24CB402/24CM402/24DS402/24IT402

II/IVB. Tech (Regular) DEGREE EXAMINATION

Scheme of Evaluation

April,2026 **Common to CB, CM, DS &IT**

Fourth Semester **Operating Systems**

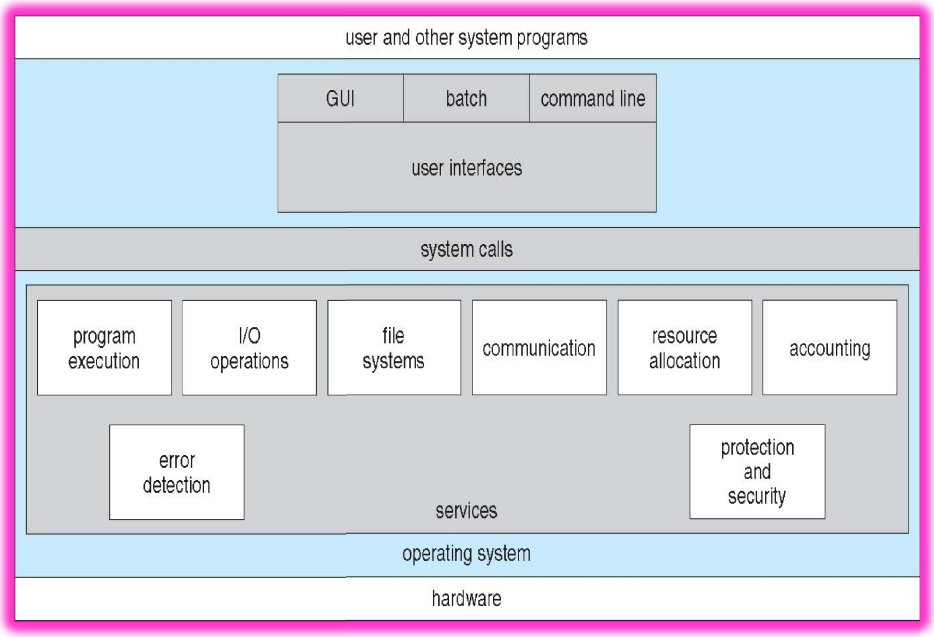
Time: Three Hours **Maximum:60 Marks**

Answer question 1 compulsorily. **(12X1 = 12 Marks)**

Answer one question from each unit. **(4X12=48 Marks)**

			CO	BL	M
1.	a)	List out any two operations of OS Here are some operations of an Operating System (OS): 1. **Process Management** 2. **Memory Management** 3. **File Management** 4. **Device Management** 5. **Security Management** 6. **Input/Output Management** 7. **User Interface Management** 8. **Resource Allocation** Student can write any of these.	CO1	L1	1M
	b)	Define a system call. ➢ **System calls provide the interface between a process and the operating system.**	CO1	L2	1M
	c)	What is multiprogramming? It is an operating system technique in which **multiple programs are kept in main memory at the same time**, and the CPU is switched to another program **only when the currently running program is waiting for an I/O operation**, in order to **maximize CPU utilization**.	CO1	L1	1M
	d)	Define Semaphore A **semaphore S is an integer variable** that, apart from initialization, is accessed only through two **standard atomic operations: wait() and signal().** The **wait() operation"to test", signal() "to increment".**	CO2	L2	1M
	e)	What is Mutex A **Mutex (Mutual Exclusion)** is a synchronization mechanism used in operating systems to **prevent multiple processes or threads from accessing a shared resource at the same time**.	CO2	L3	1M
	f)	Define Turnaround time. **amount of time to execute a particular process.** **TAT = Completion time – Arrival Time**	CO2	L1	1M
	g)	Define deadlock. A **deadlock** is a situation in an operating system where **two or more processes are permanently blocked** because each process is waiting for a resource that is held by another process.	CO3	L3	1M

		OR 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**			
	h)	What is Paging? **Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.**	CO3	L1	1M
	i)	Define Virtual memory **Virtual Memory** is a memory management technique used by an **Operating System** that allows a computer to run programs **larger than the available physical memory (RAM)** by using a part of the **secondary storage (hard disk/SSD)** as an extension of RAM.	CO3	L1	1M
	j)	What is RAID? → **A variety of disk-organization techniques, collectively called Redundant Arrays of Independent Disks (RAID).**	CO3	L1	1M
	k)	What is Protection? • Protection in an operating system ensures that **system resources are accessed in a controlled and authorized manner**.	CO4	L2	1M
	l)	Define Access Matrix An **Access Matrix** is a **protection model** used in an operating system **to control how processes access resources**	CO4	L1	1M
Unit –I					
2.	a)	What is an operating system? Explain its services. → A program that acts as an **intermediary between** a **user of a computer and the computer hardware.** **1) User Interface** **2) Program Execution** **3) I/O Ooperations** **4) File-SystemManipulation** **5) Communications** **6) Error Detection** **7) Resource Allocation** **8) Accounting** **9) Protection and Security**	CO1	L2	6M



1) User Interface - Almost all operating systems have an user interface (**UI**). Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**,**Batch Interface**

2) 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).

3) I/O Operations - A running program may require I/O, which may involve a file or an I/O device.

4) File-System Manipulation - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

5) Communications – Processes may exchange information, on the same computer or between computers over a network Communications may be via shared memory or through message passing (packets moved by the OS)

6) Error Detection – **The operating system needs to detect and correct errors constantly**

→ May occur in the CPU and memory hardware, in I/O devices, in user program

- → For each type of error, OS should take the appropriate action to ensure correct and consistent computing
- → Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

7) Resource Allocation - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them.

→ Many types of resources - CPU cycles, main memory, file storage, I/O devices.

8) Accounting - To keep track of which users use how much and what kinds of computer resources.

