The Art of Operating System

The Ultimate Quick Reference Guide

Santosh Kumar Mishra
SDE @Microsoft, Author

Preface

Welcome to "The Art of Operating System: The Ultimate Quick Reference Guide"! In this book, we explore the world of operating systems, providing a comprehensive understanding of their concepts. Whether you're a student,software engineer, or simply curious, this guide offers a clear and accessible resource.

Authored by Santosh Kumar Mishra, a Software Engineer at Microsoft, this book simplifies operating system complexities.

My previous work, "The Art of Data Structures and Algorithms: The Ultimate Quick Reference Guide," is acclaimed for its user-friendly approach. Building upon my expertise,I bring you a comprehensive guide to operating systems, focusing on clarity and ease of understanding.

This book covers the breadth of operating system concepts, incorporating graphics and images for better comprehension. Our goal is to provide a quick reference guide for easy revision, suitable for readers of all levels.

I express my gratitude to Vijaya Lakshmi Vaspula for her unwavering support,invaluable assistance, contributions and insights has been instrumental throughout the book's creation.

Enjoy your journey through "The Art of Operating System: The Ultimate Quick Reference Guide" as you delve into the inner workings of operating systems. Let this book be your companion in navigating this vital field.

Happy reading!
Santosh Kumar Mishra

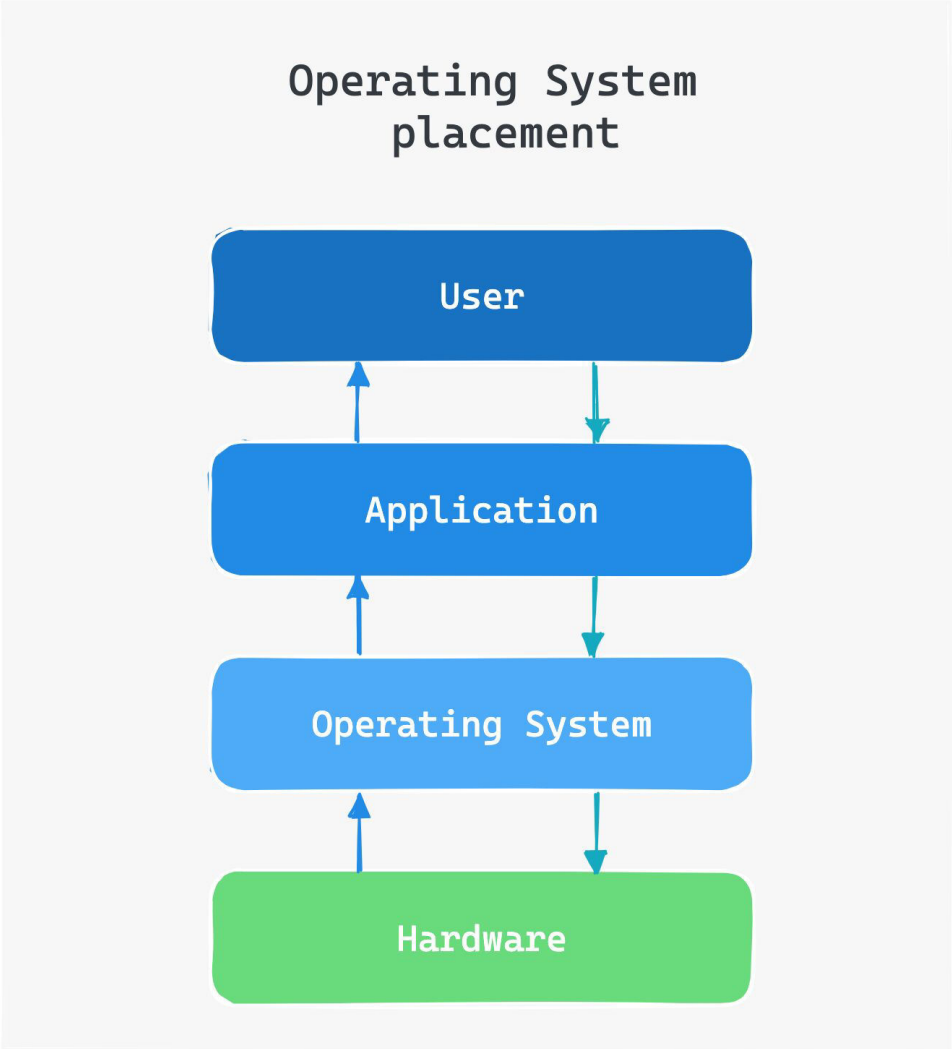
Table of Contents

Chapter-1

Operating System

1.1 Operating System:

- An operating system acts as an intermediary between the user of a computer and the computer hardware.
- It controls and manages the hardware and coordinates its use among the various application programs for the various users.



- A computer system can be divided roughly into four components:
  the hardware, the operating system, the application programs,and the users.

→ The Hardware :
The central processing unit (CPU), the memory, and the input/output (I/O) devices : provides the basic computing resources for the system.

→ The Application programs :
Such as word processors, spreadsheets, compilers, and Web browsers : define the ways in which these resources are used to solve users' computing problems.

Note: The operating system provides a means for proper use of these resources in the computer system.

→ Operating system's Role from two viewpoints:

User View

System View

1.2 User View:

- The operating system is designed mostly for ease of use.
- The goal is to maximize the work the user is performing.
- In mainframe or a minicomputer many users are accessing the same computer through other terminals.These users share resources and may exchange information.
- The operating system in such cases is designed to maximize resource-utilization to assure that all available CPU time, memory, and I/Oare used efficiently and that no individual user takes more than their fair share.

1.3 System View:

- In this context, we can view an operating system as a resource allocator.

- A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on.

- The operating system acts as the manager of these resources.

- It must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently.

- An operating system is a control program which manages the execution of user programs to prevent errors and improper use of the system and its resources.

→ A computer system can be organized as :

- Single-Processor Systems – On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.

- Multiprocessor Systems – Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

1.4 Multiprocessor Systems:

→ Multiprocessor systems have the main advantage of :

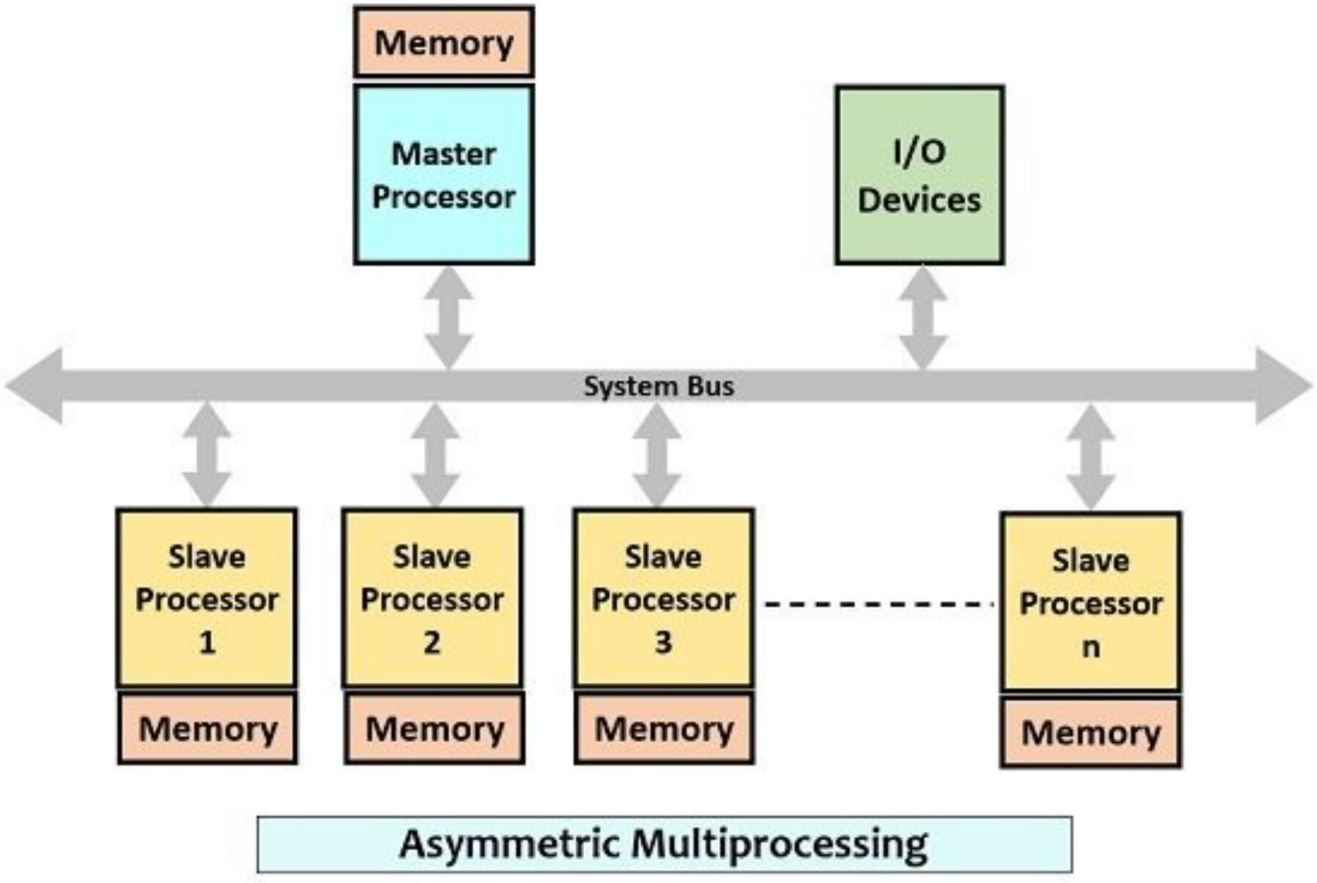
- Increased throughput – Increasing the number of processors, we expect to get more work done in less time.

- Increased reliability – In multiprocessor systems the failure of one processor will not halt the system, as the other processors can pick up a share of the work of the failed processor

Multiple-processor systems are of two types:

→ Asymmetric multiprocessing :

- In which each processor is assigned a specific task.
- A processor(boss) controls the system, the other processors(workers) either look to the boss for instruction or have predefined tasks.
- The boss processor schedules and allocates work to the worker processors.
- This scheme defines a boss-worker relationship.



Asymmetric Multiprocessing

→ Symmetric multiprocessing(SMP) :

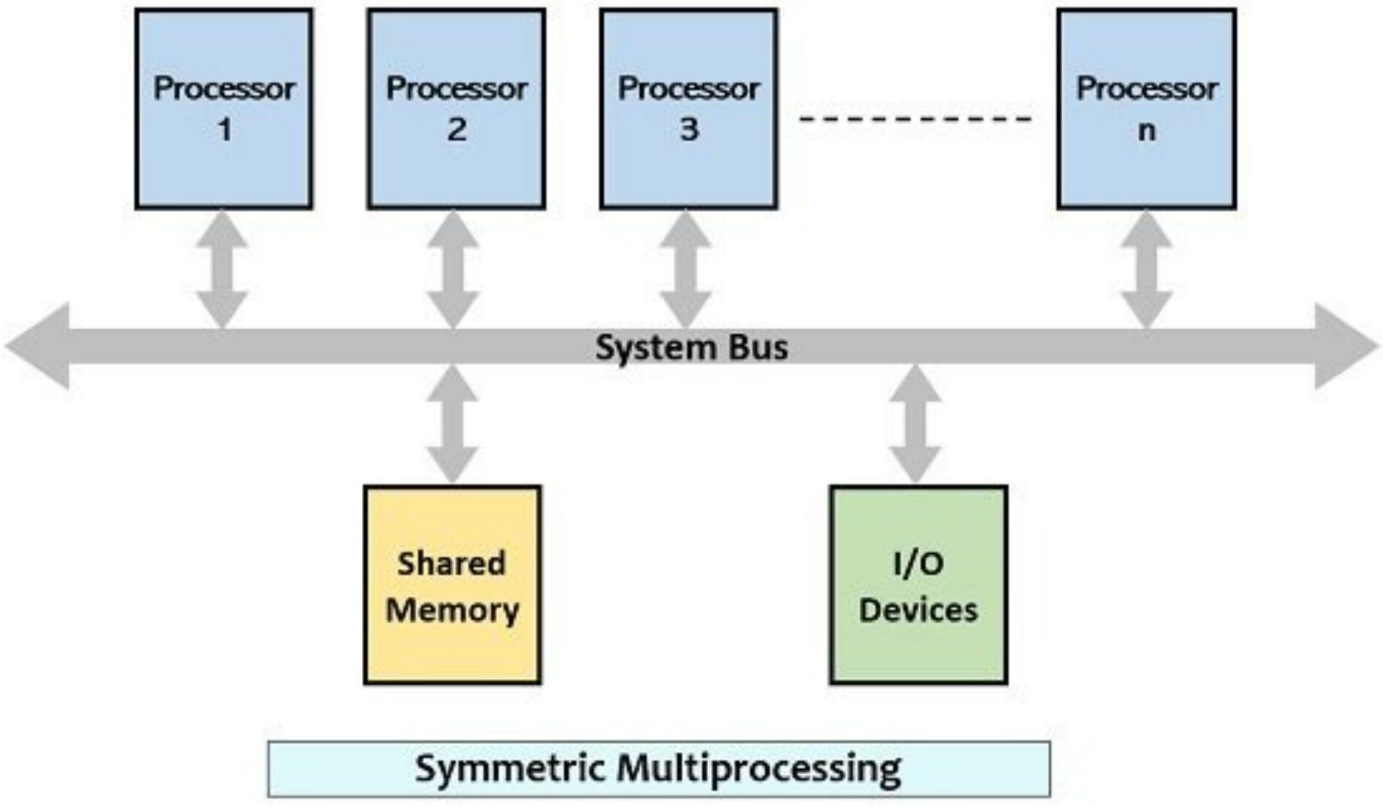
- In which each processor performs all tasks within the operating system.

- SMP means that all processors are peers; no boss-worker relationship exist between processors.

- Each processor has its own set of registers, as well as a private or local cache. However, all processors share same physical memory.

- The benefit of this model is that many processes can run simultaneously.

  i.e, N processes can run if there are N CPUs. However, we must carefully control I/O to ensure that the data reach the appropriate processor.

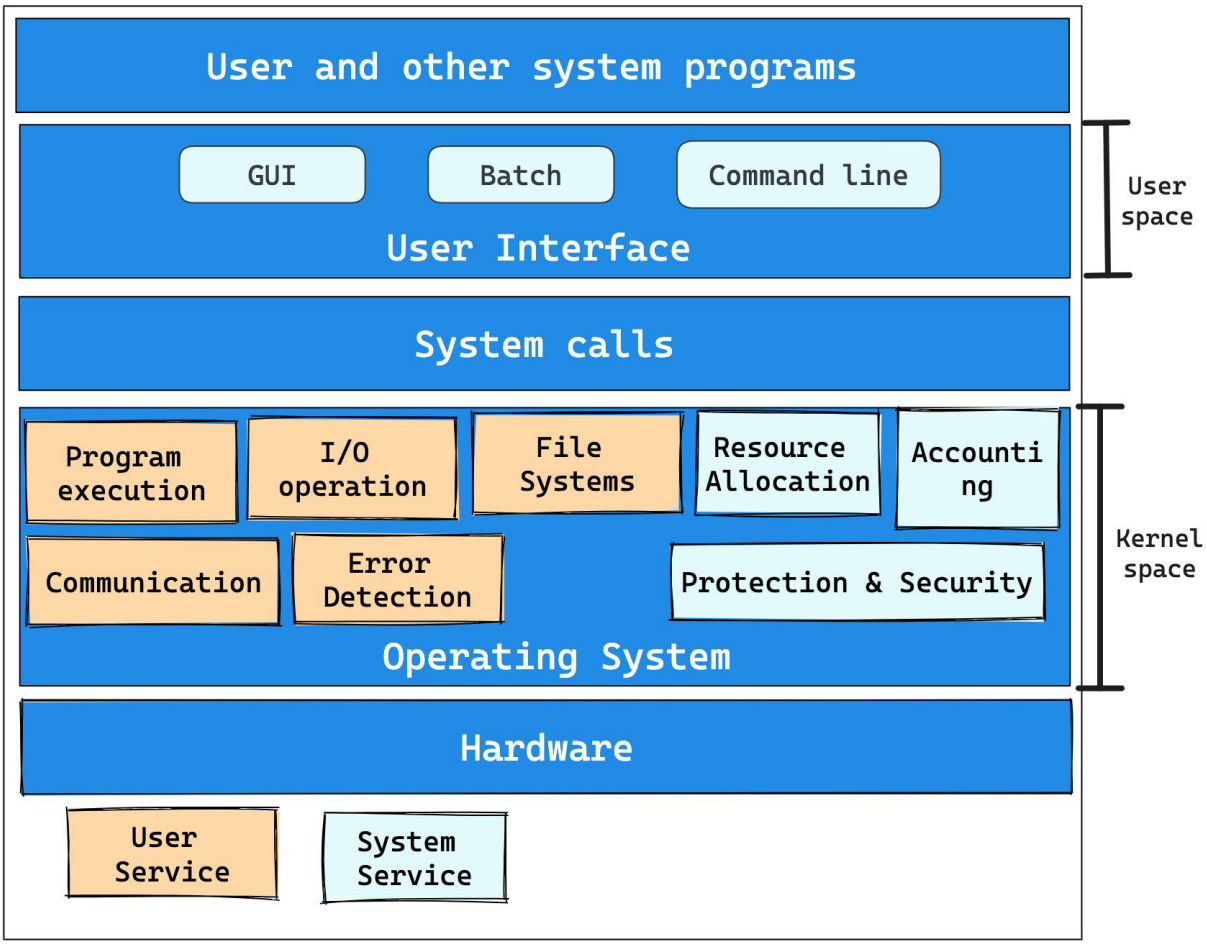


Chapter-2

Operating System Structure

- An operating system provides the environment within which programs are executed.

- It provides certain services to programs and to the users of those programs.



2.1 Operating-System Services:

→ User Services:

- User interface :
  All operating systems have a user interface (UI).

  The interface could be :

  → Command-line interface : Uses text commands.

  → A batch interface : Commands are entered into files and those files are executed.

  → Graphical User Interface(GUI)

- Program Execution :
  → The system must be able to load a program into memory and to run that program.

  → The program must be able to end its execution, either normally or abnormally (indicating error).

- I/O operations :
  → A running program may require I/O, which may involve a file or an I/O device.

  → For efficiency and protection, users usually cannot control I/O devices directly. Hence the operating system will provide a means to do I/O.

- File-System Manipulation :

  → Programs need to read and write from/ to files and directories. They also need to create/ delete them, search for a given file etc.

  → Operating systems include permissions management to allow or deny access to files and directories based on file ownership.

- Communications : There are many circumstances in which one process needs to exchange information with another process.

  Communications may be implemented via :

  → Shared Memory : In which two or more processes read and write to a shared section of memory.

  → Message Passing : In which packets of information is exchanged between processes by the operating system.

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

- Error detection : Errors may occur in the CPU, memory hardware(such as a memory error), in I/O devices (a connection failure on a network), and in the user program (such as an arithmetic overflow, accessing an illegal memory location).

  So for each type of error, the operating system should take the appropriate action to ensure that the system runs smoothly.

System Services:

- Resource allocation : The operating system manages many different types of resources. such as CPU cycles, main memory, and file storage.

  When there are multiple users or multiple jobs running at the same time, operating system fairly allocates resources to each of them.

- Accounting : We want to keep track of which users use how much and what kinds of computer resources.

  Resource usage statistics are valuable for developers to reconfigure the system to improve computing services.

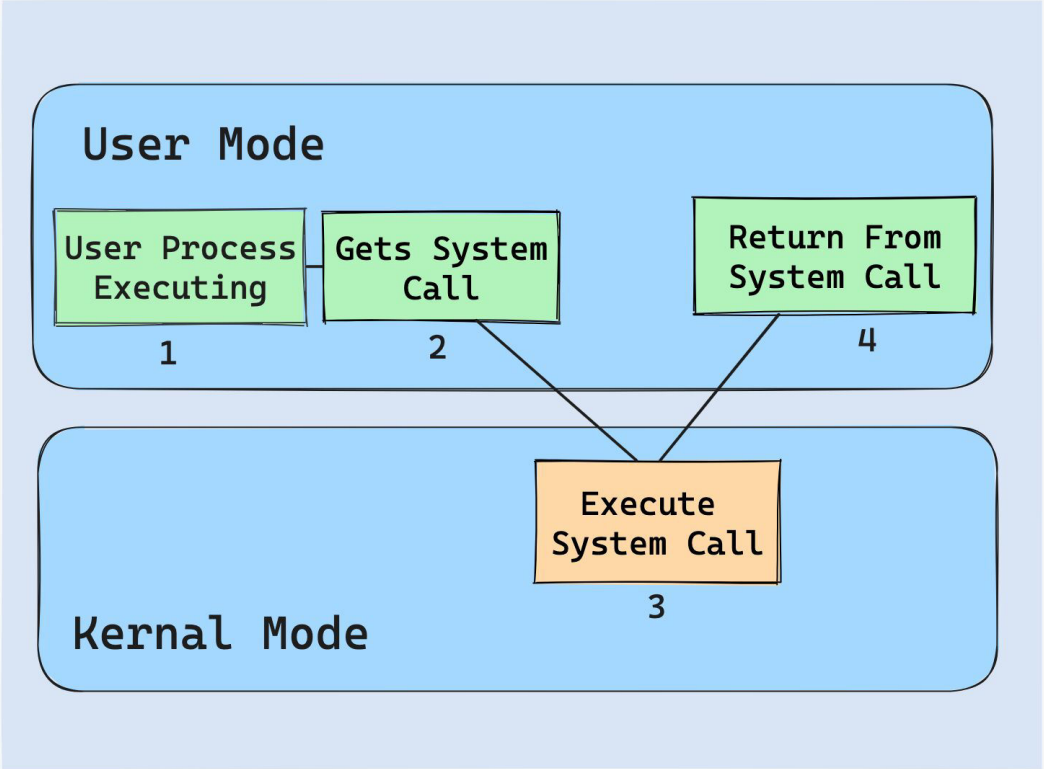
- Protection and security : Protection involves ensuring that all access to system resources are protected and controlled.

  Eg : When several separate processes execute concurrently, it should not be possible for one process to interfere with the others.

2.2 System Calls:

- System calls provide an interface to the services made availablen by an operating system.

Working Of A System Call



→ Let's take an example to illustrate how system calls are used :

Writing a simple program to read data from one file and copy them to another file

→ The first input that the program will need is the names of the two files:

- Input file
- Output file

→ Then the program must open the input file open() system call and create the output file create() system call.

→ When both files are set up successfully, the program reads from the input file read() system call and writes to the output file write() system call.

→ After the entire file is copied, the program may close both files close() system call.

→ Finally after successful completion the program can terminate normally end() system call.

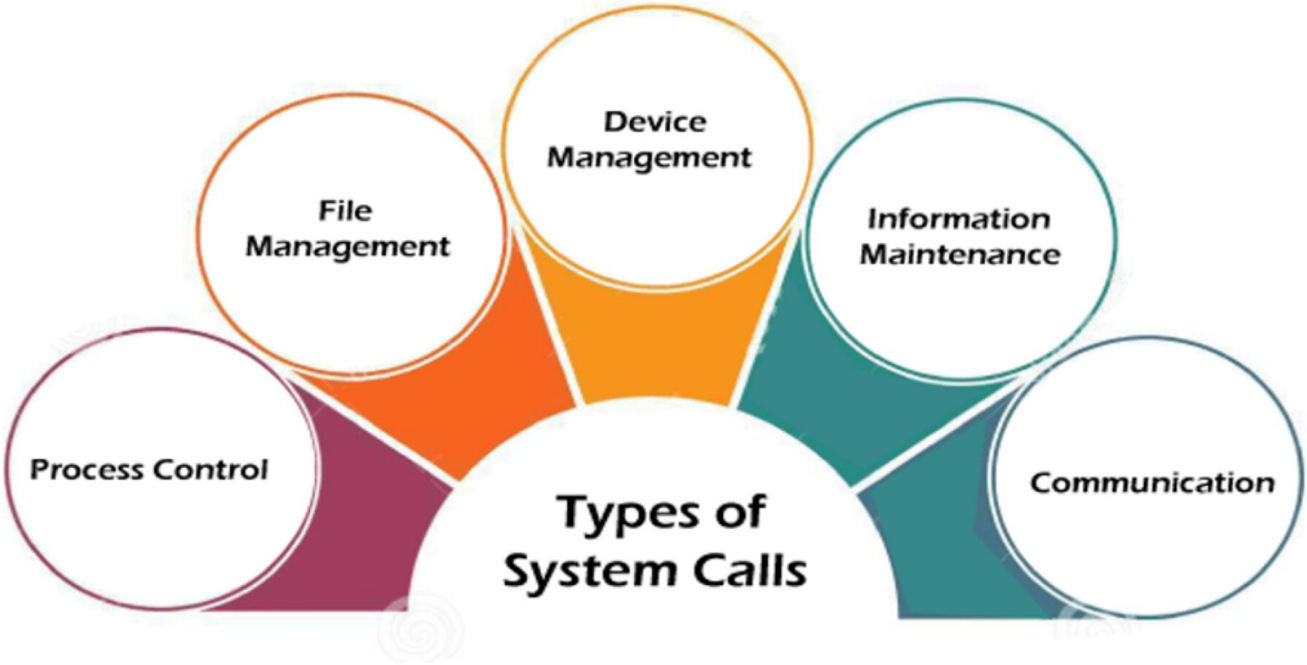
Note : There are several possibilities of occurrence of errors during process execution.

