CS2ON Operating Systems and Computer Networking

Processes & Scheduling

Lecture Notes

Week 02: Process Management & CPU Scheduling

Table of Contents

Session 1: Computing Environments and Process Fundamentals

This session introduces the foundational concepts that underpin modern operating systems, beginning with the evolution of computing environments and arriving at the central abstraction of a process. Understanding these ideas is essential because every subsequent topic in this module—scheduling, synchronisation, memory management—builds upon the notion of a process and the environment in which it executes.

1.1 Batch Processing and Multiprogramming

Early computing systems operated in a batch-processing mode. A Batch Processing Operating System (sometimes called a BatchOS) manages multiple jobs submitted in sequence: users prepare their programs and data, submit them as a batch, and the system processes each job one after the other without interactive input. While conceptually simple, batch processing suffers from poor resource utilisation because the CPU sits idle whenever a job waits for input/output (I/O) operations to complete.

Multiprogramming was developed to address this inefficiency. The key insight is that a single user cannot keep the CPU and I/O devices busy at all times. In a multiprogrammed system, the operating system keeps a subset of all submitted jobs loaded in main memory simultaneously. One job is selected for execution through a process called job scheduling. When the currently running job must wait—for example, because it has issued a disk read request—the operating system switches the CPU to another job that is ready to execute. This overlap of computation and I/O dramatically increases overall system throughput.

Key Concept: Multiprogramming

Multiprogramming organises multiple jobs in memory so that the CPU always has a job to execute. When the running job blocks on I/O, the OS switches to another ready job, thereby maximising CPU utilisation and system throughput.

1.2 Multitasking

Multitasking extends the principle of multiprogramming to interactive computing. It is the operating system's ability to switch the CPU rapidly between tasks so that users perceive multiple programs running at the same time. Whereas multiprogramming focuses on keeping the CPU busy when jobs block, multitasking adds the goal of providing responsive interaction. The operating system allocates small time slices to each task, cycling through them quickly enough that a human user experiences near-instantaneous feedback.

This distinction matters because it introduces the concept of time sharing—a design philosophy in which the CPU's attention is divided fairly among all active tasks. Time-sharing systems were a revolutionary step in the 1960s, enabling multiple users to work on a single mainframe

simultaneously through terminal connections. Modern desktop and server operating systems are all multitasking systems, even when only one user is logged in.

Feature	Batch Processing	Multiprogramming	Multitasking
User interaction	None (offline)	Minimal	Full interactive
CPU idle time	High (waits for I/O)	Reduced (overlaps I/O)	Minimised (time-sliced)
Primary goal	Job throughput	CPU utilisation	Responsiveness
Job switching trigger	Job completion	I/O wait	Timer interrupt / I/O wait

1.3 The Process Concept

A process is a program in execution. This deceptively simple definition captures a crucial distinction: a program is a passive entity—a file on disk containing instructions—whereas a process is an active entity with a program counter, a set of associated resources, and a current state. A program becomes a process when its executable file is loaded into memory and the operating system allocates the resources it needs to run.

One important consequence is that the same program can give rise to multiple processes. For instance, several users might each open the same web browser application; each instance is a distinct process with its own address space, even though all of them execute the same underlying program code. The operating system treats each process independently, scheduling and managing them as separate units of work.

Key Concept: Process

A process is a program in execution. It is the fundamental unit of work in an operating system. Unlike a program (a static file), a process is a dynamic entity with its own memory space, program counter, and system resources.

1.4 Process Memory Layout

Every process occupies a region of memory that is logically divided into several sections. Understanding this layout is important for topics such as memory management, debugging, and security, all of which are covered in later sessions.

