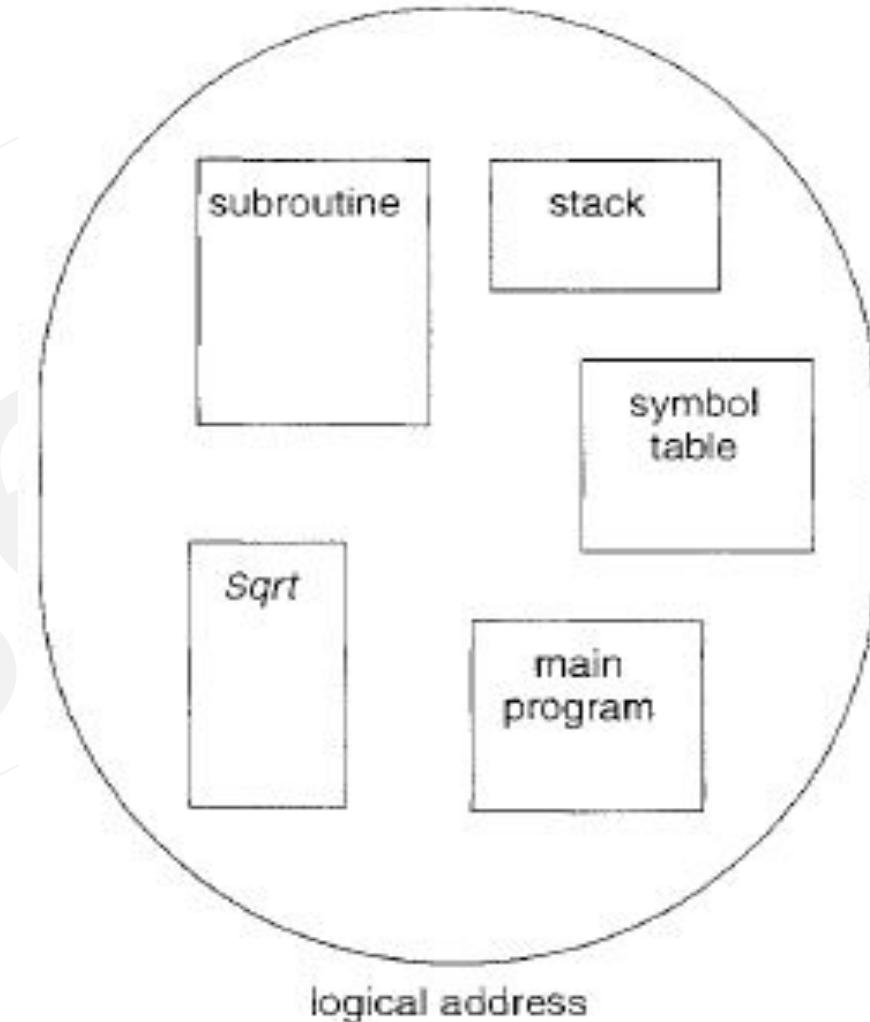
The value of $X+Y$ is _____

Q. A computer system supports a logical address space of 2^{32} bytes. It uses two-level hierarchical paging with a page size of 4096 bytes. A logical address is divided into a b -bit index to the outer page table, an offset within the page of the inner page table, and an offset within the desired page. Each entry of the inner page table uses eight bytes. All the pages in the system have the same size.

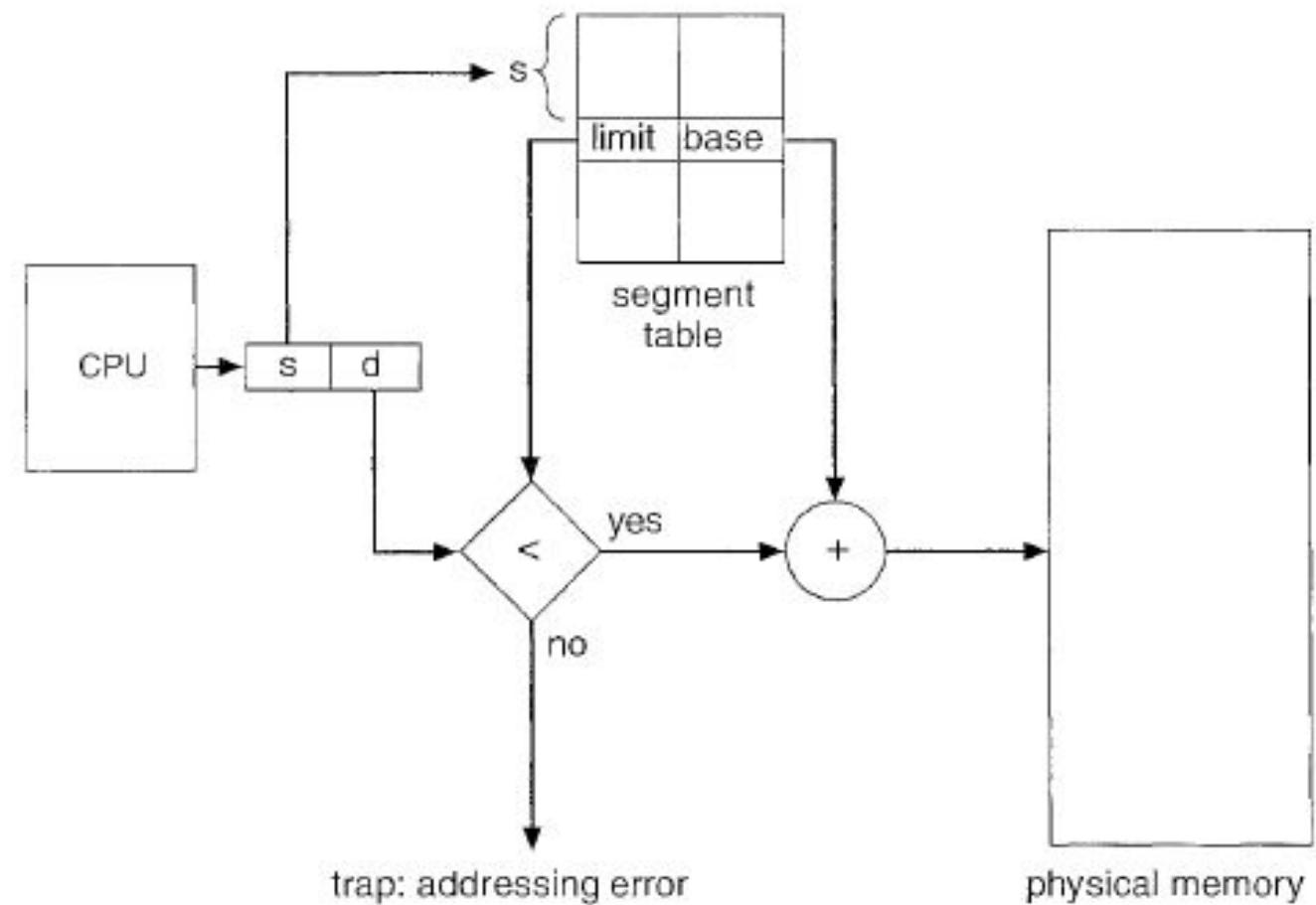
The value of b is _____ . (Answer in integer) **(Gate 2025)**

Segmentation

- Paging is unable to separate the user's view of memory from the actual physical memory.
- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: **a segment name and an offset**.
- Thus, a logical address consists of a two tuple:
- **<segment-number, offset>**.
- Segments can be of variable lengths unlike pages and are stored in main memory.

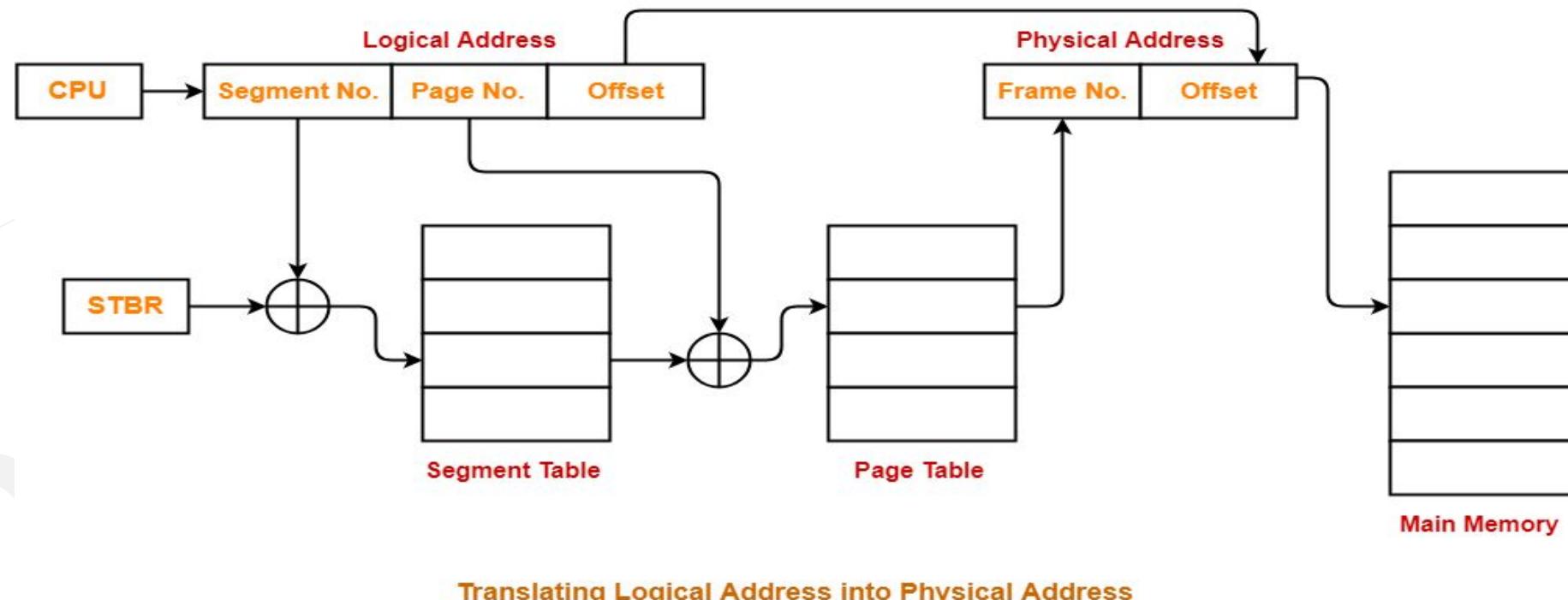


- **Segment Table:** Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
- The segment number is used as an index to the segment table.
- The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system.
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- Segmentation suffers from **External Fragmentation**.



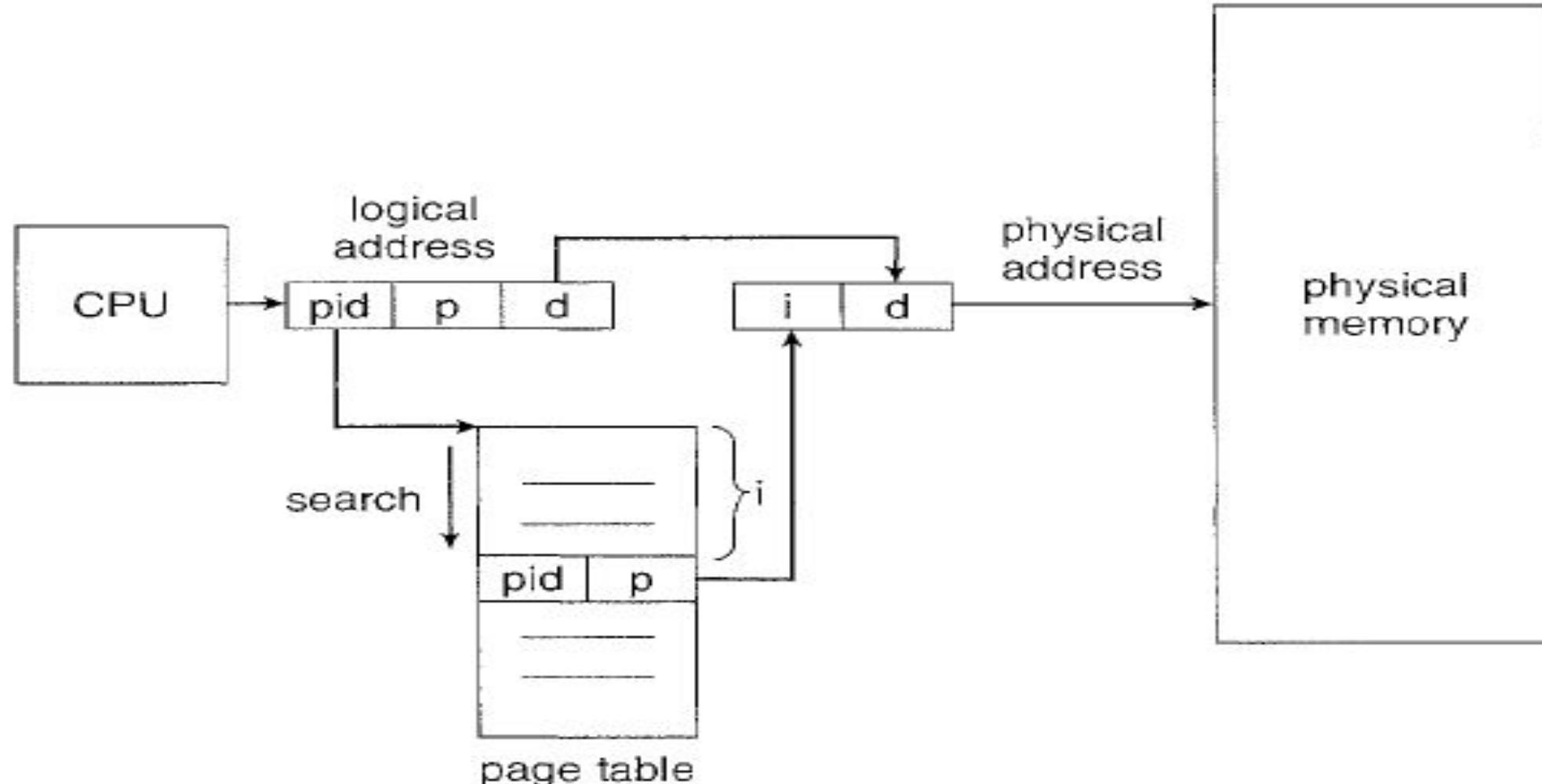
Segmentation with Paging

- Since segmentation also suffers from external fragmentation, it is better to divide the segments into pages as the segments size increases.
- In segmentation with paging scheme a process is divided into segments and further the segments are divided into pages.
- One can argue it is segmentation with paging is quite similar to multilevel paging, but actually it is better, because here when page table is divided the size of partition can be different (as actually the size of different chapters can be different).



- All properties of segmentation with ~~paging is~~ ~~is~~ ~~the same as~~ multilevel paging.

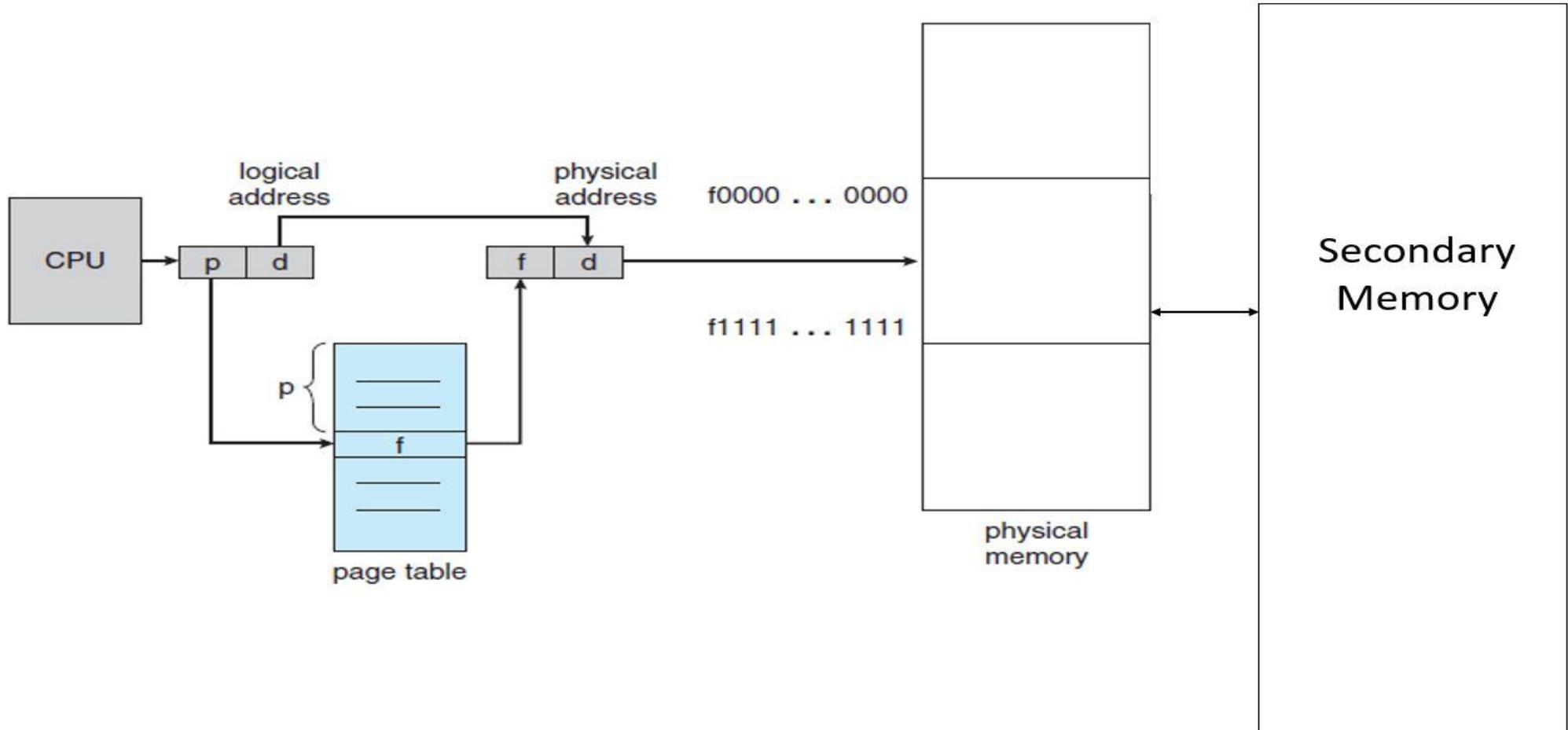
- The drawback of previous methods is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.



