ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3583

Program: PGD(CS)

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Operating System Concepts (3583) • Level: PGD(CS)

Semester: Autumn 2026

Assignment No. 2

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant parts of the course book, understand the concepts, try the worked scheduling and disk examples themselves on paper, and then write the assignment in their own words. Copying this guide word for word is not recommended. No units are printed in the question booklet for this course, so none are shown on this title page.

USMAN NOTES www.usmannotes.com

Question 1: Disk Scheduling — SCAN, LOOK and SSTF (20 Marks)

Question as printed: What is disk scheduling? Compare SCAN, LOOK, and SSTF algorithms.

Introduction

Of all the resources an operating system manages, the disk is the slowest by far: a mechanical hard disk takes milliseconds to reach data, while the CPU works in nanoseconds — a difference of millions of times. When several processes ask for disk data at once, requests queue up, and the order in which the operating system serves that queue decides how far the disk's read–write head must travel and therefore how long everyone waits. Choosing that order is disk

scheduling. This answer explains disk scheduling and the hardware reasons behind it, then describes and compares three important algorithms — Shortest Seek Time First (SSTF), SCAN and LOOK — using one fully worked example so the comparison rests on numbers, not adjectives.

What Is Disk Scheduling?

Disk scheduling is the operating system's method of ordering pending disk I/O requests so as to use the disk efficiently. To understand why order matters, consider how a hard disk works. Data lives on rotating platters in concentric tracks; a movable arm positions the read–write head over the required track (the seek), the platter then rotates until the required sector passes under the head (rotational latency), and only then can data flow (transfer time). Of these, seek time — the time to move the arm — dominates, and it grows with the distance travelled. Disk scheduling therefore aims, above all, to minimise total seek distance, while also keeping response fair so that no request waits forever. When requests arrive in a bad order, the head thrashes back and forth across the disk; a good algorithm rearranges the queue (something a CPU scheduler cannot do as freely, because disk requests have no urgency differences of the same kind) into an order that sweeps efficiently.

For comparison, the simplest possible policy is First-Come First-Served: serve requests in arrival order. FCFS is perfectly fair but performs badly — in the worked example below it travels 640 tracks where a smarter order needs only 236. The three algorithms of this question each improve on FCFS in a different way: SSTF by greedily taking the nearest request, SCAN by

sweeping like a lift, and LOOK by sweeping without wasting travel to the physical ends of the disk.

Shortest Seek Time First (SSTF)

SSTF examines the pending queue and always selects the request closest to the head's current position — the request needing the shortest seek. It is the disk analogue of Shortest Job First in CPU scheduling, and like SJF it is greedy: each individual choice is the best available now. SSTF substantially reduces total head movement compared with FCFS and is simple to implement. Its weakness is the same as SJF's: starvation. If new requests keep arriving near the head's current region, requests far away may be postponed indefinitely, and average response is unfair between the middle and the edges of the disk — requests near the current sweep are served quickly, distant ones slowly. SSTF is therefore attractive only where arrival rates are modest.

SCAN (The Elevator Algorithm)

SCAN removes starvation by imposing a discipline: the disk arm starts at one end of the disk and moves towards the other end, servicing every request in its path, like a lift that keeps going in one direction and stops at every floor where someone is waiting. When it reaches the last track it reverses direction and sweeps back, servicing requests on the return journey. No request can be starved, because the head is guaranteed to reach every track on every sweep; waiting is, however, uneven in a subtle way — requests just missed by the passing head at the start of a sweep wait almost a full double sweep, while requests near the far end are served quickly on arrival. A variant, C-SCAN (circular SCAN), returns the head directly to the beginning after one pass — treating the disk as circular — which makes waiting time more uniform.

LOOK and C-LOOK

LOOK is SCAN with wasted motion removed. In SCAN the head travels all the way to the last physical track even if no request is waiting there; in LOOK — the name comes from the arm 'looking' ahead for requests — the head reverses direction as soon as there are no more requests in the current direction, at the last requested track rather than at the end of the disk. C-LOOK applies the same idea to the circular version: sweep in one direction to the last request, then jump straight back to the first request and sweep again. LOOK and C-LOOK keep all of SCAN's fairness while always travelling less, which is why real disk schedulers (including the historical Linux elevator and its descendants) are based on the LOOK family.

