B.TECH — COMPUTER SCIENCE & ENGINEERING

Operating Systems

CPU Scheduling & Memory Management

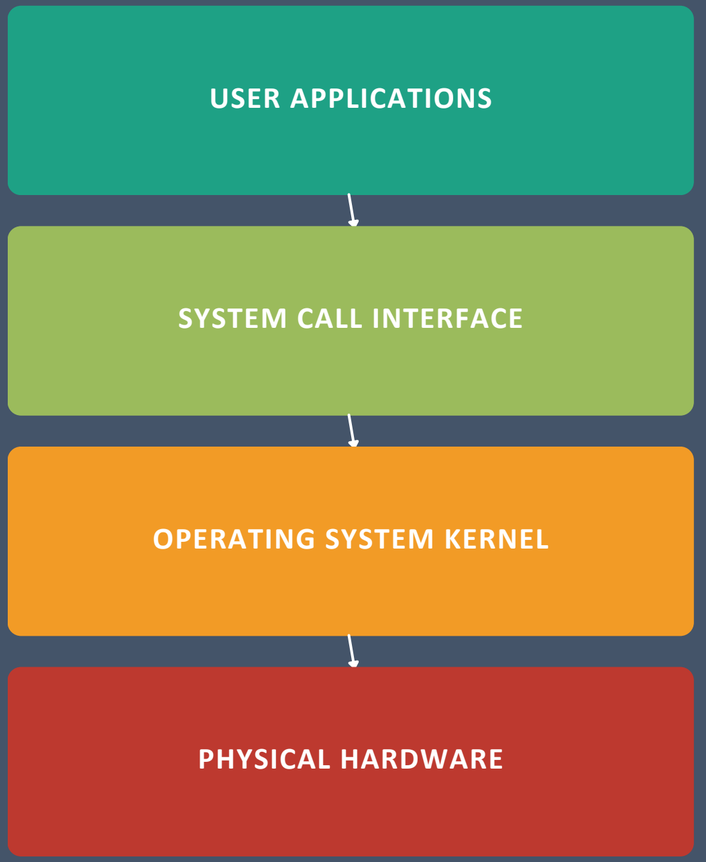
How a modern operating system decides which process runs next, and how it keeps every process's data safely in memory.

PRESENTED BY
Sakshi Awasthi

INSTITUTION
MIT WPU

DEPARTMENT
Department of Computer Science & Engineering

DATE
August 2026



Why CPU & Memory Management Matter

Every computer runs many programs at once — a browser, a music player, background updates — yet there is only one CPU (or a handful of cores) and a fixed amount of RAM. The operating system's job is to share these scarce resources fairly, safely, and efficiently.

Maximize CPU utilization

Keep the processor busy instead of idle while processes wait for I/O.

Fairness among processes

No single process should starve others of CPU time or memory.

Responsiveness

Interactive tasks must feel instant even when heavy jobs run in the background.

WHERE THIS RUNS EVERY DAY

Desktop OS

Windows / macOS juggle dozens of open apps on a few CPU cores.

Mobile OS

Android schedules foreground apps ahead of background services to save battery.

Cloud servers

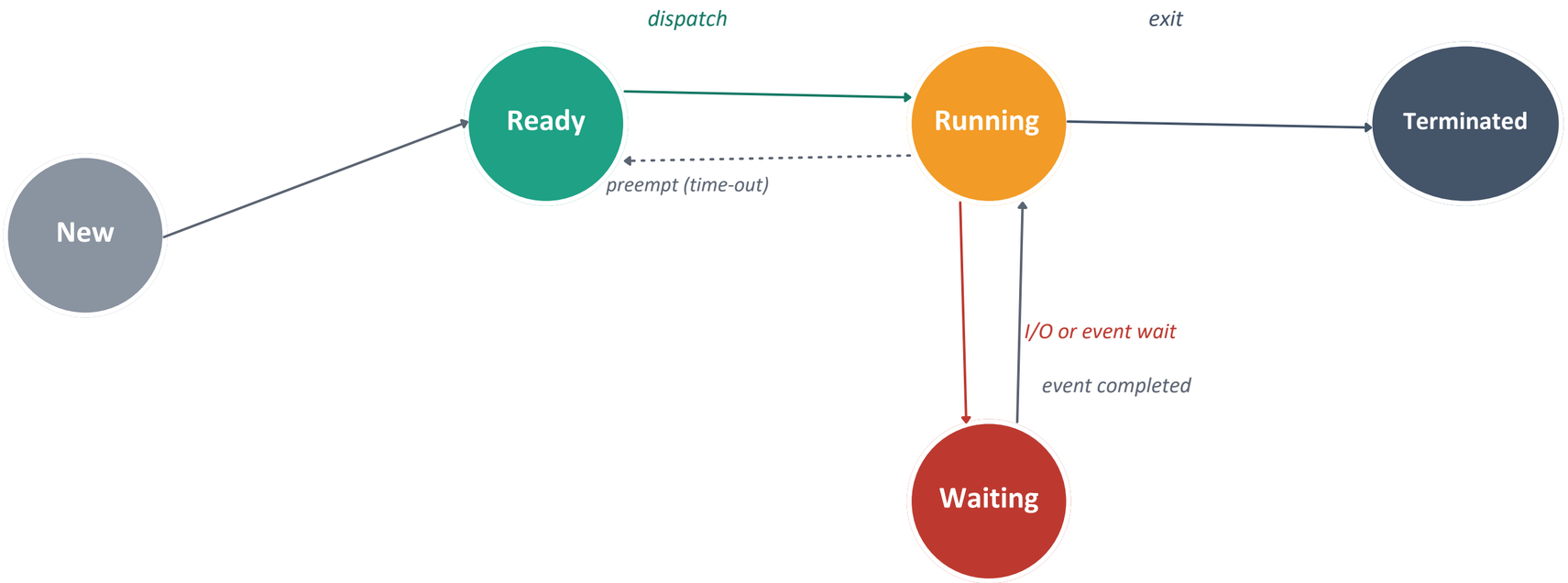
One physical server runs hundreds of virtual machines or containers.

