

# Syllabus(Semester Exam)

**Unit – I Introduction :** Operating system and functions, Classification of Operating systems- Batch, Interactive, Time sharing, Real Time System, Multiprocessor Systems, Multiuser Systems, Multiprocess Systems, Multithreaded Systems, Operating System Structure- Layered structure, System Components, Operating System services, Reentrant Kernels, Monolithic and Microkernel Systems.

**Unit – II CPU Scheduling:** Scheduling Concepts, Performance Criteria, Process States, Process Transition Diagram, Schedulers, Process Control Block (PCB), Process address space, Process identification information, Threads and their management, Scheduling Algorithms, Multiprocessor Scheduling. Deadlock: System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock.

**Unit – III Concurrent Processes:** Process Concept, Principle of Concurrency, Producer / Consumer Problem, Mutual Exclusion, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Test and Set operation; Classical Problem in Concurrency- Dining Philosopher Problem, Sleeping Barber Problem; Inter Process Communication models and Schemes, Process generation.

**Unit – IV Memory Management:** Basic bare machine, Resident monitor, Multiprogramming with fixed partitions, Multiprogramming with variable partitions, Protection schemes, Paging, Segmentation, Paged segmentation, Virtual memory concepts, Demand paging, Performance of demand paging, Page replacement algorithms, Thrashing, Cache memory organization, Locality of reference.

**Unit – V I/O Management and Disk Scheduling:** I/O devices, and I/O subsystems, I/O buffering, Disk storage and disk scheduling, RAID. File System: File concept, File organization and access mechanism, File directories, and File sharing, File system implementation issues, File system protection and security.

# Chapters of This Video

**(Chapter-1: Introduction)**- Operating system, Goal & functions, System Components, Operating System services, Classification of Operating systems- Batch, Interactive, Multiprogramming, Multiuser Systems, Time sharing, Multiprocessor Systems, Real Time System.

**(Chapter-2: Operating System Structure)**- Layered structure, Monolithic and Microkernel Systems, Interface, System Call.

**(Chapter-3: Process Basics)**- Process Control Block (PCB), Process identification information, Process States, Process Transition Diagram, Schedulers, CPU Bound and i/o Bound, Context Switch.

**(Chapter-4: CPU Scheduling)**- Scheduling Performance Criteria, Scheduling Algorithms.

**(Chapter-5: Process Synchronization)**- Race Condition, Critical Section Problem, Mutual Exclusion,, Dekker's solution, Peterson's solution, Process Concept, Principle of Concurrency,

**(Chapter-6: Semaphores)**- Classical Problem in Concurrency- Producer/Consumer Problem, Reader-Writer Problem, Dining Philosopher Problem, Sleeping Barber Problem, Test and Set operation.

**(Chapter-7: Deadlock)**- System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock.

**(Chapter-8)**- Fork Command, Multithreaded Systems, Threads and their management

**(Chapter-9: Memory Management)**- Memory Hierarchy, Locality of reference, Multiprogramming with fixed partitions, Multiprogramming with variable partitions, Protection schemes, Paging, Segmentation, Paged segmentation.

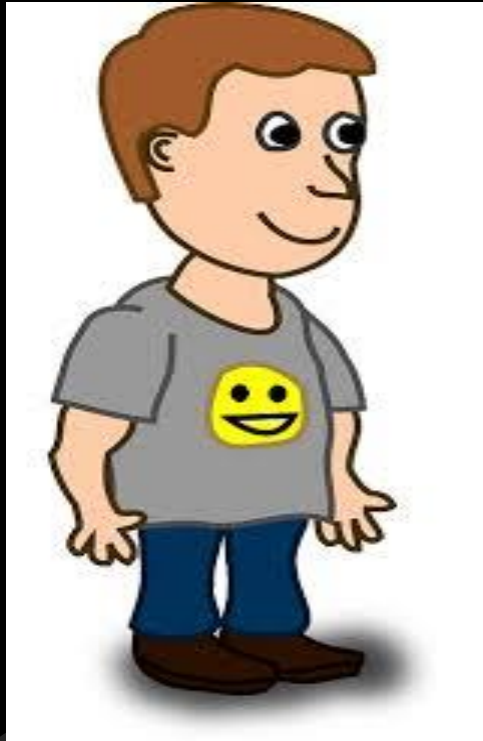
**(Chapter-10: Virtual memory)**- Demand paging, Performance of demand paging, Page replacement algorithms, Thrashing.

**(Chapter-11: Disk Management)**- Disk Basics, Disk storage and disk scheduling, Total Transfer time.

**(Chapter-12: File System)**- File allocation Methods, Free-space Management, File organization and access mechanism, File directories, and File sharing, File system implementation issues, File system protection and security.

# What is Operating System

1. Intermediary – Acts as an intermediary between user & h/w .



# What is Operating System

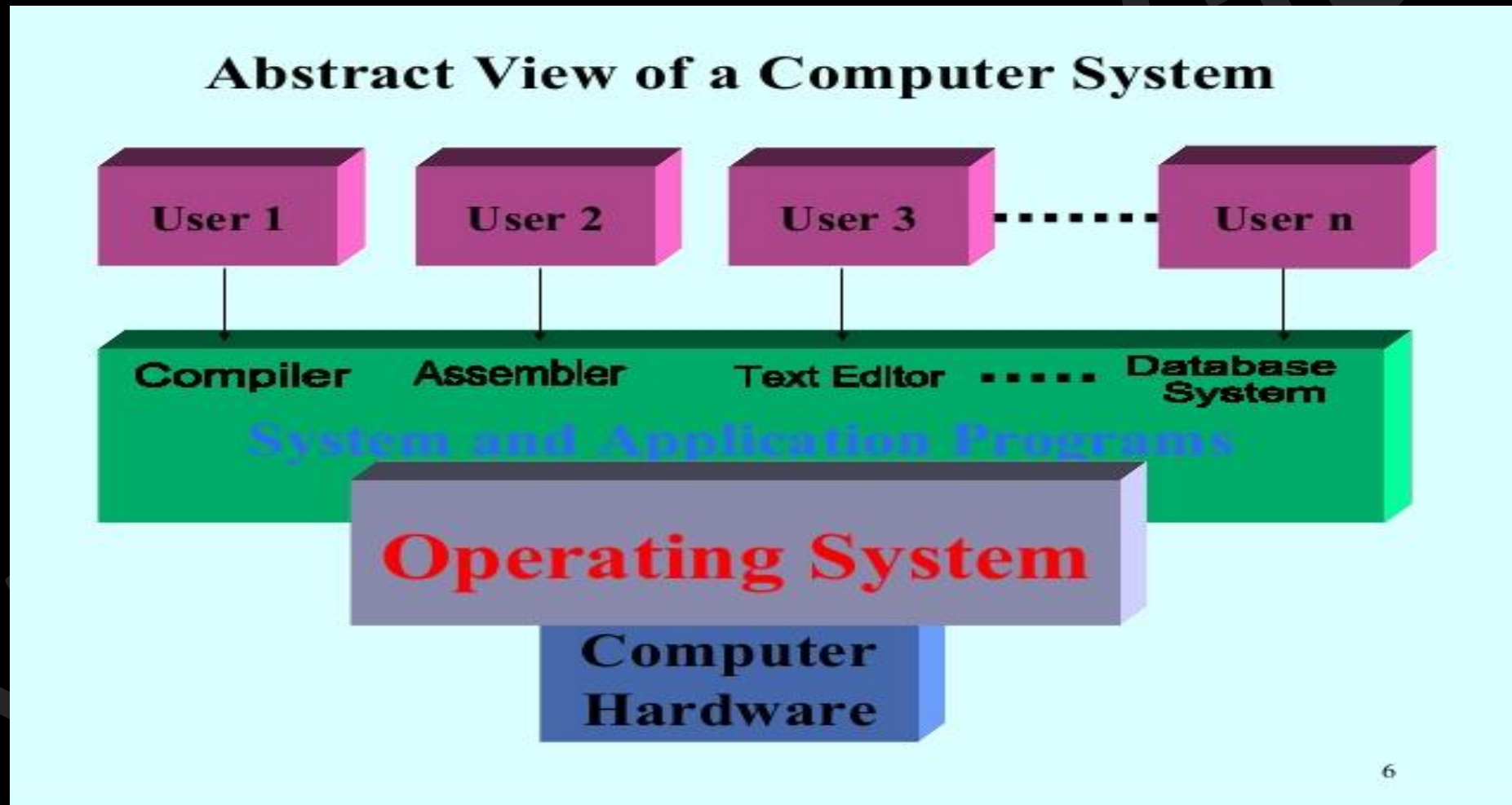
2. Resource Manager/Allocator – Operating system controls and coordinates the use of system resources among various application programs in an unbiased fashion.





# What is Operating System

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



# Example



## Goals and Functions of operating system

- Goals are the ultimate destination, but we follow functions to implement goals.



सबका साथ सबका विकास

Ministries	Departments
58	93

[Knowledge Gate Website](http://www.knowledgegate.com)

# Goals of operating system

1. Primary goals (Convenience / user friendly)
2. Secondary goals (Efficiency (Using resources in efficient manner) / Reliability / maintainability)

# Functions of operating system

1. **Process Management**: Involves handling the creation, scheduling, and termination of processes, which are executing programs.
2. **Memory Management**: Manages allocation and deallocation of physical and virtual memory spaces to various programs.
3. **I/O Device Management**: Handles I/O operations of peripheral devices like disks, keyboards, etc., including buffering and caching.
4. **File Management**: Manages files on storage devices, including their information, naming, permissions, and hierarchy.
5. **Network Management**: Manages network protocols and functions, enabling the OS to establish network connections and transfer data.
6. **Security & Protection**: Ensures system protection against unauthorized access and other security threats through authentication, authorization, and encryption.



# Major Components of operating system

## 1. Kernel

- Central Component: Manages the system's resources and communication between hardware and software.

## 2. Process Management

- Process Scheduler: Determines the execution of processes.
- Process Control Block (PCB): Contains process details such as process ID, priority, status, etc.
- Concurrency Control: Manages simultaneous execution.

## 3. Memory Management

- Physical Memory Management: Manages RAM allocation.
- Virtual Memory Management: Simulates additional memory using disk space.
- Memory Allocation: Assigns memory to different processes.

## 4. File System Management

- File Handling: Manages the creation, deletion, and access of files and directories.
- File Control Block: Stores file attributes and control information.
- Disk Scheduling: Organizes the order of reading or writing to disk.

## 5. Device Management

- Device Drivers: Interface between the hardware and the operating system.
- I/O Controllers: Manage data transfer to and from peripheral devices.

## 6. Security and Access Control

- Authentication: Verifies user credentials.
- Authorization: Controls access permissions to files and directories.
- Encryption: Ensures data confidentiality and integrity.

## 7. User Interface

- Command Line Interface (CLI): Text-based user interaction.
- Graphical User Interface (GUI): Visual, user-friendly interaction with the OS.

## 8. Networking

- Network Protocols: Rules for communication between devices on a network.
- Network Interface: Manages connection between the computer and the network.

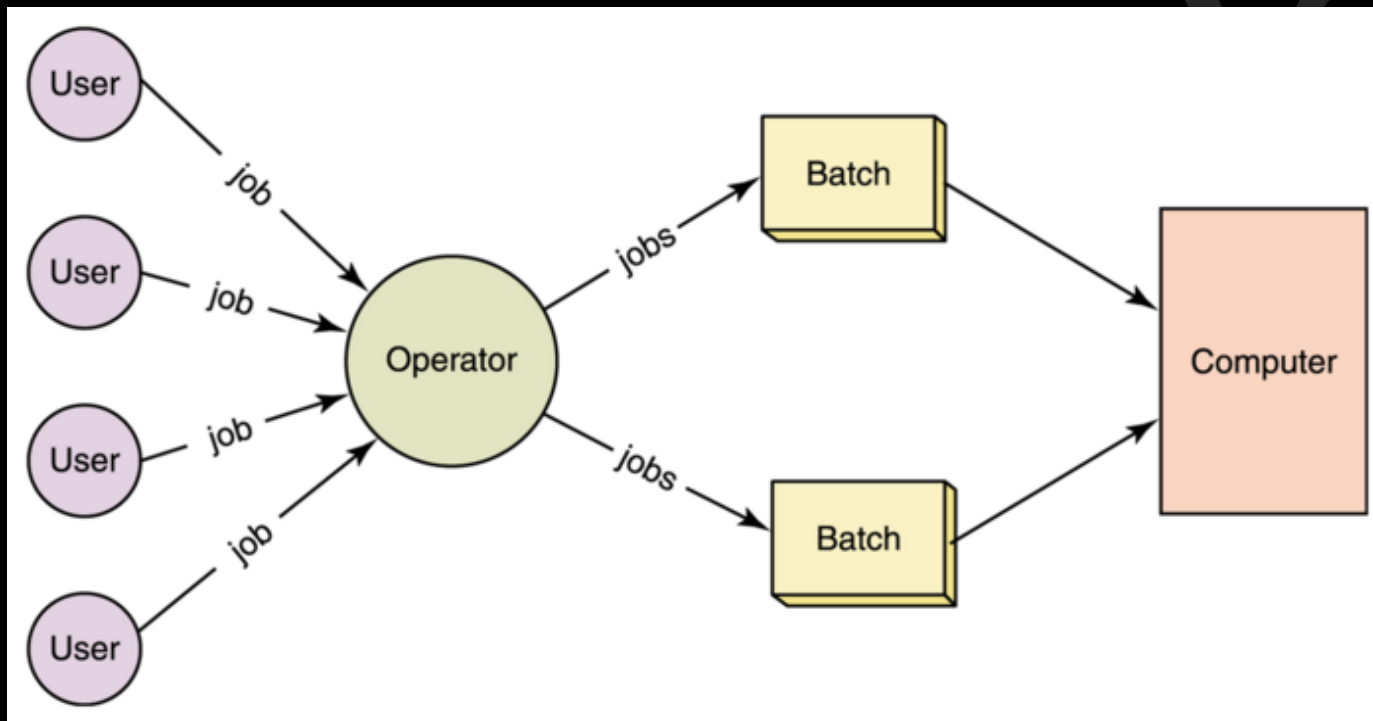
# Batch Operating System

1. Early computers were not interactive device, there user use to prepare a job which consist three parts
  1. Program
  2. Control information
  3. Input data
2. 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.
3. Common input/output device were punch card or tape drives. So these devices were very slow, and processor remain ideal most of the time.



# Batch Operating System

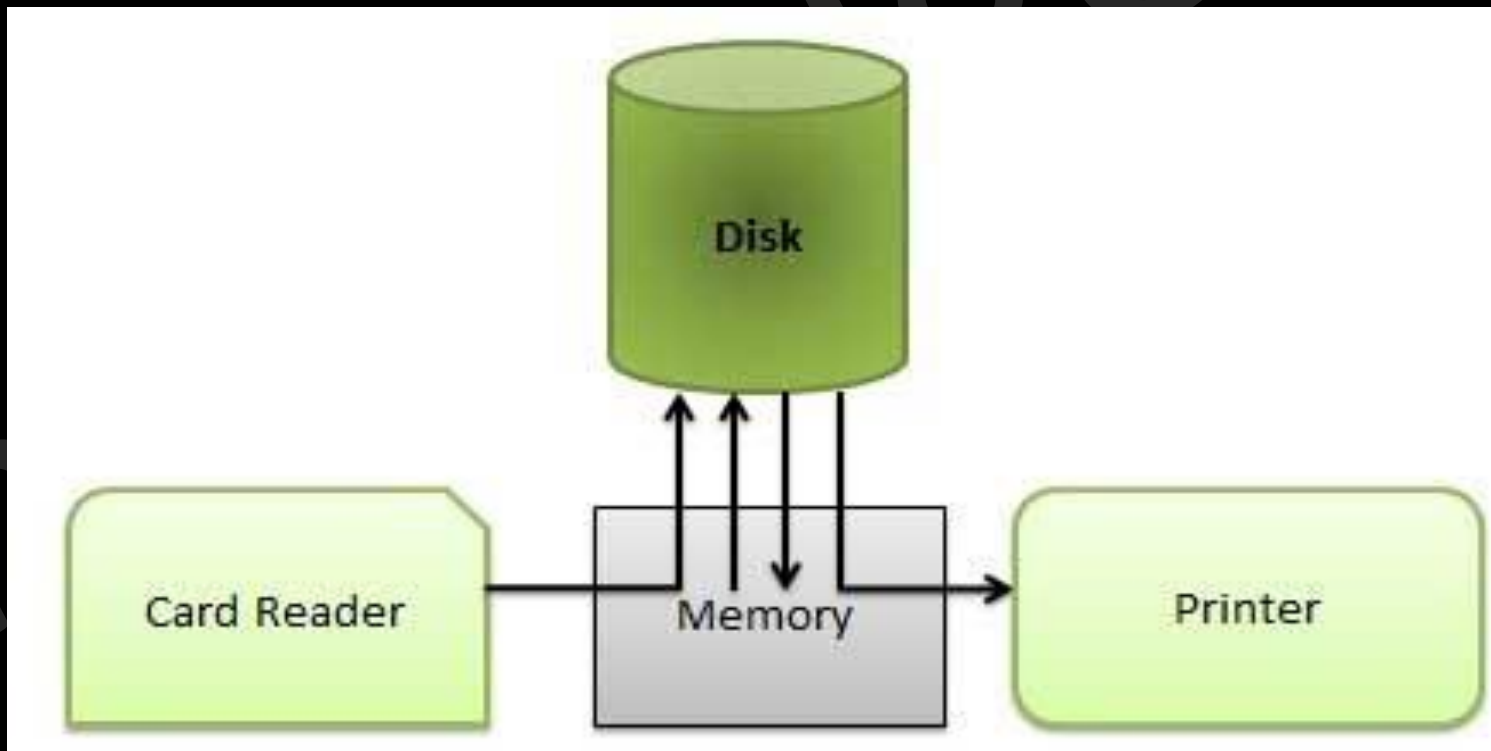
4. To speed up the processing job with similar types (for e.g. FORTRAN jobs, COBOL jobs etc. ) were batched together and were run through the processor as a group (batch).
5. In some system grouping is done by the operator while in some systems it is performed by the 'Batch Monitor' resided in the low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



# Spooling

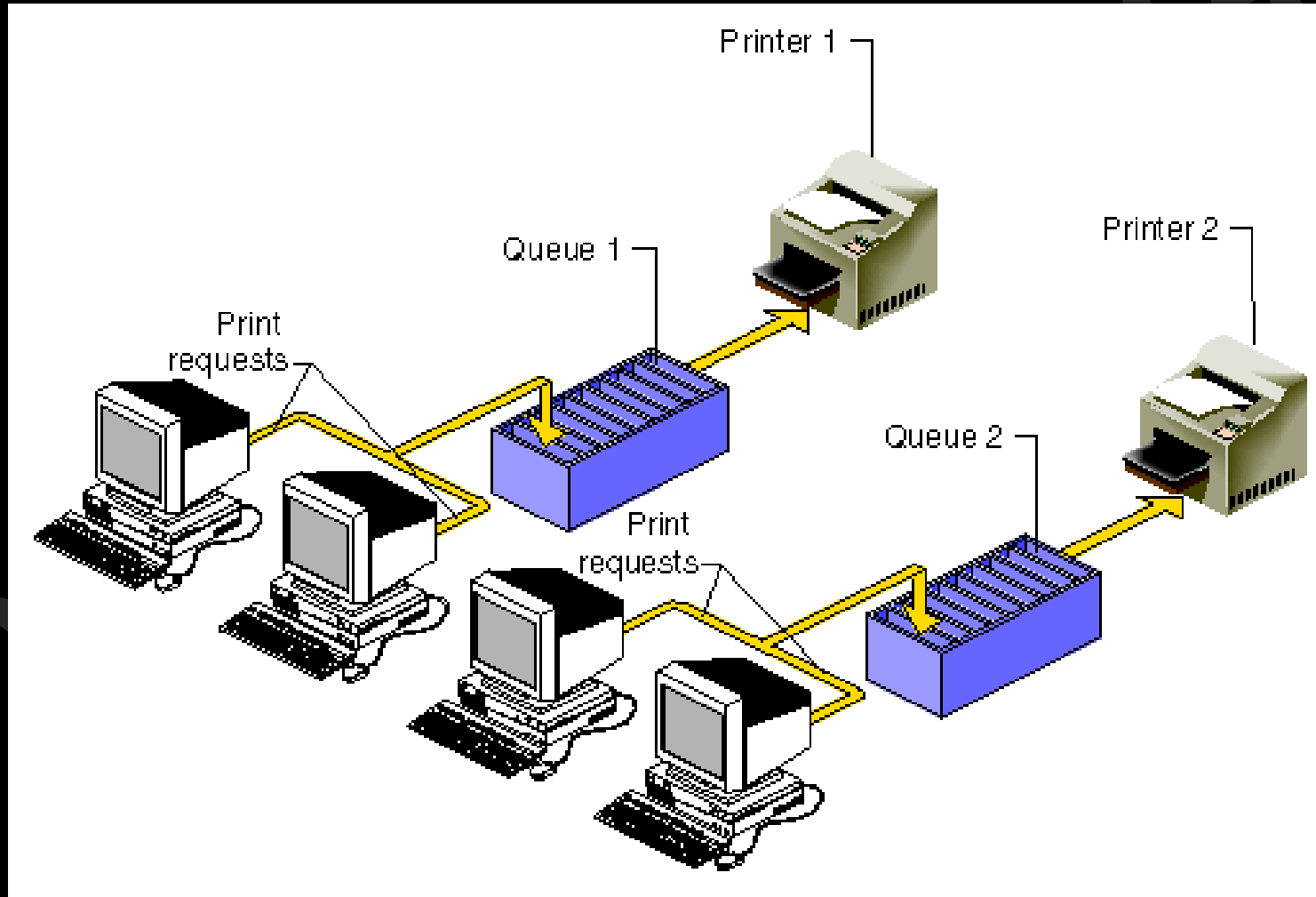
## Simultaneous peripheral operations online

1. In a computer system input-output devices, such as printers are very slow relative to the performance of the rest of the system.
2. Spooling is a process in which data is temporarily held in memory or other volatile storage to be used by a device or a program.





3. 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.



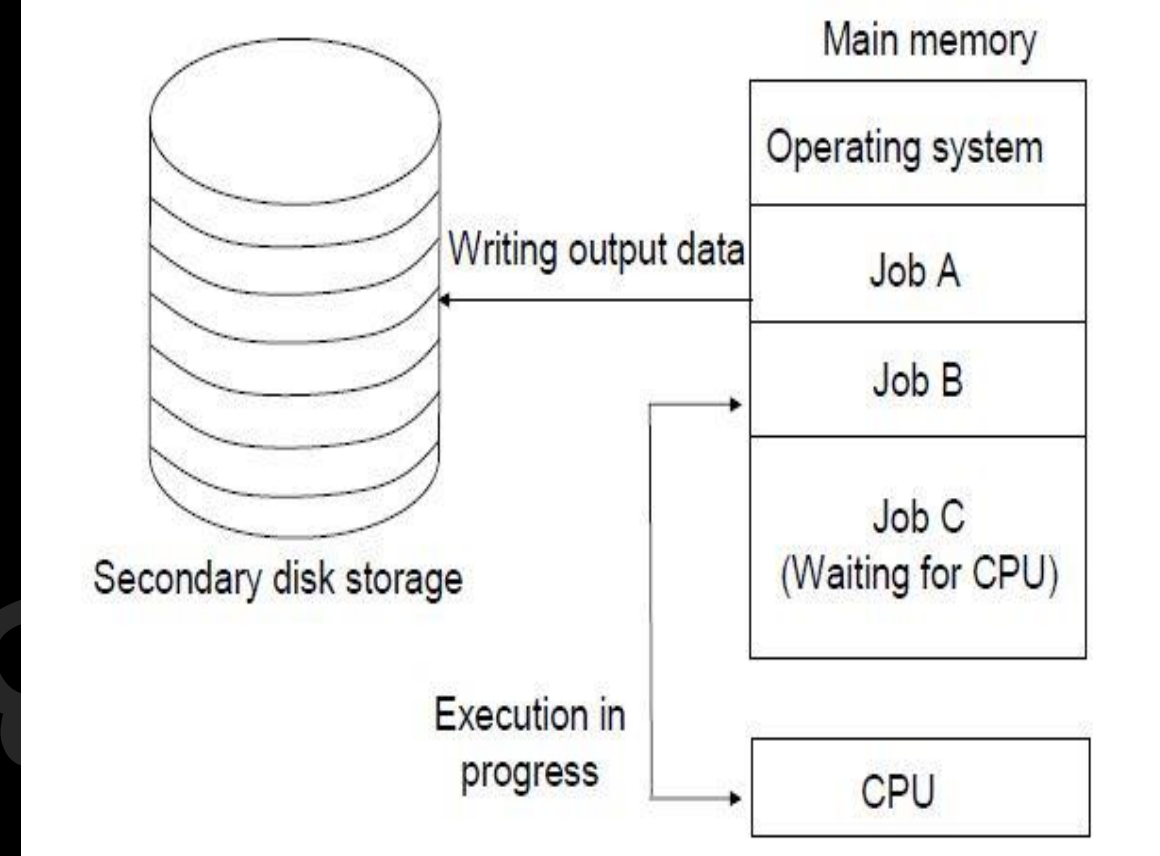
4. Ever had your mouse or keyboard freeze briefly? We often click around to test if it's working. When it unfreezes, all those stored clicks execute rapidly due to the device's spool.



[Knowledge Gate Website](https://www.knowledgegate.com/)

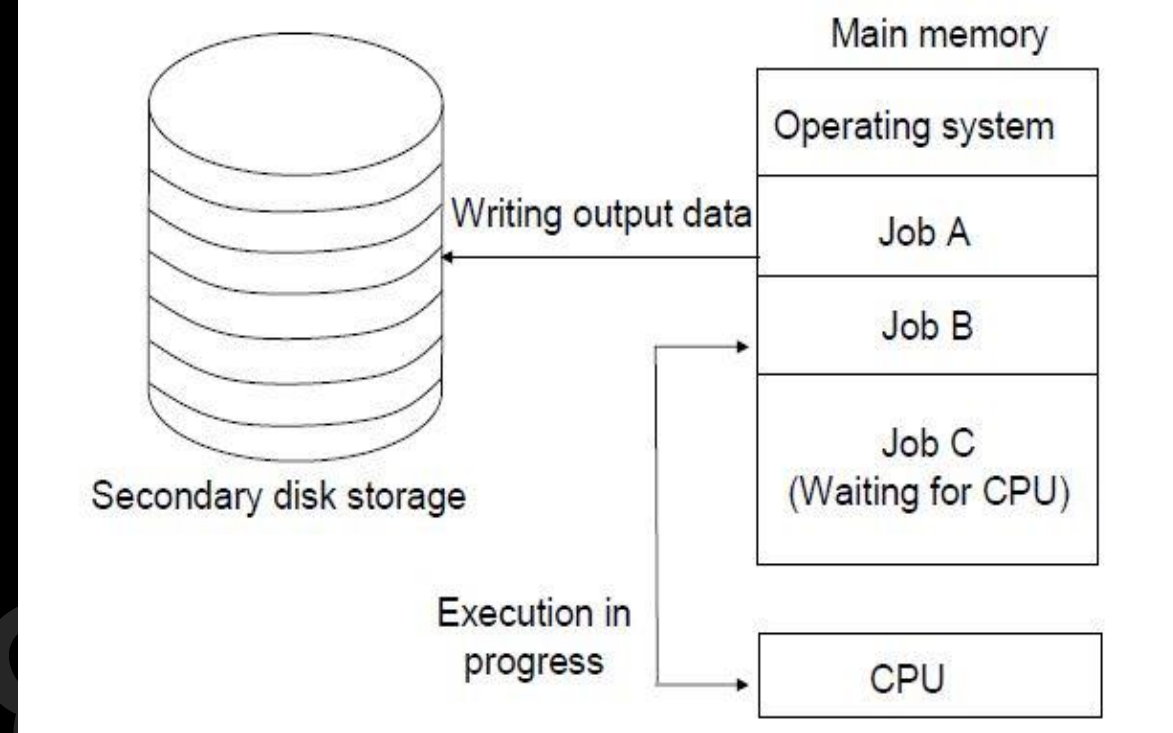
# Multiprogramming Operating System

- **Multiple Jobs:** Keeps several jobs in main memory simultaneously, allowing more efficient utilization of the CPU.
- **Job Execution:** The OS picks and begins to execute one of the jobs in memory.
- **Waiting Jobs:** Eventually, a job may need to wait for a task, such as an I/O operation, to complete.



Processor किसी के लिए wait नहीं करेगा

- **Non-Multiprogrammed:** CPU sits idle while waiting for a job to complete.
- **Multiprogrammed:** The OS switches to and executes another job if the current job needs to wait, utilizing the CPU effectively.



*Show must go on*

- **Conclusion**
- **Efficient Utilization:** Ensures that the CPU is never idle as long as at least one job needs to execute, leading to better utilization of resources.

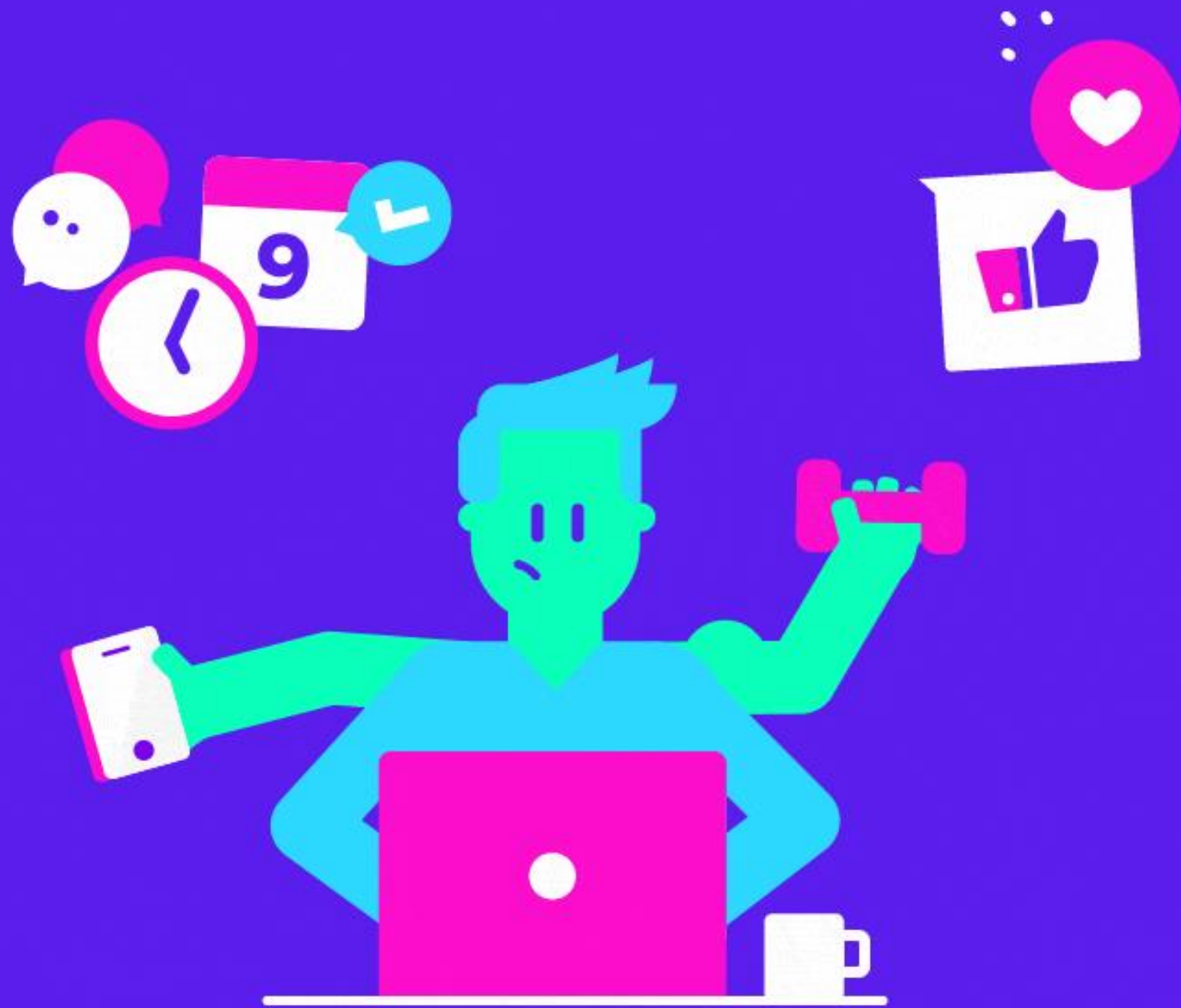
- **Advantages:**
  - High CPU Utilization: Enhances processing efficiency.
  - Less Waiting Time: Minimizes idle time.
  - Multi-Task Handling: Manages concurrent tasks effectively.
  - Shared CPU Time: Increases system efficiency.
- **Disadvantages:**
  - Complex Scheduling: Difficult to program.
  - Complex Memory Management: Intricate handling of memory is required.



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

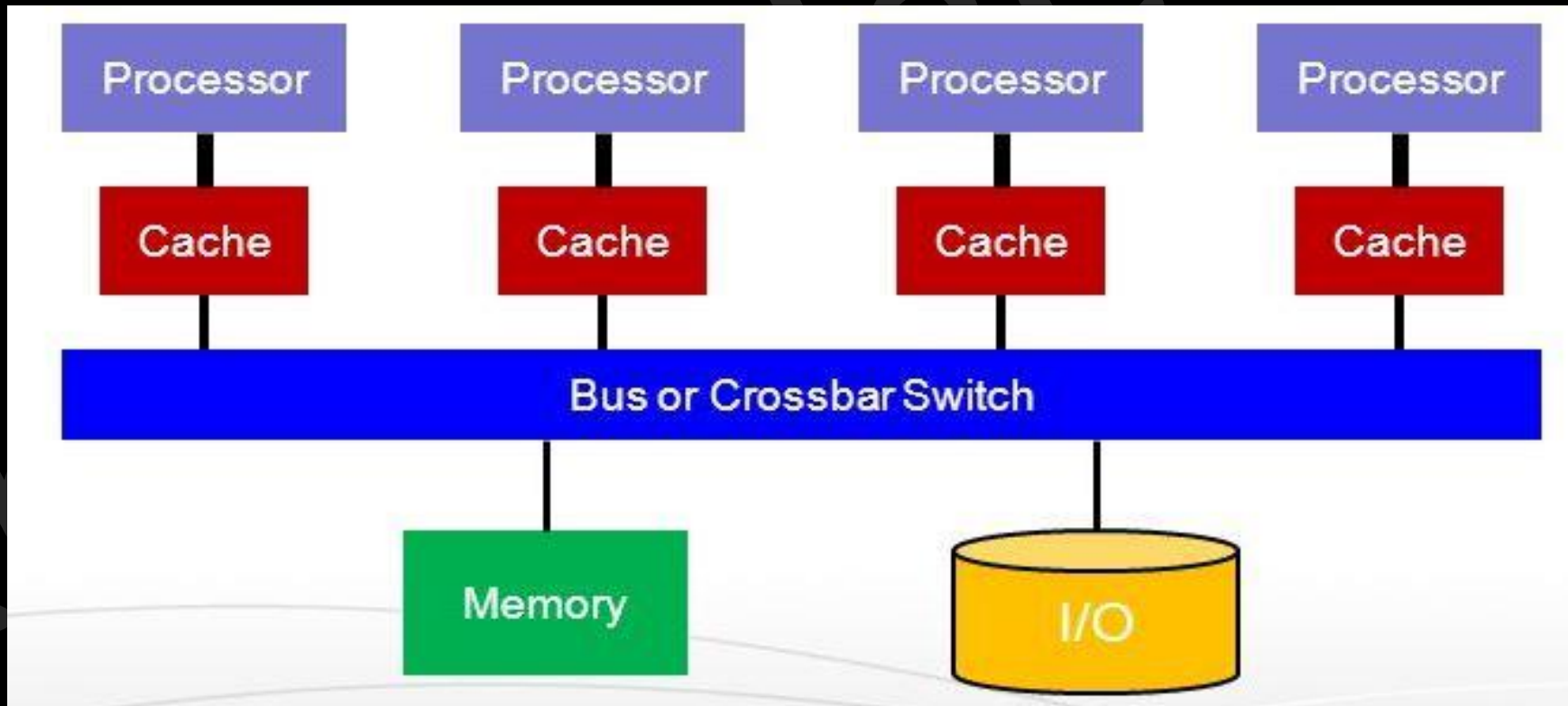
1. Time sharing (or multitasking) is a logical extension of multiprogramming, it allows many users to share the computer simultaneously. the CPU executes multiple jobs (May belong to different user) by switching among them, but the switches occur so frequently that, each user is given the impression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





## Multiprocessing Operating System/ tightly coupled system

1. Multiprocessor Operating System refers to the use of two or more central processing units (CPU) within a single computer system. These multiple CPU's share system bus, memory and other peripheral devices.
2. Multiple concurrent processes each can run on a separate CPU, here we achieve a true parallel execution of processes.
3. Becomes most important in computer system, where the complexity of the job is more, and CPU divides and conquers the jobs. Generally used in the fields like artificial intelligence and expert system, image processing, weather forecasting etc.



<b>Point</b>	<b>Symmetric Processing</b>	<b>Asymmetric Processing</b>
<b>Definition</b>	All processors are treated equally and can run any task.	Each processor is assigned a specific task or role.
<b>Task Allocation</b>	Any processor can perform any task.	Tasks are divided according to processor roles.
<b>Complexity</b>	Generally simpler as all processors are treated the same.	More complex due to the dedicated role of each processor.
<b>Scalability</b>	Easily scalable by adding more processors.	May require reconfiguration as processors are added.
<b>Performance</b>	Load is evenly distributed, enhancing performance.	Performance may vary based on the specialization of tasks.

Point	Multi-Programming	Multi-Processing
<b>Definition</b>	Allows multiple programs to share a single CPU.	Utilizes multiple CPUs to run multiple processes concurrently.
<b>Concurrency</b>	Simulates concurrent execution by rapidly switching between tasks.	Achieves true parallel execution of processes.
<b>Resource Utilization</b>	Maximizes CPU utilization by keeping it busy with different tasks.	Enhances performance by allowing tasks to be processed simultaneously.
<b>Hardware Requirements</b>	Requires only one CPU and manages multiple tasks on it.	Requires multiple CPUs, enabling parallel processing.
<b>Complexity and Coordination</b>	Less complex, primarily managing task switching on one CPU.	More complex, requiring coordination among multiple CPUs.

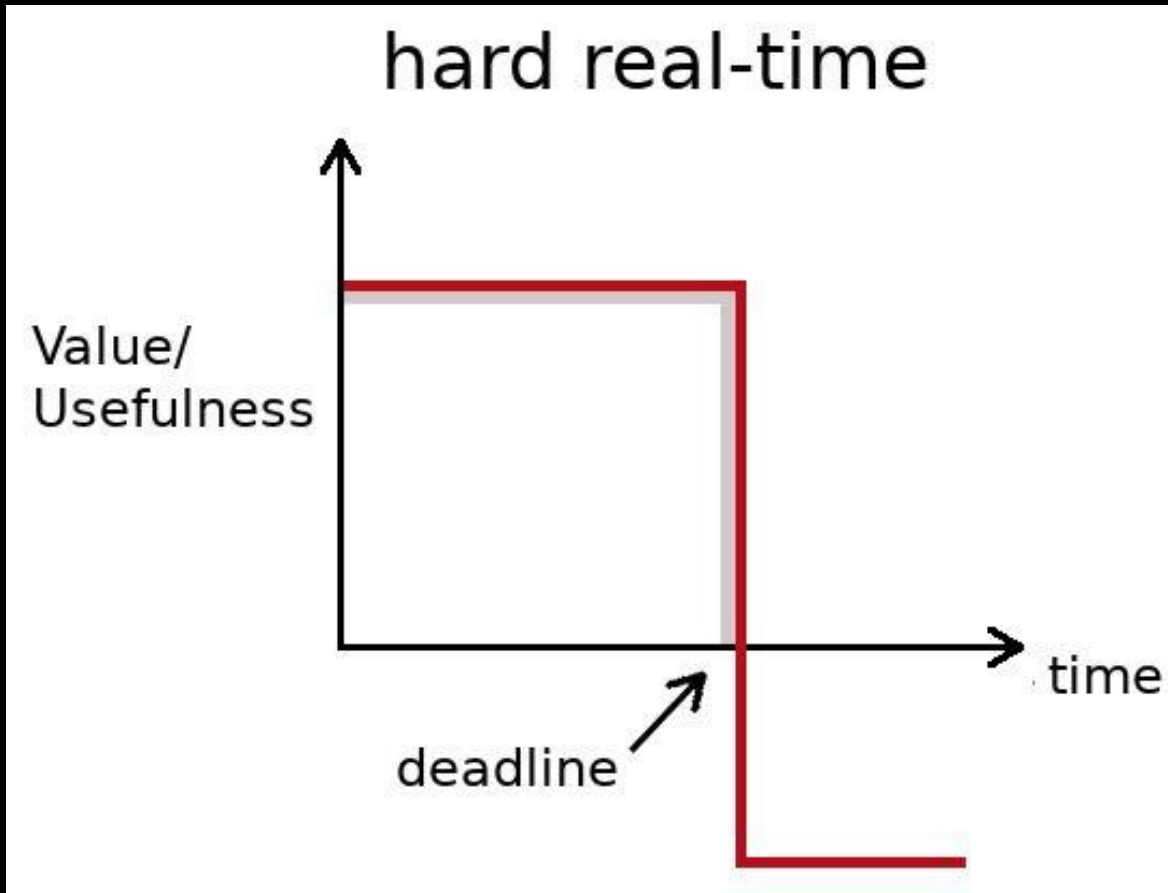


# Real time Operating system

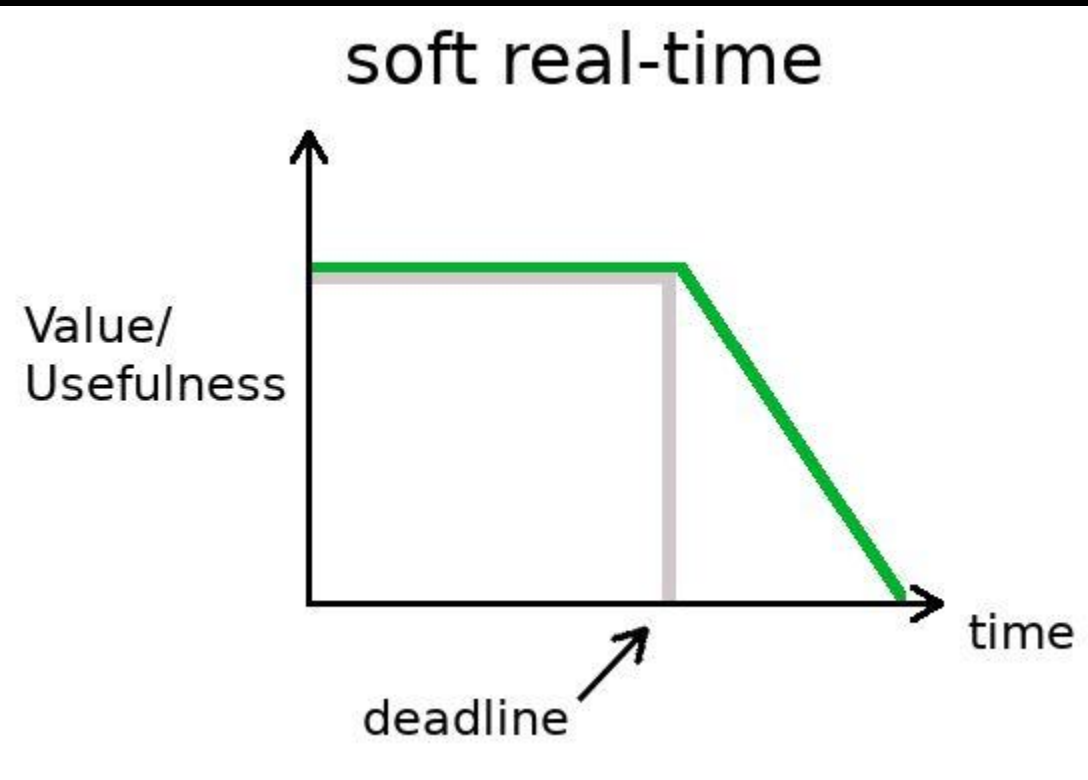
1. A real time operating system is a special purpose operating system which has well defined fixed time constraints. Processing must be done within the defined time limit or the system will fail.
2. Valued more for how quickly or how predictably it can respond, without buffer delays than for the amount of work it can perform in a given period of time.
3. For example, a petroleum refinery, Airlines reservation system, Air traffic control system, Systems that provide up to the minute information on stock prices, Defense application systems like as RADAR.



