



KAMALDNP

kamaldnp.com • IT Training, Varanasi

PDF created by Raj Kamal using AI

Short Notes — Operating Systems

BTEUP Diploma in Computer Science & Engineering • 3rd Semester • Short Notes
Session 2026-27

Short Revision Notes — prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus. This is NOT a BTEUP board paper or publication.

These **short revision notes** cover all five units of Operating Systems from the official BTEUP 2026-27 syllabus for CSE 3rd semester, in syllabus order. Each point gives the definition, steps, formulas and differences that a short or five-mark answer needs, in plain English.

Use them with the **official syllabus PDF** and the matching practice paper: revise one unit, recall its points without looking, then write the likely question at the end of that unit. These are revision notes only, not a replacement for your textbook and class notes.

Unit 1 — Overview of Operating System (6 Periods, 14%)

Topics: Introduction; different types of operating systems; functions and services of an OS; UNIX/Linux architecture; kernel; services and system calls; system programs.

Introduction

An **operating system (OS)** is system software that acts as an interface between the user, application programs and the computer hardware. The user never controls the CPU, memory or disk directly; the OS does it on the user's behalf.

Main goals of an OS:

- **Convenience** — make the computer easy to use.
- **Efficiency** — use CPU, memory and devices fully, with minimum wastage.

- **Ability to evolve** — allow new hardware and services to be added without disturbing old work.

Common examples are Windows, Linux, UNIX, Android and macOS. In a diagram, always show the layers in this order: user at the top, then applications, then the OS, then hardware at the bottom. Remember this one line for any introduction answer: without an OS, every program would have to control the hardware itself, which would be slow, difficult and unsafe.

Different types of operating systems

Operating systems are classified by how they handle jobs and users:

- **Batch OS** — similar jobs are grouped and run together without user interaction during execution; suitable for payroll and billing, but debugging is difficult and CPU may idle while a job waits for I/O.
- **Multiprogramming OS** — several jobs are kept in memory and CPU switches to another when one waits for I/O, improving CPU use.
- **Multiprocessing OS** — uses two or more CPUs for parallel execution, giving higher speed and reliability.
- **Time-sharing (multitasking) OS** — CPU time is divided into small slices among many users, so each feels the machine is his own; response time is short.
- **Real-time OS (RTOS)** — responds within a fixed deadline; used in process control, medical and defence systems. Hard RTOS never misses a deadline, soft RTOS tolerates small delays.
- **Distributed OS** — manages many connected computers as one system.

Functions and services of an OS

The OS performs the following main **functions** for every program:

- **Process management** — creating, scheduling and terminating processes.
- **Memory management** — allocating and freeing main memory and keeping track of which part is used by whom.
- **File management** — creating, reading, writing, naming and protecting files and directories.
- **Device (I/O) management** — controlling devices through drivers and buffering.
- **Security and protection** — passwords, access permissions and protection against unauthorised use.
- **Error detection** — watching CPU, memory and devices for failures and reporting them.
- **Job accounting** — recording time and resources used by each user.

These functions are offered to programs as **services**: program execution, I/O operations, file-system manipulation, communication between processes, and error reporting. In an exam, write the function name in bold first, then one line on what it does.

UNIX/Linux architecture

The **UNIX/Linux architecture** is shown in four layers, from outside to inside:

- **Hardware** — CPU, memory, disks and terminals at the centre.
- **Kernel** — the core program around the hardware; it manages processes, memory, files and devices, and applications reach hardware only through it.
- **Shell** — the command interpreter around the kernel. It reads a command typed by the user, asks the kernel to execute it and shows the result. Examples: sh, bash, csh.
- **Applications and utilities** — the outer layer: compilers, editors, user programs and commands such as cp, mv and ls.

Diagram in words: draw four concentric boxes — hardware in the middle, then kernel, then shell, then applications and users outside. Key features of UNIX/Linux to quote: written largely in C, multi-user and multitasking, portable across machines, and built on the idea that everything is treated as a file.

Kernel

The **kernel** is the central, resident part of the operating system — the first part loaded at booting and the part that stays in memory while the computer runs. It runs in a privileged **kernel mode**, while ordinary programs run in restricted **user mode**.

Main jobs of the kernel:

- Scheduling processes on the CPU and switching between them.
- Allocating memory and handling page faults.
- Handling interrupts from hardware and system calls from programs.
- Managing device drivers, files and inter-process communication.