Embedded & IoT

Smart devices manage tasks within a few kilobytes of RAM.

The Process and Its Lifecycle

A process is a program in execution. The OS represents each process with a Process Control Block (PCB) that stores its state, program counter, CPU registers, and memory limits. Every process moves through a well-defined set of states.

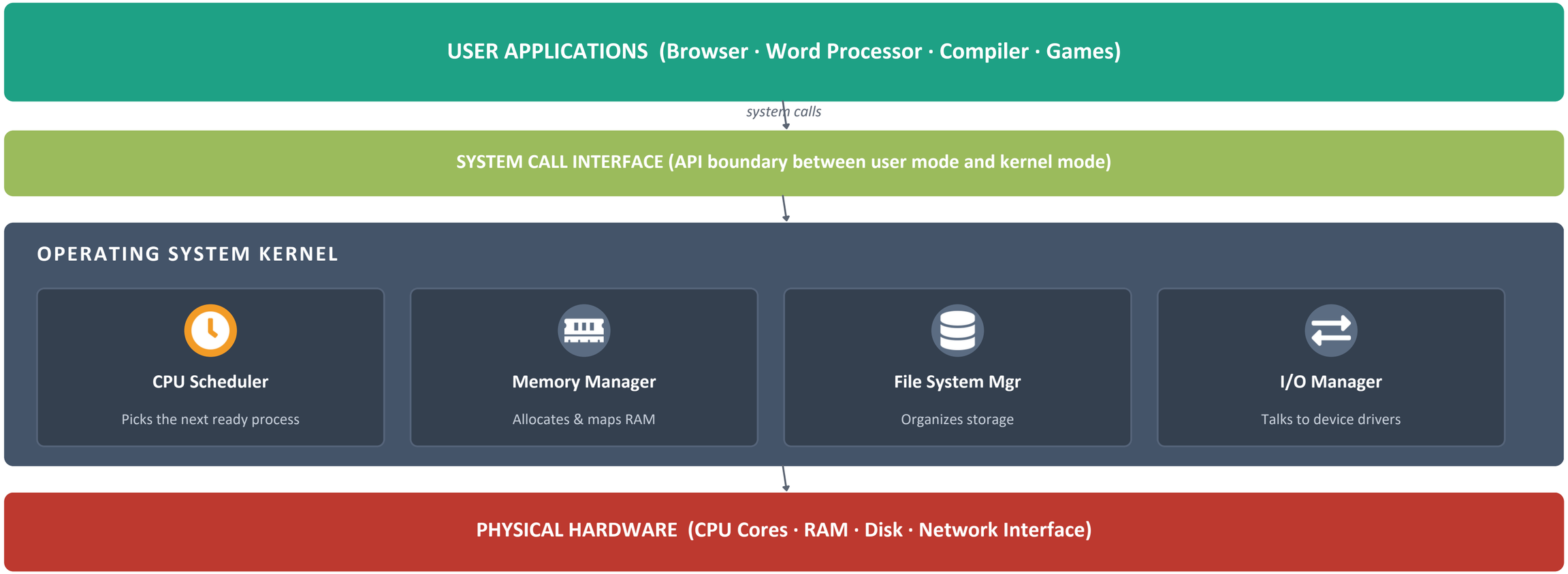


PROCESS CONTROL BLOCK (PCB) — WHAT THE OS TRACKS PER PROCESS

- Process ID (PID)
- Process state
- Program counter
- CPU registers
- CPU scheduling info
- Memory limits
- I/O status info