Worked Example: One Queue, Three Algorithms

Take the classic example: requests are pending for tracks 98, 183, 37, 122, 14, 124, 65 and 67 (in that arrival order), the disk has tracks 0–199, and the head starts at track 53, initially moving towards higher track numbers. Figure 1 plots the head's journey under each algorithm.

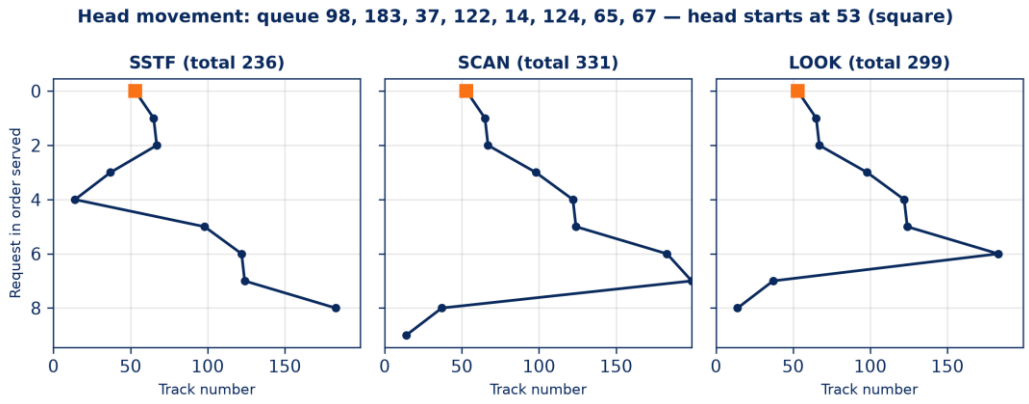


Figure 1: Head movement for SSTF, SCAN and LOOK on the same request queue (head starts at track 53).

Under SSTF the head serves 65, 67, 37, 14, 98, 122, 124 and 183: each step chooses the nearest waiting request, and the total movement is 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59 = 236 tracks. Under SCAN the head continues upwards serving 65, 67, 98, 122, 124 and 183, travels on to the physical end at 199, then reverses and serves 37 and 14: total = 331 tracks, the price of the pointless trip from 183 to 199 and the long return. Under LOOK the head serves the same upward requests but reverses immediately at 183, the last request in that direction, and then serves 37 and 14: total = 299 tracks — 32 tracks saved over SCAN simply by not visiting the empty end of the disk. For reference, plain FCFS on this queue travels 640 tracks, which shows how much all three algorithms matter.

Comparison of SCAN, LOOK and SSTF

Table 1: SSTF versus SCAN versus LOOK

Criterion	SSTF	SCAN	LOOK
Basic rule	Always serve the request nearest to the head	Sweep end to end, serving in passing, then reverse	Like SCAN, but reverse at the last request, not the disk end
Total movement in worked example	236 tracks (best)	331 tracks	299 tracks
Starvation possible?	Yes — distant requests may wait indefinitely	No — every track is visited each sweep	No — same guarantee as SCAN
Fairness	Favours requests near the head's current position	Fair overall; requests just missed wait longest	Fair, with less wasted travel than SCAN

Overhead / complexity	Must search the queue for the nearest request each time	Simple: keep a direction and the sorted queue	Simple: SCAN plus a check for pending requests ahead
Weakness	Unfair under heavy, clustered load	Wastes travel to the physical ends of the disk	Slightly more logic than SCAN; still no priority for urgent requests

The table and the example together tell a balanced story. SSTF wins the raw distance contest — 236 tracks — because it never makes a long journey while a short one is available, but it pays for that greed with the possibility of starvation and uneven service. SCAN is the fairest simple policy and cannot starve anyone, but wastes movement reaching empty ends of the disk. LOOK keeps SCAN's guarantees and trims the waste, and therefore dominates SCAN in practice: any system that would use SCAN should use LOOK instead. Modern systems add further ideas — deadlines so no request waits too long, and merging of adjacent requests — but all of them stand on the foundation these three algorithms laid.

Conclusion

Disk scheduling orders queued disk requests to minimise the seek-dominated cost of serving them. SSTF greedily takes the nearest request and achieves the least head movement (236 tracks in the example) but can starve distant requests. SCAN sweeps the disk like a lift, guaranteeing service to all (331 tracks), while LOOK improves SCAN by reversing at the last outstanding request instead of the disk's physical end (299 tracks) with the same freedom from starvation. The comparison shows the recurring operating-system trade-off between pure

efficiency and fairness — and why the LOOK family, balancing both, is what practical disk schedulers use.