Types: a **monolithic kernel** keeps all OS services in one large kernel (Linux) — fast, but a fault in one part can crash the whole system. A **microkernel** keeps only basic services such as scheduling and communication in the kernel and runs other services in user space — slower due to message passing, but more reliable and easier to extend.

Services and system calls

A **system call** is the programmed way by which a user program requests a service from the kernel, for example to read a file or create a process. It works like a controlled gate: the program cannot enter kernel mode itself, so it makes a system call and the kernel does the privileged work safely.

Types of system calls:

- **Process control** — create, terminate, load, execute a process (fork, exit).
- **File management** — create, open, read, write, close a file (open, read, write).
- **Device management** — request, release, read or write a device.
- **Information maintenance** — get or set time, date and process data.
- **Communication** — create a connection, send and receive messages.

Programs normally call system calls through an **API** such as POSIX or the Windows API, and parameters are passed by registers, a block in memory, or the stack.

System programs

System programs sit between the user and the system calls. They provide a convenient environment for program development and execution, and most users judge the OS by these programs, not by system calls.

Categories to write in an answer:

- **File management** — create, delete, copy, rename, list files (cp, mv, rm, ls).
- **Status information** — date, time, free memory and disk space, list of users (date, df, top).
- **File modification** — text editors and search tools (vi, grep).
- **Programming-language support** — compilers, assemblers, interpreters and debuggers (gcc).
- **Program loading and execution** — loaders that place a program in memory and run it.
- **Communications** — mail, web browsing, remote login and file transfer.

The **shell (command interpreter)** is the most important system program: it takes the user's next command and executes it.

Likely question: Describe the architecture of UNIX/Linux with a neat diagram. What is a kernel? Differentiate between a monolithic kernel and a microkernel.

Unit 2 — Process Management (10 Periods, 25%)

Topics: Process concepts; operations on processes; CPU scheduling; scheduling algorithms; IPC (inter-process communication); process synchronisation; critical section; deadlock — condition, avoidance and prevention.

Process concepts

A **process** is a program in execution. A program is a passive file on disk; when it is loaded into memory and runs, it becomes an active process. One program, such as a browser, can run as many processes.

A process in memory has four parts: **text** (code), **data** (global variables), **heap** (memory allocated during running) and **stack** (temporary data such as function parameters and return addresses). The **Process Control Block (PCB)** stores everything the OS needs about it: process ID, state, program counter, CPU registers, scheduling and memory information, and I/O status.

Process states: New → Ready (waiting for CPU) → Running → Waiting (for I/O or an event) → back to Ready, and finally Terminated. Only one process per CPU core is Running at a time; the rest wait in ready or waiting queues.

Operations on processes

The OS mainly performs two operations on processes — creation and termination — along with scheduling.

Process creation: a parent process creates child processes, forming a tree. In UNIX, **fork()** creates a child that is a copy of the parent, and **exec()** then loads a new program into it. At creation the parent may share resources with the child or give it separate resources, and may execute together with it or wait for it to finish.

Process termination: a process ends by calling **exit()** after finishing its work, returning a status to its parent, which collects it with **wait()**. The OS then takes back its memory, files and other resources. A process may also be terminated by its parent, or by the OS for errors such as exceeding time limits, memory shortage, or a protection violation. If a parent dies first, its children may be adopted or terminated (“cascading termination” in some systems).

CPU scheduling

CPU scheduling is the basis of multiprogramming: whenever the CPU becomes idle — a process finishes, waits for I/O, or its time slice ends — the OS selects the next process from the ready queue. The short-term scheduler (CPU scheduler) does this very frequently; the **dispatcher** then performs the context switch, changes to user mode and jumps to the correct location to resume that process.

- **Preemptive scheduling** — the OS can take the CPU away from a running process (needed for time-sharing).
- **Non-preemptive scheduling** — a process keeps the CPU until it finishes or blocks voluntarily.

Criteria for judging an algorithm: CPU utilisation (keep CPU busy), throughput (processes completed per unit time), turnaround time (submission to completion), waiting time (time spent in the ready queue) and response time (time to the first response). A good algorithm keeps utilisation and throughput high and the times low.

Scheduling algorithms

1. FCFS (First Come, First Served), non-preemptive: run processes in arrival order. Waiting time = start time – arrival time; average waiting time is the mean for all processes. Simple, but a long job at the front makes short jobs wait (convoy effect).