Possible error conditions for each operation may require additional system calls.

Eg: When the program tries to open the input file and there is no file with that name or that the file is protected against access.

In this case, the program should print a message on the console (another system call) and then terminate abort() system call.

2.3 Types of System Calls:



Process Control

→ Operating system loads the program to main memory and executes.

→ The operating system has the ablity to manage and control the process excecution .

- A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()).

- A process has the capability to create a new process using fork( ) command.

- A process or job executing one program may want to load() and execute() another program.

- We can manipulate process using it's attributes like the job's priority, its maximum allowable execution time, and so on get process attributes() and set process attributes().

- We can also want to wait for a specific event to occur (wait event()). The jobs or processes should then signal when that event has occurred (signal event()).

- Two or more processes may share data. To ensure the data integrity operating systems provides system calls allowing a process to lock shared data(lock()). Then no other process can access the data until the lock is released (release lock()).

- When the process completes it's execution it invokes exit() system call to terminate.(The process can return a zero or non-zero status code).

File Management

→ File management basically deals with files manipulation.

- We may need to to create() and delete() files. Once the file is created, we need to open() it and read() or write() from/into it. Finally, we need to close() the files after use.

- get file attributes() and set file attributes() are used for retrieving file attributes like file name, file type and so on.

Device Management

→ A process may need several resources to execute—main memory, disk drives, access to files, and so on.

→ If the resources are available, they can be granted. Otherwise, the process will have to wait until sufficient resources are available.

- A system with multiple users may require us to first request() a device, to use it and after we are finished with the device, we release() it.

- (These functions are similar to the open() and close() system calls for files)

Information Maintenance

→ System calls may also return information about the system,such as the number of current users, the amount of free memory or disk space and so on.

→ It is also possible to manipulate :

- processes using get process attributes() and set process attributes()
- files using get file attributes() and set file attributes()
- I/O devices get device attributes() and set device attributes()

Communication

→ There are two common models of interprocess communication:

- Shared Memory : In which two or more processes read and write to a shared section of memory.

  The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

In the shared-memory model, processes use shared_memory_create() and shared_ memory_ attach() system calls to create and gain access to regions of memory owned by other processes.

- Message Passing : In which packets of information is exchanged between processes by the operating system Before communication can take place, a connection must be opened.

  open connection() and close connection() system calls are used opening and closing connection.

  The recipient process should give permission for communication to take place with an accept connection() system call.

→ Note : Shared memory allows maximum speed and convenience of communication, since both the communication processes are running within same computer.

Protection

→ With the advent of networking and the Internet, all computer systems, from servers to mobile devices, must be protected.

So there should be controlled access to these resources

- System calls set permission() and get permission(), provide protection by allowing only users who have right permissions to use the resources.

- The system calls allow_user() and deny_user() specify whether particular users can-or cannot-be allowed access to certain resources.

Chapter-3

Processes

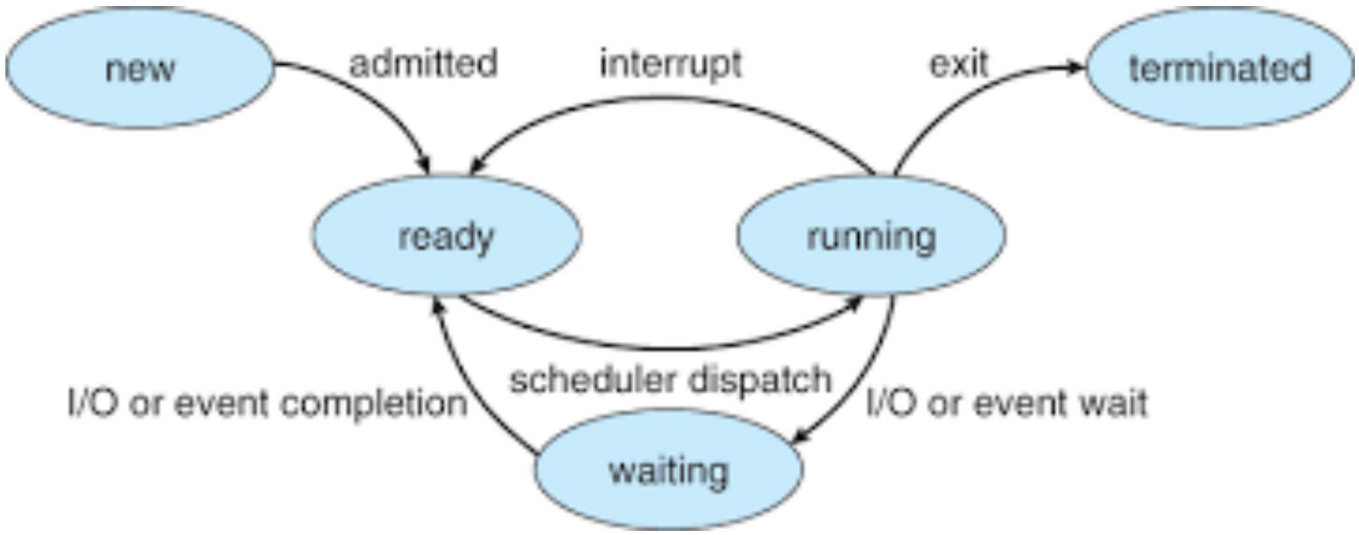
→ A process is the unit of work in a modern time-sharing system.

Modern computer systems allow multiple programs to be loaded into memory and executed concurrently.

3.1 Process State:

→ As a process executes, it changes state.

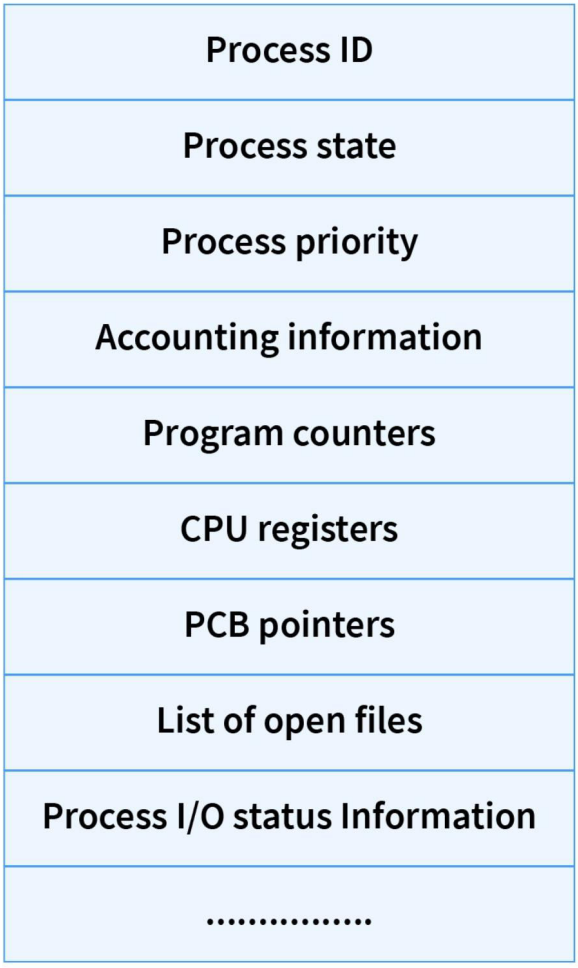
The state of a process is defined in part by the current activity of that process.



➔ A process may be in one of the following states:

- New : The process is being created.
- Running : Instructions are being executed.
- Waiting : The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready : The process is waiting to be assigned to a processor.
- Terminated : The process has finished execution.

3.2 Process Control Block:



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

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

➔ It contains information associated with a specific process like:

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

- CPU Registers : The registers include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, the state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward

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

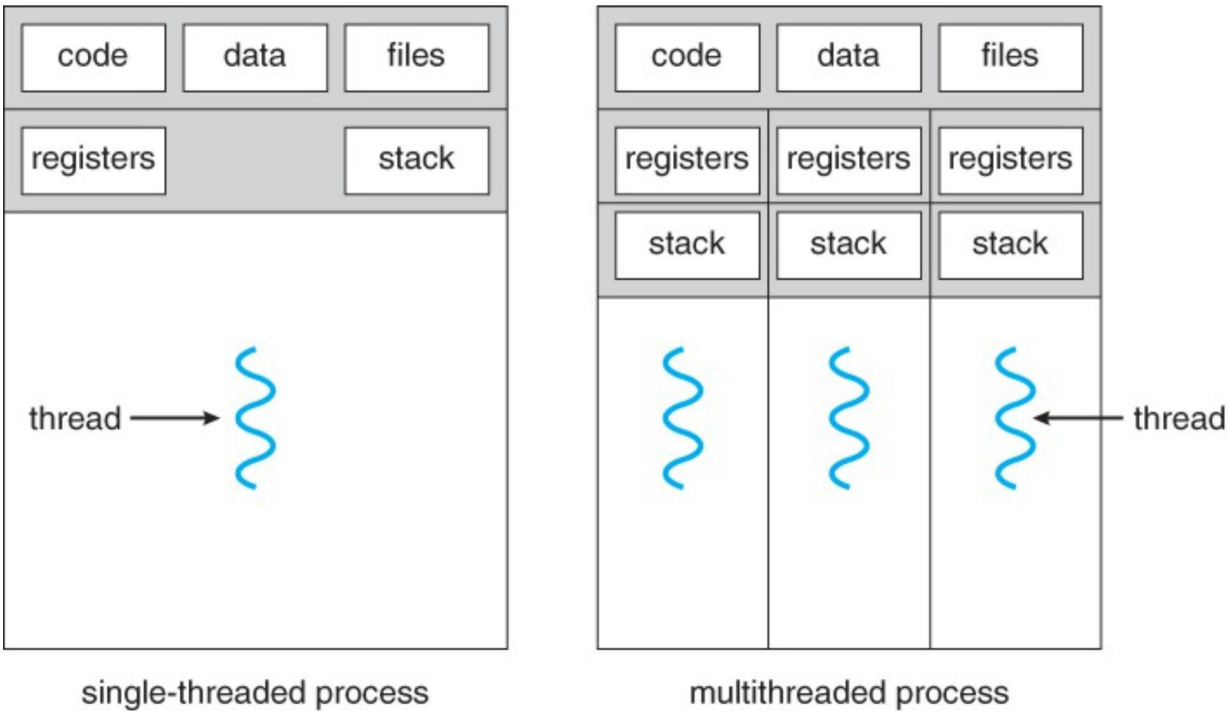
- Memory-Management Information : This information may include such items such as the value of the base and limit registers and the page tables or the segment tables

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

- I/O Status Information : This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

- NOTE : PCB simply serves as the repository for any information that may vary from process to process.

3.3 Threads:



single-threaded process multithreaded process

- A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control.

→ For example, when a process is running a word-processor program, a single thread of instructions is being executed.

- Single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time

- Multiple threads can run in parallel. On a system that supports threads, the PCB is expanded to include information for each thread.

3.4 Process Scheduling:

→ The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

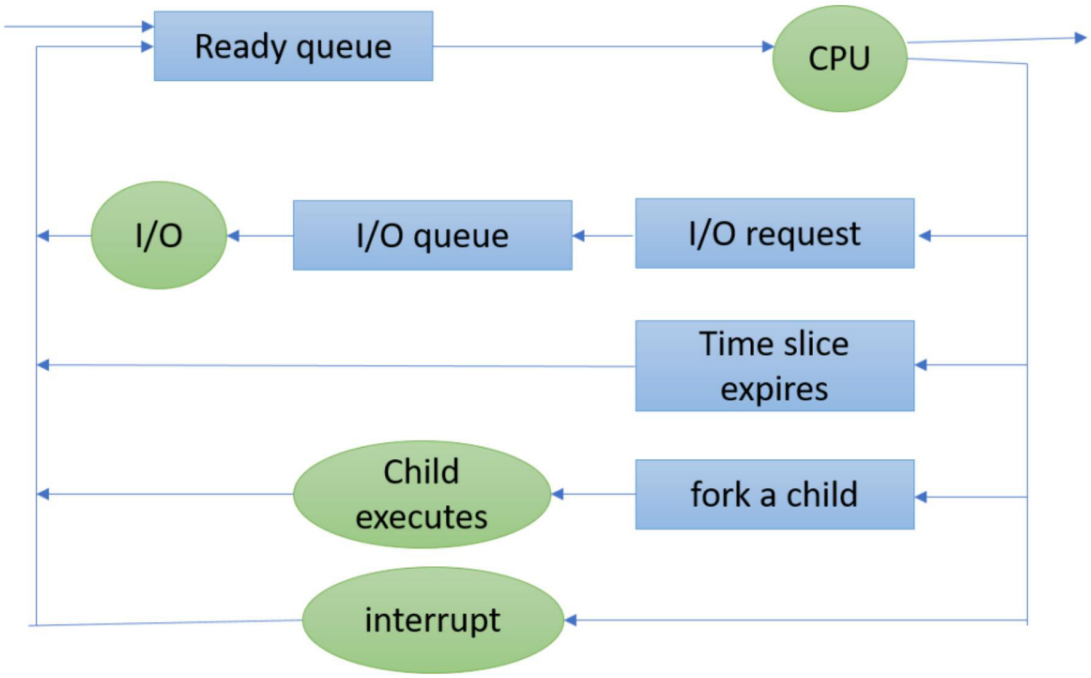
- For a single-processor system, there will neverbe more than one running process.

- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

3.5 Scheduling Queues:

→ As processes enter the system, they are put into a job queue, which consists of all processes in the system.

- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.
  This queue is generally stored as a linked list.



- A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

- The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits or is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

→ Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.
The process therefore may have to wait for the disk.

- The list of processes waiting for a particular I/O device are place in it's device queue.
Each device has its own device queue.

→ A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched.

→ Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

- Note: In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.

→ A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

- Each rectangular box represents a queue.
- Two types of queues are present: the ready queue and a set of device queues.
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

3.6 Schedulers :

→ A process migrates among the various scheduling queues throughout its lifetime.

- The operating system must select processes from these queues in some fashion.

- The selection process is carried out by the appropriate scheduler.

→ Long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.

→ Short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

- The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

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

Later, the process can be reintroduced into memory,and its execution can be continued where it left off.

- This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler.

- NOTE: The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

→ It is important that the long-term scheduler make a careful selection.
In general, most processes can be described as either I/O bound or CPU bound.

→ An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.

→ A CPU-bound process, in contrast, 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.

- If all processes are CPU bound, the I/O waiting queue will always be empty, devices will go unused, and again the system will be unbalanced.

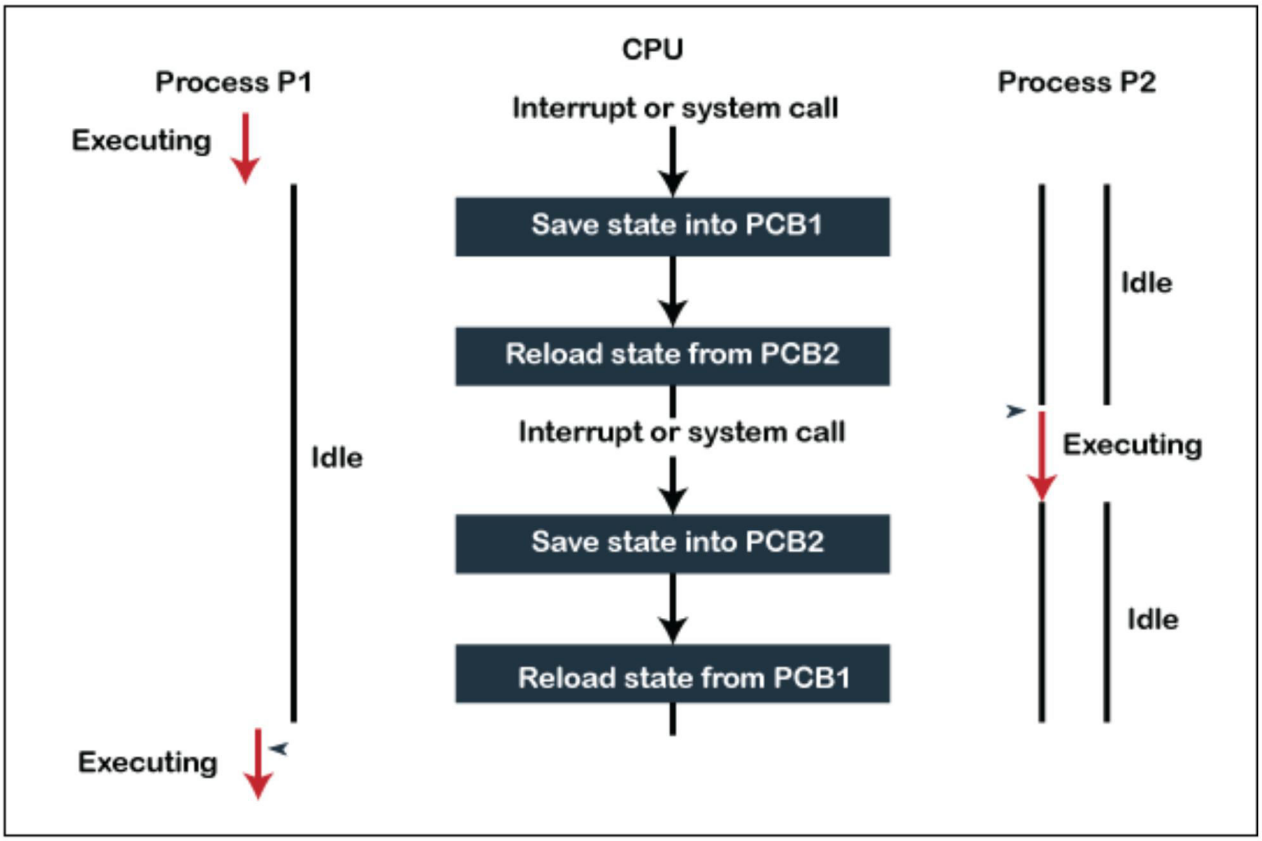
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

3.7 Context Switch :

→ Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.

- When an interrupt occurs, the system needs to save the current context of the process (state save) running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.



→ 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.

3.8 Interprocess Communication :

→ Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

- A process is independent if it cannot affect or be affected by the other processes executing in the system.

- Any process that does not share data with any other process is independent.

- A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

→ Reasons for providing an environment that allows process cooperation:

- Information Sharing : Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- Computation Speedup : If we want a particular task to run faster,we must break it into subtasks, each of which will be executing in parallel with the others.

- Modularity : We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

- Convenience : Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music,and compiling in parallel.

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and in formation.

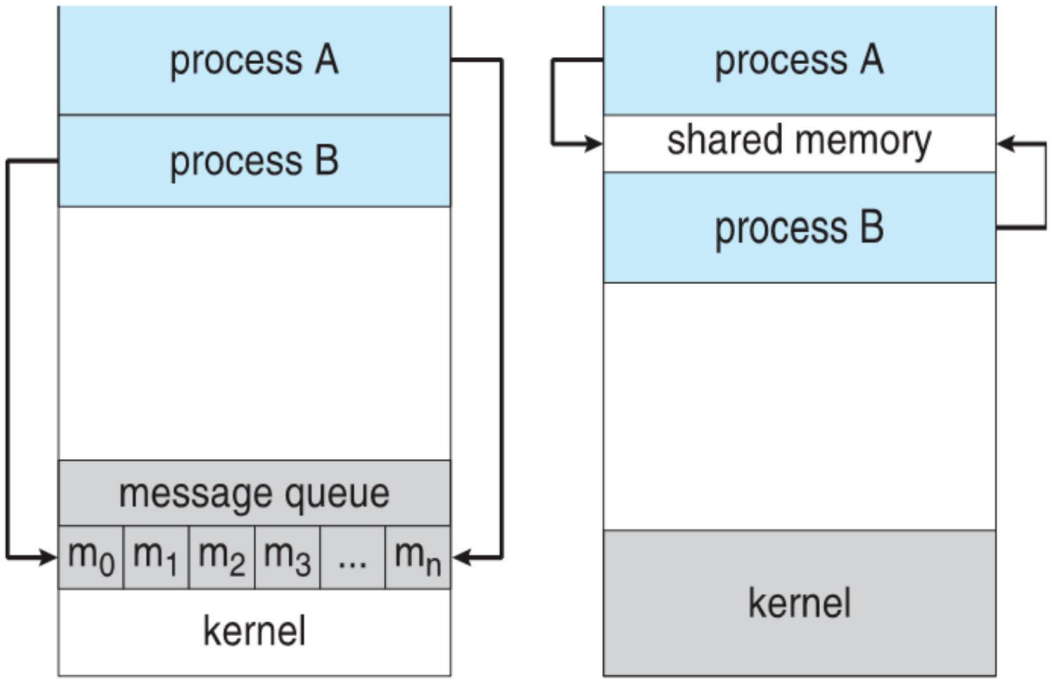
→ There are two fundamental models of interprocess communication: shared memory and message passing.

- In the shared-memory model, a region of memory that is shared by cooperating processes is established.

- Processes can then exchange information by reading and writing data to the shared region.

- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

(a) Message passing. (b) shared memory.



- Message passing is also easier to implement in a distributed system than shared memory.

- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

- In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

- Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches.

3.8 Shared-Memory Systems :

→ Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

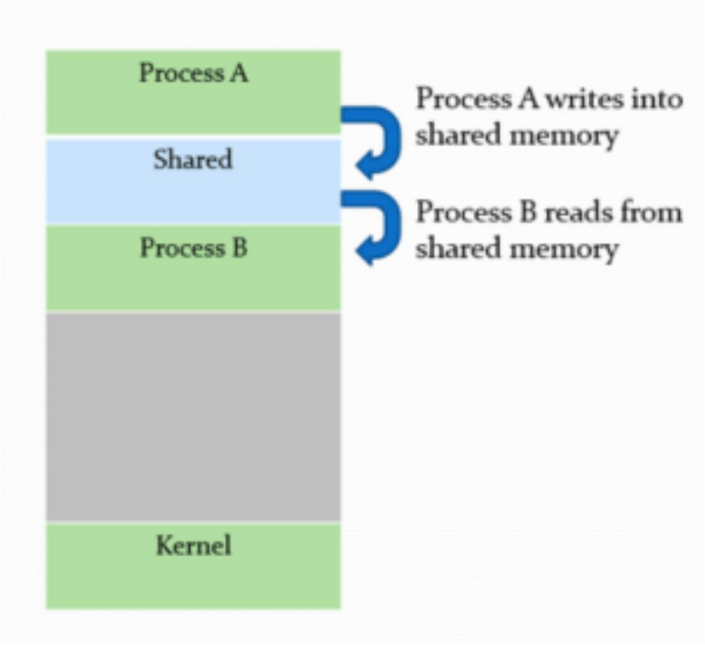
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment

- NOTE: Normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction.

- They can then exchange information by reading and writing data in the shared areas.

- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Eg: Producer-Consumer problem

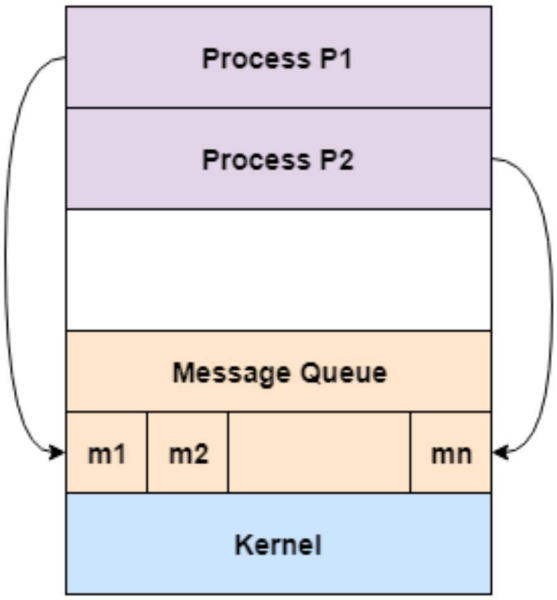


3.9 Message-Passing Systems :

→ Cooperating processes to communicate with each other via a message-passing facility.

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.



Message Passing Model

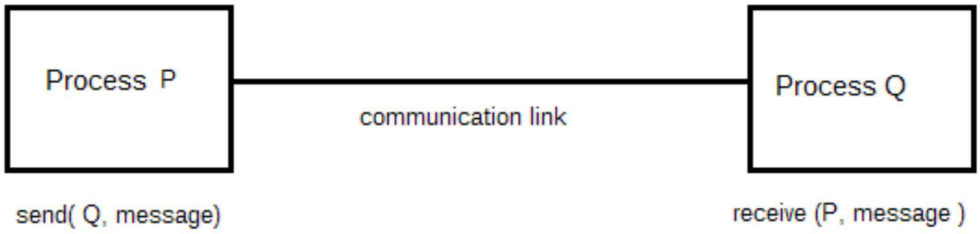
→ If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link.