- **Hard real-time operating system** - This is also a type of OS and it is predicted by a deadline. The predicted deadlines will react at a time  $t = 0$ . Some examples of this operating system are air bag control in cars.



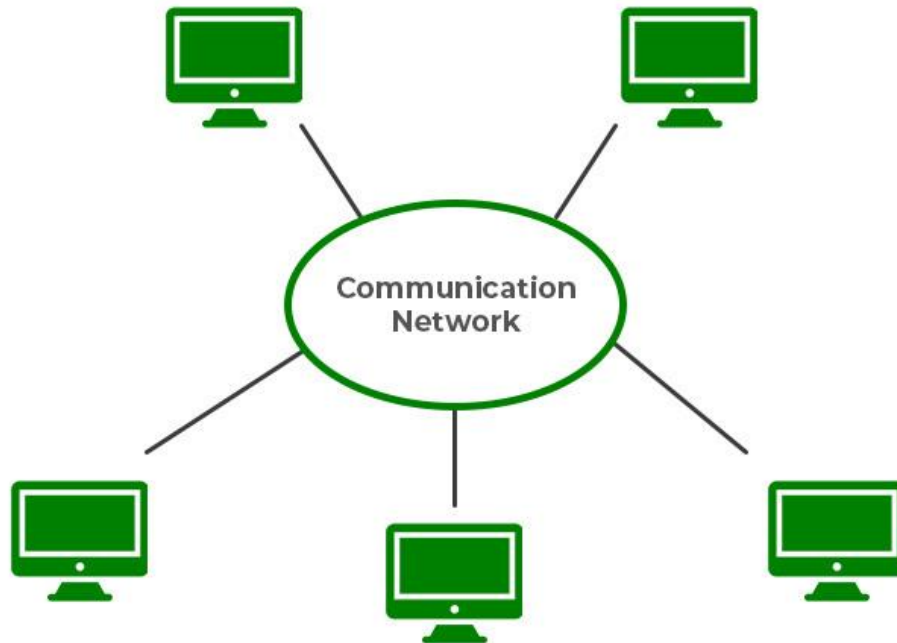
- **Soft real-time operating system** - The soft real-time operating system has certain deadlines, may be missed and they will take the action at a time  $t=0+$ . The critical time of this operating system is delayed to some extent. The examples of this operating system are the digital camera, mobile phones and online data etc.



<b>Point</b>	<b>Hard Real-Time Operating System</b>	<b>Soft Real-Time Operating System</b>
<b>Deadline Constraints</b>	Must meet strict deadlines without fail.	Can miss deadlines occasionally without failure.
<b>Response Time</b>	Fixed and guaranteed.	Predictable, but not guaranteed.
<b>Applications</b>	Used in life-critical systems like medical devices, nuclear reactors.	Used in multimedia, user interfaces, etc.
<b>Complexity and Cost</b>	Typically more complex and costlier.	Less complex and usually less expensive.
<b>Reliability</b>	Must be highly reliable and fault-tolerant.	High reliability desired, but some failures are tolerable.

# Distributed OS

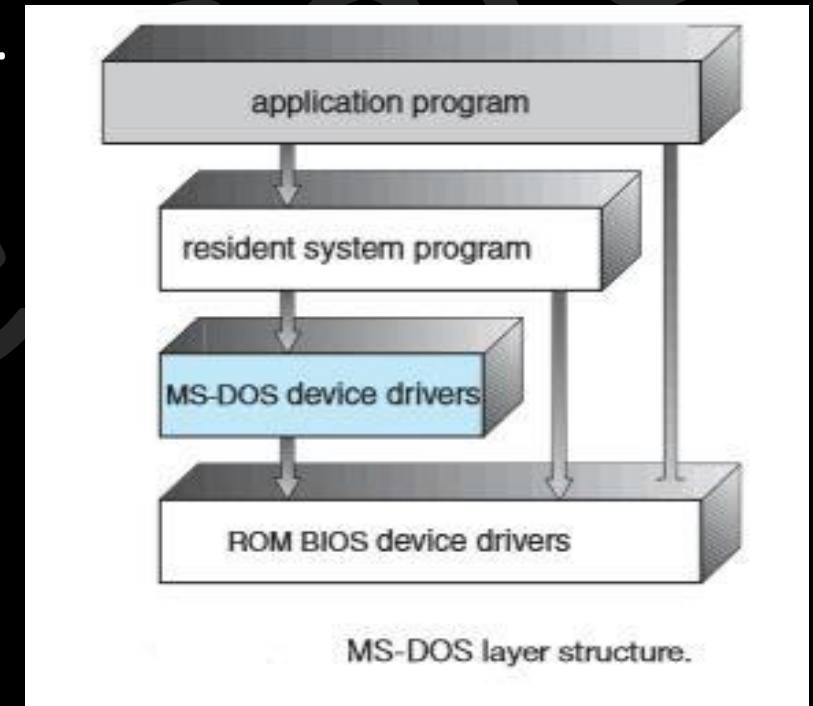
1. A **distributed operating system** is a software over a collection of independent, networked, communicating, loosely coupled nodes and physically separate computational nodes.
2. The nodes communicate with one another through various networks, such as high-speed buses and the Internet. They handle jobs which are serviced by multiple CPUs. Each individual node holds a specific software subset of the global aggregate operating system.
3. There are four major reasons for building distributed systems: resource sharing, computation speedup, reliability, and communication.





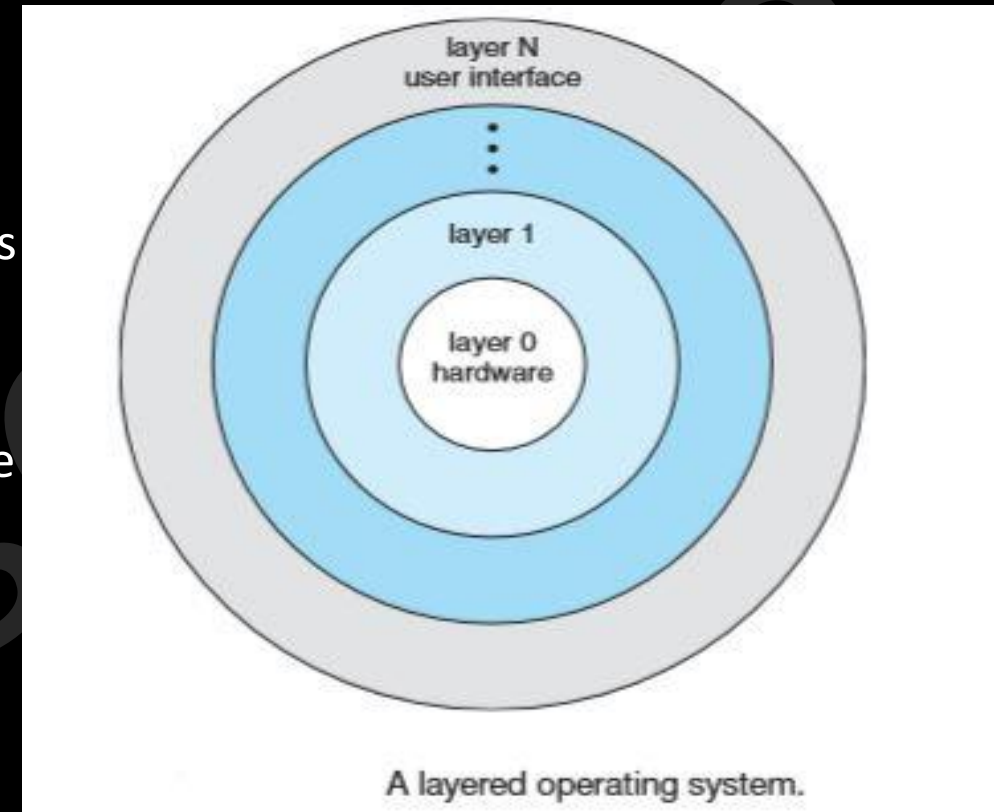
# Structure of Operating System

- 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.
- Not divided into modules. Its interface, levels and functionality are not well separated



- **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.

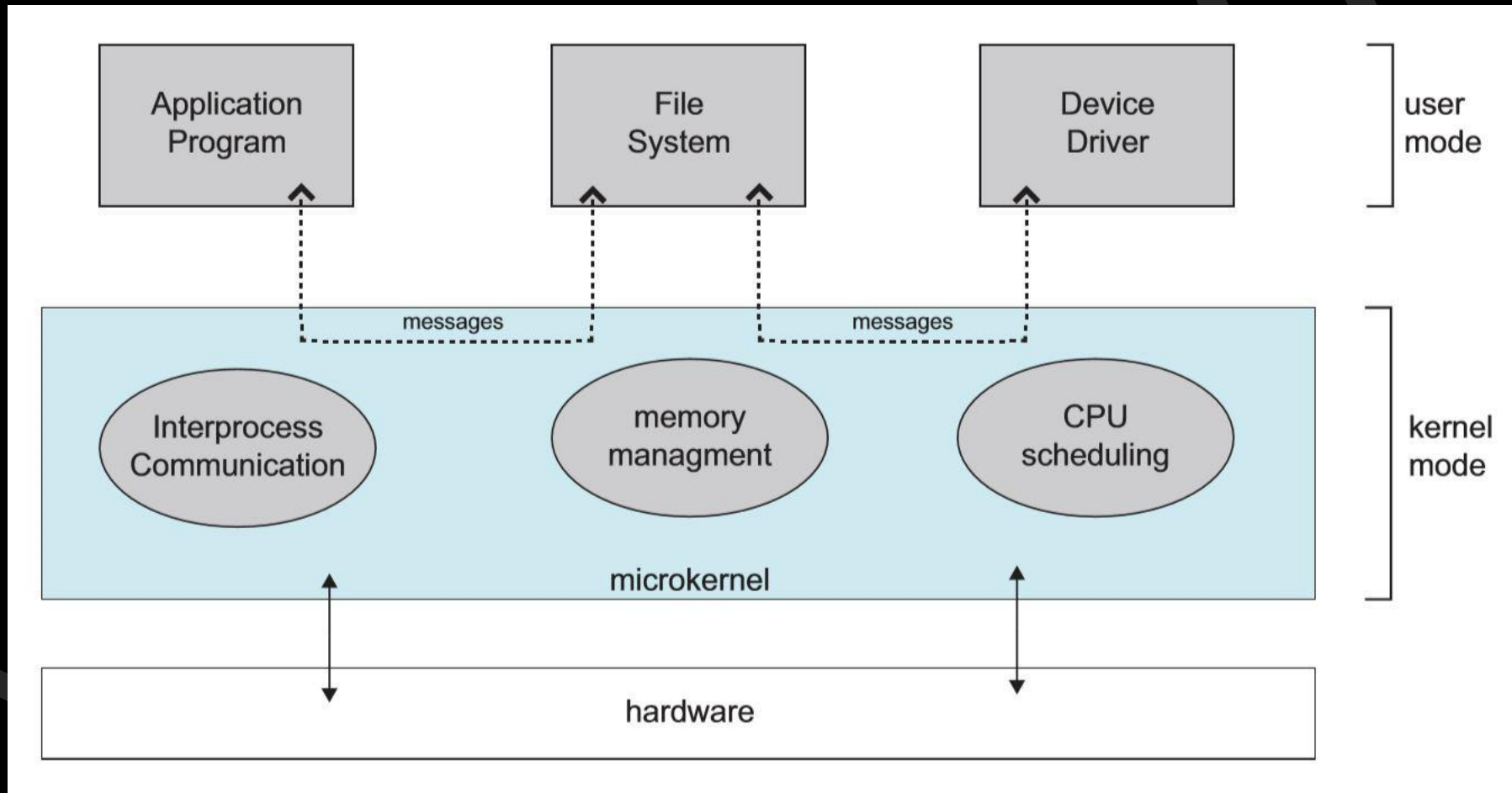
1. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems.
2. Under a top-down approach, the overall functionality and features are determined and are separated into components.
3. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit.
4. 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.



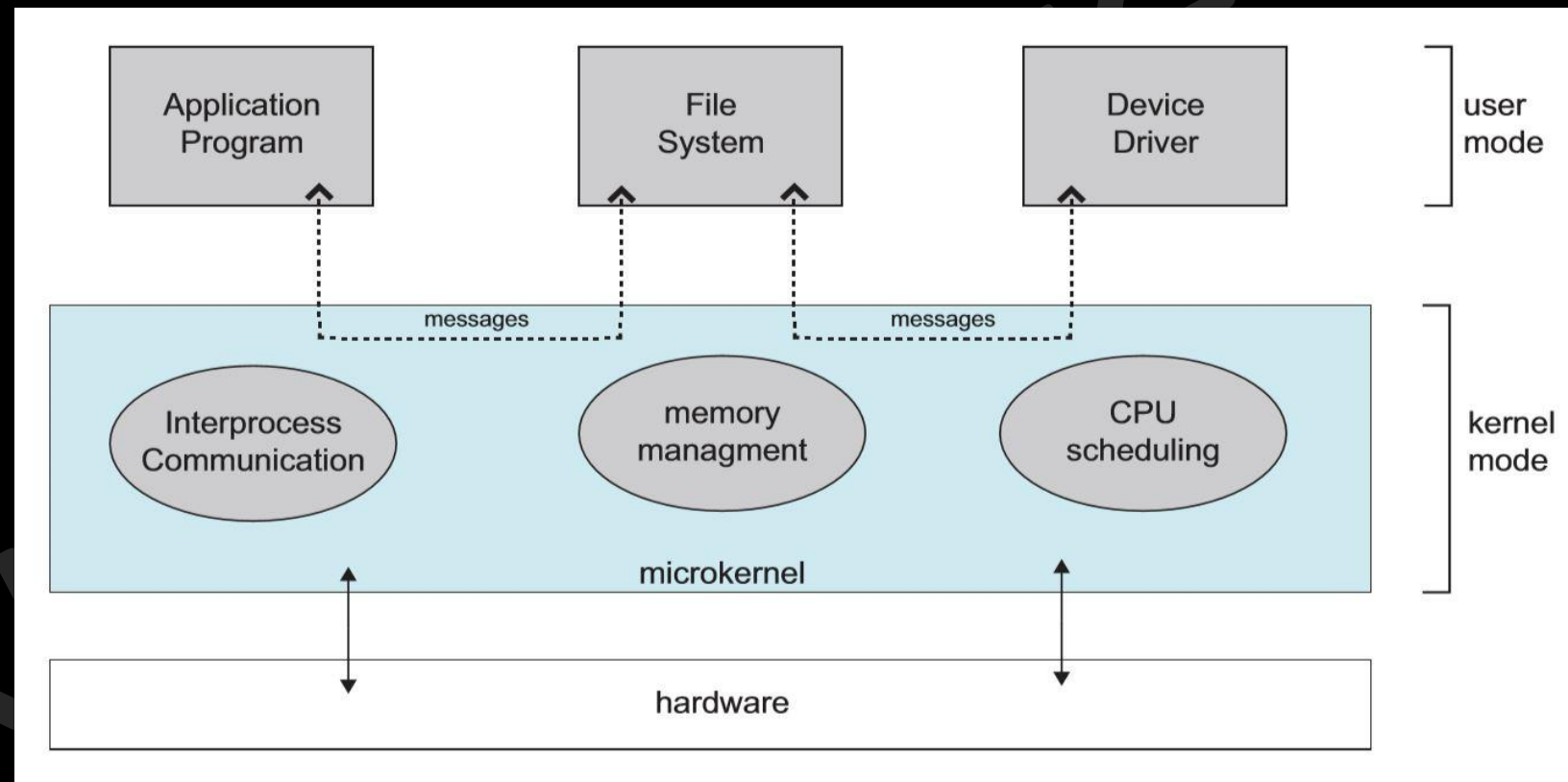


## Micro-Kernel approach

- 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.

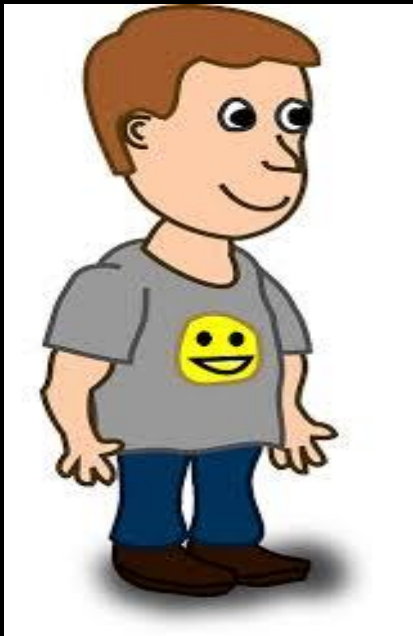


- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a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



# User and Operating-System Interface

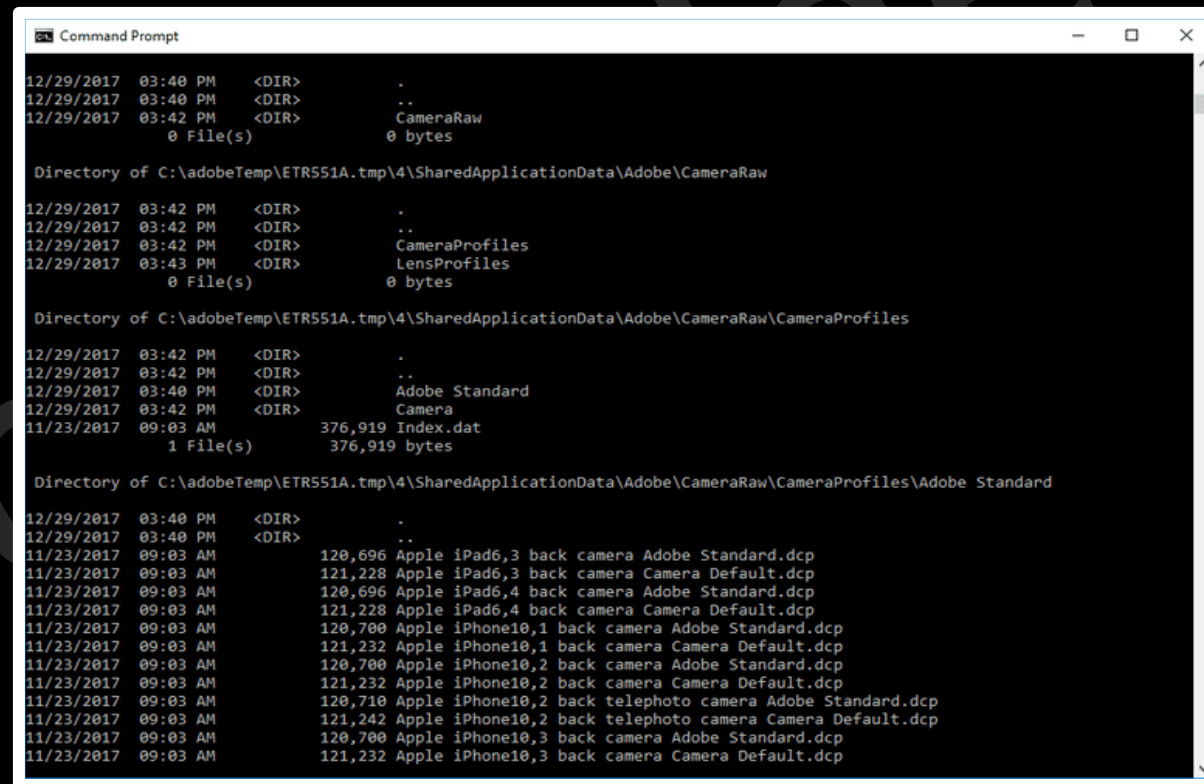
- There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches.
  - Command-line interface, or command interpreter.
  - Graphical User Interfaces.



**Operating  
System**



- Command Interpreters - Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems).
- On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the *Bourne shell*, *C shell*, *Bourne-Again shell*, *Korn shell*, and others.



```
Command Prompt
12/29/2017 03:40 PM <DIR> .
12/29/2017 03:40 PM <DIR> ..
12/29/2017 03:42 PM <DIR> CameraRaw
0 File(s) 0 bytes

Directory of C:\adobeTemp\ETR551A.tmp\4\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 03:43 PM <DIR> LensProfiles
0 File(s) 0 bytes

Directory of C:\adobeTemp\ETR551A.tmp\4\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\4\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 120,696 Apple iPad6,4 back camera Adobe Standard.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 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 Adobe Standard.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 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 Adobe Standard.dcp
11/23/2017 09:03 AM 121,232 Apple iPhone10,3 back camera Camera Default.dcp
```

- **Graphical User Interfaces** - A second strategy for interfacing with the operating system is through a user- friendly graphical user interface, or GUI. Here, users employ a mouse-based window- and-menu system characterized by a desktop.
- The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder —or pull down a menu that contains commands.



- Because a mouse is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touchscreen interface.
- Here, users interact by making gestures on the touchscreen—for example, pressing and swiping fingers across the screen.





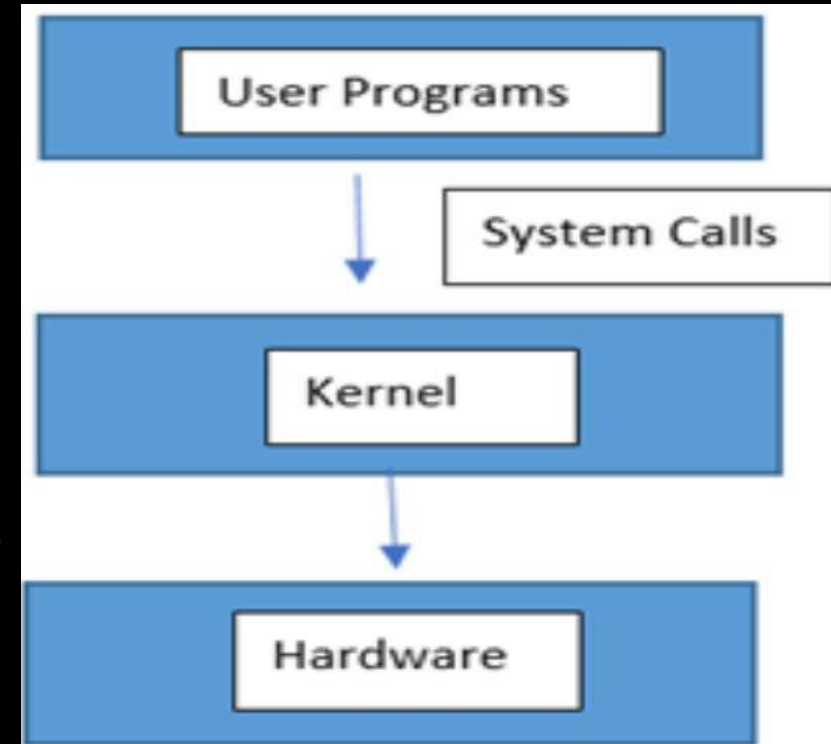
- The choice of whether to use a command-line or GUI interface is mostly one of personal preference.
- System administrators who manage computers and power users who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform.
- Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable.





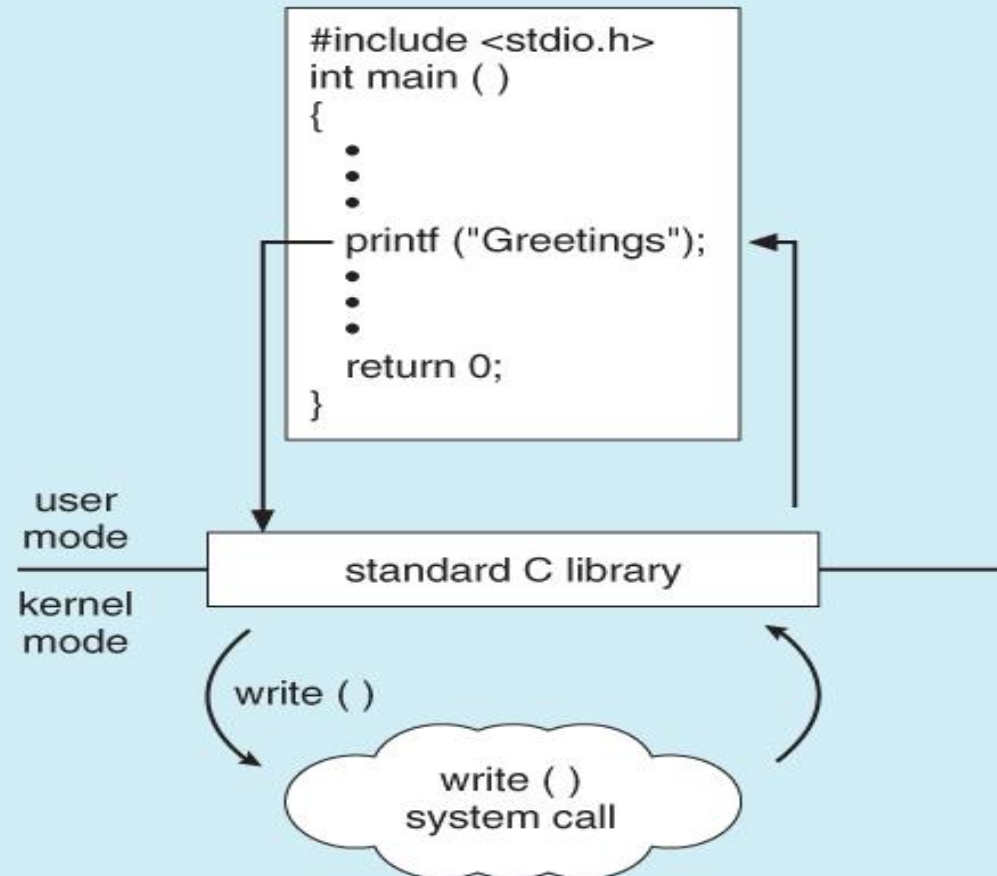
# System call

- 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.



## EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. For example, let's assume a C program invokes the `printf( )` statement. The C library intercepts this call and invokes the necessary `system` call(s) 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**
  1. end, abort
  2. load, execute
  3. create process, terminate process
  4. get process attributes, set process attributes
  5. wait for time
  6. wait event, signal event
  7. allocate and free memory

## • File management

1. create file, delete file
2. open, close
3. read, write, reposition
4. get file attributes, set file attributes

## • Device management

1. request device, release device
2. read, write, reposition
3. get device attributes, set device attributes
4. logically attach or detach devices

## • Information maintenance

1. get time or date, set time or date
2. get system data, set system data
3. get process, file, or device attributes
4. set process, file, or device attributes

## • Communications

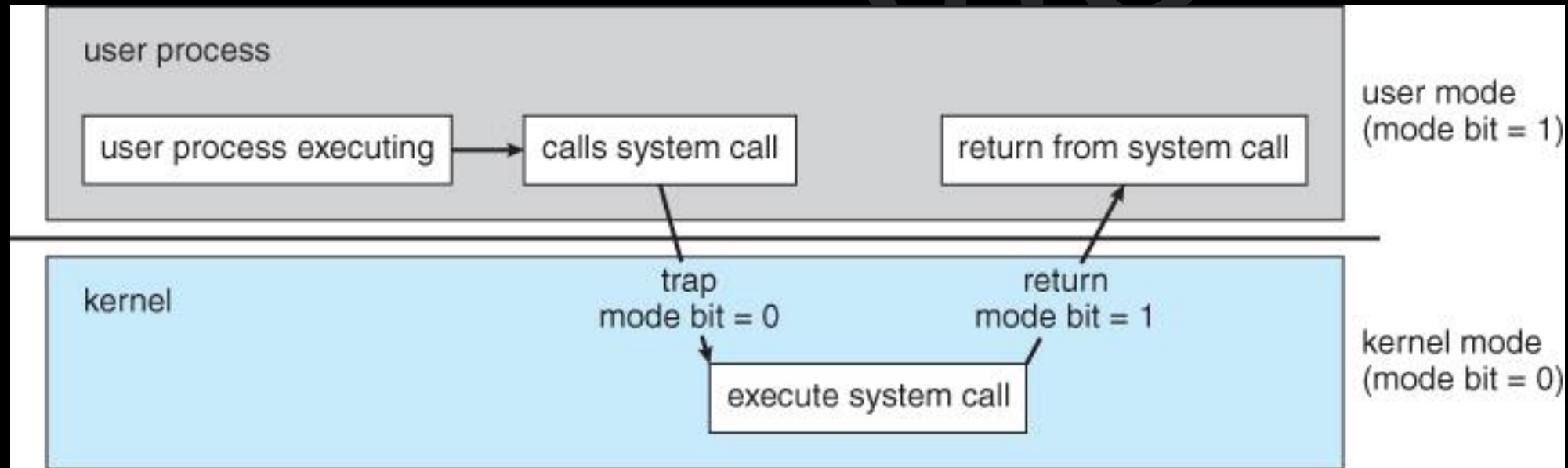
1. create, delete communication connection
2. send, receive messages transfer status information

Knowledge Gate



# Mode

- 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

[Knowledge Gate Website](https://www.knowledgegate.com/)

# Break

[Knowledge Gate Website](#)

# Process

In general, a process is a program in execution.

A Program is not a Process by default. A program is a passive entity, i.e. a file containing a list of instructions stored on disk (secondary memory) (often called an executable file).

A program becomes a Process when an executable file is loaded into main memory and when it's PCB is created.

A process on the other hand is an Active Entity, which require resources like main memory, CPU time, registers, system buses etc.

process state
process number
program counter
registers
memory limits
list of open files
* *

Even if two processes may be associated with same program, they will be considered as two separate execution sequences and are totally different process.

For instance, if a user has invoked many copies of web browser program, each copy will be treated as separate process. even though the text section is same but the data, heap and stack sections can vary.

Knowledge Gate

[Knowledge Gate Website](#)

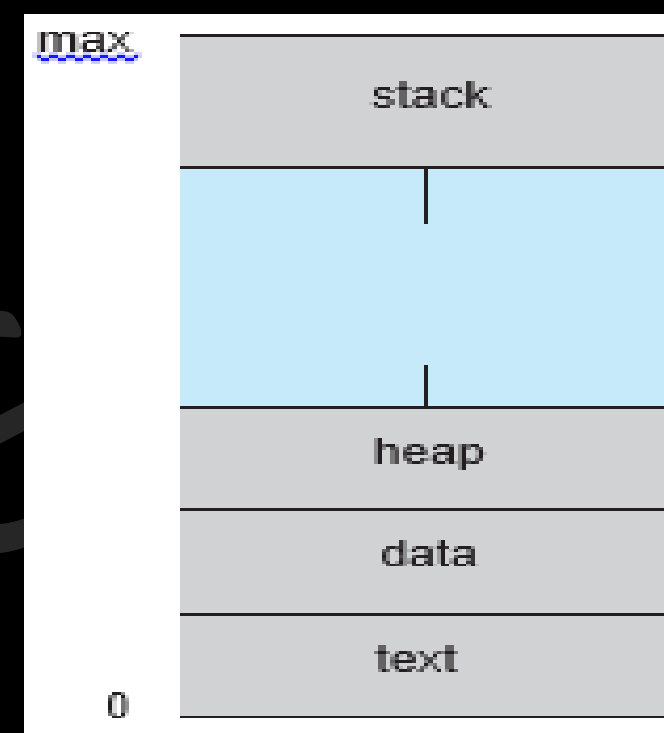
A Process consists of following sections:

**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.



Point	Program	Process
<b>Definition</b>	A set of instructions written to perform a specific task.	An instance of a program being executed.
<b>State</b>	Static; exists as code on disk or in storage.	Dynamic; exists in memory and has a state (e.g., running, waiting).
<b>Resources</b>	Does not require system resources when not running.	Requires CPU time, memory, and other resources during execution.
<b>Independence</b>	Exists independently and is not executing.	Can operate concurrently with other processes.
<b>Interaction</b>	Does not interact with other programs or the system.	Can interact with other processes and the operating system through system calls and inter-process communication.



## Process Control Block (PCB)

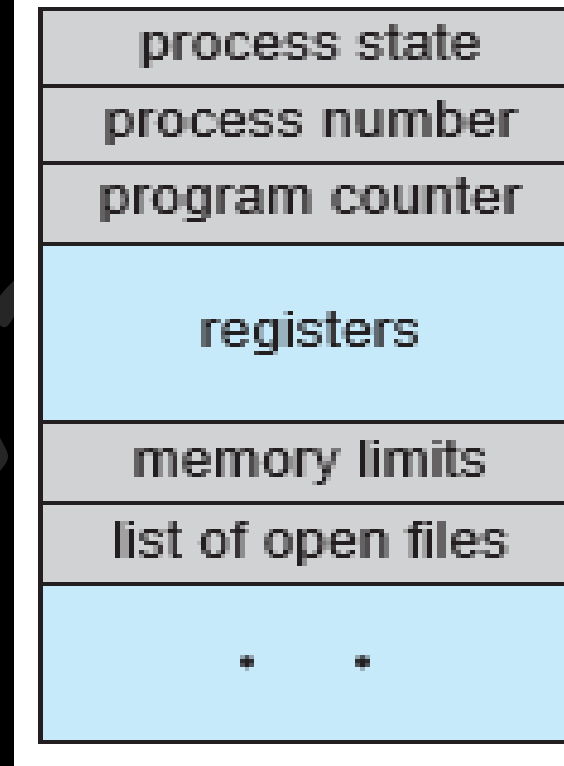
Each process is represented in the operating system by a process control block (PCB) — also called a task control block.

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:

**1. Process state:** The state may be new, ready, running, waiting, halted, and so on.

**2. Program counter:** The counter indicates the address of the next instruction to be executed for this process.

**3. 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.



**4. CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**5. 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.

**6. Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

**7. 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.

process state
process number
program counter
registers
memory limits
list of open files
* *

# 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:

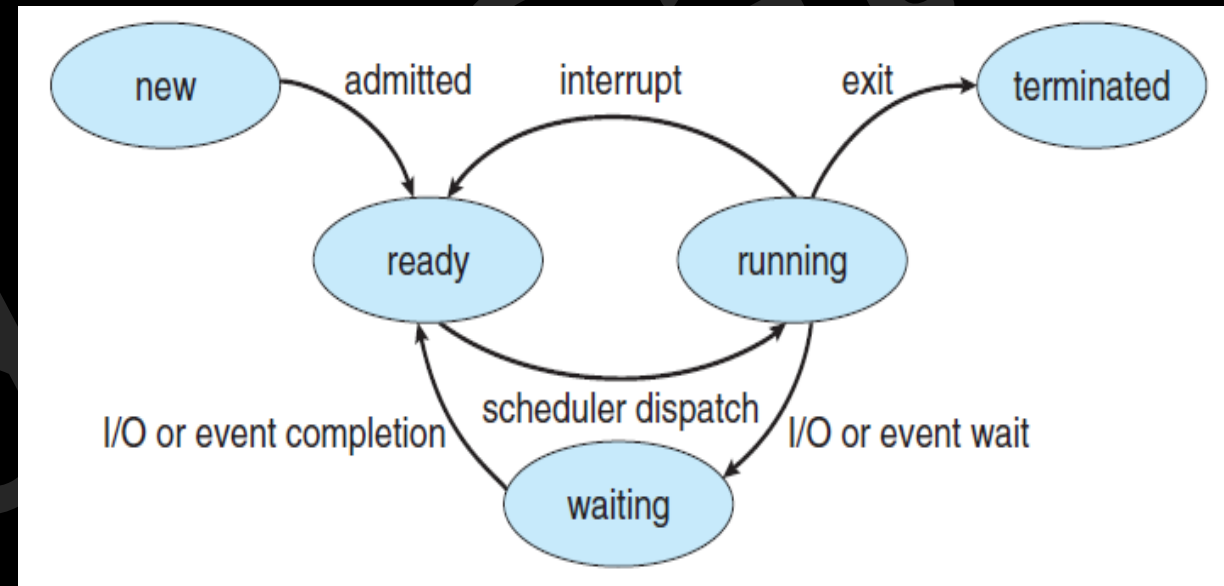
**New**: The process is being created.

**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.



# 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 Scheduler (LTS)/Spooler**: Long-term schedulers determine which processes enter the ready queue from the job pool. Operating less frequently than short-term schedulers, they focus on long-term system goals such as maximizing throughput.

**Medium-term scheduler**: The medium-term scheduler swaps processes in and out of memory to optimize CPU usage and manage memory allocation. By doing so, it adjusts the degree of multiprogramming and frees up memory as needed. Swapping allows the system to pause and later resume a process, improving overall system efficiency.

**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.

<b>Point</b>	<b>Long-Term Scheduler</b>	<b>Short-Term Scheduler</b>	<b>Middle Scheduler</b>
<b>Function</b>	Controls the admission of new processes into the system.	Selects which ready process will execute next.	Adjusts the degree of multiprogramming, moving processes between ready and waiting queues.
<b>Frequency</b>	Executes infrequently as it deals with the admission of new processes.	Executes frequently to rapidly switch between processes.	Executes at an intermediate frequency, balancing long-term and short-term needs.
<b>Responsibility</b>	Determines which programs are admitted to the system from the job pool.	Manages CPU scheduling and the switching of processes.	Controls the mix of CPU-bound and I/O-bound processes to optimize throughput.
<b>Impact on System Performance</b>	Influences overall system performance and degree of multiprogramming.	Directly impacts CPU utilization and response time.	Balances system load to prevent resource bottlenecks or idle resources.
<b>Decision Making</b>	Makes decisions based on long-term goals like system throughput.	Makes decisions based on short-term goals like minimizing response time.	Makes decisions considering both short-term and long-term goals, optimizing resource allocation.



**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.

## Context Switch

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.

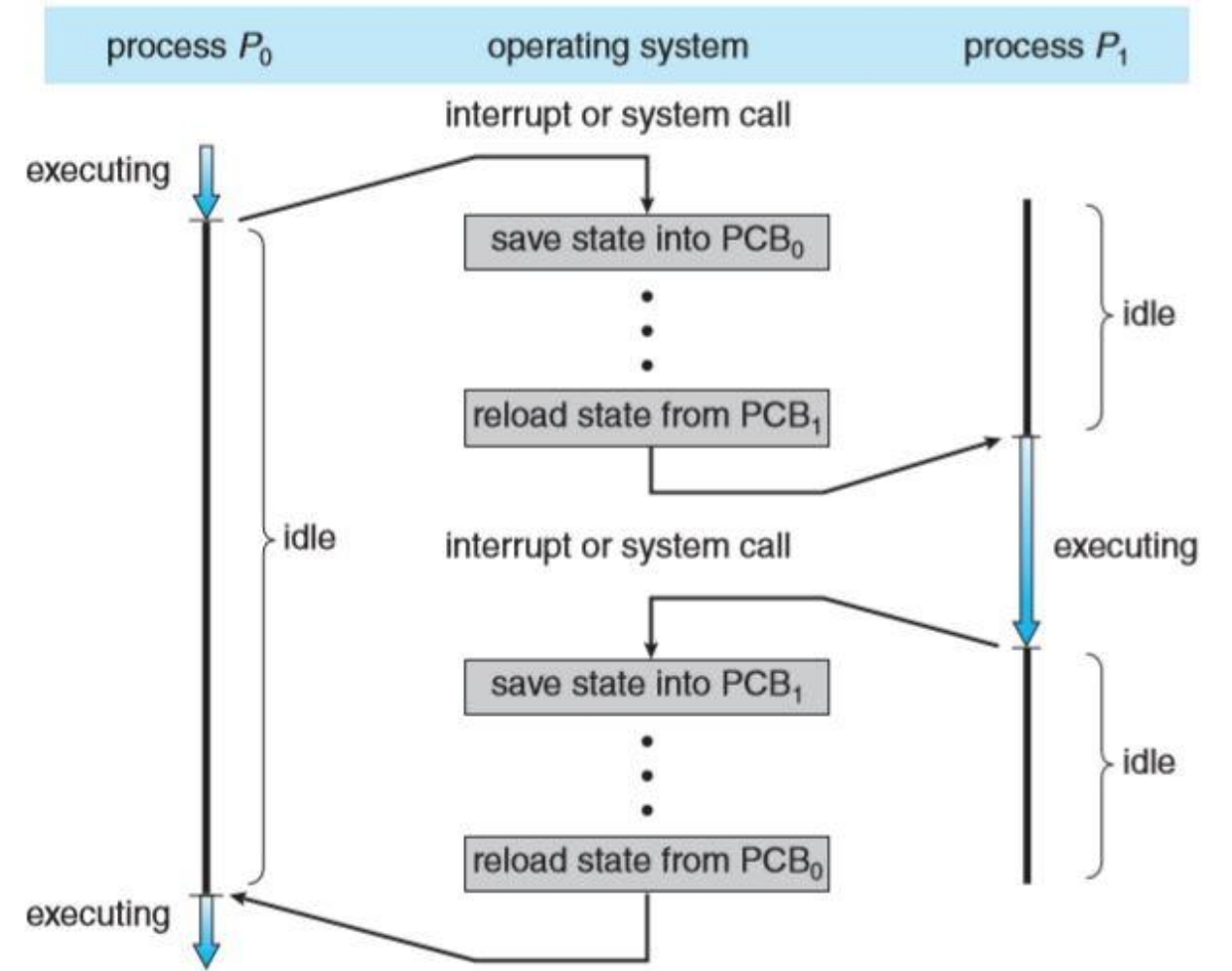


Diagram showing CPU switch from process to process.