2. SJF (Shortest Job First): run the process with the smallest next CPU burst. Gives the minimum average waiting time. Can be non-preemptive or preemptive (then called SRTF — shortest remaining time first). Weakness: exact burst time is unknown and is only predicted, and long jobs may starve.

3. Round Robin (preemptive): give each ready process a fixed **time quantum** (say 10–100 ms) in circular order; if it does not finish, move it to the end of the queue. Fair for time-sharing; too small a quantum causes many context switches, too large behaves like FCFS.

4. Priority scheduling: run the process with the highest priority (usually the smallest priority number). May be preemptive or non-preemptive. Problem: low-priority processes may starve; solution is **ageing** — gradually raise the priority of a waiting process. In numericals, draw the Gantt chart first, then calculate waiting and turnaround times from it.

IPC (inter-process communication)

Processes are **independent** if one cannot affect another, and **cooperating** if they work together and affect each other. Cooperating processes need **IPC — inter-process communication** — to exchange data, share information, speed up computation by dividing a task, and build systems in a modular way.

Two fundamental IPC models:

- **Shared memory** — a region of memory is shared by the processes; they read and write it directly. Fast, because the kernel is involved mainly in setting it up, but processes must avoid writing at the same time.
- **Message passing** — processes send and receive messages through the kernel using `send()` and `receive()`. Slower, but no shared data, so no conflict; useful also between different computers. Communication may be direct (name the other process) or indirect (through a mailbox/port), and blocking (synchronous) or non-blocking (asynchronous).

Examples: pipes in UNIX carry a stream of bytes from one process to another; sockets are used for communication over a network.

Process synchronisation

When several cooperating processes access the same shared data at the same time, the result can depend on the exact order of execution. This unwanted situation is called a **race condition** — for example, two processes reading a counter value of 5, both adding 1, and both writing back 6 instead of 7.

Process synchronisation is the set of techniques that controls the order of execution so that shared data stays correct. The main tool is to allow only one process at a time into the code that touches shared data.

Common mechanisms in brief:

- **Mutex lock** — a lock that a process must acquire before entering and release after leaving; only the holder can proceed.
- **Semaphore** — an integer variable accessed only by two atomic operations, `wait()` (P) and `signal()` (V). A binary semaphore (value 0/1) works like a mutex; a counting semaphore controls access to a fixed number of identical resources.

- **Monitor** — a high-level construct in which only one process is active inside at a time, so the programmer need not code locks separately.

Critical section

The part of a program where a process accesses shared data (a common variable, table or file) is called its **critical section**. The problem is to design a protocol so that no two processes are in their critical sections at the same time.

Each process follows this structure: **entry section** (request permission) → critical section → **exit section** (release) → remainder section (other code).

Any correct solution must satisfy three requirements:

- **Mutual exclusion** — if one process is in its critical section, no other process may enter its own.
- **Progress** — if no process is inside and some want to enter, the choice of who enters next cannot be postponed forever, and processes outside must not block it.
- **Bounded waiting** — after a process requests entry, there is a limit on how many times others may enter before it, so no process waits forever (no starvation).

These three lines are asked directly; learn them word for word.

Deadlock — condition, avoidance and prevention

A **deadlock** is a situation in which two or more processes are each waiting for a resource held by another, so none can proceed. Deadlock can occur only if all **four conditions** hold together: **(1) Mutual exclusion** — a resource can be used by only one process at a time; **(2) Hold and wait** — a process holds one resource while waiting for another; **(3) No preemption** — a resource is released only voluntarily by its holder; **(4) Circular wait** — P1 waits for P2, P2 for P3, and P3 for P1.

Prevention means breaking at least one condition in advance: request all resources at once or release held ones before a new request (breaks hold and wait), allow resources to be taken back (breaks no preemption), or impose a fixed ordering for requesting resources (breaks circular wait). Prevention is safe but can waste resources.

Avoidance allows requests but grants one only if the resulting state is **safe** — a state from which every process can still finish in some order. The **Banker's algorithm** does this: before granting a request it pretends to grant it, then checks whether a safe sequence exists using Available, Max, Allocation and Need ($\text{Need} = \text{Max} - \text{Allocation}$); if no safe sequence exists, the process must wait.

Likely question: Processes P1, P2 and P3 arrive together with CPU bursts of 24, 3 and 3 ms. Draw Gantt charts and find the average waiting time using FCFS, SJF and Round Robin (quantum 4 ms). Also state the four conditions necessary for deadlock.