- A message-passing facility provides at least two operations:

  send(message) receive(message)

  - Direct or indirect communication(Naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

3.9.1 Naming :



→ Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- In this scheme, the send() and receive() primitives are defined as:

  - send(P, message)–Send a message to process P.
  - receive(Q, message)–Receive a message from process Q.

→ A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate.
  The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.

- Between each pair of processes, there exists exactly one link.

- This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate

→ A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

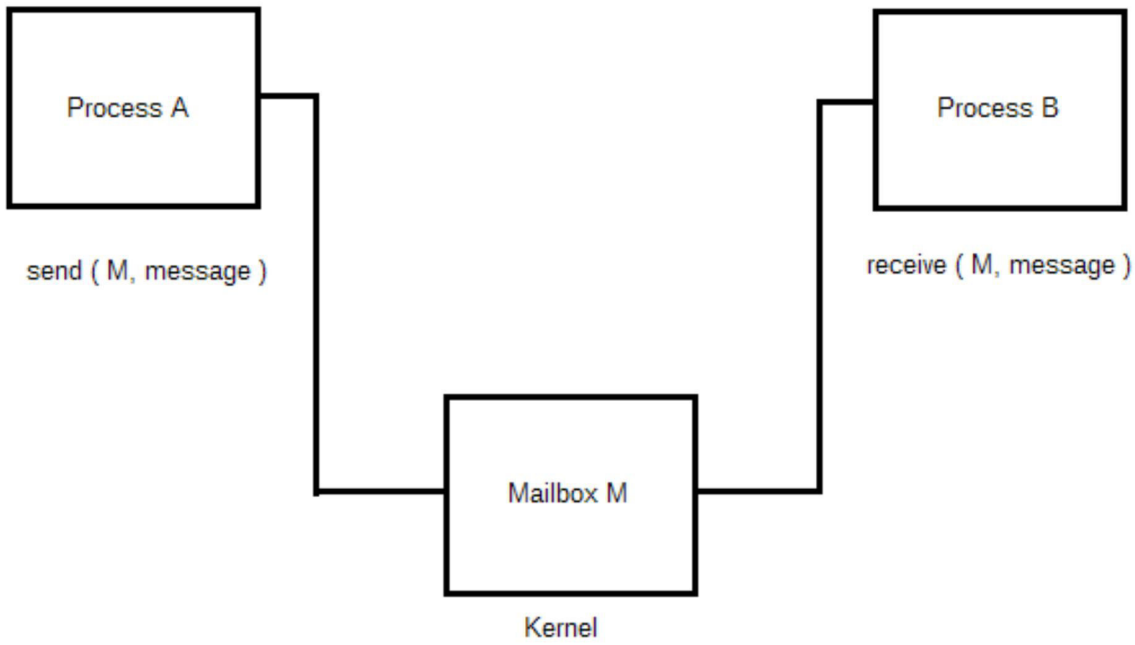
- In this scheme, the send() and receive() primitives are defined as follows:

  - send(P, message)–Send a message to process P.

  - receive(id, message)–Receive a message from any process.

  The variable id is set to the name of the process with which communication has taken place.

→ With indirect communication, the messages are sent to and received from mailboxes, or ports.



→ A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

→ A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

→ In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.

3.9.2 Synchronization :

→ Communication between processes takes place through calls to send() and receive() primitives.

• Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.

- Blocking send : The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Nonblocking send : The sending process sends the message and resumes operation.
- Blocking receive : The receiver blocks until a message is available.
- Nonblocking receive : The receiver retrieves either a valid message or a null.

3.9.3 Buffering :

→ Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity : The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it

- Bounded capacity : The queue has finite length n ;thus, at most 'n' messages can reside in it.If the queue is not full when a new message is sent, the message is placed in the queue.

- Unbounded capacity : The queue's length is potentially infinite thus , any number of messages can wait in it. The sender never blocks.

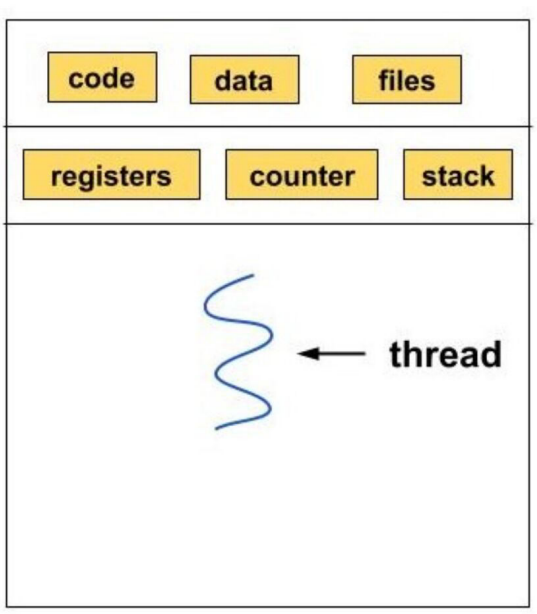
- NOTE: The zero-capacity case is sometimes referred to as a message system with no buffering

Chapter-4

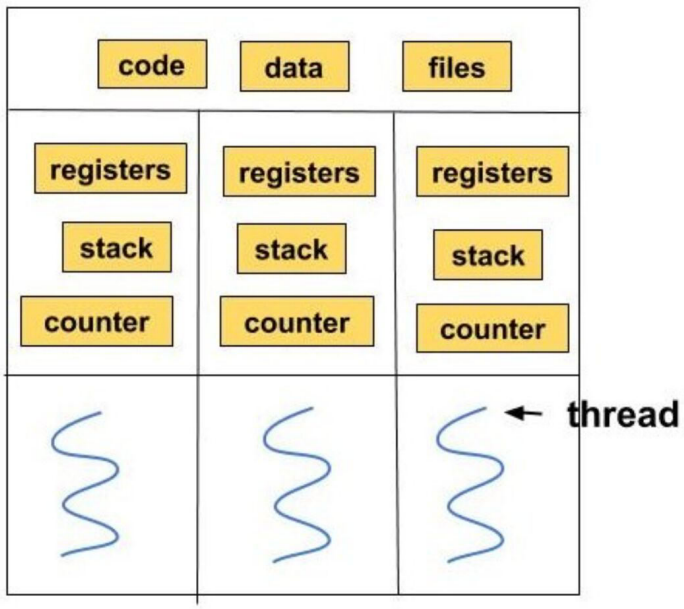
Threads

→ If a process has multiple threads of control, it can perform more than one task at a time.

- When a process with multiple threads is called multi-threaded process.



Single-threaded process



Multi-threaded process

→ Eg-1: A web browser might have one thread display images or text while another thread retrieves data from the network.

→ Eg-2: A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

→ In certain situations, a single application may be required to perform several similar tasks.

- For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it.

- If the web server ran as a single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

→ One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request.

→ Process creation is time consuming and resource intensive,

- If the new process will perform the same tasks as the existing process, why incur all the overhead of creating new process?

- It is generally more efficient to use one process that contains multiple threads.If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.

- When a request is made, the server creates a new thread to service the request and resume listening for additional requests. This allows the server to service several concurrent requests.

4.1 Benefits of Multithreading:

1. Responsiveness : Multithreading an interactive application may allow a program to continue running even if part of it is blocked(thread blocking) thereby increasing responsiveness to the user.

2. Resource Sharing : Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. Economy : Threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

4. Scalability : The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

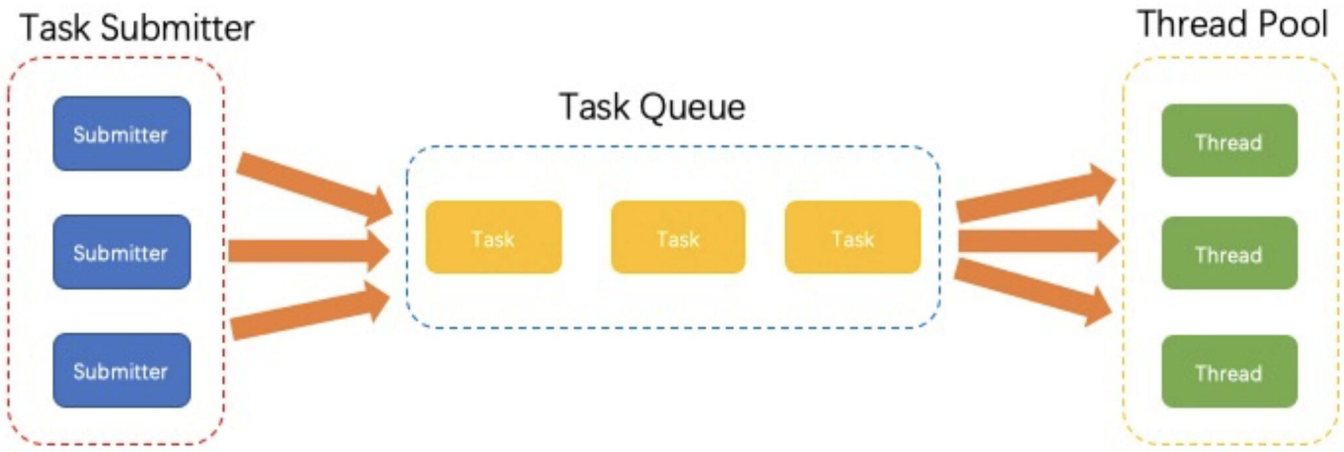
4.2 Thread Pools:

→ Whenever the server receives a request, it creates a separate thread to service the request.

- The first concern is the amount of time required to create the threa and that the thread will be discarded once it has completed its work.

- If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system.

- Unlimited threads could exhaust system resources, such as CPU time or memory.



→ One of the solution for this could be : Create a number of threads at process startup and place them into a pool, where they sit and wait for work.

→ When a server receives a request, it awakens a thread from this pool—if one is available and passes it the request for service.

- Once the thread completes its service, it returns to the pool and awaits more work.

- If the pool contains no available thread, the server waits until one becomes free.

→ Thread pools offer these benefits:

1. Servicing a request with an existing thread is faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

3. Separating the task to be performed from the mechanics of creating the task.

- The number of threads in the pool can be based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.

4.3 Multicore Programming:

→ Multicore or multiprocessor systems – multiple computing cores on a single chip. (Each core appears as a separate processor to the operating system).

- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

- Concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.

→ There are two types of parallelism:

1. Data parallelism
2. Task parallelism

- Data parallelism focuses on distributing subsets of the same data across multiple computing cores.

- Task parallelism involves distributing not data but tasks (threads) across multiple computing cores.

4.4 Multithreading Models

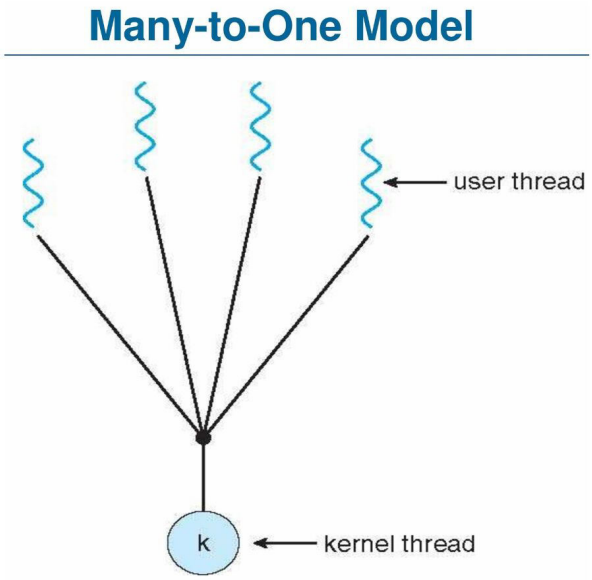
→ There are two types of threads:

- User Thread : User threads are implemented by users.
- Kernel Thread : Kernel threads are are supported and managed directly by the operating system.

4.4.1 Many-to-One Model:

→ The many-to-one model maps many user-level threads to one kernel thread.

- Many-to-one model allows the developer to create as many user threads as possible, it does not result in true concurrency, because the kernel can schedule only one thread at a time.



4.4.2 One-to-One Model:

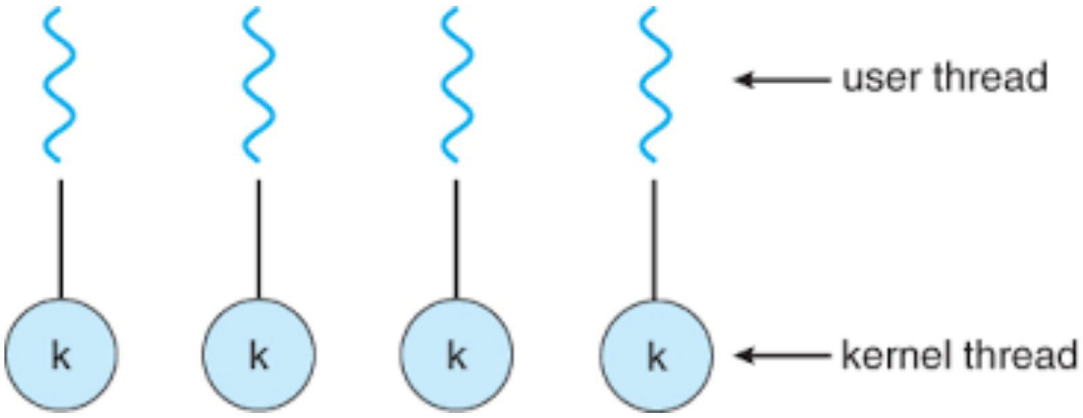
→ The one-to-one model maps each user thread to a kernel thread.

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

→ NOTE : The running thread will block when it must wait for some event to occur

- It also allows multiple threads to run in parallel on multiprocessors.

- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

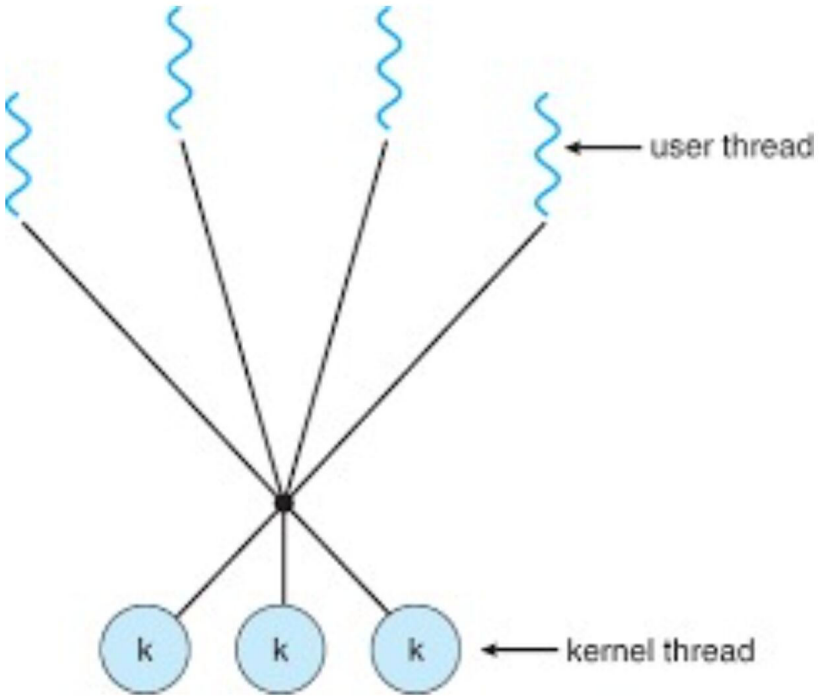


- Since the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

- One-to-one model allows the developer to create as many user threads as possible, it does not result in true concurrency, because the kernel can schedule only one thread at a time.

4.4.3 Many-to-Many Model:

→ The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.



- In many-to-many model developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

Chapter-5

Process Synchronization

→ A cooperating process is one that can affect or be affected by other processes executing in the system.

- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages(message passing).

- Concurrent access to shared data may result in data inconsistency.

→ A system consisting of threads, all running asynchronously and possibly sharing data, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

- Race conditions typically occur in scenarios where multiple processes or threads access and manipulate shared resources concurrently.

→ Let's consider a simple example to illustrate a race condition: Suppose there is a shared variable called "counter" initialized to 0.

- Two concurrent processes, A and B, both read the value of the counter, increment it by 1, and write the new value back.

→ The execution steps of these processes might look like this:

Process A reads the value of counter (0).
Process B reads the value of counter (0).

Process A increments the counter by 1 (resulting in 1).
Process B increments the counter by 1 (also resulting in 1).

(since both processes A & B are running concurrently, so B does not see the updated data nor does A).

Process A writes the new value of the counter (1).
Process B writes the new value of the counter (1).

→ In this scenario, both processes intended to increment the counter, but due to the interleaving of their operations, the final value of the counter is incorrect (1 instead of 2).
This is an example of a race condition.

→ To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.

To make such a guarantee, the processes need to be synchronized in some way.

5.1 The Critical-Section Problem:

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

- Each process has a segment of code, called a critical section,in which the process may be changing common variables, updating a table, writing a file, and so on.

- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section to avoid data inconsistencies.

→ The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

- The section of code implementing the request to enter into critcal section is the entry section.

  The critical section may be followed by an exit section.

  The remaining code is the remainder section.

→ Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels.

- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be preempted.

- NOTE : Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources.

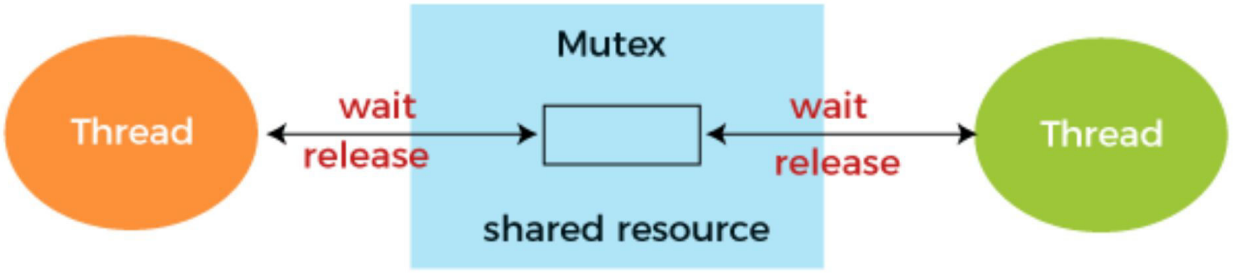
→ A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time but the same is not true for preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.

5.2 Mutex Locks:

→ The term mutex is short for mutual exclusion.

- We use the mutex lock to protect critical regions it provides mutual exclusion, meaning that it allows concurrent processes or threads to coordinate their access and prevent race conditions.

- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.



→ The basic operations associated with a mutex are:

- Lock Operation: Also known as the "acquire" or "wait" operation, it is used by a process or thread to request ownership of the mutex.

- If the mutex is currently unlocked, the requesting process or thread acquires it and continues executing.

- If the mutex is locked, the process or thread is blocked until the mutex becomes available.

→ Unlock Operation: Also known as the "release" operation, it is used to release ownership of the mutex.

- When a process or thread finishes executing its critical section of code, it releases the mutex, allowing other waiting processes or threads to acquire it.

→ The acquire() function acquires the lock, and the release() function releases the lock.

```
acquire() {
      while (!available);

 /* Until the available is true process must wait
 once the process is available, the process will acquire
 the lock and make the availability of lock as false. */

      available = false;;
}
```

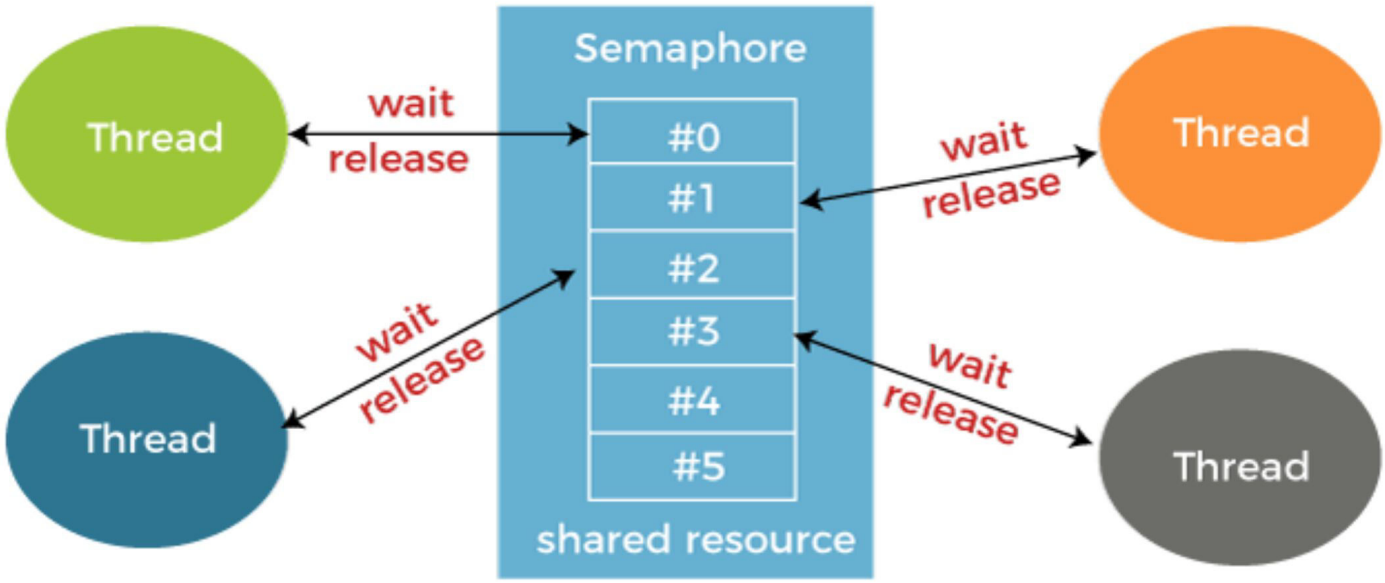
```
release() {
available = true;
}
```

5.3 Semaphores:

→ A semaphore is an integer variable, shared among multiple processes.

- The main aim of using a semaphore is process synchronization and access control for a common resource in a concurrent environment.

- Semaphores are a fundamental building block for synchronization in operating systems, and they play a crucial role in preventing race conditions and ensuring the correct and coordinated execution of concurrent processes/threads.



→ Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). (Refer to the image above).

- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- In case of counting semaphore,the semaphore can be modified to contain an integer and a waiting list of processes that requires the resource instance it represents.

- So each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes.

→ A signal() operation removes one process from the list of waiting processes and awakens that process.

→ The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

→ A semaphore has two indivisible (atomic) operations, namely: {wait} and {signal}.

```
wait(S) {
while (S ≤ 0);
        // busy wait
           S--;
}
```

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

Semaphore Types

→ There are two types of semaphores:

Binary semaphore
Counting Semaphore

→ A binary semaphore can have only two integer values: 0 or 1. Binary semaphores behave similarly to mutex locks.

→ A counting semaphore is again an integer value, which can range over an unrestricted domain. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

The semaphore is initialized to the number of resources available.

→ When a call to signal() invoked process, the process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

- NOTE: The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely(indefinite blocking or starvation) for an event that can be caused only by one of the waiting processes.

→ i.e, Let's say process A is waiting for the resource instance held by process B and process B is waiting for the resource instance held by process A.

- These 2 processes are in the waiting queue of some semaphore and waiting for resources indefinitely since none of the processes can finish their execution.

- When such a state is reached, these processes are said to be deadlocked.

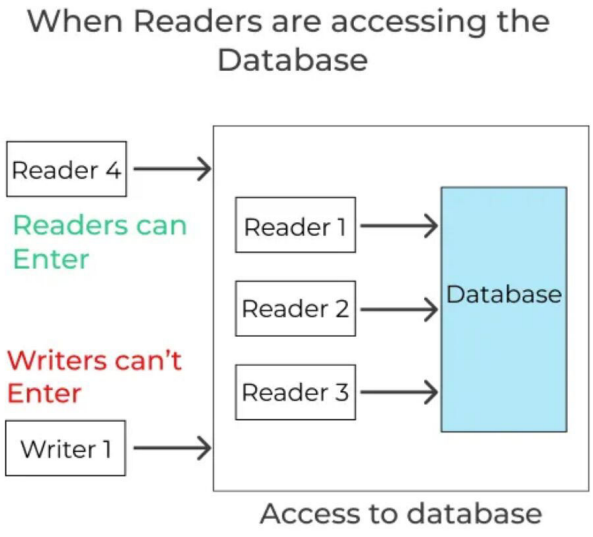
5.4 The Readers–Writers Problem:

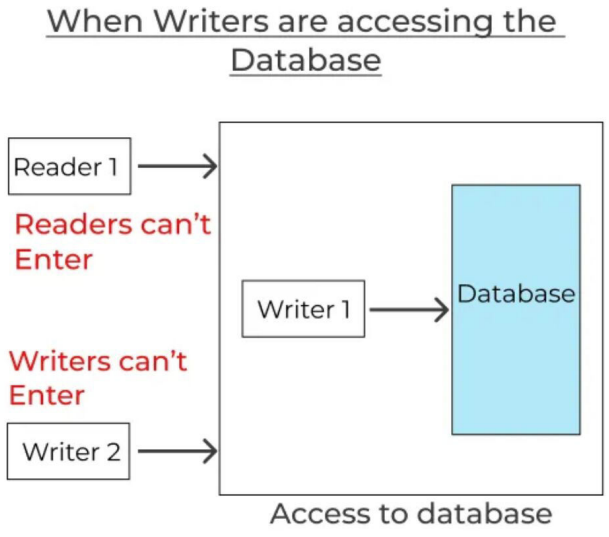
➔ Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database(reader processes), whereas others may want to update(writer processes) the database.