Question 2: I/O Management and Spooling (20 Marks)

Question as printed: Explain I/O management in operating systems. What is spooling?

Introduction

A computer without input and output would be a sealed box: it could compute, but nothing could enter and no result could leave. Yet I/O is the hardest part of the operating system to manage, because devices are slow, unreliable, wildly different from one another and shared by competing processes. The operating system's I/O management subsystem exists to hide that chaos — to give every program a simple, uniform way of using keyboards, disks, printers and networks while keeping the devices busy and the CPU free. This answer explains the goals and components of I/O management, the techniques by which the CPU communicates with devices, and then explains spooling, one of its most successful ideas, in detail.

The Problem I/O Management Solves

Three facts make I/O management necessary. First, speed mismatch: a disk is millions of times slower than the CPU, so a processor that waited idle for every device would waste almost its entire life; I/O must overlap with computation, letting the CPU work on other processes while devices operate. Second, diversity: a keyboard delivers a few bytes per second from a human, a network card delivers millions of bytes per second in packets, a printer consumes pages — yet applications want one consistent interface ('read', 'write', 'open', 'close') for all of them. Third,

sharing and protection: devices must be allocated safely among processes, errors must be reported and handled, and no process may seize a device directly. The I/O subsystem meets these needs with a layered design described next.

Components of the I/O Subsystem

At the top, applications see system calls and a device-independent software layer: this layer offers uniform naming and buffering for all devices, enforces protection, and presents the same read/write interface whether the target is a file, a terminal or a printer. Below it sit the device drivers — one for each type of hardware — which translate the uniform commands into the device's private language (Question 3 is devoted to them). Below the drivers, interrupt handlers connect the hardware to the kernel: when a device finishes an operation it raises an interrupt, and the handler wakes the waiting driver or process. At the bottom is the hardware itself: controllers, ports and direct-memory-access (DMA) engines. Between the user and this machinery, the kernel adds two performance aids: buffering, which keeps data in memory to absorb speed differences and to assemble small transfers into efficient large ones, and caching, which keeps copies of recently used data in memory to avoid device access altogether.

How the CPU Talks to Devices: Three Techniques

Historically, three techniques have been used, each freeing the CPU further. In programmed I/O, the CPU itself repeatedly tests the device's status register ('busy-waiting' or polling) until the device is ready, then moves data one unit at a time — simple, but it burns CPU time doing nothing. In interrupt-driven I/O, the CPU starts the transfer and goes away to other work; the device interrupts when each unit (or block) is ready, and the CPU moves the data then. This is

efficient for slow and medium devices. For high-speed devices, direct memory access (DMA) goes further: a DMA controller is given the source, destination and count, and transfers the whole block between the device and memory by itself, interrupting the CPU only once — when everything is done. A modern disk read therefore costs the CPU two brief moments of attention, at the start and at the end, however large the data.

What Is Spooling?

Spooling — the name is an acronym of Simultaneous Peripheral Operations On-Line — is a technique in which the output (or input) for a device that can serve only one job at a time is first written to a reserved area on disk, the spool, and the device then works through the spooled jobs one by one at its own pace. The classic application is the printer. A printer can print only one document at a time and prints slowly, but many processes may 'print' simultaneously. With spooling, when a process prints, its output is not sent to the printer at all: it is written quickly to a spool file on disk, the process is told 'printing done' and continues its work immediately, and a background system process (the spooler or print daemon) feeds the spooled files to the real printer one at a time, in order.

Spooling achieves three things at once. It converts a device that cannot be shared into one that appears shareable: every process gets the illusion of its own printer. It overlaps computation with I/O perfectly: no process ever waits for the slow device, only for the fast disk. And it orders and protects the work: documents can never interleave page by page (as they would if several processes wrote to the printer directly), and the queue can be managed — jobs can be listed, held, prioritised or cancelled. The same principle appears in many places: mail servers spool

outgoing messages, batch systems spool jobs, and network print servers are spoolers. It is worth distinguishing spooling from ordinary buffering: a buffer merely holds data temporarily between two stages of one transfer, while a spool is a complete queue of whole jobs held on disk for a device that serves one job at a time.