Section	Contents	Growth Direction
Text (Code)	The compiled program instructions (machine code)	Fixed size
Data	Global and static variables initialised before execution	Fixed size
Heap	Dynamically allocated memory	Grows upward

	(e.g., via malloc or new)	
Stack	Temporary data: function parameters, return addresses, local variables	Grows downward

The text section is typically read-only and shared among processes that run the same program, which saves memory. The data section holds global variables that persist for the lifetime of the process. The heap grows dynamically at runtime whenever the process requests additional memory—this is where objects and data structures created during execution reside. The stack, by contrast, grows and shrinks automatically as functions are called and return; it stores local variables, function parameters, and return addresses. If the heap and stack grow towards each other and collide, the process will crash with a stack overflow or out-of-memory error.

Session 2: Process Lifecycle and Management

Having established what a process is, this session examines how the operating system manages the lifecycle of a process—from creation through to termination. Central to this management is the concept of process states, the data structure that records each process's information (the Process Control Block), and the mechanism the CPU uses to switch between processes (context switching).

2.1 Process States and Transitions

As a process executes, it passes through a series of well-defined states. The operating system tracks these states to make scheduling and resource-allocation decisions. The five standard process states are as follows.

State	Description
New	The process is being created. The OS is allocating resources and initialising the PCB.
Ready	The process is loaded in memory and waiting to be assigned to a processor for execution.
Running	The process's instructions are currently being executed on the CPU.
Waiting (Blocked)	The process cannot proceed until some event occurs, such as completion of an I/O operation.
Terminated	The process has finished execution. Its resources are being deallocated.

Transitions between these states are governed by specific events. A newly created process moves to the ready state once initialisation is complete. The scheduler dispatches a ready process to the running state. A running process may transition to the waiting state if it requests I/O, back to the ready state if it is preempted by the scheduler, or to the terminated state upon completion. When the awaited event (e.g., I/O completion) occurs, a waiting process returns to the ready queue.

Key Concept: Process State Diagram

Every process transitions through five states: New → Ready → Running → Terminated, with the possibility of moving between Running and Waiting (Blocked) and between Running and Ready (via preemption). The OS scheduler manages these transitions.

2.2 The Process Control Block (PCB)

The Process Control Block (PCB) is the data structure the operating system uses to store all information about a process. Each process has exactly one PCB, and it is created when the process

enters the new state. The PCB is the mechanism by which the OS can stop a process, save its state, and later resume it exactly where it left off.

A typical PCB contains the following information: the process state (new, ready, running, waiting, or terminated), the program counter (the address of the next instruction to execute), CPU register values (which must be saved and restored during context switches), CPU scheduling information (such as the process priority and pointers to scheduling queues), memory-management information (such as page tables or segment tables), accounting information (CPU time used, time limits, job or process numbers), and I/O status information (a list of open files and allocated I/O devices). The PCB is stored in a protected region of memory that only the kernel can access, preventing user processes from corrupting one another's control information.

- Process state (new, ready, running, waiting, terminated)
- Program counter — address of the next instruction
- CPU registers — saved/restored on context switches
- Scheduling information — priority, queue pointers
- Memory-management information — page/segment tables
- Accounting information — CPU time used, limits
- I/O status — open files, allocated devices

2.3 Context Switching

A context switch occurs when the CPU switches from executing one process to executing another. This involves saving the entire state (context) of the currently running process into its PCB and loading the previously saved state of the next process from its PCB. Context switching is a fundamental operation in any multitasking operating system; without it, the CPU could only run a single process at a time.

It is important to recognise that context-switch time is pure overhead—the system performs no useful work during a switch. The duration depends on hardware support (e.g., the number of registers to save), the complexity of the operating system's data structures, and the speed of memory. Modern CPUs provide hardware features such as multiple register sets to accelerate context switches, but the cost is never zero. Operating system designers must therefore balance the frequency of context switches (which affects responsiveness) against the overhead each switch introduces.

Key Concept: Context Switch

A context switch saves the state of the current process (registers, program counter, etc.) to its PCB and restores the state of the next process. It is essential for multitasking but introduces overhead, as the CPU does no productive work during the switch.

2.4 Process Creation

Processes are created in a hierarchical manner: a parent process creates one or more child processes, each of which may in turn create further children, forming a tree of processes. Every

process is typically identified by a unique process identifier (PID). The root of the process tree is the init process (PID 1 on UNIX/Linux systems), which is started by the kernel during boot.