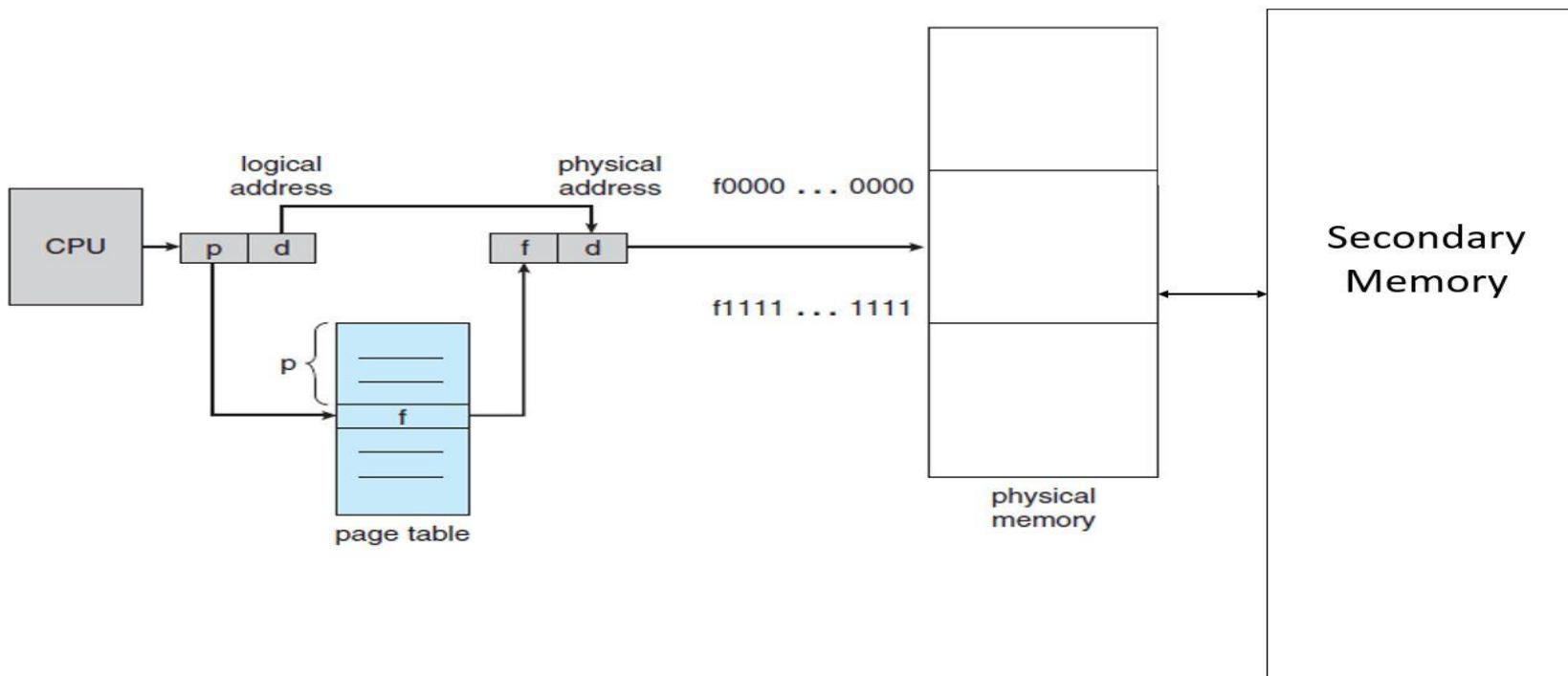
- To solve this problem, we can use an **Inverted Page Table**.
- An inverted page table has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
- Thus number of entries in the page table is equal to the number of frames in the physical memory.

Virtual Memory

- One important goal in now-a-days computing environment is to keep many processes in memory simultaneously to follow multiprogramming, and use resources efficiently especially main memory.



- **Pure Demand Paging/Demand Paging:** We can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that it can execute with no more faults. This scheme is **Pure Demand Paging:** never bring a page into memory until it is required.



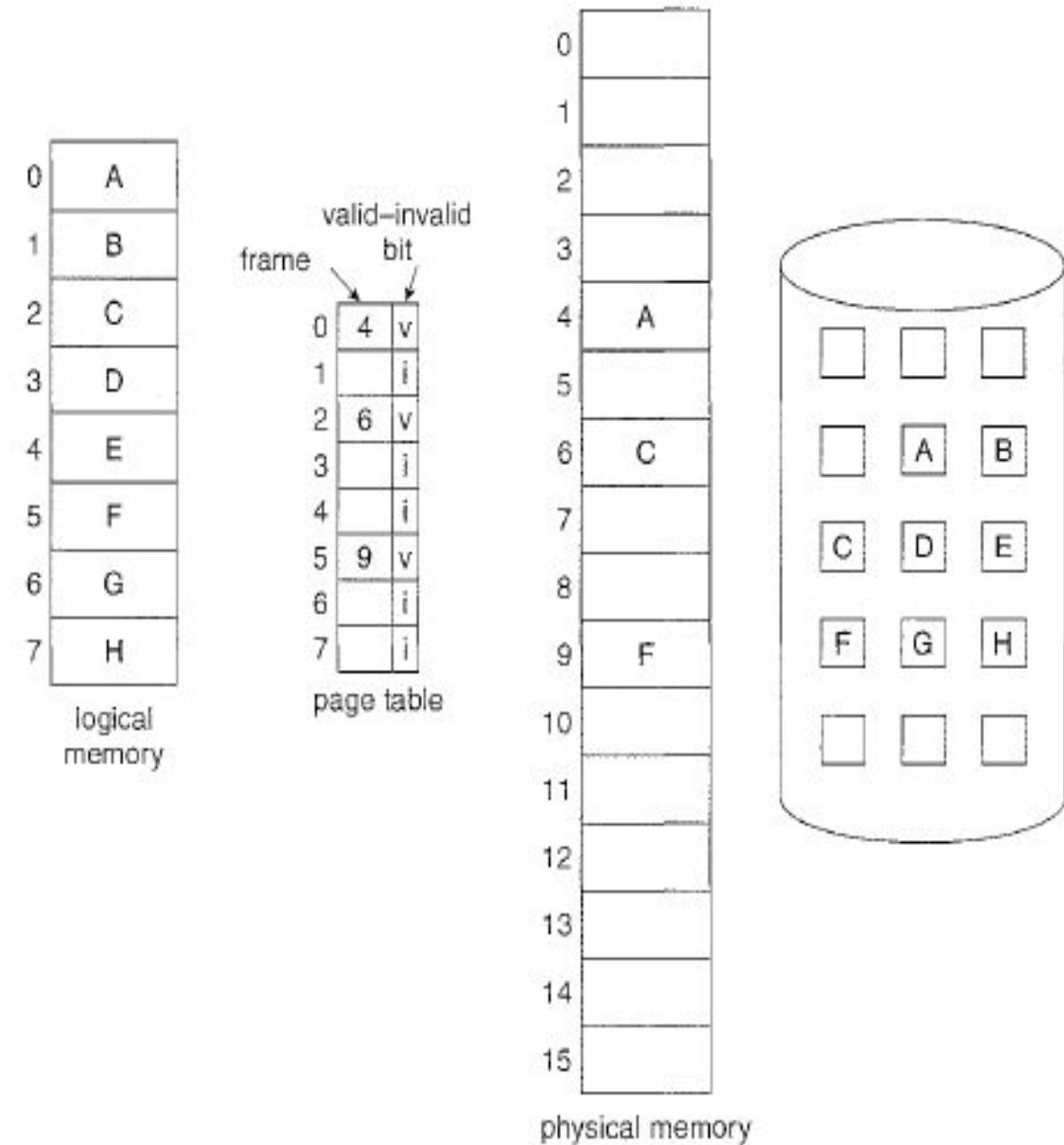
- **Advantage**
 - A program would no longer be constrained by the amount of physical memory that is available, Allows the execution of processes that are not completely in main memory, i.e. process can be larger than main memory.
 - More programs could be run at the same time as use of main memory is less. Virtual memory also allows processes to share files easily and to implement shared memory.
- **Disadvantages**
 - Virtual memory is not easy to implement.
 - It may substantially decrease performance if it is used carelessly (Thrashing)

Q Which of the following statements is false? (GATE-2001) (1 Marks)

- (A) Virtual memory implements the translation of a program's address space into physical memory address space
- (B) Virtual memory allows each program to exceed the size of the primary memory
- (C) Virtual memory increases the degree of multiprogramming
- (D) Virtual memory reduces the context switching overhead

Valid -invalid bit scheme

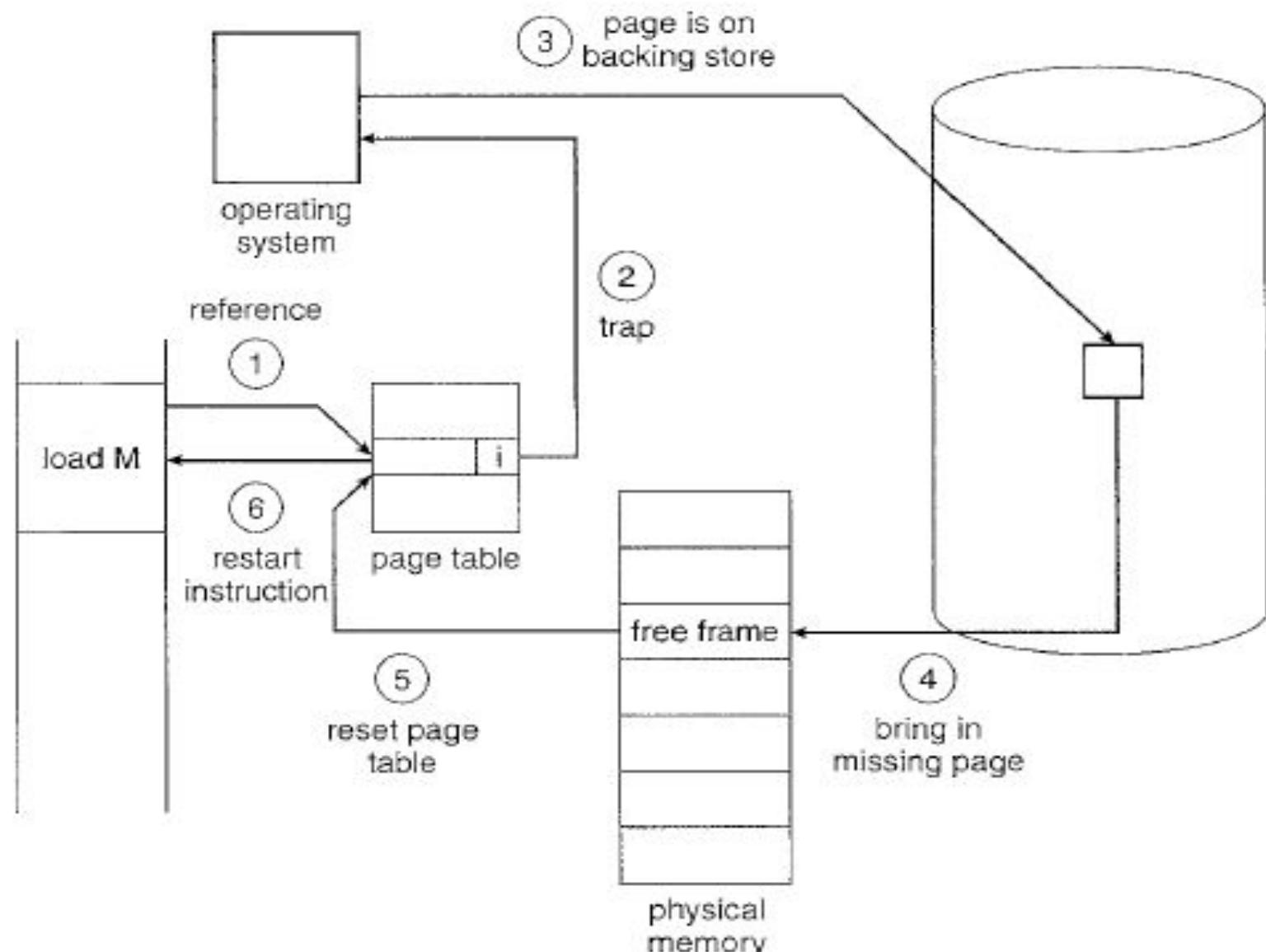
- Now we need some hardware to distinguish between which pages are in memory and which are not, so **valid -invalid bit scheme** can be used for it.
- When this bit is set to "**valid**" the associated page is both legal and in memory. If the bit is set to "**invalid**" the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.



Page Fault: - When a process tries to access a page that is not in main memory then a Page Fault Occurs.

Steps to handle Page Fault

- Page Fault Handling
- Allocating Frames
- Disk Operation
- Updating Tables
- Restarting the Process
- Instruction Restart Capability



Performance of Demand Paging

- **Effective Access time for Demand Paging:**
 - $(1 - p) \times ma + p \times \text{page fault service time.}$
- Here, p: Page fault rate or probability of a page fault.
- ma is memory access time.

Q Consider a process executing on an operating system that uses demand paging. The average time for a memory access in the system is M units if the corresponding memory page is available in memory, and D units if the memory access causes a page fault. It has been experimental measured that the average time taken for a memory access in the process is X units. Which one of the following is the correct expression for the page fault rate experienced by the process? **(GATE-2018) (2 Marks)**

- (A) $(D - M) / (X - M)$
- (B) $(X - M) / (D - M)$
- (C) $(D - X) / (D - M)$
- (D) $(X - M) / (D - X)$

Q Let the page fault service time be 10ms in a computer with average memory access time being 20ns. If one-page fault is generated for every 10^6 memory accesses, what is the effective access time for the memory? **(GATE-2011) (1 Marks)**

- (A)** 21ns **(B)** 30ns **(C)** 23ns **(D)** 35ns

Q Suppose the time to service a page fault is on the average 10 milliseconds, while a memory access takes 1 microsecond. Then a 99.99% hit ratio results in average memory access time of? **(GATE-2000) (1 Marks).**

(A) 1.9999 milliseconds **(B)** 1 millisecond

(C) 9.999 microseconds **(D)** 1.9999 microseconds

Q. Consider a demand paging memory management system with 32-bit logical address, 20-bit physical address, and page size of 2048 bytes. Assuming that the memory is byte addressable, what is the maximum number of entries in the page table? **(Gate 2025)**

- A) 2^{21}
- B) 2^{20}
- C) 2^{22}
- D) 2^{24}

**Q Increasing the RAM of a computer typically improves performance because:
(Gate-2005) (1 Marks)**

- (A) Virtual memory increases**
- (B) Larger RAMs are faster**
- (C) Fewer page faults occur**
- (D) Fewer segmentation faults occur**

Page Replacement

- With the increase in multiprogramming each process will get less amount of space in main memory so, the rate of page faults may rise. thus, to reduce the degree of multiprogramming the operating system swaps out processes from the memory freeing the frames and thus the process that requires to execute can now execute.
- If no frames are free, two-page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **Modify bit or Dirty Bit**. When this scheme is used, each page or frame has a modify bit associated with it in the hardware.

