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. 1

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: Structure of an Operating System — Monolithic and Microkernel Architectures (20 Marks)

Question as printed: Explain the structure of an operating system. Compare monolithic and microkernel architectures.

Introduction

An operating system is not a single, shapeless program. It is a large and complex collection of cooperating components — schedulers, memory managers, file systems, device drivers, security checks and user interfaces — and the way these components are arranged, connected

and separated from one another is called the structure of the operating system. Structure matters because an operating system is among the largest pieces of software ever written: millions of lines of code that must be correct, fast, secure and easy to extend when new hardware appears. A poor structure makes the system slow, unreliable and hard to maintain; a good structure hides complexity, confines faults to small parts of the system, and lets developers add or replace components without rewriting everything. This answer first explains what the structure of an operating system means and describes the main structural approaches used in real systems, and then compares the two most influential designs — the monolithic architecture and the microkernel architecture — point by point.

What 'Structure' Means in an Operating System

When we talk about the structure of an operating system we are asking three questions. First, where does each service run: inside the protected kernel space, where it can access all hardware and memory, or in ordinary user space, where it is restricted like any application? Second, how do the parts communicate: by direct procedure calls inside one address space, or by messages passed between separate processes? Third, how are the parts layered and protected from each other, so that a fault in one part does not crash the whole machine? The simplest mental picture is the layered view shown in Figure 1: users work with application programs, applications request services from the operating system, and the operating system alone controls the hardware. Everything inside the middle layer — how it is organised internally — is the subject of operating-system structure.

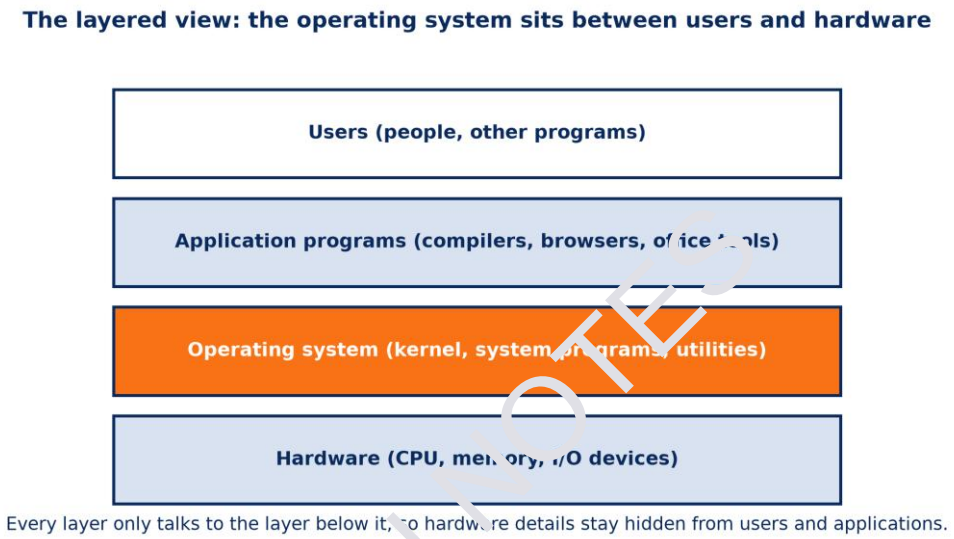


Figure 1: The layered view of a computer system — the operating system sits between application programs and the hardware.

Main Components That Every Structure Must Organise

Whatever structure is chosen, the same set of components must be placed somewhere. The process manager creates, schedules and destroys processes. The memory manager allocates RAM, protects one process from another and runs virtual memory. The file system organises data on disks into files and directories. The I/O and device-driver subsystem talks to keyboards, displays, printers and network cards. The protection and security component checks who may do what. Finally, the user interface — a shell or graphical desktop — accepts commands. Structural designs differ only in how these components are packaged: in one single kernel program, in a tiny kernel plus user-level servers, or in stacked layers.

Common Structural Approaches

Historically, four approaches dominate. In a simple or monolithic structure, the entire operating system is one large program running in kernel mode; early systems such as MS-DOS and the

original UNIX were built this way. In a layered structure, the system is divided into levels (as in Dijkstra's THE operating system), where each layer uses only the services of the layer below it, which makes debugging systematic but can cost speed. In a microkernel structure, only the most basic functions stay in the kernel and everything else runs as user-space servers. Modern commercial systems mostly use a modular or hybrid structure: a kernel that can load and unload modules (as in Linux) or a mixture of monolithic and microkernel ideas (as in Windows and macOS), chosen to balance performance with reliability.