When a parent creates a child, three resource-sharing models are possible: the parent and child may share all resources, the child may use a subset of the parent's resources, or the parent and child may share no resources at all. Likewise, two execution models exist: the parent and child may execute concurrently, or the parent may wait (using a system call such as wait()) until the child terminates.

2.4.1 UNIX Process Creation: fork() and exec()

In UNIX and Linux, the fork() system call creates a new process by duplicating the calling process. The child receives a copy of the parent's address space (including code, data, heap, and stack), but the two processes then execute independently. The fork() call returns twice: in the parent it returns the child's PID, and in the child it returns zero. This return value allows each process to determine its role.

Frequently, the child process calls exec() immediately after fork(). The exec() family of system calls replaces the process's memory space with a new program loaded from disk. This two-step fork-then-exec pattern is the standard mechanism for launching new programs on UNIX systems. It provides flexibility: the child can set up file descriptors, environment variables, and other attributes between the fork() and exec() calls.

2.5 Process Termination

A process terminates when it finishes executing its final statement and invokes the exit() system call. At this point, the process may return a status value to its parent via the wait() system call. The operating system then deallocates all of the process's resources, including memory, open files, and I/O buffers.

A parent may also terminate a child process explicitly using the abort() or kill() system call. Reasons for doing so include the child exceeding its allocated resources, the child's task no longer being required, or the parent itself terminating. Some operating systems enforce the rule that a child cannot exist if its parent has terminated; in such systems, all descendants of a terminated process are also terminated—a phenomenon known as cascading termination. The termination is initiated by the operating system itself.

A parent may wait for a child to terminate by calling wait(&status). This call blocks the parent until one of its children exits, at which point it returns the child's PID and exit status. If a child terminates but its parent has not yet called wait(), the child becomes a zombie process—it has finished execution but still occupies an entry in the process table. Conversely, if the parent terminates without calling wait(), the child becomes an orphan process, which is typically adopted by the init process.

Key Concept: Cascading Termination

Some operating systems require that all children, grandchildren, and further descendants of a terminating process must also be terminated. This cascading termination is initiated by the OS and ensures that no orphaned processes remain without a managing parent.

Session 3: Threads and Multithreading

Modern applications routinely need to perform multiple tasks simultaneously—a web browser, for example, must render pages, handle user input, download files, and run scripts all at once. While separate processes could be used for each task, this approach is heavyweight: creating a new process involves duplicating the address space, allocating new resources, and incurring significant overhead. Threads offer a lighter-weight alternative.

3.1 The Thread Model

A thread is the smallest unit of CPU utilisation. It consists of a thread ID, a program counter, a register set, and a stack. Crucially, threads belonging to the same process share the process's code section, data section, heap, and other operating-system resources such as open files and signals. This sharing makes thread creation and context switching significantly cheaper than process creation and process-level context switching.

In a multithreaded process, multiple tasks within the application can be implemented by separate threads. Consider a word processor: one thread might handle keyboard input, another performs spell checking in the background, a third manages the display, and a fourth auto-saves the document periodically. Because all these threads share the same address space, they can communicate efficiently through shared variables without the overhead of inter-process communication mechanisms.

Aspect	Process	Thread
Creation cost	High (duplicate address space, resources)	Low (share existing address space)
Context switch cost	High (save/restore full address space info)	Low (save/restore registers, stack)
Memory sharing	Separate address spaces; need IPC	Shared address space within same process
Independence	Fully independent	Dependent on parent process
Failure impact	Crash isolated to process	One thread crash may affect entire process

3.2 Benefits of Multithreading

Multithreading offers several important advantages. First, it increases responsiveness: if one thread blocks (e.g., waiting for network data), other threads in the same process can continue executing, keeping the application responsive to the user. Second, resource sharing between threads is efficient because threads within the same process naturally share memory and resources. Third, thread creation and context switching are far less expensive than their process-level equivalents, which is why most modern kernels are themselves multithreaded.