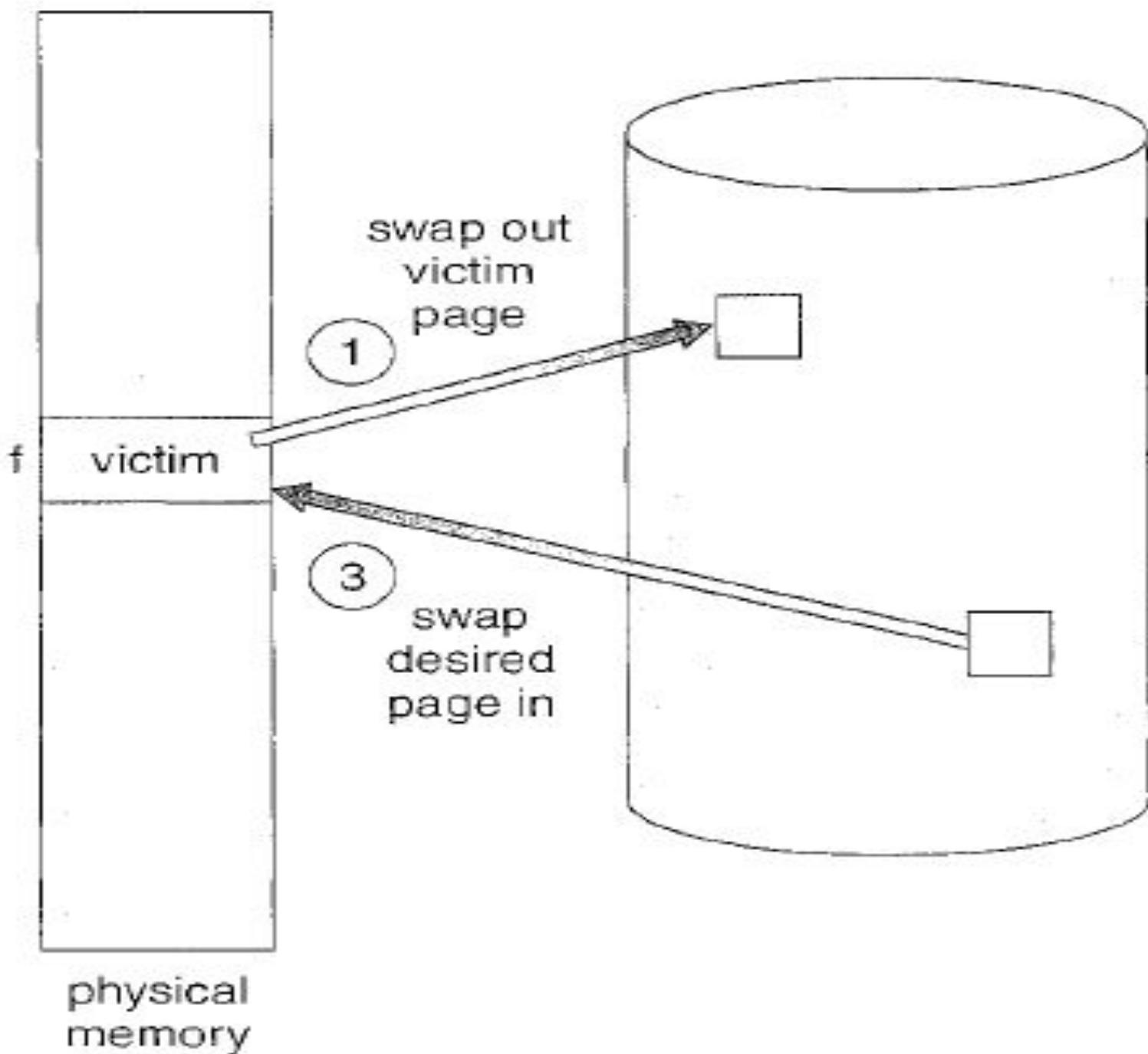
frame valid-invalid bit



② change to invalid

page table

④ reset page table for new page



swap out
victim
page

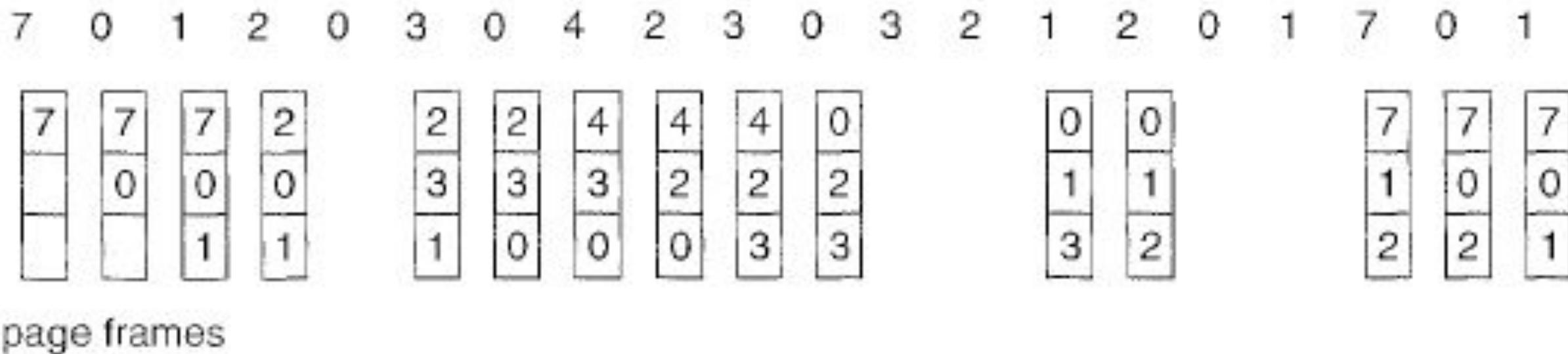
1

3 swap
desired
page in

- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- **If the bit is set:** the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- **If the modify bit is not set:** however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.
- Now, we must solve two major problems to implement demand paging: We must be able to develop a **frame allocation algorithm** and a **page replacement algorithm**.
- Frame Allocation will decide how much frames to allocate to a process.(already discussed)
- Page Replacement will decide which page to replace next.

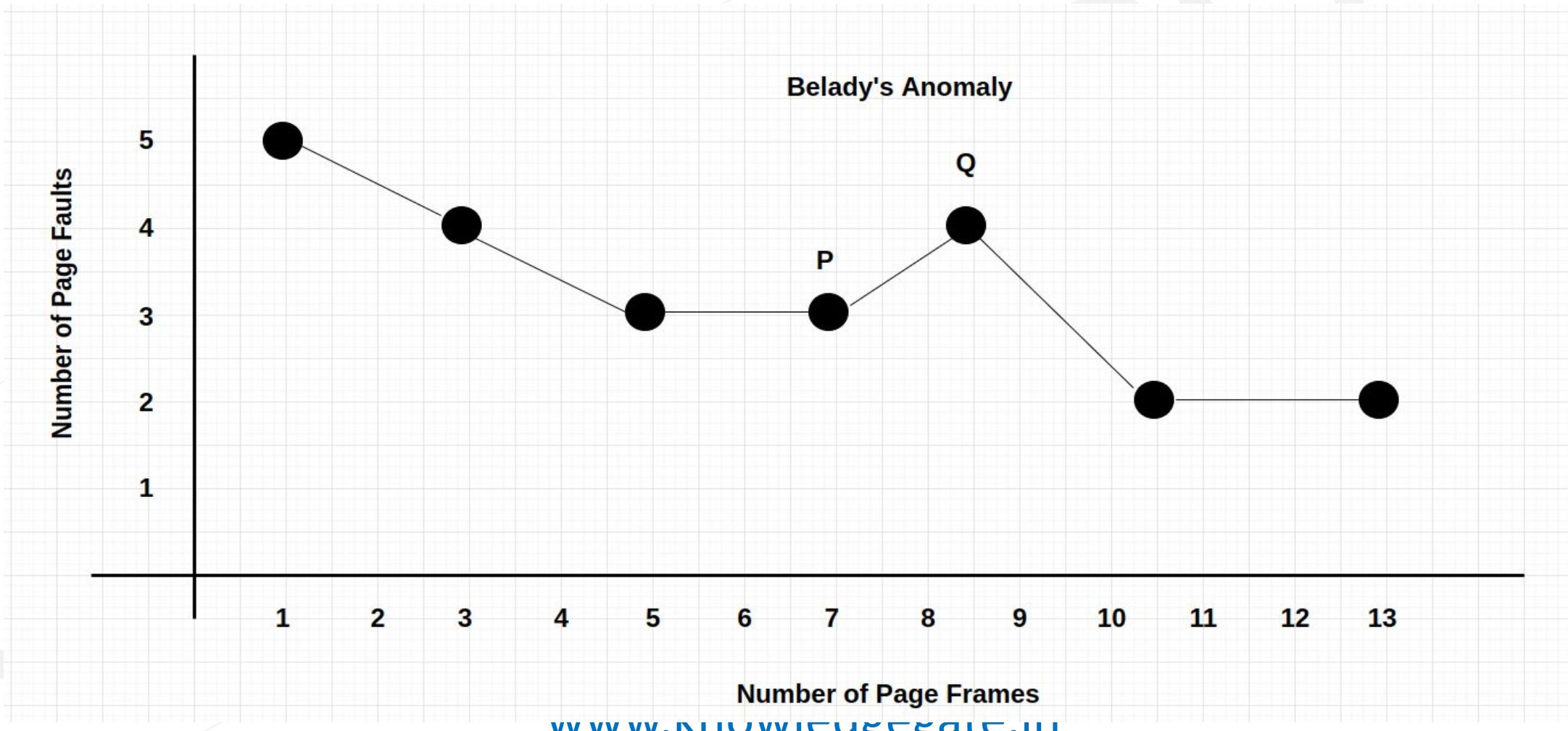
Page Replacement Algorithms

- **First In First Out Page Replacement Algorithm:** - A FIFO replacement algorithm associates with each page, the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. i.e. the first page that came into the memory will be replaced first.



- In the above example the number of page fault is 15.

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
- **Belady's Anomaly**: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.



Q. In which one of the following page replacement algorithms it is possible for the page fault rate to increase even when the number of allocated frames increases?

(GATE-2016) (1 Marks)

(a) LRU (Least Recently Used)

(b) OPT (Optimal Page Replacement)

(c) MRU (Most Recently Used)

(d) FIFO (First In First Out)

Q A system uses FIFO policy for page replacement. It has 4-page frames with no pages loaded to begin with. The system first accesses 100 distinct pages in some order and then accesses the same 100 pages but now in the reverse order. How many page faults will occur? **(GATE-2010) (1 Marks)**

- (A) 196
- (B) 192
- (C) 197
- (D) 195

Q In which one of the following page replacement policies, Belady's anomaly may occur? (GATE-2009) (1 Marks)

- (A) FIFO
- (B) Optimal
- (C) LRU
- (D) MRU

Q A virtual memory system uses First In First Out (FIFO) page replacement policy and allocates a fixed number of frames to a process. Consider the following statements: **(GATE-2007) (2 Marks)**

P: Increasing the number of page frames allocated to a process sometimes increases the page fault rate.

Q: Some programs do not exhibit locality of reference.

Which one of the following is TRUE?

- (A)** Both P and Q are true, and Q is the reason for P
- (B)** Both P and Q are true, but Q is not the reason for P.
- (C)** P is false, but Q is true
- (D)** Both P and Q are false.

Q Consider a virtual memory system with FIFO page replacement policy. For an arbitrary page access pattern, increasing the number of page frames in main memory will **(GATE-2001) (1 Marks)**

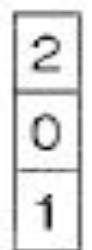
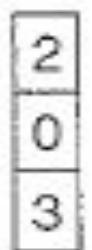
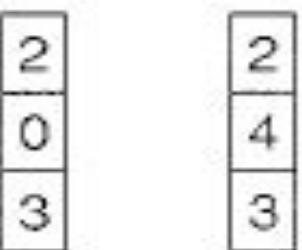
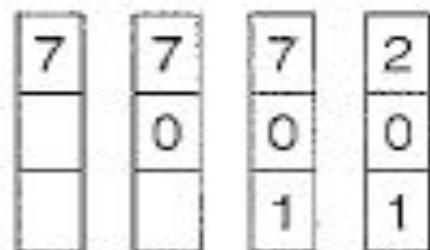
- (A) always decrease the number of page faults
- (B) always increase the number of page faults
- (C) sometimes increase the number of page faults
- (D) never affect the number of page faults

Optimal Page Replacement Algorithm

- Replace the page that will not be used for the longest period of time. It has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. It is mainly used for comparison studies.

reference string

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



page frames

Q Assume that there are 3 page frames which are initially empty. If the page reference string is 1, 2, 3, 4, 2, 1, 5, 3, 2, 4, 6, the number of page faults using the optimal replacement policy is _____. **(GATE-2014) (2 Marks)**

Q A process has been allocated 3-page frames. Assume that none of the pages of the process are available in the memory initially. The process makes the following sequence of page references (reference string): 1, 2, 1, 3, 7, 4, 5, 6, 3, 1. If optimal page replacement policy is used, how many page faults occur for the above reference string?

(GATE-2007) (2 Marks)

- (A) 7
- (B) 8
- (C) 9
- (D) 10

Q The optimal page replacement algorithm will select the page that **(GATE-2002) (1 Marks)**

- (A)** Has not been used for the longest time in the past.
- (B)** Will not be used for the longest time in the future.
- (C)** Has been used least number of times.
- (D)** Has been used most number of times.

Q. In optimal page replacement algorithm, information about all future page references is available to the operating system (OS). A modification of the optimal page replacement algorithm is as follows:

The OS correctly predicts only up to next 4 page references (including the current page) at the time of allocating a frame to a page.

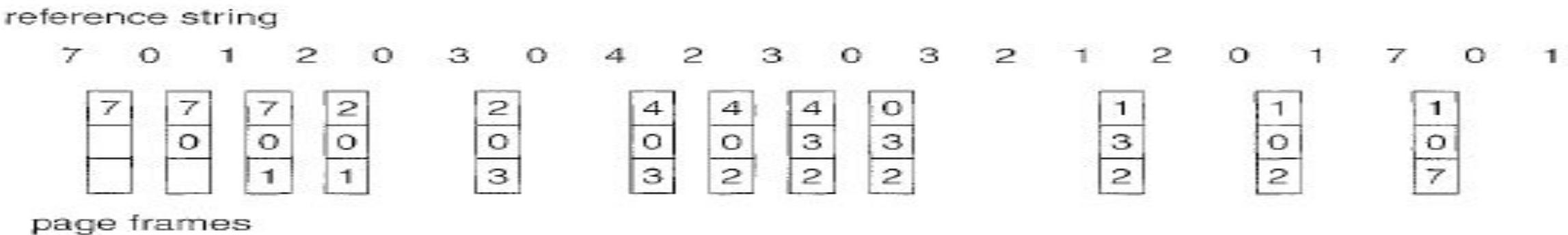
A process accesses the pages in the following order of page numbers:

1, 3, 2, 4, 2, 3, 1, 2, 4, 3, 1, 4.

If the system has three memory frames that are initially empty, the number of page faults that will occur during execution of the process is _____. (Answer in integer)? (**Gate 2025**)

Least Recently Used (LRU) Page Replacement Algorithm

- Replace the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.



- LRU is much better than FIFO replacement. The LRU policy is often used as a page-replacement algorithm and is considered to be good. LRU also does not suffer from Belady's Anomaly.

- The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance.
- LRU and OPT belong to a class of page-replacement algorithms, called **stack algorithms** and can never exhibit Belady's anomaly.
- A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n + 1$ frames.

Q Consider a demand paging system with four page frames (initially empty) and LRU page replacement policy. For the following page reference string, the page fault rate, defined as the ratio of number of page faults to the number of memory accesses (rounded off to one decimal place) is _____. **(GATE 2022) (2 MARKS)**

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

Q Recall that Belady's anomaly is that the page-fault rate may increase as the number of allocated frames increases. Now, consider the following statements:

S₁: Random page replacement algorithm (where a page chosen at random is replaced) suffers from Belady's anomaly

S₂: LRU page replacement algorithm suffers from Belady's anomaly

Which of the following is CORRECT? (GATE-2017) (1 Marks)

(a) S₁ is true, S₂ is true

(b) S₁ is true, S₂ is false

(c) S₁ is false, S₂ is true

(d) S₁ is false, S₂ is false

Q Consider a main memory with five page frames and the following sequence of page references: 3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 2, 1, 3. Which one of the following is true with respect to page replacement policies First-In-First Out (FIFO) and Least Recently Used (LRU)?

(GATE-2015) (2 Marks)

- (A)** Both incur the same number of page faults
- (B)** FIFO incurs 2 more-page faults than LRU
- (C)** LRU incurs 2 more-page faults than FIFO
- (D)** FIFO incurs 1 more page faults than LRU

Q A system uses 3-page frames for storing process pages in main memory. It uses the Least Recently Used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below? 4, 7, 6, 1, 7, 6, 1, 2, 7, 2 **(GATE-2014) (2 Marks)**

Q Consider the virtual page reference string 1, 2, 3, 2, 4, 1, 3, 2, 4, 1 On a demand paged virtual memory system running on a computer system that main memory size of 3 pages frames which are initially empty. Let LRU, FIFO and OPTIMAL denote the number of page faults under the corresponding page replacements policy. Then **(GATE-2012) (2 Marks)**

- (A) OPTIMAL < LRU < FIFO
- (B) OPTIMAL < FIFO < LRU
- (C) OPTIMAL = LRU
- (D) OPTIMAL = FIFO

Q A process, has been allocated 3-page frames. Assume that none of the pages of the process are available in the memory initially. The process makes the following sequence of page references (reference string): 1,2,1,3,7,4,5,6,3,1 Least Recently Used (LRU) page replacement policy is a practical approximation to optimal page replacement. For the above reference string, how many more page faults occur with LRU than with the optimal page replacement policy? **(GATE-2007) (2 Marks)**

- (A) 0
- (B) 1
- (C) 2
- (D) 3

Solution: Two implementations are feasible:

- **Counters.** In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter.
- The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock registers are copied to the time-of-use field in the page-table entry for that page.
- In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.

Stack: Another approach to implementing LRU replacement is to keep a stack of page numbers.

- Whenever a page is referenced, it is removed from the stack and put on the top.
- In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Q. Consider a demand paging system with three frames, and the following page reference string: 1 2 3 4 5 4 1 6 4 5 1 3 2. The contents of the frames are as follows initially and after each reference (from left to right):

| <i>initially</i> | <i>after</i> | | | | | | | | | | | | |
|------------------|--------------|----|----|----|----|---|---|----|---|---|----|----|----|
| - | 1* | 2* | 3* | 4* | 5* | 4 | 1 | 6* | 4 | 5 | 1* | 3* | 2* |
| - | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 2 |
| - | - | 2 | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 1 | 1 | 1 |
| - | - | - | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 3 | 3 |

The *-marked references cause page replacements.