# Break

[Knowledge Gate Website](#)

Sector-78

Rohtak

Sadar bazar

Grand father paper industry machanical

May-2023,

Cacha 1 km away

Elder brother marrage in

Knowledge Gate

[Knowledge Gate Website](https://www.knowledgegate.com)

Knowledge Gate

[Knowledge Gate Website](#)



## CPU Scheduling

1. CPU scheduling is the process of determining which process in the ready queue is allocated to the CPU.
2. Various scheduling algorithms can be used to make this decision, such as First-Come-First-Served (FCFS), Shortest Job Next (SJN), Priority and Round Robin (RR).
3. Different algorithm support different class of process and favor different scheduling criterion.

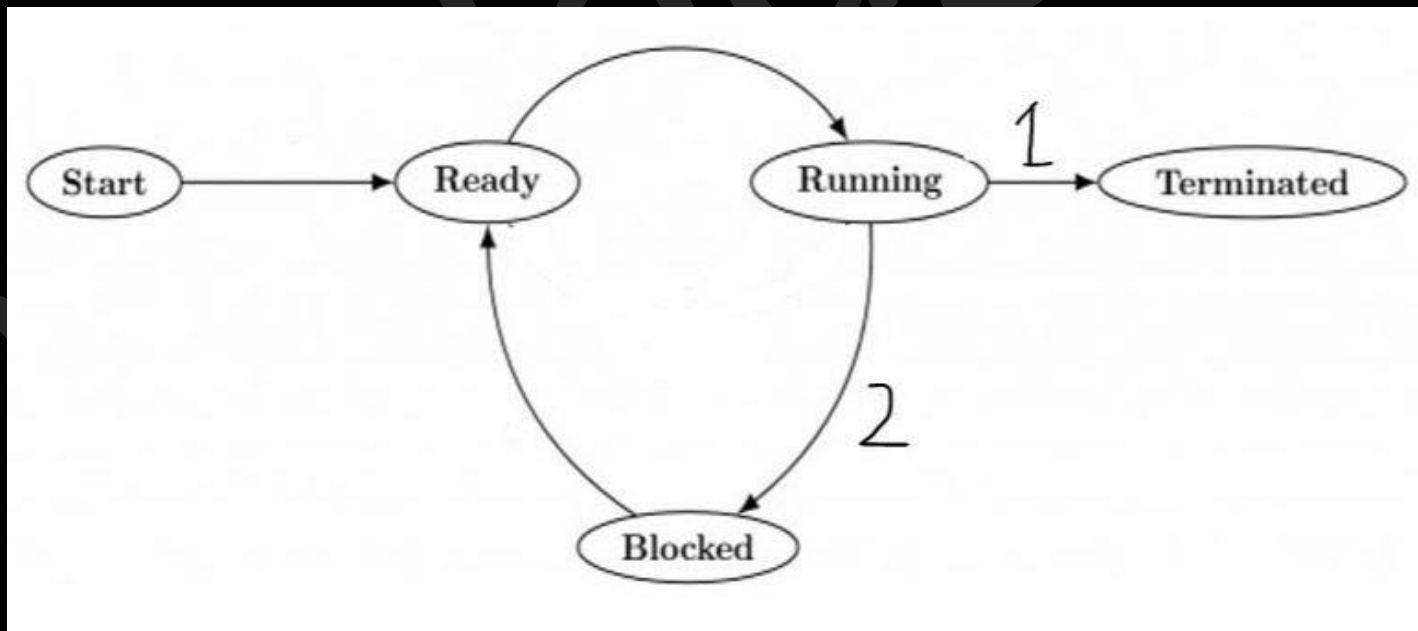
# Type of scheduling

**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

When a process completes its execution (Termination state)

When a process wants to perform some i/o operations(Blocked state)



# Pre-emptive

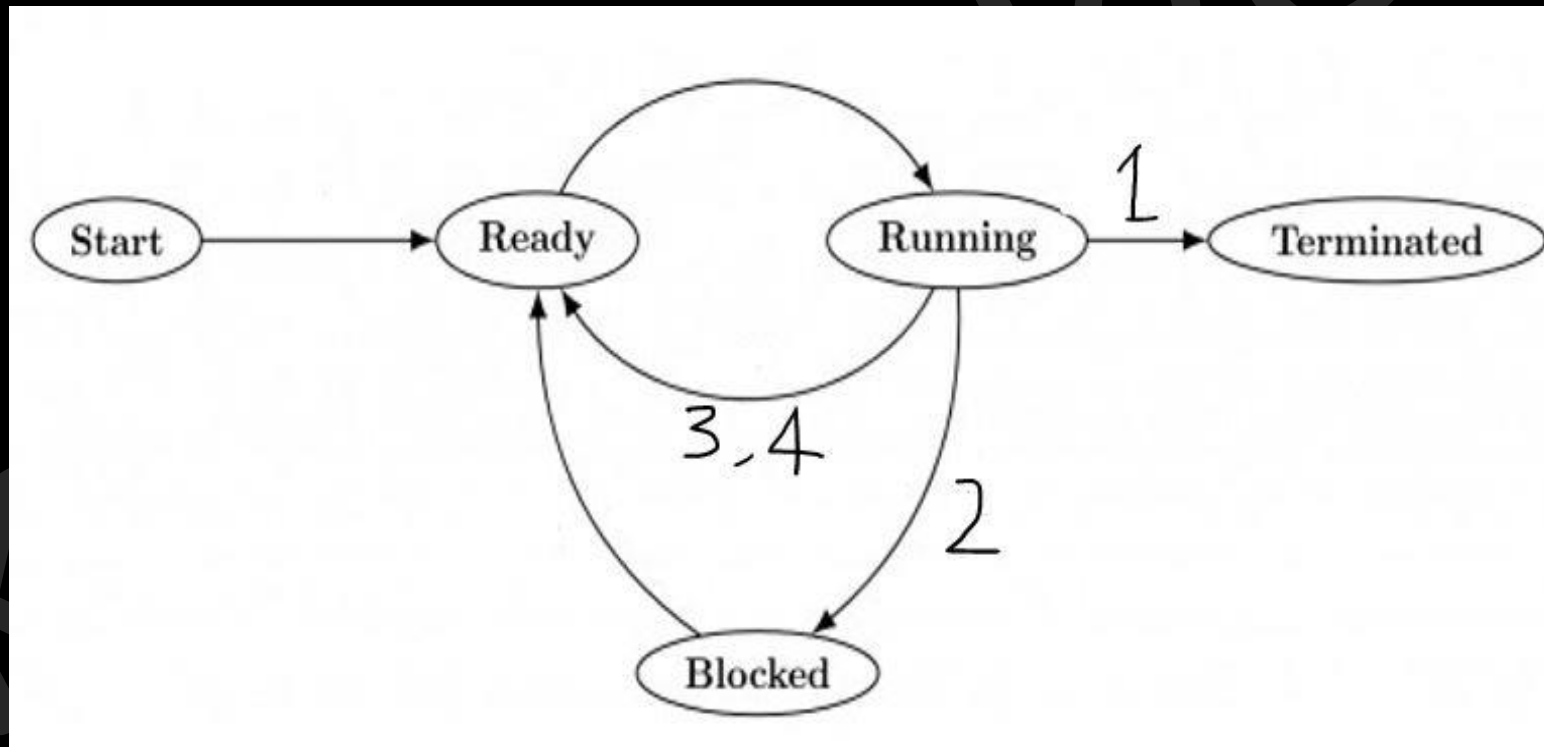
Under Pre-emptive scheduling, once the CPU has been allocated to a process, A process will leave the CPU willingly or it can be forced out. So it will leave the CPU

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.



<b>Point</b>	<b>Non-Pre-emptive Scheduling</b>	<b>Pre-emptive Scheduling</b>
<b>CPU Allocation</b>	Once a process starts, it runs to completion or wait for some event.	A process can be interrupted and moved to the ready queue.
<b>Response Time</b>	Can be longer, especially for short tasks.	Generally shorter, as higher-priority tasks can pre-empt others.
<b>Complexity</b>	Simpler to implement.	More complex, requiring careful handling of shared resources.
<b>Resource Utilization</b>	May lead to inefficient CPU utilization.	Typically more efficient, as it can quickly switch tasks.
<b>Suitable Applications</b>	Batch systems and applications that require predictable timing.	Interactive and real-time systems requiring responsive behavior.

**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:

Knowledge Gate

[Knowledge Gate Website](#)

CPU utilization: Keeping the CPU as busy as possible.



Gate

[Knowledge Gate Website](#)



**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.



Gate



**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 or perform 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.

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)

## 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 (FIRST 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 <sub>0</sub>	2	4			
P <sub>1</sub>	1	2			
P <sub>2</sub>	0	3			
P <sub>3</sub>	4	2			
P <sub>4</sub>	3	1			
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.

Knowledge Gate

P. No	AT	BT	TAT=CT-AT	WT=TAT -BT
P <sub>0</sub>	0	100		
P <sub>1</sub>	1	2		
Average				

P. No	AT	BT	TAT=CT-AT	WT=TAT -BT
P <sub>0</sub>	1	100		
P <sub>1</sub>	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.

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)

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 whether, this new process with the smaller CPU burst requirement than 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 as it guarantees 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 <sub>0</sub>	1	7			
P <sub>1</sub>	2	5			
P <sub>2</sub>	3	1			
P <sub>3</sub>	4	2			
P <sub>4</sub>	5	8			
Average					

## 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

Here process with the longer CPU burst requirement goes into starvation and have response time.

This algo cannot be implemented as there is no way to know the length of the next CPU burst. As SJF is not implementable, we can use the one technique where we try to predict the CPU burst of the next coming process.

# Priority scheduling



# 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.

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.

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 <sub>0</sub>	1	4	4			
P <sub>1</sub>	2	2	5			
P <sub>2</sub>	2	3	7			
P <sub>3</sub>	3	5	8(H)			
P <sub>4</sub>	3	1	5			
P <sub>5</sub>	4	2	6			
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



## Round robin

This algo is designed for time sharing systems, where it is not, the idea to complete one process and then start another, but to be responsive and divide time of CPU among the process in the ready state(circular).

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a maximum of 1 Time quantum say  $q$ . Up to which a process can hold the CPU in one go, with in which either a process terminates if process have a CPU burst of less than given time quantum or context switch will be executed and process must release the CPU voluntarily and enter the ready queue and wait for the next chance.

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 <sub>0</sub>	0	4			
P <sub>1</sub>	1	5			
P <sub>2</sub>	2	2			
P <sub>3</sub>	3	1			
P <sub>4</sub>	4	6			
P <sub>5</sub>	6	3			
<b>Average</b>					

## 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

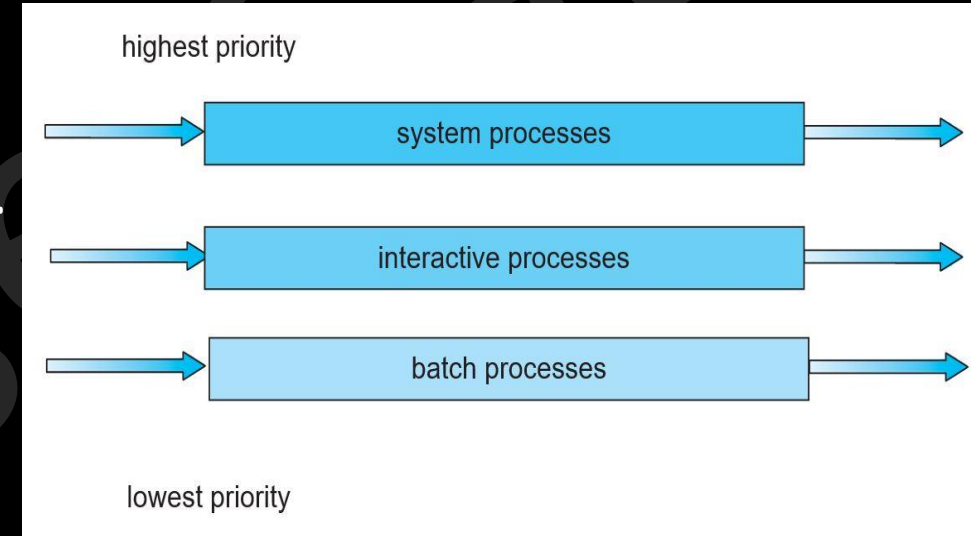
Longer process may starve

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.

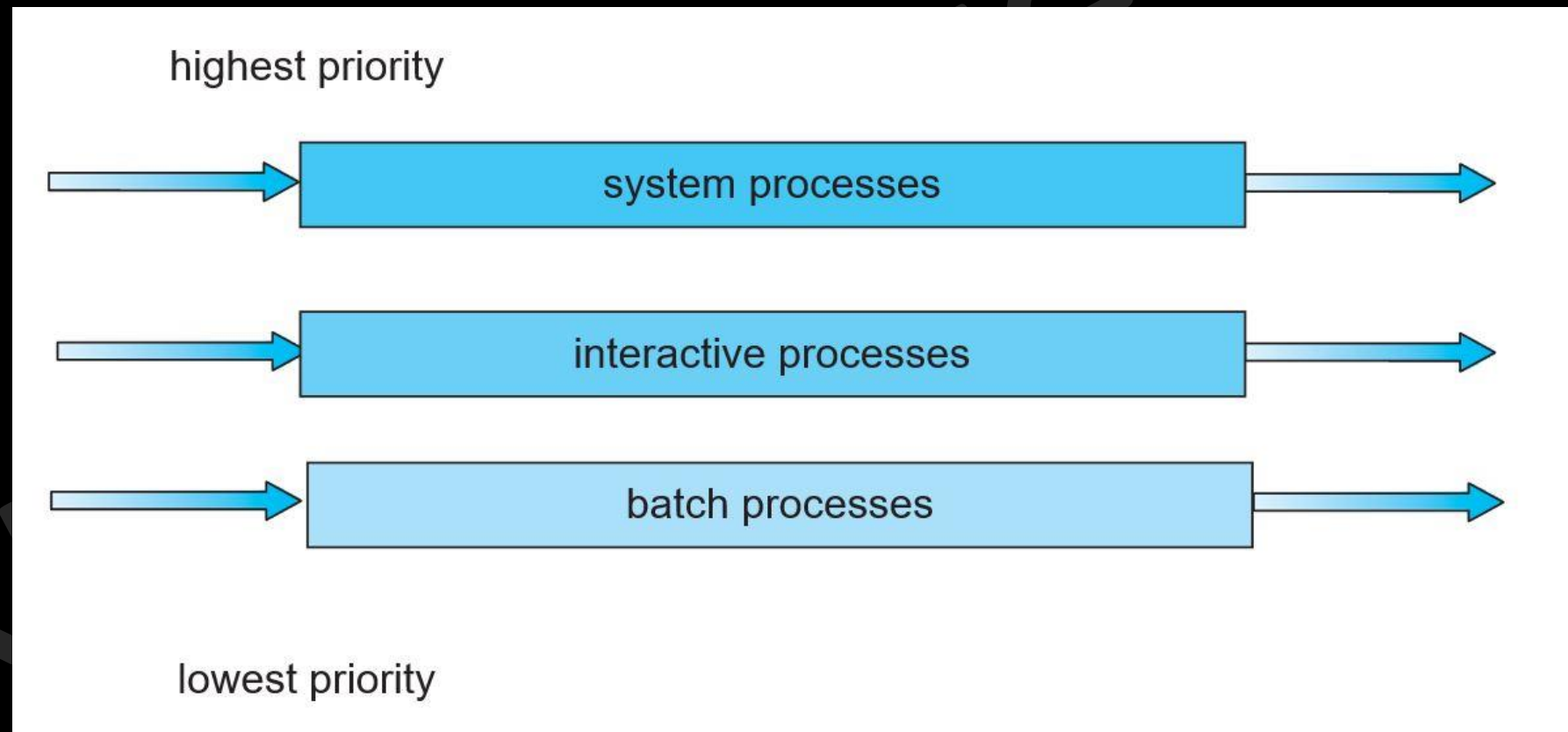
No idea of priority

# Multi Level-Queue Scheduling

- After studying all important approach to CPU scheduling, we must understand anyone of them alone is not good for every process in the system, as different process have different scheduling needs so, we must have a kind of hybrid scheduling idea, supporting all classes of processes.
- Here processes are easily classified into different groups.
  - **System** process
  - **foreground** (interactive) processes
  - **background** (batch) processes.
- A **multilevel queue** scheduling algorithm, partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on properties and requirement of the process.



- Each queue has its own scheduling algorithm. For example
  - System process might need priority algorithm
  - Interactive process might be scheduled by an RR algorithm
  - Batch processes is scheduled by an FCFS algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling or round robin with different time quantum.

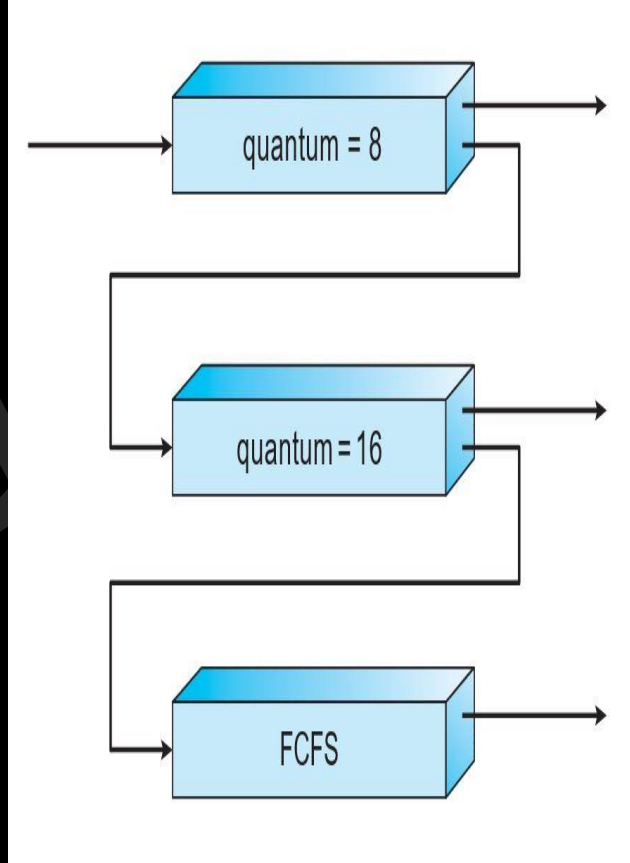


# Multi-level Feedback Queue Scheduling

Problem with multi-level queue scheduling is how to decide number of ready queue, scheduling algorithm inside the queue and between the queue and once a process enters a specific queue we can not change and queue after that.

The **multilevel feedback queue** scheduling algorithm, 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. 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.



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

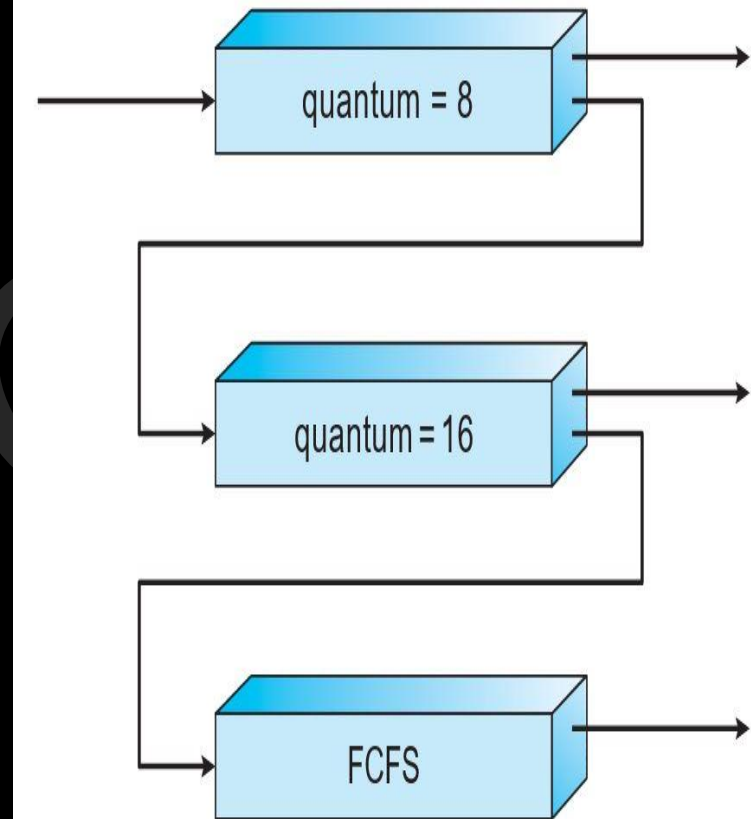
- The number of queues

- 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 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.





# Break

[Knowledge Gate Website](#)

# Process Synchronization & Race Condition

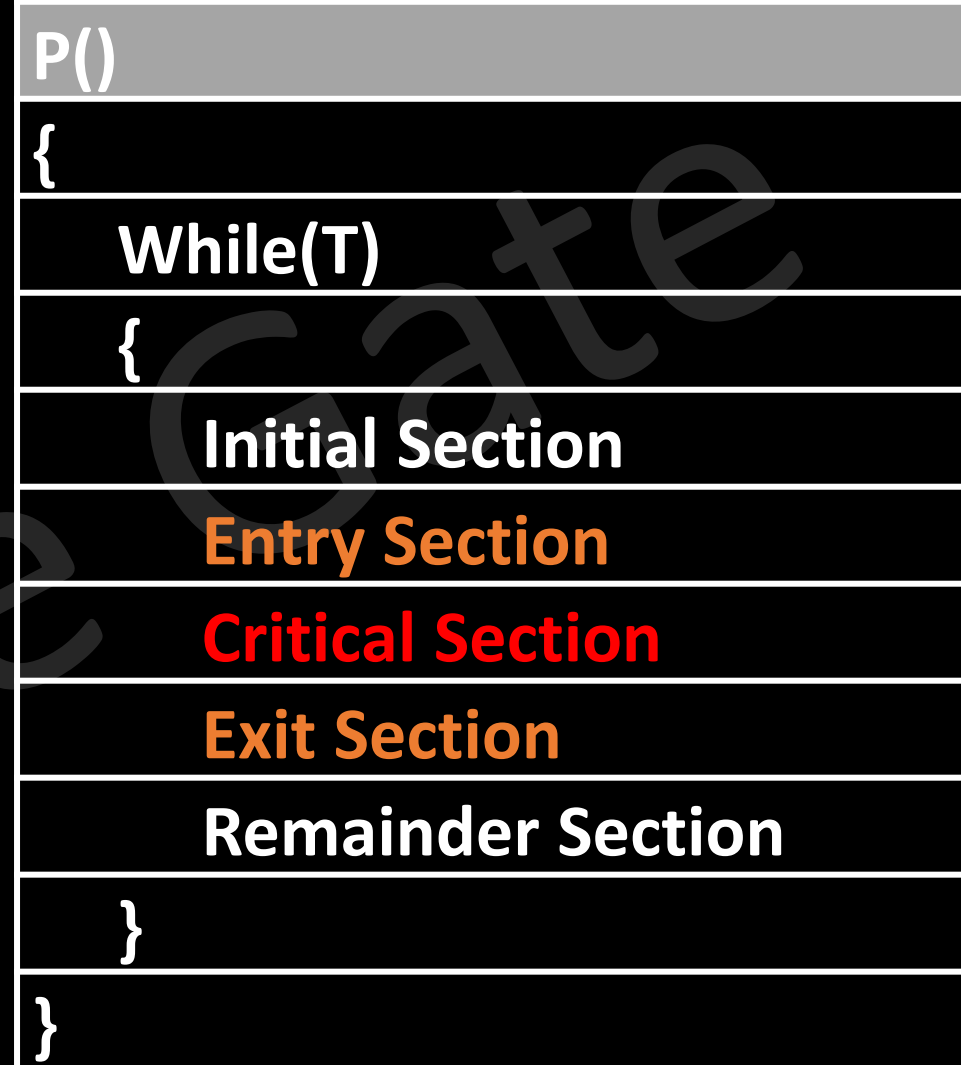
- As we understand in a multiprogramming environment a good number of processes compete for limited number of resources. Concurrent access to shared data at some time may result in data inconsistency for e.g.

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

- Race condition is a situation 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.

# 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.



# 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.

Knowledge Gate

# 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).

$P_0$	$P_1$
<pre>while (1) {     while (turn! = 0);     Critical Section     turn = 1;     Remainder section }</pre>	<pre>while (1) {     while (turn! = 1);     Critical Section     turn = 0;     Remainder Section }</pre>

- 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?

Knowledge Gate

- 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.

Knowledge Gate

# Dekker's algorithm

$P_i$

$P_j$

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

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

- Peterson's solution is a classic Software-based solution to the critical-section problem for two process. We will be using both: **turn** and **Boolean flag**.

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

- This solution ensures **Mutual Exclusion, Progress and Bounded Wait**.

# Operation System Solution (Semaphores)

1. Semaphores are synchronization tools using which we will attempt n-process solution.
2. 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).
3. The wait(S) operation was originally termed as P(S) and signal(S) was originally called V(S).

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.

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

Signal(S)
{
s++;
}

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

# 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/ Bounded Buffer Problem
  - Reader-Writer problem
  - Dining Philosopher problem
  - The Sleeping Barber 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.
- 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

Knowledge Gate



	Producer()	Consumer()
	Producer()	Consumer()
Semaphore S =	{	{
	while(T)	while(T)
	{	{
Semaphore E =		
Semaphore F =		
	}	}

Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
// Produce an item	
	// Pick item from buffer
// Add item to buffer	
	// Consume item
}	}



Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
// Produce an item	
	wait(S)
wait(S)	// Pick item from buffer
// Add item to buffer	signal(S)
signal(S)	
	// Consume item
}	}

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)	wait(E)
wait(F)	Consume item
}	}
}	}

# 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).
- 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.
- 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.

```
mysql> select title, release_year, length, replacement_cost from title
mysql> where length > 120 and replacement_cost = 29.99
mysql> order by title desc;
```

title	release_year	length	replacement_cost
West Side	2006	159	29.99
Virgin Delay	2006	179	29.99
Mount Surtides	2006	172	29.99
Tracy Elder	2006	142	29.99
Song Hedwig	2006	140	29.99
Slacker Lizards	2006	179	29.99
Sassy Packer	2006	154	29.99
River Gullow	2006	140	29.99
Right Cranes	2006	133	29.99
Quest Muzalier	2006	177	29.99
Posidon Forever	2006	159	29.99
Leathing Legally	2006	146	29.99
Loeless Viter	2006	181	29.99
Jungle Sagebrush	2006	124	29.99
Bericho Nulan	2006	171	29.99
Japanese Run	2006	130	29.99
Stilmore Boliver	2006	160	29.99
Flats Garden	2006	146	29.99
Fantasia Park	2006	131	29.99
Extraordinary Computer	2006	127	29.99
Everyone Craft	2006	163	29.99
Dirty Ace	2006	147	29.99
Clyde Theory	2006	139	29.99
Clockwork Paradise	2006	140	29.99
Ballroom Rockingbird	2006	172	29.99

- **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.

Knowledge Gate

## • Solution using Semaphores

- The reader processes share the following data structures:
- semaphore mutex = 1, wrt = 1; // Two semaphores
- int readcount = 0; // Variable

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

**Mutex =**

**Wrt =**

**Readcount =**

Writer()	Reader()
CS //Write	CS //Read

**Mutex =**

**Wrt =**

**Readcount =**

Writer()	Reader()
<b>Wait(wrt)</b>	
CS //Write	CS //Read
<b>Signal(wrt)</b>	



**Mutex =**

**Wrt =**

**Readcount =**

	<b>Readcount++</b>
Wait(wrt)	
CS //Write	CS //Read
Signal(wrt)	
	<b>Readcount--</b>

**Mutex =**

**Wrt =**

**Readcount =**

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

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

## Dining Philosopher Problem

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.



- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she can't pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.





Knowledge Gate

[Knowledge Gate Website](https://www.knowledgegate.com)



# Indian Chopsticks



Knowledge Gate



## 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.

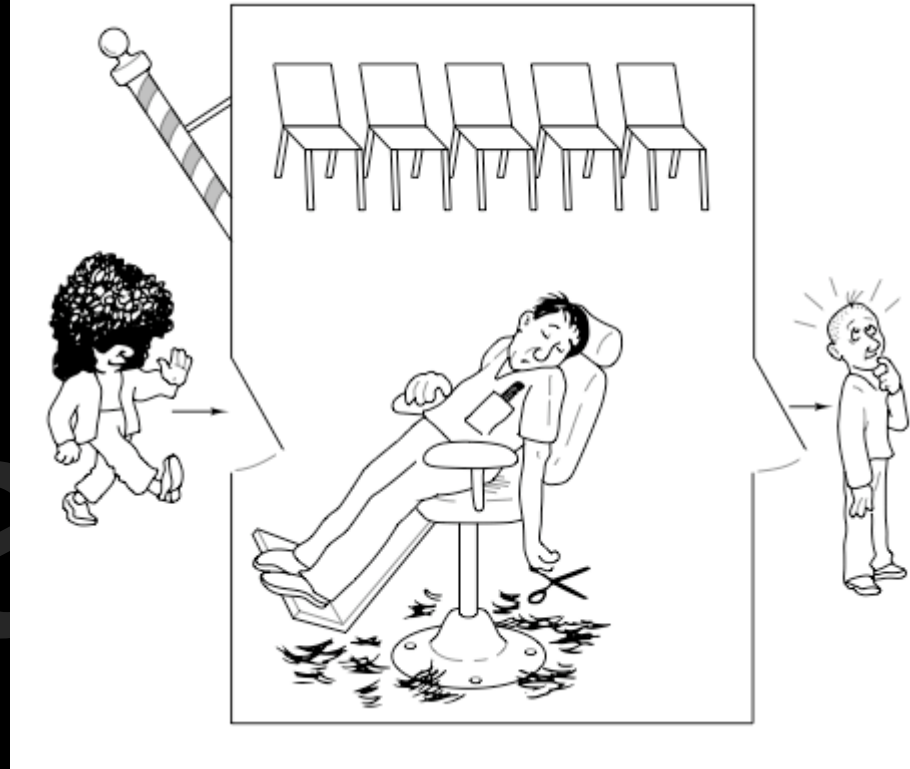
Knowledge Gate

- 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.



# The Sleeping Barber problem

- **Barbershop**: A barbershop consists of a waiting room with  $n$  chairs and a barber room with one barber chair.
- **Customers**: Customers arrive at random intervals. If there is an available chair in the waiting room, they sit and wait. If all chairs are taken, they leave.
- **Barber**: The barber sleeps if there are no customers. If a customer arrives and the barber is asleep, they wake the barber up.
- **Synchronization**: The challenge is to coordinate the interaction between the barber and the customers using concurrent programming mechanisms.



```
semaphore barber = 0; // Indicates if the barber is available
semaphore customer = 0; // Counts the waiting customers
semaphore mutex = 1; // Mutex for critical section
int waiting = 0; // Number of waiting customers
```

## Barber

```
while(true)
{
    wait(customer);
    wait(mutex);
    waiting = waiting - 1;
    signal(barber);
    signal(mutex);
    // Cut hair
}
```

## Customer

```
wait(mutex);
if(waiting < n)
{
    waiting = waiting + 1;
    signal(customer);
    signal(mutex);
    wait(barber);
    // Get hair cut
}
else
{
    signal(mutex);
}
```