- Responsiveness — a blocked thread does not block the entire application
- Resource sharing — threads share code, data, and heap within the same process
- Economy — thread creation is lightweight compared to process creation
- Scalability — threads can run in parallel on multiprocessor or multicore systems

Key Concept: Threads vs Processes

A thread is a lightweight unit of execution within a process. Threads share the process's address space and resources, making them cheaper to create and switch between than full processes. Multithreading improves responsiveness, resource utilisation, and scalability.

3.3 Observing Threads in Practice

You can observe threads on your own machine using the operating system's built-in tools. On Windows, pressing Ctrl+Shift+Esc opens the Task Manager. Navigating to the Details tab reveals each running process along with its thread count. Similarly, on Linux, the htop utility or the command 'ps -eLf' lists individual threads. Examining these tools helps build intuition for how heavily modern software relies on multithreading—even a seemingly simple application such as a text editor may employ dozens of threads.

It is instructive to open the Task Manager while performing everyday tasks and note how the number of threads for applications like web browsers can reach into the hundreds. Each browser tab, extension, and background service may run in its own thread (or even its own process, in the case of Chromium-based browsers). This practical observation reinforces the theoretical concepts discussed above and demonstrates why efficient thread management is a critical operating-system capability.

Session 4: Process Scheduling

In a multiprogrammed or multitasking operating system, multiple processes compete for the CPU. The component responsible for deciding which process runs next is the process scheduler. Effective scheduling is vital: it determines system responsiveness, throughput, fairness, and overall efficiency. This session covers the queues the scheduler uses, the different types of schedulers, and the criteria by which scheduling algorithms are evaluated.

4.1 Scheduling Queues

The operating system maintains several queues to organise processes as they move through the system. When a process enters the system, it is placed on the job queue, which contains all processes in the system. Processes that are resident in main memory and are ready and waiting to execute sit on the ready queue. When a process is waiting for a particular I/O device, it is placed on the I/O queue (also called a device queue) for that device.

Processes migrate among these queues during their lifetime. A new process starts on the job queue. Once admitted to memory, it joins the ready queue. When dispatched, it moves to the CPU. From the CPU, it may return to the ready queue (if preempted), move to an I/O queue (if it initiates I/O), or terminate. Understanding this queueing model is essential for analysing system behaviour and performance.

Queue	Contents	Purpose
Job queue	All processes in the system	Holds every process from submission to completion
Ready queue	Processes in memory, ready to execute	Waiting for CPU allocation
I/O queue(s)	Processes waiting for a specific I/O device	One queue per device; processes wait for I/O completion

4.2 Types of Schedulers

Operating systems may employ multiple levels of scheduling. The long-term scheduler (or job scheduler) selects processes from the pool of submitted jobs and loads them into memory. It controls the degree of multiprogramming—that is, the number of processes simultaneously in memory. The short-term scheduler (or CPU scheduler) selects from among the processes in the ready queue and allocates the CPU to one of them. Because it runs very frequently (potentially every few milliseconds), the short-term scheduler must be extremely fast. Some systems also employ a medium-term scheduler that can temporarily remove (swap out) a process from memory to reduce the degree of multiprogramming, and later swap it back in.

Scheduler	Frequency	Role
Long-term (Job)	Infrequent	Selects jobs from disk and loads them into memory; controls

		degree of multiprogramming
Short-term (CPU)	Very frequent (ms)	Selects a ready process and dispatches it to the CPU
Medium-term (Swapper)	As needed	Swaps processes in/out of memory to manage multiprogramming level

4.3 Preemptive versus Non-Preemptive Scheduling

Scheduling decisions can be classified as either preemptive or non-preemptive. Under non-preemptive (or cooperative) scheduling, once a process is allocated the CPU, it retains the CPU until it voluntarily releases it—either by terminating or by switching to the waiting state (e.g., for I/O). Under preemptive scheduling, the operating system can forcibly remove the CPU from a running process, typically in response to a timer interrupt or the arrival of a higher-priority process.