Which one or more of the following could be the page replacement policy/policies in use? **(Gate 2025)**

- A) Least Recently Used page replacement policy
- B) Least Frequently Used page replacement policy
- C) Most Frequently Used page replacement policy
- D) Optimal page replacement policy

Counting-Based Page Replacement

- We keep a counter of the number of references that have been made to each page and develop the following two schemes.
- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced.
- The reason for this selection is that an actively used page should have a large reference count.
- A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- **The most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
- Neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Q Consider a computer system with ten physical page frames. The system is provided with an access sequence $(a_1, a_2, \dots, a_{20}, a_1, a_2, \dots, a_{20})$, where each a_i is a distinct virtual page number. The difference in the number of page faults between the last-in-first-out page replacement policy and the optimal page replacement policy is _____. (GATE-2016) (1 Marks)

Q A computer has twenty physical page frames which contain pages numbered 101 through 120. Now a program accesses the pages numbered 1, 2, ..., 100 in that order, and repeats the access sequence THREE. Which one of the following page replacement policies experiences the same number of page faults as the optimal page replacement policy for this program? (GATE-2014) (2 Marks)

- (A) Least-recently-used
- (B) First-in-first-out
- (C) Last-in-first-out
- (D) Most-recently-used

Q Increasing the RAM of a computer typically improves performance because
(GATE-2005) (1 Marks) (ISRO-2015)

(A) Virtual memory increases

(B) Larger RAMs are faster

(C) Fewer page faults occur

(D) Fewer segmentation faults occur

Frame Allocation Algorithms

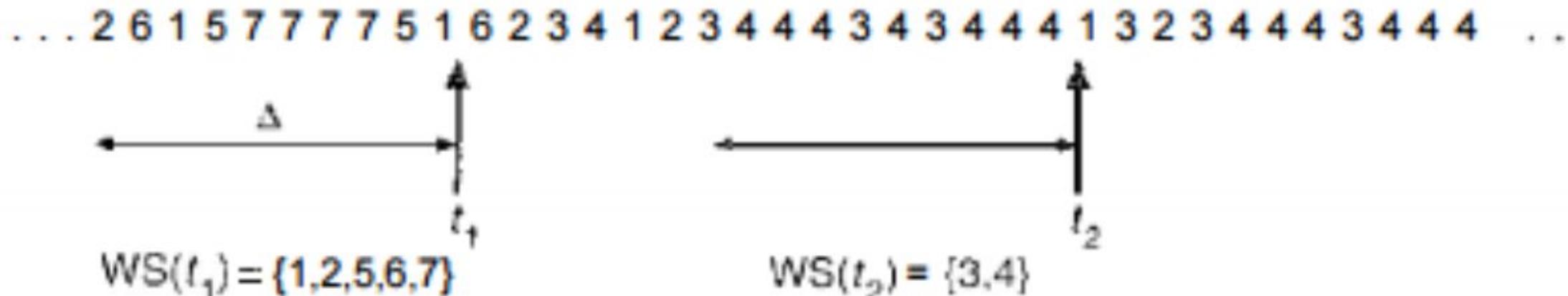
- Minimum Number of frames to be allocated to each process depends on Instruction Set Architecture, and the maximum number of allocation of frames depends on the size of the process.
- **Equal Allocation:** Frames will be equally allocated to each process. Ex: if we have 30 frames and 3 processes, each will get 10 regardless of their size.

- **Weighted / Proportional Allocation:** Frames will be allocated according to the weights of the process. The allocation of frames per process: $a(i) = s(i)/S \times m$.
- Where, $a(i)$ is the number of allocated frames, $s(i)$ is the process and S is the size of the virtual memory in general, $S = \sum s(i)$, and m is the total number of frames available.
- Example: If we have 3 processes of size 20 k, 30k and 50 k then 30 frames will be allocated as:
- P1 gets: $20/100 * 30 = 6$ frames, similarly P2 and P3 will get 9 and 15 frames respectively.

Solution (The Working Set Strategy)

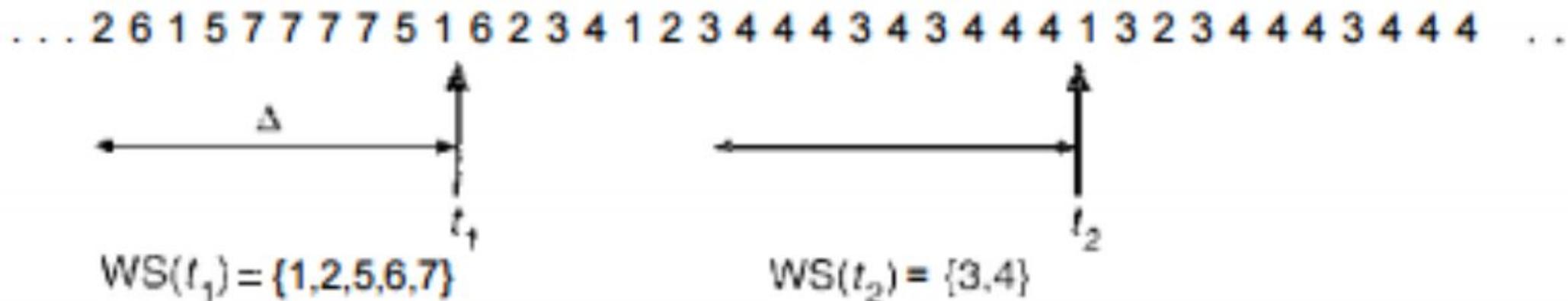
- This approach defines the **locality model** of process execution. The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.
- This model uses a parameter Δ , to define the **working set window**. The set of pages in the most recent Δ page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. The working set is an approximation of the program's locality.

page reference table



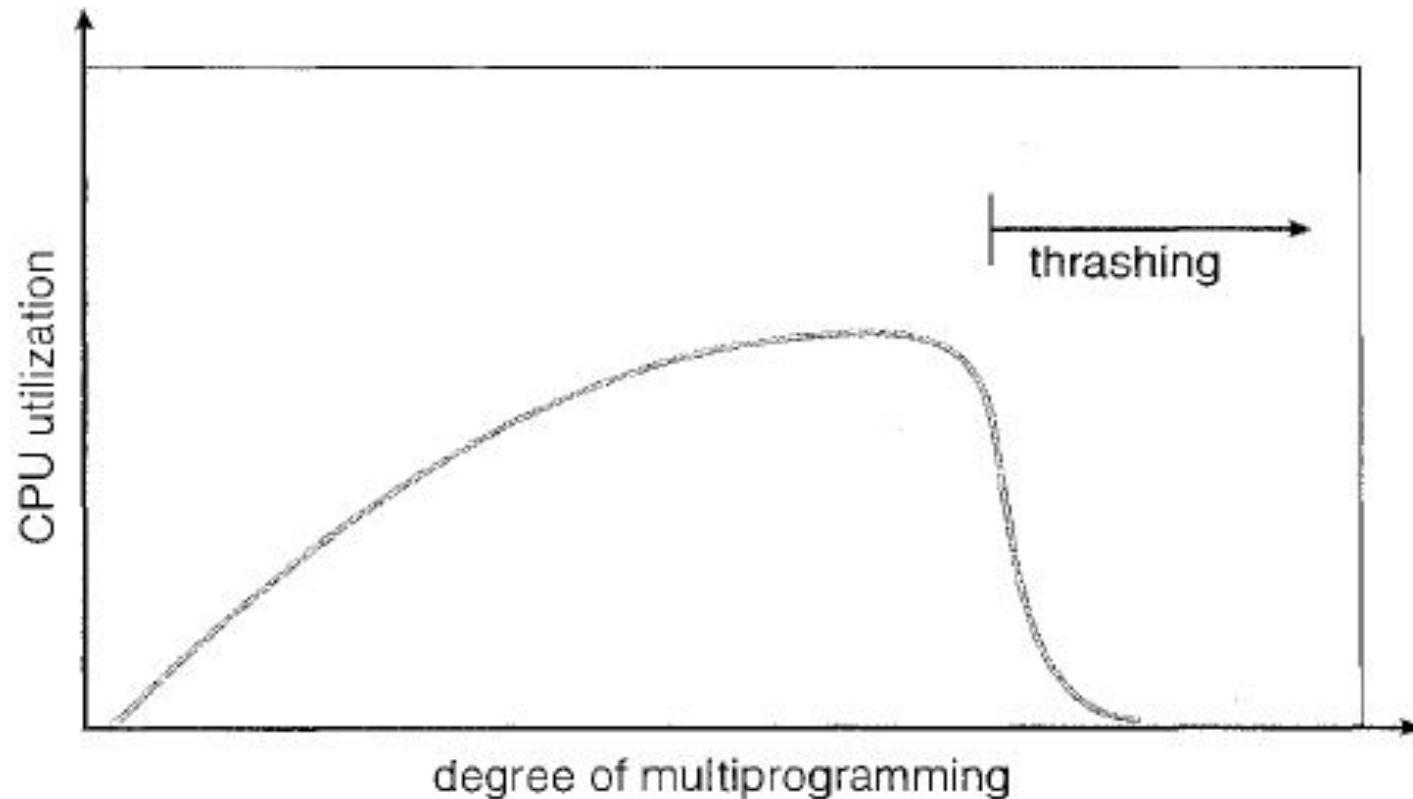
- The accuracy of the working set depends on the selection of Δ . If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities.
- If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

page reference table



Thrashing

- A process is thrashing if it is spending more time paging than executing. High paging activity is called Thrashing.



Causes of Thrashing

- When CPU utilization is low, an increase in the number of processes (multiprogramming) may lead to more page faults as processes compete for memory.
- This excessive paging causes a queue at the paging device, emptying the ready queue and further dropping CPU utilization. In response, the CPU scheduler may increase multiprogramming, worsening the situation.
- This cycle, known as thrashing, significantly decreases system throughput as processes spend most of their time managing memory rather than executing, leading to a sharp drop in overall performance.

Some points to remember

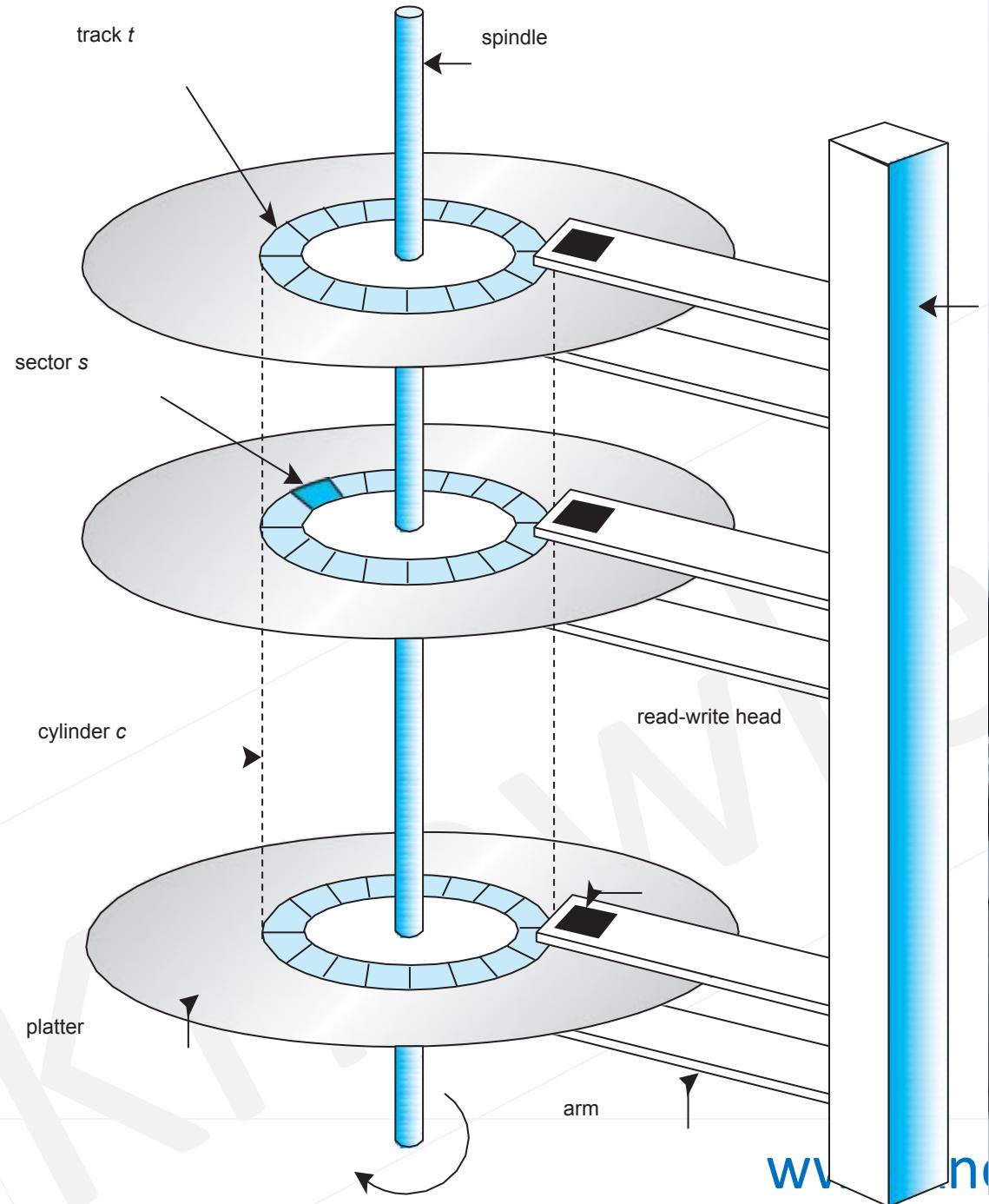
- If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
- Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Local Replacement: Local replacement requires that each process select from only its own set of allocated frames.

- With a local replacement strategy, the number of frames allocated to a process does not change.
- Under local replacement, the set of pages in memory for a process is affected by the paging behaviour of only that process.
- Local replacement might hinder a process, however, by not making available to it other, less used pages of memory.

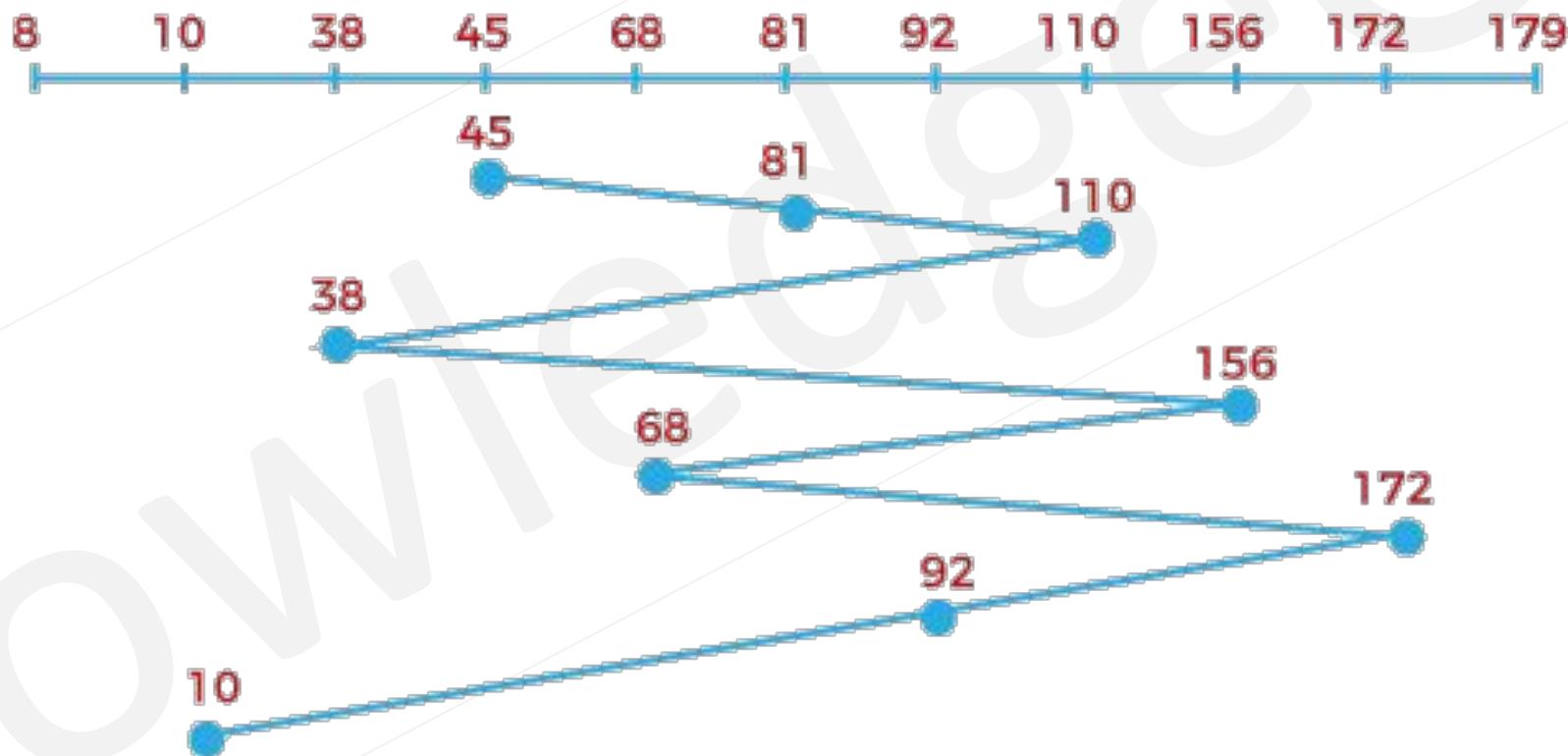
Global Replacement: Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.
- One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. As the set of pages in memory for a process depends not only on the paging behaviour of that process but also on the paging behaviour of other processes.
- Global replacement generally results in greater system throughput and is therefore the more common method.