## 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)
```

```
{  
    Boolean rv = *target;  
    *target = true;  
    return rv;  
}
```

```
While(1)
```

```
{  
    while (test and set(&lock));  
    /* critical section */  
    lock = false;  
    /* remainder section */  
}
```

Knowledge Gate



- 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.

## Basics of Dead-Lock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- 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.

$P_1$

$P_2$

$R_1$

$R_2$









Tax

Services



Knowledge Gate



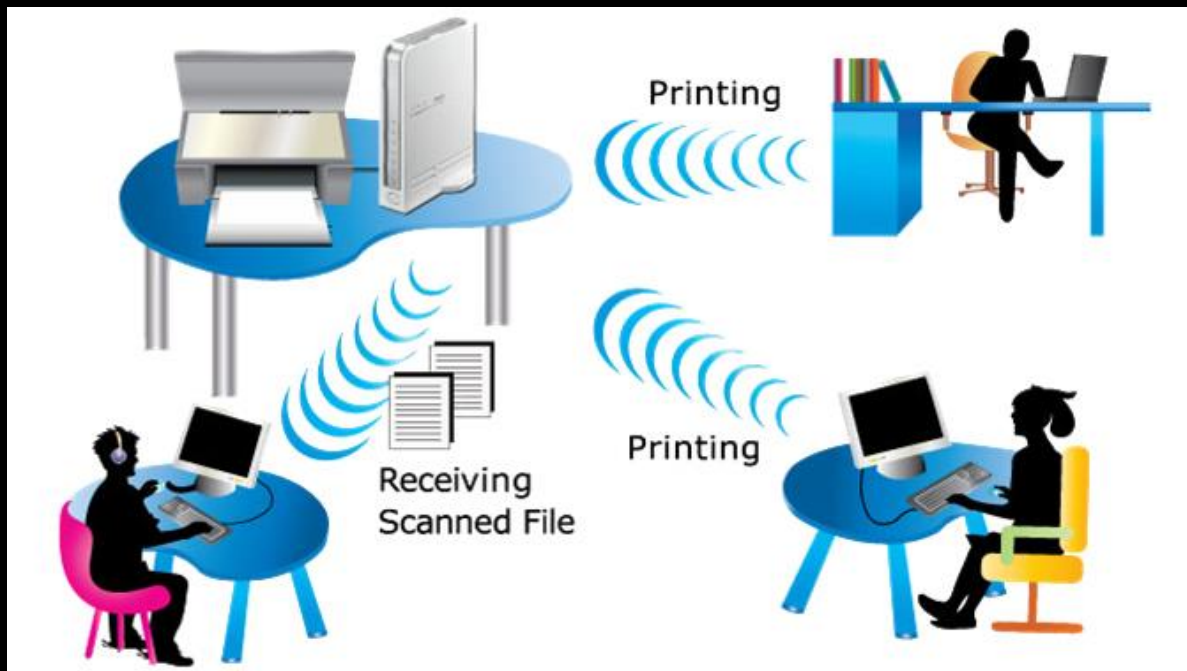
KNOWLEDGE

## 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  $P_0$ .

Knowledge Gate

# Deadlock Handling methods

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

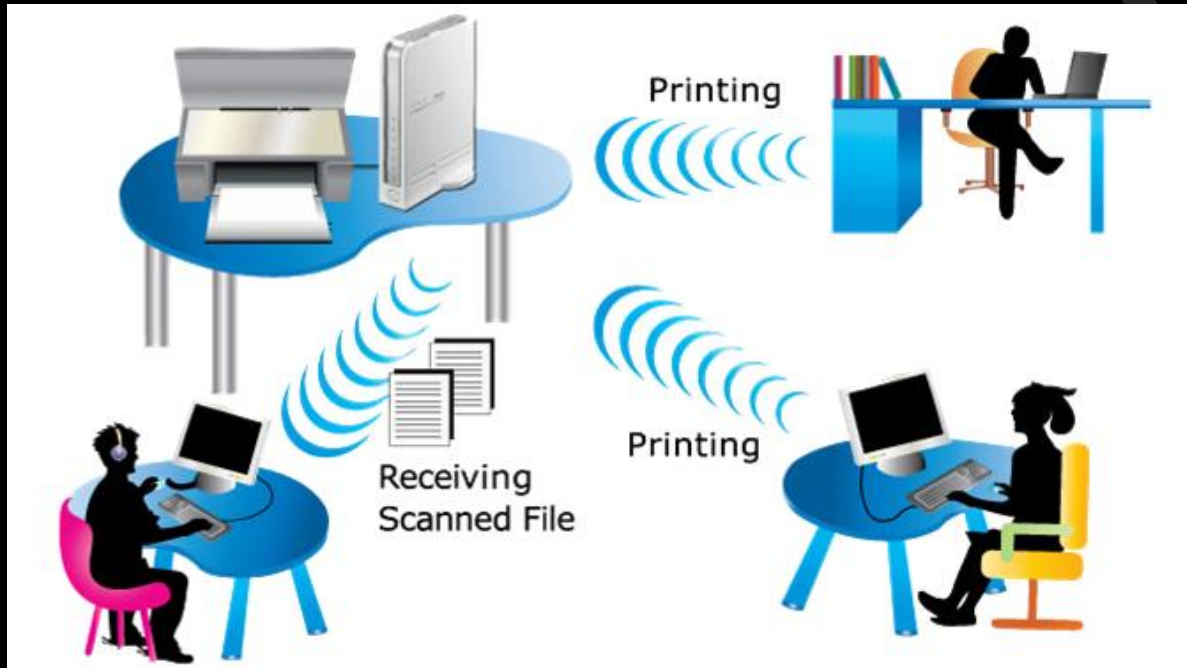
## Prevention

- 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

1. Conservative Approach: Process is allowed to run if & only if it has acquired all the resources.
2. Alternative protocol: 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.
3. Wait time out: We place a max time outs up to which a process can wait. After which process must release all the holding resources & exit.

$P_1$

$P_2$

$R_1$

$R_2$



## 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 (Considering Priority).
    - If the resources are neither available nor held by a waiting process, the requesting process must wait, or may allow to pre-empt resource of a running process Considering Priority.

## 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.

Knowledge Gate

- 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.



[Knowledge Gate Website](http://www.knowledgegate.com)

# Avoidance





- 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 avoiding deadlocks we require additional information about how resources are to be requested. which resources a process will request during its lifetime.
- With this additional knowledge, the operating system can decide for each request whether process should wait or not.

Max Need			
	E	F	G
P <sub>0</sub>			
P <sub>1</sub>			
P <sub>2</sub>			
P <sub>3</sub>			

Allocation			
	E	F	G
P <sub>0</sub>			
P <sub>1</sub>			
P <sub>2</sub>			
P <sub>3</sub>			

Current Need			
	E	F	G
P <sub>0</sub>			
P <sub>1</sub>			
P <sub>2</sub>			
P <sub>3</sub>			

System Max		
E	F	G

Available		
E	F	G

Max Need			
	E	F	G
P <sub>0</sub>	4	3	1
P <sub>1</sub>	2	1	4
P <sub>2</sub>	1	3	3
P <sub>3</sub>	5	4	1

Allocation			
	E	F	G
P <sub>0</sub>	1	0	1
P <sub>1</sub>	1	1	2
P <sub>2</sub>	1	0	3
P <sub>3</sub>	2	0	0

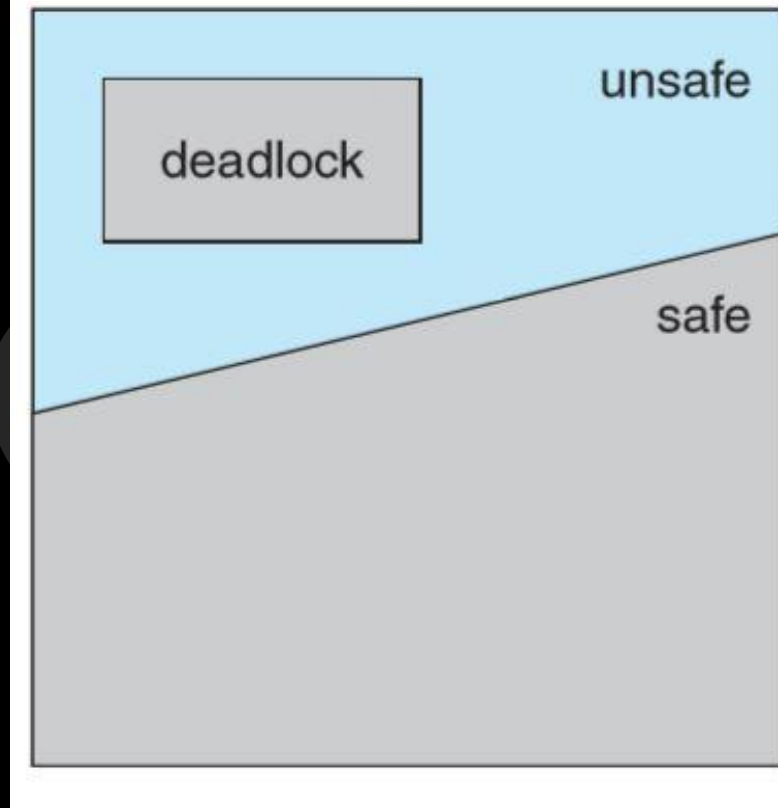
Current Need			
	E	F	G
P <sub>0</sub>			
P <sub>1</sub>			
P <sub>2</sub>			
P <sub>3</sub>			

System Max		
E	F	G
8	4	6

Available		
E	F	G



- **Safe sequence**: some sequence in which we can satisfies demand of every process without going into deadlock, if yes, this sequence is called safe sequence.
- **Safe Sate**: If their exist at least one possible safe sequence.
- **Unsafe Sate**: If their exist no possible safe sequence.



# 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  $\text{Available}[j]$  equals  $k$ , then  $k$  instances of resource type  $R_j$  are available.

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

Available		
E	F	G
3	3	0

	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  $n \times m$  matrix defines the number of resources of each type currently allocated to each process. If  $\text{Allocation}[i][j]$  equals  $k$ , then process  $P_i$  is currently allocated  $k$  instances of resource type  $R_j$ .

**Need/Demand/Requirement:** An  $n \times m$  matrix indicates the remaining resource need of each process. If  $\text{Need}[i][j]$  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.

	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

## 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  
 $Need_i \leq Work$   
If no such i exists, go to step 4.

3- Work = Work + Allocation<sub>i</sub>  
Finish[i] = true  
Go to step 2.

4- If Finish[i] == true for all i, then the system is in a safe state.  
This algorithm may require an order of  $m \cdot n^2$  operations to determine whether a state is safe.

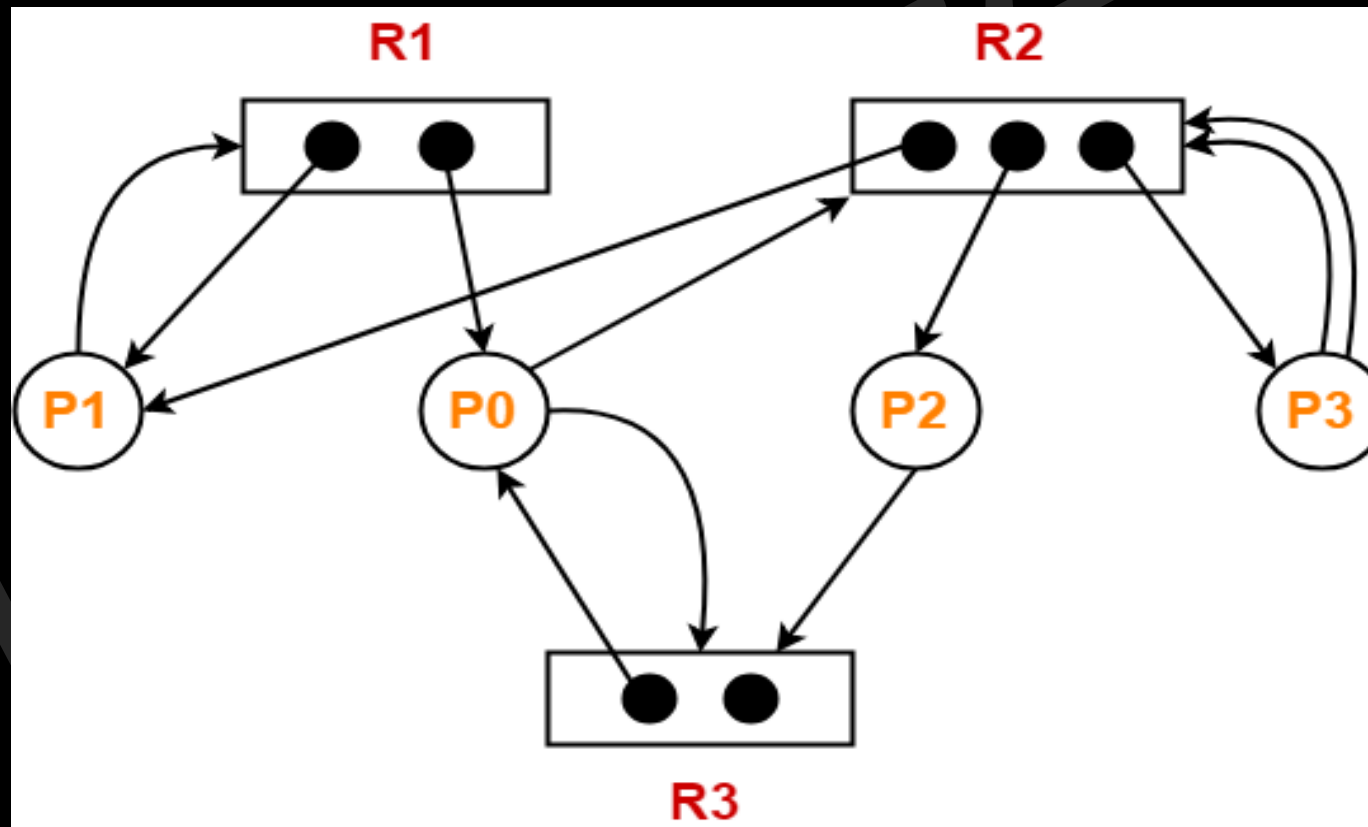
	Need		
	E	F	G
P <sub>0</sub>	3	3	0
P <sub>1</sub>	1	0	2
P <sub>2</sub>	0	3	0
P <sub>3</sub>	3	4	1

Work		
E	F	G
3	3	0

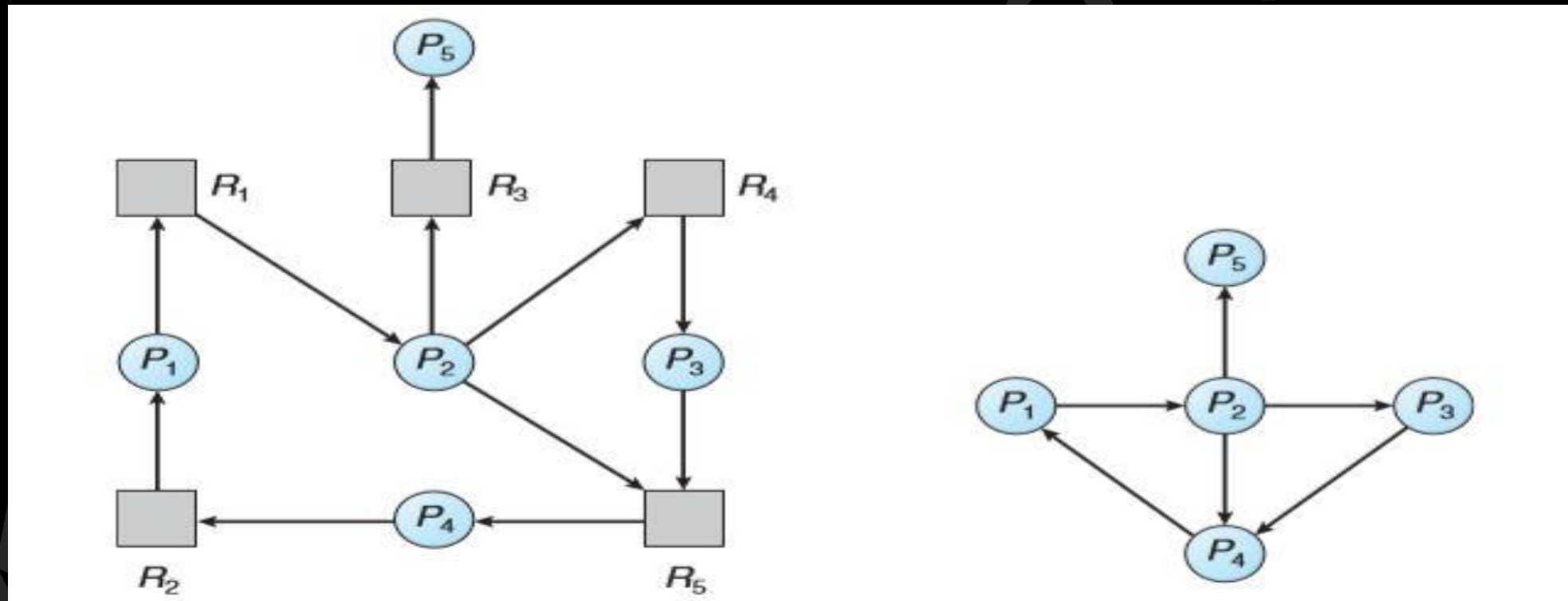
Finish[i]		
E	F	G
F	F	F

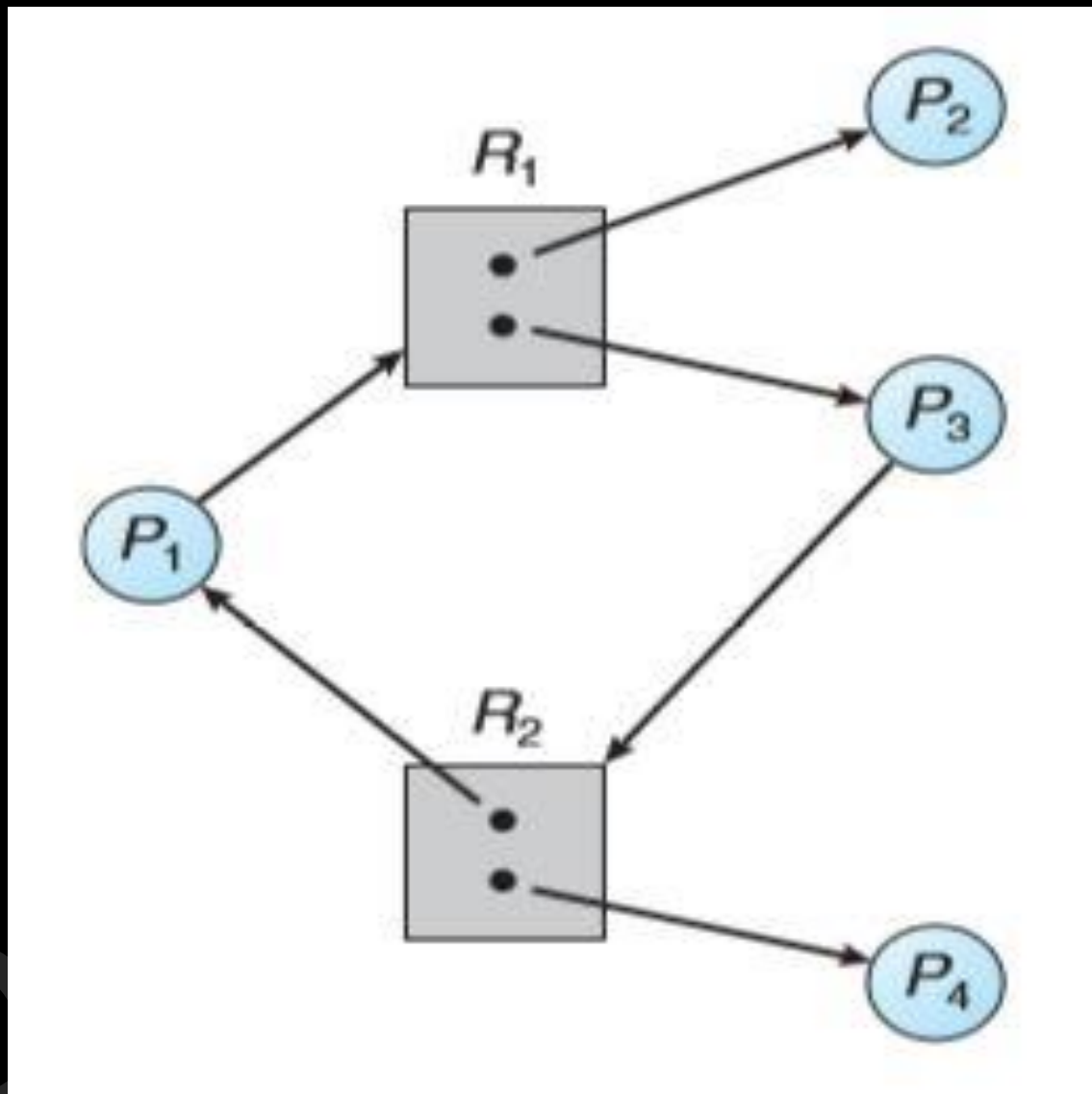
# Resource Allocation Graph

- Deadlock can also be described 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.

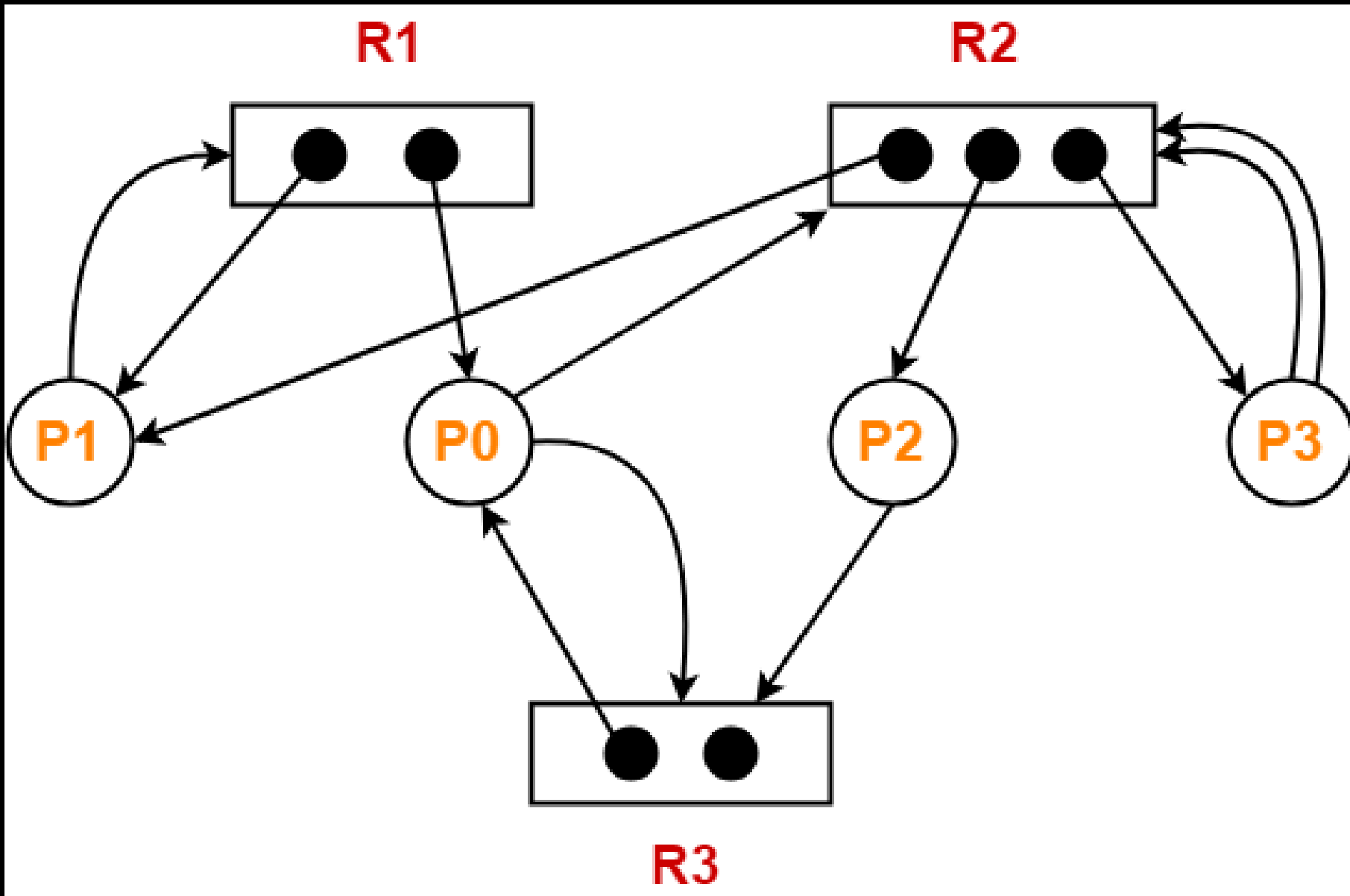


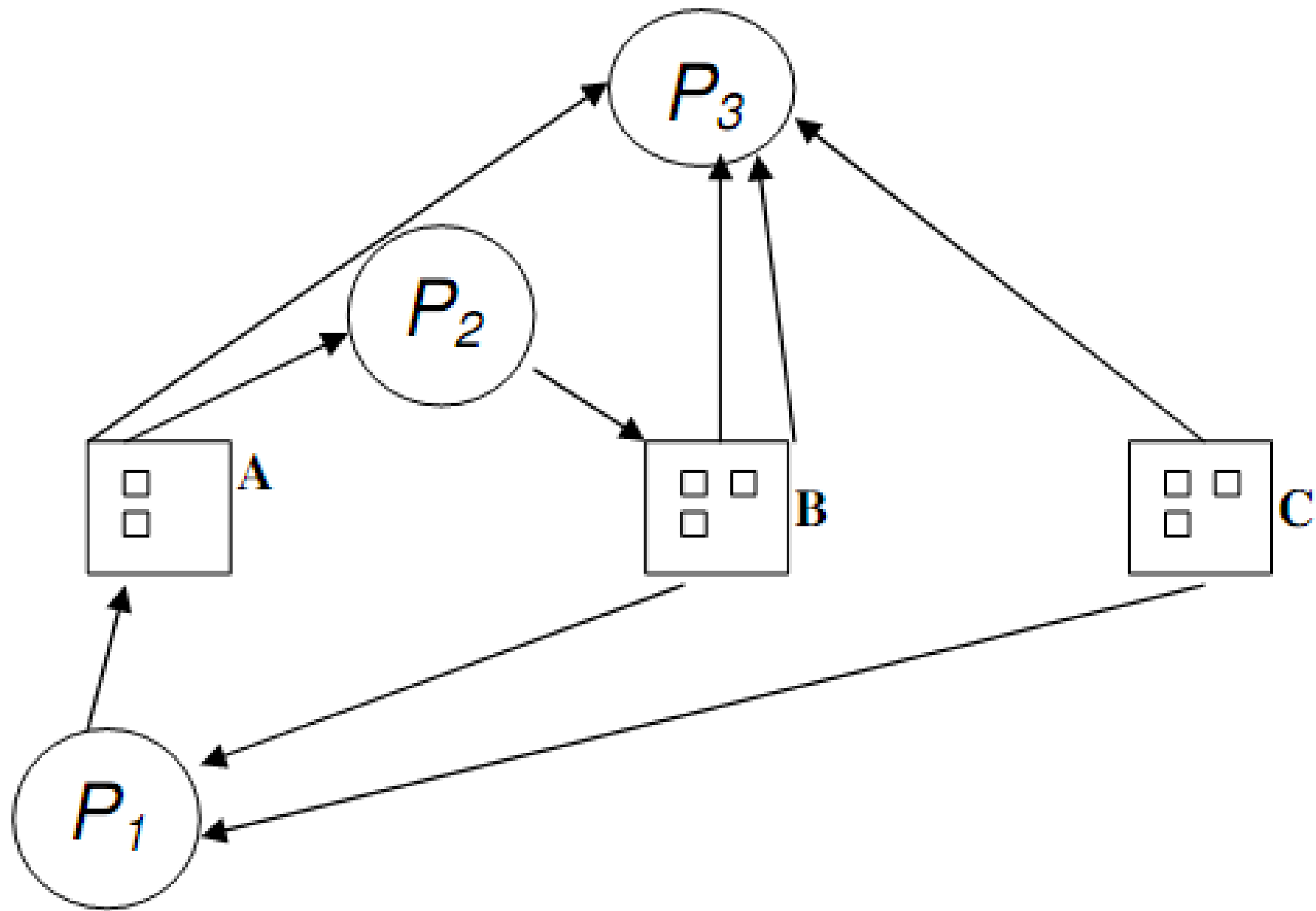
- A directed edge from process  $P_i$  to resource type  $R_j$  is denoted by  $P_i \rightarrow R_j$  is called a request edge; it signifies that process  $P_i$  has requested an instance of resource type  $R_j$  and is currently waiting for that resource.
- A directed edge from resource type  $R_j$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











- 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.

Knowledge Gate

## Deadlock detection and recovery

- Once a dead-lock is detected there are two options for recovery from a deadlock
  - Process Termination
    - Abort all deadlocked processes
    - Abort one process at a time until the deadlock is removed
  - Recourse pre-emption
    - Selecting a victim
    - Partial or Complete Rollback

# Ignorance(Ostrich Algorithm)

1. Operating System behaves like there is no concept of deadlock.
2. Ignoring deadlocks can lead to system performance issues as resources get locked by idle processes.
3. Despite this, many operating systems opt for this approach to save on the cost of implementing deadlock detection.
4. Deadlocks are often rare, so the trade-off may seem justified. Manual restarts may be required when a deadlock occurs.



# Break

[Knowledge Gate Website](#)

# Fork

## Requirement of Fork command

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.



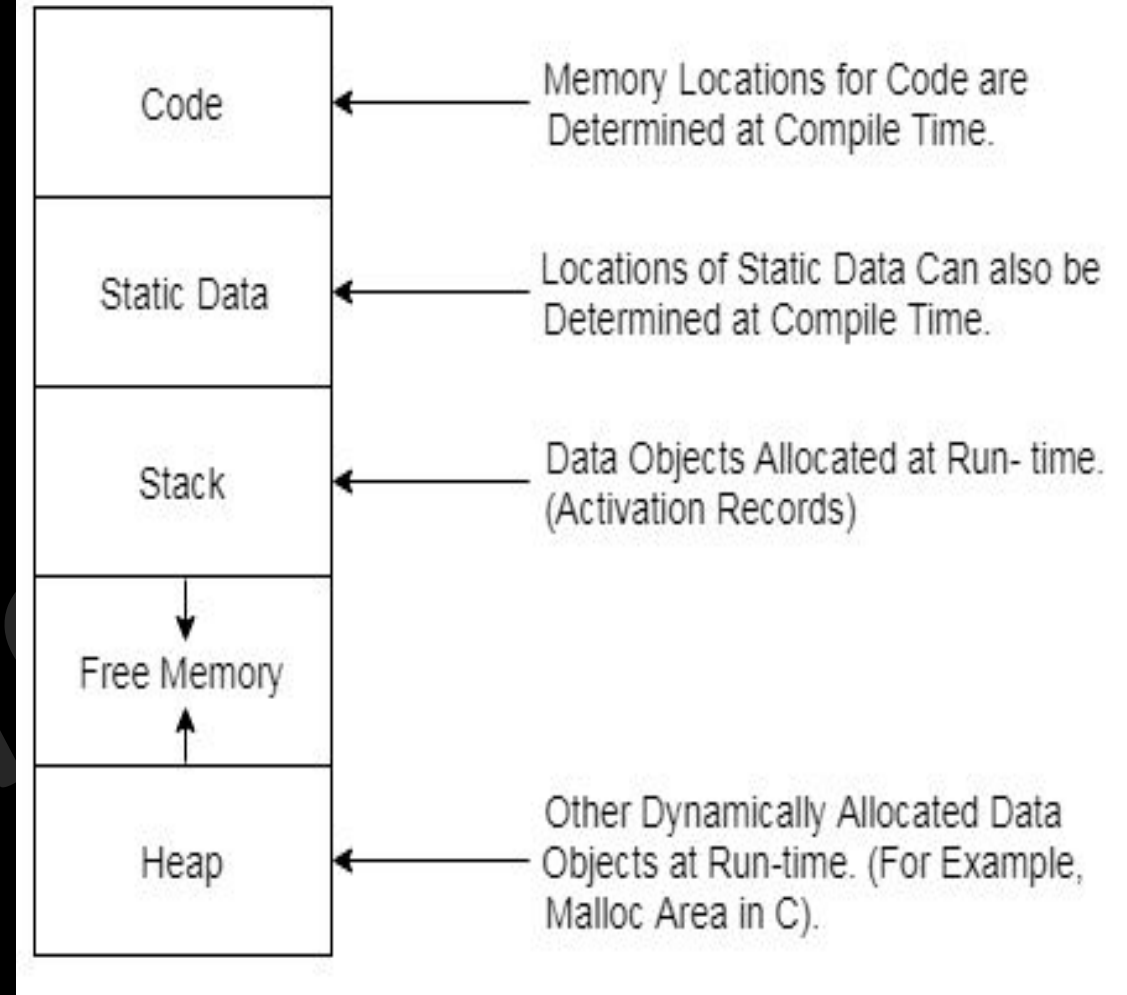
## Idea of fork command

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

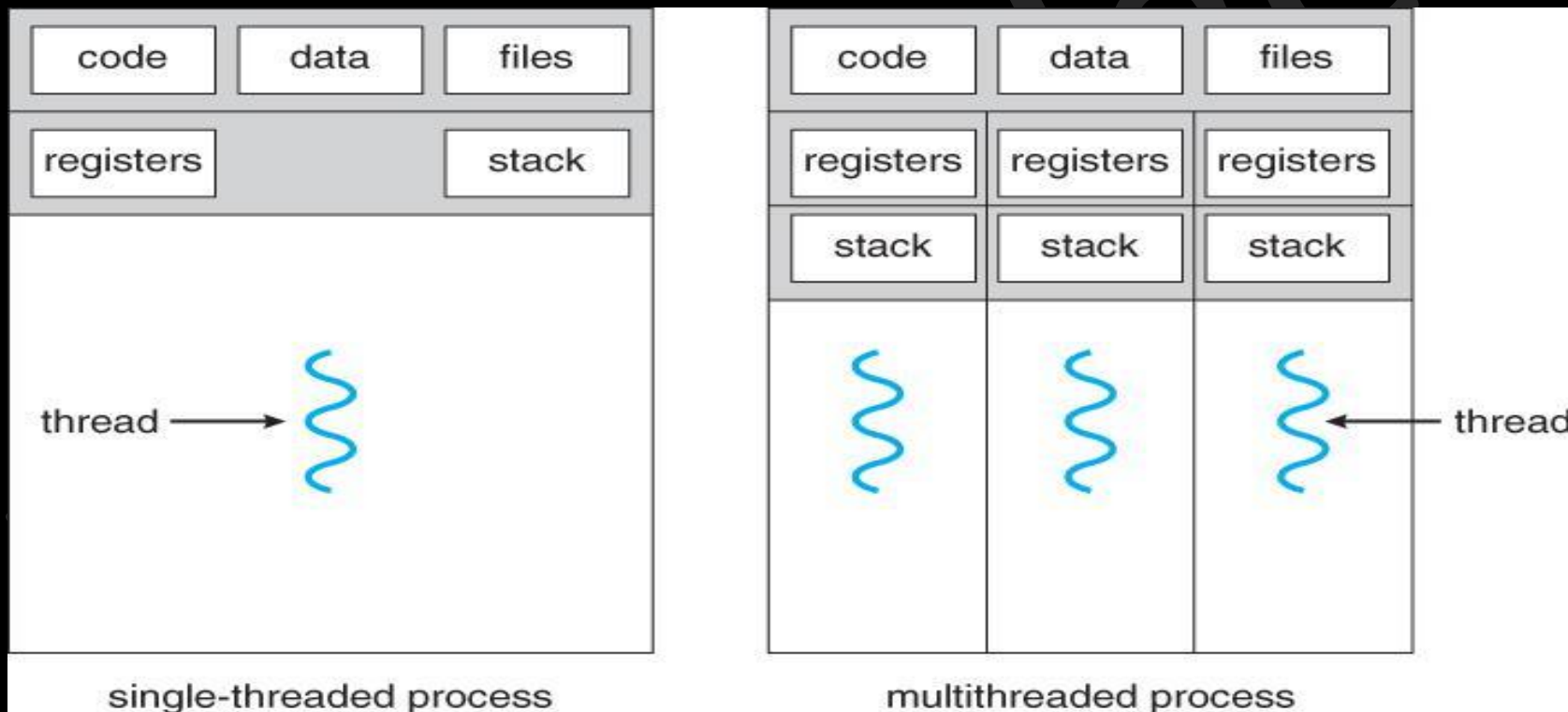
Increase the burden over Operating System

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

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. )

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.



## 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.

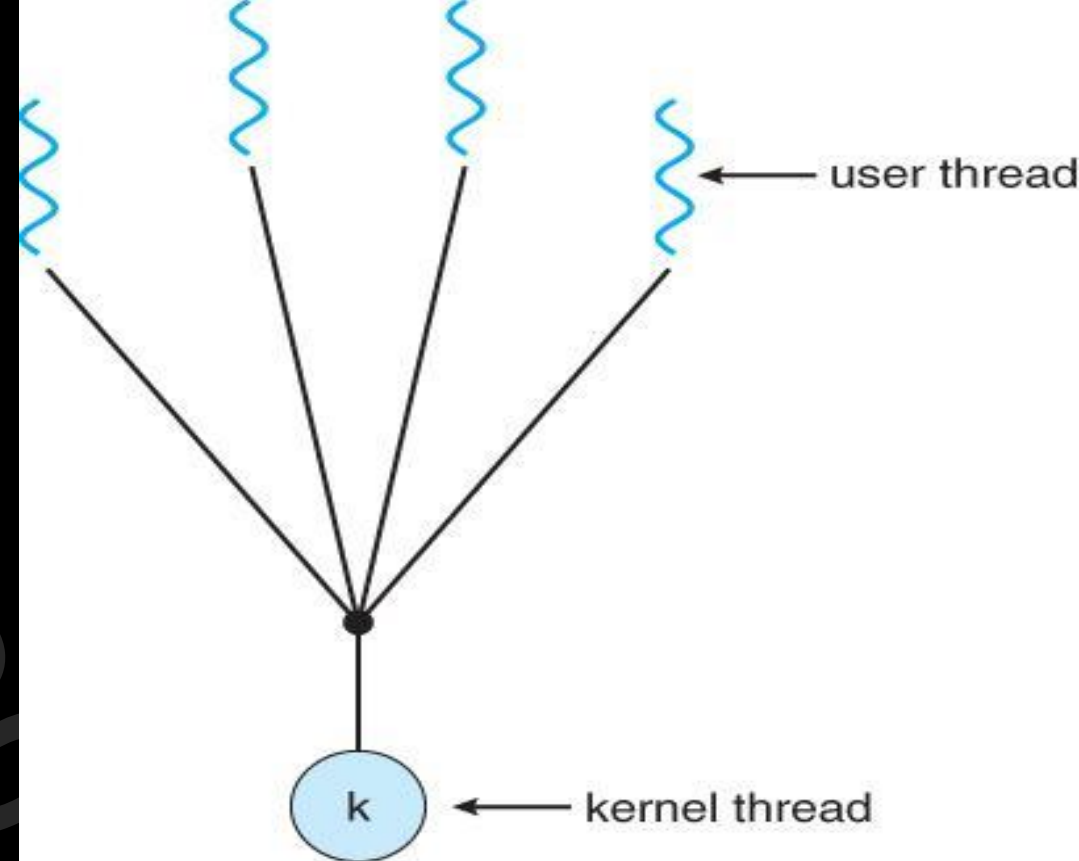
## Many-To-One Model

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.

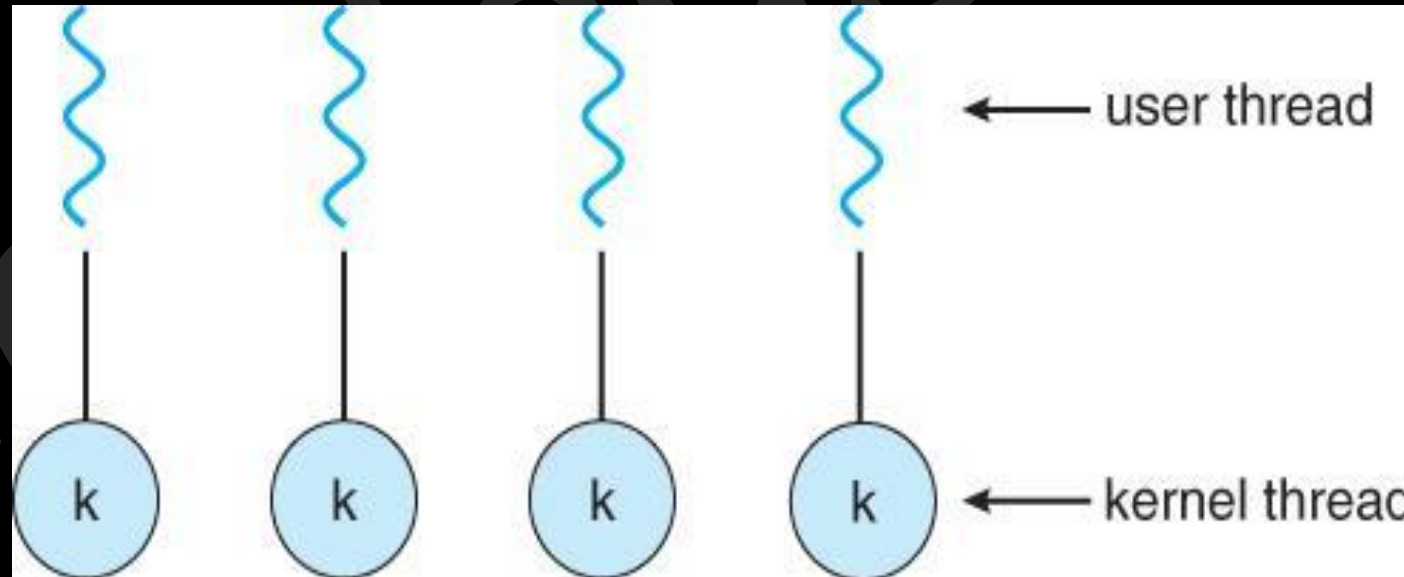


## One-To-One Model

The one-to-one model creates a separate kernel thread to handle each user thread. It 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.

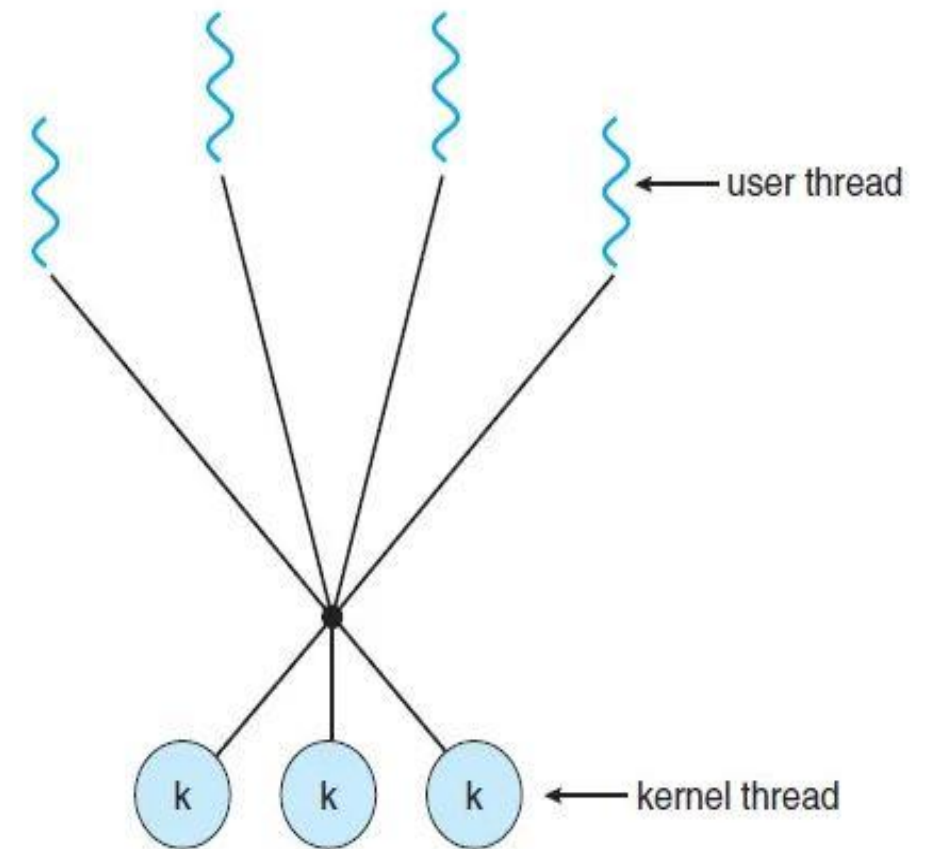


## Many-To-Many Model

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.





Knowledge Gate

[Knowledge Gate Website](#)

# Memory Hierarchy

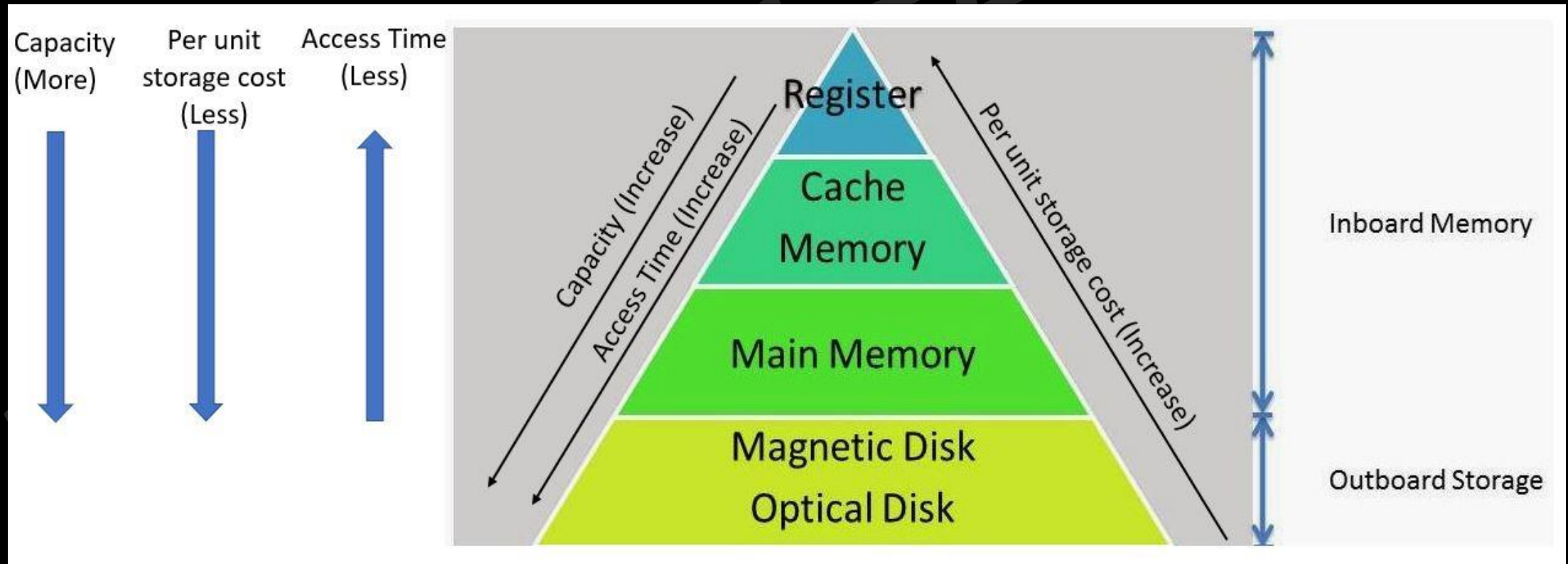
Let first understand what we need from a memory

Large capacity

Less per unit cost

Less access time(fast access)

The memory hierarchy system consists of all storage devices employed in a computer system.





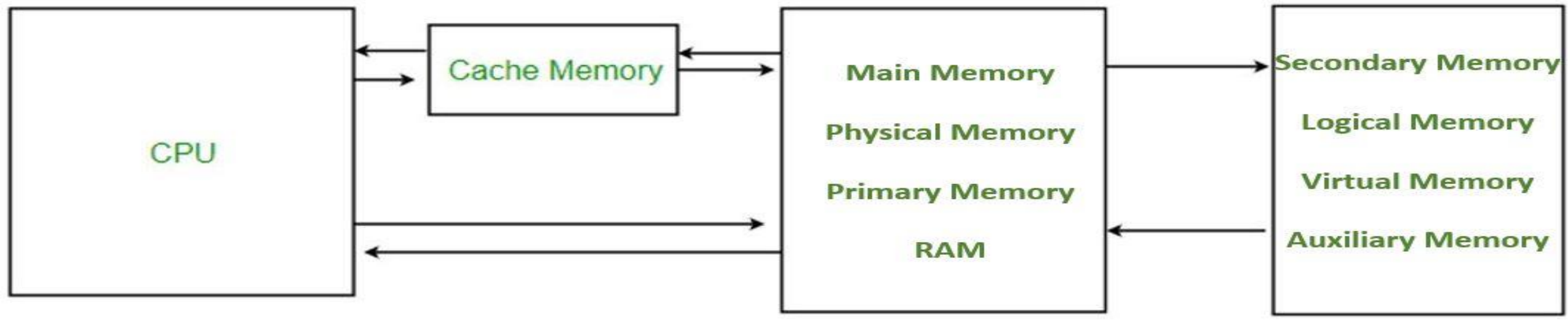
Cycle



Car



Airbus



Showroom

4 MB



Go down

32 GB



Factory

8 TB

## 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:** Use of data elements in the nearby locations.

**Temporal Locality:** Temporal locality refers to the reuse of specific data or resources, within a relatively small-time duration, i.e. Most Recently Used.

# Duty of Operating System

Operating system is responsible for the following activities in connection with memory management:

1. **Address Translation**: Convert logical addresses to physical addresses for data retrieval.
2. **Memory Allocation and Deallocation**: Decide which processes or data segments to load or remove from memory as needed.
3. **Memory Tracking**: Monitor which parts of memory are in use and by which processes.
4. **Memory Protection**: Implement safeguards to restrict unauthorized access to memory, ensuring both process isolation and data integrity.



There can be two approaches for storing a process in main memory.

**Contiguous allocation policy**

**Non-contiguous allocation policy**

Knowledge Gate



## 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.

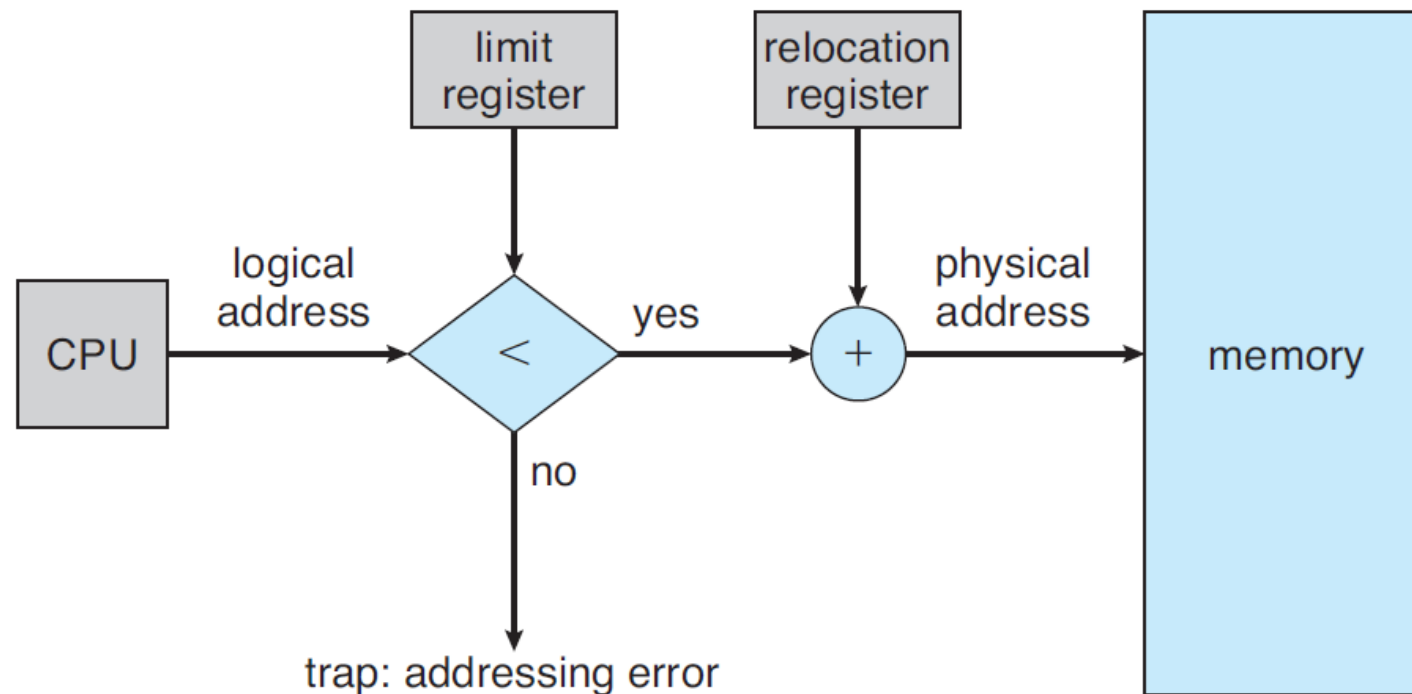
Must be stored in main memory in contiguous fashion.

## Address Translation in Contiguous Allocation

Here we use a Memory Management Unit(OS) which 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.

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.



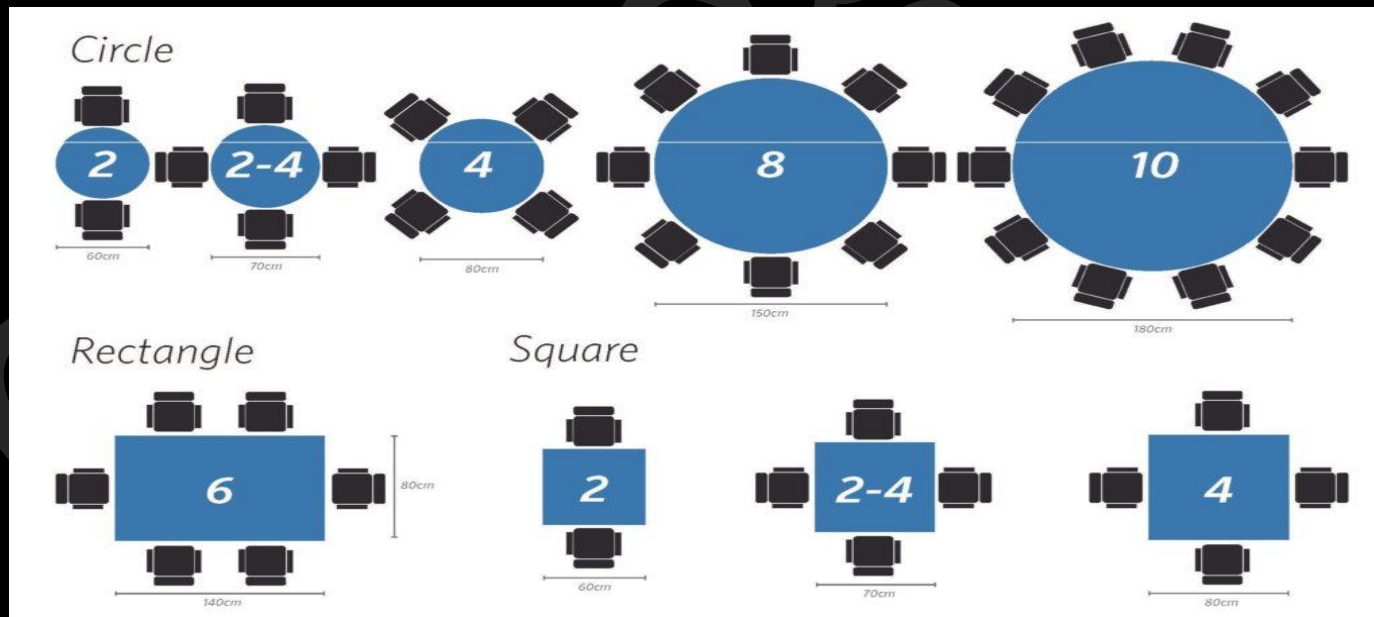
## Space Allocation Method in Contiguous Allocation

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.

XCVXC



**First fit policy:** - 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

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)

**Best fit policy:** - 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.

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)

**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.

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)



Q Consider five memory partitions of size 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB, where KB refers to kilobyte. These partitions need to be allotted to four processes of sizes 212 KB, 417 KB, 112 KB and 426 KB in that order?

100	500	200	300	600

100	500	200	300	600

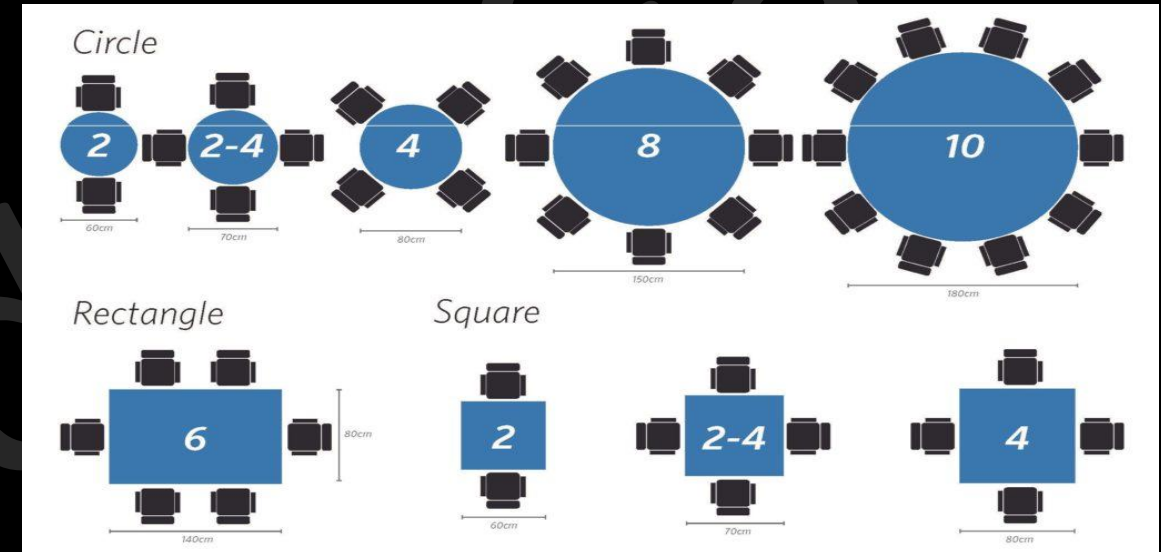
100	500	200	300	600

**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.

Knowledge Gate

[Knowledge Gate Website](#)

**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.



**The Big Cason Family want to celebrate a party**

**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?

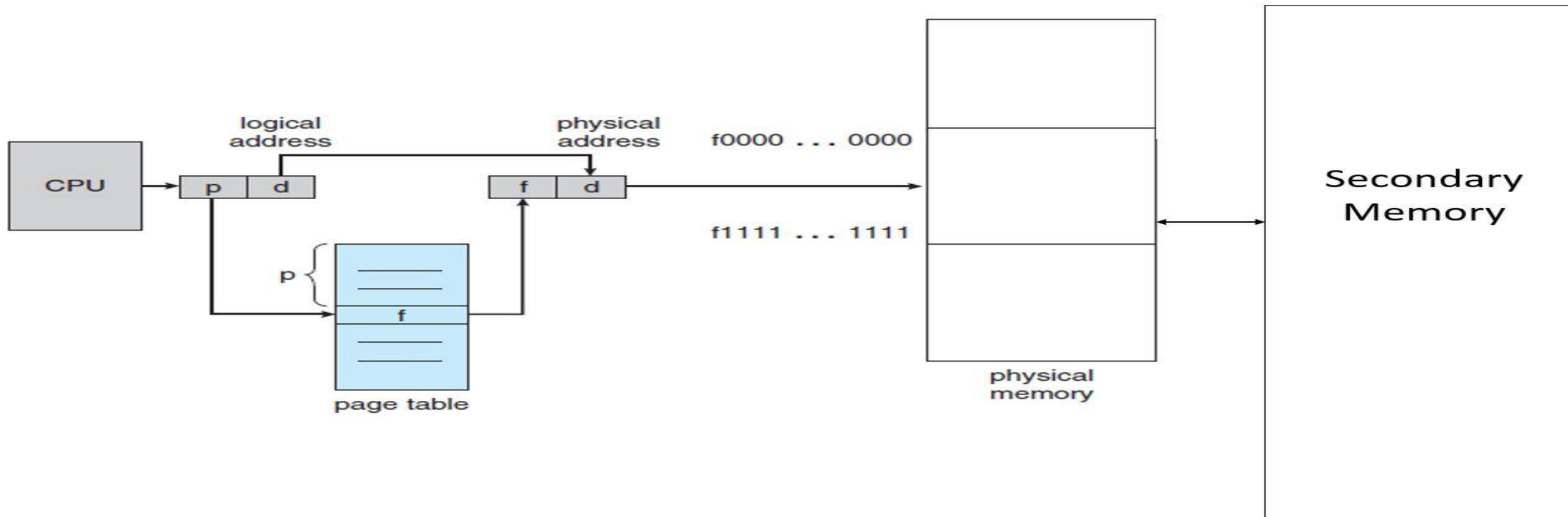
We can also swap processes in the main memory after fixed intervals of time & they can be swapped in one part of the memory and the other part become empty (Compaction, defragmentation). This solution is very costly in respect to time as it will take a lot of time to swap process when system is in running 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



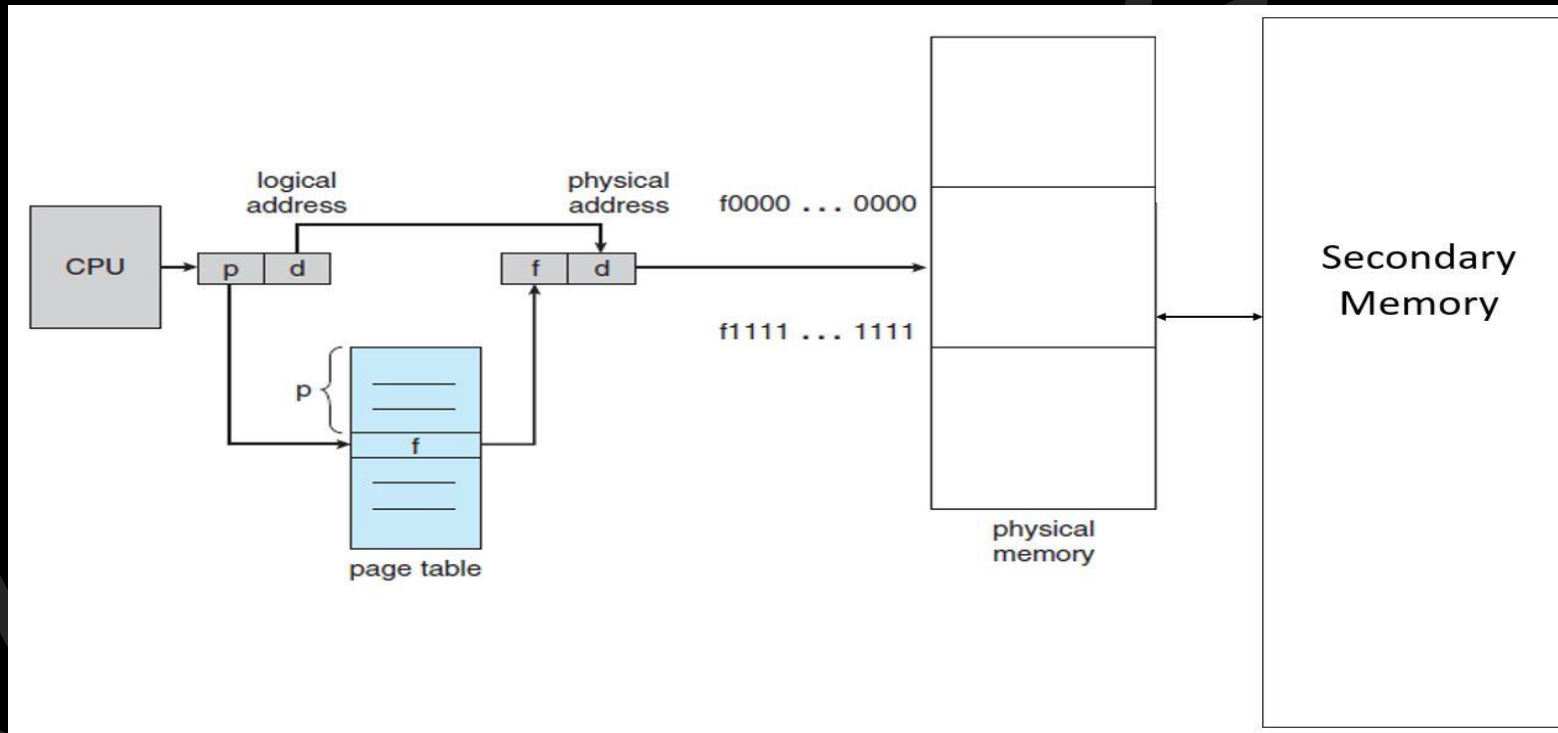


Secondary memory is divided into fixed size partition (because management is easy) all of them of same size called pages (easy swapping and no external fragmentation).

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

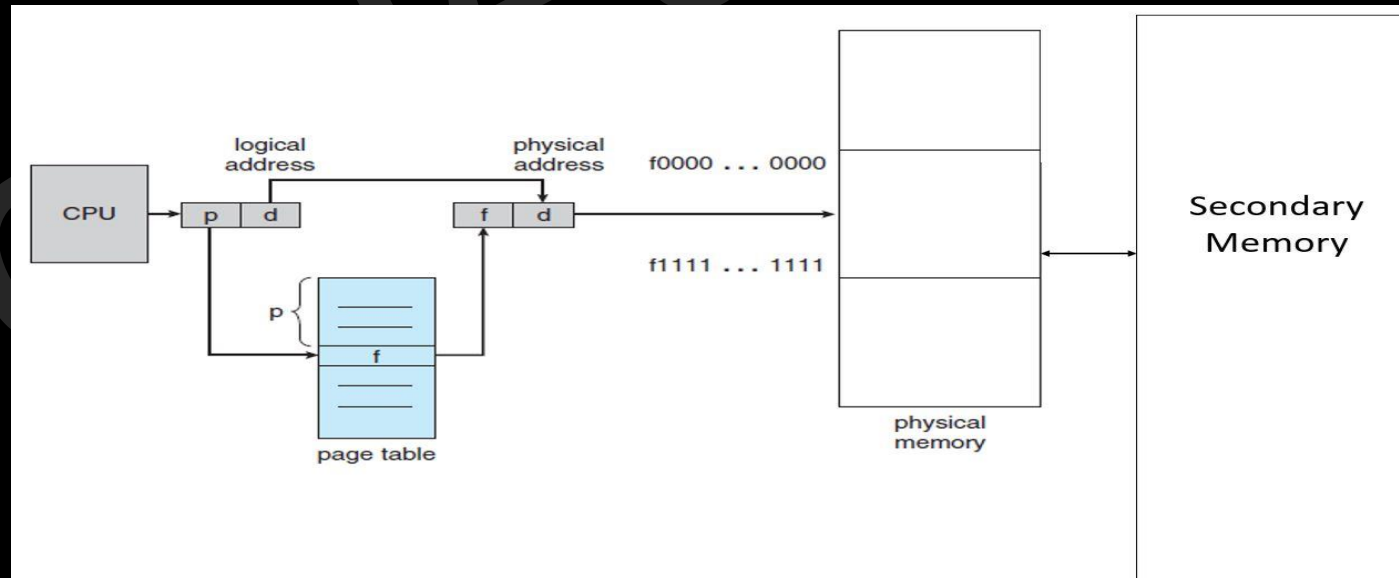
CPU generate a logical address is divided into two parts - **p** and **d** where **p** stands for page no and **d** stands for instruction offset.

The **page number(p)** is used as an index into a **Page table**

**Page table base register(PTBR)** provides the base of the page table and then the corresponding page no is accessed using **p**.

Here we will find the corresponding frame no (the base address of that frame in main memory in which the page is stored)

Combine corresponding frame no with the instruction offset and get the physical address. Which is used to access main memory.



# Page Table

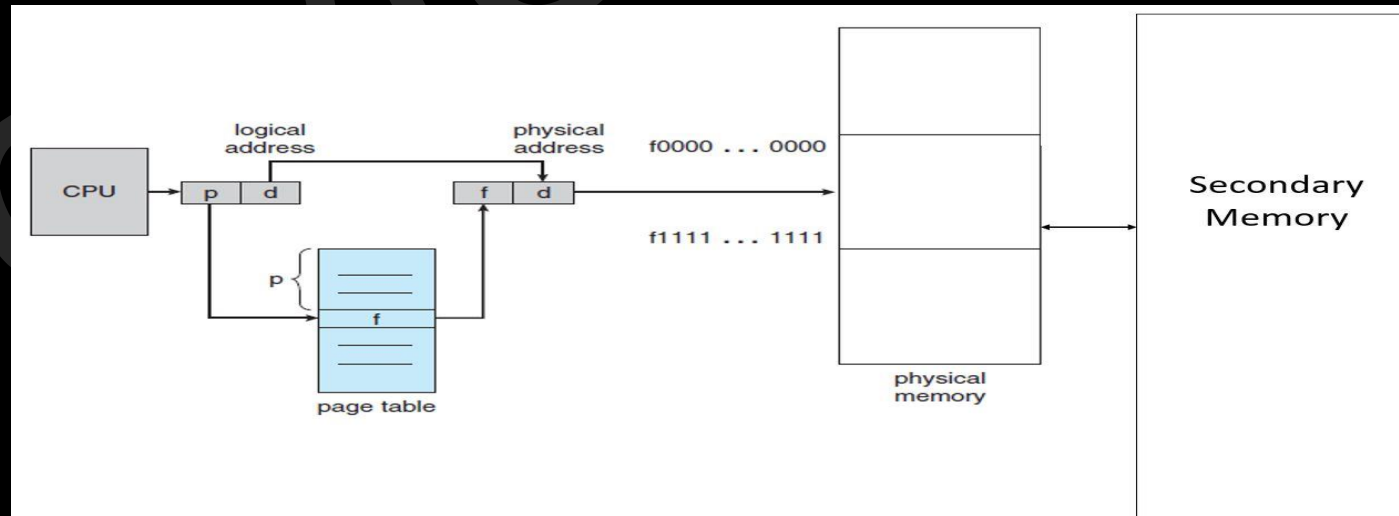
Page table is a data structure not hardware.

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).

Translation process is difficult and complex to understand and implement.

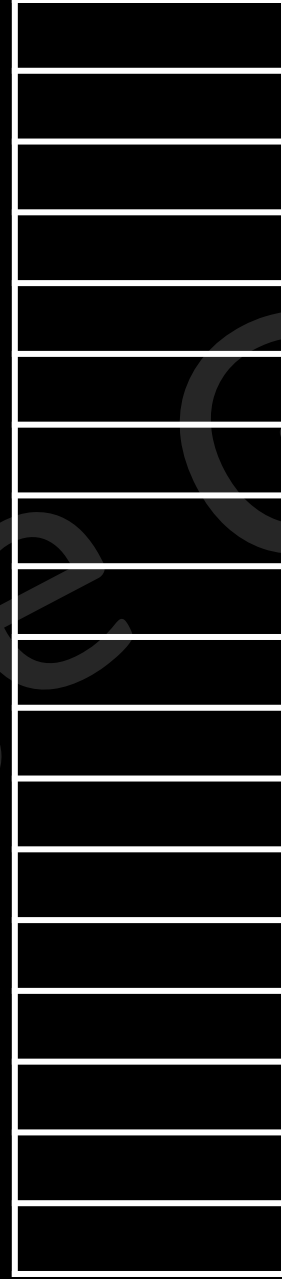
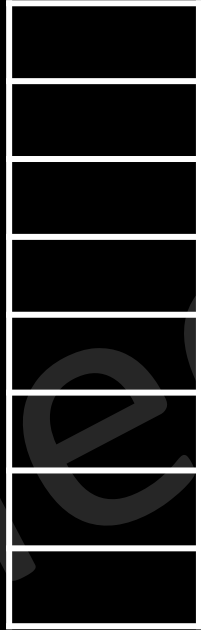
$10^3$	1 Thousand	$10^3$	1 kilo	$2^{10}$	1 kilo
$10^6$	1 Million	$10^6$	1 Mega	$2^{20}$	1 Mega
$10^9$	1 Billion	$10^9$	1 Giga	$2^{30}$	1 Giga
$10^{12}$	1 Trillion	$10^{12}$	1 Tera	$2^{40}$	1 Tera
		$10^{15}$	1 Peta	$2^{50}$	1 Peta
		$10^{18}$	1 Exa	$2^{60}$	1 Exa
		$10^{21}$	1 Zetta	$2^{70}$	1 Zetta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

Address Length in bits	
No of Locations	$2^n$

Knowledge Gate

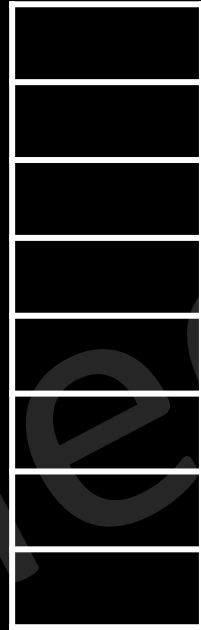
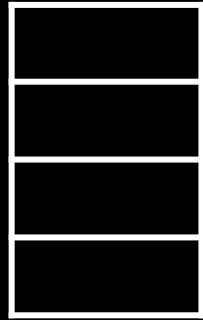
[Knowledge Gate Website](http://www.knowledgegate.com)

Address Length in bits	
No of Locations	$2^n$



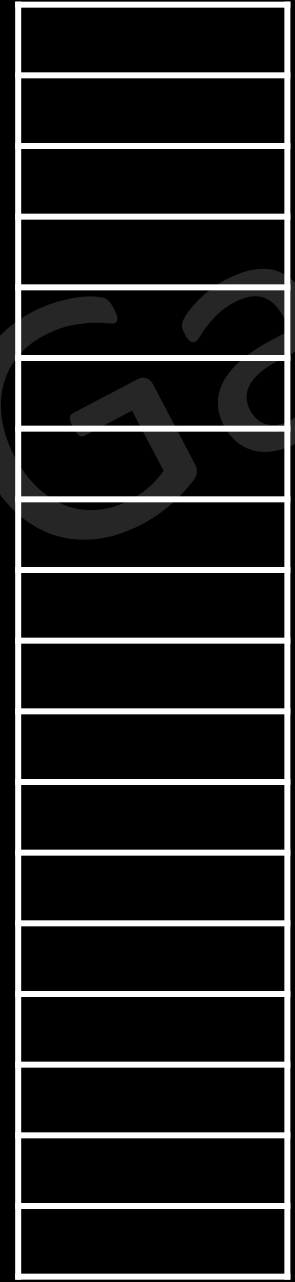
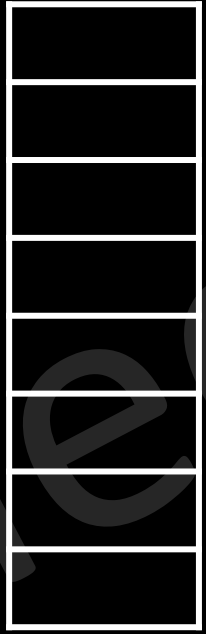
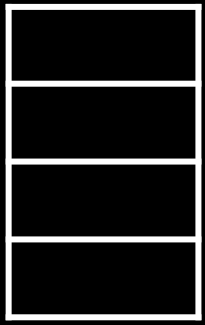
Memory Size = Number of Location \* Size of each Location

<b>Address Length in bits</b>	<b>Upper Bound(<math>\log_2 n</math>)</b>
<b>No of Locations</b>	<b>n</b>





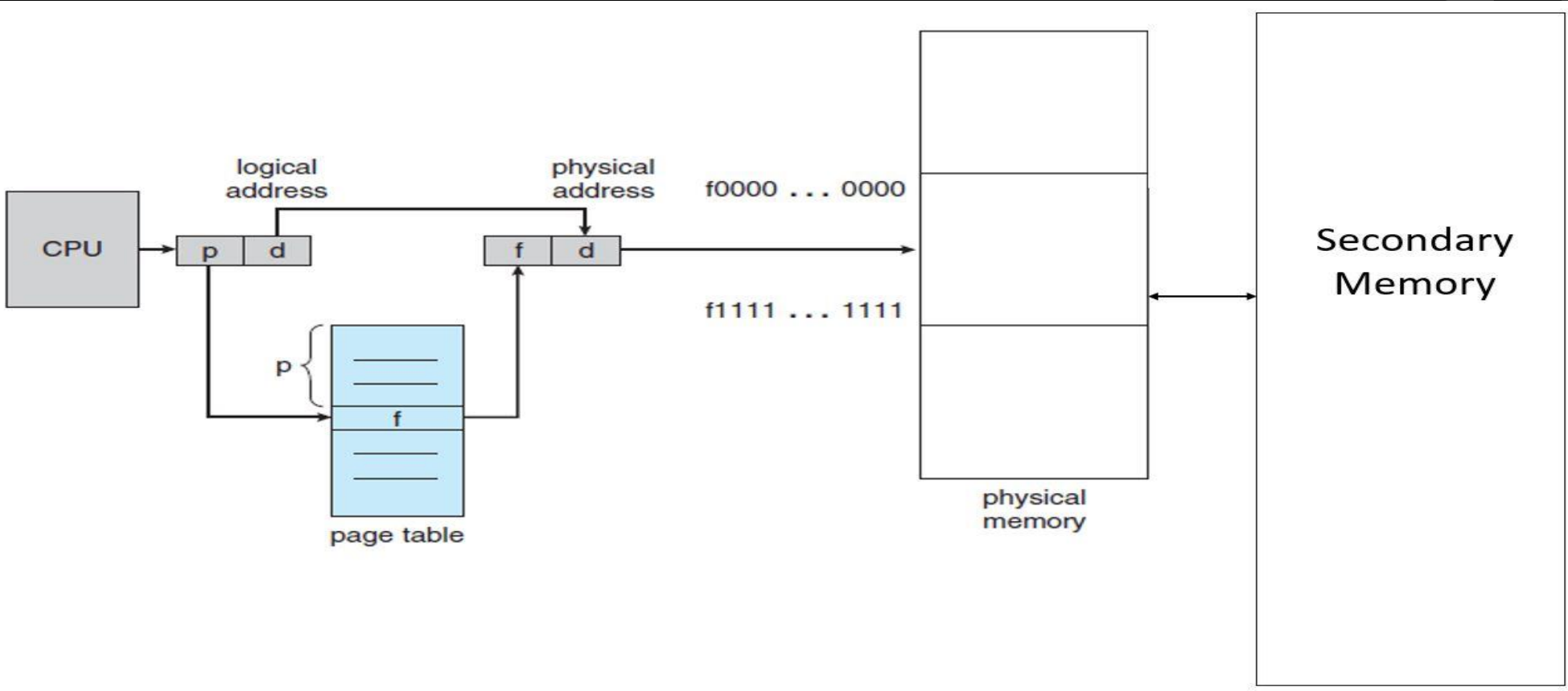
Address Length in bits	Upper Bound( $\log_2 n$ )
No of Locations	n



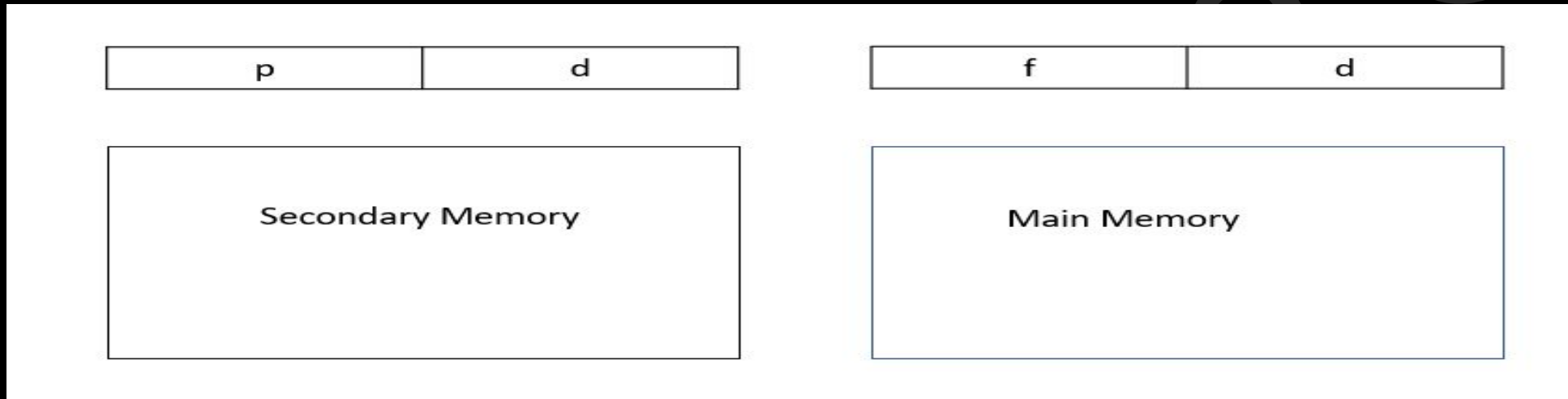
Memory Size/ Size of each Location = Number of Location

Page Table Size = No of entries in Page table \* Size of each entry(f)

Process Size = No of Pages \* Size of each page

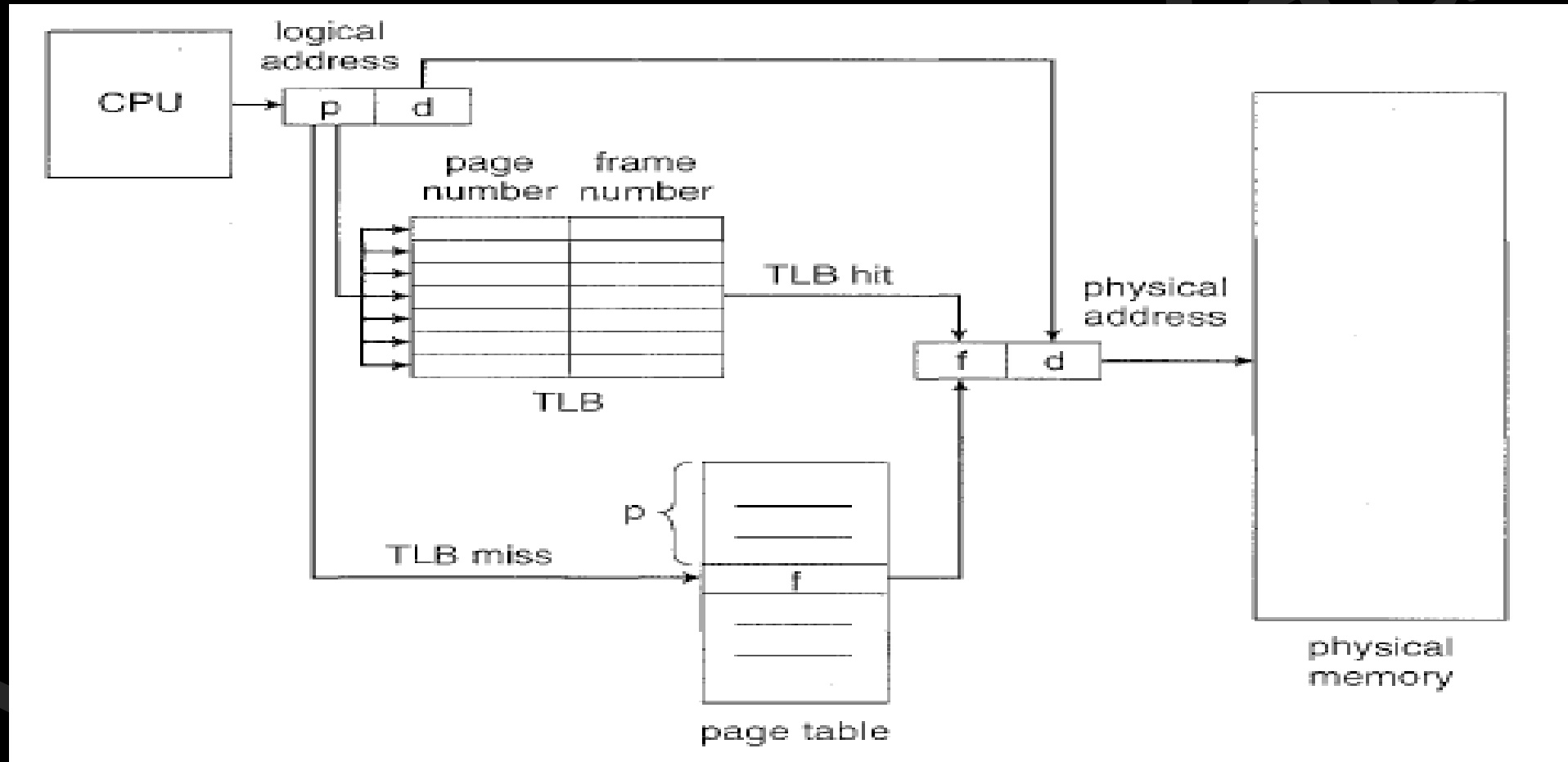


S No	SM	LA	MM	PA	p	f	d	addressable	Page Size
1	32 GB		128 MB					1B	1KB
2		42		33			11	1B	
3	512GB			31				1B	512B
4	128GB		32GB		30			1B	
5					28	14			4096B



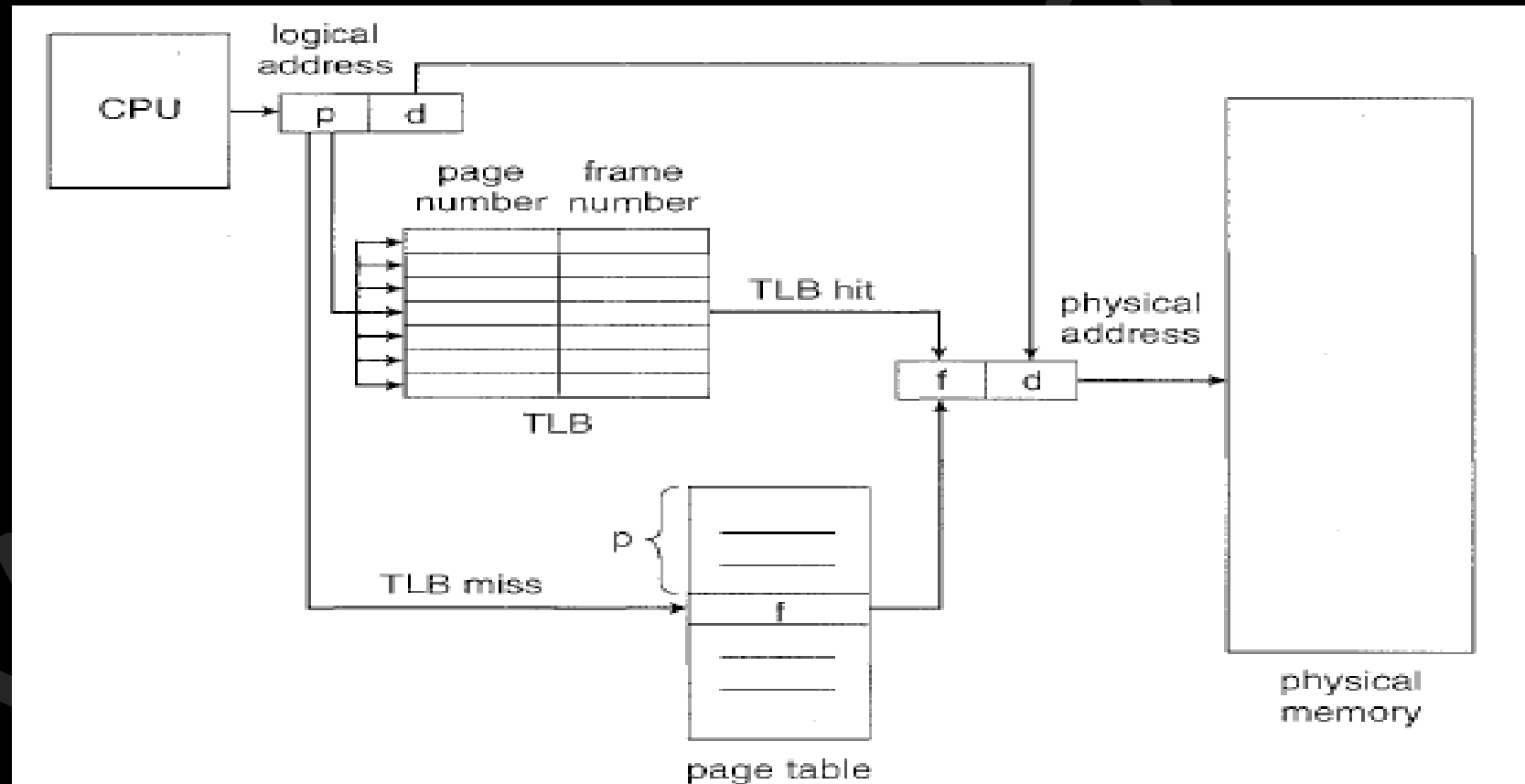
A serious problem with page is, translation process is slow as page table is accessed two times (one for page table and other for actual access).

To solve the problems in paging we take the help of TLB. The TLB is associative, high-speed memory.



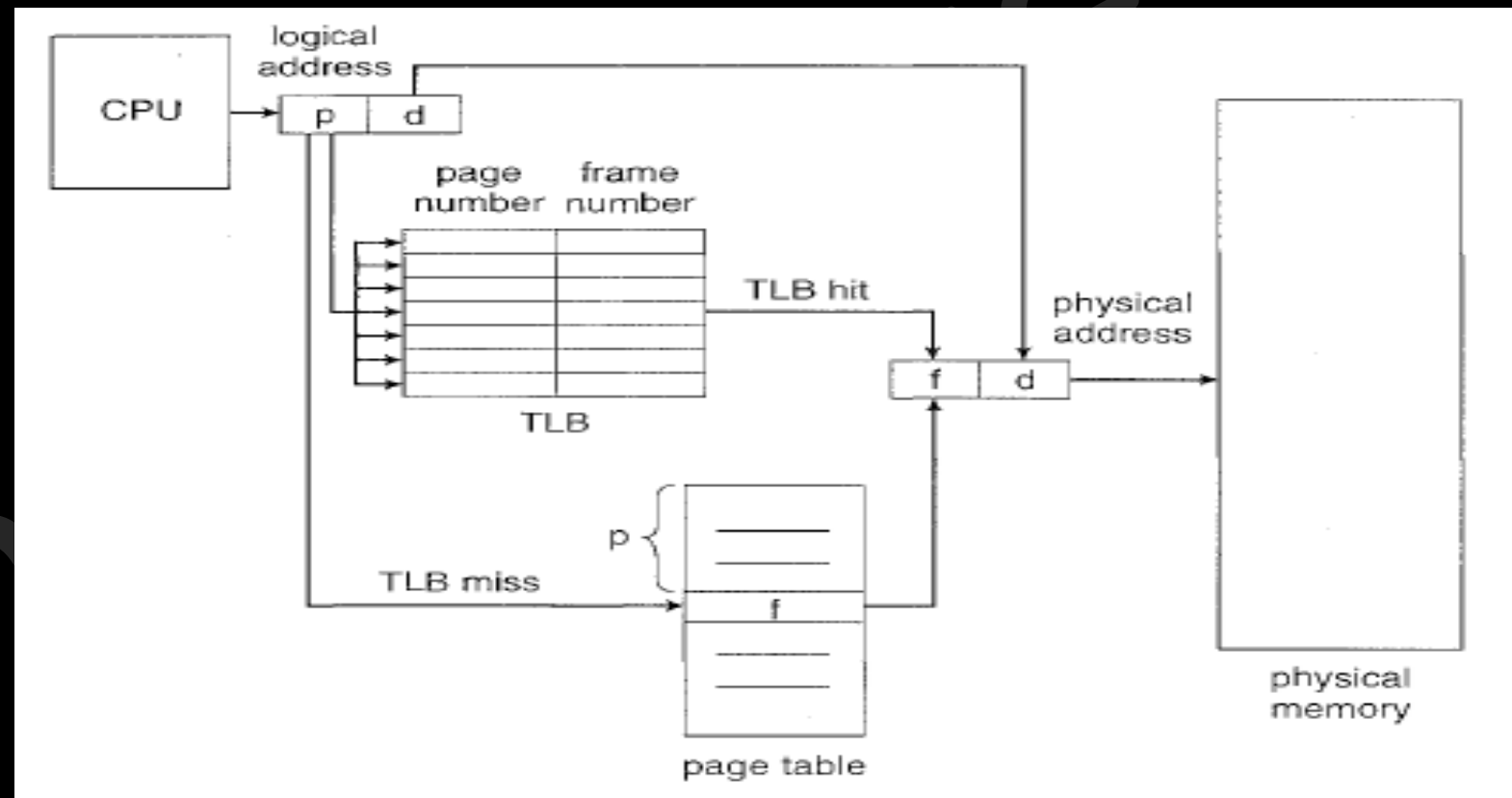
Each entry in the TLB consists of two parts: a key (Page no) and a value (frame no). When the associative memory is search for page no, the page no is compared with all page no simultaneously. If the item is found, the corresponding frame no field is returned.

The search is fast; the hardware, however, is expensive, TLB Contains the frequently referred page numbers and corresponding frame number.



The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory.

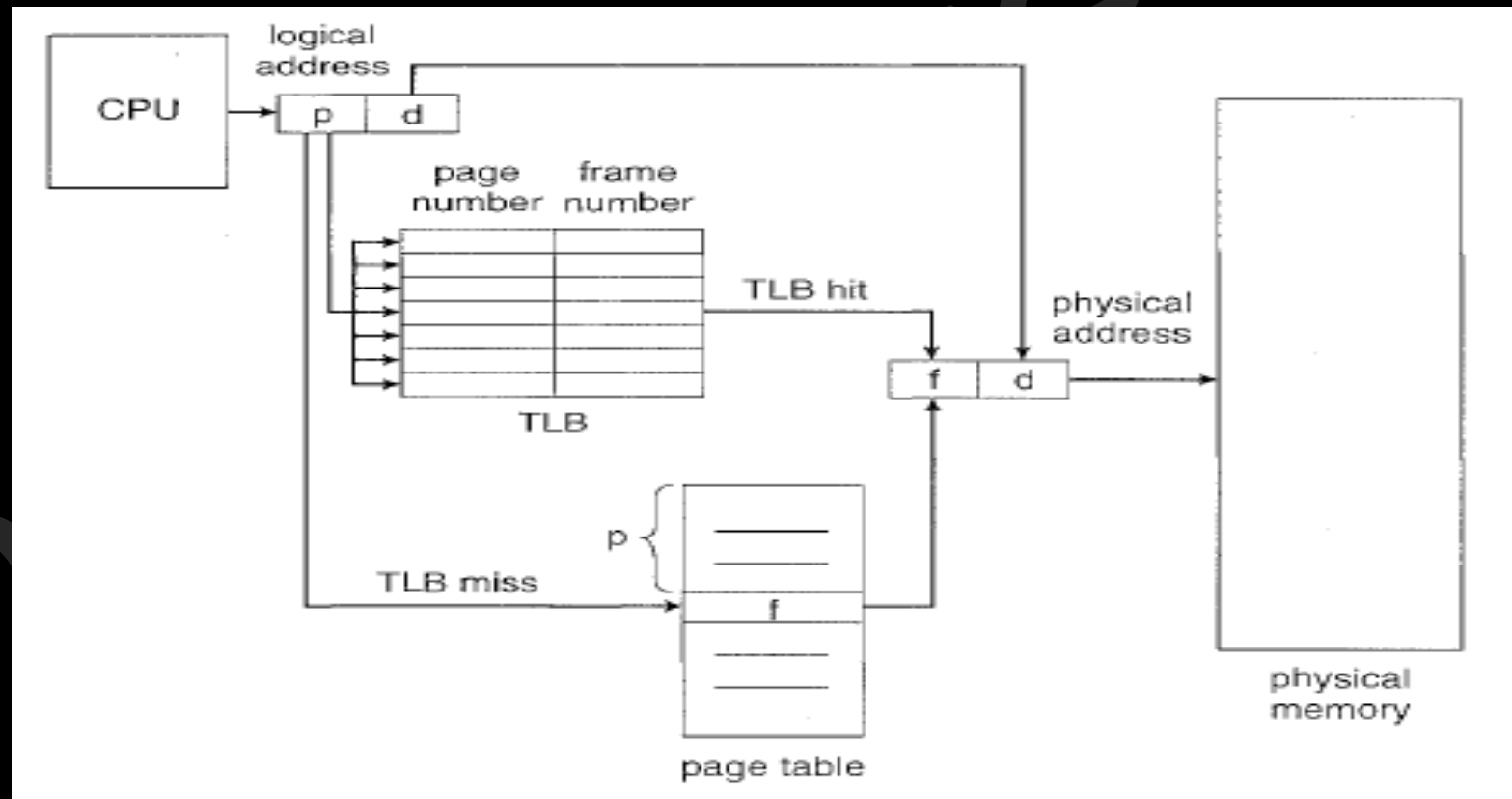
If the page number is not in the TLB (known as a **TLB Miss**), then a memory reference to the page table must be made.



Also we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, the operating system must select one for replacement i.e. Page replacement policies.

The percentage of times that a particular page number is found in the TLB is called the **Hit Ratio**.

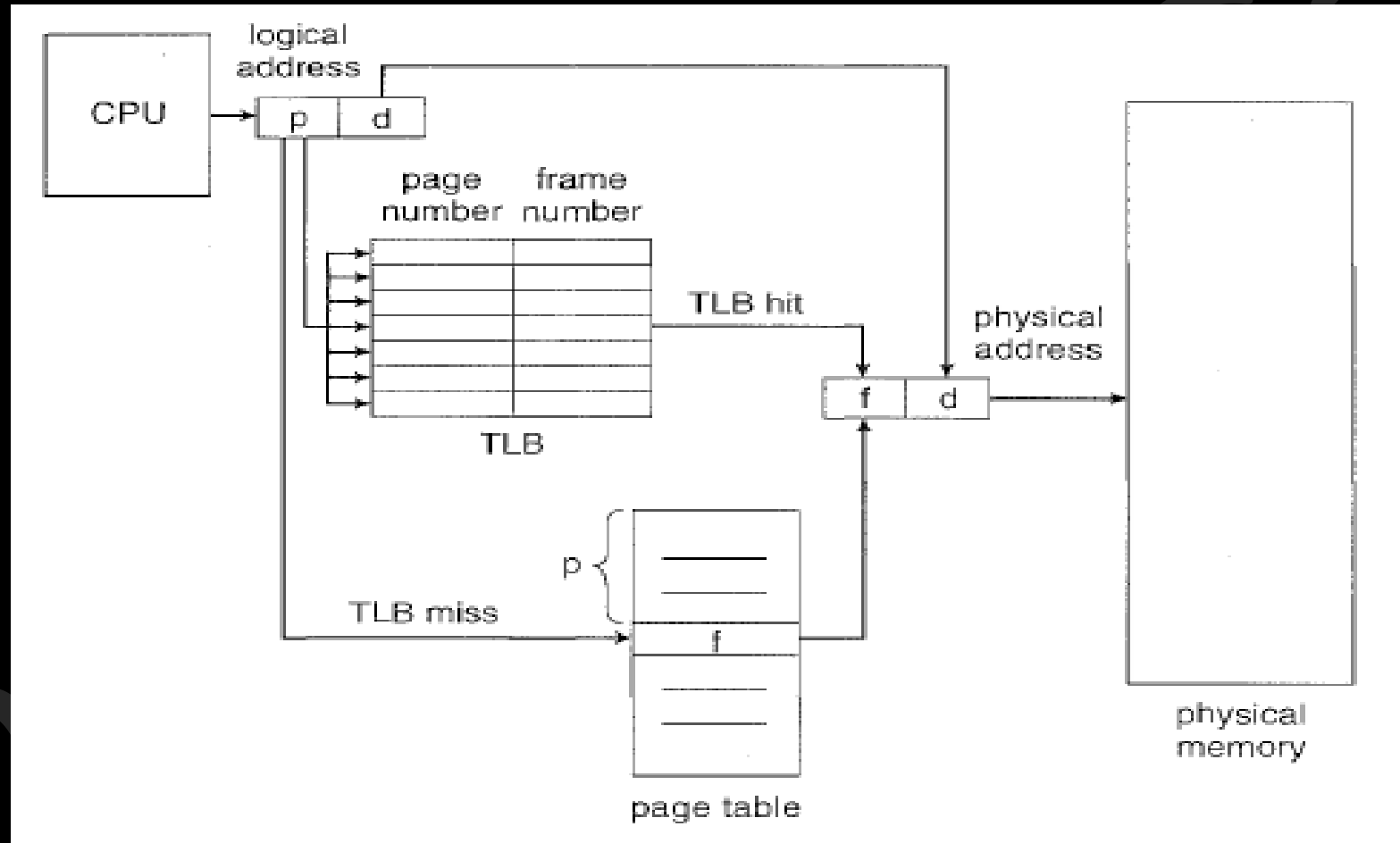




## Effective Memory Access Time:

Hit [TLB + Main Memory] + 1-Miss [TLB + 2 Main Memory]

TLB removes the problem of slow access.

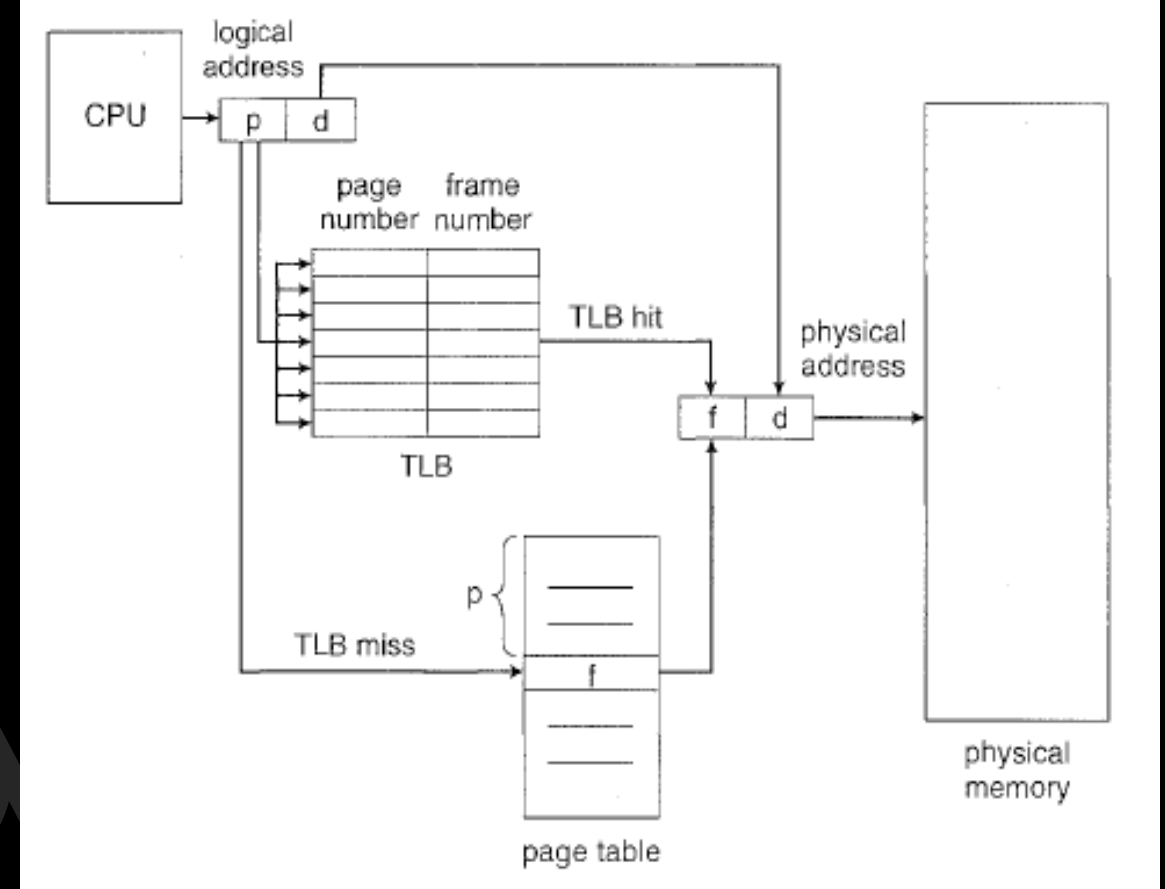


## Disadvantage of TLB:

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.



## 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.

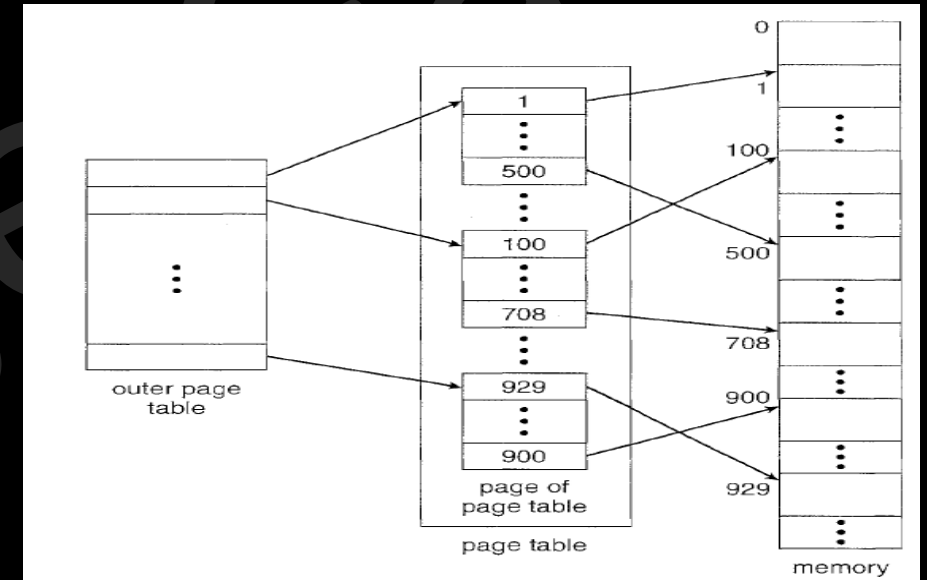
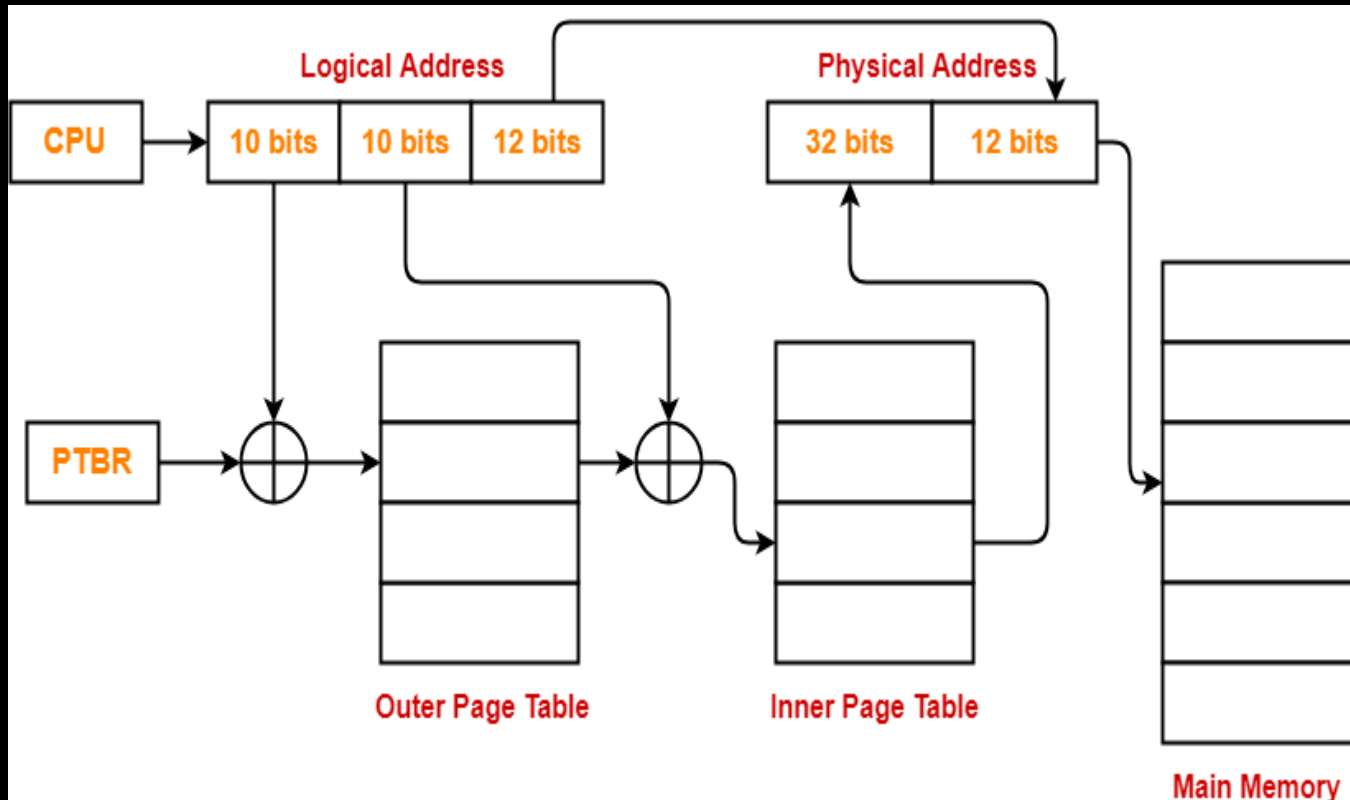
## Multilevel Paging / Hierarchical Paging

Modern systems support a large logical address space ( $2^{32}$  to  $2^{64}$ ).

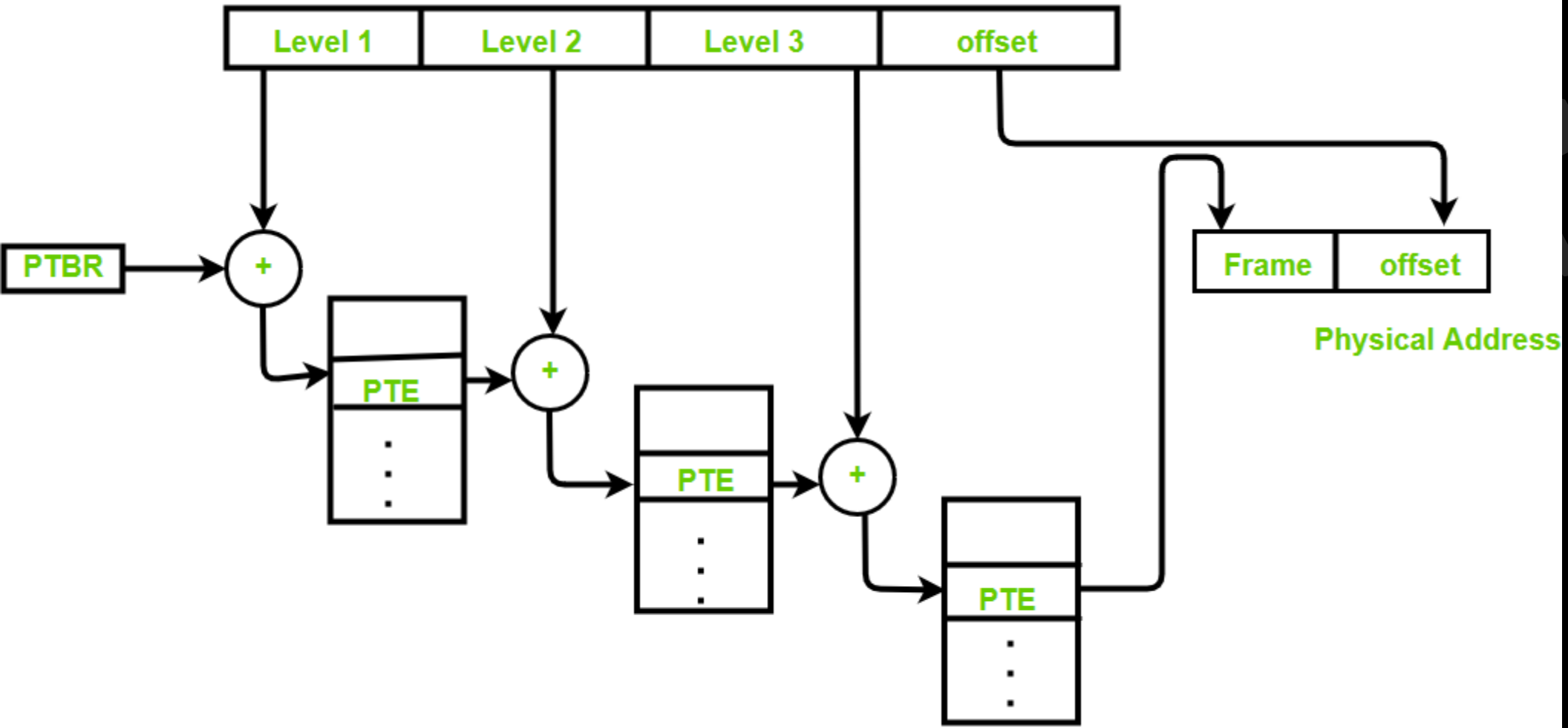
In such cases, the page table itself becomes excessively large and can contain millions of entries and can take a lot of space in memory, so cannot be accommodated into a single frame.

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



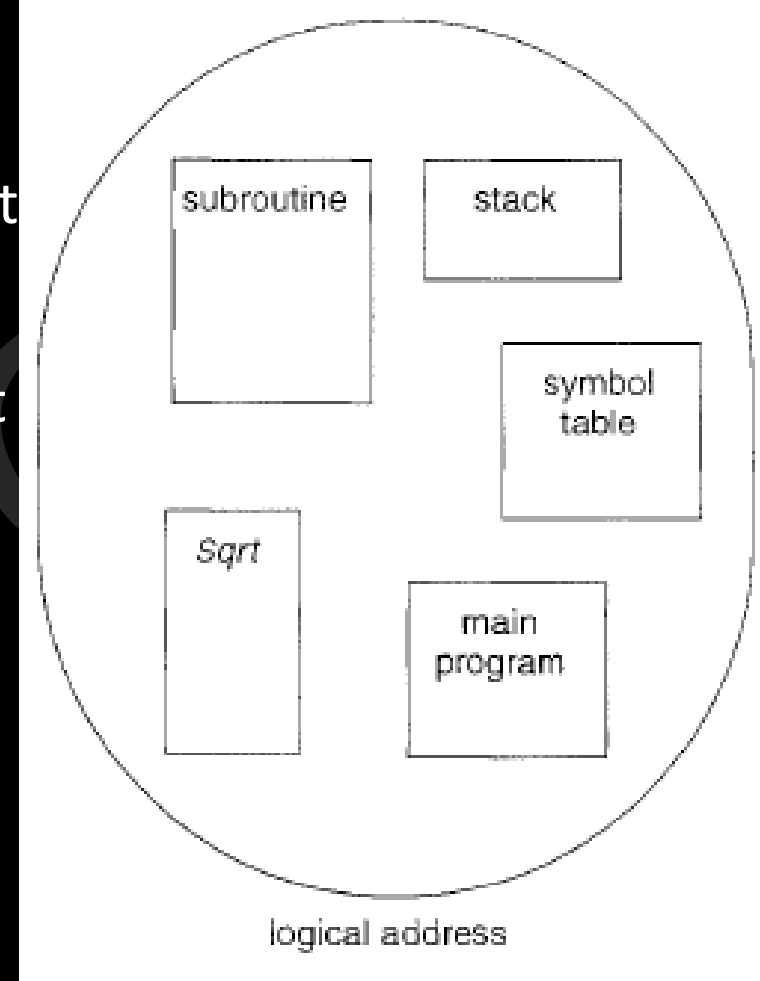
3 Level paging system

## 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.**

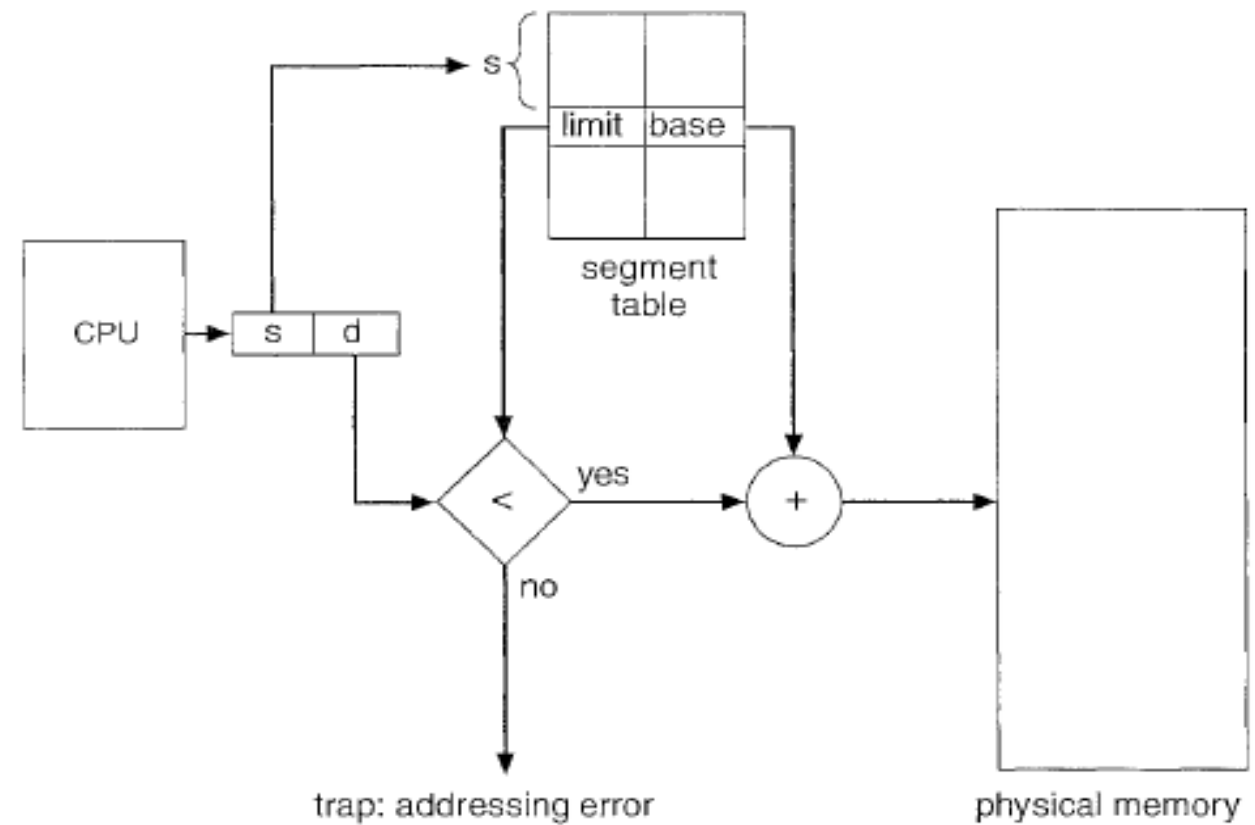
Segments can be of variable lengths unlike pages and are stored in main memory.



**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**.



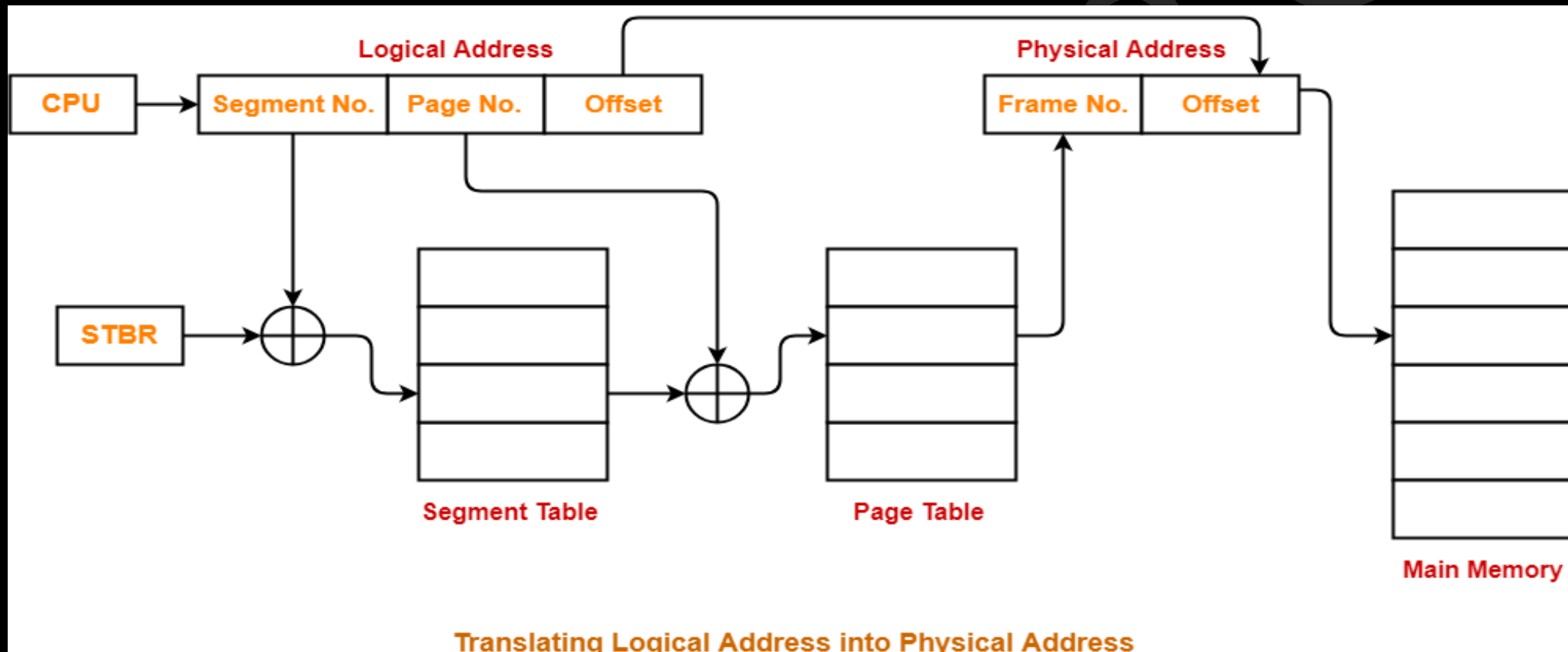


## Segmentation with Paging

Since segmentation also suffers from external fragmentation, it is better to divide the segments into pages.