Preemptive scheduling is used by virtually all modern general-purpose operating systems because it prevents any single process from monopolising the CPU. However, it introduces additional complexity: shared data structures may be left in an inconsistent state if a process is preempted mid-update, which necessitates synchronisation mechanisms such as mutexes and semaphores (covered in a later session).

Key Concept: Preemptive vs Non-Preemptive Scheduling

Non-preemptive scheduling lets a process run until it voluntarily yields the CPU. Preemptive scheduling allows the OS to interrupt a running process (e.g., via a timer) and switch to another, ensuring fair CPU sharing but requiring careful synchronisation.

4.4 Scheduling Criteria

Different scheduling algorithms optimise for different criteria. The five most commonly used metrics are listed below. No single algorithm can optimise all criteria simultaneously, so the choice of algorithm depends on the system's goals—for example, an interactive desktop system prioritises response time, whereas a batch-processing server prioritises throughput.

Criterion	Definition	Goal
CPU utilisation	Percentage of time the CPU is actively executing processes	Maximise
Throughput	Number of processes completed per unit time	Maximise
Turnaround time	Total time from process submission to completion (TAT = CT – AT)	Minimise

Waiting time	Total time a process spends in the ready queue (WT = TAT – BT)	Minimise
Response time	Time from submission of a request until the first response is produced	Minimise

Turnaround time and waiting time are typically the most important metrics for batch systems, while response time is critical for interactive systems. CPU utilisation and throughput are system-wide measures. When evaluating scheduling algorithms in this module, you will most often be asked to compute average waiting time and average turnaround time for a given set of processes.

Session 5: CPU Scheduling Algorithms

This session examines the four classical CPU scheduling algorithms covered in the lecture: First-Come, First-Served (FCFS); Shortest-Job-First (SJF) in both non-preemptive and preemptive variants; Priority Scheduling; and Round Robin (RR). For each algorithm, we describe the principle, work through a numerical example, and discuss its strengths and weaknesses.

5.1 First-Come, First-Served (FCFS)

First-Come, First-Served is the simplest CPU scheduling algorithm. Processes are executed in the order in which they arrive in the ready queue. It is non-preemptive: once a process begins execution, it runs to completion (or until it blocks for I/O). FCFS is easy to implement using a simple FIFO queue, but it can lead to poor average waiting times, particularly when a long process arrives before several short ones.

The main disadvantage of FCFS is the convoy effect: if a single CPU-bound process with a long burst time is at the head of the queue, all shorter processes behind it must wait, even though they could finish quickly. This inflates the average waiting time significantly.

5.1.1 Worked Example

Consider three processes arriving at time 0 with the following burst times:

Process	Burst Time (ms)
P1	5
P2	3
P3	8

Assuming the arrival order is P1, P2, P3, the Gantt chart is: P1 (0–5), P2 (5–8), P3 (8–16). The turnaround times are P1 = 5, P2 = 8, P3 = 16, giving an average turnaround time of (5 + 8 + 16) / 3 = 9.67 ms. The waiting times are P1 = 0, P2 = 5, P3 = 8, giving an average waiting time of (0 + 5 + 8) / 3 = 4.33 ms.

Key Concept: FCFS and the Convoy Effect

FCFS executes processes in arrival order. It is simple but can cause the convoy effect, where short processes are stuck waiting behind a long process, leading to poor average waiting times.

5.2 Shortest-Job-First (SJF)

Shortest-Job-First scheduling selects the process with the smallest next CPU burst for execution. SJF is provably optimal—it gives the minimum average waiting time for any given set of processes. The practical difficulty, however, is that the length of the next CPU burst is generally not known in advance; it can only be estimated (e.g., using exponential averaging of past burst lengths).

SJF exists in two variants: non-preemptive and preemptive. In the non-preemptive version, once a process starts executing it runs to completion. In the preemptive version—often called Shortest Remaining Time First (SRTF)—a currently running process can be preempted if a newly arriving process has a shorter remaining burst time.