One more design goal ties these parts together: error handling and fairness. Devices fail — paper jams, sectors go bad, cables are unplugged — and the I/O subsystem, not the application, must detect failures, retry when sensible and report honest errors when not. It must also arbitrate fairly when many processes want the same device, just as the printer queue does. A well-designed I/O subsystem is therefore judged not only by speed but by how invisible it makes device trouble and device diversity to the programs above it.

Conclusion

I/O management is the operating system's layered machinery — system-call interface, device-independent layer, drivers, interrupt handlers, buffering, caching and DMA — for hiding the slowness and diversity of devices behind a uniform, protected and efficient interface. Spooling is its most elegant trick for unshareable devices: by writing each job to a disk spool and letting a daemon serve the device in turn, it lets many processes 'use' one printer at once, removes all waiting for the slow hardware, and keeps every job's output intact and in order.

Question 3: Device Drivers and Their Role in the Operating System (20 Marks)

Question as printed: What are device drivers? Explain their role in OS.

Introduction

Every piece of hardware speaks its own private language: one disk controller wants commands written to one set of registers, another expects a completely different protocol, and a new device may be invented years after the operating system was written. Applications cannot possibly know all these languages, and the kernel cannot contain code for every device that will ever exist. Device drivers solve this problem. A driver is the translator and manager for one specific device or family of devices, and it is through drivers that the operating system gains its famous ability to run on hardware its authors never saw. This answer defines device drivers, explains where they sit in the I/O system, details their roles and the functions they perform, and discusses their types and their risks.

What Is a Device Driver?

A device driver is a specialised software module — usually part of, or loaded into, the operating system kernel — that knows exactly how to control one particular device or device class: how to initialise it, how to command it, how to interpret its status and error reports, and how to move data to and from it. To the rest of the operating system, the driver presents the standard interface of the I/O subsystem ('open', 'read', 'write', 'ioctl', 'close'); to the hardware, it speaks the device's own register-level protocol. Every device attached to a computer — disk, keyboard, mouse, display, network card, printer, USB stick, webcam — has a driver, and a system with no driver for a device effectively does not have that device at all: the hardware may be present, but nothing can use it.

Where Drivers Sit in the I/O Architecture

The I/O software is layered, and the driver is the layer closest to the hardware. Above the driver, device-independent operating-system software handles everything that does not depend on the particular device: uniform naming, protection, buffering and the common system-call interface. The driver receives an abstract request from that layer ('read block 4512'), translates it into the concrete steps its controller demands (position the head, wait for the sector, start the DMA transfer), and reports completion back — usually by handling the interrupt the device raises when done. This layering is the point: above the driver, the operating system is written once for all devices; below it, only the driver changes from device to device.

The Role and Functions of Device Drivers

The driver's roles can be listed as follows. (1) Translation: converting the operating system's generic requests into the device's specific command sequences, register writes and timing. (2) Initialisation and configuration: resetting the device at startup, setting its mode, speed and parameters, and registering it with the kernel so applications can find it. (3) Data transfer: moving data between the device and memory, whether by programmed I/O, interrupt-driven transfer or by programming the DMA controller. (4) Interrupt handling: responding when the device signals completion or needs attention, waking the process that was waiting for the operation. (5) Error handling: detecting device errors — a bad disk sector, a printer out of paper — retrying where sensible and reporting honest errors upwards where not. (6) Power and state

management: in modern systems, putting idle devices to sleep and waking them, so laptops and phones survive the day. Table 1 summarises these roles.

Table 1: Roles of a device driver

Role	What the driver does	Example
Translation	Converts generic OS requests into device-specific commands	Turns 'read block 4512' into seek + wait + DMA commands for one disk model
Initialisation	Resets and configures the device; registers it with the kernel	Setting a network card's speed and MAC filters at boot
Data transfer	Moves data by programmed I/O, interrupts or DMA	Programming the DMA controller for a large file read
Interrupt handling	Services device interrupts and wakes waiting processes	Noting that a keystroke has arrived from the keyboard
Error handling	Detects, retries and reports device faults	Retrying a disk read, then reporting an I/O error to the application
Power management	Sleeps and wakes the device to save energy	Spinning down an idle hard disk in a laptop