- Obviously, if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, there may be inconsistencies.

➔ 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.

- This synchronization problem is referred to as the readers – writers problem.





➔ Acquiring a reader – writer lock requires specifying the mode of the lock: either read or write access.

- When a process wishes only to read shared data, it requests the reader–writer lock in read mode.

- A process wishing to modify the shared data must request the lock in write mode.

➔ Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

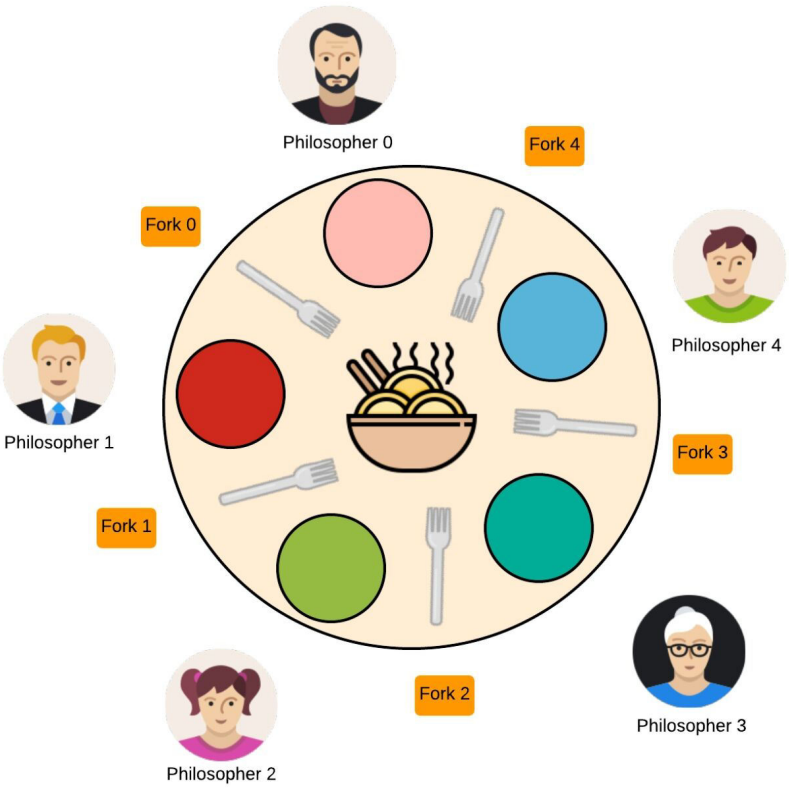
- Hence no reader be kept waiting unless a writer has already obtained permission to use the shared object.

- If a writer is waiting to access and update the object, no new readers may start reading.

5.5 The Dining-Philosophers Problem:

→ Dining-Philosophers a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

- Consider five philosophers who are seated at a round table for 5.



- The center of the table is a bowl of rice, and the table is laid with five single chopsticks.

- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him/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, a philosopher cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both chopsticks at the same time, he/she eats without releasing the chopsticks. When the philosopher is finished eating, then they put down both chopsticks.

→ Suppose that all five philosophers become hungry at the same time and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.
When each philosopher tries to grab their right chopstick, they will be delayed forever (deadlock).

→ Several possible remedies to the deadlock problem are replaced by:

- Allowing at most four philosophers to be sitting simultaneously at the table.

- Allowing a philosopher to pick up their chopsticks only if both chopsticks are available (to do this, they must pick them up in a critical section).

- One simple solution is to represent each chopstick with a semaphore.

- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore and releases his/her chopsticks by executing the signal() operation on the appropriate semaphores.

5.6 Monitors:

→ Monitor provides structured way to control access to shared resources and coordinate the execution of concurrent processes or threads.

- Monitors encapsulate shared data and the operations that can be performed on that data, ensuring that only one process or thread can execute the operations at a time.

→ A monitor consists of the following components:

- Shared Data : This refers to the data or resources that need to be accessed and manipulated by multiple processes or threads.

- The shared data is encapsulated within the monitor, making it inaccessible directly from outside.

→ Procedures or Methods : Monitors define a set of procedures or methods that operate on the shared data.

- These procedures are the only means for accessing and modifying the shared data.

- When a process or thread wants to access the shared data, it needs to call one of the procedures provided by the monitor.

→ Synchronization: Monitors ensure that only one process or thread can execute the procedures within the monitor at a time.

- This prevents concurrent access and potential race conditions.

- Typically, monitors provide mechanisms such as condition variables or entry queues to manage synchronization.

- Condition Variables: Condition variables are used within monitors to allow processes or threads to wait for specific conditions to be satisfied.

- Monitors provide a higher level of abstraction and encapsulation compared to lower-level synchronization primitives like semaphores or locks.

- They simplify the design and implementation of concurrent programs by providing a structured way to synchronize access to shared resources, reducing the likelihood of programming errors.

Chapter-6

CPU Scheduling

→ The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

- With multiprogramming, we try to keep the CPU busy all the time.

- Several processes are kept in memory at one time.If a process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.

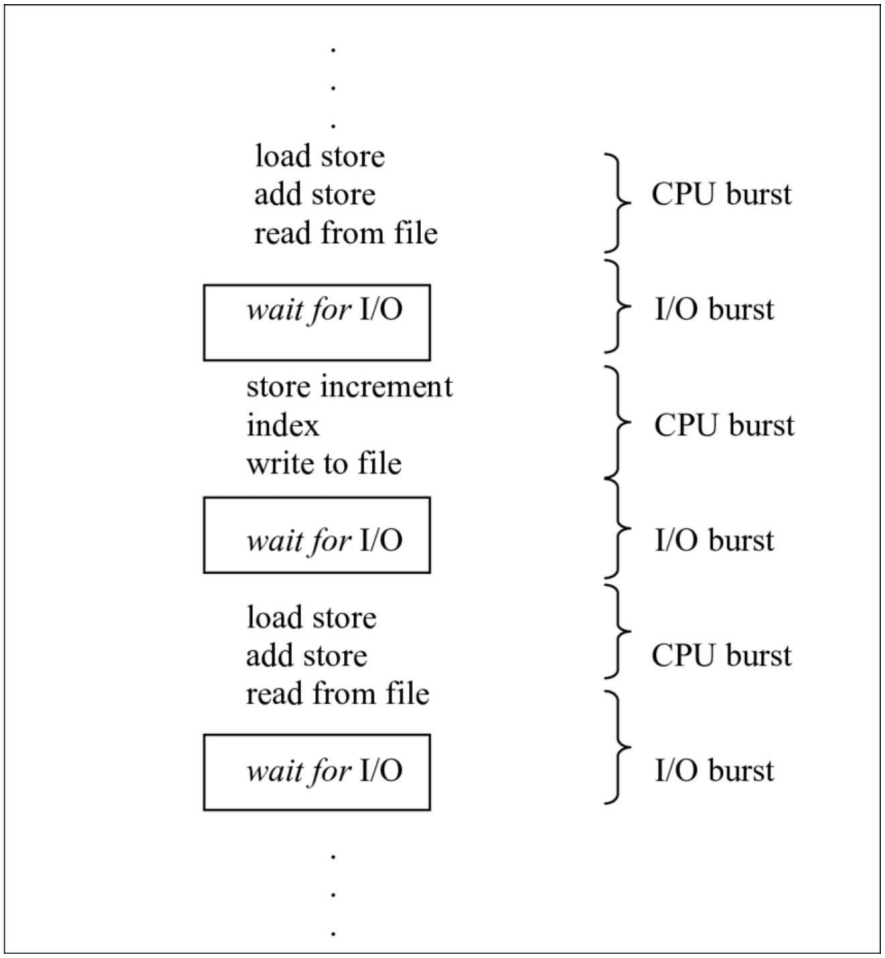
- CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

6.1 CPU–I/O Burst Cycle:

→ Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.

Processes alternate between these two states.

- Process execution begins with a CPU burst followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.



6.2 CPU Scheduler :

→ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

- The selection process is carried out by the short-term scheduler, or CPU scheduler.

- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

6.3 Preemptive Scheduling:

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

1. When a process switches from the running state to the waiting state

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).

4. When a process terminates

- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative. Otherwise, it is preemptive.

- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU.

6.4 Dispatcher:

→ The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

- The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

6.5 Scheduling Criteria:

→ Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

- CPU Utilization : We want to keep the CPU as busy as possible. CPU utilization can range from 0 to 100 present. In real time systems the CPU utuilisation is between 40 to 90 percent.

- Response Time : Submission of a request until the first response is produced.

- Throughput : The number of processes that are completed per time unit,called throughput.

- Turnaround Time : Turnaround time describes the time it takes to execute a process.

  The interval from the time of submission of a process to the time of completion is the turnaround time.

  Turnaround time is the sum of the periods spent waiting to get into memory,waiting in the ready queue, executing on the CPU, and doing I/O.

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

- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

6.6 Scheduling Algorithms:

➔ CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

There are many different CPU-scheduling algorithms.

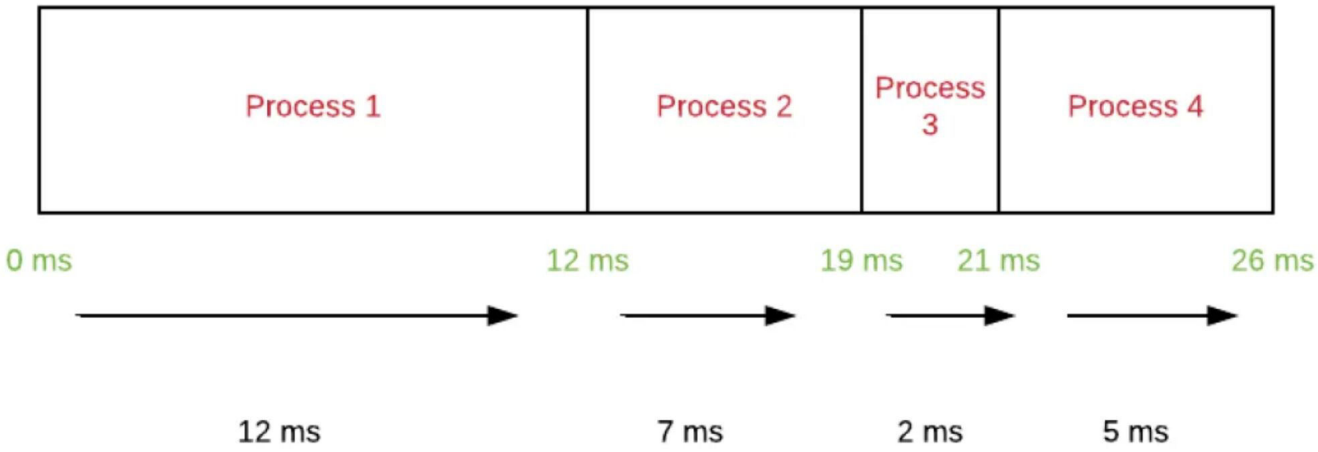
6.6.1 First-Come, First-Served Scheduling:

➔ CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm.

- With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

- When the CPU is free, it is allocated to the process then the running process is then removed from the queue.

- Note : The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.



- There is a convoy effect as all the other processes wait for the one big process to get off the CPU.
  This effect results in lower CPU and device utilization

- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

6.6.2 Shortest-Job-First Scheduling:

➔ A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm or shortest-next- CPU-burst algorithm.

- This algorithm associates with each process the length of the process's next CPU burst.

- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

➔ The SJF algorithm can be either preemptive or non preemptive.

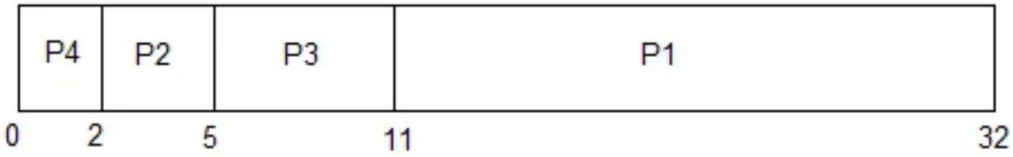
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.

- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

6.6.3 Priority Scheduling :

➔ The SJF algorithm is a special case of the general priority-scheduling algorithm.

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

➔ An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.

The larger the CPU burst, the lower the priority, and vice versa.

➔ Priority scheduling can be either preemptive or non preemptive.

- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

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

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,

The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

- A process that is ready to run but waiting for the CPU can be considered blocked.

- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

→ A solution to the problem of indefinite blockage of low-priority processes is aging.

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

6.6.4 Round-Robin Scheduling:

➔ The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems.

- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

- A small unit of time, called a time quantum or time slice, is defined.

The ready queue is treated as a circular queue.

- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of 1 time quantum.

➔ One of two things will then happen :

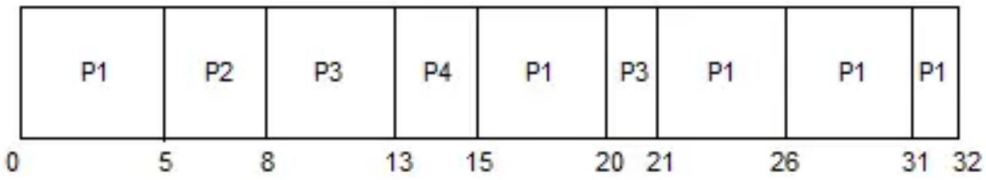
- The process may have a CPU burst of less than 1 time quantum.In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

- If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.

- A context switch will be executed, and the process will be put at the tail of the ready queue.
- The CPU scheduler will then select the next process in the ready queue.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

6.6.5 Multilevel Queue Scheduling:

➔ A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

Each queue has its own scheduling algorithm

- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type

- There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

➔ Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

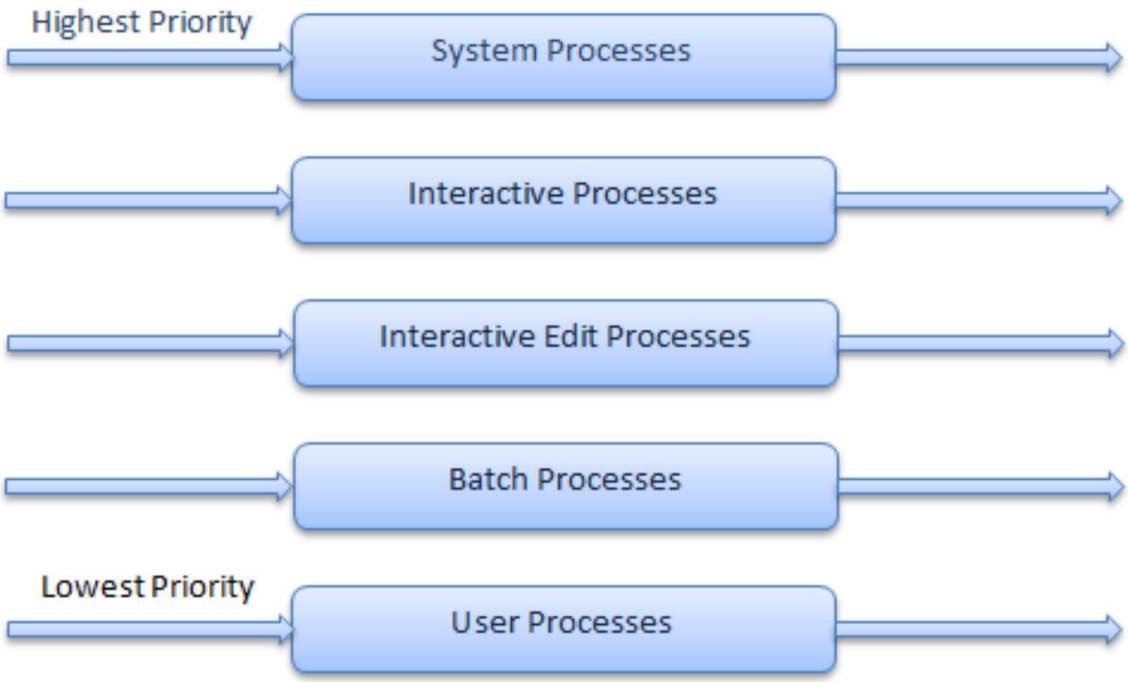
1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes

Each queue has absolute priority over lower-priority queues.

- In the above example no process in the batch queue(4), for example, could run unless the queues for system processes(1) interactive processes(2), and interactive editing processes(3) were all empty.

- If an interactive editing process(3) entered the ready queue while a batch process(4) was running, the batch process would be preempted.

- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.



6.6.6 Multilevel Feedback Queue Scheduling:

→ The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues.

- The idea is to separate processes according to the characteristics of their CPU bursts.

- If a process uses too much CPU time, it will be moved to a lower-priority queue. This allows I/O-bound and interactive processes to be movednto the higher-priority queues.

- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

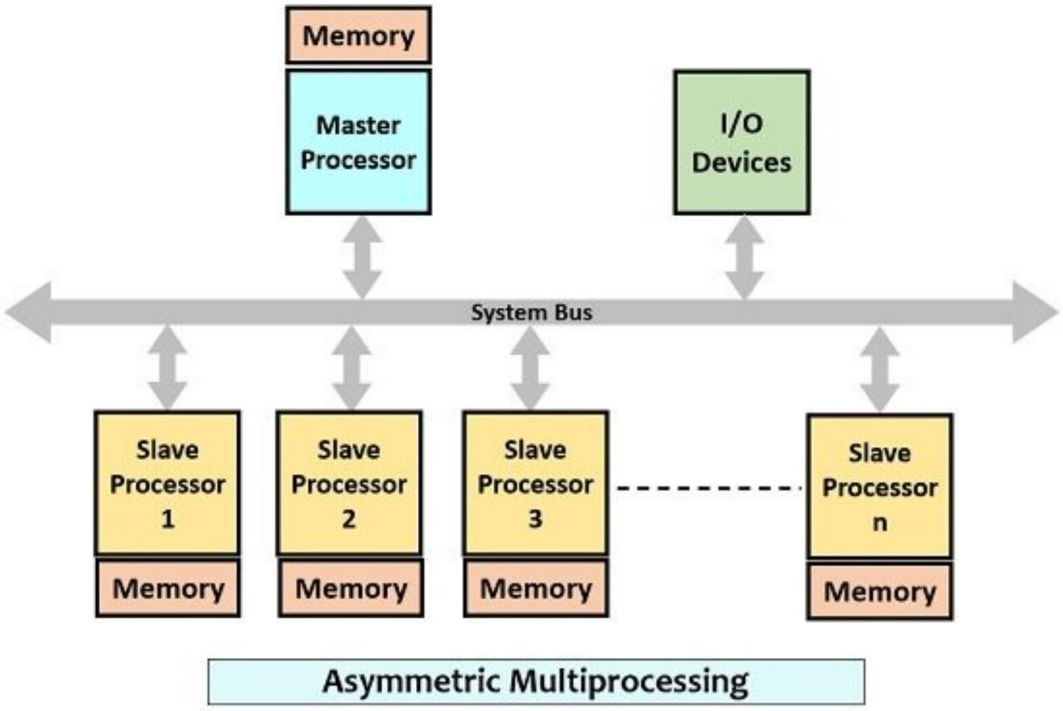
6.7 Multiple-Processor Scheduling:

➔ Approaches to Multiple-Processor Scheduling :

1. Assymetric Multiprocessing
2. Symmetric Multiprocessing

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor– the master server.

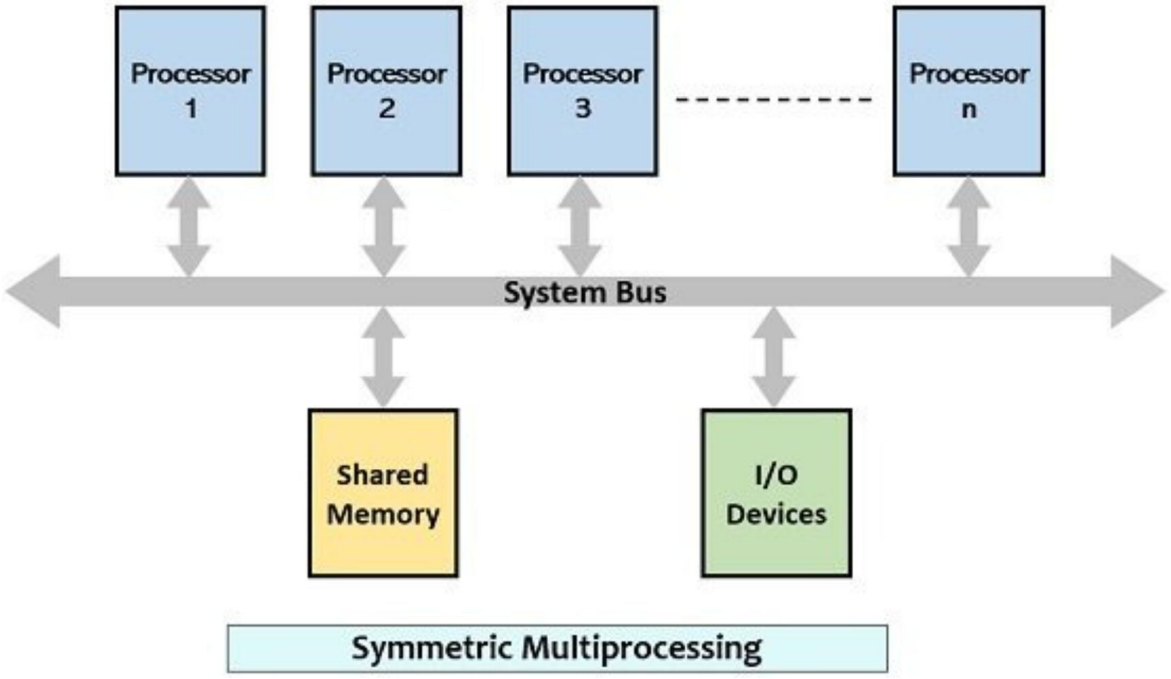
  The other processors execute only user code. This asymmetric multiprocessing



Asymmetric Multiprocessing

- A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

- The scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.



Symmetric Multiprocessing

6.8 Processor Affinity:

➔ Consider what happens to cache memory when a process has been running on a specific processor → The data most recently accessed by that process will be populated in the cache.

As a result, successive memory accesses by the process are often satisfied in cache memory.

- Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.

- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.

This is known as processor affinity that is, a process has an affinity for the processor on which it is currently running.

6.9 Interrupt Latency:

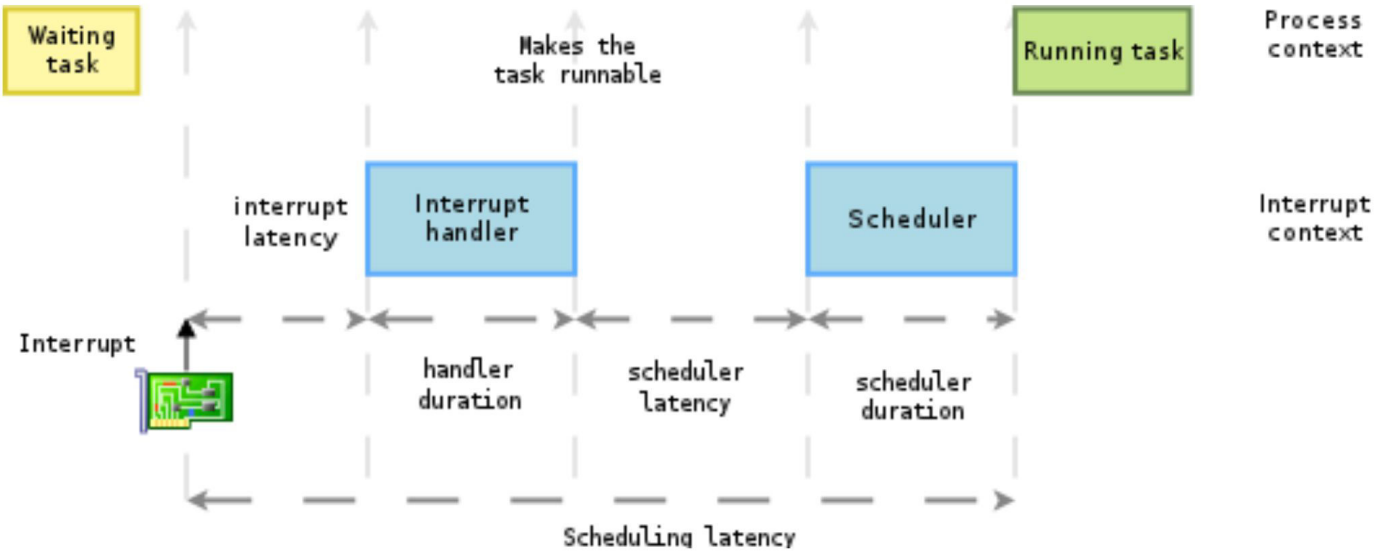
➔ Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services(resolves) the interrupt.

- When an interrupt occurs, the operating system must determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR).