Monolithic Architecture

In a monolithic (literally 'one stone') architecture, all operating-system services — process management, memory management, file systems, networking and all device drivers — are compiled into a single large kernel that runs entirely in kernel (supervisor) mode in one address space. When an application needs a service, it makes a system call: the CPU switches from user mode to kernel mode, the kernel function executes directly, and the result is returned. Because every component can call every other component as an ordinary function call, communication is extremely fast and there is almost no overhead in crossing between services.

The strengths of the monolithic design are performance and simplicity of interaction: Linux, the classic example, achieves very high throughput precisely because its file system, scheduler and network stack talk to each other directly. Its weaknesses are the mirror image of those strengths. Because everything shares one address space with full privileges, a bug in any single driver can corrupt memory and crash the entire system. The kernel becomes very large, difficult to understand, test and maintain, and adding a new service traditionally meant rebuilding and

rebooting the kernel — a problem modern Linux softens with loadable kernel modules, but does not remove completely. Security is also weaker in principle: any flaw anywhere in the kernel is a flaw in the most privileged code on the machine.

Microkernel Architecture

The microkernel approach asks: what is the smallest possible set of functions that absolutely must run in kernel mode? The answer is only the most primitive operations — basic CPU scheduling, low-level memory primitives, inter-process communication (IPC) and minimal interrupt handling. Everything else is removed from the kernel and implemented as ordinary user-space server processes: the file system is a server, each device driver is a server, the network stack is a server. When an application wants to read a file, it sends a message to the file-system server through the microkernel; the server does the work and replies by message. Mach, developed at Carnegie Mellon, and MINIX 3 are classic microkernel systems, and the QNX real-time system shows the design succeeding commercially in embedded devices.

The advantages follow from isolation. Because servers run in user space with no special privileges, a crashed driver or file-system server can often be restarted without rebooting the machine — MINIX 3 famously restarts failed drivers automatically. The system is more secure, because a bug in a server can only damage that server's small world, not the whole memory. It is also more modular and portable: new services can be added as new servers, and the tiny kernel is easier to verify and to move to new hardware. The price is performance. Every service request now involves message passing and context switches between processes instead of a cheap function call, and this IPC overhead made early microkernels such as Mach noticeably

slower than monolithic rivals. Careful engineering, as in the L4 microkernel family, has greatly reduced this overhead, but it has never disappeared entirely.

Comparison of Monolithic and Microkernel Architectures

Table 1: Monolithic versus microkernel architecture

Point of comparison	Monolithic architecture	Microkernel architecture
Basic idea	All OS services form one large kernel program	Only minimal functions in the kernel; services run as user-space servers
Where services run	Entirely in kernel mode, one address space	Mostly in user mode, separate address spaces
Communication between parts	Direct function calls — very fast	Message passing (IPC) through the kernel — slower
Size of kernel	Very large (millions of lines)	Very small (a few thousand lines)
Reliability	A fault in any component can crash the whole system	A fault is confined to one server; it can often be restarted
Security	Any kernel bug threatens the whole system	Small trusted core; damage from a bug is limited
Extensibility	Adding services means changing the kernel (modules help)	Add a new server process; kernel untouched
Performance	High — minimal overhead	Lower — IPC and context-switch overhead

USMAN NOTES www.usmannotes.com

Examples	Linux, traditional UNIX, MS-DOS	MINIX 3, Mach, QNX, L4 family

Reading the table row by row, the pattern is clear: the two designs trade the same coin from opposite sides. The monolithic kernel buys speed by giving up isolation; the microkernel buys isolation, security and modularity by giving up some speed. Neither is 'wrong' — the right choice depends on whether the system values raw throughput, as a file server does, or fault tolerance, as a medical or automotive controller does.

The Hybrid Compromise Used Today

In practice, the debate ended in a compromise. Windows runs many services in user space like a microkernel but keeps graphics and core services in the kernel for speed. macOS combines the Mach microkernel with a BSD monolithic layer in a single kernel (XNU). Linux remains monolithic but loads and unloads modules dynamically, gaining some microkernel-like flexibility. Modern designers therefore treat 'monolithic versus microkernel' not as a war but as a spectrum, placing each service where its speed and reliability requirements fit best.