Disk scheduling

- When one i/o request is completed, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used.



FCFS Disk Scheduling Algorithm

FCFS (First Come First Serve)

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. In FCFS, the requests are addressed in the order they arrive in the disk queue. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

Advantages:

- Easy to understand easy to use
- Every request gets a fair chance
- no starvation (may suffer from convoy effect)

Disadvantages:

- Does not try to optimize seek time (extra seek movements)
- May not provide the best possible service

SSTF Scheduling

- It seems reasonable to service all the requests close to the current head position before moving the head far to service other REQUESTS. This assumption is the basis for the SSTF algorithm.
- In SSTF (Shortest Seek Time First), the request nearest to the disk arm will get executed first i.e. requests having shortest seek time are executed first.
- Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

Advantages:

- Seek movements decreases
- Average Response Time decreases
- Throughput increases

Disadvantages:

- Overhead to calculate the closest request.
- Can cause Starvation for a request which is far from the current location of the header
- High variance of response time as SSTF favours only some requests
- SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests.

Q Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence 4, 34, 10, 7, 19, 73, 2, 15, 6, 20. Assuming that the head is currently at cylinder 50, what is the time taken to satisfy all requests if it takes 1ms to move from one cylinder to adjacent one and shortest seek time first policy is used? (GATE-2009) (1 Marks)

- (A) 95 ms (B) 119 ms (C) 233 ms (D) 276 ms

Q Suppose a disk has 201 cylinders, numbered from 0 to 200. At some time, the disk arm is at cylinder 100, and there is a queue of disk access requests for cylinders 30, 85, 90, 100, 105, 110, 135 and 145. If Shortest-Seek Time First (SSTF) is being used for scheduling the disk access, the request for cylinder 90 is serviced after servicing _____ number of requests.
(GATE-2014) (1 Marks)

SCAN

The disk arm starts at one end of the disk and moves towards the other end, servicing requests as it reaches each track, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the Elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.
- Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest.

Advantages:

- Simple easy to understand and use
- No starvation but more wait for some random process
- Low variance and Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm.
- Unnecessary move to the end of the disk, even if there is no request.

Q Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given: 45, 20, 90, 10, 50, 60, 80, 25, 70. Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the Shortest Seek Time First (SSTF) algorithm is used compared to the SCAN (Elevator) algorithm (assuming that SCAN algorithm moves towards 100 when it starts execution) is _____ tracks (**GATE-2015**) (2 Marks)

C-SCAN Scheduling

Circular-scan is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Advantages:

- Provides more uniform wait time compared to SCAN
- Better response time compared to scan

Disadvantage:

- More seeks movements in order to reach starting position

Q Consider the situation in which the disk read/write head is currently located at track 45 (of tracks 0-255) and moving in the positive direction. Assume that the following track requests have been made in this order: 40, 67, 11, 240, 87. What is the order in which C-SCAN would service these requests and what is the total seek distance?

- (A) 600 (B) 810 (C) 505 (D) 550

LOOK Scheduling

It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus, it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Advantage: -

- Better performance compared to SCAN
- Should be used in case to less load

Disadvantage: -

- Overhead to find the last request
- Should not be used in case of more load.

C LOOK

As LOOK is similar to SCAN algorithm, in similar way, C-LOOK is similar to C-SCAN disk scheduling algorithm. In C-LOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Advantage: -

- Provides more uniform wait time compared to LOOK
- Better response time compared to LOOK

Disadvantage: -

- Overhead to find the last request and go to initial position is more
- Should not be used in case of more load.

Q Consider the situation in which the disk read/write head is currently located at track 45 (of tracks 0-255) and moving in the positive direction. Assume that the following track requests have been made in this order: 40, 67, 11, 240, 87. What is the order in which optimized C-SCAN would service these requests and what is the total seek distance?**(GATE-2015) (2 Marks)**

- (A) 600
- (B) 810
- (C) 505
- (D) 550

Q Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given: 45, 20, 90, 10, 50, 60, 80, 25, 70. Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the Shortest Seek Time First (SSTF) algorithm is used compared to the SCAN (Elevator) algorithm (assuming that SCAN algorithm moves towards 100 when it starts execution) is _____ tracks **(GATE-2015) (2 Marks)**

Q Consider a disk queue with requests for I/O to blocks on cylinders 47, 38, 121, 191, 87, 11, 92, 10. The C-LOOK scheduling algorithm is used. The head is initially at cylinder number 63, moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____. (GATE-2018) (2 Marks)

Conclusion

Selection of a Disk-Scheduling Algorithm

- Given so many disk-scheduling algorithms, how do we choose the best one? SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem.
- With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice of where to move the disk head: they all behave like FCFS scheduling.

Q Consider an operating system capable of loading and executing a single sequential user process at a time. The disk head scheduling algorithm used is First Come First Served (FCFS). If FCFS is replaced by Shortest Seek Time First (SSTF), claimed by the vendor to give 50% better benchmark results, what is the expected improvement in the I/O performance of user programs? **(GATE-2004) (1 Marks)**

- (A) 50%
- (B) 40%
- (C) 25%
- (D) 0%

Q Which of the following disk scheduling strategies is likely to give the best throughput? **(GATE-1999) (1 Marks)**

- (a)** Farthest cylinder next
- (c)** First come first served
- (b)** Nearest cylinder next
- (d)** Elevator algorithm

Q Consider a storage disk with 4 platters (numbered as 0, 1, 2 and 3), 200 cylinders (numbered as 0, 1, ..., 199), and 256 sectors per track (numbered as 0, 1, ... 255). The following 6 disk requests of the form [sector number, cylinder number, platter number] are received by the disk controller at the same time:

[120, 72, 2], [180, 134, 1], [60, 20, 0], [212, 86, 3], [56, 116, 2], [118, 16, 1]

Currently head is positioned at sector number 100 of cylinder 80, and is moving towards higher cylinder numbers. The average power dissipation in moving the head over 100 cylinders is 20 milliwatts and for reversing the direction of the head movement once is 15 milliwatts. Power dissipation associated with rotational latency and switching of head between different platters is negligible.

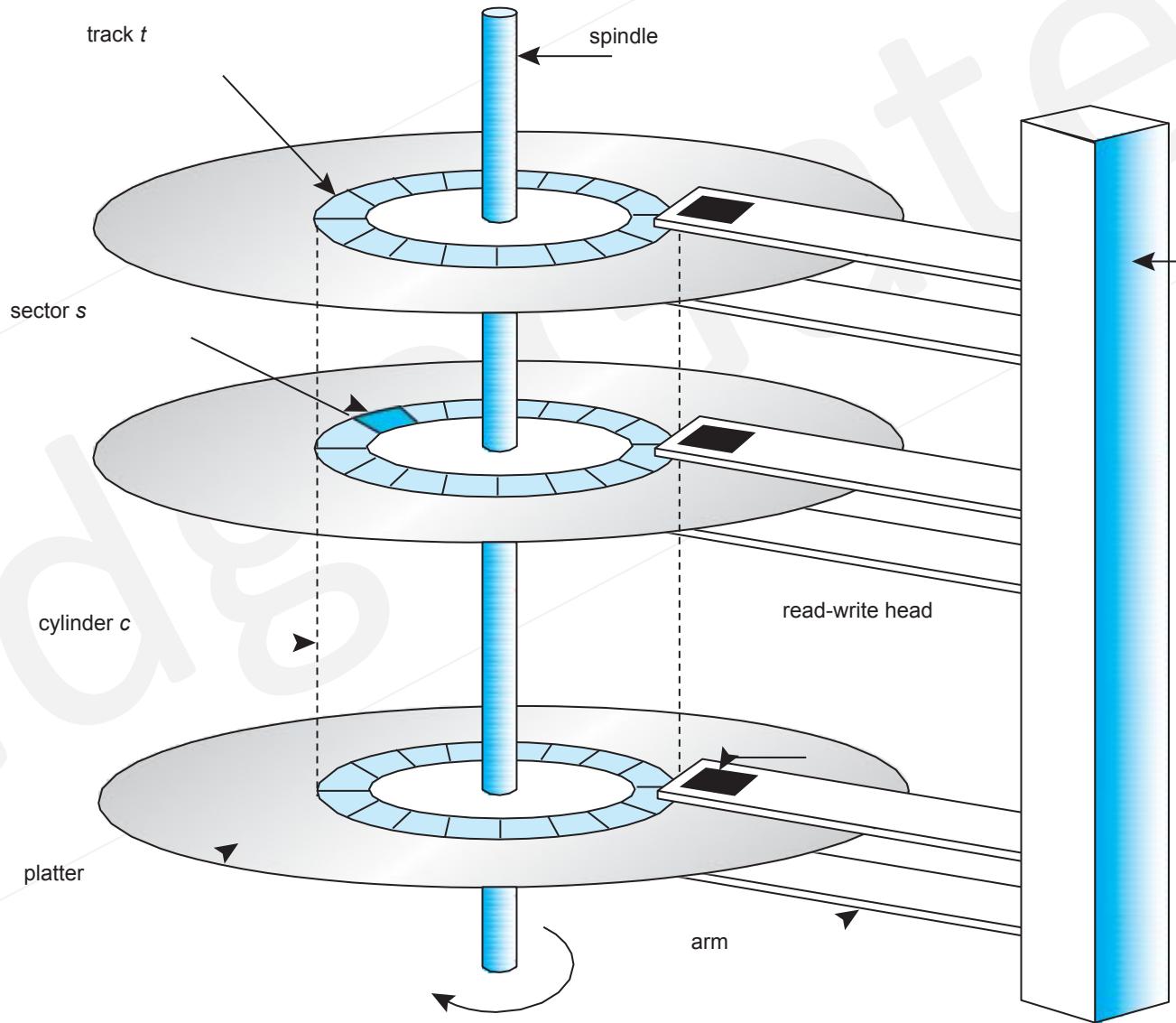
The total power consumption in milliwatts to satisfy all of the above disk requests using the Shortest Seek Time First disk scheduling algorithm is _____. **(GATE-2018) (2 Marks)**

Q For a magnetic disk with concentric circular tracks, the seek latency is not linearly proportional to the seek distance due to **(GATE-2008) (1 Marks)**

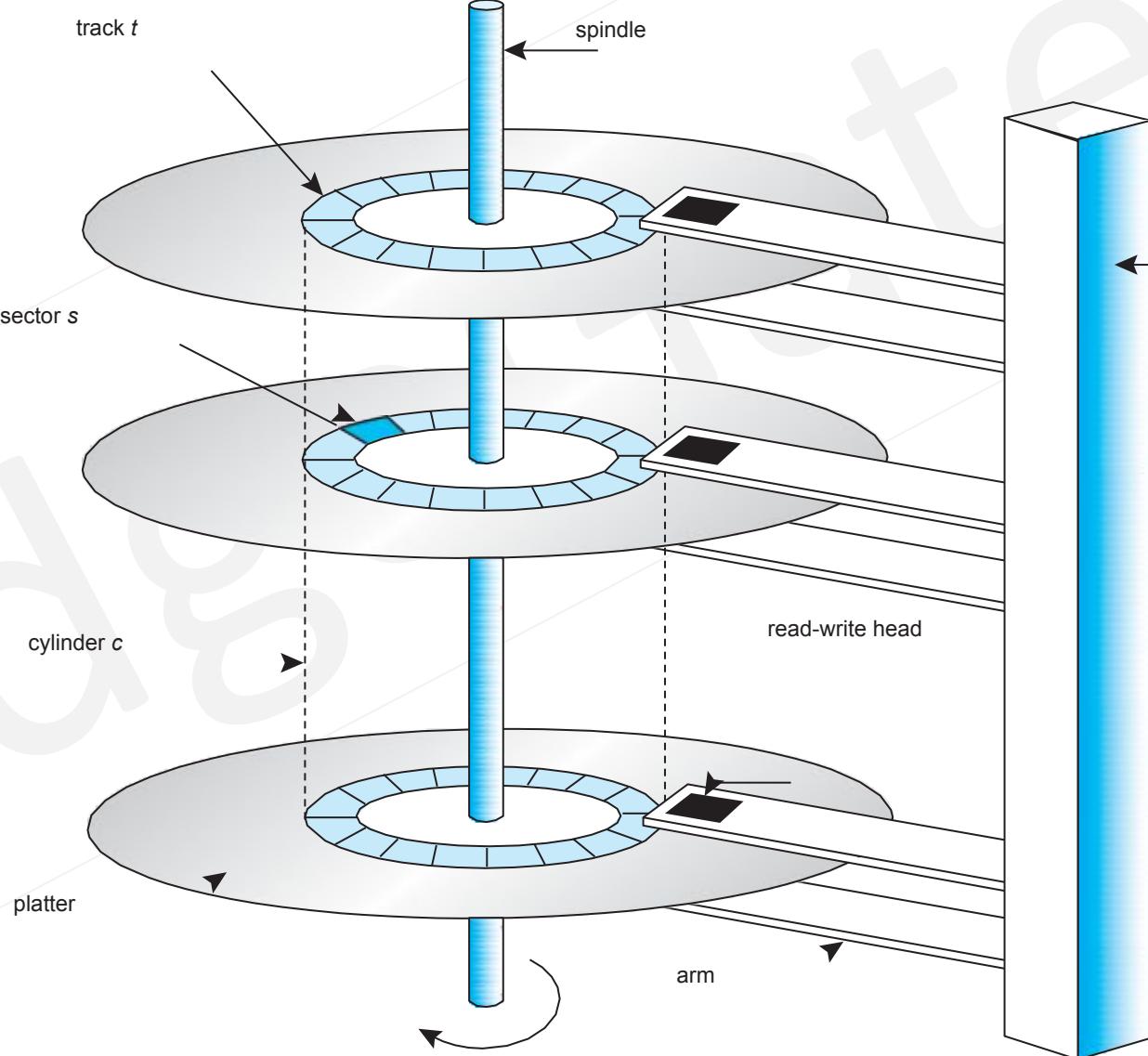
- a)** non-uniform distribution of requests
- b)** arm starting and stopping inertia
- c)** higher capacity of tracks on the periphery of the platter
- d)** use of unfair arm scheduling policies

Transfer time

- Magnetic disks provide the bulk of secondary storage for modern computer systems. Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.
- A read – write head “flies” just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors.



- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (**RPM**). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM.
- Total transfer time has two parts. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.



- Total Transfer Time = Seek Time + Rotational Latency + Transfer Time
- **Seek Time:** - It is a time taken by Read/Write header to reach the correct track. (Always given in question)
- **Rotational Latency:** - It is the time taken by read/Write header during the wait for the correct sector. In general, it's a random value, so far average analysis, we consider the time taken by disk to complete half rotation.
- **Transfer Time:** - it is the time taken by read/write header either to read or write on a disk. In general, we assume that in 1 complete rotation, header can read/write the either track, so
- total time will be = (File Size/Track Size) *time taken to complete one revolution.

Q Consider a disk where there are 512 tracks, each track is capable of holding 128 sectors and each sector holds 256 bytes, find the capacity of the track and disk and number of bits required to reach correct track, sector and disk.

Q. Consider a disk pack with 16 surfaces, 128 tracks per surface and 256 sectors per track. 512 bytes of data are stored in a bit serial manner in a sector. The capacity of the disk pack and the number of bits required to specify a particular sector in the disk are respectively

(GATE-2006) (2 Marks)

- (A) 256 Mbyte, 19 bits
- (B) 256 Mbyte, 28 bits
- (C) 512 Mbyte, 20 bits
- (D) 64 Gbyte, 28 bit

Q consider a disk where each sector contains 512 bytes and there are 400 sectors per track and 1000 tracks on the disk. If disk is rotating at speed of 1500 RPM, find the total time required to transfer file of size 1 MB. Suppose seek time is 4ms?

Q Consider a system with 8 sector per track and 512 bytes per sector. Assume that disk rotates at 3000 rpm and average seek time is 15ms standard. Find total time required to transfer a file which requires 8 sectors to be stored.