In segmentation with paging 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 same as multilevel paging.

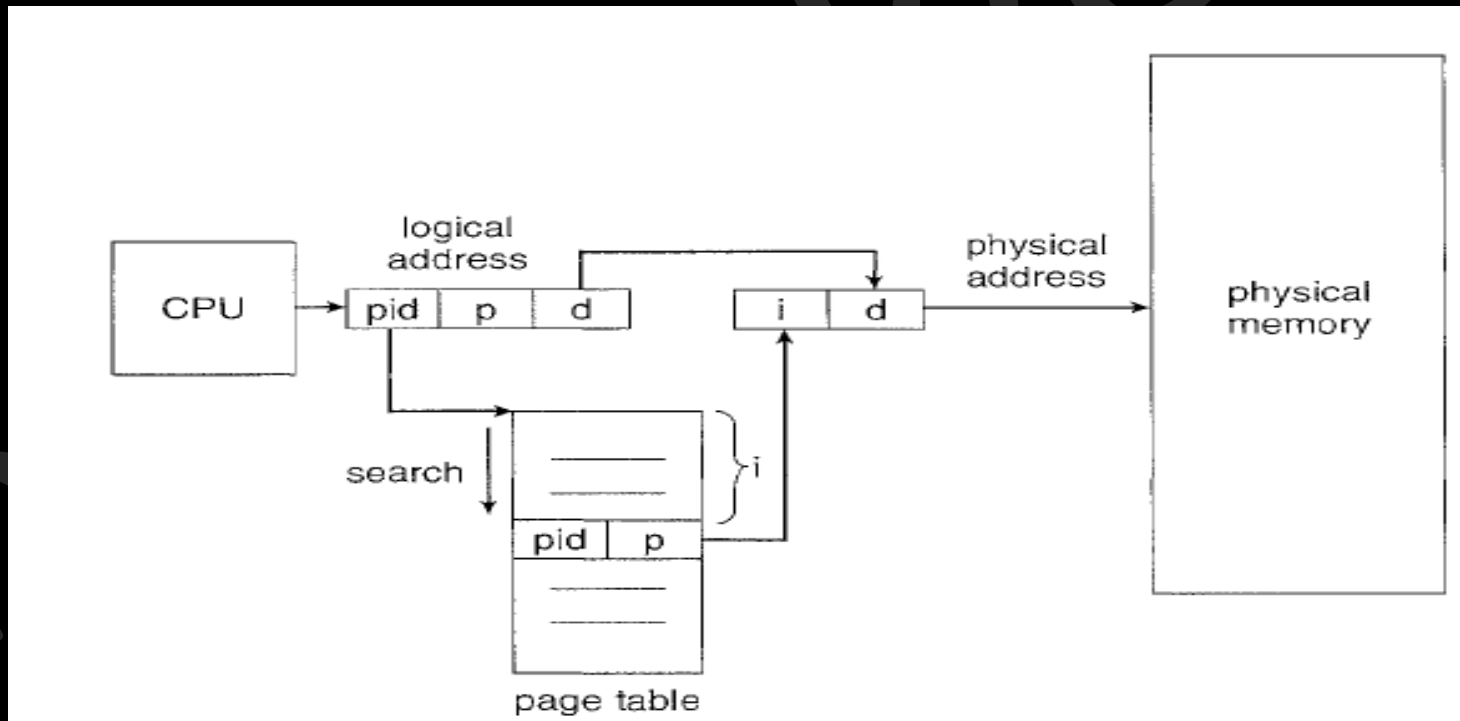


## Inverted Page Table

The drawback of paging 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

To enable multiprogramming and optimize memory, modern computing often uses pure demand paging to keep multiple processes in memory.

2. **Pure Demand Paging:** A memory management scheme where a process starts with no pages in memory. Pages are only loaded when explicitly required during execution.
  1. Process starts with zero pages in memory. Immediate page fault occurs.
  2. Load the needed page into memory. Execution resumes after the required page is loaded into memory.
  3. Additional page faults occur as the process continues and requires new pages.
  4. Execution proceeds without further faults once all necessary pages are in memory. The key principle is to only load pages when absolutely necessary.

## 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.

## Disadvantages

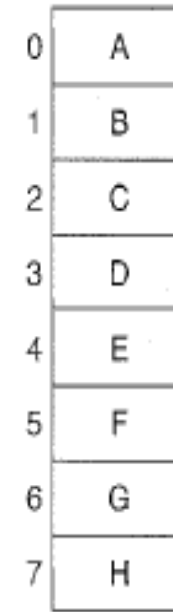
Virtual memory is not easy to implement.

It may substantially decrease performance if it is used carelessly (Thrashing)

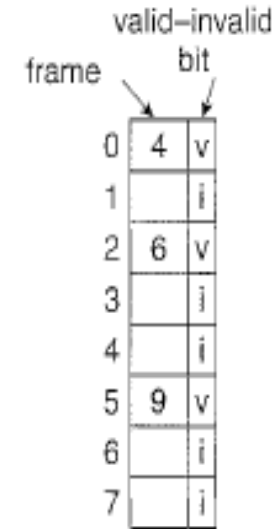
# Implementation of Virtual Memory

We add a new column in page table, which have binary value 0 or Invalid which means page is not currently in main memory, 1 or valid which means page is currently in main memory.

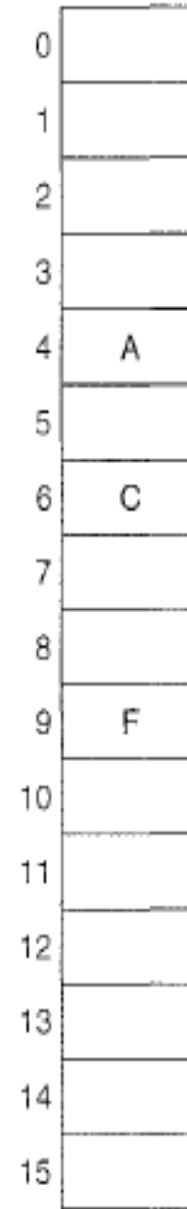
**Page Fault:** - When a process tries to access a page that is not in main memory then a **Page Fault Occurs.**



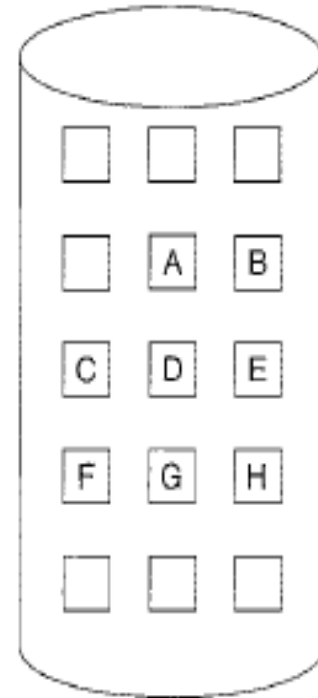
logical memory



page table

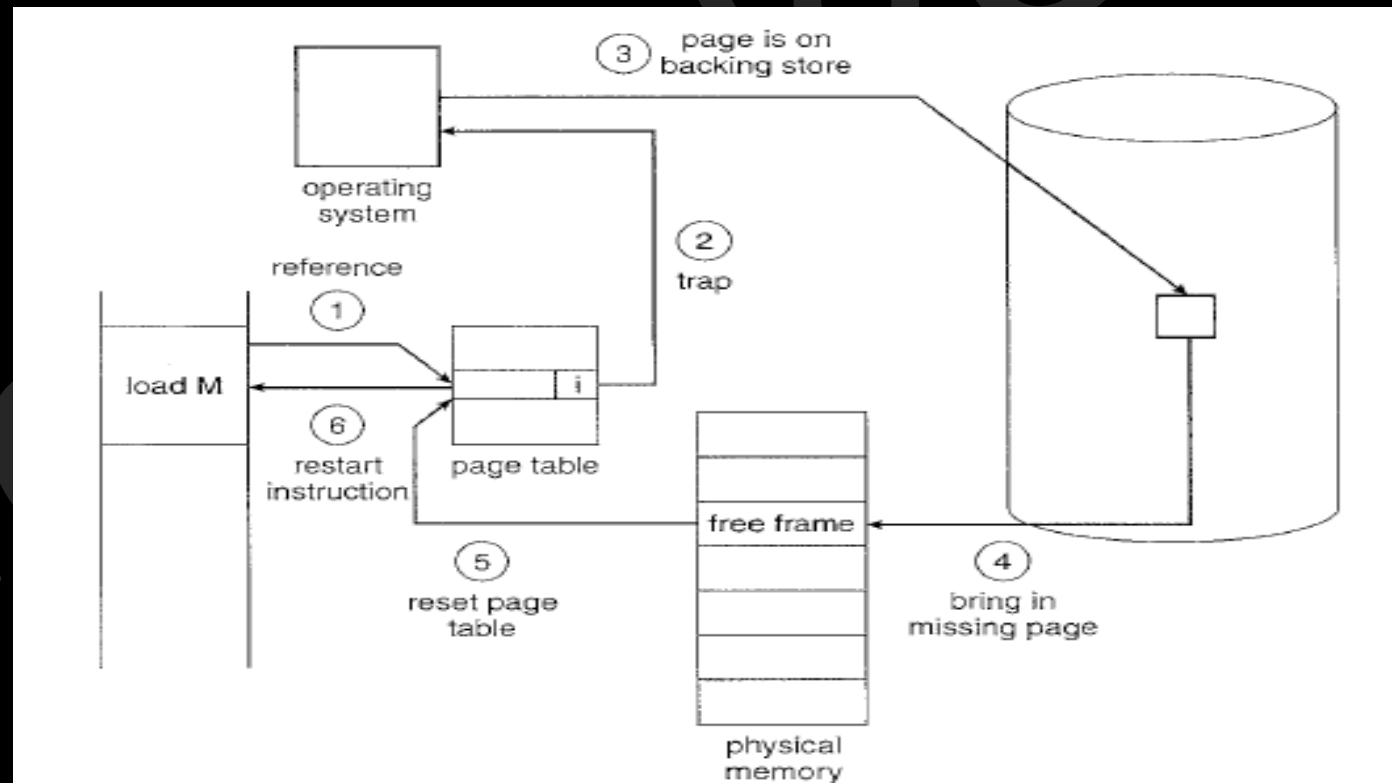


physical memory



## Steps to handle Page Fault

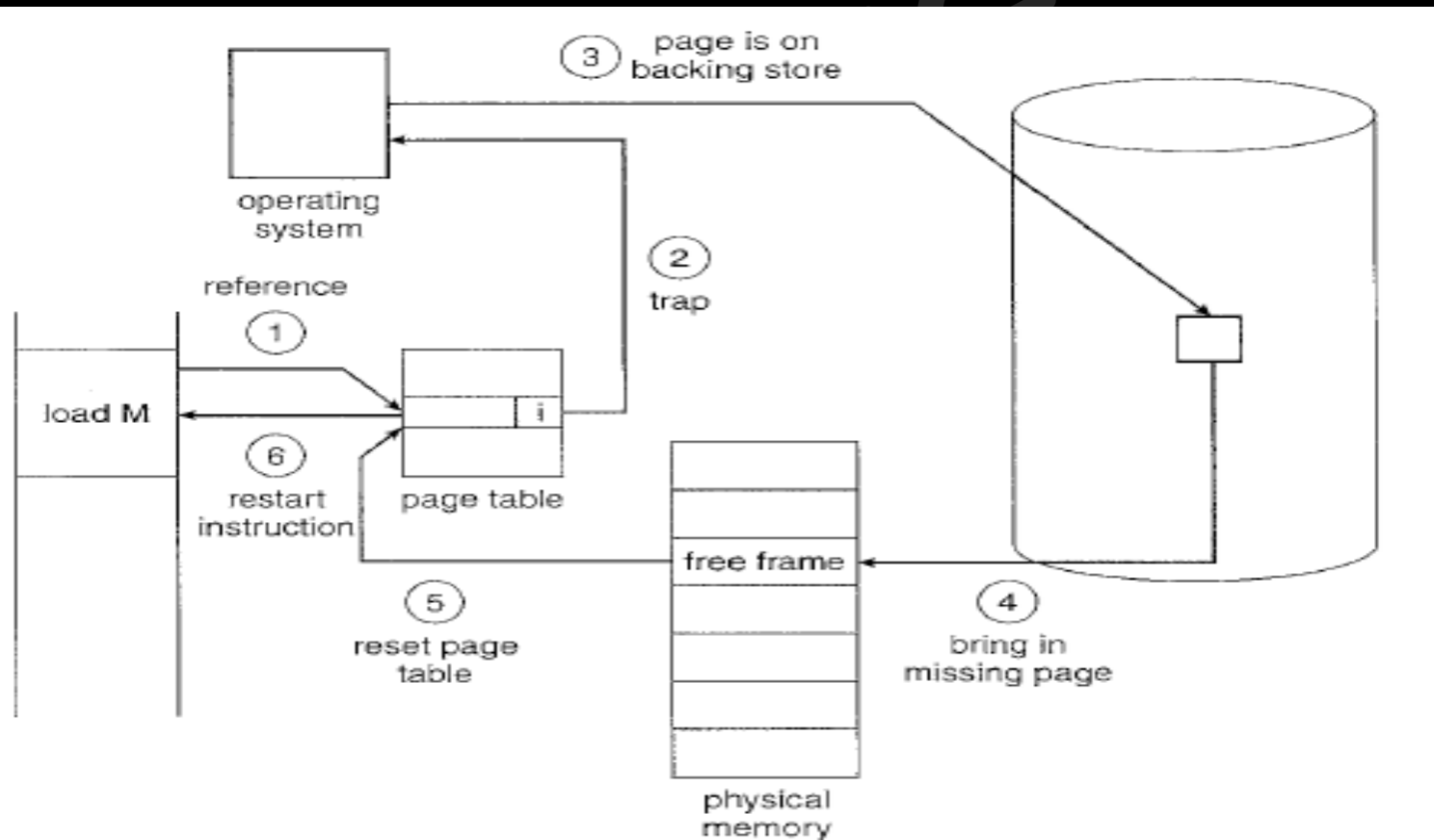
1. If the reference was invalid, it means there is a page fault and page is not currently in main-memory, now we have to load this required page in main-memory.
2. We find a free frame if available we can brought in desired page, but if not we have to select a page as a victim and swap it out from main memory to secondary memory and then swap in the desired page (situation effectively doubles the page-fault service time).



We can reduce this overhead by using a **Modify bit** or **Dirty Bit** as a new column in page table.

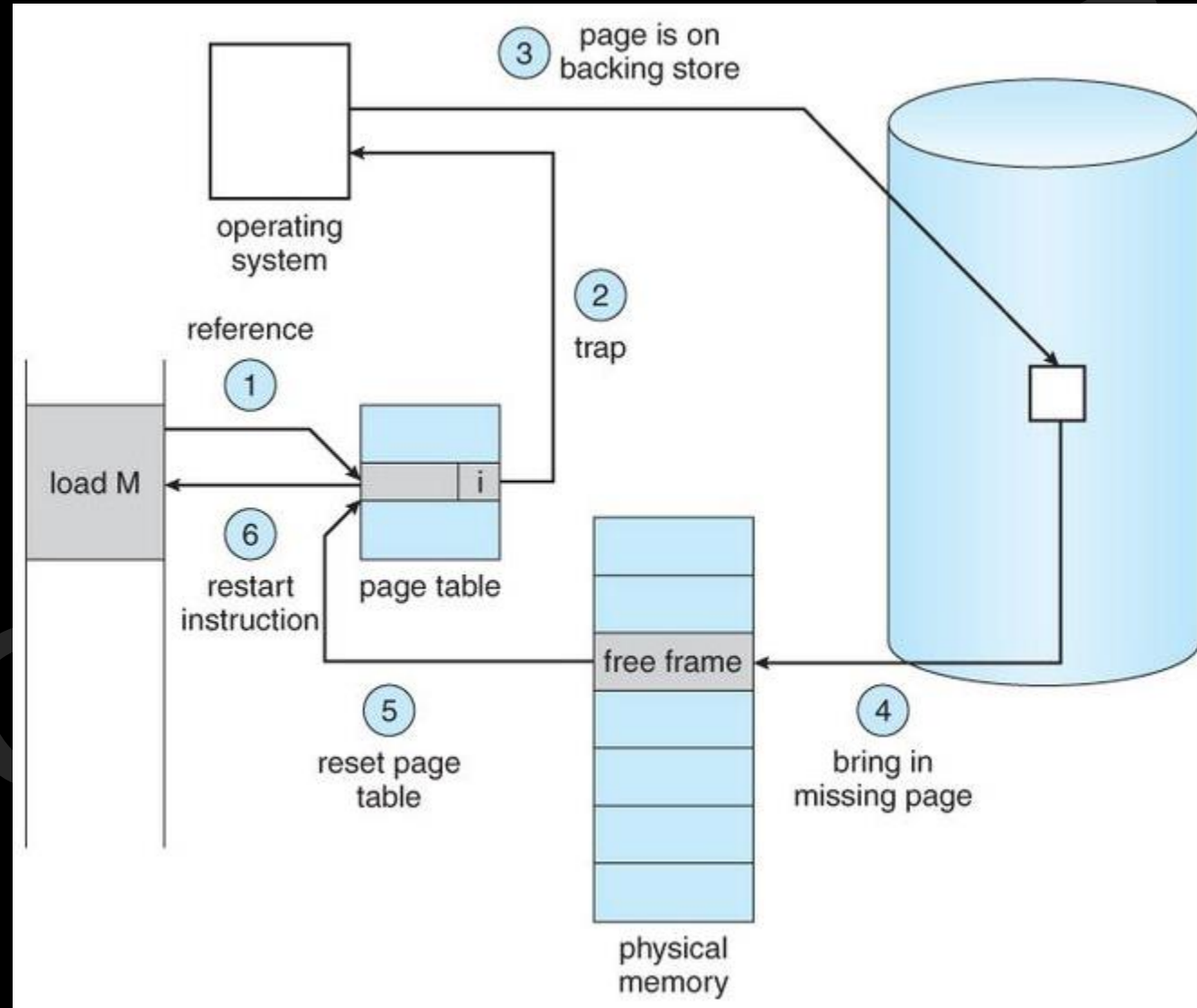
3.1. The modify bit for a page is set whenever the page has been modified. In this case, we must write the page to the disk.

3.2. **If the modify bit is not set:** It means the page has not been modified since it was read into the main memory. We need not write the memory page to the disk: it is already there.





4. Now we modify the internal table kept with the process(PCB) and the page table to indicate that the page is now in memory. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



## Performance of Demand Paging

**Effective Access time for Demand Paging:**

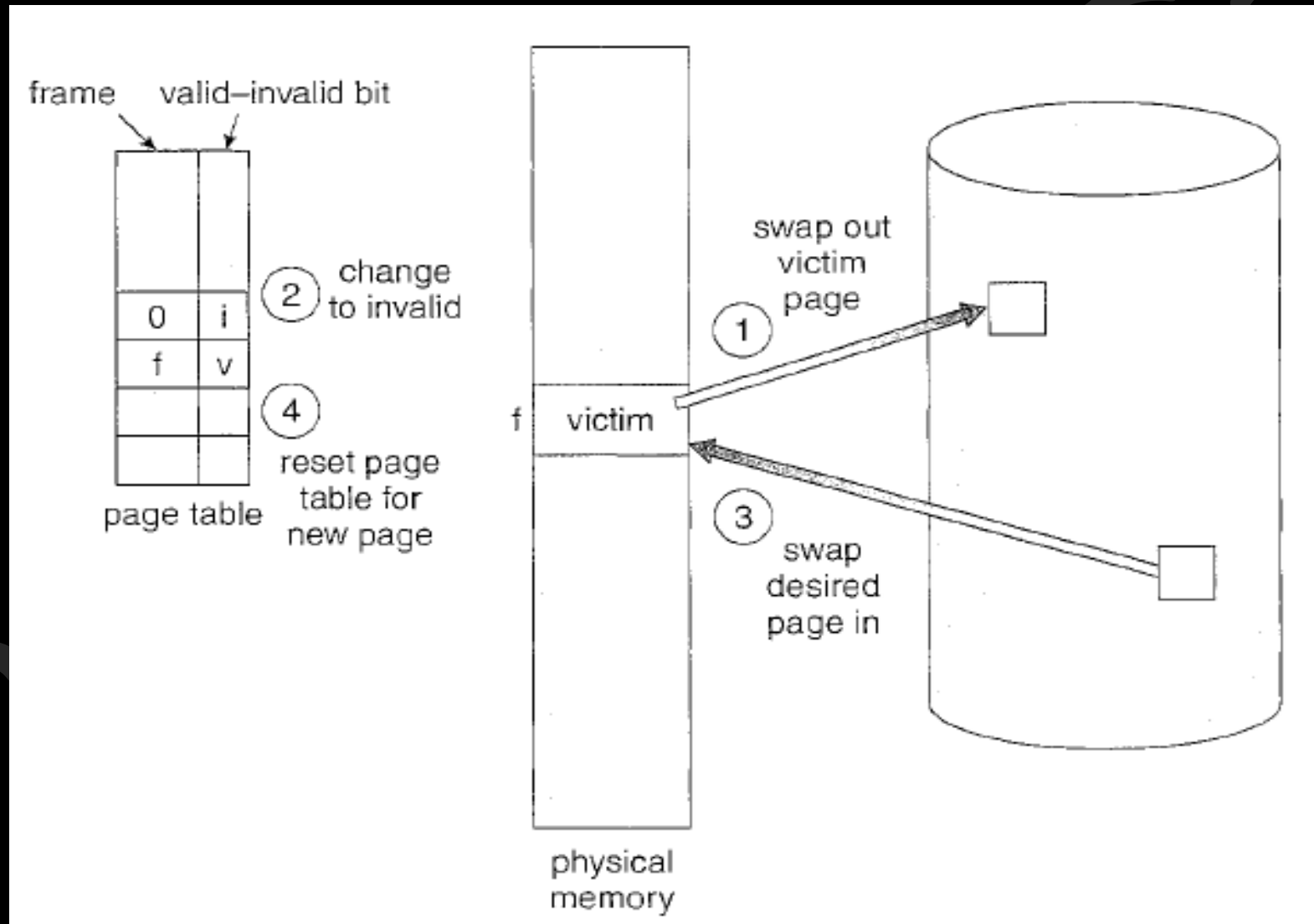
$(1 - p) \times ma + p \times \text{page fault service time.}$

Here, p: Page fault rate or probability of a page fault.

ma is memory access time.

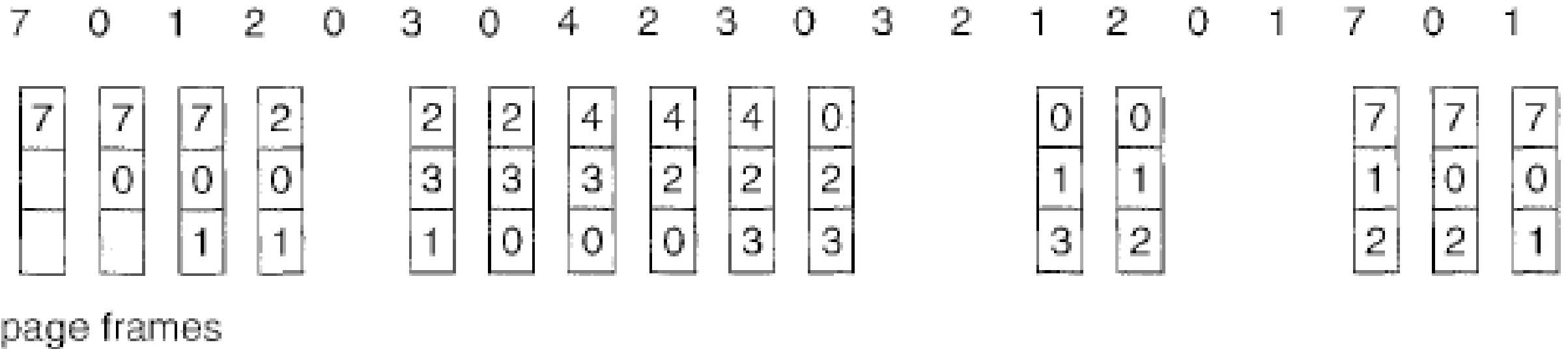
# Page Replacement Algorithms

Now, we must solve a major problems to implement demand paging i.e. a **page replacement algorithm**. 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.

Knowledge Gate

## 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.

Guarantees the lowest possible page fault rate for a fixed number of frames and will never suffer from Belady's anomaly.

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

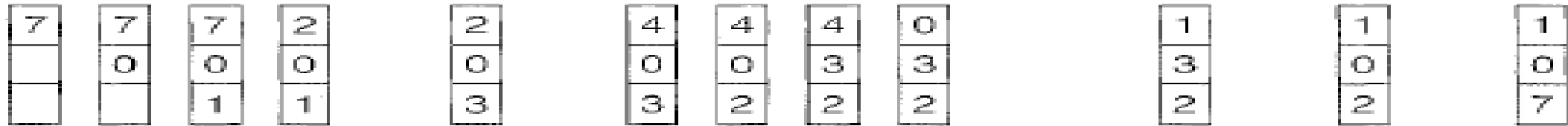
## Least Recently Used (LRU) Page Replacement Algorithm

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

LRU is much better than FIFO replacement in term of page-fault. The LRU policy is often used in industry. LRU also does not suffer from Belady's Anomaly.

reference string

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

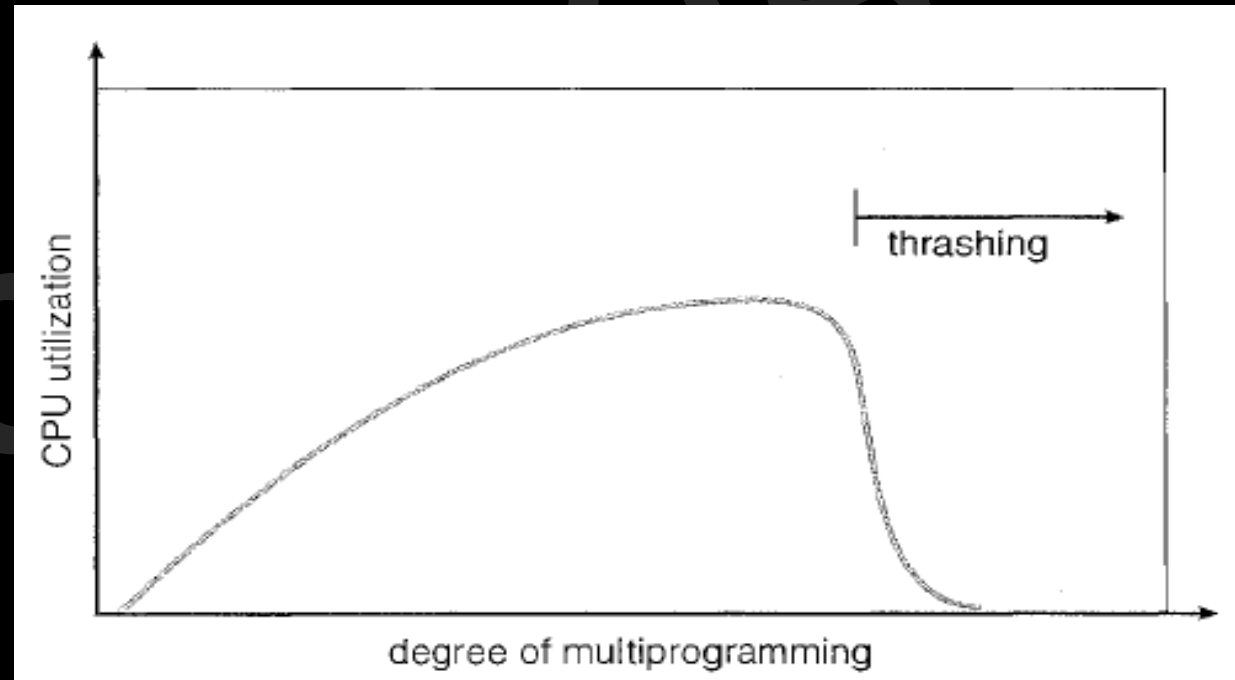


page frames



# Thrashing

1. When a process spends more time swapping pages than executing, it's called Thrashing. Low CPU utilization prompts adding more processes to increase multiprogramming.
2. If a process needs more frames, it starts taking them from others, causing those processes to also fault and swap pages. This empties the ready queue and lowers CPU utilization.
3. Responding to decreased CPU activity, the scheduler adds more processes, worsening the issue. This leads to Thrashing: a cycle of increasing page faults, plummeting system throughput, and rising memory-access times, ultimately resulting in no productive work.

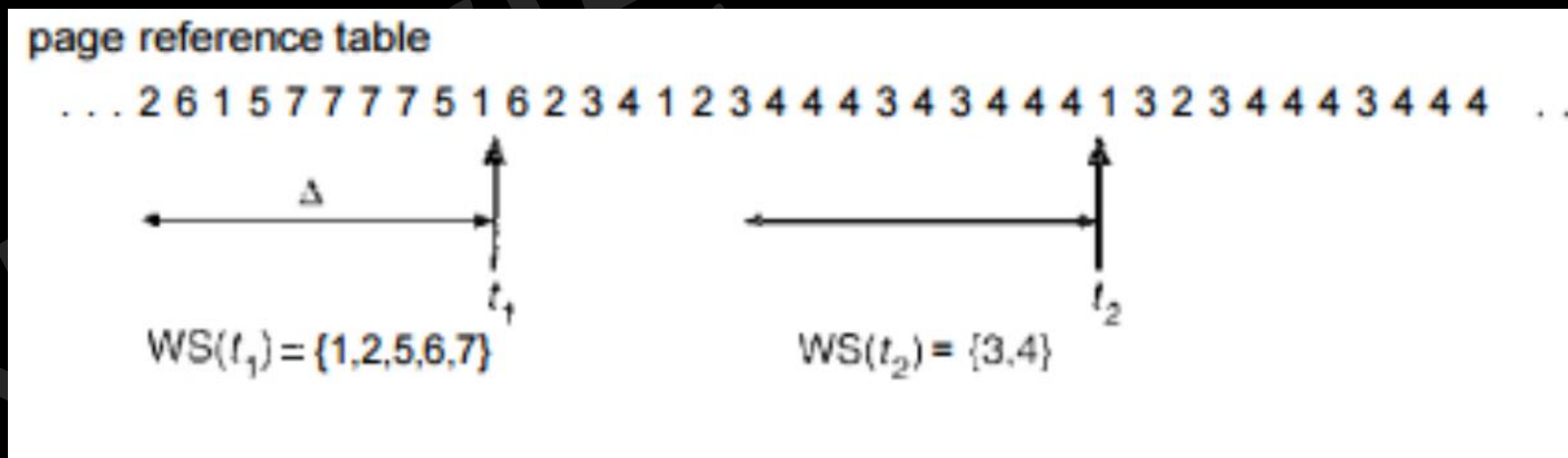


## Solution-The Working Set Strategy

This model uses a parameter  $\Delta$ , to define the **working set window**. The set of pages in the most recent  $\Delta$  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.

The working set is an approximation of the program's locality. The accuracy of the working set depends on the selection of  $\Delta$ . If  $\Delta$  is too small, it will not encompass the entire locality; if  $\Delta$  is too large, it may overlap several localities.

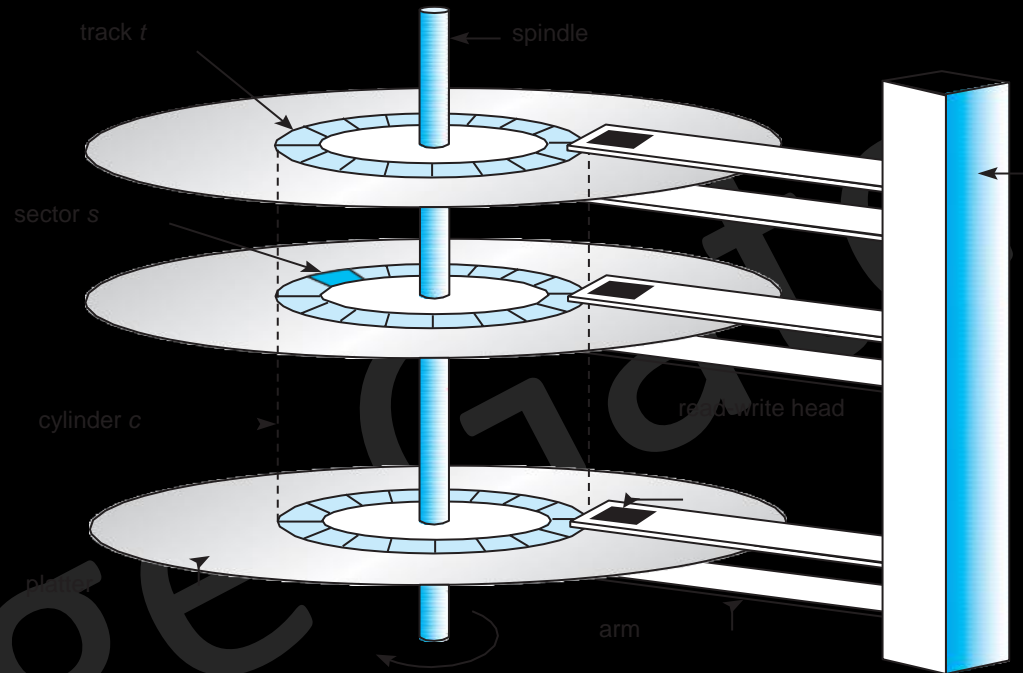


# Disk

Magnetic disks serve as the main secondary storage in computers. Each disk has a flat, circular platter with magnetic surfaces for data storage.

A read-write head hovers over these surfaces, moving in unison on a disk arm. Platters have tracks divided into sectors for logical data storage.

Disks spin at speeds ranging from 60 to 250 rotations per second, commonly noted in RPM like 5,400 or 15,000.



**Total Transfer Time = Seek Time + Rotational Latency + Transfer Time**

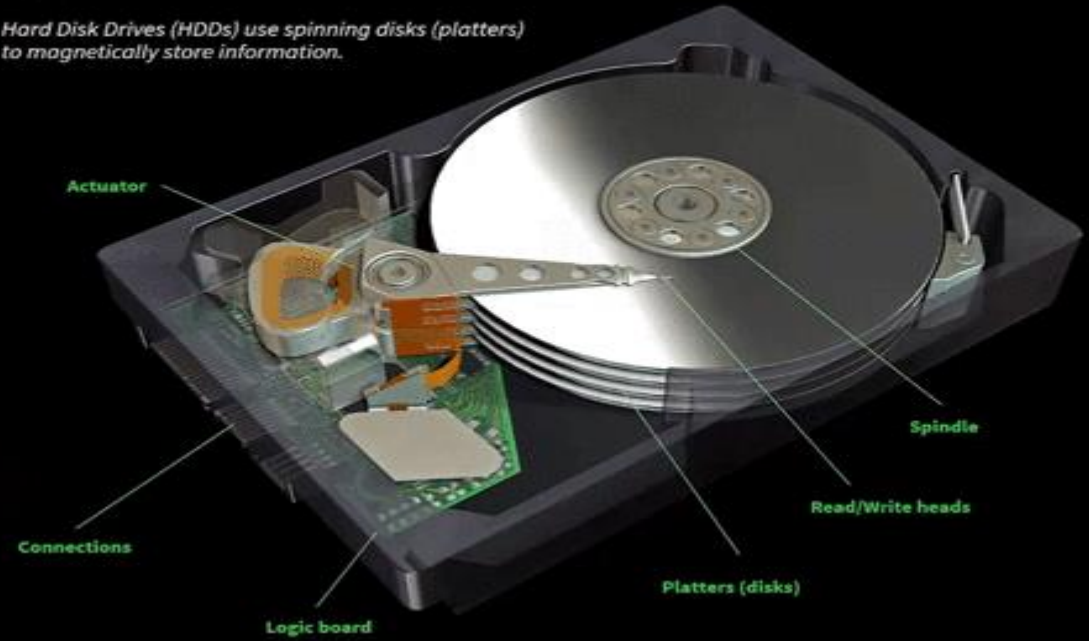
**Seek Time:** - It is a time taken by Read/Write header to reach the correct track. (Always given in question)

**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.

### How Hard Disk Drives Work

*Hard Disk Drives (HDDs) use spinning disks (platters) to magnetically store information.*



# Disk scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, efficiency means having less seek time, less waiting time and high data transfer rate. We can improve all of these by managing the order in which disk I/O requests are serviced.

2. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request may specify several pieces of information: Whether this operation is input or output, disk address, Memory address, number of sectors to be transferred.

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

When one 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 (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.

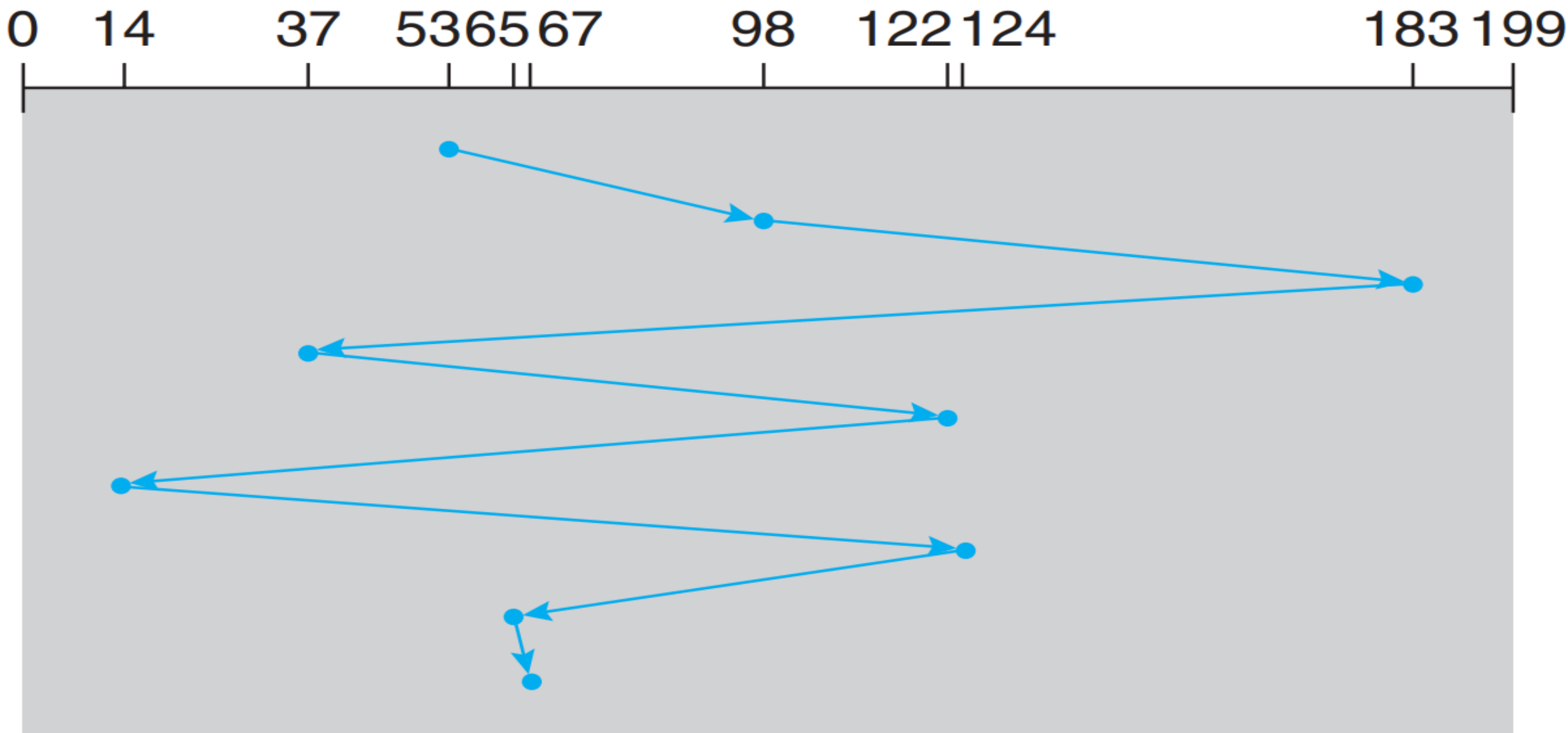
### **Advantages:**

- 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, or waiting time.

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53





## SSTF(Shortest Seek Time First) Scheduling

Major component in total transfer time is seek time, in order to reduce seek time if we service all the requests close to the current head position, this idea is the basis for the SSTF algorithm. In SSTF, 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.

- **Advantages:**

Seek movements 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

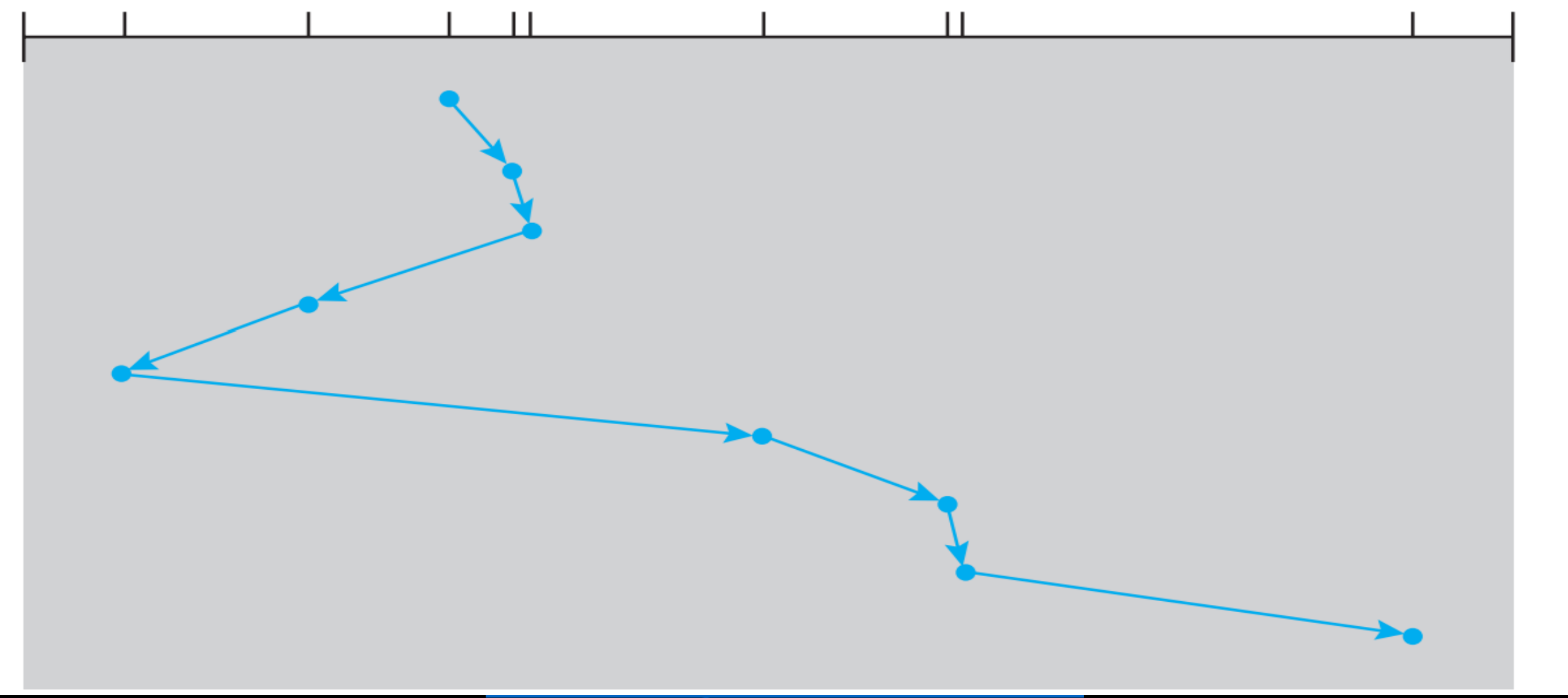
High variance of response time and waiting time as SSTF favors only closest requests



queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

0 14 37 53 65 67 98 122 124 183 199



## SCAN/ Elevator Algorithm

- 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.

### **Advantages:**

Simple easy to understand and use

No starvation but more wait for some random process