Conclusion

The structure of an operating system describes how its components — process, memory, file, I/O and security managers — are arranged, where they run, and how they communicate. The monolithic architecture places everything in one privileged kernel and wins on performance, at the cost of size, fragility and weaker isolation. The microkernel architecture keeps only scheduling, basic memory support and message passing in the kernel, runs all other services as protected user-level servers, and wins on reliability, security and extensibility, at the cost of IPC

overhead. Most real systems today are hybrids that deliberately mix the two, which shows that structure is an engineering trade-off, not a slogan.

Question 2: Process and the Process Life Cycle (20 Marks)

Question as printed: What is a process? Describe the process life cycle with a diagram.

Introduction

Every useful thing a computer does — editing a document, playing a song, downloading a file — is done by a program in execution, and the operating system's name for a program in execution is a 'process'. The process is the most central concept in all of operating-system theory: it is the unit of work that the CPU schedules, the container that memory management protects, and the entity that the process life cycle describes from the moment of its creation to the moment of its termination. This answer defines a process carefully, distinguishes it from a program, and then walks through the five states of the process life cycle with the help of a state diagram, explaining what triggers every transition.

What Is a Process?

A process is a program in execution, together with everything that belongs to that execution: the current values of the program counter and CPU registers, the process's stack, its data and heap, and the resources the operating system has given it, such as open files and an area of memory. The distinction between a program and a process is essential. A program is a passive entity — a file of instructions stored on disk, like a recipe written in a book. A process is an active entity — the cooking actually in progress, with a cook, ingredients laid out, and a counter

showing exactly which step is being performed. The same program can therefore exist as many processes at once: three students may open the same text editor program, but each has a separate process with its own memory, its own documents and its own state.

A process in memory is traditionally pictured in four sections. The text section holds the executable code. The data section holds global and static variables. The heap is memory allocated dynamically while the process runs (for example with malloc), and the stack holds temporary data such as function parameters, return addresses and local variables. Alongside this memory image, the operating system keeps a record for the process — the process control block — which Question 3 discusses in detail. A process also has an identity: a unique process identifier (PID) by which the operating system refers to it.

The Process Life Cycle: The Five States

During its lifetime, a process moves through a series of states. A state is simply the current activity of the process, and at any moment a process is in exactly one state. The five standard states are New, Ready, Running, Waiting (also called Blocked) and Terminated. Figure 1 shows how a process travels between them. It is important to remember that on a machine with one CPU core, at most one process per core can be in the Running state at any instant, while many processes may be Ready or Waiting at the same time.

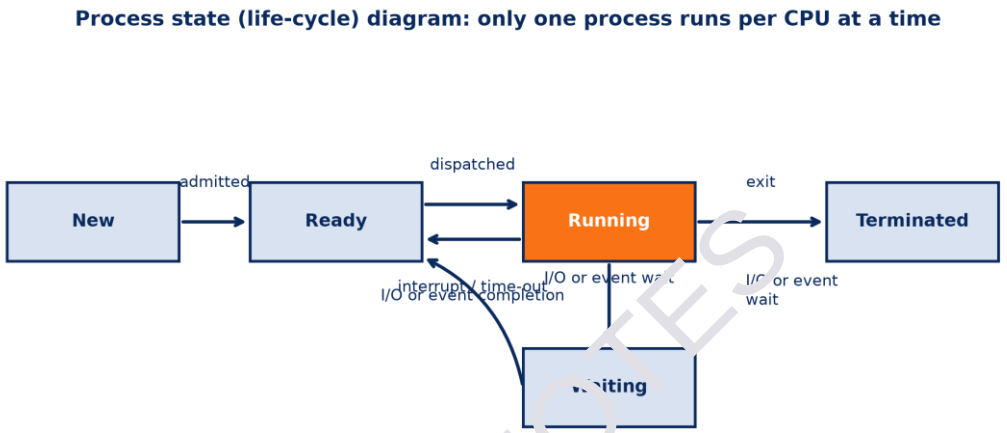


Figure 1: The process state (life-cycle) diagram, showing the five states and the transitions between them.