9) Protection and Security - The owners of information stored in a multiuser or networked computer system may want to control use of that information.

- **Protection** involves ensuring that all access to system resources is controlled
- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

1) Monolithic Structure –Traditional UNIX system structure and Linux System Structure
2) Layered Approach
3) Microkernels
4) Modules

Monolithic Structure – Original UNIX

The **simplest structure** for organizing an operating system is **no structure at all**. That is, **place all of the functionality of the kernel into a single**, static binary file that runs in a **single address space**. This approach—known as a **monolithic structure**—is a **common technique for designing operating systems**.

UNIX structure consists of **two separable parts:**
kernel and the system programs.

The **kernel is further separated into a series of interfaces and device drivers,** which have been added and expanded over the years as UNIX has evolved.
We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel.

The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into **one single address space.**

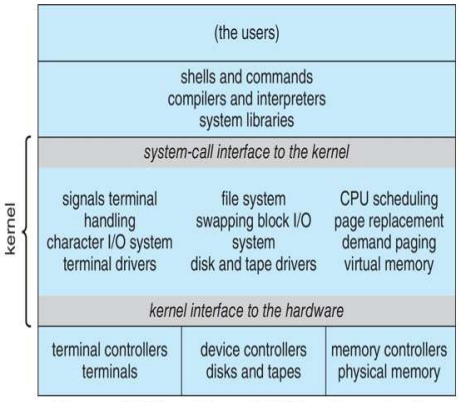


Figure 2.12 Traditional UNIX system structure.

Linux System Structure

Monolithic plus modular design

User programs run in user mode

Kernel runs in kernel mode

System calls provide safe access to OS services

Device drivers isolate hardware complexity

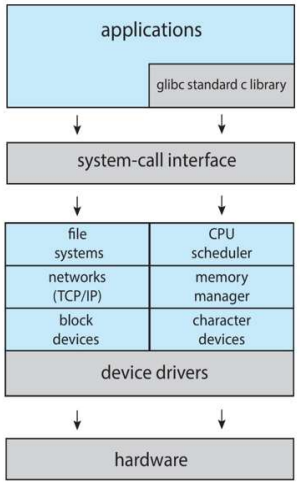


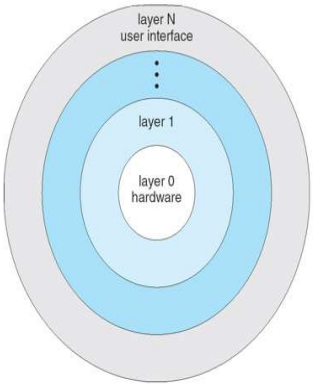
Figure 2.13 Linux system structure.

2) Layered Approach

A **system can be made modular in many ways**. **One method is the layered approach**, in which the operating system is broken into a number of layers (levels). This layering structure is depicted in Figure 2.14.

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The **bottom layer** (layer 0), is the **hardware**; the **highest** (layer N) is the **user interface**.

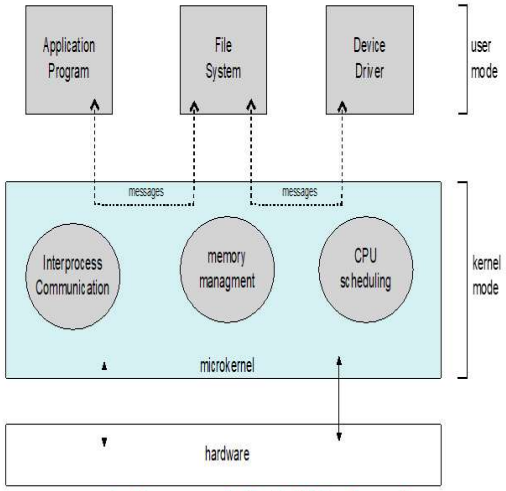
The layers are selected so that each uses functions (operations) and services of only lower-level layers



A Layered Operating System

3) Microkernel System Structure

- **Moves as much from the kernel into user space**
- **Mach OS** is an example of **microkernel approach**
- Communication takes place between user modules using **message passing**
- **Benefits:**
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- **Detriments:**
  - Performance overhead of user space to kernel space communication



Architecture of a Typical Microkernel

4) Modules

- **Many modern operating systems** implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc

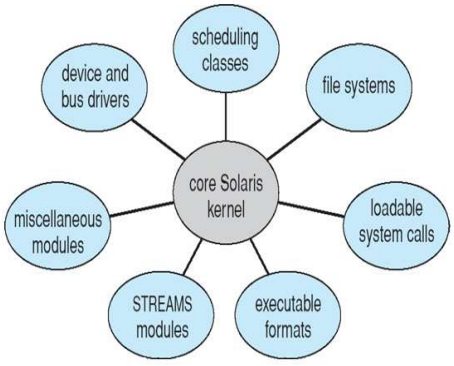


Figure 2.15 Solaris loadable modules.

The Solaris operating system structure, shown in Figure 2.15, is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules

		(OR)			
3.	a)	What are the advantages of inter-process communication? How communication takes place in a shared-memory environment? Explain.	CO1	L2	6M

advantages of inter-process communication

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. Notice that such a speedup can be achieved only if the **computer has multiple processing cores.**
- **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.

IPC in Shared-Memory Systems

- An **area of memory is shared among the processes** that wish to communicate.
- The communication is under the control of the user processes **not the operating system.**
- ✓ Major issue is to provide mechanism that will allow the user processes to **synchronize their actions** when they **access shared memory**.
- The processes are **also responsible for ensuring** that they **are not writing to the same location simultaneously.**
- **To illustrate the concept of cooperating processes**, let's consider the **producer–consumer problem**, which is a common paradigm for cooperating processes.
- A **producer** process produces information that is consumed by a **consumer** process.