Unit 3 — Memory Management (10 Periods, 25%)

Topics: Memory allocation; First Fit, Best Fit and Worst Fit; swapping; paging and segmentation; page faults; page replacement algorithms; fragmentation and compaction; concept of virtual memory.

Memory allocation

Memory management keeps track of which parts of main memory are in use and by whom, allocates memory to processes when they need it, and frees it when they finish, so that several processes can reside in memory together.

In **contiguous allocation**, each process gets one continuous block. With **fixed partitions**, memory is divided in advance into equal or unequal blocks — simple, but a block may be partly wasted. With **variable (dynamic) partitions**, a block of exactly the needed size is carved out when a process arrives and freed when it leaves, leaving holes of many sizes scattered in memory.

The OS maintains a list of allocated blocks and free holes. When a request comes, it must choose a hole using a placement strategy — First Fit, Best Fit or Worst Fit (next point). Allocation must also provide **protection**: base (relocation) and limit registers ensure a process can access only its own block.

First Fit, Best Fit and Worst Fit

These are three strategies for choosing a free hole for a process in dynamic partitioning:

- **First Fit** — allocate the **first** hole that is large enough, searching from the beginning (or from where the last search stopped). Fastest, because searching stops early.
- **Best Fit** — allocate the **smallest** hole that is large enough. Search the whole list unless an exact fit is found. It leaves the smallest leftover hole, but that leftover is often too small to use, and searching takes more time.
- **Worst Fit** — allocate the **largest** hole, again searching the whole list. The idea is that the leftover hole stays big enough to hold another process.

How to solve a numerical: draw the holes in order, place the process by the given rule, show the remaining hole, and repeat. Studies and exam answers generally conclude that First Fit and Best Fit perform better than Worst Fit; First Fit is usually fastest. Practise one example with holes like 100 KB, 500 KB, 200 KB, 300 KB and 600 KB for requests of 212 KB, 417 KB and 112 KB.

Swapping

Swapping is a memory-management technique in which a process is temporarily moved out of main memory to a backing store (disk) and brought back in later for continued execution. The area of the disk used for this is called swap space.

Why it is needed: the total memory needed by all processes can exceed physical memory. Swapping lets the system keep more processes ready and improves CPU use, at the cost of disk transfer time.

- **Swap out** — the OS copies a process from RAM to disk, normally a low-priority or blocked process.
- **Swap in** — when its turn comes, the process is copied back into RAM; in variable partitions it may return to a different location, handled by dynamic relocation.

The major cost is **swap time**, which depends mainly on the amount of data transferred and disk speed. If a process is waiting for I/O, care is needed — either I/O must be done through OS buffers, or the process must not be swapped out until the I/O completes.

Paging and segmentation

Paging divides physical memory into fixed-size blocks called **frames** and divides the process (logical memory) into blocks of the same size called **pages**. Pages are loaded into any free frames, so a process need not be contiguous and external fragmentation disappears. A **page table** maps each page to its frame; $\text{logical address} = \text{page number} + \text{page offset}$. Page size is a power of 2, commonly 4 KB. Paging can cause internal fragmentation in the last page.

Segmentation divides the process into variable-size **segments** matching the logical parts of a program — code, data, stack, functions. A **segment table** stores for each segment its base (starting address) and limit (length); $\text{logical address} = \text{segment number} + \text{offset}$, and the offset must be less than the limit. Segmentation matches the programmer's view and allows sharing and protection per segment, but suffers external fragmentation.

One line to remember: paging is for the convenience of the OS, segmentation is for the convenience of the programmer. Many systems combine both.

Page faults

A **page fault** is not an error by the user — it is an event that occurs when a process tries to access a page that is not currently in main memory. Each page-table entry has a valid-invalid bit; access to an invalid page traps to the OS.

Steps in handling a page fault:

- The process references a page marked invalid; hardware traps to the OS.
- The OS checks the reference is legal; if not, it terminates the process.
- It finds the needed page on disk and a free frame (or frees a frame by page replacement, writing back a dirty victim page if needed).
- It schedules the disk read to bring the page into the frame.
- It updates the page table, marks the entry valid, and restarts the instruction that caused the fault.

Because disk access is far slower than memory access, too many page faults badly reduce performance; if faulting becomes excessive the system is said to be **thrashing**.

Page replacement algorithms

When no frame is free, the OS must choose a **victim page** to remove. Page faults are counted on a reference string; the algorithm with fewer faults is better.