- a)** Assume contiguous allocation
- b)** Assume Non-contiguous allocation

Q. Consider a disk pack with a seek time of 4 milliseconds and rotational speed of 10000 rotations per minute (RPM). It has 600 sectors per track and each sector can store 512 bytes of data. Consider a file stored in the disk. The file contains 2000 sectors. Assume that every sector access necessitates a seek, and the average rotational latency for accessing each sector is half of the time for one complete rotation. The total time (in milliseconds) needed to read the entire file is _____. **(GATE-2015) (2 Marks)**

Q. Consider a typical disk that rotates at 15000 rotations per minute (RPM) and has a transfer rate of 50×10^6 bytes/sec. If the average seek time of the disk is twice the average rotational delay and the controller's transfer time is 10 times the disk transfer time, the average time (in milliseconds) to read or write a 512 byte sector of the disk is

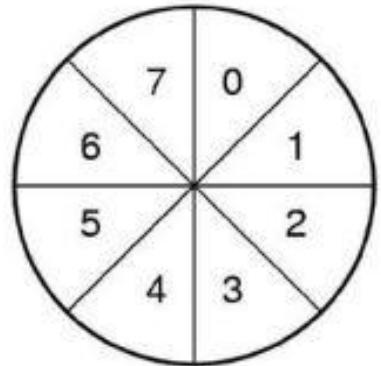
(GATE-2015) (2 Marks)

Q. An application loads 100 libraries at start-up. Loading each library requires exactly one disk access. The seek time of the disk to a random location is given as 10 milliseconds. Rotational speed of disk is 6000 rpm. If all 100 libraries are loaded from random locations on the disk, how long does it take to load all libraries? (*The time to transfer data from the disk block once the head has been positioned at the start of the block may be neglected*)

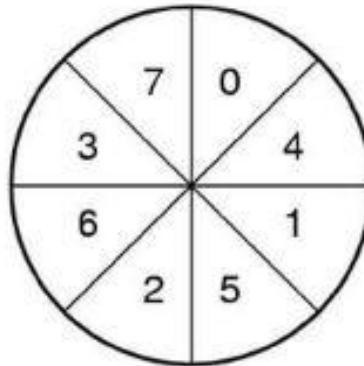
(GATE-2011) (2 Marks)

- (A) 0.50 s
- (B) 1.50 s
- (C) 1.25 s
- (D) 1.00 s

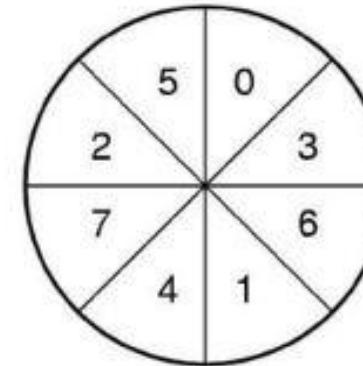
- **Single interleaving:** - in 2 rotation we read 1 track
- **Double interleaving:** - in 2.75 rotation we read 1 track



(a)



(b)



(c)

- No interleaving
- Single interleaving
- Double interleaving

File allocation methods

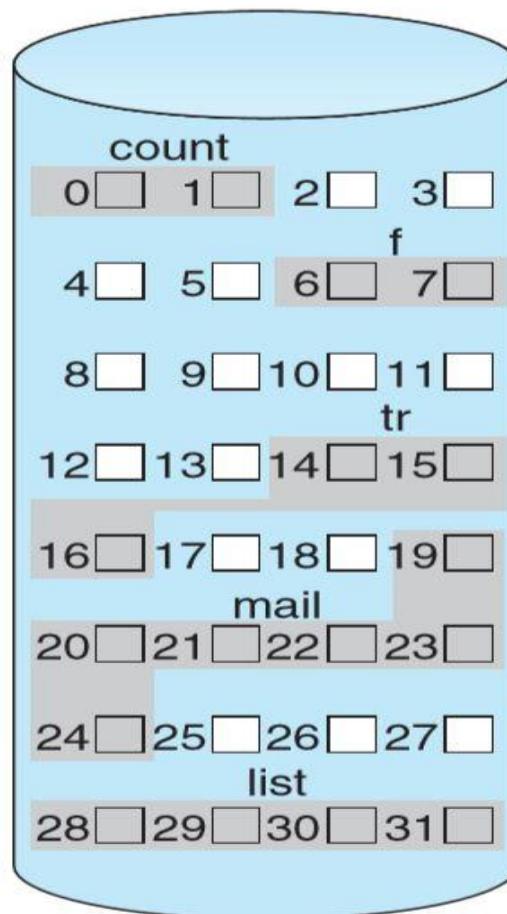
The main aim of file allocation problem is how disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use:

- **Contiguous**
- **Linked**
- **Indexed**

Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files.

Contiguous Allocation

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement.



| directory | | |
|-----------|-------|--------|
| file | start | length |
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

- **Advantage**

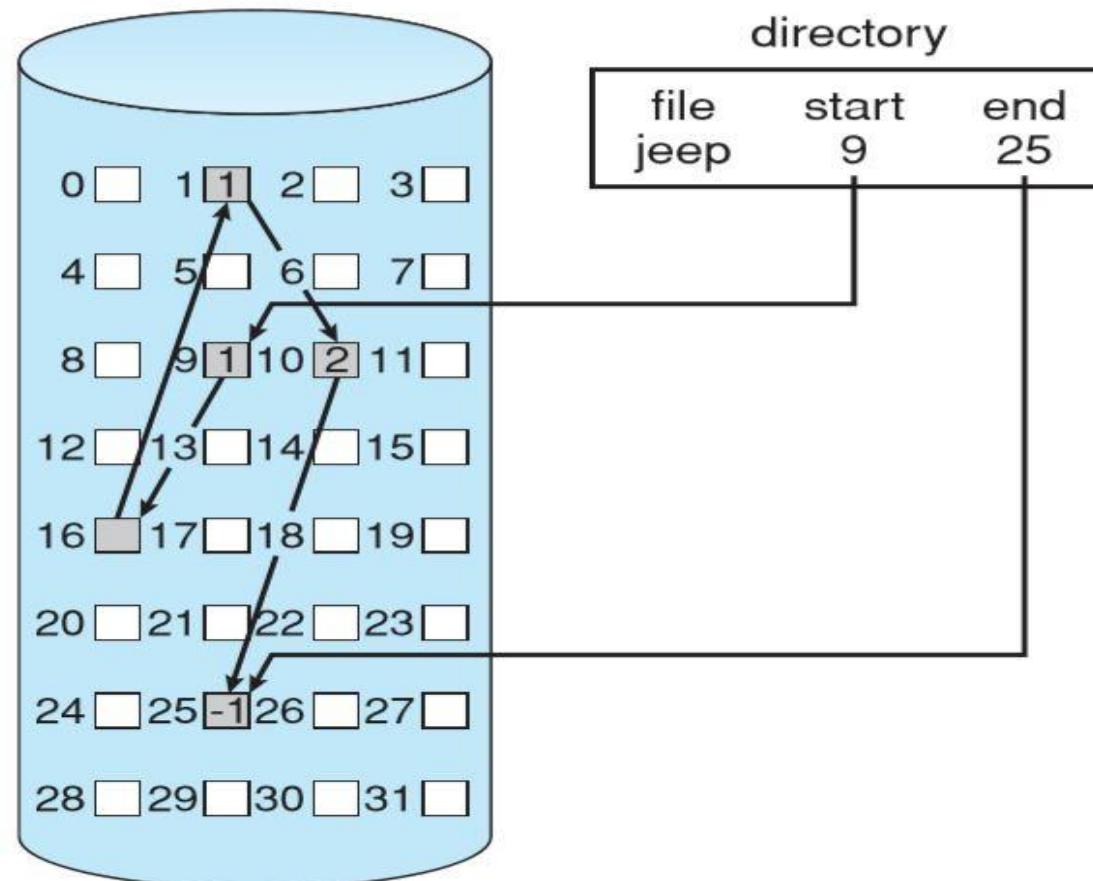
- Accessing a file that has been allocated contiguously is easy. Thus, both sequential and direct access can be supported by contiguous allocation.

- **Disadvantage**

- Suffer from the problem of external fragmentation.
- Suffer from huge amount of external fragmentation.
- Another problem with contiguous allocation file modification

Linked Allocation

- Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.



- **Advantage:** -

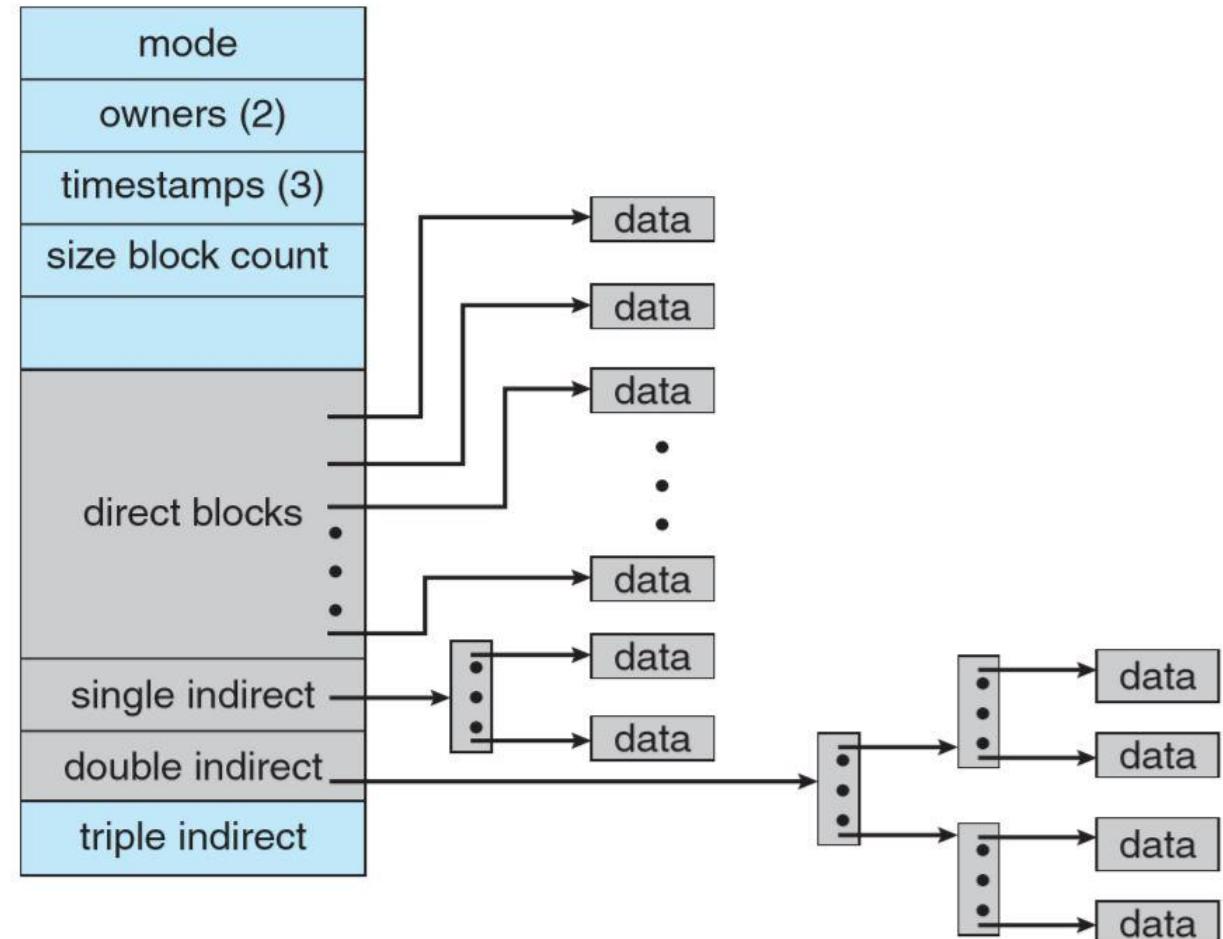
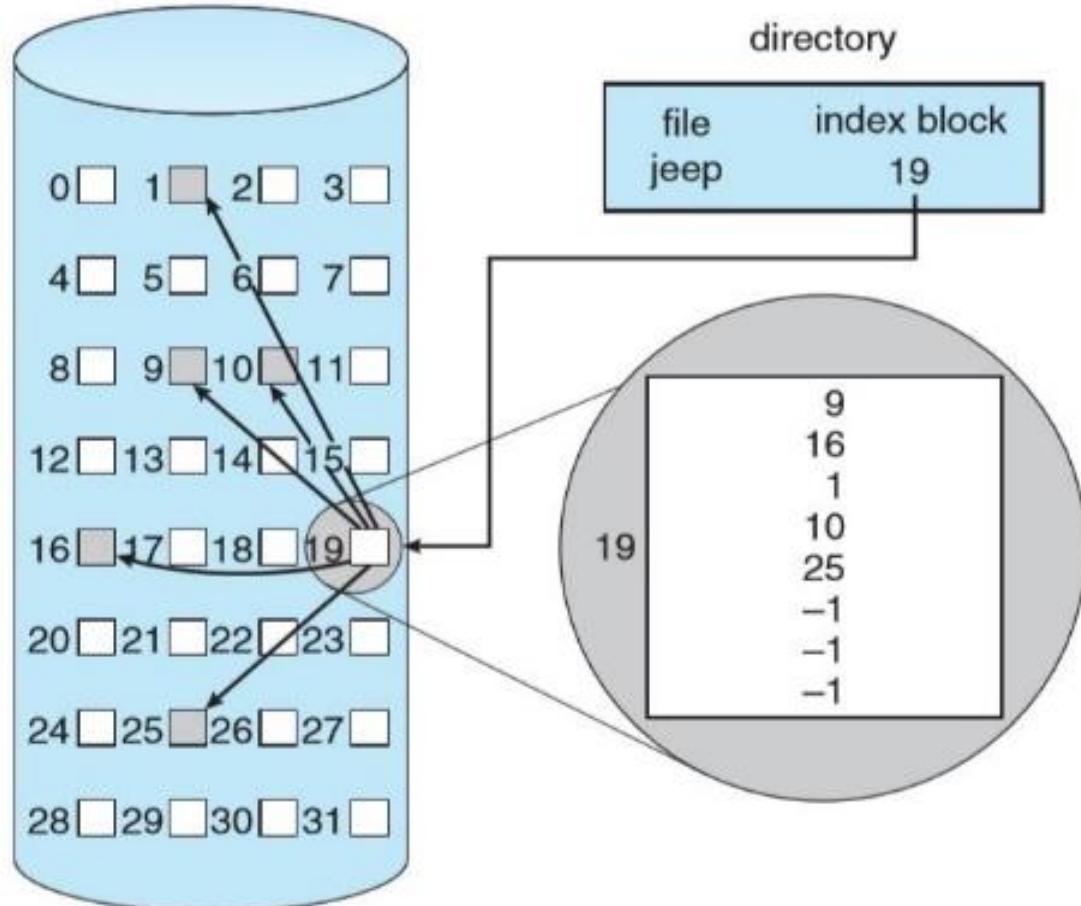
- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. To read a file, we simply read blocks by following the pointers from block to block. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available.
- There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.

- **Disadvantage:** -

- To find the i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} block. Each access to a pointer requires a disk read.
- Another disadvantage is the space required for the pointers, so each file requires slightly more space than it would otherwise.
- Yet another problem is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file.

Indexed Allocation

- Indexed allocation solves problems of contiguous and linked allocation, by bringing all the pointers together into one location: the index block.



- Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory entry contains the address of the index block. To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.
- When the file is created, all pointers in the index block are set to null. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.
- This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file.

- **Linked scheme:** To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is null (for a small file) or is a pointer to another index block (for a large file).
- **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

- **Combined scheme.** Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode.
- The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files do not need a separate index block.
- The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data.
- The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.

- **Advantage**
 - Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- **Disadvantage**
 - Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Q In a file allocation system, which of the following allocation scheme(s) can be used if no external fragmentation is allowed? **(GATE-2017) (1 Marks)**

- I. Contiguous
- II. Linked
- III. Indexed

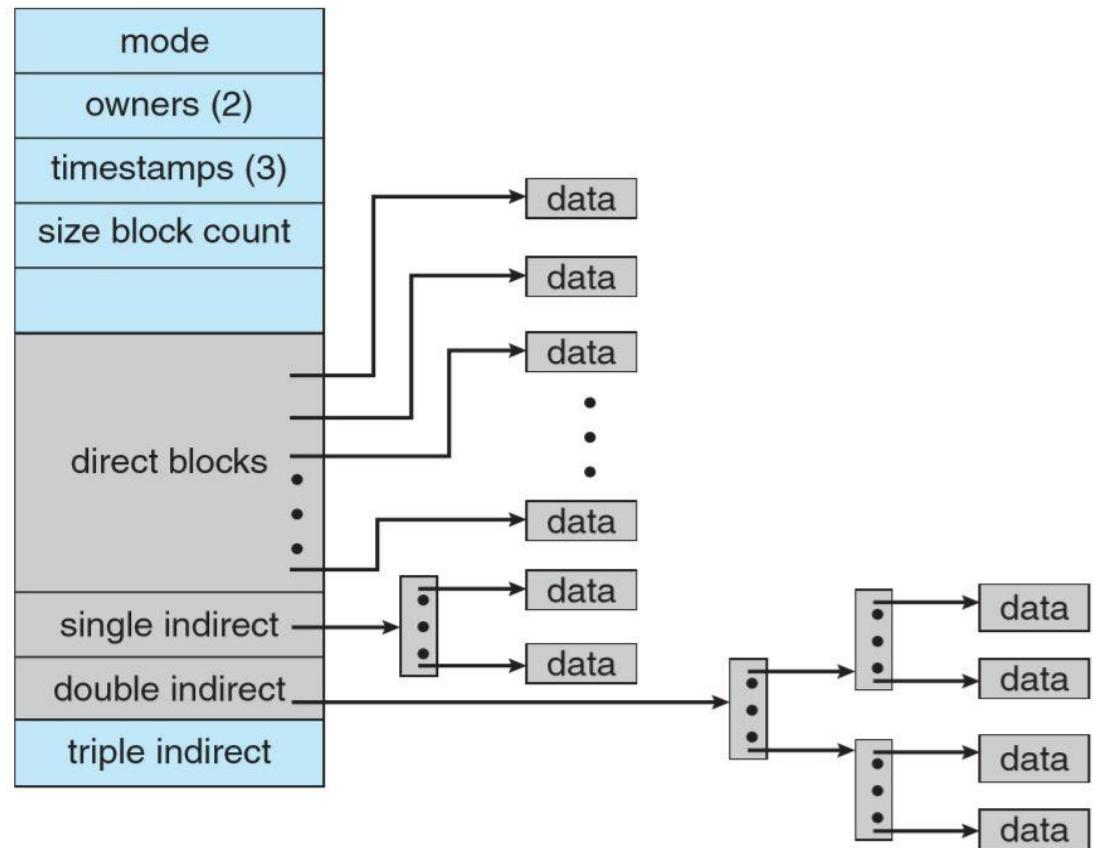
(a) I and III only

(b) II only

(c) III only

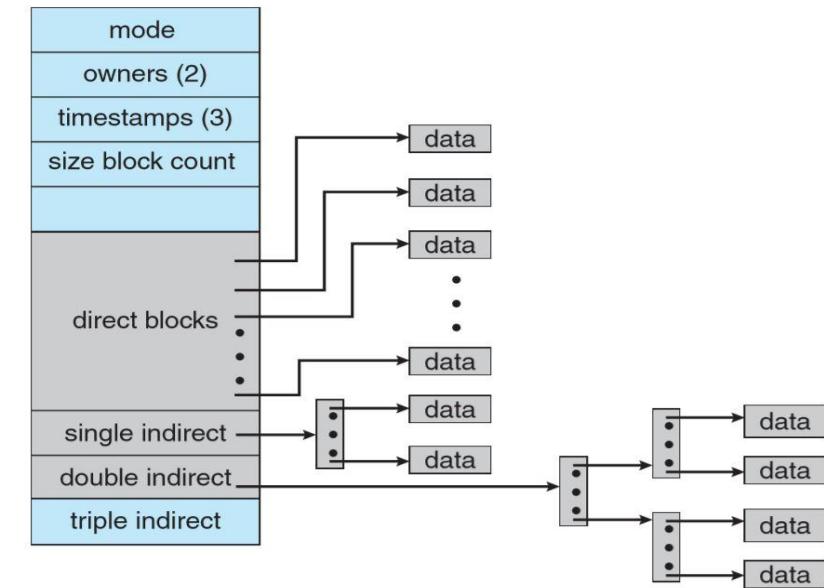
(d) II and III only

Q The index node (inode) of a Unix-like file system has 12 direct, one single-indirect and one double-indirect pointers. The disk block size is 4 kB, and the disk block address is 32-bits long. The maximum possible file size is (rounded off to 1 decimal place) _____ GB. **(GATE-2014) (2 Marks)**