❑ One solution to the producer–consumer problem **uses shared memory.** To allow producer and consumer processes to run concurrently, we must have **available a buffer of items** that can be filled by the **producer and emptied by the consumer**. This buffer will reside in a region of memory that is shared by the producer and consumer processes. **A producer can produce one item while the consumer is consuming another item.** ***The producer and consumer must be synchronized,*** **so that the consumer does not try to consume an item that has not yet been produced.**

Two types of buffers can be used.

- The **unbounded buffer** places **no practical limit on the size of the buffer**. The **consumer may have to wait for new items**, but the **producer can always produce new items**.
- The **bounded buffer** assumes a fixed buffer size. In this case, the **consumer must wait if the buffer is empty**, and the producer must wait if the buffer is full.

Shared data

in → next free position in the buffer;

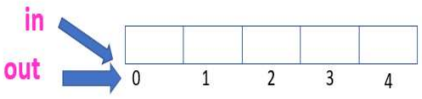
out → first full position in the buffer.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.13 The producer process using shared memory.

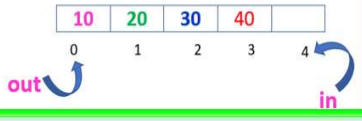
The buffer is empty when in = = out



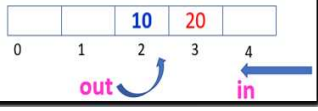

```
item next_consumed;

while (true) {
     while (in == out)
       ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.14 The consumer process using shared memory.

The buffer is full when ((in + 1) % BUFFER SIZE) == out



Let assume BUFFER SIZE = 5



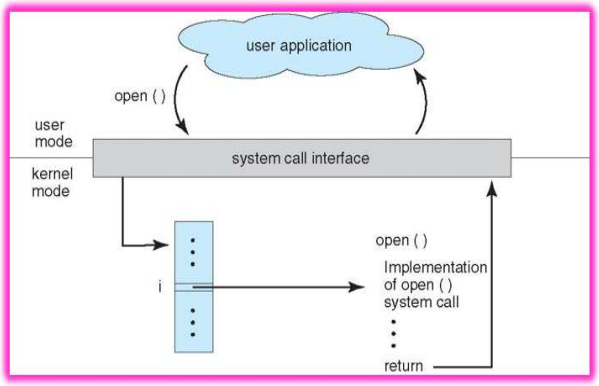
	b)	Examine the significance of system calls in process management	CO1	L2	6M

- System calls are the Programming interface to the services provided by the OS
- Programming interface are Typically written in a high-level language (C or C++)
- **System calls are** Mostly accessed by **programs** via a high-level **Application Programming Interface (API)** rather than direct **system call use**.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

System Call Implementation

- Typically, a number is associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface **invokes the intended system call in OS kernel and returns status of the system call and any return values**
- The caller need not know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface are hidden from programmer by API

API – System Call – OS Relationship



System Call Parameter Passing

Three general methods are used to pass parameters to the OS

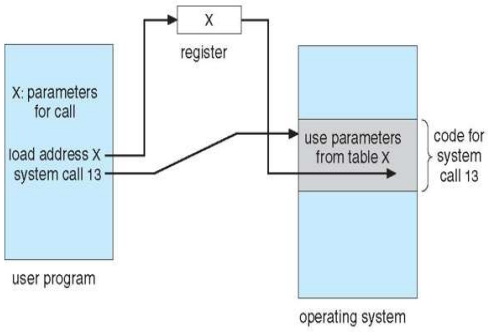
Simplest: Pass the parameters in registers

In some cases, may be **more parameters than registers**

- Parameters are stored in a **Block or table** in memory and **address of block is passed as a parameter in a register**
  - This approach is taken by **Linux and Solaris**
- Parameters placed or **pushed** onto the **stack** by the program and **popped** off the stack by the **operating system**

Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



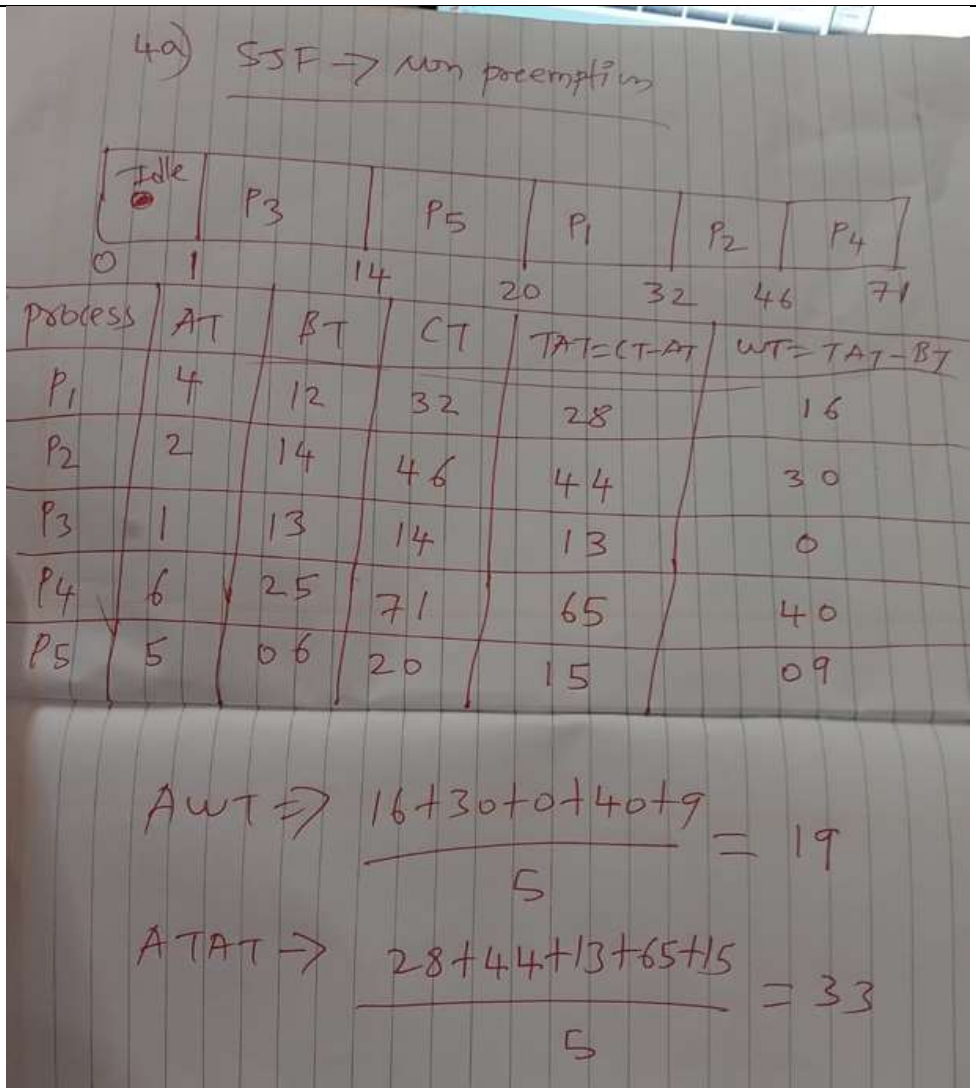
Unit –II

4.	a)	Consider the following set of processes, with the length of CPU-burst time (BT), arrival time (AT) and time quantum (TQ) of 6 milli seconds.	CO2	L3	6M

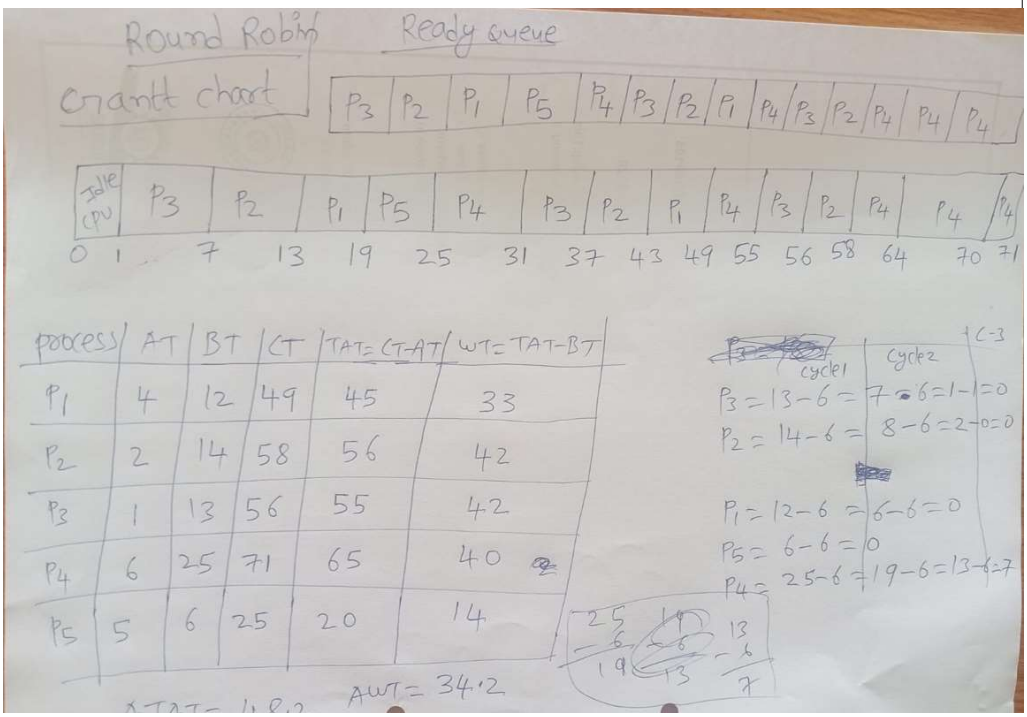
Process	Arrival Time	Burst Time
P1	4	12
P2	2	14
P3	1	13
P4	6	25
P5	5	6

Calculate average waiting time and average turn around time by
i) SJF (Non-Preemptive) ii) Round robin

SJF→3MARKS



Round Robin →3MARKS



example.

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- The wait() operation "to test"; signal() "to increment".

- Definition of the wait() operation

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

Definition of the signal() operation

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

Semaphore Usage Example:

```
mutex = initialized to 1
do {
    wait (mutex);

    // Critical Section

    signal (mutex);

                    // remainder section
} while (TRUE);
```

Two types of Semaphores

1) Counting semaphore – integer value can range over an unrestricted domain

2) Binary semaphore – integer value can range only between 0 and 1

What is a Counting Semaphore?

A **counting semaphore** is a **synchronization tool** used in operating systems to **control access to a resource that has multiple instances**.

Example:

- 5 printers
- 3 database connections
- 10 empty buffer slots

Counting semaphore keeps track of how many are free.

Definition of Binary Semaphore.

A **binary semaphore** is a synchronization mechanism that can take **only two values: 0 and 1**, and is used to ensure **mutual exclusion** by allowing **only one process** to **access a critical section at a time.**

Student can take any of the two types of semaphores as an example.

(OR)

5.	a)	What is critical section problem? Provide a software-basedsolution to it.	CO2	L2	6M

The Critical-Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing **common variables, updating table, writing file, etc**
  - **When one process is in its critical section, no other process may enter its critical section.**
  - 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. **That is, no two processes are executing in their critical sections at the same time.**

The *critical-section problem* is to design a **protocol** that the **processes can use to cooperate**.

1) Each process must **request permission to enter its critical section**. The section of code implementing this request is the **entry section**.
2) The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

The general structure of a typical process *Pi* is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Figure 5.1 General structure of a typical process P_i.

A **solution to the critical-section problem** must satisfy the following **three requirements**:

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

2. **Progress.** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the **selection of the processes that will enter the critical section** next cannot be postponed indefinitely.

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

	b)	Discuss Peterson's Solution	CO2	L2	6M

Peterson's Solution

A classic software-based solution to the **critical-section problem** known as **Peterson's solution**.

→ It is a Two process solution. (P_i & P_j)

Peterson's solution requires the two processes to share two data items:

```
int turn;
Boolean flag[2]
```

The variable **turn indicates whose turn it is to enter the critical section**.

The **flag** array is used to indicate if a process is **ready to enter the critical section**.

flag[i] = *true* implies that process P_i is ready!

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Turn value can be 0 or 1. Boolean flag[2], holds either True or False

- Provable that the **three requirements** for **critical-section problem** requirement are met:
  1. Mutual exclusion is preserved
     P_i enters **critical-section** only if:
     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Progress

If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section.

Bounded waiting

- When Pi sets flag[i] = true, it declares its intent to enter
- Pj can enter the critical section **at most once** before Pi
- After Pj finishes, it sets flag[j] = false
- Now Pi is guaranteed to enter next

Unit –III

6.	a)	Write in detail about deadlock prevention	CO3	L3	6M

Deadlock Prevention

- **Deadlock prevention means that a system has to ensure that at least one of the four deadlock conditions cannot hold.**

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

At least one resource must be non-sharable.

Sharable Resources

Can be used by **many threads at the same time**.
Do **not** require mutual exclusion.
Cannot cause deadlock.

Non-Sharable Resources

Can be used by **only one thread at a time**.
Require **mutual exclusion**.
Can lead to **deadlock**.

2) **Hold and Wait** – it must guarantee that whenever **a Thread requests a resource**, **it does not hold any other resources.**

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

OR

- Each thread must ask for and get all the resources it needs **before it starts running**.

❖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.**

Disadvantages of These Deadlock Prevention Protocols are

1. Low Resource Utilization

- Resources may be **allocated but not used immediately**.
- They remain **idle for a long time**.

2. Starvation (Indefinite Waiting)

- A thread may need **multiple resources**.
- If at least one required resource is always busy, the thread may **wait forever**.
- The thread never gets a chance to execute.
- This condition is called **starvation**.

3) No Preemption: **If preemption is allowed → deadlock can be prevented.**

Protocol 1: Preempt All Resources

•If a thread:
- Is holding some resources, and
- Requests another **resource that is not available**

•Then:
- **All currently held resources are taken back (preempted).**
- These resources are added to the thread's waiting list.
- The thread restarts only when:
  - It gets back its old resources, and
  - It gets the new requested resource.

Protocol 2: Preempt from Waiting Threads

•When a thread requests resources:
- If resources are **available** → allocate them.
- If not available:
  - Check if they are held by a **waiting thread**.
  - If yes → **preempt from that waiting thread and give to requesting thread**.

4) Circular Wait – Give each resource type a unique number.

This is called **total ordering of Resources**.

- **impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.**
- **processes are not allowed to request a lower-numbered resource after holding a higher one.**

→ we define a one-to-one function F: *R —> N, where N is the* set of natural numbers.

For example, if the set of resource types *R includes* tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1 **F(tape drive) <F(printer)**
F(disk drive) = *5*
F (printer) = 12