Low variance and Average response time

### **Disadvantages:**

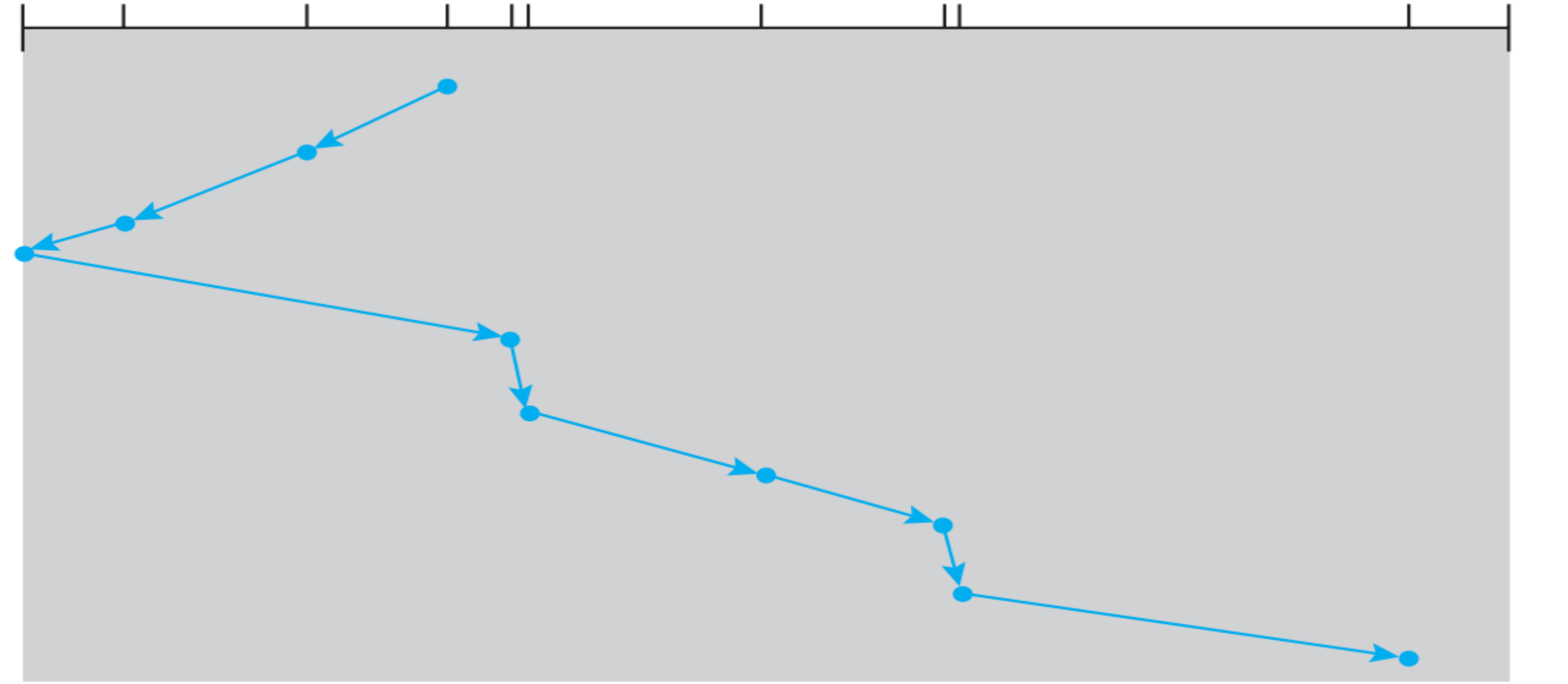
Long waiting time for requests for locations just visited by disk arm.

Unnecessary move to the end of the disk, even if there is no request.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

0 14 37 53 65 67 98 122 124 183 199



## C-SCAN Scheduling

- When the disk head reaches one end and changes direction, fewer requests are nearby since those cylinders were just serviced. Most pending requests are at the opposite end, having waited the longest.
- 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:**

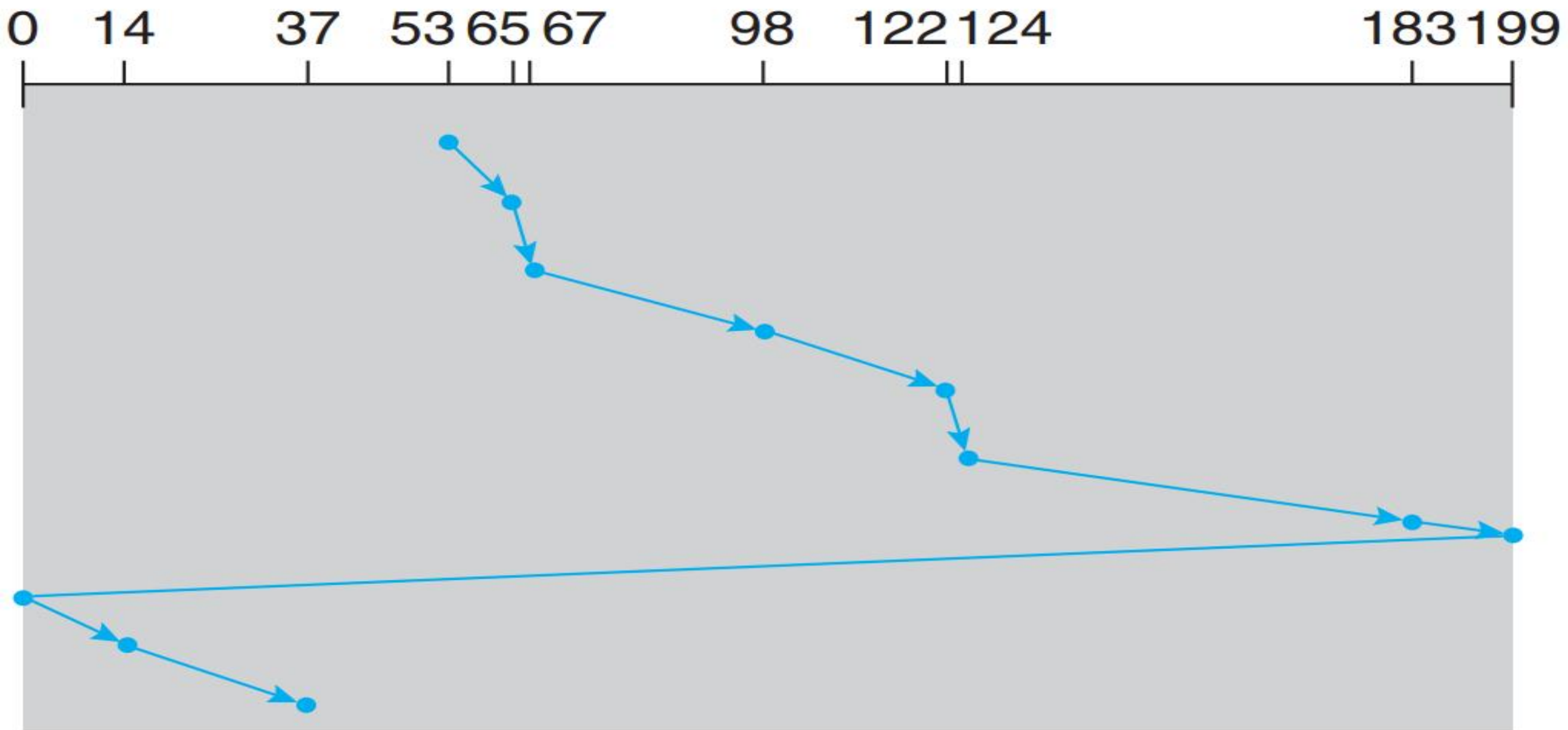
Provides more uniform wait time compared to SCAN  
Better response time compared to scan

### **Disadvantage:**

More seeks movements in order to reach starting position

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



## 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: -**

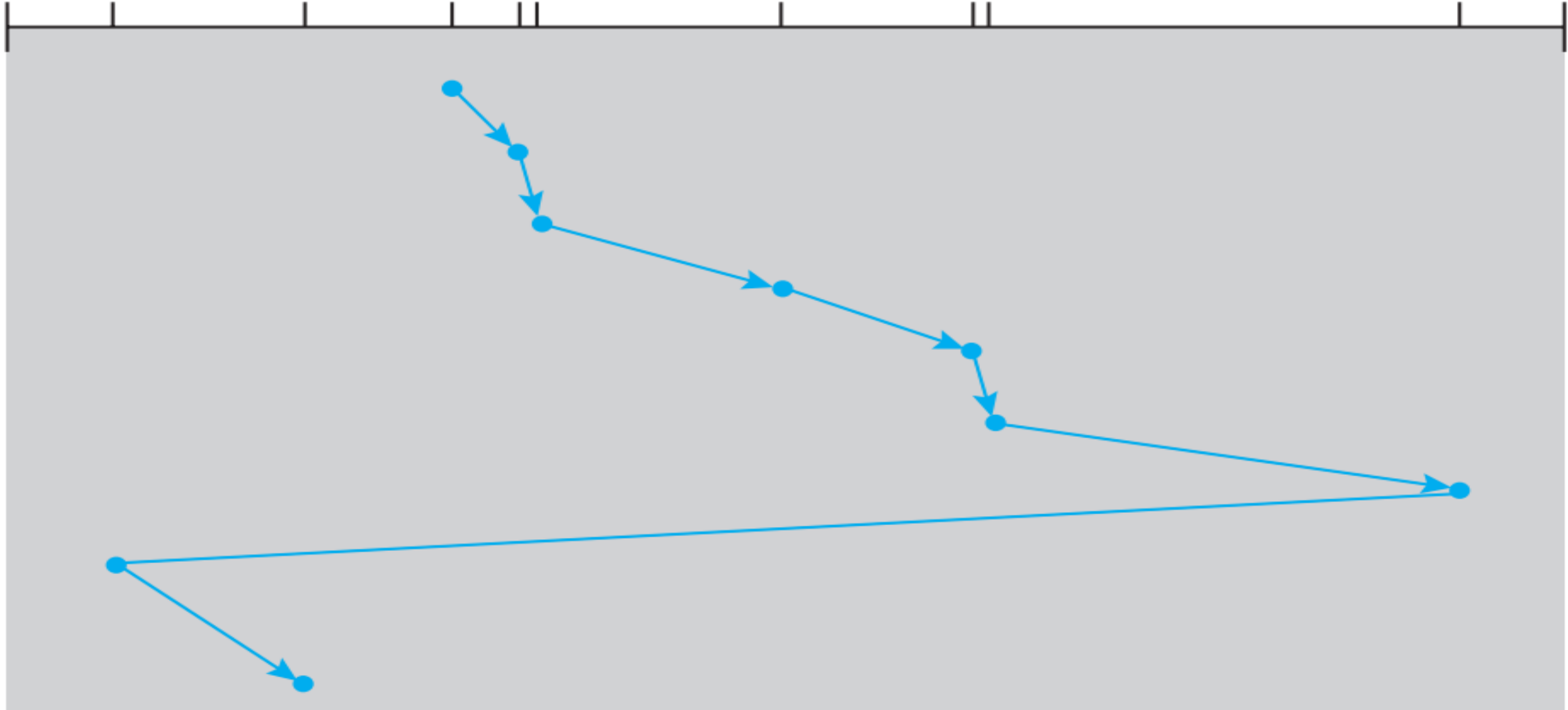
Overhead to find the last request and go to initial position is more

Should not be used in case of more load.

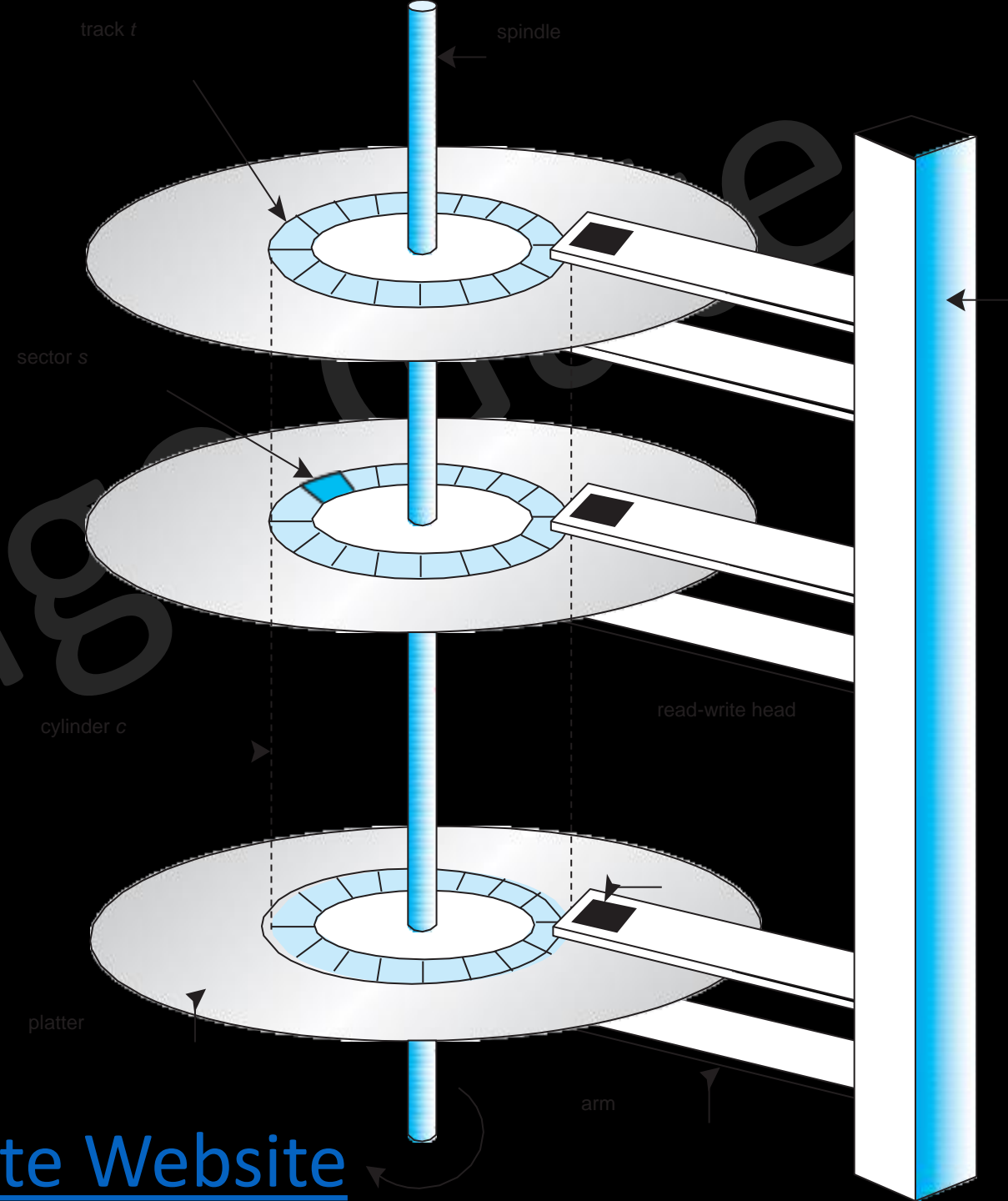
queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

0 14 37 53 65 67 98 122 124 183 199







# 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 =  $(\text{File Size}/\text{Track Size}) * \text{time taken to complete one revolution}$ .

Q Consider a disk where there are 512 tracks, each track is capable of holding 128 sector 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.

Knowledge Gate

[Knowledge Gate Website](#)

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?

Knowledge Gate

**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

Knowledge Gate

## 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**

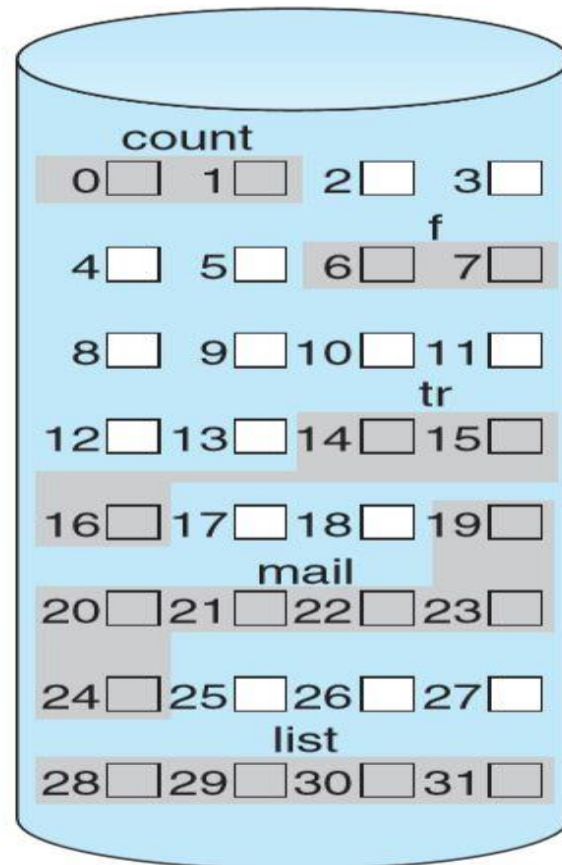
**Indexed**

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.

In directory usually we store three column file name, start dba and length of file in number of blocks.



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**

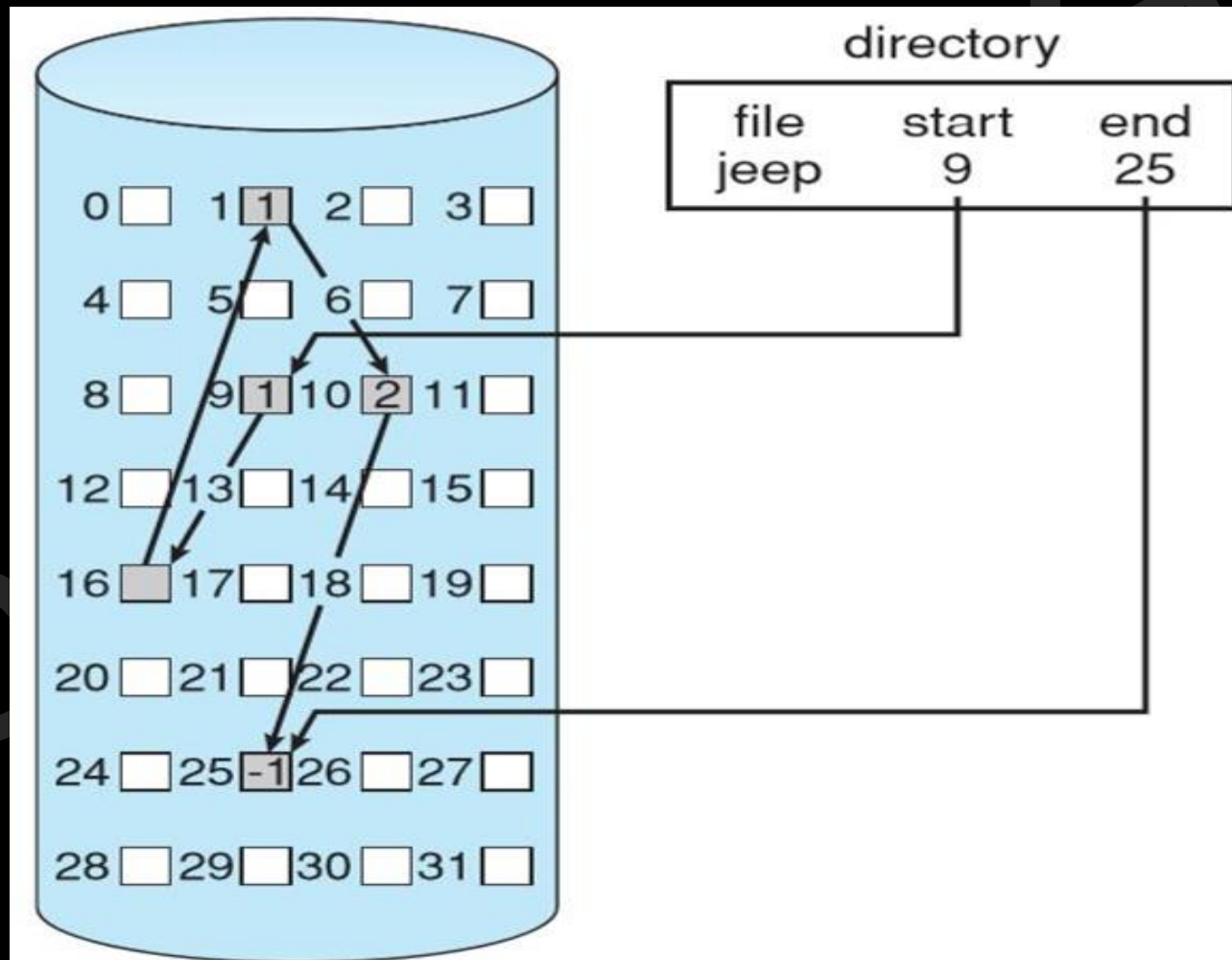
Suffer from huge amount of external fragmentation.

Another problem with contiguous allocation is file modification



# Linked 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, read, write a new file is simply easy. 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.

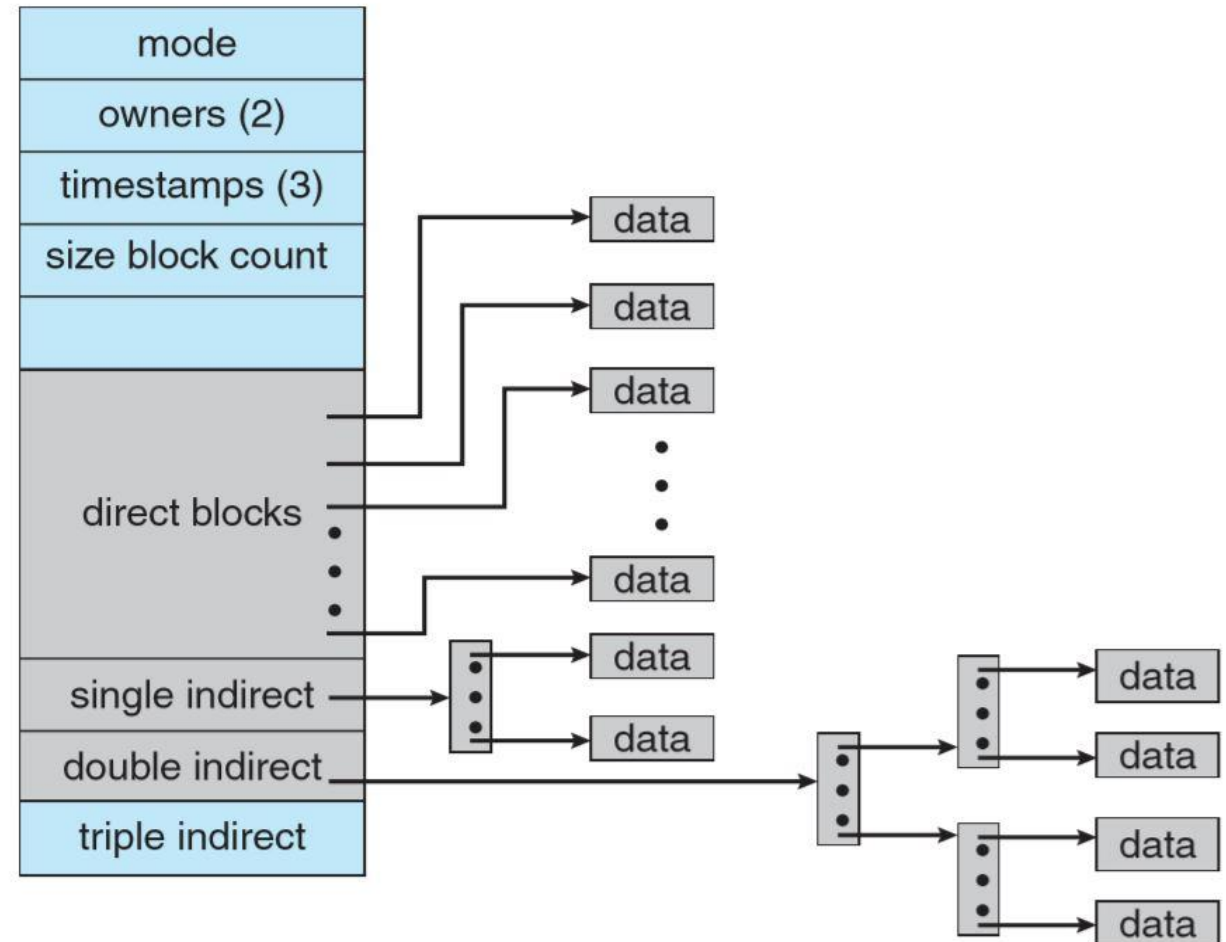
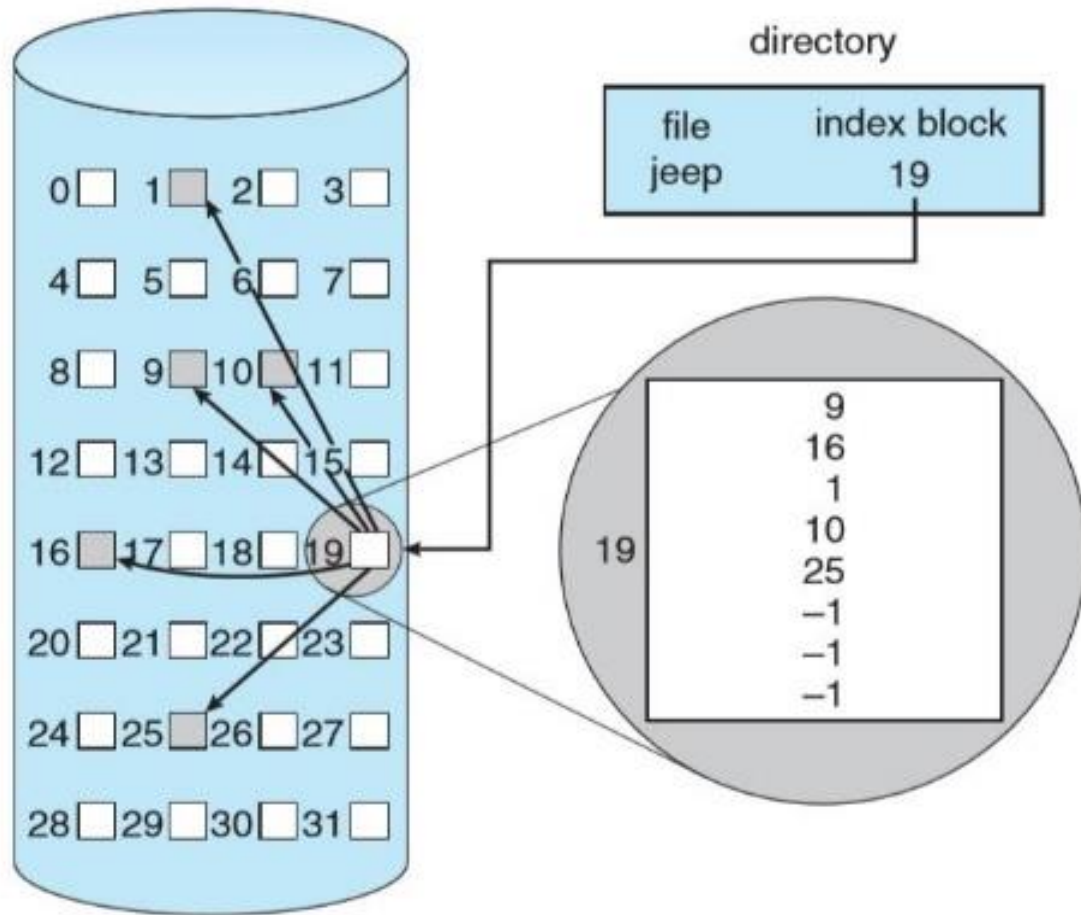
**Disadvantage: -**

Only sequential access is possible, To find the  $i^{\text{th}}$  block of a file, we must start at the beginning and follow the pointers until we get to the  $i^{\text{th}}$  block.

Another disadvantage is the space required for the pointers, so each file requires slightly more space than it would otherwise.

# 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 an index block containing an array of disk-block addresses. The directory entry points to this index block.

When a file is created, all index block pointers are null. Writing to the  $i^{\text{th}}$  block updates the corresponding index-block entry with a block address from the free-space manager.

The size of the index block is a trade-off: it should be small to save space but large enough to accommodate pointers for big files.

**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).

Knowledge Gate

[Knowledge Gate Website](#)

**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.

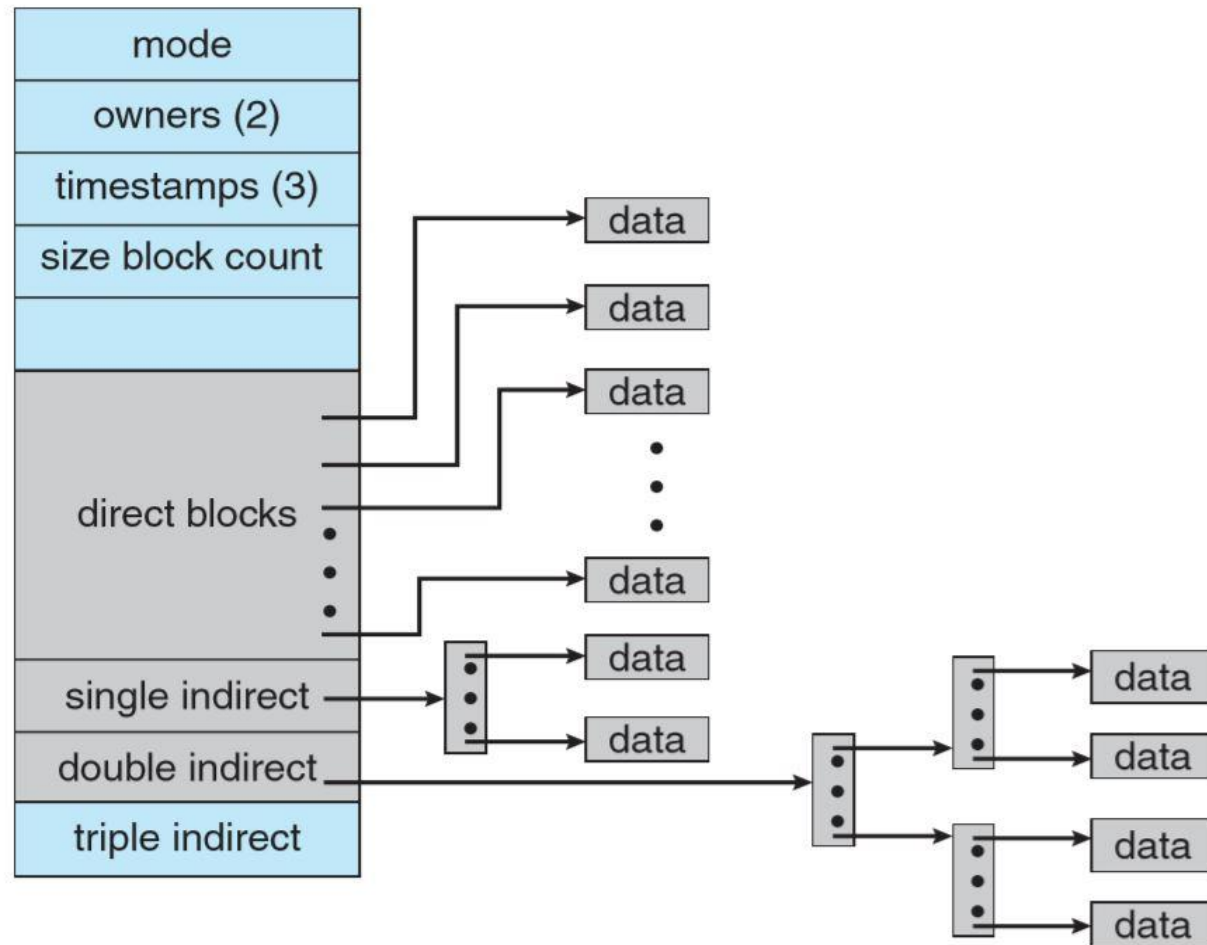
To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

Knowledge Gate

[Knowledge Gate Website](http://www.knowledgegate.com)

**Combined scheme:** In UNIX-based systems, the file's Inode stores the first 15 pointers from the index block. The first 12 point directly to data blocks, eliminating the need for a separate index block for small files.

The next three pointers are for indirect blocks: the first for a single indirect block, the second for a double indirect block, and the last for a triple indirect block, each increasingly indirecting to the actual data blocks.



### 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. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Knowledge Gate



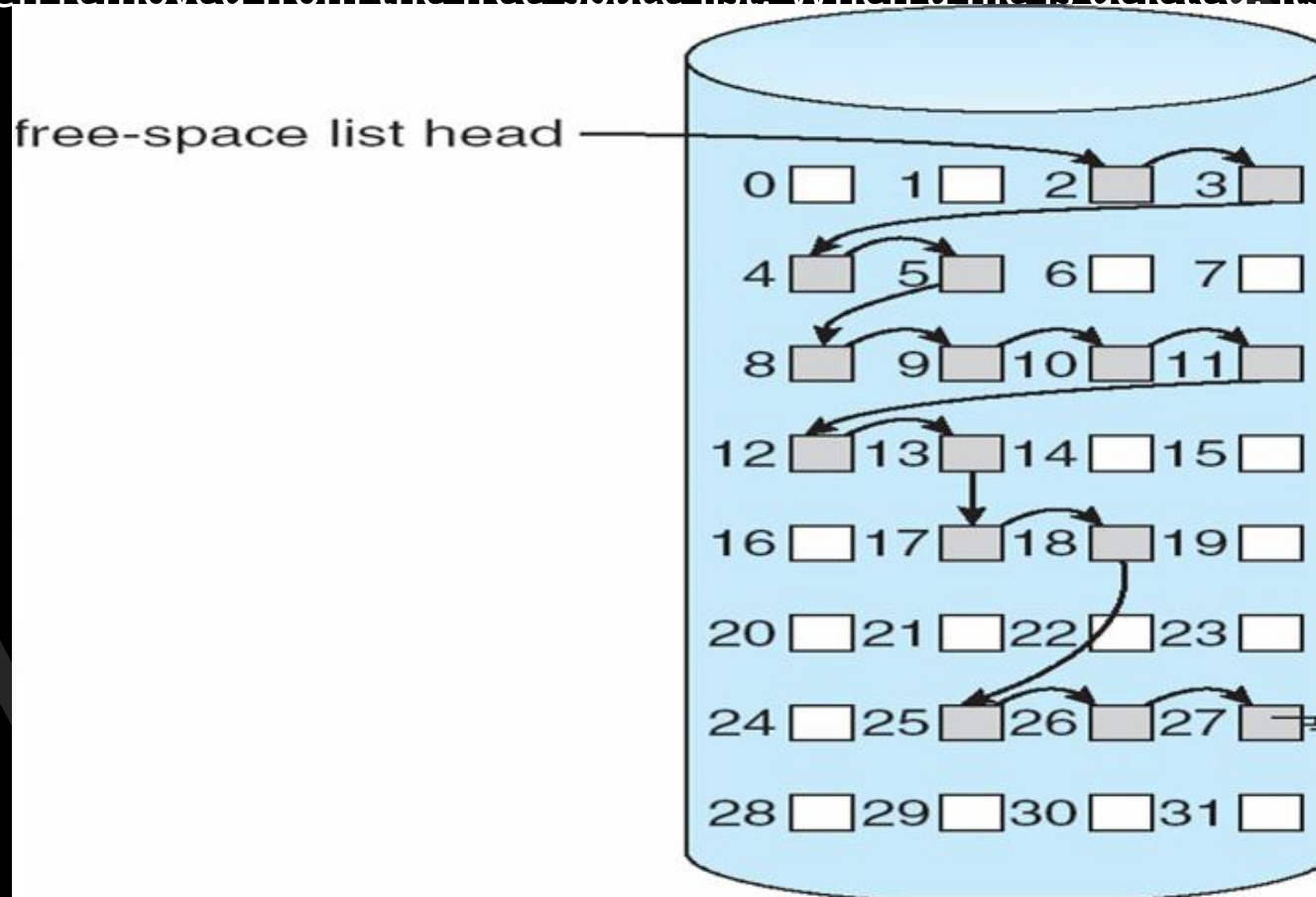
# Break

[Knowledge Gate Website](#)

# Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. 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.



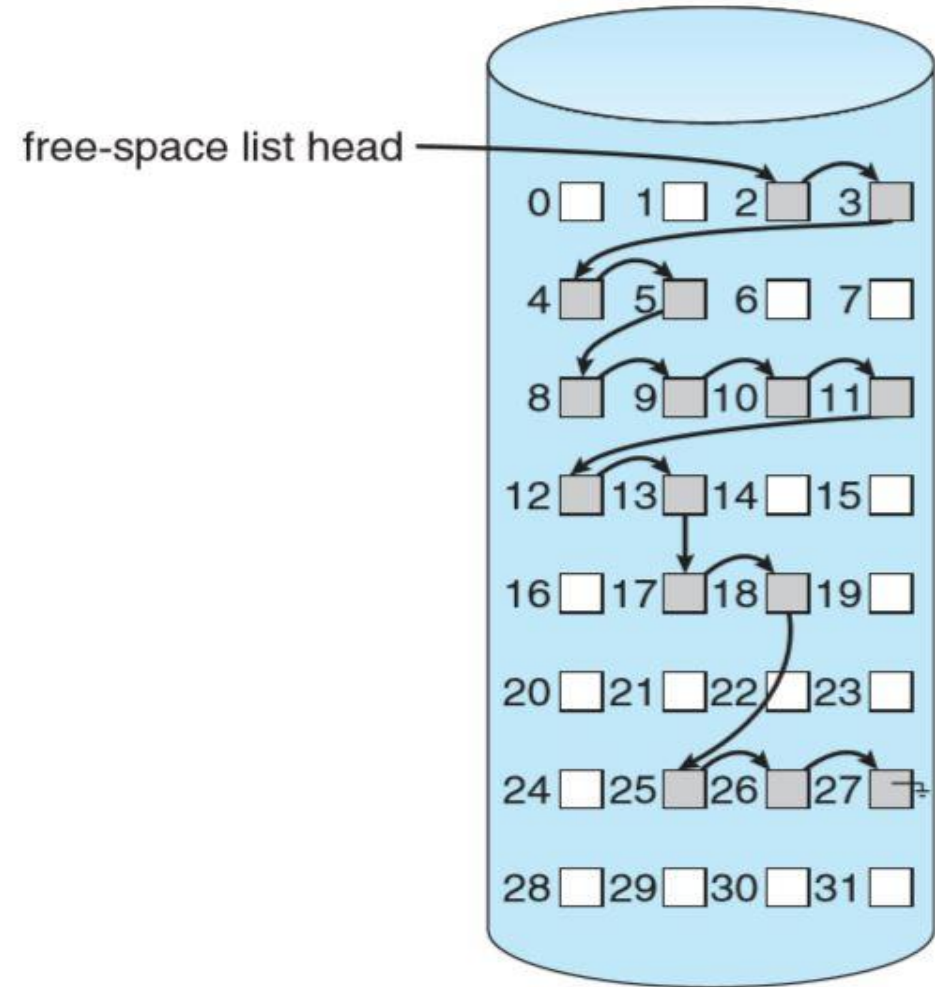
KN

## Linked List

A 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.

This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time. However, 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.

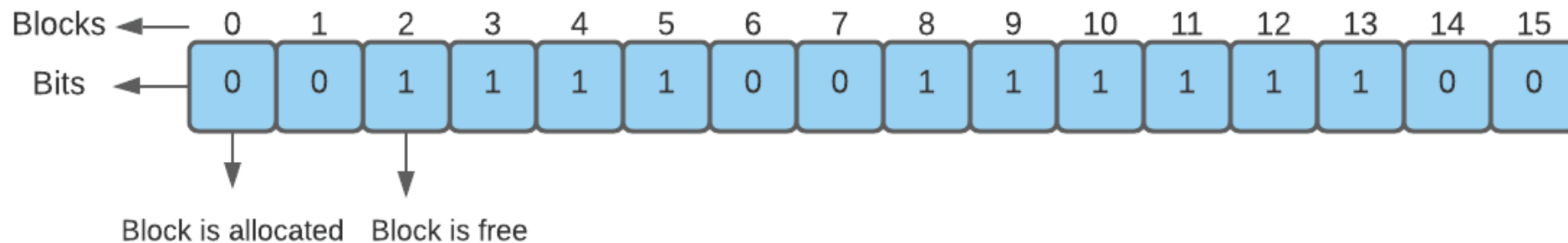


# Bit Vector

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 ...

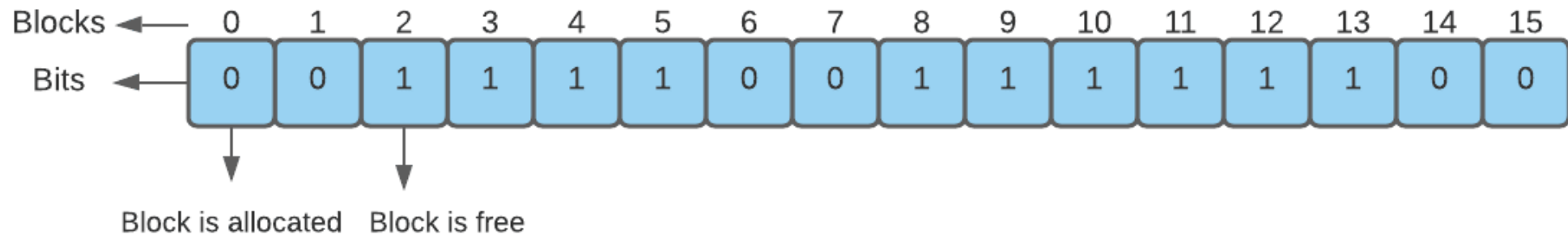
The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.



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.



# Break

[Knowledge Gate Website](#)

## File organization

- File organization refers to the way data is stored in a file. File organization is very important because it determines the methods of access, efficiency, flexibility and storage devices to use.
- Four methods of organizing files:
  - **1. Sequential file organization:**
    - a. Records are stored and accessed in a particular sorted order using a key field.
    - b. Retrieval requires searching sequentially through the entire file record by record to the end.
  - **2. Random or direct file organization:**
    - a. Records are stored randomly but accessed directly.
    - b. To access a file which is stored randomly, a record key is used to determine where a record is stored on the storage media.
    - c. Magnetic and optical disks allow data to be stored and accessed randomly.

- **3. Serial file organization:**
  - a. Records in a file are stored and accessed one after another.
  - b. This type of organization is mainly used on magnetic tapes.
- **4. Indexed-sequential file organization method:**
  - Almost similar to sequential method only that, an index is used to enable the computer to locate individual records on the storage media
  - For example, on a magnetic drum, records are stored sequentially on the tracks. However, each record is assigned an index that can be used to access it directly.

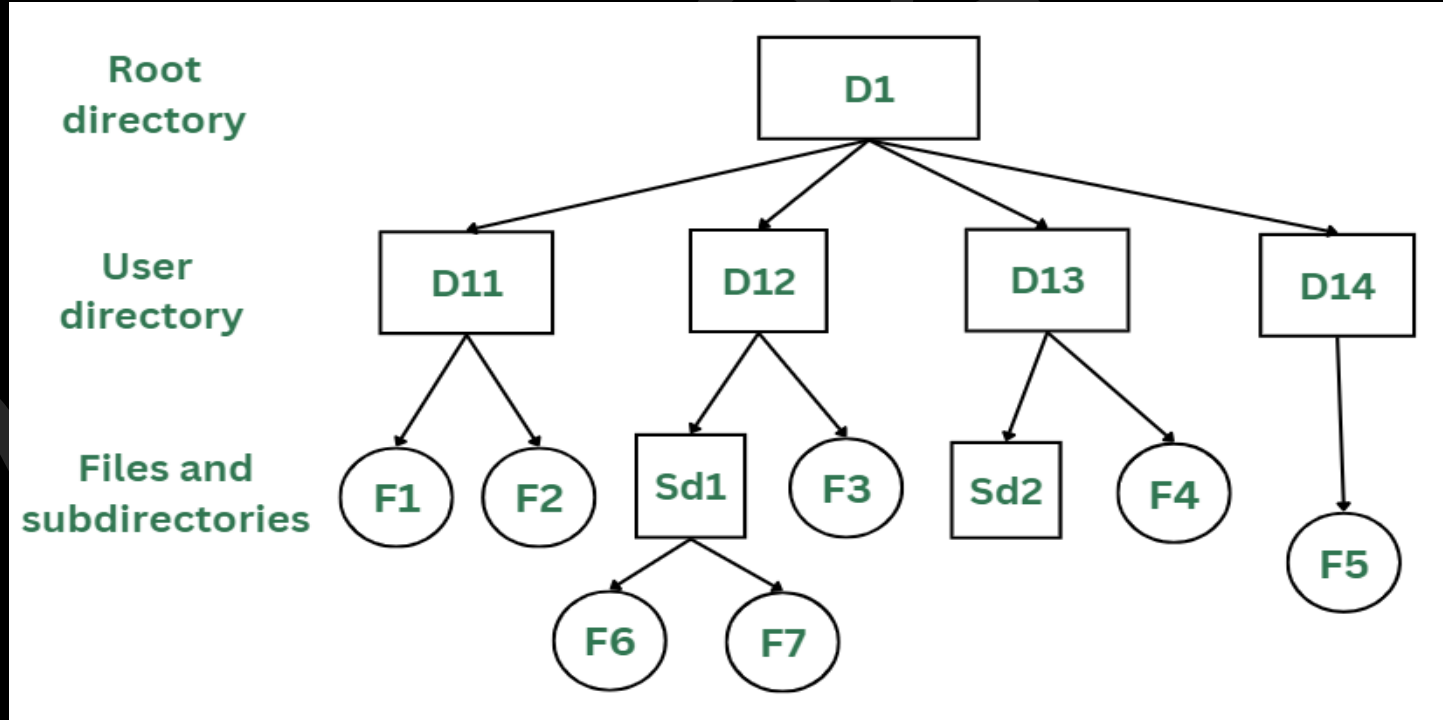


## File access mechanism

- Sequential access:
  - it is the simplest access mechanism in which information is stored in a file are exist in an order such that one record is process after the other
  - for example editors and compilers usually access file in this manner. next line
- Direct Access:
  - It is an alternative method for accessing a file, which is based on the disk model of a file, since disk allow random access to any block or a record of a file
  - for this method, a file is viewed as a numbered sequence of blocks or records which are read/written in an arbitrary manner that is there is no restriction on the order of recording or writing
  - it is well suited for database management system.
- Index access
  - In this method and alternate index is created which contain key field and a pointer to the various blocks.
  - To find and entry in the file for a key value we first search the index and then use the pointer to directly excess of file and find the desired entry

# Directory

- A directory is similar to a "folder" in everyday terminology, and it exists within a file system.
- It's a virtual container where multiple files and other directories (often called subdirectories) can reside.
- It organizes the file system in a hierarchical manner, meaning directories can contain subdirectories, which may contain further subdirectories, and so on.