- NOTE : An interrupt service routine (ISR) is a software routine that hardware invokes in response to an interrupt.

The total time required to perform these tasks is the interrupt latency.

- The amount of time required for the scheduling dispatcher to stop one process and start another is known as dispatch latency.



Chapter-7

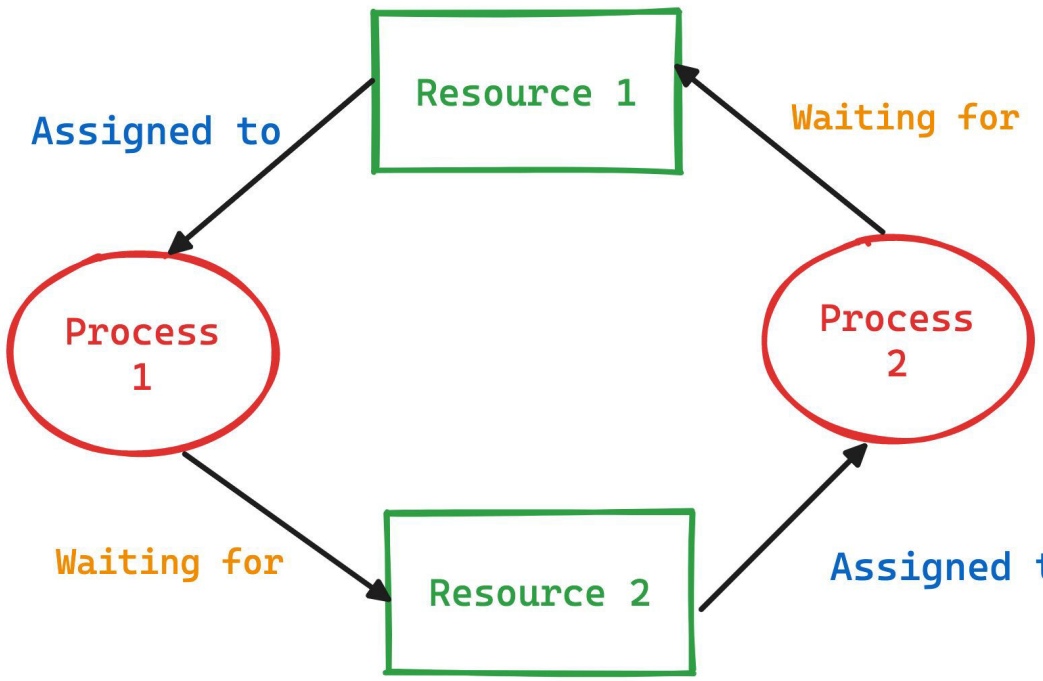
Deadlocks

➔ 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.

- The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

- A process must request a resource before using it and must release the resource after using it.

→ A process may request as many resources as it requires to carry out its designated task.

1. Request : The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release : The process releases the resource.

7.1 Deadlock Characterization:

➔ In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

7.7.1 Necessary Conditions

➔ A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion : At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait : A process must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption : Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait : A set { P0 , P1 , ... , Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn,and Pn is waiting for a resource held by P0.

7.2 Resource-Allocation Graph

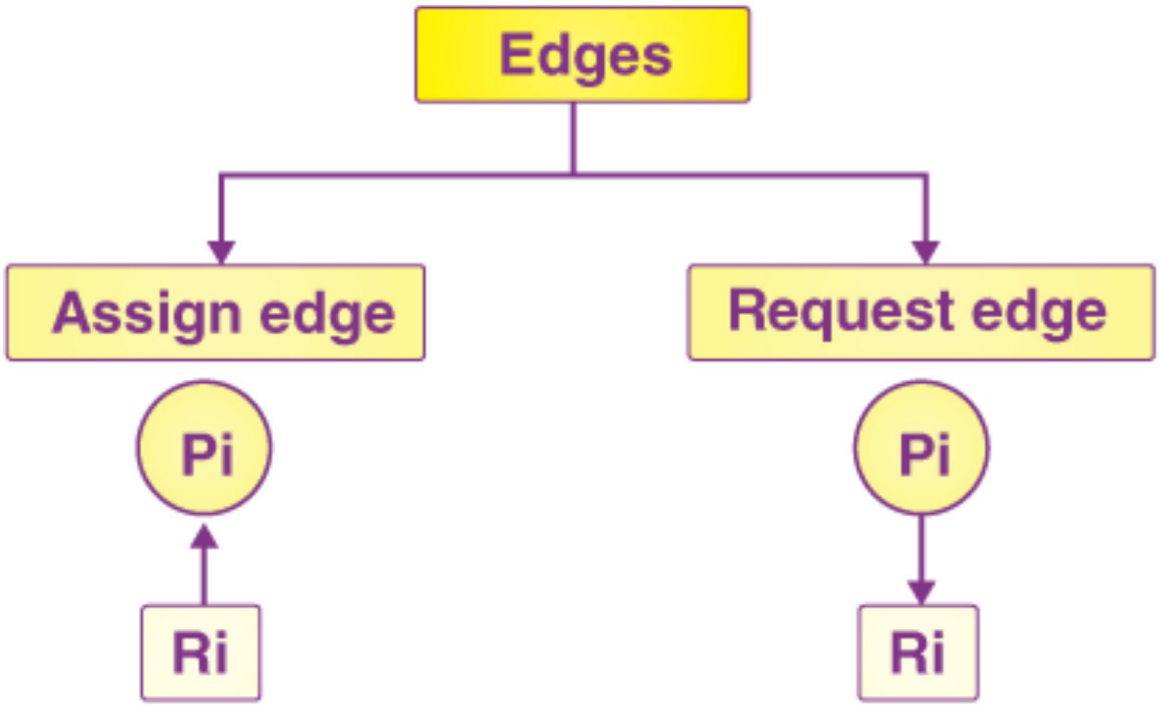
→ Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

- This graph consists of a set of vertices V and a set of edges E.

- The set of vertices V is partitioned into two different types of nodes:

  P = {P1, P2, ... , Pn}, the set consisting of all the active processes in the system.

  R = {R1, R2, ... , Rm}, the set consisting of all resource types.



→ A directed edge from process Pi to resource type Rj is denoted by Pi → Rj ; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.

- A directed edge Pi → Rj is called a request edge
  A directed edge Rj → Pi is called an assignment edge.

- When process Pi requests an instance of resource type Rj , a request edge is inserted in the resource-allocation graph.

- When the process no longer needs access to the resource, it releases the resource.

→ Given the definition of a resource-allocation graph,it can be shown that, if the graph contains no cycles,then no process in the system is deadlocked.

- If the graph does contain a cycle, then a deadlock may exist.

- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred

- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

7.3 Methods for Handling Deadlocks:

→ we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks,ensuring that the system will never enter a deadlocked state.

- We can allow the system to enter a dead locked state, detect it, and recover.

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

→ Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

→ Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

- With this additional knowledge, the operating system can decide for each request whether or not the process should wait.

→ To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

→ If a system does not employ either a deadlock-prevention or a deadlock- avoidance algorithm,The system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

7.4 Deadlock Prevention:

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

→ Mutual Exclusion:

- The mutual exclusion condition must hold.That is, atleast one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.

→ Hold and Wait:

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

- An alternative protocol allows a process to request resources only when it has none.

- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

→ No Preemption

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.

- In other words, these resources are implicitly released.

- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

→ Circular Wait

- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

7.5 Deadlock Avoidance:

→ We can avoid deadlock with some additional information about how resources will requested.

- For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.

- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

→ So with each request the system considers the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

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

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

→ The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.

- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.

- The request is granted only if the allocation leaves the system in a safe state.

7.5.1 Deadlock Detection:

→ If a system does not employ either a deadlock-prevention or a deadlock- avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock

7.5.2 Recovery from Deadlock:

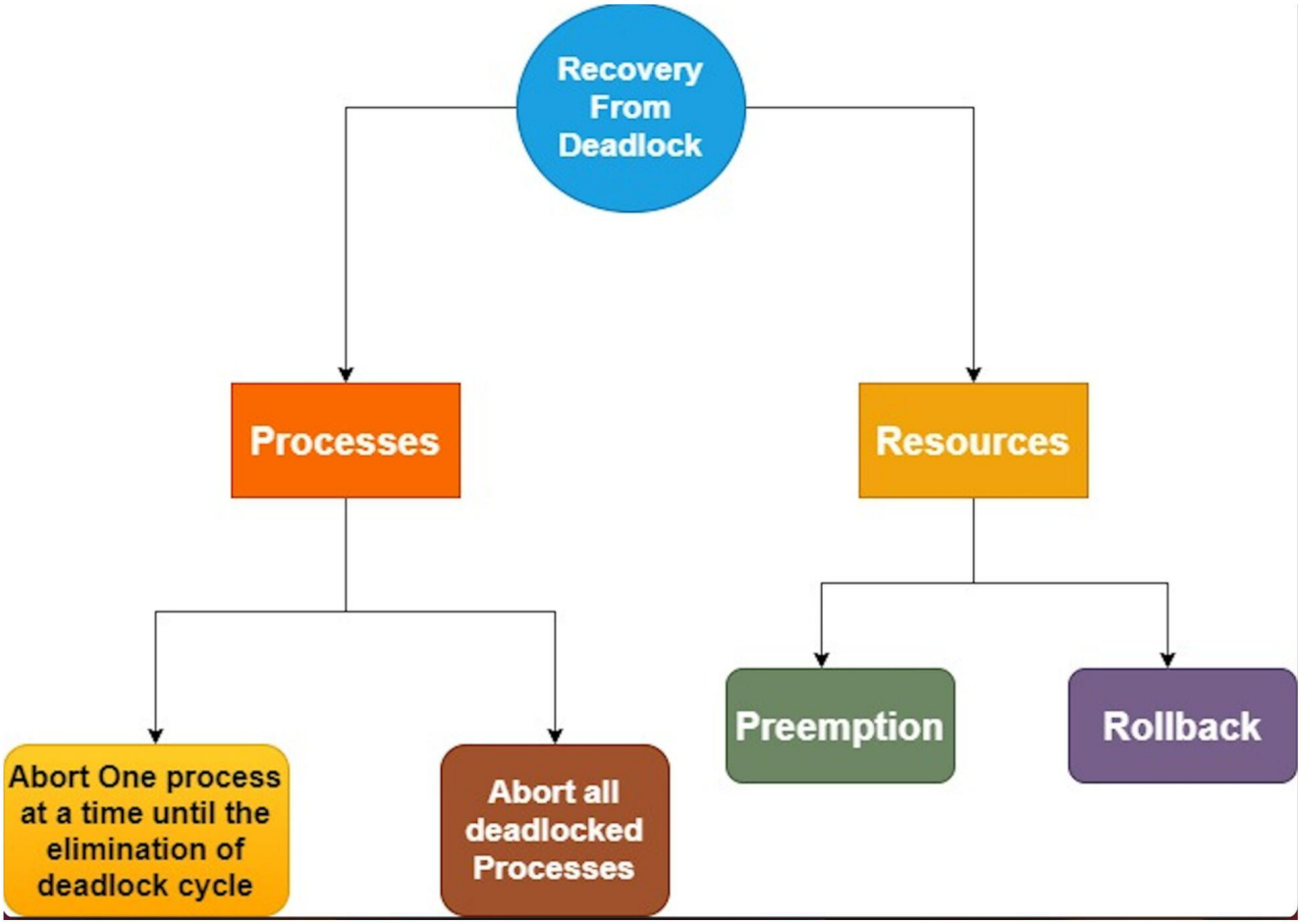
→ When a detection algorithm determines that a deadlock exists, several alternatives are available:

- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

  Another possibility is to let the system recover from the deadlock automatically.

➔ There are two options for breaking a deadlock:

- One is simply to abort one or more processes to break the circular wait.
- The other is to preempt some resources from one or more of the deadlocked processes.



7.5.3 Process Termination:

- Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

→ Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated we should abort those processes whose termination will incur the minimum cost.

→ Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

7.5.4 Resource Preemption:

→ To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

1. Selecting a victim : Which resources and which processes are to be preempted?

   As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback : If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource.

   We must roll back the process to some safe state and restart it from that state.

3. Starvation : How do we ensure that starvation will not occur?

   That is, how can we guarantee that resources will not always be preempted from the same process?

- In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task.

  Clearly,we must ensure that a process can be picked as a victim only a (small) finite number of times.

  The most common solution is to include the number of rollbacks in the cost factor.

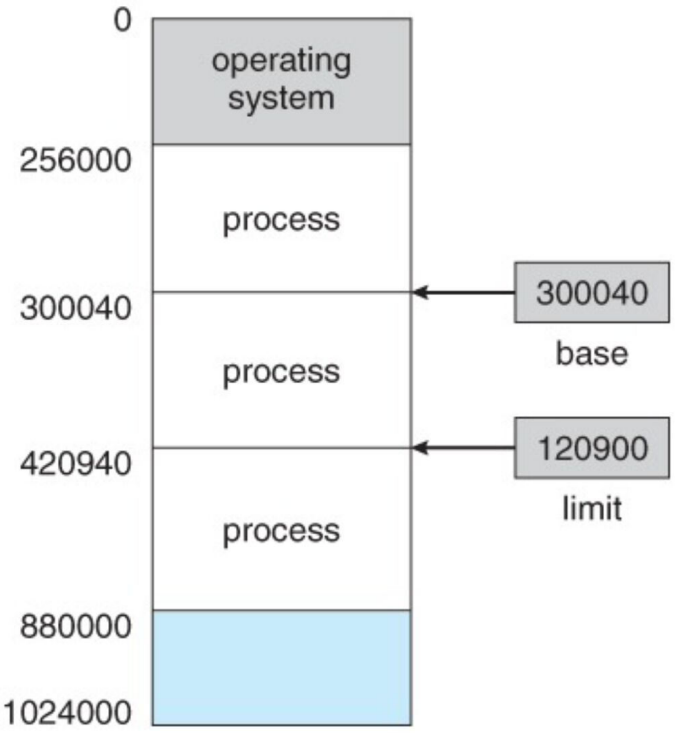
Chapter-8

Main Memory

→ Each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

- The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.



→ For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

- Any attempt by a program executing in user mode to access operating-system memory or other users' memory is treated as a fatal error.

- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

8.1 Logical Versus Physical Address Space:

→ An address generated by the CPU is commonly referred to as a logical address(virtual address).

- An address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address.

- The set of all logical addresses generated by a program is a logical address space.

→ The set of all physical addresses corresponding to these logical addresses is a physical address space.

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

8.2 Dynamic Loading:

→ With dynamic loading, a routine is not loaded until it is called.

- All routines are kept on disk. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the loader is called to load the desired routine into memory.
- Then control is passed to the newly loaded routine. The advantage of dynamic loading is that a routine is loaded only when it is needed.

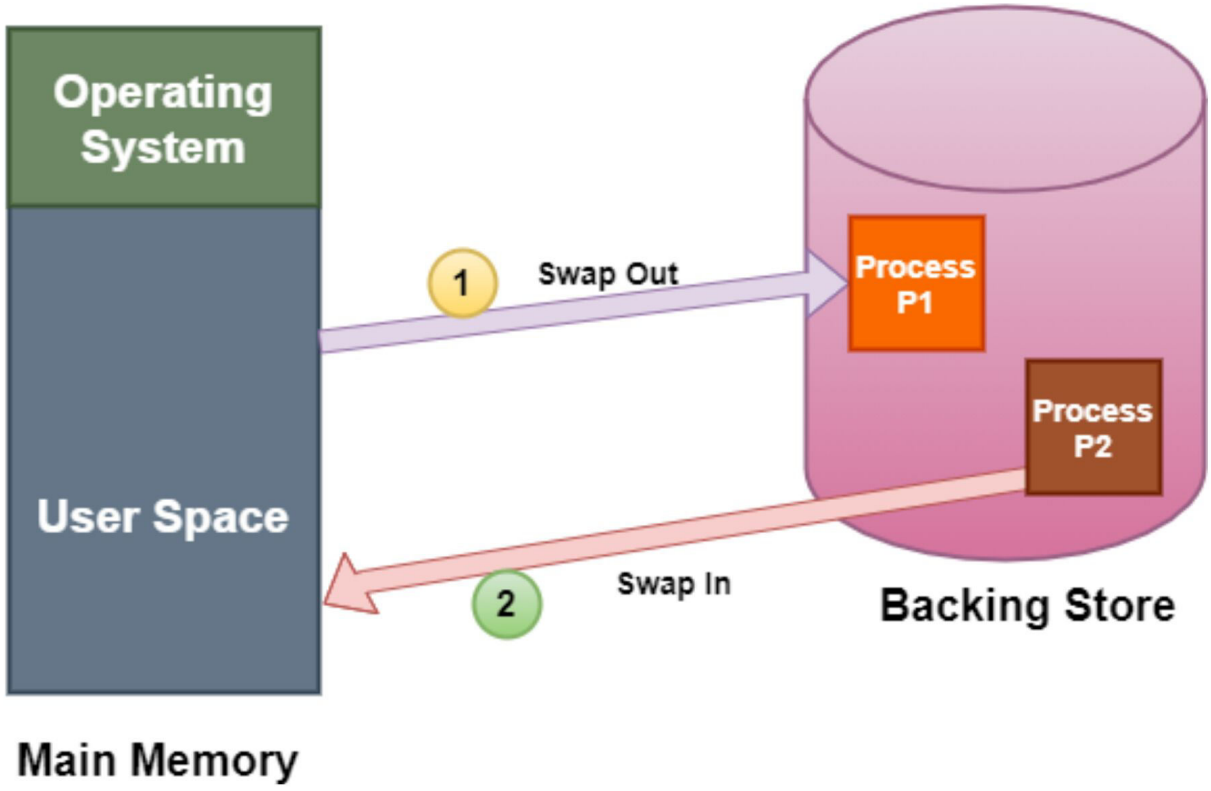
8.3 Swapping:

→ A process must be in memory to be executed.

- A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.

- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

- Standard swapping involves moving processes between main memory and a backing store.



→ The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.

- The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

- NOTE : The context-switch time in such a swapping system is fairly high.

8.4 Contiguous Memory Allocation:

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

- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.

→ In this multiple - partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

- When the process terminates, the partition becomes available for another process.

→ The operating system keeps a table indicating which parts of memory are available and which are occupied.

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

→ As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.

- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

→ Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied- that is, no available block of memory (or hole) is large enough to hold that process.

- The operating system can then wait until a large enough block is available.

- When a process terminates, it releases its block of memory which is then placed back in the set of holes.

- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

- Dynamic storage-allocation concerns how to satisfy a request of size 'n' from a list of free holes.

→ Three solutions to this are:

1. First fit : Allocate the first hole that is big enough.

2. Best fit : Allocate the smallest hole that is big enough.

3. Worst fit : Allocate the largest hole.

8.5 Fragmentation:

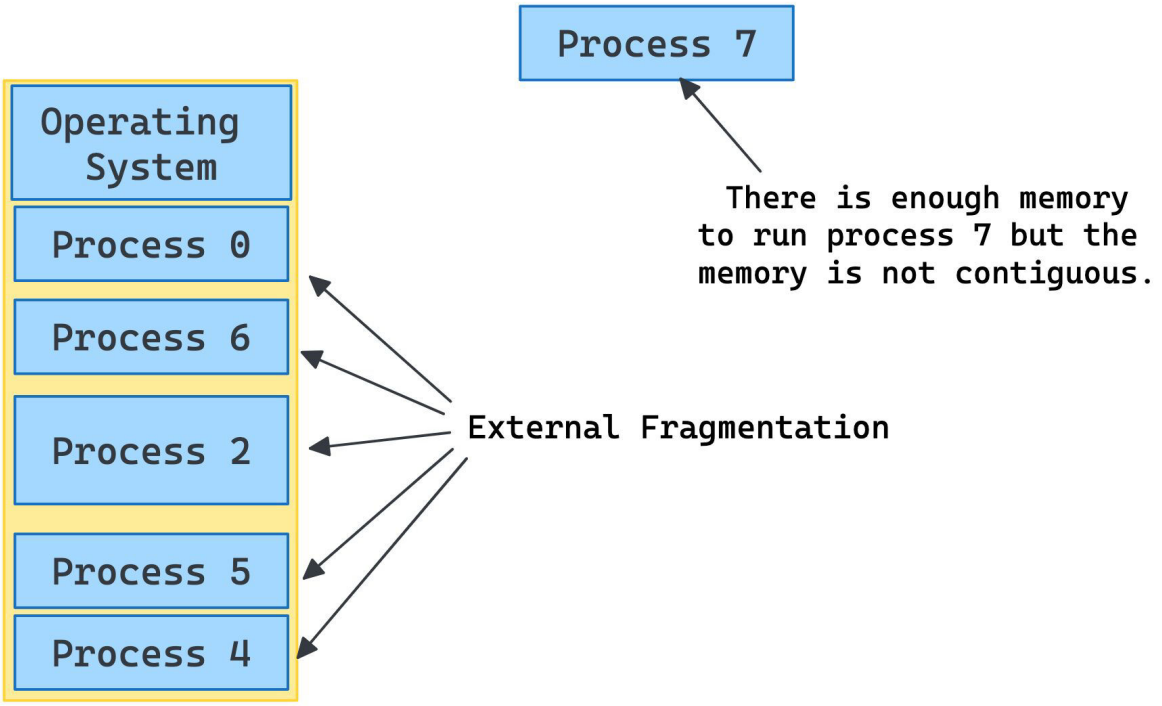
➔ There are two types of memory fragmentation :

Internal Fragmentation

External Fragmentation

8.5.1 External Fragmentation:

➔ External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.



- In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

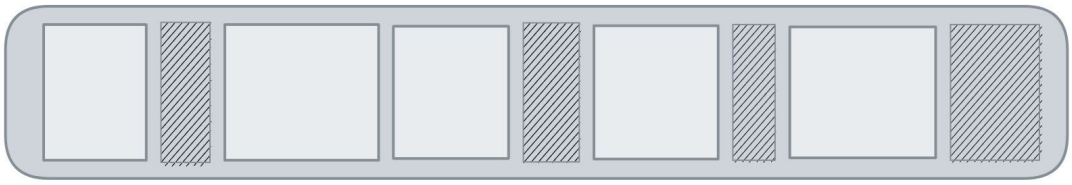
- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.

→ One solution to the problem of external fragmentation is compaction.

- Compaction: The goal is to shuffle the memory contents so as to place all free memory together in one large block.

  However compaction is not always possible.

Fragmented Memory Before Compaction



Memory After Compaction

- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

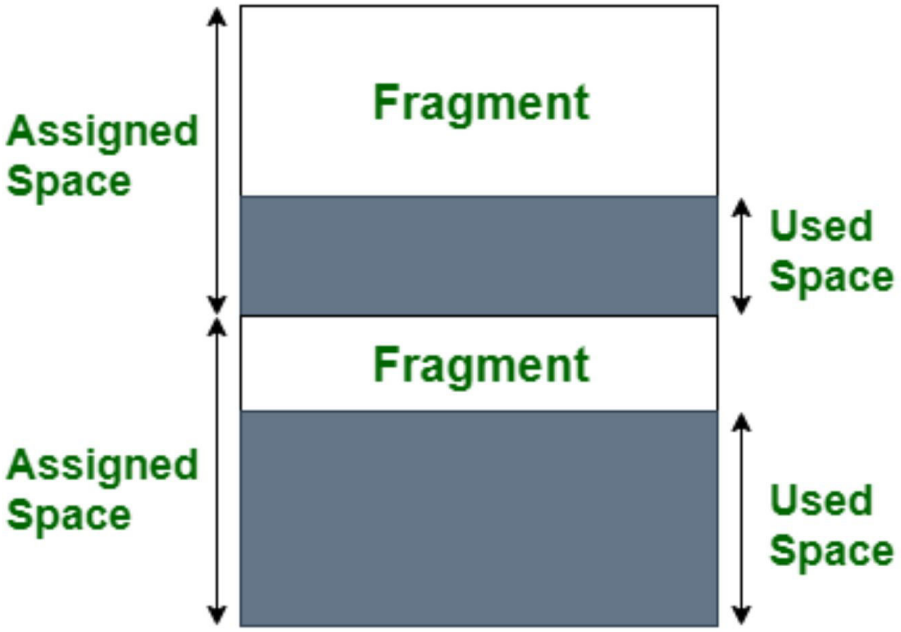
- Two complementary techniques achieve this solution: segmentation and paging.

8.5.2 Internal Fragmentation:

→ Internal fragmentation typically occurs in systems that use fixed-size memory allocation, such as partitioning memory into fixed-sized blocks or pages.

- When a process or application requests a block of memory,it is allocated a fixed-sized block, even if the amount of data it needs to store is smaller than the allocated block's size.

- The unused space within the block is then considered internal fragmentation.



Internal Fragmentation

8.6 Segmentation:

→ Segmentation is a memory management technique used in operating systems to divide a process's logical address space into segments.

- Each segment represents a logical unit of the process, such as code, data, stack, or heap.

- Segmentation allows for a flexible and efficient memory allocation scheme, as different segments can have different sizes and can grow or shrink dynamically.

- When a program is executed, the operating system maps the logical addresses used by the program to their corresponding physical addresses in memory.