Table 1: The five process states

State	Meaning	Typical cause of entry
New	The process is being created; the OS is allocating its resources and PCB	User starts a program, or a process issues a fork() call
Ready	The process has everything it needs except the CPU and is waiting in the ready queue	Creation complete, I/O finished, or the process was preempted
Running	The process's instructions are being executed by the CPU	The scheduler dispatches it from the ready queue
Waiting (Blocked)	The process cannot proceed until some event occurs	It requests I/O, waits for a signal or for a child process
Terminated	The process has finished execution and the OS is	It calls exit(), completes main(), or is killed by the OS or user

	reclaiming its resources	

Transitions Between the States

The life cycle is best understood through its four main transitions. (1) New → Ready (admitted): once the operating system has created the PCB, loaded the code and admitted the process into the system, it is placed at the end of the ready queue. Admission may be delayed by a long-term scheduler on systems that control how many processes compete for memory. (2) Ready → Running (dispatch): the CPU scheduler selects the process — according to the scheduling algorithm in use — and the dispatcher gives it the CPU; this is the only transition the scheduler itself performs. (3) Running → Terminated (exit): the process finishes its last instruction or calls exit(); the operating system then reclaims its memory, closes its files and removes its PCB, after reporting its exit status to its parent.

The remaining transitions are what make multiprogramming useful. (4) Running → Waiting: a running process may need something it cannot immediately have — it issues a read from disk, waits for keyboard input, or waits for a child — so it voluntarily gives up the CPU and moves to a waiting queue attached to that event. (5) Waiting → Ready: when the event completes — the disk delivers the data, the user presses the key — the process does not run immediately; it returns to the ready queue to compete for the CPU again. (6) Running → Ready: an involuntary transition, the interrupt or preemption, occurs when the process's time slice expires or a higher-priority process arrives; the process is pushed back to Ready so that the CPU can serve

others. A context switch — saving one process's state into its PCB and loading another's — accompanies every dispatch, which is why the state in the PCB must always be kept up to date.

Why the Life Cycle Matters

The life-cycle model explains, in one picture, how the operating system shares one CPU among many programs: processes take turns in Running, and while some are stuck in Waiting for slow devices, others use the processor, so the CPU is never idle while work remains. It also explains what the user observes: a program that "hangs" is usually not dead but Waiting, and a system that feels responsive is one whose ready queue is being served fairly. Schedulers, discussed in Question 4, are precisely the policies that decide the Ready → Running edge of this diagram.

Conclusion

A process is a program in execution — code plus data, stack, heap, registers and operating-system resources, identified by a PID — as opposed to the passive program file on disk. Its life cycle carries it through New, Ready, Running, Waiting and Terminated states, driven by admission, dispatching, interrupts, I/O requests and completion, and recorded continuously in its process control block. Understanding this cycle is the foundation for understanding CPU scheduling, memory management and almost every other topic in operating systems.

Question 3: Process Control Block (PCB) (20 Marks)

Question as printed: Explain process control block (PCB). What information does it contain?

Introduction

The operating system may be managing hundreds of processes at once, yet it has only a few CPUs, and processes are constantly stopping and starting — sometimes thousands of times per second. For this to work, the operating system needs a complete, instantly accessible record of every process: where it stopped, what it owns, how much CPU it has used, and what it is waiting for. That record is the process control block, or PCB. The PCB is arguably the most important data structure in the operating system, because the process, as far as the kernel is concerned, is its PCB: create the PCB and the process exists; delete the PCB and the process is gone. This answer explains what the PCB is, lists and explains each category of information it contains, and shows how it is used in context switching.

What Is a Process Control Block?

A process control block is a data structure maintained by the operating system kernel for every process in the system. It is created when the process is created, updated whenever the process's state changes, consulted by the scheduler, the memory manager and the accounting system, and destroyed when the process terminates. All PCBs are kept by the kernel in a process table (in Linux, as task_struct structures linked together). The PCB is a kernel structure, invisible and untouchable by the process itself — for good reason, since it contains exactly the information a malicious process would need to take over the machine. When a process is not running, its complete 'frozen' state lives in its PCB; when it runs again, that state is restored to the CPU, and execution continues as if it had never stopped.

Information Contained in the PCB

The exact fields vary between operating systems, but the information falls into a standard set of categories, summarised in Table 1 and explained below.