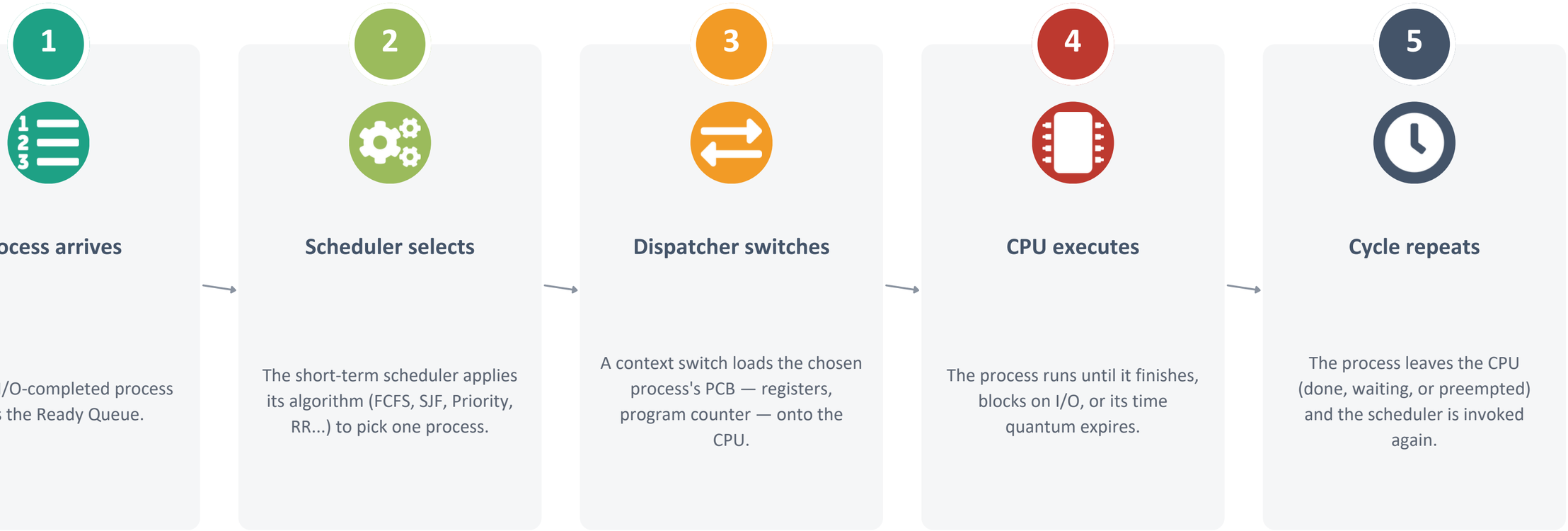
Where Scheduling & Memory Management Live

The OS kernel sits between user applications and physical hardware. Two of its core subsystems — the CPU Scheduler and the Memory Manager — are invoked constantly, on every process switch and every memory access.



The CPU Scheduling Cycle, Step by Step

Every time the CPU becomes free, the operating system repeats the same five-stage cycle to decide what runs next.



Round Robin Scheduling — Worked Example

INPUT: 4 PROCESSES, TIME QUANTUM = 4 ms

Process	Arrival	Burst
P1	0	5
P2	1	3
P3	2	8
P4	3	6

Rule: processes are queued in arrival order; each gets at most one quantum (4 ms) per turn; if not finished, it returns to the back of the ready queue.

RESULTS

Average Waiting Time = 9.25 ms

Average Turnaround Time = 14.75 ms

GANTT CHART OF EXECUTION ORDER

Process	Completion	Turnaround	Waiting
P1	16	16	11
P2	7	6	3
P3	20	18	10
P4	22	19	13

CPU Scheduling Algorithms Compared

Algorithm	Preemptive?	Selection Rule	Decision Cost	Best Suited For
FCFS	No	First process in queue order	O(1)	Simple batch systems
SJF	No*	Shortest next CPU burst first	O(log n) w/ heap	Minimizing avg. waiting time
Priority	Optional	Highest priority value first	O(log n) w/ heap	Tasks with importance levels
Round Robin	Yes	FCFS + fixed time quantum	O(1)	Time-sharing, interactive OS

**Preemptive SJF (Shortest Remaining Time First) also exists. • Decision cost = time to pick the next process from the ready queue.*

PSEUDOCODE — ROUND ROBIN DISPATCH

```
while (readyQueue is not empty) {
  p = readyQueue.dequeue()
  run(p, min(quantum, p.remainingBurst))
  if (p.remainingBurst > 0)
      readyQueue.enqueue(p)    // requeue
  else
      p.state = TERMINATED
}
```

CONTIGUOUS MEMORY ALLOCATION STRATEGIES

- **First Fit** — Allocate the first free block large enough for the request.
- **Best Fit** — Allocate the smallest free block that still fits — minimizes leftover space.
- **Worst Fit** — Allocate the largest free block — leaves a large usable remainder.