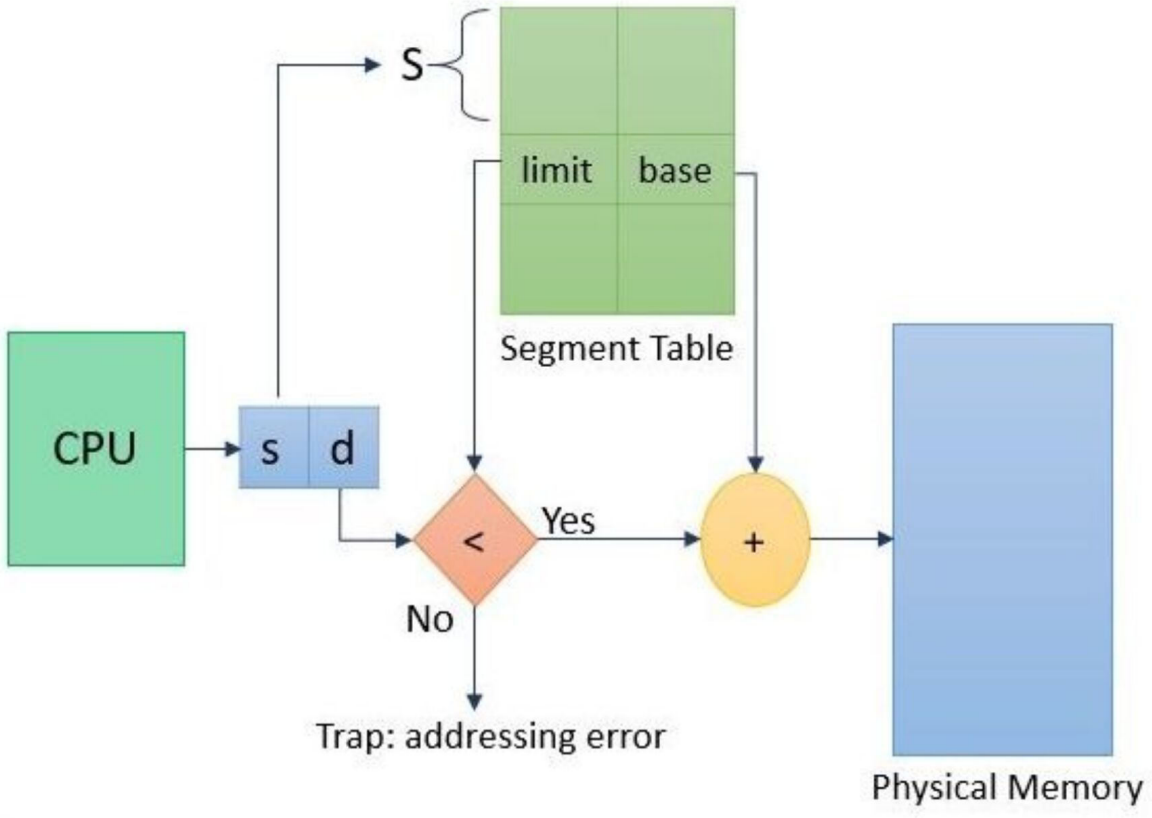
➔ When a program is executed, the operating system maps the logical addresses used by the program to their corresponding physical addresses in memory.

- This mapping is typically done through the use of segment tables.

Thus, a logical address consists of a two tuple:

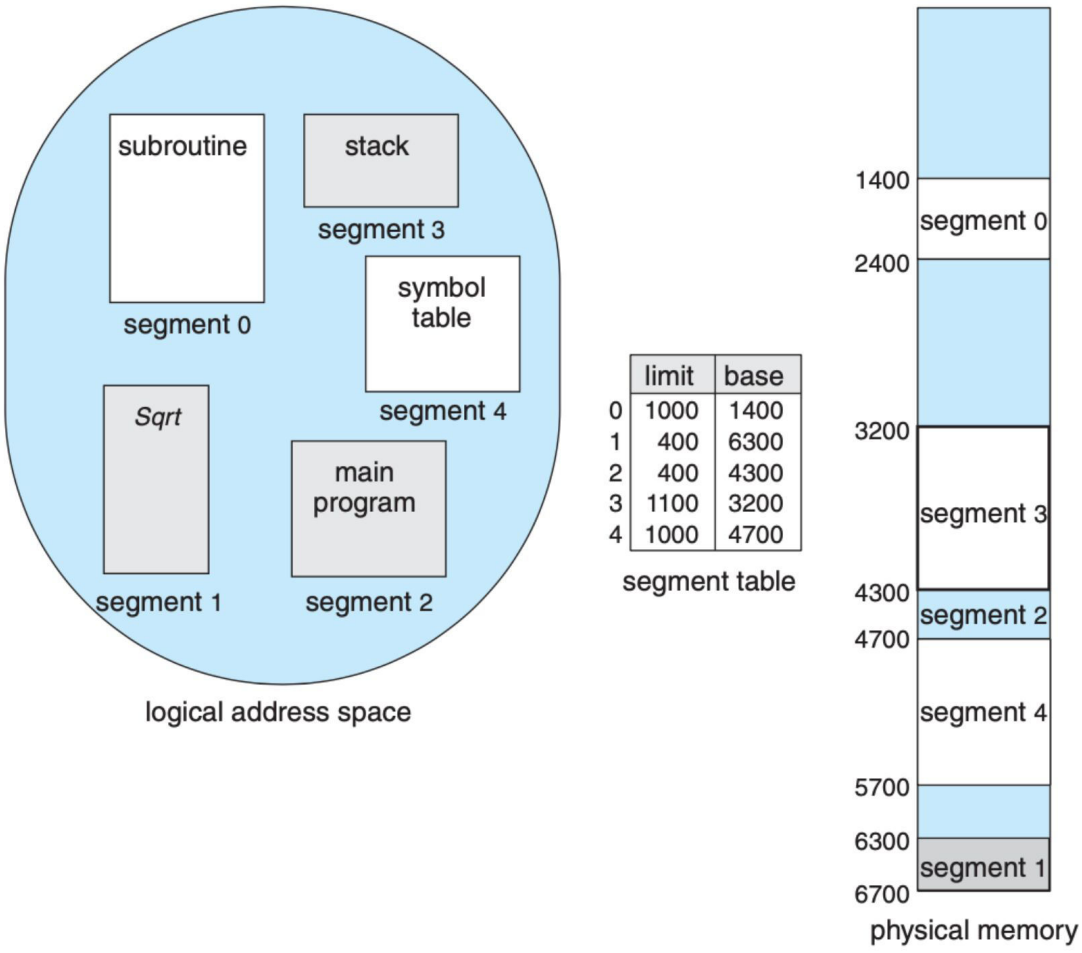
<segment-number(s), offset(d)>

➔ 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.



→ Fragmentation: Segmentation can lead to external fragmentation, as memory becomes divided into smaller segments.

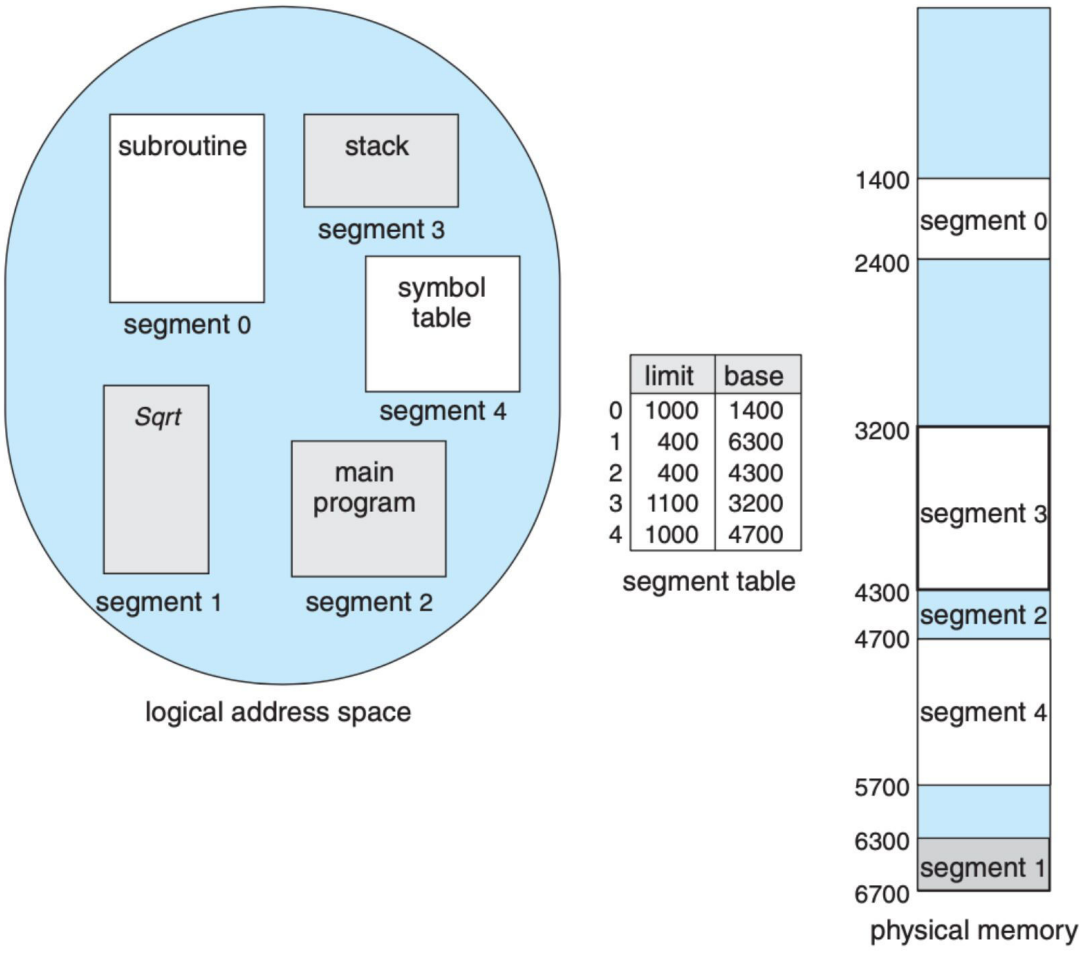
This can lead to wasted memory and decreased performance.



- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

→ Fragmentation: Segmentation can lead to external fragmentation, as memory becomes divided into smaller segments.

This can lead to wasted memory and decreased performance.



- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

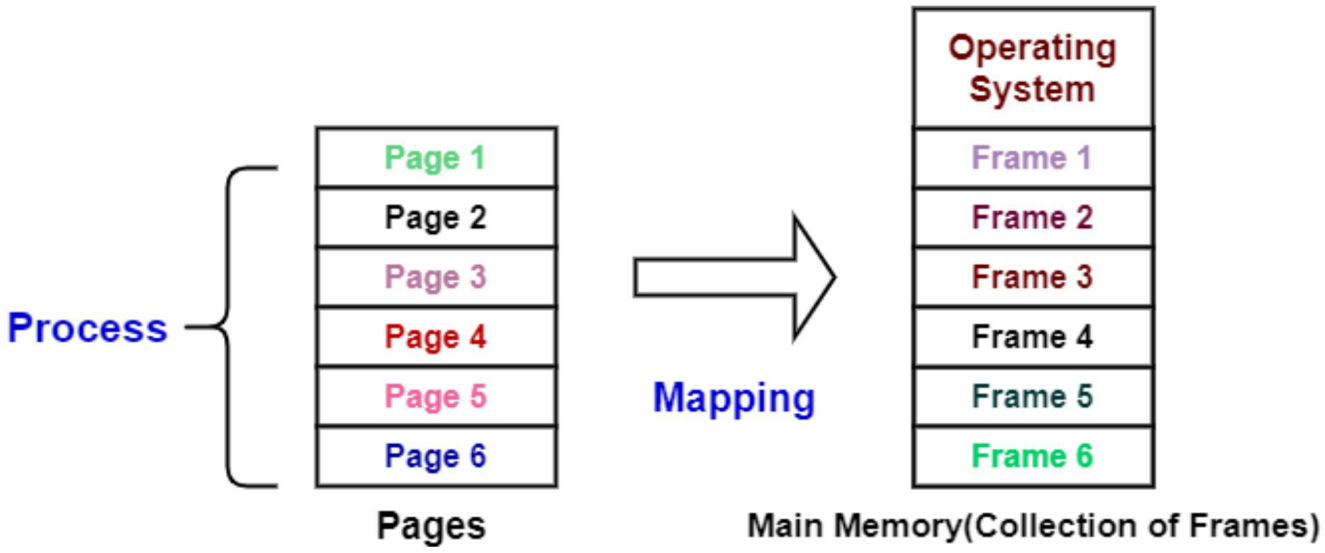
8.7 Paging:

→ Paging is a memory management technique used in operating systems to handle the virtual memory of a computer system.

- The logical memory (address space) used by a process to be divided into fixed-size blocks called pages.

- It allows the physical memory (RAM) to be divided into fixed-size blocks called frames of size same as pages.

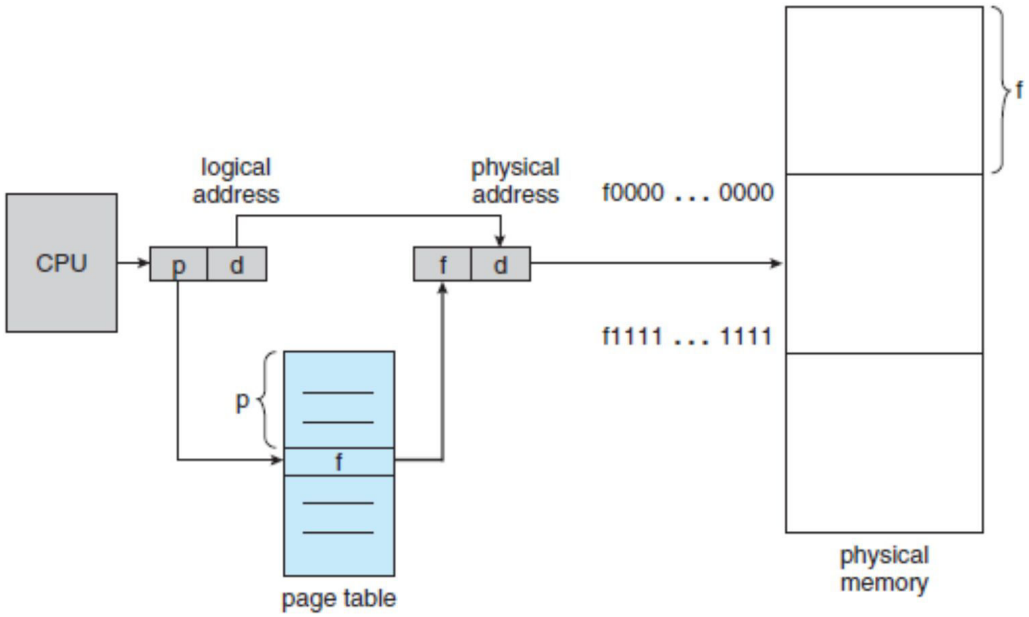
The size of a page and a frame is typically a power of two, such as 4KB or 8KB.



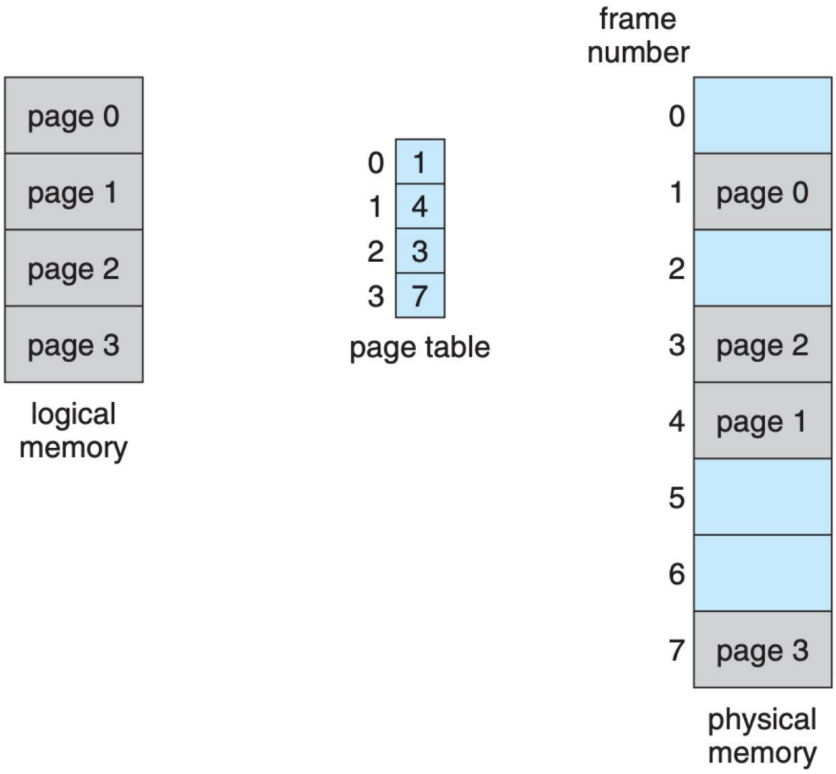
- When a process is to be executed, its pages are loaded into any available memory frames from their source.

- When a process generates a memory reference, the logical address consists of two parts: a page number(p) and an offset(d) within the page.

- The page number is used to index a page table, which is a data structure maintained by the operating system to keep track of the mapping between logical pages and physical frames.



- The page table contains the base address of each page in physical memory. This base address is combined with the page offset which points to the physical memory.



Chapter-9

Virtual Memory

9.1 Virtual Memory:

→ Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

- This technique frees programmers from the concerns of memory-storage limitations.

- The ability to execute a program that is only partially in memory would confer many benefits:

  - A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.

  - Because each user program could take less physical memory more programs could be run at the same time.

  - Less I/O would be needed to load or swap user programs into memory,so each user program would run faster.

- Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.

9.2 Demand Paging:

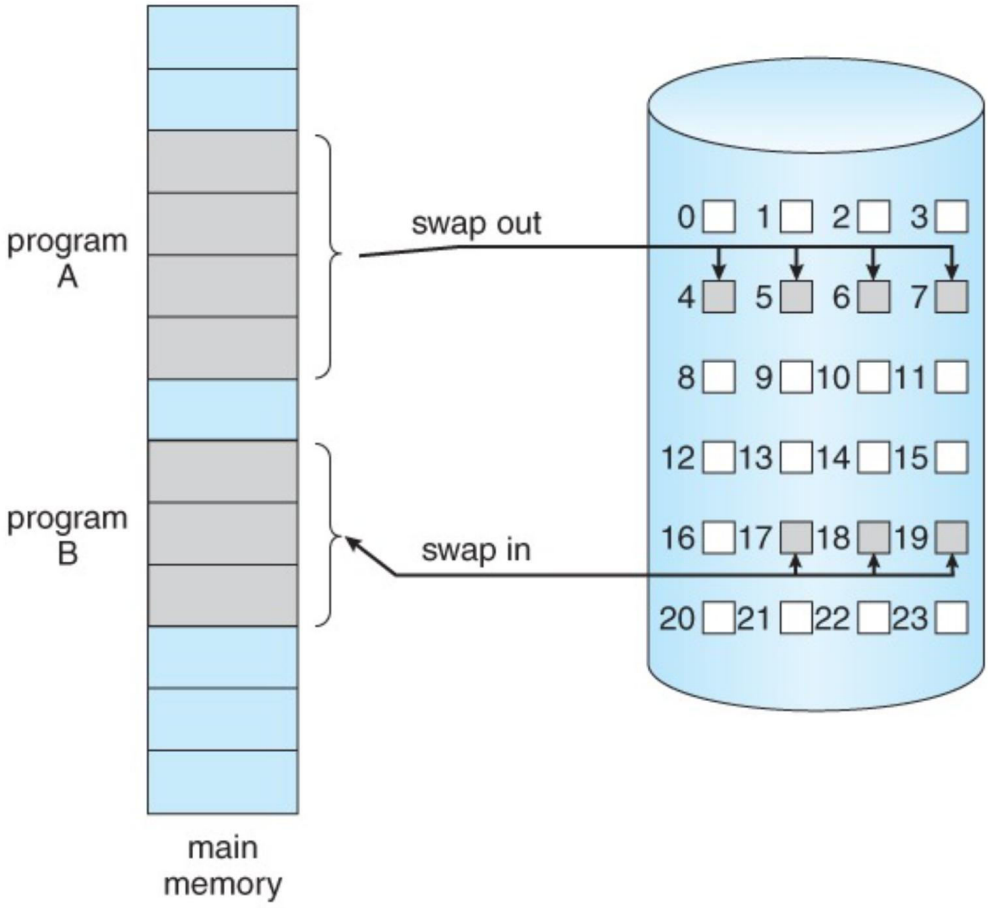
→ Demand paging technique allows the operating system to bring in only the necessary pages of a process into memory rather than loading the entire process into RAM at once.

- This approach optimizes memory usage by keeping only the actively used pages in physical memory,while the remaining pages are stored in secondary storage.

- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
Pages that are never accessed are thus never loaded into physical memory.

- A demand-paging system is similar to a paging system with swapping.

- Where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory though, we use a lazy swapper. A lazy swapper(pager) never swaps a page into memory unless that page will be needed.

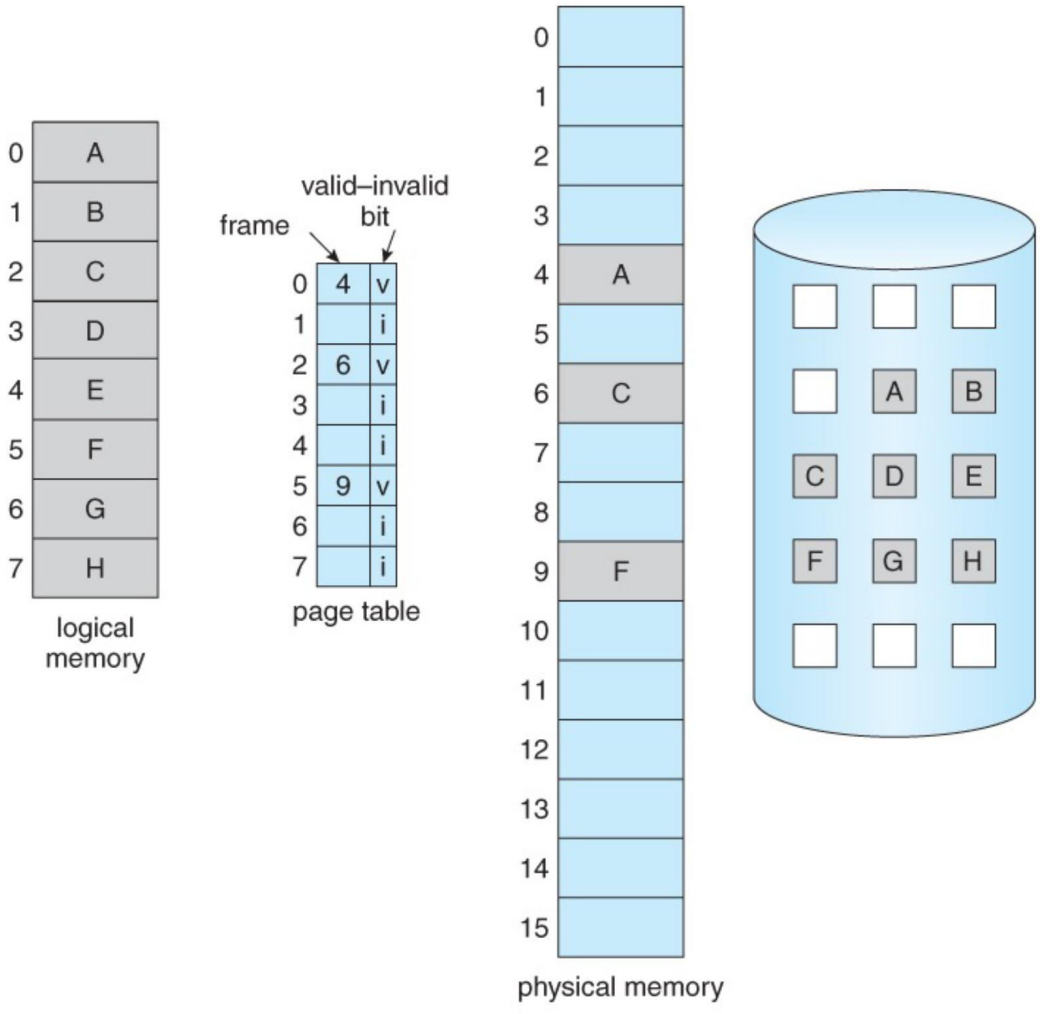


- Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

- With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. One approach is valid-invalid scheme.