Types of Drivers

Drivers are classified in several useful ways. Character-device drivers handle devices that transfer data as a stream of bytes — keyboards, mice, terminals, serial ports. Block-device

drivers handle devices that store data in fixed-size, addressable blocks — disks and SSDs — and support random access and file systems. Network drivers handle packet-based devices and plug into the networking stack. A second distinction is by where they run: kernel-mode drivers execute inside the kernel with full privilege, which makes them fast but dangerous; user-mode drivers (increasingly common for printers, USB gadgets and via frameworks such as Linux's FUSE-adjacent driver models and Windows' UMDF) run as protected processes — slower, but a crash cannot take the system down. A third distinction is between manufacturer-supplied drivers, written by the hardware vendor who knows the device best, and generic class drivers supplied with the operating system for standard devices — a USB storage device, for instance, works anywhere without vendor software because its class standardises the language.

Why Drivers Matter — and Their Risks

Drivers are what make an operating system portable and a computer expandable: install a new graphics card, install its driver, and decades-old applications use it without modification. They also let the kernel stay stable while hardware evolves at its own pace. The price is real, though. Drivers form a very large share of kernel code — in a typical Linux installation, drivers are the majority of the source tree — and because kernel drivers run with full privileges, a single faulty driver can corrupt memory or crash the entire machine; industry experience repeatedly blames a large proportion of system crashes on third-party drivers rather than on the core kernel. This is precisely the motivation for the microkernel structure examined in Assignment 1 and for the modern trend towards user-mode driver frameworks: the driver's role is so essential, and its danger so well proven, that operating-system design now revolves around performing that role safely.

Conclusion

A device driver is the per-device software module that translates the operating system's uniform I/O requests into the private command language of one piece of hardware. Its roles — translation, initialisation, data transfer, interrupt and error handling, and power management — form the bottom layer of the I/O system, insulating every application and most of the kernel from hardware detail. Thanks to drivers, new devices can join old systems and old programs can use new devices; and because drivers are powerful and error-prone, how safely a system runs them is one of the central questions of operating-system architecture.

Question 4: Protection and Security — Access Control Mechanisms (20 Marks)

Question as printed: What is protection and security in operating systems? Explain access control mechanisms.

Introduction

A modern computer holds many users' data, runs programs of doubtful origin and connects to networks full of attackers. Left unguarded, any program could read any file, any user could erase another's work, and any intruder could take the machine over. The operating system is the only software positioned to prevent all of this, and it does so under two related headings: protection and security. Students often confuse the two terms, so this answer begins by separating them precisely, then explains the principles of protection, and then describes the

access control mechanisms — the access matrix, access control lists and capability lists — by which protection is actually enforced.

Protection and Security: The Distinction

Protection is the internal mechanism that controls how processes and users access resources — files, memory, devices, CPU time — inside the system. It is about enforcing rules among actors who are, in principle, legitimate parts of the system: which process may read which file, which user may use the printer, which program may write to which memory. Protection answers the question 'who is allowed to do what to what?' and it is enforced by the kernel's checks on every operation. Security is broader and faces outwards: it is the defence of the system against external and deliberate threats — unauthorised users, malware, network attacks, theft of data, denial of service. Security includes authenticating that users are who they claim to be (passwords, fingerprints), keeping the system trustworthy in a hostile environment, and policies about the environment itself — locked rooms, encrypted backups, network firewalls. A useful summary: protection is the mechanism (the locks and the rules inside the operating system); security is the overall assurance that the whole arrangement, mechanism plus environment plus policy, actually keeps the system safe. A system can have strong protection mechanisms and still be insecure because someone wrote the password on a note stuck to the monitor.

Table 1: Protection versus security

Aspect	Protection	Security
Focus	Internal control of access to resources	Defence against external threats and attacks

Question answered	Who may do what to which resource?	Can the system resist attackers and misuse in its real environment?
Typical subjects	Processes and legitimate users inside the system	Intruders, malware, unauthorised users, hostile networks
Main tools	Access matrix, ACLs, capabilities, privilege levels	Authentication, encryption, firewalls, auditing, physical and organisational policy
Provided by	Kernel mechanisms	The whole system: OS, software, people and procedures

USMAN NOTES www.usmannotes.com

Principles of Protection

Protection design follows principles set out by Saltzer and Schroeder. The most famous is the principle of least privilege: every program and user should operate with the smallest set of privileges needed for the task — a text editor needs files, not the raw disk; a guest account needs almost nothing. Least privilege limits the damage from both accidents and successful attacks. Related principles are separation of privilege (sensitive operations require more than one key or person), complete mediation (every access is checked, not just the first), and fail-safe defaults (access is denied unless explicitly granted). The kernel enforces these by dividing execution into domains — a process running as user A, a process running as the administrator,

the kernel itself — and controlling which objects each domain may access and how domains may switch.