1. FIFO (First In, First Out): replace the page that has been in memory the longest (oldest loaded). Simple queue method. Note: adding more frames can sometimes increase faults in FIFO — **Belady's anomaly**.

2. LRU (Least Recently Used): replace the page that has not been used for the longest time in the past. It uses locality well and generally gives fewer faults than FIFO, but needs extra hardware or counters/stack to track use.

3. Optimal (OPT / MIN): replace the page that will not be used for the longest time in the future. It gives the **minimum possible** faults and is used as a benchmark, but it cannot be implemented in practice because future references are unknown.

Numerical method: draw one column per reference, show frame contents, mark each fault, and total the faults at the end.

Fragmentation and compaction

Fragmentation means memory is wasted because free space exists but cannot be used properly. It is of two types:

- **Internal fragmentation** — allocated block is larger than the process needs, so space inside the block is wasted. Occurs in fixed partitions and in the last page of paging.
- **External fragmentation** — total free memory is enough for a request, but it is scattered in small holes that are not contiguous, so no single hole can hold the process. Occurs in variable partitioning and segmentation.

Compaction is the cure for external fragmentation: move all used blocks together to one side so that all free holes join into one large block. It is possible only with dynamic relocation, and it is costly because running processes must be stopped and copied — hence it is done only when necessary. The well-known 50-percent rule states that, with First Fit, roughly one block is lost to fragmentation for every two allocated blocks.

Concept of virtual memory

Virtual memory is a technique that allows a process to execute even when it is not completely in main memory. The OS keeps only the currently needed pages in RAM and the rest on disk, loading pages on demand (**demand paging**). To the programmer, memory appears much larger than physical RAM.

Advantages:

- Programs larger than physical memory can run.
- More processes can run at the same time, improving CPU and memory use.

- Less I/O is needed to load a process, so response is faster, and logical memory is separated from physical memory.

It is implemented mainly with demand paging (sometimes combined with demand segmentation), using the page table, valid-invalid bits, page faults and page replacement already studied. The danger is **thrashing** — when the system spends more time swapping pages than executing processes, because too many processes compete for too few frames. Virtual memory is the reason a computer with 8 GB RAM can smoothly run programs whose total size is much larger.

Likely question: For the reference string 7, 0, 1, 2, 0, 3, 0, 4 with three frames, find the number of page faults using FIFO, LRU and Optimal replacement. Also differentiate between paging and segmentation.

Unit 4 — I/O System (8 Periods, 18%)

Topics: Mass storage structure — overview; disk structure; disk attachment; disk scheduling algorithms; swap space management; RAID types.

Mass storage structure — overview

Mass storage (secondary storage) holds data permanently in large amounts — the main devices are magnetic hard disks, solid-state disks (SSDs) and magnetic tapes. Main memory is too small and volatile to store all programs and data, so the OS depends on mass storage for files, swap space and backup.

- **Hard Disk Drive (HDD)** — data is stored magnetically on rotating platters; it has mechanical moving parts (arm and heads), so access includes seek and rotation delays, but cost per GB is low.
- **Solid-State Drive (SSD)** — uses flash memory, has no moving parts, is faster, quieter and more shock-resistant, but costs more per GB and has a limited number of writes.
- **Magnetic tape** — very cheap and high capacity, but access is sequential and slow, so it is used mainly for backup and archiving.

The OS presents all these devices to programs as simple files, hiding their different speeds and structures. Performance of this layer strongly affects overall system performance because it is the slowest part of the computer.

Disk structure

A hard disk consists of circular **platters** coated with magnetic material, rotating together on a spindle. Each platter surface has concentric circles called **tracks**; each track is divided into **sectors** (commonly 512 bytes or 4 KB), which are the smallest unit of transfer. The set of tracks at the same position on all surfaces is a **cylinder**. A movable arm carries a read-write **head** just above each surface.

Disk access time has three parts: **seek time** (move the arm to the correct track), **rotational latency** (wait for the sector to come under the head — on average half a rotation), and **transfer time** (actually read or write the data).

The disk is addressed as a one-dimensional array of **logical blocks**; the OS maps logical block numbers to (cylinder, track, sector) — sector 0 is the first sector of the first track of the outermost cylinder. Disks may be fixed or removable, and formatting creates the file-system structures on them.