Valid-Invalid Bit Scheme

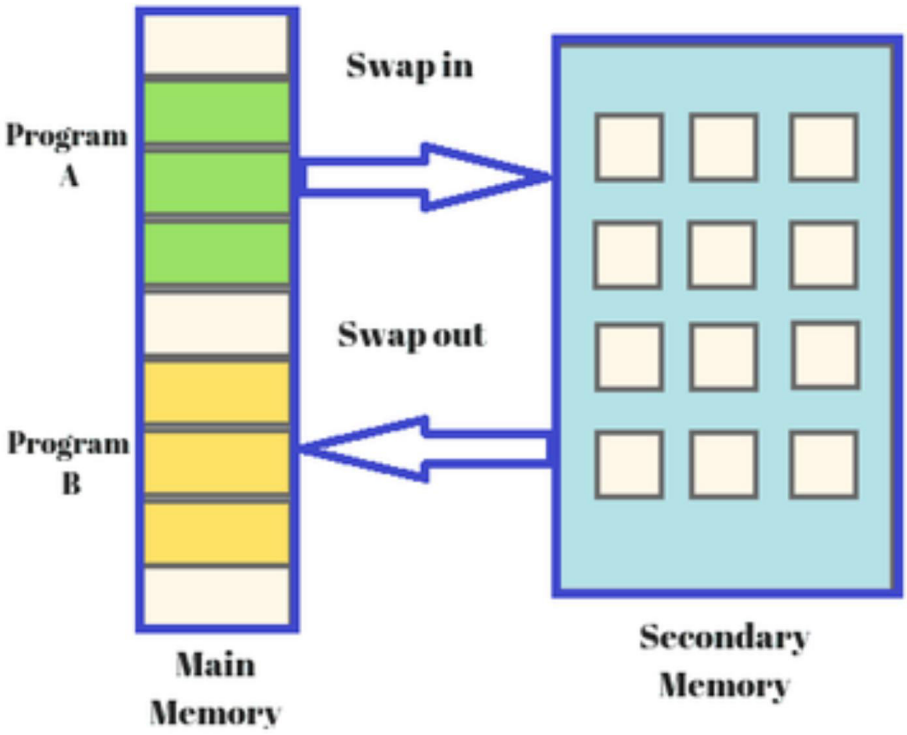
→ When this bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.



8.7.2 Page Fault:

→ A page fault is an event that occurs when a program or process attempts to access a page of virtual memory that is currently not present in the physical memory (RAM).

- When a page fault happens, the operating system needs to handle it by retrieving the required page from secondary storage (typically the hard disk) and loading it into a free frame in the physical memory.This process is known as page swapping.



- The process executes and accesses pages that are memory resident, execution proceeds normally.

- What happens if the process tries to access a page that was not brought into memory?

- Access to a page marked invalid causes a page fault. The paging hardware, will notice that the invalid bit is set in the page table, causinga trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

Procedure for handling Page fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

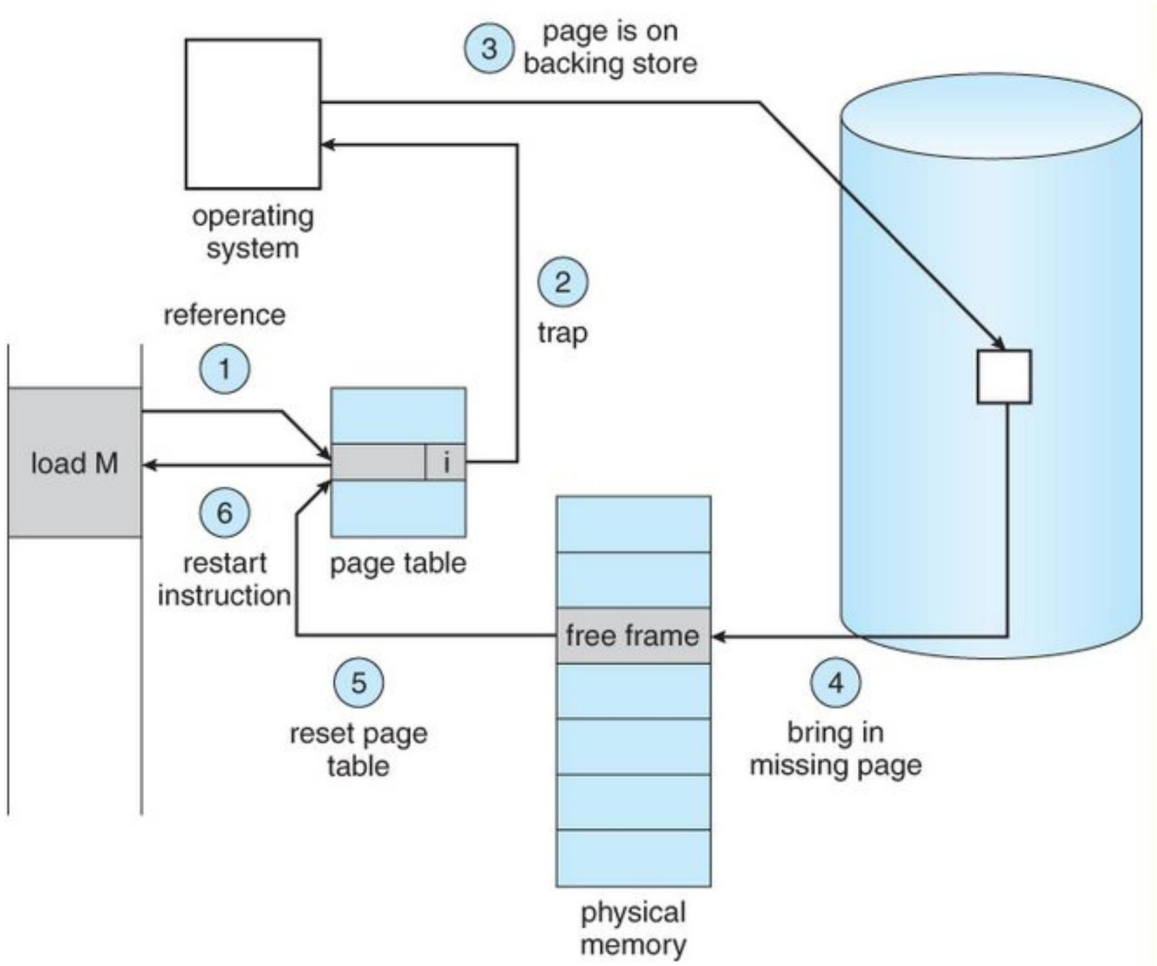
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame.

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table and page table to indicate that the page is now in memory.

6. 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.



- There could be situations some programs could access several new pages of memory possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance.

Fortunately, these behavior is exceedingly unlikely. Programs tend to have locality of reference.

- The hardware to support demand paging is the same as the hardware for paging and swapping:
  - Page table : This table has the ability to mark an entry invalid through a valid-invalid bit.
  - Secondary memory : This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk.

- A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible.

- A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again.

→ Example, consider a three-address instruction such as ADD the content of A to B, placing the result in C.

→ These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

- If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction.

→ We are faced with three major components of the page-fault service time:

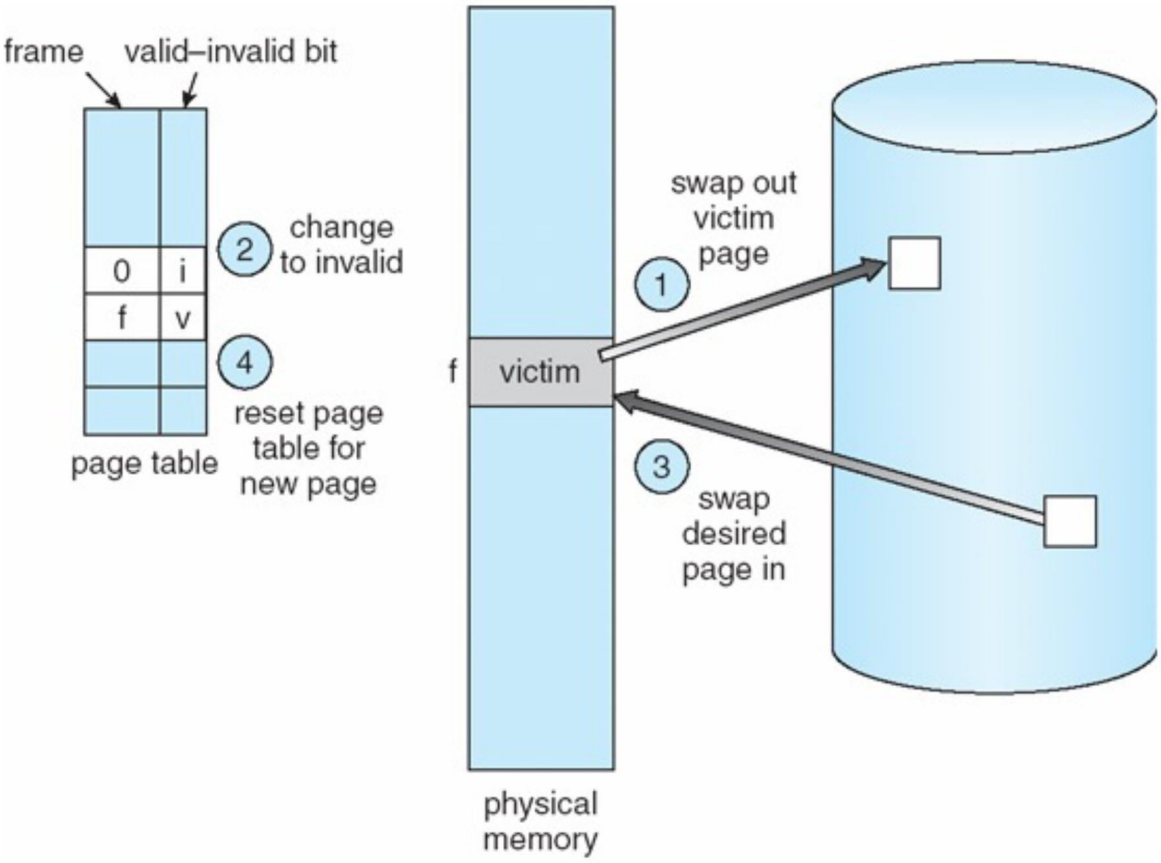
1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

9.3 Page Replacement:

→ When a program needs to access data or instructions that are not currently present in physical memory (RAM), a page fault occurs.

- Page replacement algorithms are employed to determine which pages should be evicted from memory to make room for the required page.

→ NOTE: Each page faults at most once, when it is first referenced.



- Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it.

- We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted.

Steps for Page replacement:

1. Find the location of the desired page on the disk.

2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.

4. Continue the user process from where the page fault occurred.

- If no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

- We can reduce this overhead by using a modify bit (or dirty bit).

- When this scheme is used, each page or frame has a modify bit associated with it in the hardware.

- The modify bit for a page is set whenever any byte in the page is written into, indicating that the page has been modified.

- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified. In this case, we must write the page to the disk.

- If the modify bit is not set,the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk thus, they may be discarded when desired.

- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

- We must develop a frame-allocation algorithm and a page-replacement algorithm. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.

9.3.1.2 Belady's Anamoly:

➔ Belady's anomaly, is a phenomenon that refers to the counterintuitive situation where increasing the number of page frames in memory for a process can result in an increase in the number of page faults (i.e., instances where a required page is not found in memory).

- We would expect that giving more memory to a process would improve its performance.

- The anomaly arises due to the peculiarities of the memory access patterns.

- In some cases, the page that will be accessed may have been evicted earlier by the FIFO algorithm, even though there were available free page frames.

- This results in a higher number of page faults as compared to the case when there are fewer page frames.

9.3.2 Optimal Page Replacement:

Replace the page that will not be used for the longest period of time.

9.3.1 FIFO Page Replacement:

→ The simplest page-replacement algorithm is a first-in, first-out (FIFO) 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.

- We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue.
  When a page is brought into memory, we insert it at the tail of the queue.

- So if we select a page for replacement that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page.

- Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution.

- By always making the optimal eviction decision, the optimal algorithm achieves the minimum possible number of page faults for a given memory reference sequence.

- However, the key limitation of the optimal algorithm is that it requires knowledge of future memory references, which is not practically obtainable in real-world scenarios.

9.3.3 LRU Page Replacement:

→ LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

→ Here's how the LRU page replacement algorithm works:

- Each page in memory is associated with a timestamp or a counter that indicates when it was last accessed or referenced.

- When a page fault occurs and there are no free page frames available in memory, the algorithm selects the page with the oldest timestamp or the lowest counter value for eviction.

- The newly requested page is then loaded into the freed page frame, and its timestamp or counter is updated to reflect the current time or access.

- The LRU algorithm aims to prioritize retaining pages that have been accessed recently.

- The assumption is that recently accessed pages are more likely to be accessed again in the near future due to the principle of locality.

- The problem is to determine an order for the frames defined by the time of last use.

→ Two implementations are feasible for LRU include:

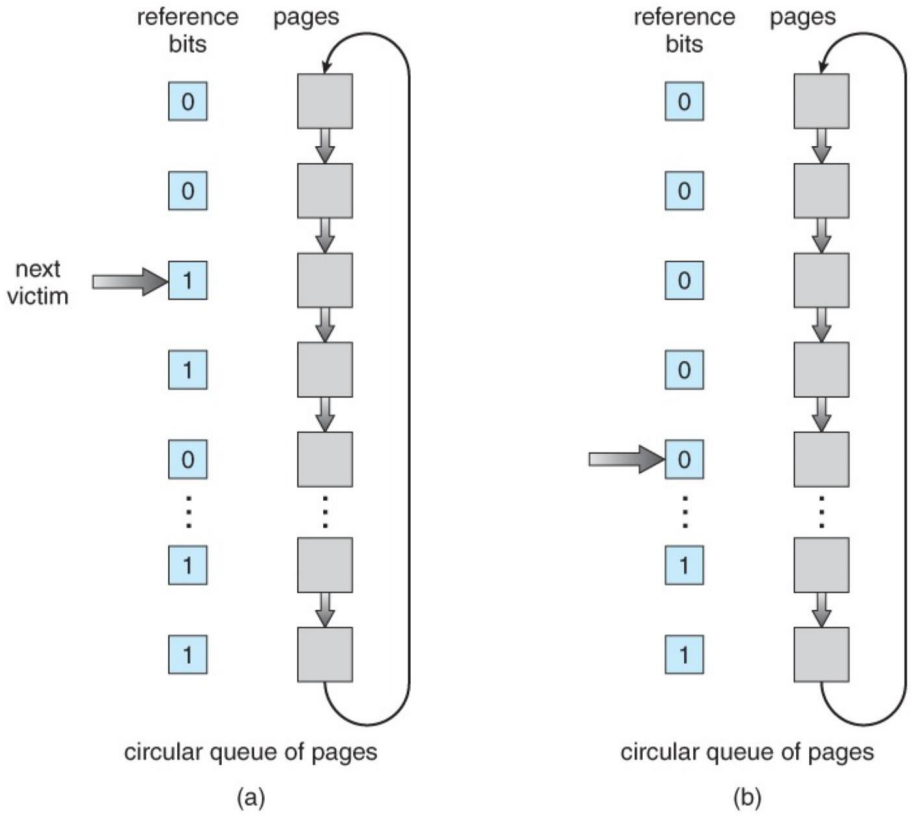
- Counters: We associate with each page-table entry a time-of-use field .This clock is incremented for every memory reference.

  Whenever a reference to a page is made, the time-of-use field is updated. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.

- Stack: Pages are placed in a stack. When a page is accessed, it is moved to the top of the stack. The page at the bottom of the stack is considered the least recently used and is evicted when a page fault occurs.

9.3.4 Second Chance Algorithm:

→ Second chance algorithm modification of the LRU (Least Recently Used) algorithm's behavior while reducing the overhead associated with maintaining timestamps or counters for each page.



→ Second-Chance algorithm:

- Each page in memory is associated with a reference bit, typically stored in the page table entry or a separate data structure.

  The reference bit indicates whether the page has been recently accessed or not.

- When a page fault occurs and there are no free page frames available in memory, the Second-Chance algorithm examines the reference bit of the pages in a circular list or queue.

- It starts scanning the list from a specific position (e.g., the head of the list) and checks the reference bit of each page encountered.

- If the reference bit is set (indicating the page has been recently accessed), the algorithm clears the reference bit and moves to the next page in the list.

- If the reference bit is not set (indicating the page has not been accessed recently), the algorithm selects that page for eviction.

- The newly requested page is then loaded into the freed page frame, and its reference bit is set.

- So in this algorithm pages that have been recently accessed are given another opportunity to stay in memory, while pages that have not been accessed recently are more likely to be evicted.

9.3.5 Least Frequently Used:

→ We can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- We use a counter - Each page has a counter associated with it. Whenever a page is accessed, its counter is incremented. During a page fault, the page with the lowest counter value is selected for eviction.

- The least frequently used(LFU)page - replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

- A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

9.4 Thrashing:

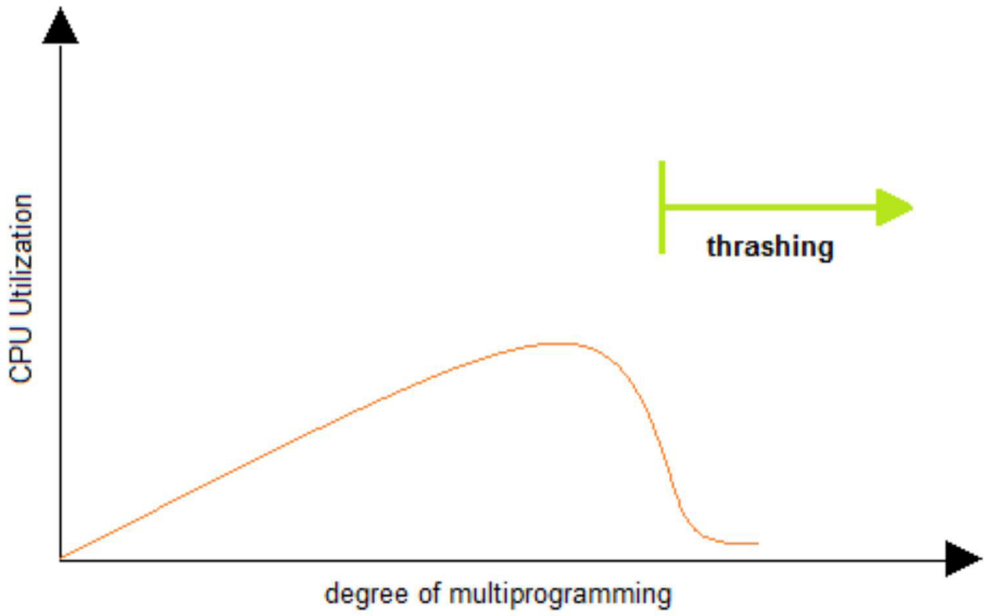
→ If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.

- At this point, it must replace some active page. However, if all its pages are in active use, it must replace a page that will be needed again right away.

- Consequently, it quickly faults again, and again, replacing pages that it must bring back in immediately.

- This high paging activity is called thrashing.

→ Note : A process is thrashing if it is spending more time paging than executing.



- So thrashing in operating systems refers to a situation where a system is spending a significant amount of time and resources continuously swapping pages in and out of main memory, rather than executing useful work.

- It occurs when the system is overwhelmed with excessive paging activity and is unable to allocate enough physical memory to satisfy the demands of active processes.

Chapter-10

File-System Interface

10.1 File Concept:

→ A file system is a method an operating system uses to store, organize, and manage files and directories on a storage device.

- The storage media include magnetic disks, magnetic tapes, and optical disks etc.

- These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

→ A file has a certain defined structure, which depends on its type:

- A text file is a sequence of characters organized into lines.

- A source file is a sequence of functions, each of which is further has a function declarations followed by executable statements.

- An executable file is a series of code sections that the loader can bring into memory and execute.

10.2 File Attributes:

- Name : The symbolic file name with which the user can access the file.
- Identifier : This unique tag,usually a number,identifies the file within the file system; it is the non-human-readable name for the file.
- Type : This information is needed for systems that support different types of files.
- Location : This information is a pointer to a device and to the location of the file on that device.
- Size : The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- Protection : Access-control information determines who can do reading, writing, executing, and so on.
- Time, date, and user identification : This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

10.3 File Operations:

→ The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- Creating a file : Two steps are necessary to create a file.

  First, space in the file system must be found for the file.
  Second, an entry for the new file must be made in the directory.

- Writing a file : To write a file, we make a system call specifying both the name of the file and the information to be written to the file.

  Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- Reading a file : To read from a file, we use a system call that specifies the name of the file the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.

  Once the read has taken place, the read pointer is updated.Because a process is usually either reading from or writing to a file.

- Repositioning with in a file : The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.

  This file operation is also known as a file seek.

- Deleting a file : To delete a file,we search the directory for the named file.
  Having found the associated directory entry, we release all file space, sothat it can be reused by other files, and erase the directory entry.

- Truncating a file : The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged – except for file length–but lets the file be reset to length zero and its file space released.

- Other common operations include appending new information to the end of an existing file and renaming an existing file.

- Most of the file operations mentioned involve searching the directory for the entry associated with the named file.

- To avoid this constant searching, many systems require that an open() system call be made before a file is first used.

- The operating system keeps a table, called the open-file table, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

- When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.

10.4 File Types:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

10.5 Access Methods:

→ The information in the file can be accessed in several ways:

Sequential Access:

→ The simplest access method is sequential access. Information in the file is processed in order, one record after the other.

- A read operation – read-next() : reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- Similarly, the write operation– write-next() : appends to the end of the file and advances to the end of the newly written material (the new end of file).
- Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n—perhaps only for n = 1.

Direct Access(Relative Access):

→ Here, a file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

- For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

- Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives,we compute which block contains the answer and then read that block directly to provide the desired information.

- For the direct-access method, the file operations must be modified to include the block number as a parameter.

  Thus,we have read(n), where n is the block number, rather than read next(), and write(n) rather than write next().

10.6 Directory Structure:

10.6.1 Directory Overview:

→ The directory can be viewed as a symbol table that translates file names into their directory entries.

➔ The operations that are to be performed on a directory:

- Search for a file : We need to be able to search a directory structure to find the entry for a particular file.Related files can be stored together within a directory.

- Create a file : New files need to be created and added to the directory.

- Delete a file : When a file is no longer needed,we want to be able to remove it from the directory.

- List a directory : We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

- Rename a file : Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.

- Traverse the file system : We may wish to access every directory and every file within a directory structure.

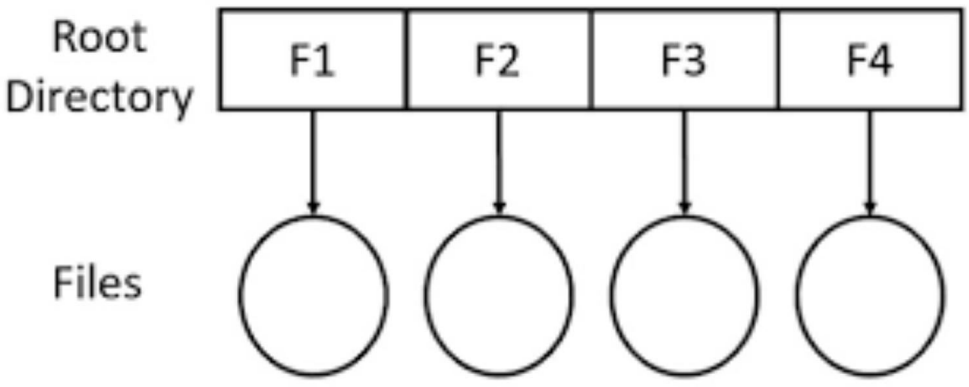
10.6.2 Directory Structure

10.6.2.1 Single-Level Directory:

➔ The simplest directory structure is the single-level directory All files are contained in the same directory, which is easy to support and understand.

➔ A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

- If two users call their data file test.txt, then the unique-name rule is violated.



Single-level Directory Structure

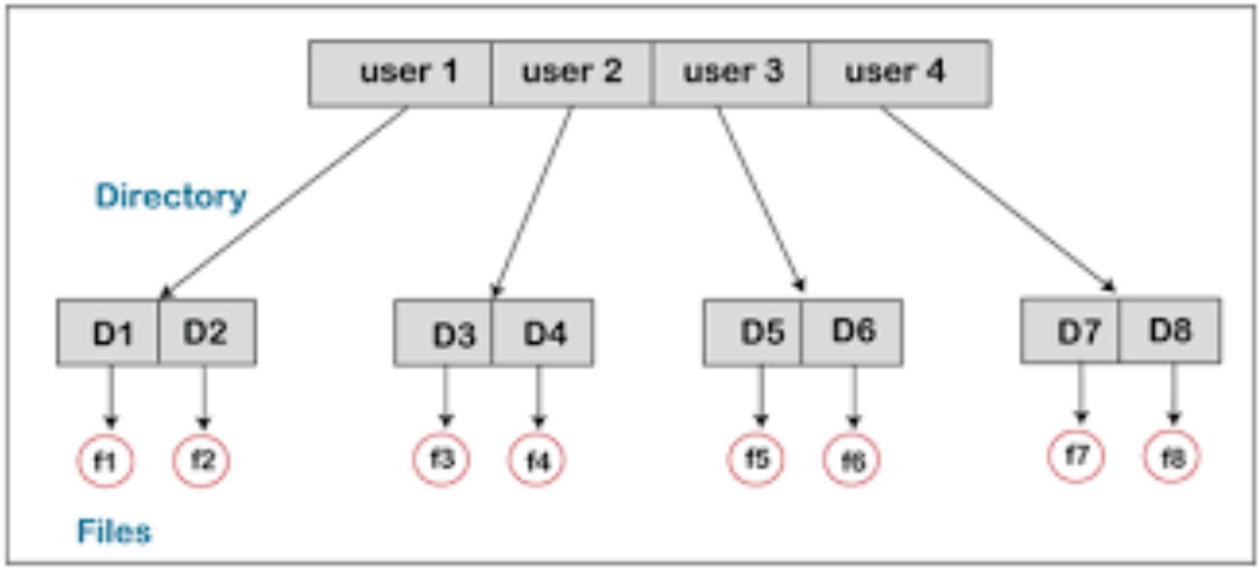
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

10.6.2.2 Two-Level Directory:

→ In the two-level directory structure, each user has his own user file directory (UFD).