5.2.1 Non-Preemptive SJF Example

Process	Arrival Time	Burst Time (ms)
P1	0	7
P2	2	4
P3	4	1
P4	5	4

P1 arrives first and runs from 0 to 7. At time 7, P2, P3, and P4 have all arrived. The shortest burst is P3 (1 ms), so it runs next (7–8). Then P2 and P4 both have burst time 4; P2 arrived earlier, so it runs from 8–12, followed by P4 from 12–16. The average waiting time is (0 + 6 + 3 + 7) / 4 = 4.0 ms, and the average turnaround time is (7 + 10 + 4 + 11) / 4 = 8.0 ms.

5.2.2 Preemptive SJF (SRTF) Example

Process	Arrival Time	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

At time 0, only P1 is available and begins executing. At time 1, P2 arrives with burst time 4, which is shorter than P1's remaining time of 7, so P1 is preempted and P2 runs. At time 2, P3 arrives with burst 9—longer than P2's remaining 3, so P2 continues. At time 3, P4 arrives with burst 5—longer than P2's remaining 2, so P2 continues. P2 finishes at time 5. Now the remaining bursts are P1 (7), P3 (9), P4 (5); P4 is shortest and runs from 5–10. Then P1 runs from 10–17, and finally P3 from 17–26. The average waiting time is 6.5 ms.

Key Concept: SJF Optimality

SJF (non-preemptive) is provably optimal for minimising average waiting time among non-preemptive algorithms. Its preemptive variant (SRTF) is optimal overall. The main challenge is that future burst times must be estimated, as they are not known in advance.

5.3 Priority Scheduling

In priority scheduling, each process is assigned a priority number (typically an integer), and the CPU is allocated to the process with the highest priority. By convention, a smaller integer often denotes a higher priority, although this varies by system. Priority scheduling can be either preemptive (a new, higher-priority process can interrupt the running process) or non-preemptive (the running process keeps the CPU until it finishes or blocks).

The chief problem with priority scheduling is starvation: low-priority processes may wait indefinitely if higher-priority processes continually arrive. The standard solution is aging—gradually increasing the priority of processes that have been waiting for a long time. Eventually, even the lowest-priority process will have its priority raised high enough to be scheduled.

5.3.1 Worked Example

Consider five processes, all arriving at time 0:

Process	Burst Time (ms)	Priority
P1	10	3
P2	1	1 (highest)
P3	2	4
P4	1	5 (lowest)
P5	5	2

Execution order (non-preemptive): P2 (0–1), P5 (1–6), P1 (6–16), P3 (16–18), P4 (18–19). The average waiting time is (6 + 0 + 16 + 18 + 1) / 5 = 8.2 ms, and the average turnaround time is (16 + 1 + 18 + 19 + 6) / 5 = 12.0 ms.

Key Concept: Starvation and Aging

Priority scheduling can cause starvation, where low-priority processes never execute. Aging solves this by progressively increasing the priority of waiting processes over time, guaranteeing that every process will eventually be scheduled.

5.4 Round Robin (RR)

Round Robin is the scheduling algorithm most commonly associated with time-sharing systems. Each process is assigned a fixed time quantum (or time slice). The ready queue is treated as a circular queue: the scheduler dispatches the first process in the queue and sets a timer for one quantum. If the process completes before the quantum expires, it releases the CPU voluntarily. If it does not, the timer fires an interrupt and the process is preempted and moved to the back of the ready queue.

The choice of time quantum is critical. If the quantum is too large, Round Robin degenerates into FCFS. If the quantum is too small, excessive context switching occurs and overhead dominates. A

typical quantum ranges from 10 to 100 milliseconds, and a rule of thumb is that 80% of CPU bursts should be shorter than the time quantum.

5.4.1 Worked Example

Process	Burst Time (ms)
P1	4
P2	5
P3	2

With a time quantum of 2 ms and all processes arriving at time 0, the Gantt chart is: P1 (0–2), P2 (2–4), P3 (4–6), P1 (6–8), P2 (8–10), P2 (10–11). The turnaround times are P1 = 8, P2 = 11, P3 = 6, giving an average turnaround time of (8 + 11 + 6) / 3 = 8.33 ms. The waiting times are P1 = 4, P2 = 6, P3 = 4, giving an average waiting time of (4 + 6 + 4) / 3 = 4.67 ms.