Q A file system with 300 Gbyte disk uses a file descriptor with 8 direct block addresses, 1 indirect block address and 1 doubly indirect block address. The size of each disk block is 128 Bytes and the size of each disk block address is 8 Bytes. The maximum possible file size in this file system is
(GATE-2012) (2 Marks)

- (A)** 3 Kbytes
- (B)** 35 Kbytes
- (C)** 280 Bytes
- (D)** Dependent on the size of the disk



Q The data blocks of a very large file in the Unix file system are allocated using
(GATE-2008) (1 Marks)

- (A)** contiguous allocation
- (B)** linked allocation
- (C)** indexed allocation
- (D)** an extension of indexed allocation

Q A Unix-style i-node has 10 direct pointers and one single, one double and one triple indirect pointer. Disk block size is 1 Kbyte, disk block address is 32 bits, and 48-bit integers are used. What is the maximum possible file size? **(GATE-2004) (2 Marks)**

(A) 2^{24} bytes

(B) 2^{32} bytes

(C) 2^{34} bytes

(D) 2^{48} bytes

Q In the index allocation scheme of blocks to a file, the maximum possible size of the file depends on **(GATE-2002) (1 Marks)**

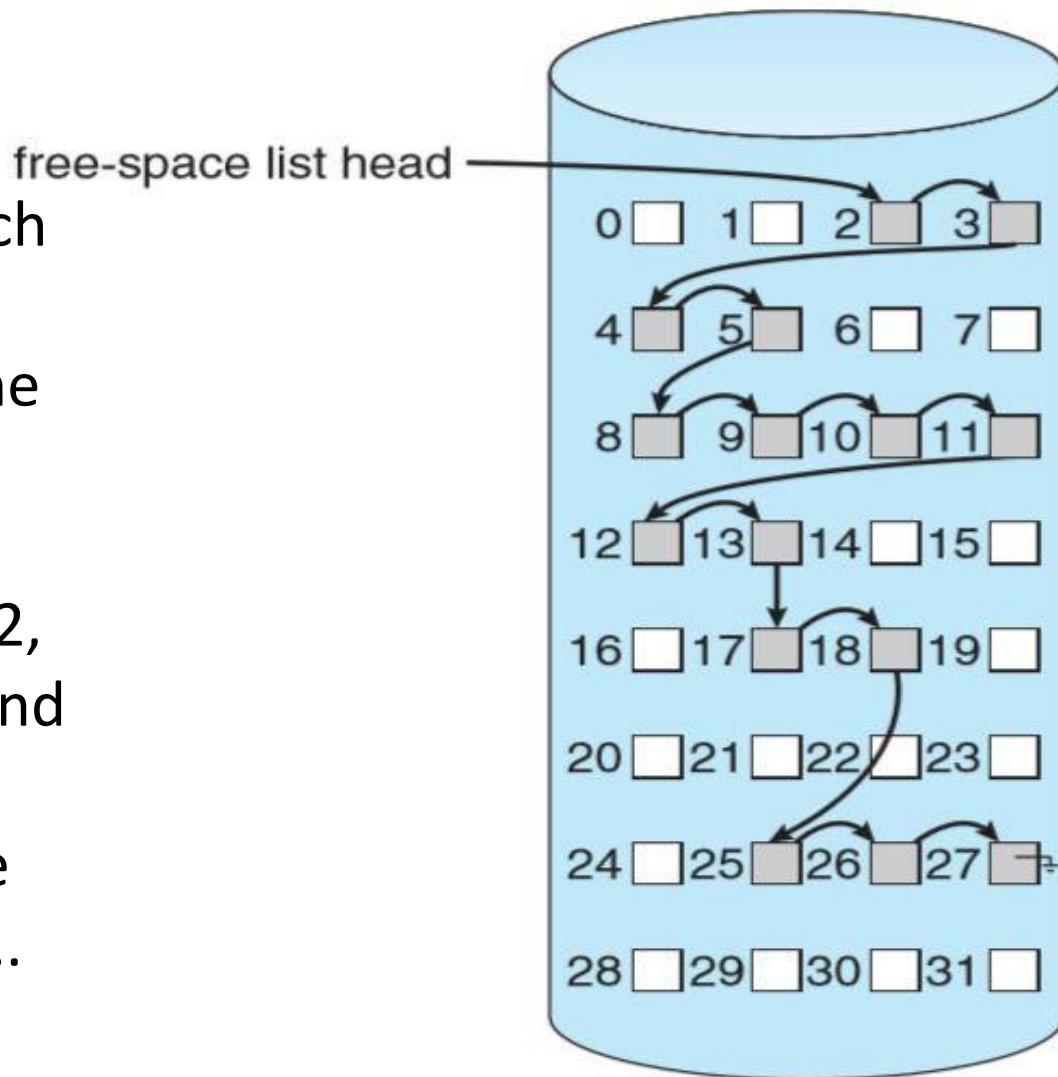
- (a)** the size of the blocks, and the size of the address of the blocks.
- (b)** the number of blocks used for the index, and the size of the blocks.
- (c)** the size of the blocks, the number of blocks used for the index, and the size of the address of the blocks.
- (d)** None of the above

Free-Space Management

- To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks—those not allocated to some file or directory.
- To create a file, we search the free-space list for the required amount of space and allocate that space to the new file.
- This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

Bit Vector

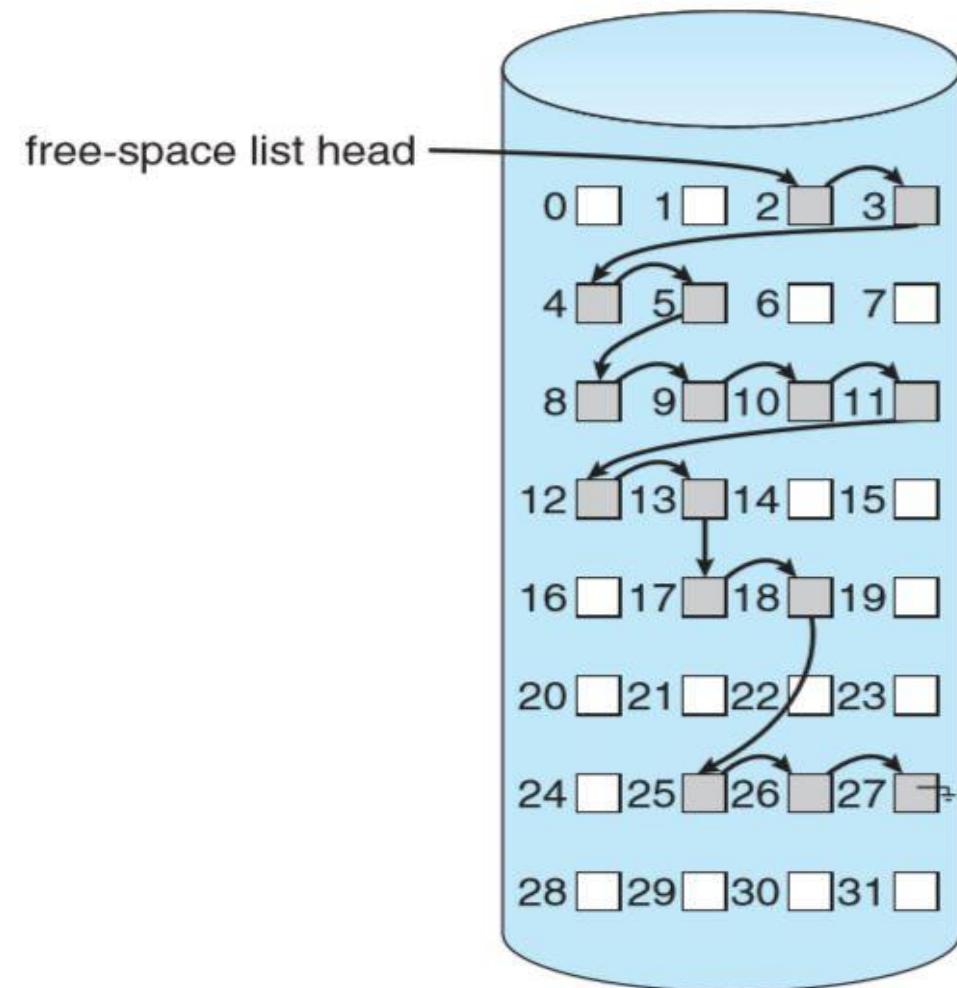
- Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000 ...
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.



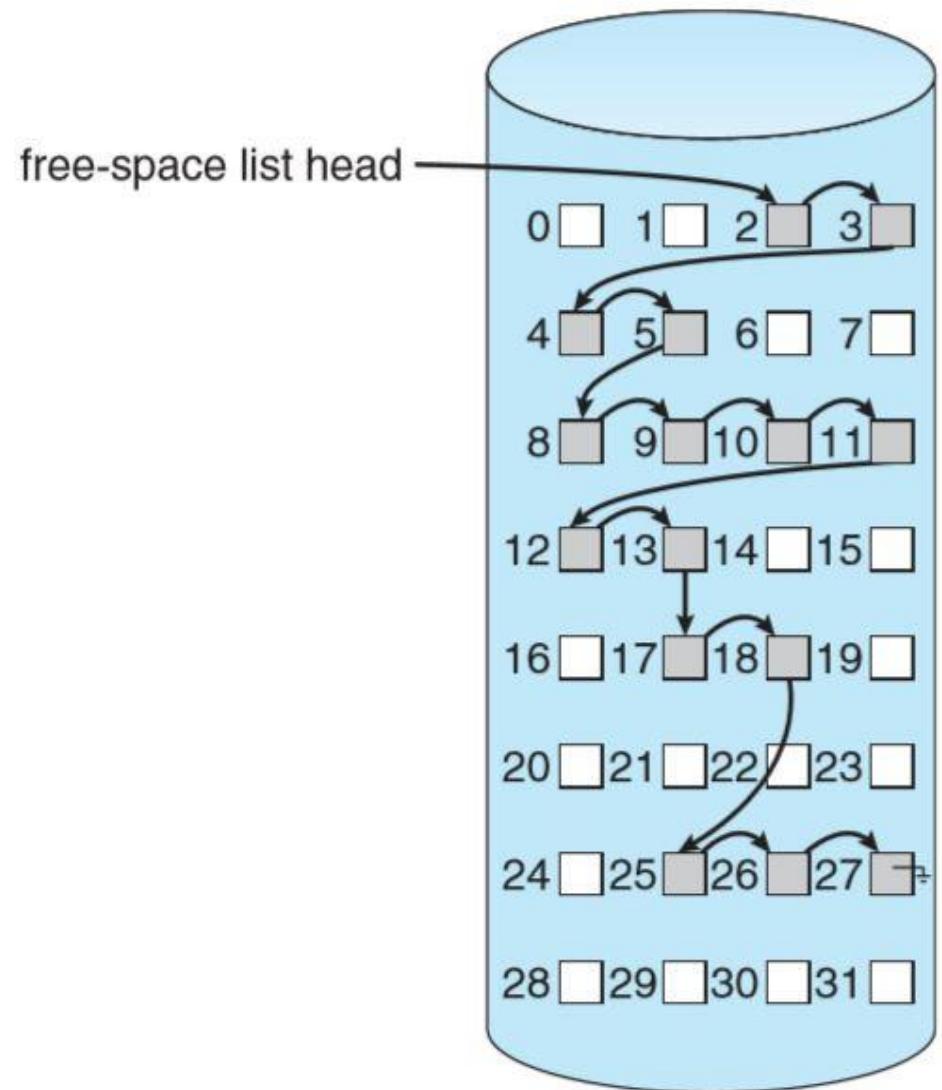
- Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory. Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.
- A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks. A 1-TB disk with 4-KB blocks requires 256 MB to store its bit map. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on. Recall our earlier example, in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.



- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.
- Fortunately, however, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.



Q A FAT (file allocation table) based file system is being used and the total overhead of each entry in the FAT is 4 bytes in size. Given a 100×10^6 bytes disk on which the file system is stored and data block size is 10^3 bytes, the maximum size of a file that can be stored on this disk in units of 10^6 bytes is _____. **(GATE-2014) (2 Marks)**

Q. A disk of size 512M bytes is divided into blocks of 64K bytes. A file is stored in the disk using linked allocation. In linked allocation, each data block reserves 4 bytes to store the pointer to the next data block. The link part of the last data block contains a NULL pointer (also of 4 bytes). Suppose a file of 1M bytes needs to be stored in the disk. Assume, $1K = 2^{10}$ and $1M = 2^{20}$. The amount of space in bytes that will be wasted due to internal fragmentation is _____. (Answer in integer)? **(Gate 2025)**

Q Consider a linear list based directory implementation in a file system. Each directory is a list of nodes, where each node contains the file name along with the file metadata, such as the list of pointers to the data blocks. Consider a given directory foo. Which of the following operations will necessarily require a full scan of foo for successful completion?

(GATE 2021)

- (a)** Creation of a new file in foo
- (b)** Deletion of an existing file from foo
- (c)** Renaming of an existing file in foo
- (d)** Opening of an existing file in foo

Q Consider two file systems A and B , that use contiguous allocation and linked allocation, respectively. A file of size 100 blocks is already stored in A and also in B. Now, consider inserting a new block in the middle of the file (between 50th and 51st block), whose data is already available in the memory. Assume that there are enough free blocks at the end of the file and that the file control blocks are already in memory. Let the number of disk accesses required to insert a block in the middle of the file in A and B are n_A and , n_B respectively, then the value of $n_A + n_B$ is _____. **(GATE 2022) (2 MARKS)**

Fork(Requirement)

- In number of applications specially in those where work is of repetitive nature, like web server i.e. with every client we have to run similar type of code. Have to create a separate process every time for serving a new request.
- So it must be a better solution that instead to creating a new process every time from scratch we must have a short command using which we can do this logic.

- **Idea of fork command**

- Here fork command is a system command using which the entire image of the process can be copied and we create a new process, this idea help us to complete the creation of the new process with speed.
- After creating a process, we must have a mechanism to identify weather in newly created process which one is child and which is parent.

- **Implementation of fork command**

- In general, if fork return 0 then it is child and if fork return 1 then it is parent, and then using a programmer level code we can change the code of child process to behave as new process.

- **Advantages of using fork commands**

- Now it is relatively easy to create and manage similar types of process of repetitive nature with the help of fork command.

- **Disadvantage**

- To create a new process by fork command we have to do system call as, fork is system function
 - Which is slow and time taking
 - Increase the burden over Operating System
- Different image of the similar type of task have same code part which means we have the multiple copy of the same data waiting the main memory

Code

Memory Locations for Code are Determined at Compile Time.

Static Data

Locations of Static Data Can also be Determined at Compile Time.

Stack

Data Objects Allocated at Run-time.
(Activation Records)

Free Memory

Other Dynamically Allocated Data Objects at Run-time. (For Example, Malloc Area in C).