The Access Matrix

The abstract model behind all access control is the access matrix. Rows are domains (users or processes acting for them), columns are objects (files, devices, even domains themselves), and each cell lists the access rights that domain holds over that object: read, write, execute, delete, own, copy, and so on. The matrix is conceptually complete but practically impossible to store in full — with thousands of users and millions of files it would be enormous and almost entirely empty. Real systems therefore store it by columns or by rows, which gives the two practical mechanisms described next.

Access Control Lists (ACLs)

An access control list stores the matrix by column: attached to each object is a list of who may access it and in what way. When a process requests access to a file, the kernel looks up the file's ACL and checks whether the requester appears with the requested right. UNIX compresses the idea into its famous owner–group–others permission bits (read, write, execute for each class), which is simply a very compact ACL; Windows and modern UNIX systems support full ACLs in which individual users and groups can each be granted or denied individual rights. ACLs make it easy to answer 'who can access this object?' — the natural question when securing a file — and to revoke someone's access by editing one list. Their cost is that finding 'everything this user can access' is laborious, and checking a long list on every access takes time.

Capability Lists

A capability list stores the matrix by row: each domain (in practice, each process) holds a set of capabilities — unforgeable tokens, each naming an object together with a set of rights to it. To access an object, the process presents its capability; possession of the capability is the permission, so the kernel does not need to know who the process is, only that it holds the right token. Capabilities can be passed from process to process, which makes controlled delegation natural, and checking is fast because the capability travels with the request. Their weaknesses mirror the ACL's strengths: it is hard to answer 'who can access this object?', and revoking a right that has been copied and widely distributed is notoriously difficult. Real systems mix the two ideas — UNIX file descriptors, for example, behave like capabilities (a process that holds an open descriptor may use it regardless of later permission changes), while the file permissions that govern opening are ACLs.

USMAN NOTES www.usmannotes.com

Table 2: Access control lists versus capability lists

Point	Access Control List (ACL)	Capability List
Stored with	Each object (column of the matrix)	Each domain/process (row of the matrix)
Check made	Is this requester on the object's list with this right?	Does the requester present a valid capability for this right?
Easy question	Who can access this object?	What can this process access?
Revocation	Easy — edit the object's list	Hard — issued capabilities may have been copied and passed on

Delegation	Awkward — administrator must edit lists	Natural — pass a capability to another process
Example in practice	Windows NTFS permissions; UNIX owner/group/other bits	UNIX open file descriptors; capability-based systems such as seL4 and KeyKOS

Conclusion

Protection is the operating system's internal mechanism for controlling access by processes and users to resources; security is the wider defence of the system against external threats, built on authentication, encryption, policy and the protection mechanism itself. Guided by the principle of least privilege, protection is modelled by the access matrix and implemented by storing it in one of two ways: access control lists, attached to objects, which make auditing and revocation easy, and capability lists, held by processes as unforgeable tokens, which make delegation natural and checking fast. Every real operating system is some deliberate mixture of the two, and understanding both is the key to understanding how systems stay safe while staying usable.

Question 5: Distributed Operating Systems — Features and Advantages (20 Marks)

Question as printed: What is distributed operating system? Explain its features and advantages.

Introduction

A single computer, however powerful, has limits: one CPU complex, one memory, one point of failure. Yet the work of the world — banking, the web, global communication — runs on thousands of machines cooperating as if they were one. The software that creates that illusion is the distributed operating system. Where a traditional operating system manages one machine's resources, a distributed operating system manages a collection of independent computers and presents them to users and programs as a single, coherent system. This answer defines the distributed operating system, distinguishes it from a mere network of machines, and then explains its main features and the advantages that make it the foundation of modern computing.

What Is a Distributed Operating System?