Round Robin typically produces higher average turnaround times than SJF, but it offers significantly better response times because every process receives CPU time within at most one full cycle of the ready queue. This makes it particularly suitable for interactive systems where responsiveness is paramount.

Key Concept: Round Robin Scheduling

Round Robin allocates a fixed time quantum to each process in turn. It guarantees fairness and bounded response time but may have higher average turnaround time than SJF. The quantum size is a crucial design parameter: too large becomes FCFS, too small causes excessive context-switch overhead.

5.5 Algorithm Comparison

The table below provides a high-level comparison of the four scheduling algorithms discussed in this session. Each algorithm occupies a different point in the trade-off space between simplicity, fairness, and optimality.

Algorithm	Type	Preemptive?	Optimal?	Starvation?	Key Advantage
FCFS	Non-preemptive	No	No	No	Simplest to implement
SJF (Non-preemptive)	Non-preemptive	No	Yes (for non-preemptive)	Possible	Minimum average waiting time
SJF (Preemptive / SRTF)	Preemptive	Yes	Yes (overall)	Possible	Minimum average waiting time
Priority	Either	Either	No	Yes	Supports urgency-based

					ordering
Round Robin	Preemptive	Yes	No	No	Fair, bounded response time

In practice, real operating systems often use hybrid approaches—for example, multilevel feedback queues that combine elements of priority scheduling and Round Robin. The goal is to balance the competing demands of throughput, responsiveness, and fairness in a way that suits the system's workload.

Session 6: Summary and Further Reading

This week's lecture covered the foundational concepts of process management and CPU scheduling. We began with the evolution of computing environments from batch processing through multiprogramming to multitasking. We then defined the process abstraction, explored its memory layout, and examined the five process states and the transitions between them. The Process Control Block and context-switching mechanism were discussed as the key data structures and operations that enable multitasking.

We explored process creation (using the fork/exec model in UNIX) and termination (including cascading termination and zombie/orphan processes). The concept of threads was introduced as a lightweight alternative to processes for concurrent execution within a single application.

The second half of the session focused on CPU scheduling. We covered the different scheduling queues (job, ready, and I/O queues), the three types of schedulers (long-term, short-term, and medium-term), and the distinction between preemptive and non-preemptive scheduling. Four classical scheduling algorithms were examined in detail: FCFS, SJF (both variants), Priority, and Round Robin.

Key Concept: Key Takeaways

Processes are the fundamental unit of work in an OS. They transition through five states managed by the PCB and context switches. CPU scheduling algorithms (FCFS, SJF, Priority, Round Robin) each offer different trade-offs between simplicity, optimality, fairness, and responsiveness. Real systems use hybrid approaches.

6.1 Learning Objectives Revisited

- Understand the concept of processes and their states and transitions — covered in Sessions 1–2.
- Describe scheduling queues and the role of different schedulers in an operating system — covered in Session 4.
- Explain and compare major CPU scheduling algorithms — covered in Session 5.
- Evaluate and select appropriate scheduling algorithms based on system performance criteria — covered in Sessions 4–5.

6.2 Recommended Reading

The following textbooks are recommended for deeper exploration of the topics covered in this week's lecture. Both are standard references in operating systems courses worldwide.

Reference	Title	Authors	Edition	Publisher
R1	Operating System Concepts	Silberschatz, Galvin, and Gagne	9th edition	John Wiley & Sons
R2	Modern Operating Systems	Andrew S. Tanenbaum and	4th edition (3rd also acceptable)	Pearson Education Ltd

		Herbert Bos		

For Operating System Concepts, Chapters 3 (Processes) and 6 (CPU Scheduling) are directly relevant to this week's material. For Modern Operating Systems, Chapters 2 (Processes and Threads) and the scheduling sections of Chapter 2 provide complementary explanations and additional examples.