Table 1: Information stored in a process control block

Category	Contents	Why the OS needs it
Process identification	Process ID (PID), parent PID, user ID	To name, organise and take responsibility for the process
Process state	New, Ready, Running, Waiting or Terminated	So the scheduler knows which queue the process belongs to
Program counter	Address of the next instruction to execute	So execution resumes at exactly the right instruction
CPU registers	Accumulators, index and stack registers, condition code	To restore the process's exact working state after a switch
CPU scheduling information	Priority, pointers to scheduling queues, time used	So the scheduler can make fair and informed choices
Memory-management information	Page tables / segment tables, base and limit registers	To locate the process's memory and protect other processes
Accounting information	CPU time consumed, time limits, job/account numbers	For billing, monitoring and enforcing limits
I/O status information	Allocated devices, list of open files	To track resources and release them at termination

USMAN NOTES www.usmannotes.com

The Fields Explained

Process identification comes first: each PCB carries the process's unique PID, the PID of its parent (which allows process trees and lets the parent collect the child's exit status), and the identity of the owning user, against which permissions are checked. Process state records which of the five life-cycle states the process is in, so that queues can be maintained: in practice the kernel keeps queues of PCBs — a ready queue, and separate waiting queues for each device — usually implemented as linked lists of PCB pointers.

The processor state information — program counter and CPU registers — is the heart of the PCB. When the operating system interrupts a running process, every register value and the exact address of the next instruction are copied into the PCB; this is the only reason a preempted process can later continue mid-calculation without noticing any gap. CPU scheduling information holds the process's priority, its position in scheduling queues, and any parameters the scheduler needs. Memory-management information describes where the process lives in memory — pointers to its page or segment tables, or base and limit register values — both so its instructions can be found and so it can be prevented from touching anyone else's memory. Accounting information records how much CPU time the process has consumed, its real time in the system and any limits imposed on it; and I/O status information lists the devices allocated to it and the files it currently has open, so that on termination everything can be closed and released in an orderly way.

The PCB in Action: Context Switching

The PCB earns its keep during a context switch. Suppose process P0 is running and the operating system decides to run P1 instead. The kernel saves P0's program counter and registers into PCB0 and marks P0 Ready; it then loads P1's registers and program counter from PCB1, switches to P1's memory map, marks P1 Running, and jumps to the saved instruction. From P1's point of view, nothing happened — its registers contain exactly its own values. A context switch is pure overhead: the system does no useful work while performing it, which is why its cost (typically a few microseconds) is dominated by how much state the PCB must save and restore, and why operating systems work hard to make PCBs compact. The same mechanism, on a larger scale, is used when a process blocks for I/O and when it is finally terminated: in every case the PCB is the single source of truth about the process.

Conclusion

The process control block is the operating system's complete record of a process: its identity and state, its saved program counter and registers, its scheduling and memory information, its accounting data and its I/O resources. It is created at the process's birth, maintained throughout the life cycle, and destroyed at termination. Without the PCB there could be no multiprogramming, because there would be no way to stop a process and later resume it exactly where it left off; with it, hundreds of processes can share a single CPU and each believe it owns the machine.

Question 4: CPU Scheduling — FCFS, SJF and Round Robin (20 Marks)

Question as printed: What is CPU scheduling? Compare FCFS, SJF, and Round Robin algorithms.

Introduction

In a multiprogrammed system, several processes sit in memory at once, all competing for a CPU that can execute only one of them (per core) at any instant. Every time one process stops — because it finished, because it asked for I/O, or because its time ran out — someone must choose which waiting process runs next. That choosing is CPU scheduling, and the policy used to choose is the CPU scheduling algorithm. Scheduling decisions may seem small, but they are taken thousands of times a day by every computer, and they decide whether a system feels responsive or sluggish, whether the CPU is used efficiently, and whether every process gets a fair share. This answer defines CPU scheduling, states the criteria by which algorithms are judged, and then explains and compares the three classic algorithms: First-Come First-Served, Shortest Job First and Round Robin, with a fully worked numerical example.

What Is CPU Scheduling?

CPU scheduling is the activity of the operating system's short-term scheduler: selecting, from among the processes in the ready queue, the one to which the CPU will be allocated next. It is needed because of the way processes behave — a process alternates between CPU bursts, in which it computes, and I/O bursts, in which it waits for a device. Scheduling takes place whenever a process switches from Running to Waiting (for example, on an I/O request), from Running to Ready (on an interrupt or time-slice expiry), or from Waiting to Ready (when I/O

completes), and when a process terminates. If the decision is made only in the voluntary cases, scheduling is non-preemptive: once given the CPU, the process keeps it until it releases it. If the scheduler can also snatch the CPU away — on a timer interrupt, for instance — scheduling is preemptive, which modern systems use to protect responsiveness.