A distributed operating system is an operating system that runs across a set of independent, networked computers — each with its own CPU, memory and local devices — and coordinates them so that the collection appears to its users as one single system. The defining property is transparency: a user runs a program, opens a file or prints a document without knowing, and without needing to know, on which physical machine the work is executed or the data stored. This is stronger than what a network operating system offers: in a network OS (such as ordinary file sharing between desktop machines), the machines remain visibly separate — the user logs into a specific server, copies files explicitly, and knows where everything is. In a true distributed OS, the system itself decides placement, migration and replication, and offers a single system

image. Amoeba, Mach-based distributed systems and, in modern dress, cluster and cloud platforms that schedule containers across fleets of machines all embody this idea.

Features of a Distributed Operating System

The features follow from the definition. (1) Resource sharing: any user on any node can use any resource in the system — processors, storage, printers, specialised hardware — as if it were local, because the system locates and allocates resources globally. (2) Transparency, in its several forms: access transparency (the same operations work on local and remote resources), location transparency (names do not reveal where things are), migration transparency (things can move without breaking names), replication transparency (users cannot tell that several copies exist) and failure transparency (faults are masked where possible). (3) Openness and heterogeneity: the system is built on standard protocols and interfaces, so machines of different types and from different vendors can join and cooperate. (4) Concurrency and parallelism: many processes genuinely execute at the same time on different nodes, and a single job may be split across nodes to finish sooner. (5) Scalability: new nodes can be added as load grows, in principle without limit and without redesigning the system. (6) Fault tolerance: because there are many machines and data can be replicated, the failure of one node degrades the system rather than destroying it; work can be reassigned and copies take over. (7) Distributed coordination services: beneath the surface, the system must provide the hard primitives the features rest on — a global naming service, distributed file storage, synchronisation between processes on different machines, agreement on the order of events without a single global clock, and security across machines that do not fully trust each other.

Table 1: Distributed OS versus single-machine and network OS

Aspect	Single-machine OS	Network OS	Distributed OS
Machines managed	One computer	Several, but visibly separate	Several, presented as one system
User's view	One system	Many systems — user logs into specific machines	One system (single system image)
Who decides placement	Only one machine exists	The user, explicitly	The operating system, automatically
File access	Local only	Explicit copy/remote login to the owning machine	Uniform access from anywhere, location hidden
Fault behaviour	Machine fails = system fails	One machine fails; user works around it	System continues; work migrates, replicas take over
Examples	MS-DOS, standalone desktop OS	Workgroup file/printer sharing	Amoeba; modern cluster/cloud operating layers

USMAN NOTES www.usmannotes.com

Advantages

The advantages are the reasons distributed systems conquered the world. Resource sharing means expensive or idle equipment is never wasted: a idle workstation's CPU can run another node's job, and specialised devices serve everyone. Reliability and availability rise sharply: with

data replicated on several nodes and services able to restart elsewhere, no single failure is fatal — this is why banks and airlines, which cannot stop, are distributed. Performance improves two ways: total computing power is the sum of all nodes, and work can be placed near the data or the user, cutting delay; load balancing spreads jobs so no machine is overloaded while others idle. Scalability and incremental growth let an organisation start small and add machines as demand rises, paying as it grows rather than buying one huge machine in advance. Cost-effectiveness follows from using many cheap commodity machines instead of a single specialised mainframe. Finally, distribution matches the natural shape of modern organisations — branches, campuses and continents each have machines, and a distributed system joins them into one working whole rather than forcing everything into one room.

Honesty requires noting the costs alongside the advantages, since they shape real designs: distributed systems are far more complex to build (partial failure — where one node fails while others continue — has no counterpart on a single machine); they depend on a network whose delays and failures they cannot control; and securing many machines and the messages between them is much harder than securing one box. The advantages are won only by engineering that takes these difficulties seriously — replication protocols, consensus algorithms and careful failure handling — which is why distributed systems is an advanced subject in its own right.

Conclusion

A distributed operating system coordinates a network of independent computers and presents them as a single coherent system, hiding location, movement, replication and even failure from

its users. Its features — global resource sharing, many forms of transparency, openness, true concurrency, scalability, fault tolerance and supporting coordination services — deliver decisive advantages: shared resources, high reliability, greater performance through parallelism and balance, smooth growth and good economics. Those advantages, achieved despite real complexity, are why the world's serious computing — from university clusters to the global cloud — is built the distributed way.

USMAN NOTES
www.usmannotes.com