	b)	Explain resource-allocation-graph algorithm for deadlock avoidance.	CO3	L1	6M

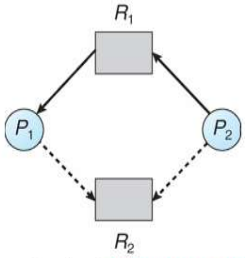
Avoidance Algorithms

- **Single instance of a resource type**
  - **Use a resource-allocation graph**

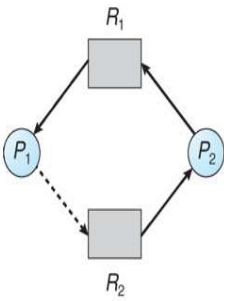
Resource-Allocation-Graph Algorithm

- Claim edge P_i --> R_j indicated that process P_i may request resource R_j in the future; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

P1, P2 Claim edge R2;
R2 still free



Unsafe state



Algorithm: Request can be granted only if converting request into assignment does not lead to an unsafe state.

(OR)

7. a) Explain the concept of paging — CO3 — L2 — 6M

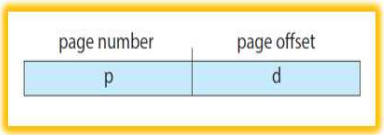
→ Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.

→ Divide physical memory into fixed-sized blocks called frames.

→Divide logical memory into blocks of same size called pages.

When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d):



Every address generated by the CPU is divided into two parts: a **page number (p) and a page offset (d).**

The page number is used as an index into a page table.

The page table contains the base address of each page in physical memory.

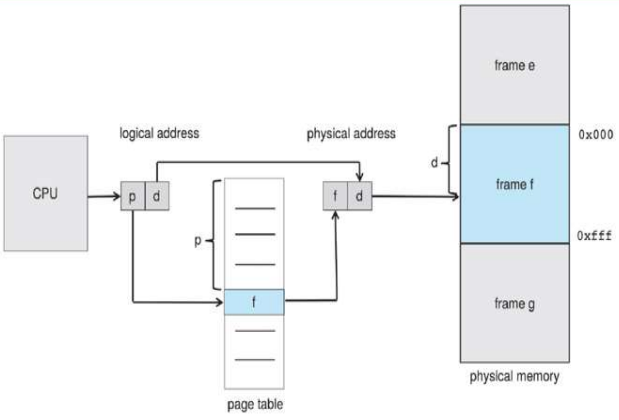


Figure 9.8 Paging hardware.

- This **base address is combined with the page offset to define the physical memory address that is sent to the memory unit.**

3. Protection

4. Shared Pages

b) Illustrate Demand paging with a suitable example — CO3 — L2 — 6M

Demand Paging

Define Demand Paging:

Demand paging is a method of **virtual memory** in which **pages are brought into main memory only when they are needed for execution**.

→ A demand-paging system is similar to a paging system with swapping (Figure 9.4) 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 never swaps a page into memory unless that page will be needed.

→ Bring a page into memory **only when it is needed. Thus** Less memory needed.

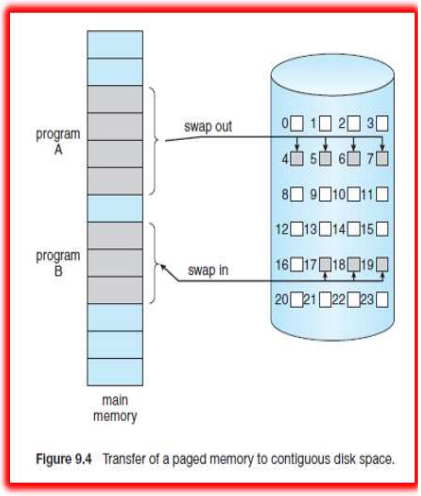


Figure 9.4 Transfer of a paged memory to contiguous disk space.

Example

Suppose a program has **10 pages**, but initially only **3 pages** are loaded into memory.

If the CPU tries to access **page 7**, which is not in memory:

1. A **page fault** occurs.
2. The OS loads **page 7** from disk into RAM.
3. The program continues execution.

Define Page Fault:

When the CPU tries to access a page that is not in memory.

The procedure for handling this page fault is straightforward (Figure):

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 (by taking one from the free-frame list, for example).

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 kept with the process and the 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.

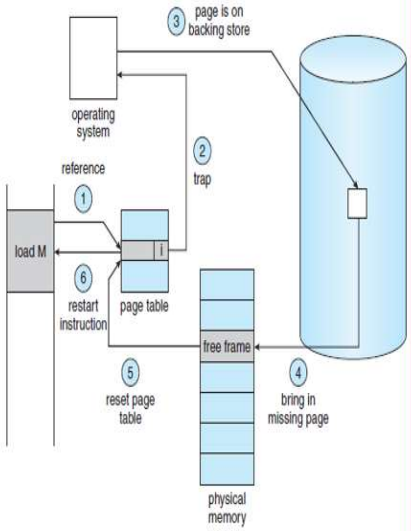


Figure 9.6 Steps in handling a page fault.

Unit –IV

8. a) What are the attributes of a file and also what are basic file operations? CO4 L1 6M

1) File Attributes

A file's attributes typically consist of these:

1) Name – only information kept in human-readable form.

2) Type – needed for systems that support different types.

3) Location – pointer to file location on device.

4) Size – current file size.

5) Protection – controls who can do reading, writing, executing.

6) Time, date, and user identification –this information may be kept for creation, last modified and last use(used for data for protection, security, and usage monitoring).

→ Information about files are kept in the directory structure, which is maintained on the disk.

2) File Operations

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