Four measures are used to compare scheduling algorithms. CPU utilisation is the percentage of time the CPU is kept busy, which we want high. Throughput is the number of processes completed per unit time, again to be maximised. Turnaround time, from a process's arrival to its completion, and waiting time, the total time a process spends sitting in the ready queue, are to be minimised — waiting time is the fairest comparison measure because scheduling can affect only time spent waiting, never the process's own burst lengths. Finally, response time — the delay until the first output appears — matters most in interactive systems. Different algorithms optimise different measures, which is why no single algorithm is best for every system.

First-Come First-Served (FCFS)

FCFS is the simplest algorithm: processes are served strictly in the order in which they arrive in the ready queue — exactly like a queue at a ticket window. It is non-preemptive and trivially easy to implement with a first-in first-out queue. Its great weakness is the convoy effect: if one long process arrives first, all the short processes behind it must wait for it, even though they could finish in a moment. Average waiting time under FCFS is therefore often very poor, and it depends luckily or unluckily on arrival order. FCFS is fair in the everyday sense that nobody jumps the queue, but it is neither efficient nor responsive enough for interactive systems.

Shortest Job First (SJF)

SJF takes the opposite approach: from the ready queue, always run the process with the shortest next CPU burst. SJF is provably optimal in one precise sense — it gives the minimum possible average waiting time for any given set of processes, because moving a short job ahead of a long one reduces total waiting more than any other ordering. SJF comes in non-preemptive form (a running job keeps the CPU to the end of its burst) and preemptive form — called Shortest Remaining Time First — in which a newly arrived shorter job can displace the one currently running. Its practical problem is fundamental: the operating system cannot know the future. Burst lengths must be predicted, usually by an exponential average of past bursts, and a poor prediction loses SJF's advantage. Its second problem is starvation: a steady stream of short jobs can postpone a long job indefinitely. Even so, SJF is the yardstick against which other algorithms are measured, and its principle lives on inside many real schedulers.

Round Robin (RR)

Round Robin was designed for time-sharing systems, where many users must all feel that the machine is responding to them. Each process is given a fixed slice of CPU time called a time quantum (typically 10 to 100 milliseconds), and processes are served in rotation from a circular ready queue. If a process finishes within its quantum, it leaves; if not, it is preempted at the end of the quantum and rejoins the tail of the queue. No process ever waits longer than $(n - 1) \times$ quantum for its next turn, so response time is excellent and the CPU is shared democratically. The quantum size is a delicate trade-off: too large and RR degenerates into FCFS; too small and

the system wastes its time in context switches — a common rule of thumb is that the quantum should be long enough that about 80 per cent of CPU bursts finish within a single quantum.

Worked Example: The Same Three Processes Under All Three Algorithms

Consider three processes that all arrive at time 0, in the order P1, P2, P3, with CPU bursts of 24, 3 and 3 milliseconds respectively (the classic textbook example). Figure 1 draws the Gantt chart — the timeline of who runs when — for each algorithm.