## Operations that Can Be Performed on a Directory

- 1. Create Directory:** Make a new directory to store files and subdirectories.
- 2. Delete Directory:** Remove an existing directory, usually only if it's empty.
- 3. Rename Directory:** Change the name of a directory.
- 4. List Contents:** View the files and subdirectories within a directory.
- 5. Move Directory:** Relocate a directory to a different path in the file system.
- 6. Copy Directory:** Make a duplicate of a directory, including its files and subdirectories.
- 7. Change Directory:** Switch the working directory to a different one.
- 8. Search Directory:** Find specific files or subdirectories based on certain criteria like name or file type.
- 9. Sort Files:** Arrange the files in a directory by name, date, size, or other attributes.
- 10. Set Permissions:** Change the access controls for a directory (read, write, execute).

Feature	Single-Level Directory	Two-Level Directory
<b>User Isolation</b>	No user-specific directories. All users share the same directory space.	Each user has their own private directory.
<b>Organization</b>	All files are stored in one directory, making it less organized.	Files can be organized under user-specific directories, allowing for better file management.
<b>Search Efficiency</b>	Can be less efficient as all files are in a single directory, requiring more time to find a specific file.	More efficient due to fewer files in each user-specific directory.
<b>Access Control</b>	Hard to implement user-specific access controls because all files reside in the same directory.	Easier to implement user-specific access controls, enhancing security.
<b>Complexity</b>	Simpler to implement but can become cluttered and difficult to manage with many files.	Slightly more complex due to the need for user management, but offers better organization.

## Why It's Necessary

- **Organization:** It helps in sorting and locating files more efficiently.
- **User-Friendliness:** Directories make it easier for users to categorize their files by project, file type, or other attributes.
- **Access Control:** Using directories, different levels of access permission can be applied, providing an extra layer of security.

## Features of Directories

- **Metadata:** Directories also store metadata about the files and subdirectories they contain, such as permissions, ownership, and timestamps.
- **Dynamic Nature:** As files are added or removed, the directory dynamically updates its list of contents.
- **Links and Shortcuts:** Some systems support the creation of pointers or links within directories to other files or directories.

[Knowledge Gate Website](#)

Feature	Sequential File	Indexed File
Access Method	Records accessed one after another in order	Records can be accessed directly using an index
Speed of Access	Slower, especially for large files	Faster for random access, thanks to index
Storage Efficiency	Generally more efficient as no space is used for index	Less efficient due to storage needed for index
Update Complexity	Simpler, usually involves appending	More complex; updating index needed for record change
Use Case	Suitable for batch processing, backups <a href="https://www.knowledgegate.com">Knowledge Gate Website</a>	Suitable for databases, directories with quick lookup

# File Protection System

- **Reliability:**
  - Reliability in a file protection system ensures that files are accessible and retrievable whenever needed, without loss of data. Techniques such as backup, mirroring, and RAID configurations contribute to high reliability.
- **Security:**
  - Security mechanisms protect files from unauthorized access, modification, or deletion. Encryption, firewall settings, and secure file transfer protocols like SFTP can be employed to safeguard files.
- **Controlled Access:**
  - Controlled access specifies who can do what with a file. Users are given permissions like read, write, and execute (r-w-x), often categorized into roles for easy management. Controlled access is crucial for maintaining the integrity and confidentiality of files.
- **Access Control:**
  - Access control mechanisms like Access Control Lists (ACLs) or Role-Based Access Control (RBAC) define rules specifying which users or system processes are granted access to files and directories. They can also specify the type of operations (read, write, execute) permitted.
- In summary, a robust file protection system is multi-layered, incorporating reliability measures, strong security protocols, and detailed access control mechanisms to ensure the safe and efficient management of files.

## Access Matrix

- The Access Matrix is a conceptual framework used in computer security to describe the permissions that different subjects (such as users or processes) have when accessing different objects (such as files, directories, or resources).
- In this matrix, each row represents a subject and each column represents an object. The entry at the intersection of a row and column defines the type of access that the subject has to the object.



- **Matrix Form**: In its most straightforward representation, the Access Matrix is a table where the cell at the intersection of row  $i$  and column  $j$  contains the set of operations that subject  $i$  can perform on object  $j$ .

	File A	File B	File C
User 1	r-w	r	-
User 2	r	w	r-w
User 3	-	r	w

- Here, 'r' indicates read permission, 'w' indicates write permission, and '-' indicates no permission.

- Access Control Lists (ACLs): Each object's column in the matrix can be converted to an Access Control List, which lists all subjects and their corresponding permissions for that object.
- File A:
  - User 1: r-w
  - User 2: r
- File B:
  - User 1: r
  - User 2: w
  - User 3: r

- **Capability Lists**: Each subject's row in the matrix can be converted into a Capability List, which lists all objects and the operations the subject can perform on them.
- User 1:
  - File A: r-w
  - File B: r
- User 2:
  - File A: r
  - File B: w
  - File C: r-w

- **Sparse Matrix**: In large systems, the Access Matrix is usually sparse. Special data structures can be used to represent only the non-empty cells to save space.

Knowledge Gate

# Implementation of Access Matrix

1. **Global Table**: A global table is essentially the raw access matrix itself, where each cell denotes the permissions a subject has on an object. While straightforward, this method is not practical for large systems due to the sparsity of the matrix and the associated storage overhead.
2. **Access Lists for Objects**: Here, the focus is on objects like files or directories. Each object maintains an Access Control List (ACL) that records what operations are permissible by which subjects. ACLs are object-centric and make it easy to determine all access rights to a particular object. However, this approach makes it cumbersome to list all capabilities of a particular subject across multiple objects.
3. **Capability Lists for Domains**: In this subject-centric approach, each subject or domain maintains a list of objects along with the operations it can perform on them, known as a Capability List. This makes it straightforward to manage and review the permissions granted to each subject. On the downside, revoking or changing permissions across all subjects for a specific object can be more challenging.
4. **Lock-Key Mechanism**: In a lock-key mechanism, each object is assigned a unique "lock," and subjects are granted "keys" to unlock these locks. When a subject attempts to access an object, the system matches the key with the lock to determine if the operation is permissible. This approach can be seen as an abstraction over the access matrix and can be used to dynamically change permissions with minimal overhead.