Disk attachment

Disks are attached to a computer in two main ways:

- **Host-attached storage** — the disk is connected directly to the local computer through an I/O bus and a host controller, using interfaces such as SATA, SCSI or USB. The computer issues I/O commands through ports; this is simple and is used for internal and external drives of a PC.
- **Network-attached storage (NAS)** — storage is attached to a network and clients access it over the network using file-sharing protocols, as remote files rather than as local blocks. It is convenient for sharing in offices, but is generally slower than direct attachment and depends on network load.

A third arrangement, a **storage-area network (SAN)**, is a dedicated private network of storage devices used in large data centres; servers see the storage as if it were local. In an exam answer, state the method, how data travels, one advantage and one limitation for each type. The **device driver** and controller together hide device details from the OS kernel.

Disk scheduling algorithms

Many processes may request disk I/O together. **Disk scheduling** decides the order of servicing requests waiting for different cylinders, mainly to reduce total seek time. Assume requests (in queue order) for cylinders 98, 183, 37, 122, 14, 124, 65, 67 with the head at 53 — a standard practice example.

1. **FCFS**: serve requests in arrival order. Fair — no starvation — but the head swings widely, so total head movement is large (about 640 cylinders in the example).
2. **SCAN (elevator algorithm)**: the head moves in one direction serving all requests on the way, reaches the end of the disk, then reverses and serves while returning — like a lift. Movement in the example is much less (about 236). Requests at the far end may wait longer.
3. **C-SCAN (Circular SCAN)**: like SCAN, but after reaching the end the head returns to the beginning without serving on the way back, then starts serving again. This gives more uniform waiting time for all cylinders.

For numericals: draw the head-movement path, add the differences between successive positions, and report the total head movement for each algorithm.

Swap space management

Swap space is the disk area used as an extension of main memory: pages and whole processes that do not fit in RAM are kept there (see swapping and page faults in Unit 3). Because disk is much slower than RAM, how swap space is organised directly affects performance.

- **Location** — swap space may be a separate disk partition (raw, faster, used only by the OS) or a normal file inside a file system (easier to create and resize, but slower and subject to fragmentation).
- **Size** — it must be large enough to hold the pages of processes that may be swapped; too little causes out-of-memory failures, while too much wastes disk.
- **Management** — the OS keeps a swap map recording which swap blocks are free and which belong to which process or page. Some systems swap complete processes; demand-paging systems swap individual pages, often copying a page to swap space the first time it is written out so later write-backs are faster.

Modern systems reduce swapping with larger RAM but still keep swap space for inactive pages and safety.

RAID types

RAID (Redundant Array of Independent Disks) combines several physical disks into one logical unit to improve performance, reliability, or both. Data distribution across disks is called **striping**; copying data is **mirroring**; extra check information is **parity**.

- **RAID 0 — Striping:** data split across disks; fastest read/write and full capacity, but no redundancy — failure of one disk loses data.
- **RAID 1 — Mirroring:** same data on two disks; excellent safety and read speed, but only half the capacity is usable.
- **RAID 2 and 3** — bit/byte-level striping with error-correcting or parity disks; rarely used today, mention only in brief.
- **RAID 4** — block-level striping with one dedicated parity disk; the parity disk becomes a bottleneck.
- **RAID 5 — Striping with distributed parity:** blocks and parity are spread over all disks (minimum three); survives one disk failure with reasonable capacity — very popular.
- **RAID 6** — like RAID 5 but with two distributed parities; survives two disk failures, at extra cost.

RAID improves reliability but is not a substitute for backup.

Likely question: Disk requests arrive for cylinders 98, 183, 37, 122, 14, 124, 65, 67 and the head is at 53. Find the total head movement using FCFS, SCAN and C-SCAN, and write short notes on RAID 0, RAID 1 and RAID 5.

Unit 5 — File Management (8 Periods, 18%)

Topics: Concept of a file; access methods; directory structure; file system mounting; file sharing and protection; file system structure and implementation; directory implementation; free-space management; efficiency and performance; different types of file systems.

Concept of a file

A **file** is a named collection of related information recorded on secondary storage — it is the smallest unit of storage from the user's point of view. Programs, documents, images and data are all stored as files, and the OS hides the physical details of disks behind this simple idea.

File attributes (kept in the directory): name, identifier, type, location on disk, size, protection permissions, and dates/times of creation, modification and last access, along with owner information.

File operations: create, write, read, reposition (seek), delete and truncate (shorten while keeping the file). An **open-file table** is maintained by the OS for files in use.