- The UFDs have similar structures, but each lists only the files of a single user. When a user user logs in, the system's master file directory (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user.

→ When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.



→ A two-level directory can be thought of as a tree, or an inverted tree, of height 2.

→ The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves.The files are the leaves of the tree.

→ Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).

- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

- Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another.

- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.

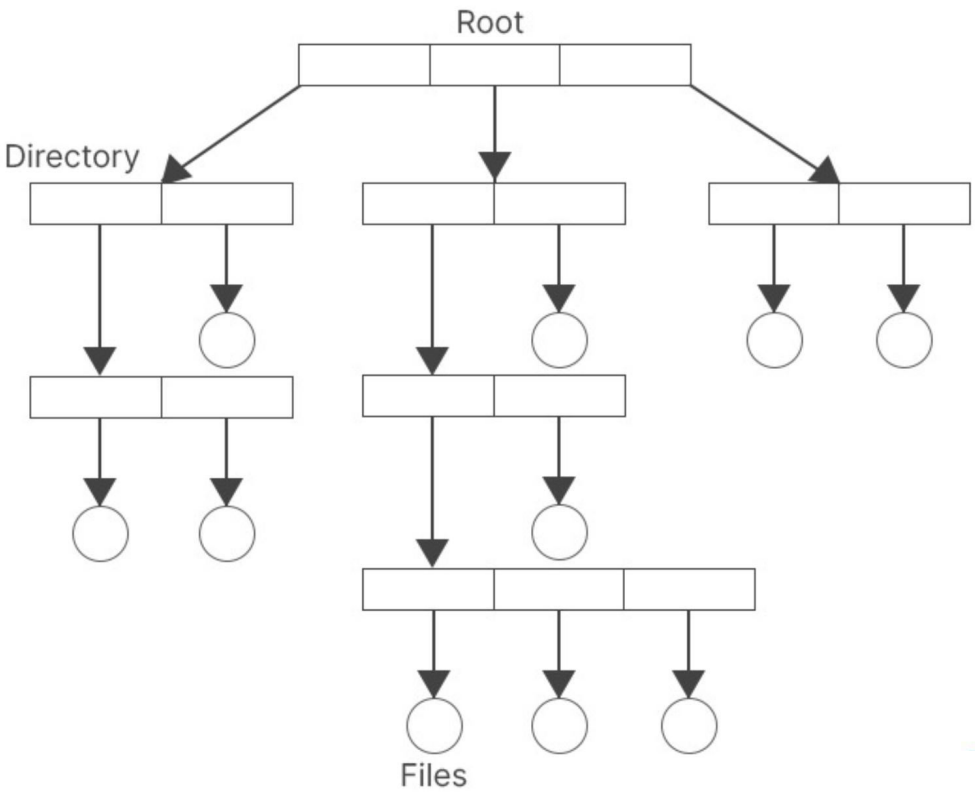
Some systems simply do not allow local user files to be accessed by other users.

10.6.2.3 Tree-Structured Directories:

➔ The two level directory structure could be extended to a multi-level generalization. This generalization allows users to create their own subdirectories and to organize their files accordingly.

- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

- A directory (or subdirectory) contains a set of files or subdirectories.



→ In normal use, each process has a current directory.

- The current directory should contain most of the files that are of current interest to the process. When reference is made to a file,the current directory is searched.

- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.

- To change directories, a system call- change directory() is provided that takes a directory name as a parameter and uses it to redefine the current directory.

→ Path names can be of two types: absolute and relative.

- An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.

- A relative path name defines a path from the current directory.

→ For example, in the tree-structured file system if the current directory is root/spell/mail, then

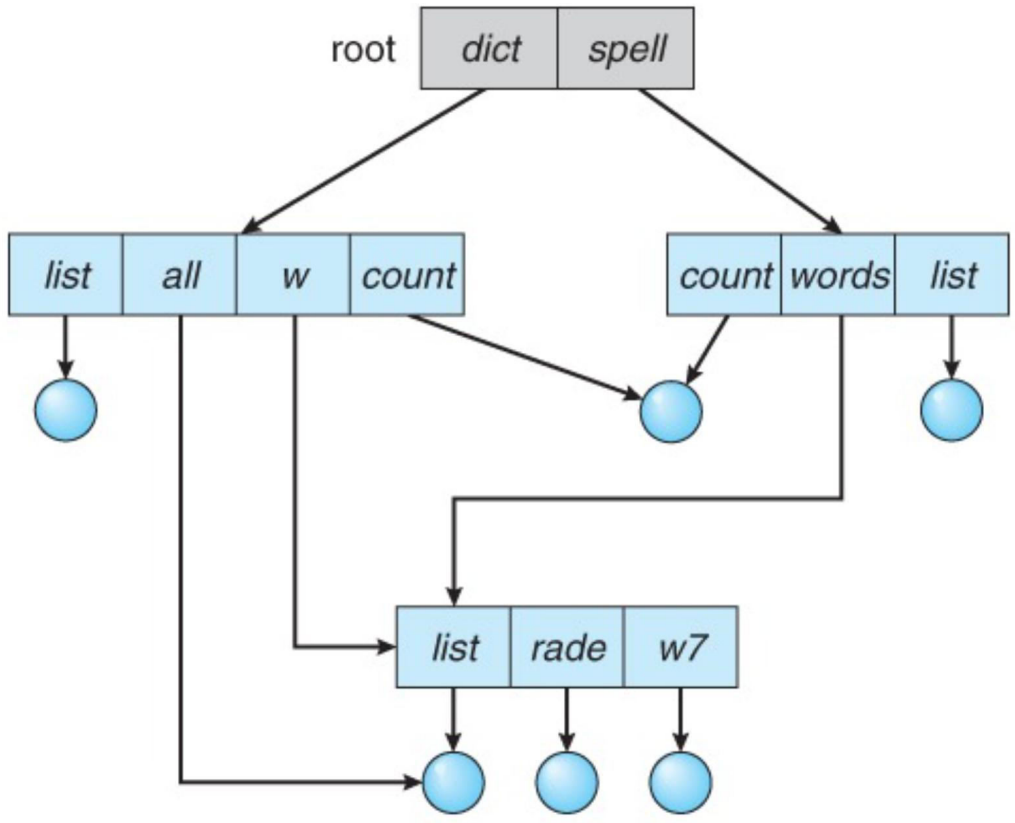
The relative path name : prt/first
The absolute path name : root/spell/mail/prt/first

10.6.2.4 Acyclic-Graph Directories:

→ Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory separating them from other projects and files of the two programmers.

→ But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be shared.

- A shared directory or file exists in the file system in two (or more) places at once.



- A tree structure prohibits the sharing of files or directories. An acyclic graph – that is, a graph with no cycles–allows directories to share subdirectories and files.

- It is important to note that a shared file (or directory) is not the same as two copies of the file.

- With two copies,each programmer can view the copy rather than the original,but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.

- Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

- When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.

10.7 Protection:

→ When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.

- Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

- In a larger multiuser system various file protection mechanisms are needed.

10.7.1 Types of Access:

- Systems that do not permit access to the files of other users do not need protection.

- Protection mechanisms provide controlled access by limiting the types of file access that can be made.

- Access is permitted or denied depending on several factors, one of which is the type of access requested.

- Read : Read from the file.
- Write : Write or rewrite the file.
- Execute : Load the file in to memory and execute it.
- Append : Write new information at the end of the file.
- Delete : Delete the file and free its space for possible reuse.
- List : List the name and attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled.

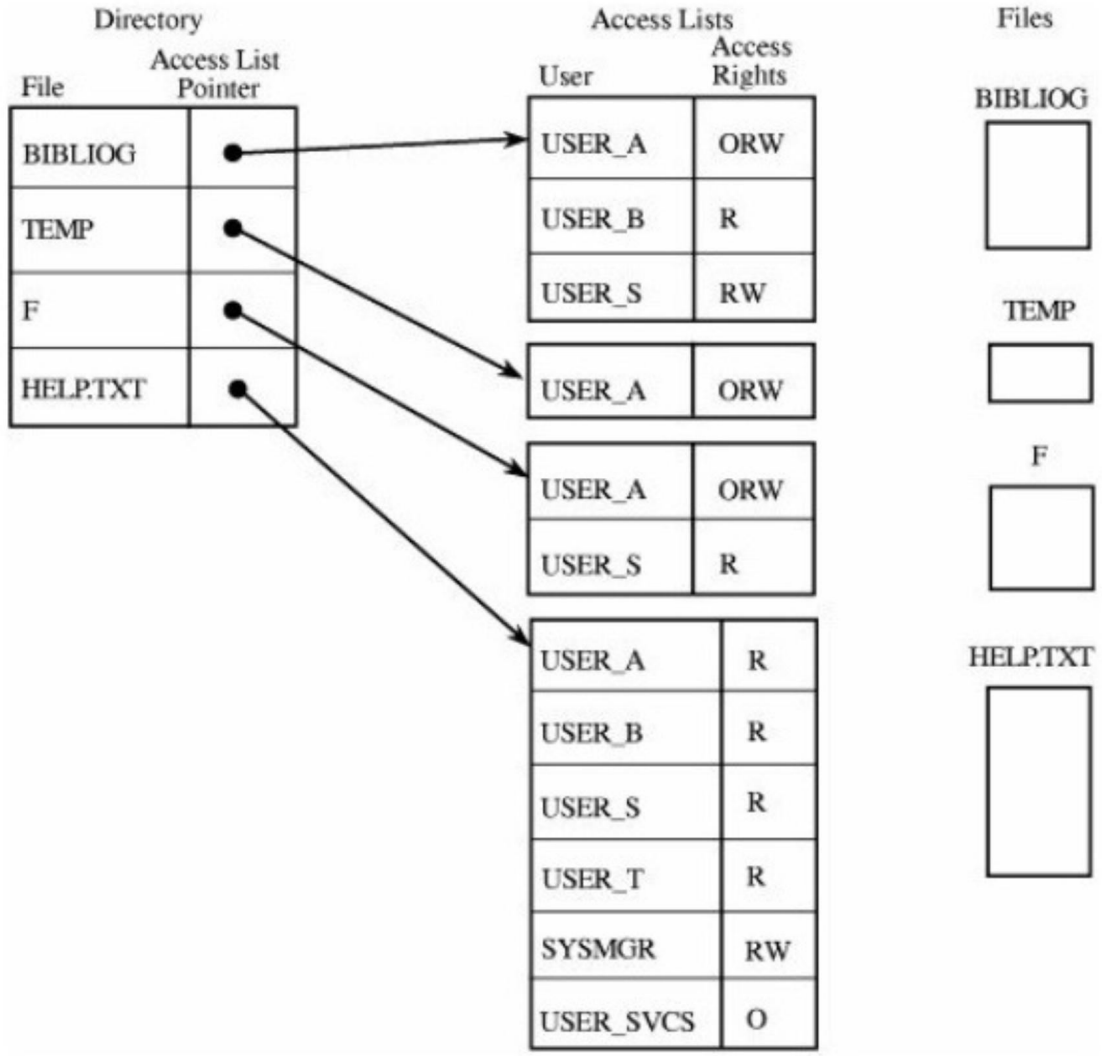
10.7.2 Access Control:

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- Different users may need different types of access to a file or directory.

The most general scheme to implement identity- dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.

- When a user requests access to a particular file, the operating system checks the access list associated with that file.

- If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.



- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.

→ To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- Owner : The user who created the file is the owner.
- Group : A set of users who are sharing the file and need similar access is a group, or work group.
- Universe : All other users in the system constitute the universe.

10.8 File Sharing:

→ When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent.

• Given a directory structure that allows files to be shared by users, the system must mediate the file sharing.

The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

→ Systems use the concepts of file (or directory) owner (or user) and group. The owner is the user who can change attributes and grant access and who has the most control over the file.

10.9 Allocation Methods:

→ The direct-access nature of disks gives us flexibility in the implementation of files.

- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

- Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed.

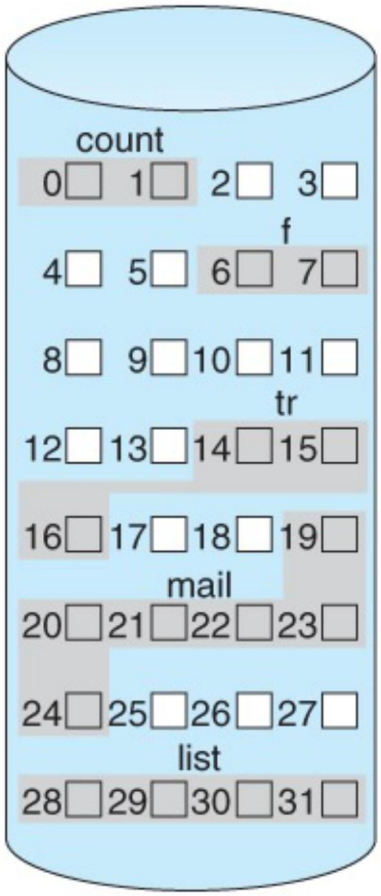
10.9.1 Contiguous Allocation:

→ Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.

- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.

- If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ... , b + n − 1.

- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Accessing a file that has been allocated contiguously is easy.

→ For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.

For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation.

- As files are allocated and deleted, the free disk space is broken into little pieces.

- External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.

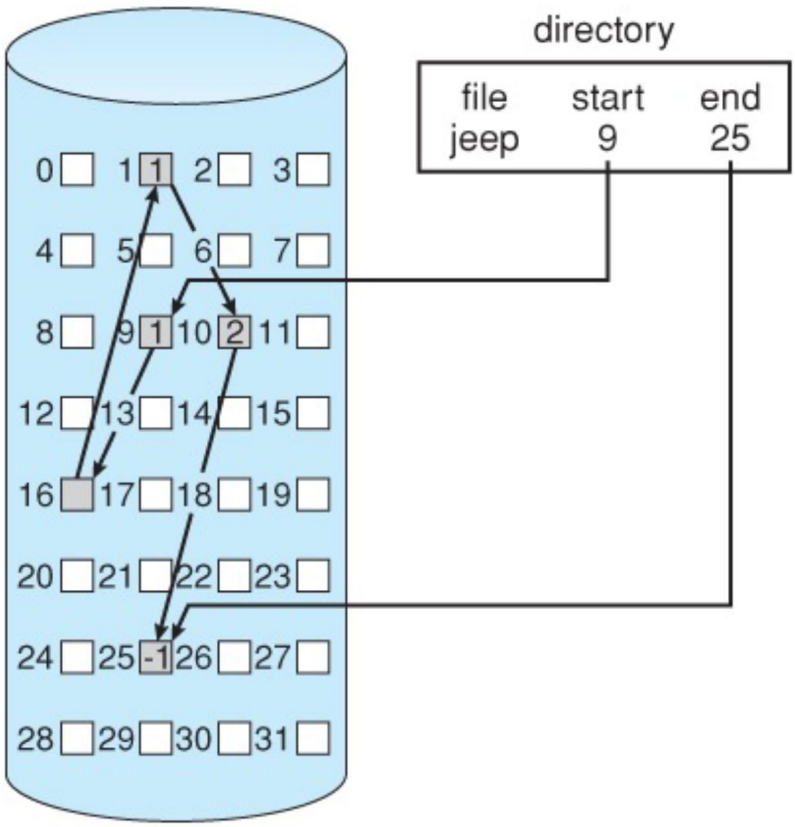
- One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space.

- We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem.

- The cost of this operation can be particularly high for large hard disks.

10.9.2 Linked Allocation:

→ Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.



- The directory contains a pointer to the first and last blocks of the file.

→ For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25.
Each block contains a pointer to the next block.

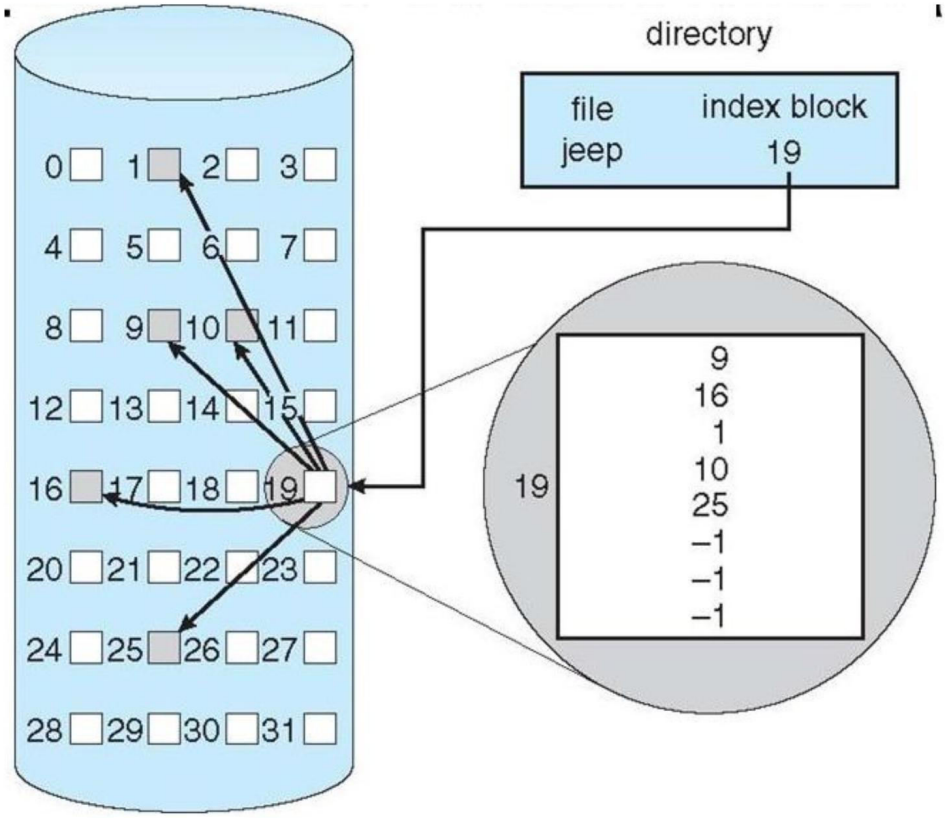
→ The advantages of indexed allocation include:

- Efficient file access: With the help of the index block, locating and accessing specific portions of a file becomes faster. Instead of traversing through the entire file allocation table, the index block provides direct access to the required data blocks.

→ Indexed allocation also has some limitations:

- Increased overhead: Each file requires an index block, which introduces additional overhead in terms of disk space The space required for index blocks can be significant depending on the size of the file.

- Limitation on file size: The size of the index block limits the maximum size of a file that can be stored using indexed allocation. If the index block size is fixed, there is a maximum number of disk block addresses that can be stored, restricting the file size.



- The directory contains the address of the index block. To find and read the ith block, we use the pointer in the ith index-block entry.

- When the file is created, all pointers in the index block are set to null. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.

→ Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.

→ A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file.

10.9.3 Indexed Allocation:

→ Indexed allocation is a file allocation method used by operating systems to manage disk space and keep track of files stored on a storage device

- Indexed allocation is basically extension of linked allocation.

→ Index allocation introduces a new concept of index block which stores all pointers into one location.So instead of iterating through the blocks to reach ith block in the file, we can access the ith block directly from the index block.

- Each file has its own index block, which is an array of disk-block addresses.
- The ith entry in the index block points to the ith block of the file.

- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.

- This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block and this new block is written.

- The major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.

- Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

- The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks.

- For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units.Pointers then use a much smaller percentage of the file's disk space.

10.10 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.

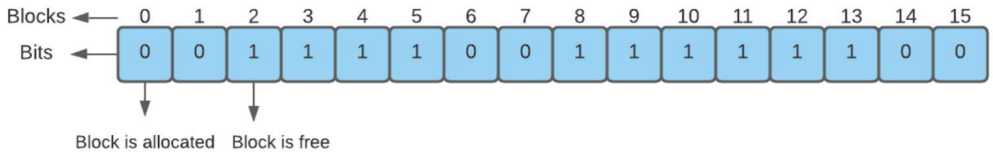
- The free-space list records all free disk blocks-those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file.

- 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.

10.10.1 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.

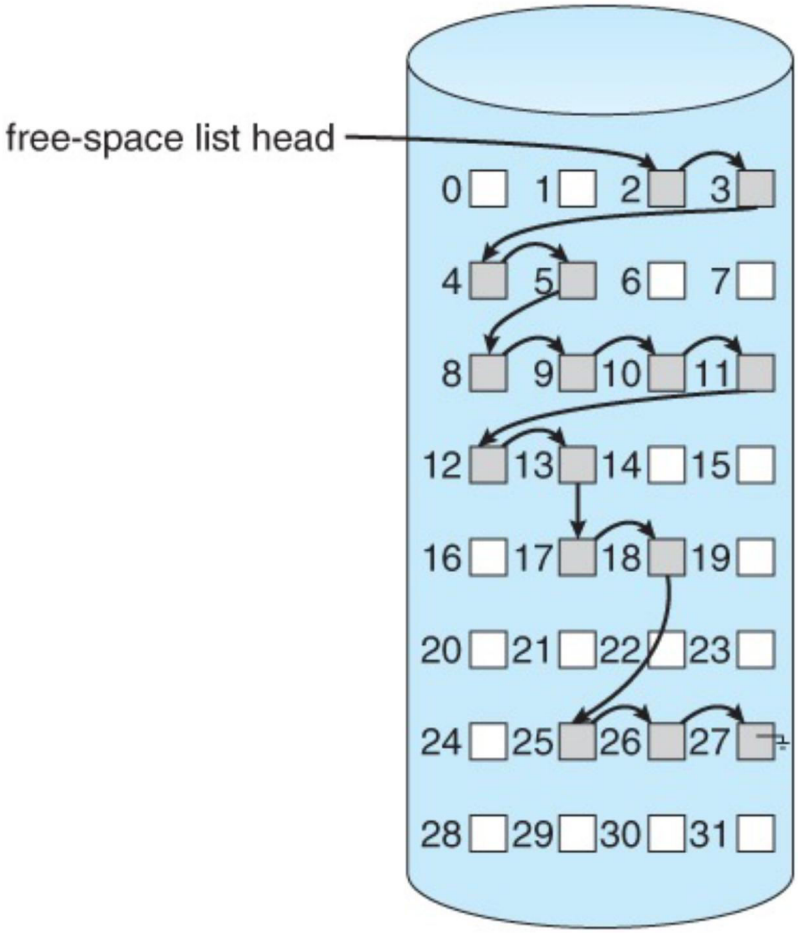
- The free-space bit map would be :

  00111100111111000110000011100000 ...

- 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.

10.10.2 Linked List:

➔ Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.



- Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.

- In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.

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

11. Top 30 most important Operating System questions frequently Asked In Technical Interviews

1. What is an operating system, and what are its primary functions?
2. Explain the differences between a process and a thread.
3. What is the role of the kernel in an operating system?
4. What is virtual memory, and how does it work?
5. Describe the process of context switching in an operating system.
6. What are system calls? Provide some examples.
7. Explain the differences between symmetric and asymmetric multiprocessing.
8. How do you handle deadlocks in an operating system?
9. What is the purpose of the page replacement algorithm, and discuss some popular algorithms.
10. Explain the concept of demand paging.
11. What is the difference between user-level threads and kernel-level threads?
12. Describe the boot process of an operating system.
13. Explain the concept of file systems and their organization.
14. What is a semaphore, and how is it used for process synchronization?
15. Describe the various scheduling algorithms used in operating systems.
16. Explain the working principle of the buddy memory allocation algorithm.

17. What are I/O operations, and how does the operating system handle them?

18. Discuss the concept of multi-core processors and their implications for operating systems.

19. What is a trap, and how is it used in operating systems?

20. Explain the differences between monolithic and microkernel operating system architectures.

21. How does the operating system manage CPU utilization and efficiency?

22. What is the purpose of a file descriptor in an operating system?

23. Describe the critical sections and the methods to protect them in concurrent programming.

24. How is process scheduling affected by real-time operating systems?

25. What are the advantages and disadvantages of using paging and segmentation in memory management?

26. Explain the concept of thrashing and how it can be avoided.

27. Discuss the differences between preemptive and non-preemptive scheduling.

28. What is the role of a device driver in an operating system?

29. Describe the concept of virtual machines and their benefits.

30. How do you handle memory leaks in an operating system?

References : Operating System Concepts,Ninth Edition,By Abraham Silberschatz, Peter B. Galvin, Greg Gagne

Internet

Was this helpful ?

Follow For More

Santosh Kumar Mishra

SDE@Microsoft, Author

FOLLOW ME
@iamsantoshmishra

FOLLOW ME
@iamsantoshmishra

SUBSCRIBE YT
@InterviewCafe

FOLLOW ME
@iamsanmishra