- Creating a file
- Opening a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncate a file

Repositioning within a file **means moving the file pointer to a new location inside the file**. This is called a **file seek**, and it usually does not require reading or writing data.

b) Explain the following Disk Scheduling algorithms with the string of cylinders: 98, 183, 37,122,14,124,65,67 and current head position is at 53. — CO4 — L3 — 6M

a) FCFS scheduling
b) SSTF scheduling
c) SCAN scheduling
d) C- SCAN scheduling
e) C-LOOK scheduling

1) FCFS → The simplest form of disk scheduling is, the first-come, first-served (FCFS) algorithm.

Consider, for example, a disk queue with requests for I/O **to blocks on cylinders.**

queue (0-199).

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

Head pointer 53

If the disk head is **initially** at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a **total head movement of 640 cylinders**. This schedule is diagrammed in Figure.

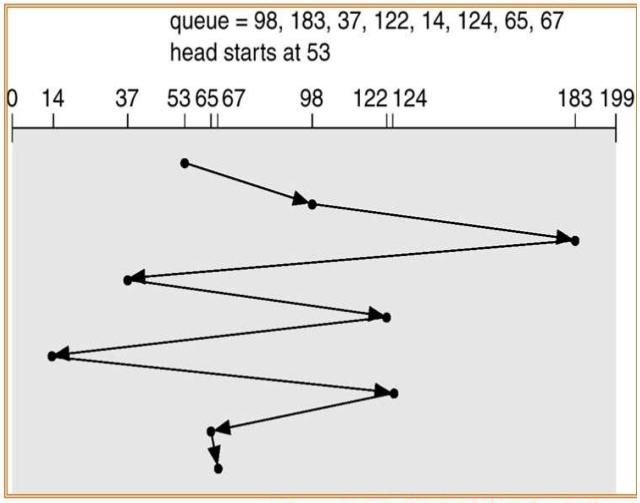


Figure 11.6 FCFS Disk Scheduling

Illustration shows total head movement of 640 cylinders.

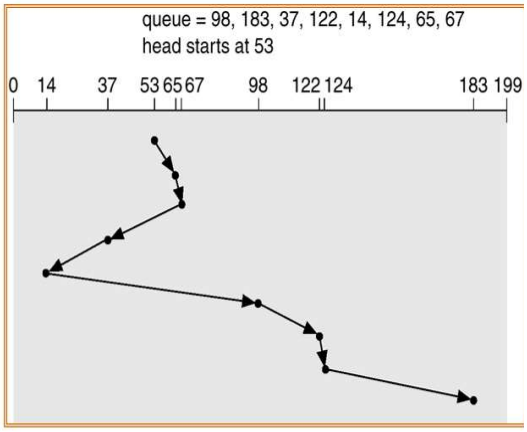
2) SSTF

→ Shortest-seek-time-first (SSTF) algorithm.
The SSTF algorithm selects the request with the minimum seek time from the current head position.

→ SSTF chooses the pending request closest to the current head position.

→ For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122, 124, and finally 183 (Figure 12.5).

→ SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling;



This scheduling method results in a total head movement of only 236 cylinders.
Although the SSTF algorithm is a substantial improvement over the FCFS algorithm.

3) SCAN

→ In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.

→ At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

→ The SCAN algorithm is sometimes called the elevator algorithm.

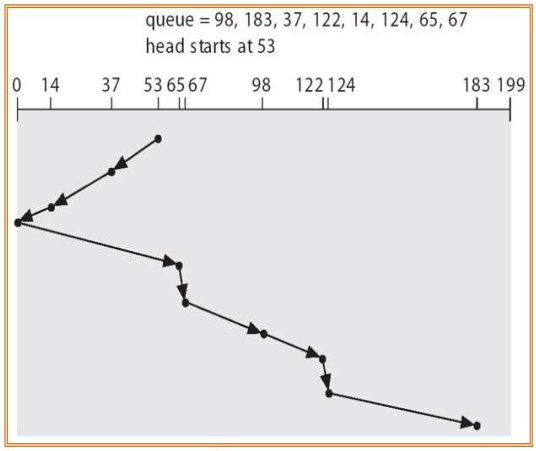


Figure → SCAN Disk Scheduling

4) C - SCAN

→ Circular SCAN (C-SCAN).

- C-SCAN Disk Scheduling is an improved version of the SCAN (elevator) algorithm

→ Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

- When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure).

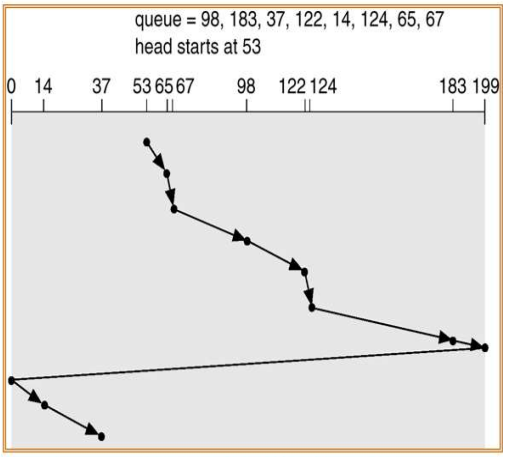


Figure → C-SCAN Disk Scheduling

5) C-LOOK

C-LOOK → the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

- The disk head **does NOT go to the physical end (0 or max track)**.
- It stops at the **last request present in that direction**.

In C-LOOK, **the disk arm moves only up to the last request** and then jumps to the first request without going to the disk ends, **reducing unnecessary head movement.**

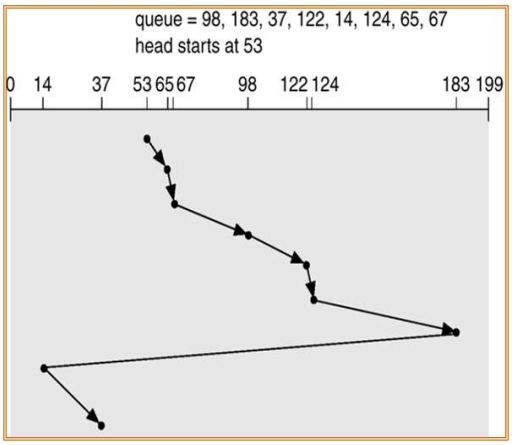


Figure → C-LOOK Disk Scheduling

(OR)

9.	a)	Discuss in detail about various directory structures.	CO4	L2	6M

1) Single-Level Directory

2) Two-Level Directory

3) Tree-Structured Directories

4) Acyclic-Graph Directories

5) General Graph Directory

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 (Figure 13.7).

A single-level directory means all files are stored in just one folder. It is very simple to use, but it creates problems when there are **many files or multiple users**, because **every file must have a unique name.**

•Example:
•Two users cannot use same name like **test.txt**
•Causes **confusion in multi-user systems.**

Not Suitable for Multiple Users
•Different users may want same file names
•**System cannot allow duplicate names**

Difficult File Management
•When number of files increases:
- Hard to remember file names
- **Difficult to search files**

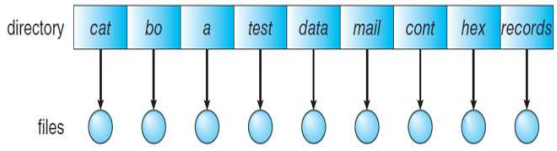


Figure 13.7 Single-level directory.

2) Two-Level Directory

A **two-level directory solves the problem** of **file name conflicts** by giving **each user their own directory**.

Two levels:
1.MFD (Master File Directory) → contains user names
2.UFD (User File Directory) → contains files of each user

Faster Search
•Search is limited to **one user directory**

No Name Conflict
•Different users can have **same file name**
- Example:
  - User A → test.txt
  - User B → test.txt

Disadvantages
(a) No File Sharing
•Users cannot easily share files with others
(b) Limited Structure
•No subfolders inside UFD
•Still **not suitable for very large systems**

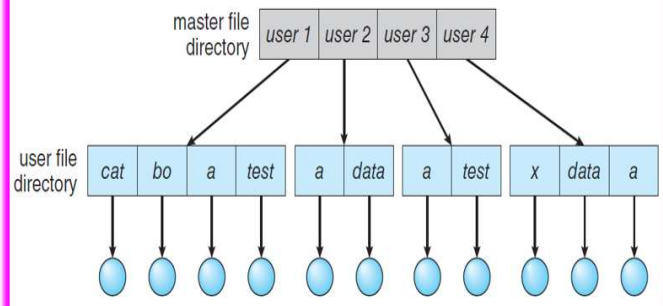


Figure 13.8 Two-level directory structure.

3) Tree-Structured Directories

A **tree-structured directory** is like a hierarchy of folders (like a tree).
There is a **root directory at the top**, and inside it are folders (directories), which can again contain subfolders and files.

Tree-structured directory Structure contains

- **Root directory (top level)**
- **Subdirectories (folders inside folders)**
- **Files stored at any level**

Path Names

- **Used to locate files**

(a) Absolute Path
Starts from root
Example: /home/user/docs/file.txt

(b) Relative Path
Starts from current directory
Example: docs/file.txt

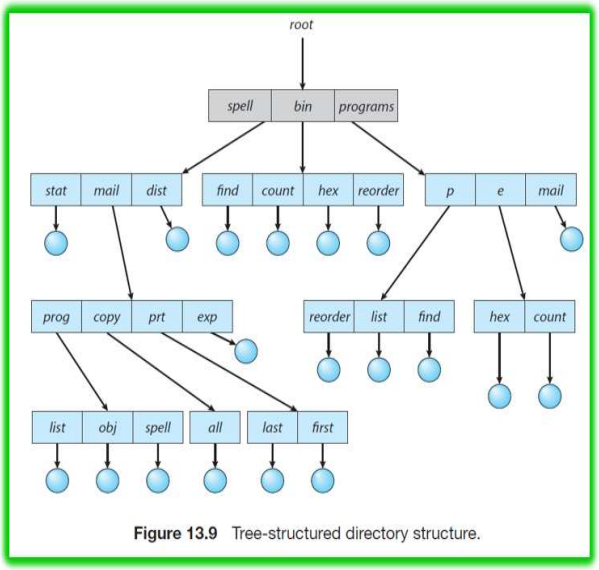


Figure 13.9 Tree-structured directory structure.

4) Acyclic-Graph Directories

- Have shared subdirectories and files

Definition:
An **acyclic-graph directory** allows **sharing of files and directories**, but it does **not allow cycles (loops)**.

- A file/folder can be **shared by multiple users.**
- **But you cannot come back to the same directory through a loop.**

Advantages

- Saves **memory space** (no duplicate files)
- Easy **file sharing between users**

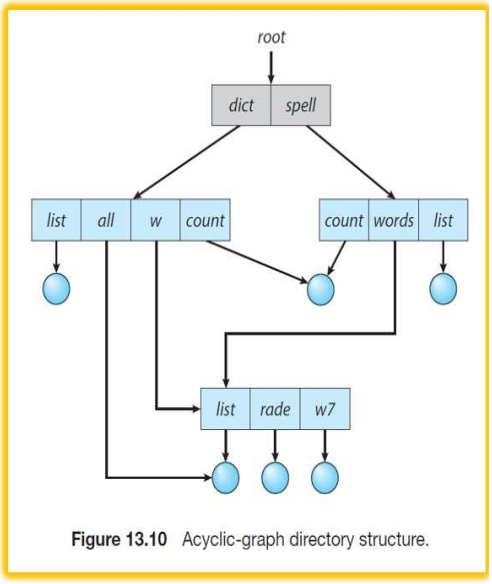


Figure 13.10 Acyclic-graph directory structure.

5) General Graph Directory

A **general graph directory** is similar to acyclic graph **but allows cycles (loops).**

Cycle → **You can navigate in a loop and come back to same directory.**

How the Cycle Works

- Start at **A**
- Go to **B → C**
- Inside **C**, there is a link pointing back to **A**

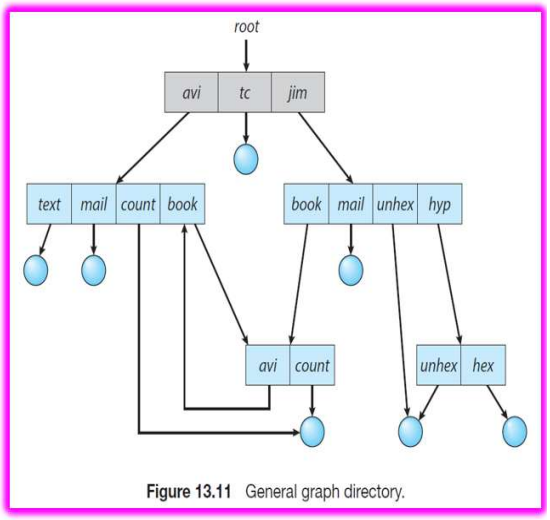


Figure 13.11 General graph directory.

	b)	List out the Principles of Protection	CO4	L1	6M

Principles of Protection

Principle of Least Privilege

- Give **minimum access rights** required to **perform a task**.
- Prevents:
  - Accidental damage
  - Misuse of system resources

Protection as Defense

- Permissions act like an **immune system**
- They:
  - Block unauthorized actions
  - Reduce impact of attacks

Risks of High Privileges

- Root/Admin has **full control**
- Problems:
  - Human errors (accidental delete)
  - Malware attacks (virus, buffer overflow)
- If attacker gets admin access → **system failure possible**

Compartmentalization

•Divide system into **smaller protected parts**

•**Each part** has **restricted access**

If one part is attacked, Other parts **remain safe**.

Audit Trail

•Record of all system activities

•Stored in **system logs**

Uses:

•Detect suspicious behavior

•Analyze attacks

•Track damage

Defense in Depth

•Use **multiple layers of security**

•Even if one layer fails, Others still protect system

Prepared by **HOD-CSE(AIML)**

1) Signature of Faculty (Data Science)

2) Signature of Faculty (Cyber Security)

4) Signature of Faculty (AIML)