File types are often shown by the extension: .exe (executable), .txt (text), .pdf (document), .jpg (image), .c (C source). Internally, a file may be structured as a simple sequence of bytes (used by UNIX/Linux and Windows), a sequence of fixed-length records, or a tree of records.

Access methods

Access methods define how the information stored in a file is reached:

- **Sequential access** — information is processed in order, one record after another, from the beginning. Reading is automatic forward movement; rewinding is possible but skipping ahead is not convenient. It suits tapes and files processed as a whole, such as payroll files. Operations: read next, write next, rewind.
- **Direct (random) access** — any block can be read or written in any order, directly by its block number, with no need to pass through earlier records. It is based on the disk model and suits databases, where a particular record is needed quickly. Operations include read(n) and write(n), where n is the relative block number.
- **Indexed access** — an index, like a book index, holds pointers to blocks; to find a record, search the index first and then go directly to the data. Used for very large files, sometimes with multiple index levels.

In answers, always pair each method with its suitable device and one example use.

Directory structure

A **directory** is a collection of nodes containing information about files — it organises files so they can be found, named, shared and protected. Directory operations are: search for a file, create, delete, rename, list and traverse the file system.

Common structures, in increasing order of usefulness:

- **Single-level directory** — one directory for all users; simple, but naming conflicts occur and searching is slow when files are many.
- **Two-level directory** — a master directory holds a separate user directory for each user; users can use the same file name, but groups cannot share easily.
- **Tree-structured directory** — directories contain files and sub-directories in a tree, using absolute and relative path names; this is the structure used today.
- **Acyclic-graph directory** — allows shared files and directories to appear in more than one place using links, while keeping no cycles.
- **General graph directory** — also allows cycles, but needs care to avoid infinite searching and uses garbage collection.

File system mounting

Before a file system can be used, it must be **mounted**: the OS attaches it to a point in the existing directory tree, called the **mount point**, so its files become accessible by normal path names.

How mounting works:

- The user or system supplies the device name and the mount point (typically an empty directory, for example /mnt/usb).
- The OS verifies that the device contains a valid file system, by reading its boot/control structures.
- It records the mounted file system in its mount table and links the root of the new file system at the mount point.

After mounting, opening a path under the mount point automatically goes to that device. On removal, the file system is **unmounted** — pending data is written back and the mount point becomes an ordinary empty directory again. In Windows, mounted drives appear as drive letters (C:, D:); in Linux/UNIX they appear inside one single tree. Mounting is also used for network file systems.

File sharing and protection

File sharing allows several users to use the same file, which is essential in multi-user systems but must be controlled. Sharing may be through a common directory, links to one physical file, or remote file systems over a network. The system must handle issues such as multiple simultaneous users, consistency of changes, and remote access.

Protection controls who may do what with a file. Need: prevent unauthorised reading, modification or deletion, and protect the system from faulty programs.

- **Access types** — read, write, execute, append, delete and list.
- **Access control list (ACL)** — with each file, store the users allowed and their permissions; precise but can become long.
- **Owner-group-others method (UNIX)** — permissions are given as rwx sets for the file owner, its group and all others, for example `rw-r--r--`.
- **Passwords** — a file or directory may be protected by a password, though one password gives all-or-no access.

Sharing without protection is unsafe; protection without sharing makes a multi-user system weak — a good file system provides both.

File system structure and implementation

A file system is implemented in **layers**, each using the features of the layer below it:

- **Logical file system** — manages metadata and directory structure, and uses **File Control Blocks (FCBs)** — one per file, storing permissions, dates, owner, size and the locations of data blocks.
- **File-organisation module** — translates logical block numbers to physical block numbers and manages free space and allocation.
- **Basic file system** — issues generic read/write commands for physical blocks to the device driver and manages buffers and caches.
- **I/O control** — device drivers and interrupt handlers that actually transfer data between memory and disk.

On-disk structures: boot block (starts the system), volume control block (volume size, free blocks), directory structure and FCBs. **In-memory structures:** mount table, directory cache, system-wide and per-process open-file tables. **File allocation methods** in brief: contiguous (fast, but external fragmentation), linked (each block points to the next; no external fragmentation, slow random access) and indexed (an index block holds all pointers; fast access, extra space for the index).