Heap

Q The following C program is executed on a Unix/Linux system:

```
#include <unistd.h>
int main ()
{
    int i ;
    for (i=0; i<10; i++)
        if (i%2 == 0)
            fork () ;
    return 0 ;
}
```

The total number of child processes created is ____ (GATE-2019) (1 Marks)

Q A process executes the code

```
fork();  
fork();  
fork();
```

the total number of child processes created is **(GATE - 2012) (1 Marks)**

a) 3

b) 4

c) 7

d) 8

Q A process executes the following code (GATE - 2008) (2 Marks)

```
for (i=0; i<n; i++)  
fork();
```

- a) n
- b) $(2^n) - 1$
- c) 2^n
- d) $(2^{n+1}) - 1$

Q Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE? **(GATE-2005)(2 Marks)**

- a) $u = x + 10$ and $v = y$
- b) $u = x + 10$ and $v \neq y$
- c) $u + 10 = x$ and $v = y$
- d) $u + 10 = x$ and $v \neq y$

```
if (fork() == 0)
{
    a = a + 5;
    printf ("%d, %d /n", a, &a);
}
else
{
    a = a -5;
    printf("%d, %d /n", a, &a);
}
```

Q A process executes the following segment of code:

```
for(i=1;i<=n;i++)  
    fork();
```

The number of new processes created is (GATE-2004) (1 Marks)

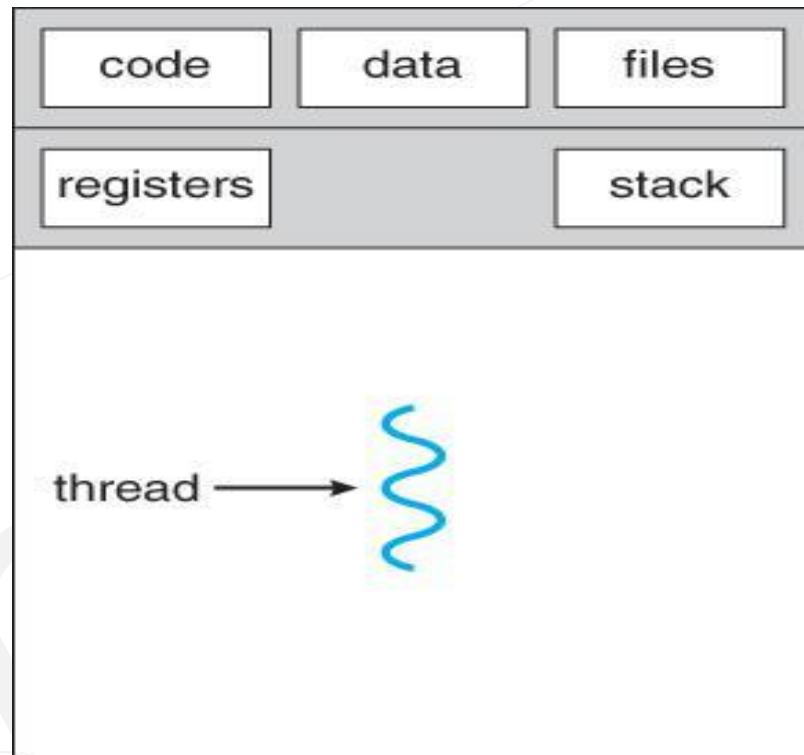
- a) n
- b) $(n(n+1))/2$
- c) $2^n - 1$
- d) $3^n - 1$

Q57. Consider the following code snippet using the fork() and wait() system calls. Assume that the code compiles and runs correctly, and the system calls run successfully without any errors. **(Gate 2024,CS)(2 Marks) (NAT)**

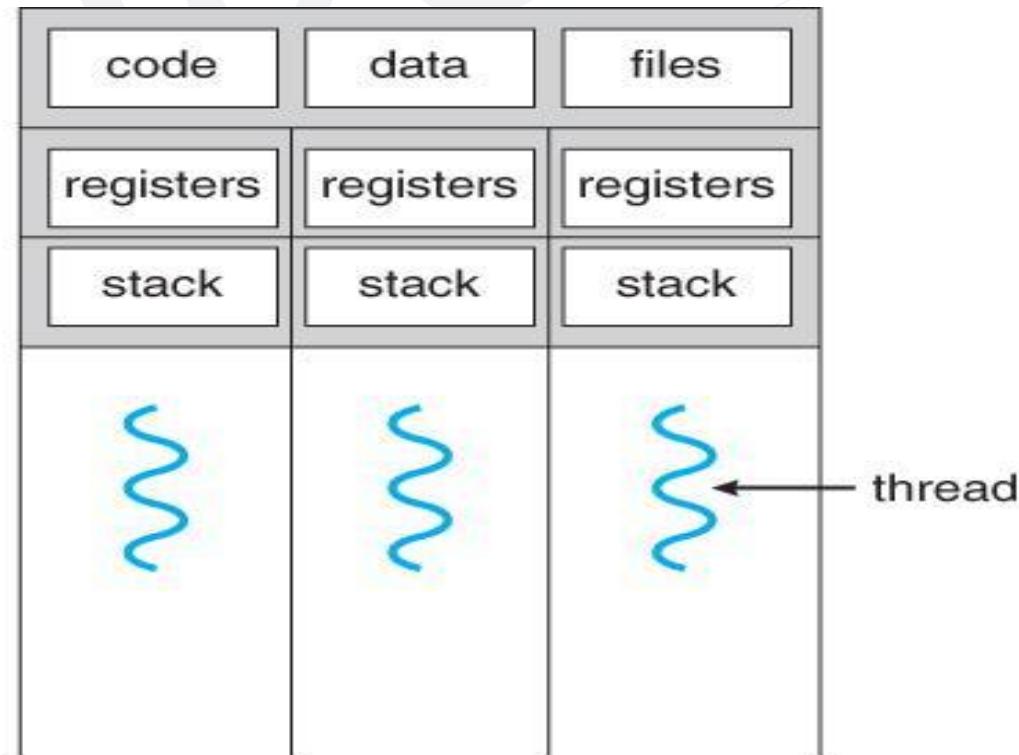
```
int x=3;  
While {x>0}  
{  
    fork ();  
    Printf("hello");  
    wait(NULL);  
    x-- ;  
}
```

The total number of times the printf statement is executed is _____

- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files. A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)



single-threaded process



multithreaded process

Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

Q Which of the following is/are shared by all the threads in a process? (GATE - 2017) (1 Marks)

- I. Program Counter
- II. Stack
- III. Address space
- IV. Registers

(A) I and II only

(B) III only

(C) IV only

(D) III and IV only

Q Threads of a process share (GATE - 2017) (1 Marks)

- (A) global variables but not heap**
- (B) heap but not global variables**
- (C) neither global variables nor heap**
- (D) both heap and global variables**

Q A thread is usually defined as a “light weight process” because an operating system (OS) maintains smaller data structures for a thread than for a process. In relation to this, which of the following is TRUE? **(GATE - 2011) (1 Marks)**

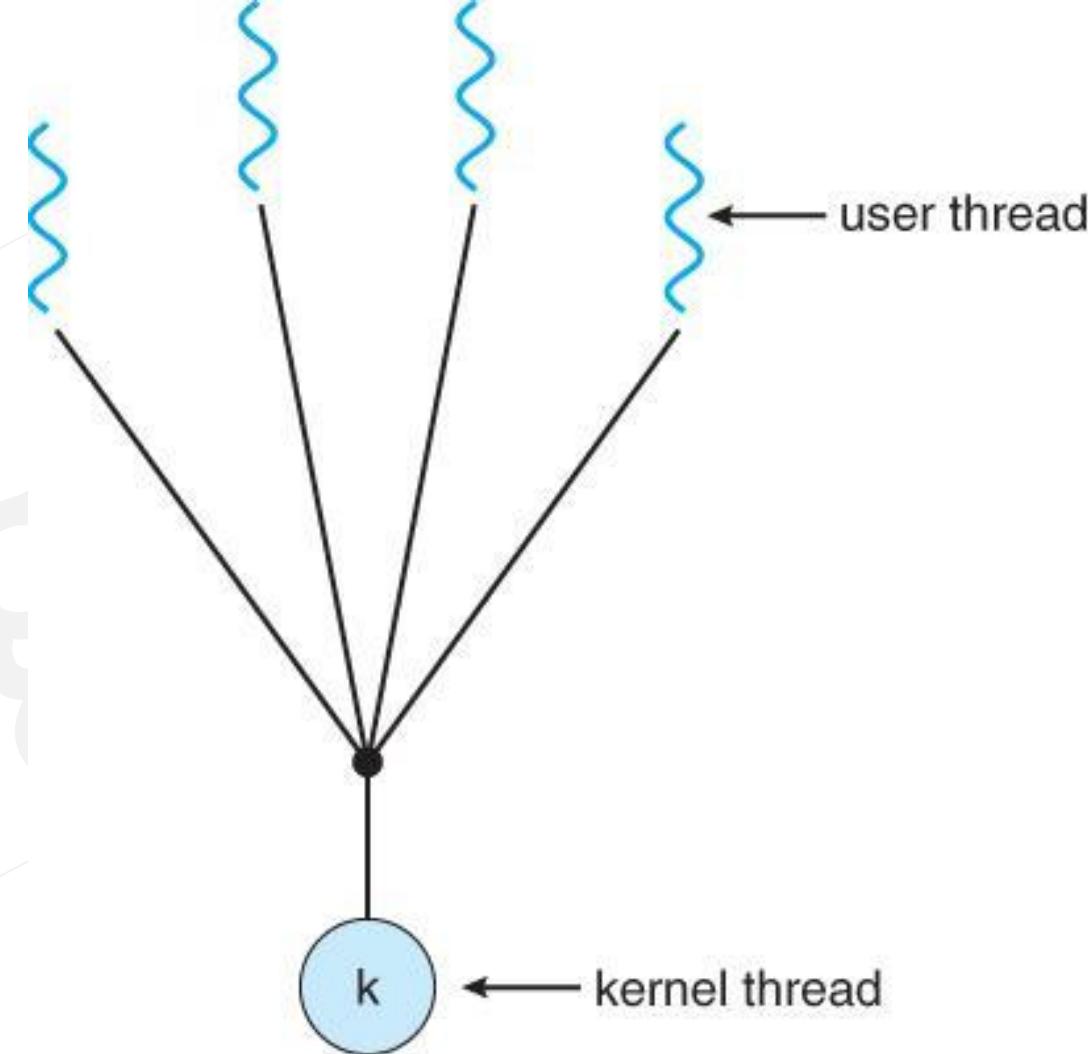
- (A)** On per-thread basis, the OS maintains only CPU register state
- (B)** The OS does not maintain a separate stack for each thread
- (C)** On per-thread basis, the OS does not maintain virtual memory state
- (D)** On per-thread basis, the OS maintains only scheduling and accounting information

Multithreading Models

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

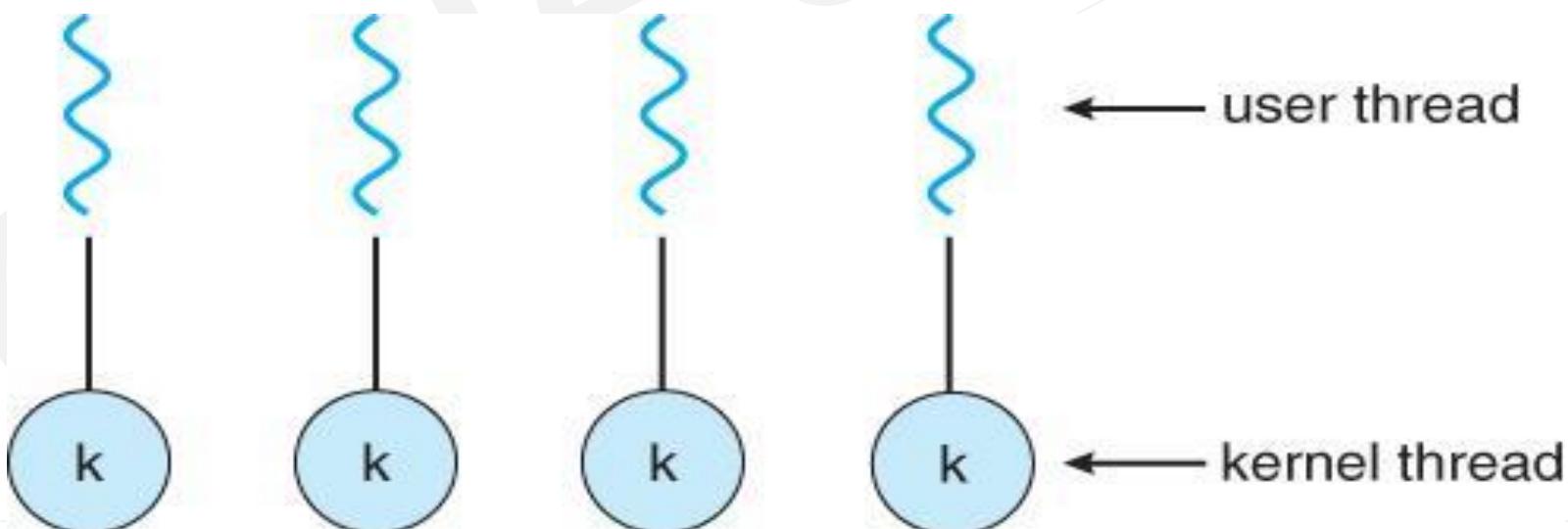
Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris implement the many-to-one model in the past, but few systems continue to do so today.



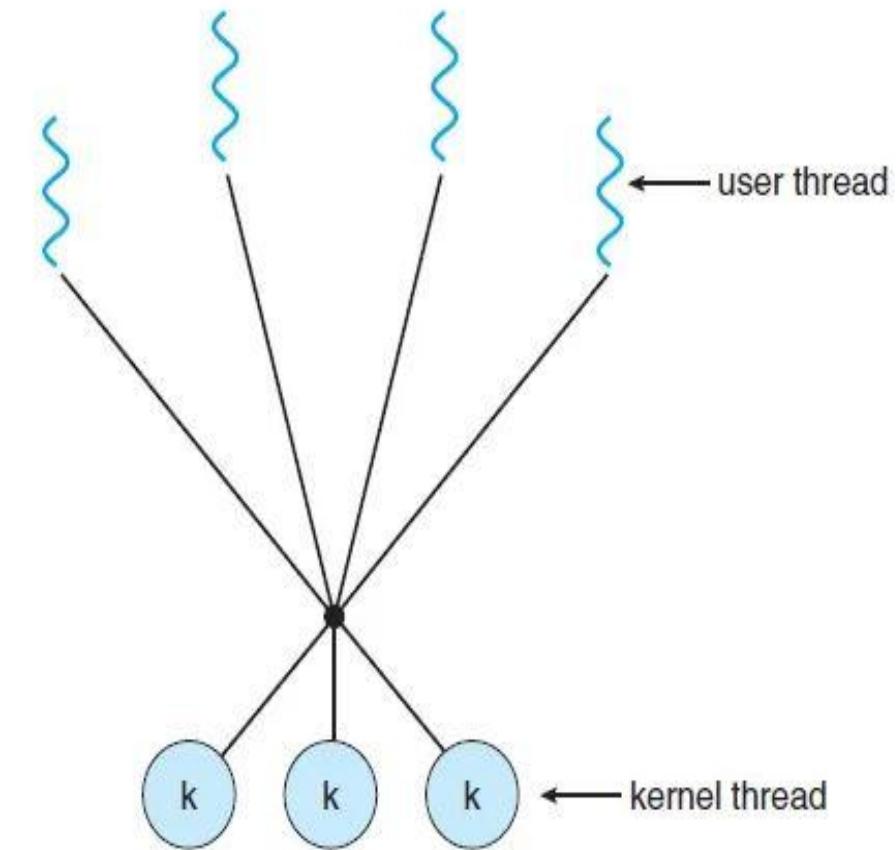
One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread. One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However, the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system. Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created. Blocking kernel system calls do not block the entire process. Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.



Q Which one of the following is FALSE? (GATE - 2014) (1 Marks)

- (A) User level threads are not scheduled by the kernel.
- (B) When a user level thread is blocked, all other threads of its process are blocked.
- (C) Context switching between user level threads is faster than context switching between kernel level threads.
- (D) Kernel level threads cannot share the code segment

Q Let the time taken to switch between user and kernel modes of execution be t_1 , while the time taken to switch between two processes be t_2 . Which of the following is TRUE? (GATE-2011) (1 Marks)

- (A) $t_1 > t_2$
- (B) $t_1 = t_2$
- (C) $t_1 < t_2$
- (D) nothing can be said about the relation between t_1 and t_2

Q Consider the following statements about user level threads and kernel level threads. Which one of the following statements is FALSE? **(GATE - 2007) (1 Marks)**

- A)** Context switch time is longer for kernel level threads than for user level threads.
- B)** User level threads do not need any hardware support.
- C)** Related kernel level threads can be scheduled on different processors in a multi-processor system
- D)** Blocking one kernel level thread blocks all related threads

Q Consider the following statements with respect to user-level threads and kernel supported threads

- i. context switch is faster with kernel-supported threads
- ii. for user-level threads, a system call can block the entire process
- iii. Kernel supported threads can be scheduled independently
- iv. User level threads are transparent to the kernel

Which of the above statements are true? **(GATE-2004) (1 Marks)**

(A) (ii), (iii) and (iv) only

(B) (ii) and (iii) only

(C) (i) and (iii) only

(D) (i) and (ii) only

Q. Which of the following statements about threads is /are TRUE? (Gate 2024,CS) (1 Marks) (MSQ)

- (a) Threads can only be implemented in Kernel space
- (b) Each thread has its own file descriptor table for open files
- (c) All the threads belonging to a process share a common stack
- (d) Threads belonging to a process are by default not protected from each other

Q Which of the following standard C library functions will always invoke a system call when executed from a single-threaded process in a UNIX/Linux operating system? **(GATE 2021) (1 Marks)**

- (a) exit
- (b) malloc
- (c) sleep
- (d) strlen

Q Consider the following multi-threaded code segment (in a mix of C and pseudo-code), invoked by two processes P1 and P2, and each of the processes spawns two threads T₁ and T₂:

```
int x = 0; // global  
Lock L1; // global  
main ()  
{  
    create a thread to execute foo( ); // Thread T1  
    create a thread to execute foo( ); // Thread T2  
    wait for the two threads to finish execution;  
    print(x);  
}
```

```
foo()  
{  
    int y = 0;  
    Acquire L1;  
    x = x + 1;  
    y = y + 1;  
    Release L1;  
    print (y);  
}
```

Which of the following statement(s) is/are correct? (GATE 2021) (2 MARKS)

- a) Both P1 and P2 will print the value of x as 2.
- b) At least one of P1 and P2 will print the value of x as 4.
- c) At least one of the threads will print the value of y as 2.
- d) Both T1 and T2, in both the processes, will print the value of y as 1.