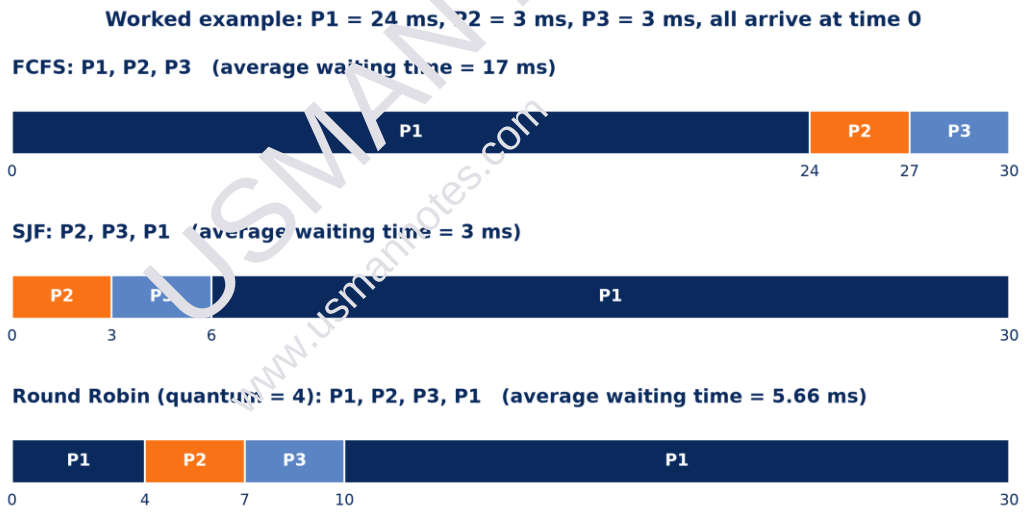


Figure 1: Gantt charts for FCFS, SJF and Round Robin (quantum 4 ms) for P1 = 24 ms, P2 = 3 ms, P3 = 3 ms.

Under FCFS, P1 runs first for its full 24 ms, so P2 waits 24 ms and P3 waits 27 ms; average waiting time = (0 + 24 + 27) / 3 = 17 ms — the convoy effect in numbers. Under SJF, the two short jobs run first: P2 waits 0, P3 waits 3, and P1 waits 6 ms; average waiting time = (0 + 3 + 6) / 3 = 3 ms — the best possible for this set, exactly as the theory promises. Under Round Robin with quantum 4 ms, P1 runs 0–4, P2 runs 4–7 and finishes, P3 runs 7–10 and finishes, and P1 then runs alone 10–30. Waiting times are P1 = 6 ms (it waits from time 4 to 10... more precisely,

its turnaround is 30 ms minus its 24 ms burst = 6 ms), P2 = 4 ms and P3 = 7 ms, giving an average of (6 + 4 + 7) / 3 = 5.66 ms. The example shows the general truth: SJF wins on average waiting time, FCFS is worst when a long job leads, and Round Robin sits between them on waiting time while beating both on fairness and response time.

Comparison of the Three Algorithms

Table 1: FCFS versus SJF versus Round Robin

Criterion	FCFS	SJF	Round Robin
Basic rule	Serve in arrival order	Serve the shortest next burst first	Serve each process for one time quantum, in rotation
Preemptive?	No	Both forms exist (SRTF is preemptive)	Yes — preemption at quantum expiry is the core idea
Average waiting time	Often high; suffers the convoy effect	Provably minimum (optimal) for a given set	Moderate; depends on quantum size
Starvation possible?	No — everyone is served in turn	Yes — long jobs may wait indefinitely	No — every process gets a turn within (n − 1) quanta
Knowledge required	None — just arrival order	Future burst lengths, which must be predicted	None, but the quantum must be tuned

USMAN NOTES www.usmannotes.com

Best suited to	Simple batch systems	Batch systems with predictable jobs	Interactive and time-sharing systems
Overhead	Negligible	Low, plus prediction effort	Context-switch overhead if the quantum is too small

Conclusion

CPU scheduling is the short-term scheduler's repeated choice of which ready process receives the CPU, judged by utilisation, throughput, turnaround, waiting and response times. FCFS is simple and fair in order but suffers the convoy effect; SJF delivers the provably lowest average waiting time but needs unknowable future information and can starve long jobs; Round Robin sacrifices a little average efficiency to guarantee that every process — and every user — receives regular, responsive service. The worked example (average waiting 17 ms, 3 ms and 5.66 ms respectively) shows all three characters clearly, and explains why real time-sharing systems are built on Round Robin ideas while SJF remains the theoretical benchmark.

Question 5: Threads — Multithreading versus Multiprocessing (20 Marks)

Question as printed: Define threads. Differentiate between multithreading and multiprocessing.

Introduction

For decades the process was the only unit of execution an operating system offered, but processes are heavy: each carries its own memory, its own resources and a high creation and switching cost. Many programs, however, are naturally made of several simultaneous activities that all work on the same data — a web server answering many clients at once, or a word processor that must accept typing, check spelling and repaginate at the same time. Threads were invented to serve exactly this need: they bring parallelism inside the process, at a fraction of the cost. This answer defines a thread precisely, explains what threads share and what each thread keeps privately, describes the benefits and difficulties of multithreading, and then draws a systematic distinction between multithreading and multiprocessing.

What Is a Thread?