Directory implementation

The **directory** must be implemented so that searching for a name, adding a file and deleting a file are all fast. Two main methods are used:

- **Linear list** — the directory is a simple list of entries, each with a file name and a pointer to its FCB or data blocks. Creating a file needs a search to avoid duplicate names; deleting frees an entry (marked unused, or filled by the last entry). It is easy to program but searching is slow when the directory is large, because the whole list may be scanned.
- **Hash table** — a hash function on the file name gives the position of its entry, so lookup is very fast. Problems: collisions (two names giving the same position, handled by chained overflow), and the fixed size of the table.

In practice, systems combine methods — for example, a hash table with chaining, plus a **cache** of recently used directory entries so repeated searches become almost instant. Directory entries in UNIX store the name and an inode number; the inode itself holds all attributes and block pointers.

Free-space management

The OS must know which disk blocks are free so it can allocate them to new or growing files. The free-space list is stored on disk itself and is managed by these methods:

- **Bit vector (bitmap)** — one bit per block: 1 means free, 0 means allocated. Simple, and finding a group of contiguous free blocks is easy, but the vector itself takes space (a 1 TB disk with 4 KB blocks needs 32 MB).
- **Linked list** — each free block contains the address of the next free block; the OS keeps a pointer to the first. No space is wasted, but random searching is impractical.
- **Grouping** — the first free block stores the addresses of many free blocks, the last of which points to another such block; many free addresses are found quickly.
- **Counting** — because free blocks often occur in contiguous runs, store (starting address, count of free blocks) pairs in a small table; entries are fewer and large runs are found fast.

Efficiency and performance

File-system **efficiency** depends mainly on the allocation method, the type of directory structure, and how much metadata is stored with each file — storing unnecessary data, such as last-access time updated on every read, increases disk traffic. Choosing the right block size is also important: too small means many blocks and pointers per file; too large wastes space on small files.

Performance is improved by:

- **Disk cache / buffer cache** — a part of RAM holding recently used blocks, so many reads need no disk access at all.
- **Read-ahead** — on sequential reading, the next blocks are fetched in advance because they will probably be needed.
- **Reducing arm movement** — placing FCBs/inodes and data blocks close together and using good disk scheduling.
- **Defragmentation and careful allocation** — keeping a file's blocks near each other.
- **Asynchronous writes and journaling** — writes are reported complete early using cache, while a log (journal) allows quick recovery after a crash without scanning the whole disk.

Summary line: efficiency is about not wasting space or work; performance is about giving data fast, mostly by good caching and layout.

Different types of file systems

Different operating systems and devices use different **file systems**. Common types to name in an answer:

- **FAT / FAT32** — simple and supported almost everywhere; used on pen drives and memory cards. Limitations: maximum file size of 4 GB in FAT32 and almost no protection features.
- **NTFS** — the standard Windows file system; supports large files, permissions, encryption, compression and journaling for recovery.
- **ext3 / ext4** — the standard Linux file systems; ext4 supports very large volumes, journaling and better performance.
- **UNIX File System (UFS) and Network File System (NFS)** — UFS uses inodes; NFS allows directories on a remote machine to be mounted and used like local ones.
- **ISO 9660 / UDF** — used on CDs and DVDs.
- **Temporary and special file systems** — such as a RAM-disk file system kept entirely in memory for speed, and /proc in Linux, which presents process information as files.

While choosing a file system, consider compatibility, maximum file and volume size, security features and recovery support.

Likely question: Explain the different directory structures with diagrams. How is free space managed in a file system? Write a short note on file sharing and protection.

Quick Revision — One Line Each

- OS is the interface between user, applications and hardware; its goals are convenience and efficiency.
- Batch, time-sharing, real-time and distributed are the main types of OS.
- Kernel is the resident core of the OS; monolithic is fast, microkernel is more reliable.
- A system call is the gate through which a program requests a kernel service.
- A process is a program in execution; its full record is kept in the PCB.
- FCFS is simplest; SJF gives minimum average waiting time; Round Robin uses a time quantum.
- Critical section solution needs mutual exclusion, progress and bounded waiting.
- Deadlock needs all four: mutual exclusion, hold and wait, no preemption, circular wait.
- Banker's algorithm grants a request only if the resulting state is safe.
- Paging removes external fragmentation; segmentation follows the programmer's view of the program.
- FIFO can show Belady's anomaly; Optimal gives minimum faults but is impractical; LRU uses past use.
- SCAN is the elevator algorithm; C-SCAN returns to the start without serving on the way back.
- RAID 0 is striping only, RAID 1 is mirroring, RAID 5 is striping with distributed parity.