A thread is the smallest unit of CPU execution within a process: a single sequential flow of control that has its own program counter, its own set of CPU registers and its own stack, but that shares the process's code, data, heap and open files with all the other threads of the same process. A traditional process can be described as a resource container plus exactly one thread; a multithreaded process is a container with many threads running inside it. Each thread is given a thread ID by the system, just as each process has a PID. Because threads are so much cheaper to create and switch between than processes, they are often called lightweight processes, while the ordinary heavyweight process is the unit that owns the resources.

A useful picture is a house and its occupants. The process is the house: one address, one kitchen, one set of furniture (the shared memory and files). The threads are the people living in

it: each person is doing their own task and keeps their own to-do list and personal items (stack and registers), but everyone shares the kitchen and can leave notes for each other on the fridge (shared variables). Communication between threads is therefore as easy as reading and writing shared memory — which is both the great advantage of threads and, as we shall see, the source of their main difficulty, the need for synchronisation.

Benefits of Multithreading

Four benefits explain why nearly all modern software is multithreaded. Responsiveness: a program can continue responding to the user while a long task proceeds in a background thread — the browser keeps scrolling while a file downloads. Resource sharing: threads share memory and files by default, so cooperating activities exchange data without the operating-system machinery (pipes, shared-memory segments) that separate processes need. Economy: creating a thread, and switching between threads of one process, is far cheaper than creating or switching processes, because no new memory map is needed and caches stay warm; measurements commonly put thread creation at a small fraction of the cost of process creation. Scalability: on a multiprocessor or multicore machine, the threads of one process genuinely execute in parallel on different cores, so a single application can use the whole machine — the decisive benefit in the multicore era.

Difficulties: Synchronisation and Safety

The honest account must include the costs. Because threads share data, two threads updating the same variable at the same moment can corrupt it — the classic race condition. Programmers must therefore protect shared data with mutexes, semaphores and similar

synchronisation tools, and used wrongly these introduce deadlocks, in which threads wait for each other forever. A second difficulty is that a fatal error in any one thread — an illegal memory access, for instance — usually destroys the whole process and all its threads together: threads share fate as well as memory. Thread programming is consequently harder to write and much harder to debug than single-threaded programming, a cost paid willingly only when the four benefits above are genuinely needed.

Multithreading versus Multiprocessing

Multiprocessing means running several independent processes — either on multiple CPUs or cores (parallel processing in the hardware sense), or cooperating as separate processes even on one machine. The contrast with multithreading is summarised in Table 1. The essential difference is one of sharing and isolation: threads share an address space and live or die together, while processes are isolated in separate address spaces and survive each other's crashes.

Table 1: Multithreading versus multiprocessing

Point of difference	Multithreading	Multiprocessing
Unit of execution	Several threads inside one process	Several independent processes
Address space	All threads share one address space	Each process has its own separate address space
What is private	Only stack, registers and program counter per thread	Everything — memory, files and resources per process

Communication	Directly through shared variables (fast, but needs synchronisation)	Through IPC: pipes, messages, sockets or shared memory (slower, but safer)
Creation and switching cost	Low — lightweight	High — heavyweight; full context and memory-map switch
Effect of a crash	One thread's fatal error usually kills the whole process	One process can crash without harming the others
Security and protection	Weak — threads can read and write each other's data freely	Strong — the OS protects processes from one another
Typical use	Parallel activities on shared data: web servers, GUIs, games	Independent or untrusted tasks: browser tabs, separate applications, micro-services

A real-world illustration makes the table concrete. A web browser such as Chrome uses multiprocessing between tabs — each tab is a separate process, so a crashed or malicious page cannot bring down the others — but multithreading inside each process, where a rendering thread, a network thread and a JavaScript thread share the page's data and must cooperate at full speed. The two techniques are not rivals; good system design uses processes where isolation matters and threads where cheap cooperation matters.

Conclusion

A thread is a lightweight flow of execution inside a process, with its own program counter, registers and stack but sharing the process's code, data and files. Multithreading exploits this sharing to gain responsiveness, economy, easy communication and true parallelism on multicore hardware, at the price of synchronisation complexity and shared fate in a crash. Multiprocessing, by contrast, keeps complete isolation between address spaces: communication is slower and switching heavier, but faults are contained and protection is strong. Knowing which of the two a task needs — cooperation or isolation — is one of the most practical judgements in operating-system design.

USMAN NOTES www.usmannotes.com