Paging vs. Segmentation

Both are memory management schemes that avoid the need for one process to sit in one contiguous block — but they divide memory on very different principles.

Basis	Paging	Segmentation
Division of memory	Fixed-size blocks (pages)	Variable-size logical units (segments)
Basis of division	Physical — decided by hardware/OS	Logical — matches program modules (code, stack, heap)
Address form	Page number + offset	Segment number + offset
Fragmentation	Internal fragmentation (unused space inside a page)	External fragmentation (gaps between segments)
Programmer visibility	Invisible / transparent	Visible — reflects program structure
Protection & sharing	Harder to share logically related data	Natural to protect/share at module level
Mapping table	Page table (per process)	Segment table (per process)

Scheduling Concepts in Production Systems

Linux — CFS

The Completely Fair Scheduler keeps runnable tasks in a red-black tree keyed by virtual runtime, always picking the leftmost (least-served) task.

Windows

A multilevel feedback queue with dynamic priority boosting — interactive threads are temporarily promoted after waiting on I/O.

Android

Built on the Linux kernel scheduler, extended with wakelocks and thermal-aware throttling to balance responsiveness and battery life.

Real-Time OS

Systems like FreeRTOS/VxWorks use Rate-Monotonic or Earliest-Deadline-First scheduling so critical tasks never miss a deadline.

Kubernetes / Cloud

Container orchestrators set CPU requests & limits enforced through Linux cgroups, layering fair-share quotas over CFS.

Database Servers

Query engines schedule worker threads across cores and manage buffer-pool memory with LRU-style page replacement.

Advantages, Limitations & Challenges

Advantages

- Maximizes CPU utilization instead of leaving cores idle
- Enables safe multitasking across many processes
- Improves responsiveness for interactive workloads
- Virtual memory lets programs use more memory than physically exists

Limitations

- Context switching consumes CPU cycles — pure overhead
- No single algorithm is optimal for every workload mix
- Fairness and low latency are often conflicting goals
- More complex algorithms cost more to compute per decision

Challenges

- Starvation: low-priority processes may wait indefinitely
- Thrashing: excessive paging when memory is over-committed
- Internal / external fragmentation waste usable memory
- Priority inversion can stall high-priority real-time tasks

Cloud Web Server Handling Mixed Workloads

THE PROBLEM

A multi-core cloud VM serves short, interactive API requests alongside long-running batch analytics jobs. Under plain FCFS scheduling, one long batch job blocks every short request behind it (the convoy effect), and frequent process creation fragments memory as short-lived handler processes are spawned and destroyed.

THE SOLUTION — MULTILEVEL FEEDBACK QUEUE + DEMAND PAGING

New/interactive jobs start in Queue 0

so API requests get near-instant response.

A job using its full quantum is demoted

long batch jobs sink to Queue 2 automatically.

Demand paging + LRU replacement

only active pages of each job occupy RAM.

cgroup CPU quotas cap batch jobs

so they can never fully starve interactive traffic.

CONCLUSION

Key Takeaways

The OS constantly cycles processes through New → Ready → Running → Waiting → Terminated.

Scheduling algorithms trade off simplicity (FCFS), fairness (Round Robin) and optimality (SJF/Priority).

Every algorithm choice is measurable — waiting time and turnaround time quantify performance.

Paging and segmentation solve memory allocation differently: physical fixed blocks vs. logical variable units.

Real systems (Linux CFS, Kubernetes) blend these textbook ideas at massive scale.

Overhead, starvation, and fragmentation are the recurring costs every design must manage.

Future Scope: multicore-aware & energy-aware scheduling, NUMA-conscious memory placement, and ML-driven adaptive schedulers.

Thank You !

Sources & Further Reading

Silberschatz, A., Galvin, P. B., & Gagne, G.

Operating System Concepts (10th ed.). Wiley.

Tanenbaum, A. S., & Bos, H.

Modern Operating Systems (4th ed.). Pearson.

The Linux Kernel Documentation

“CFS Scheduler” — kernel.org/doc/html/latest/scheduler/sched-design-CFS.html

Kubernetes Documentation

“Resource Management for Pods and Containers” — kubernetes.io/docs

All process data, timings, and computed results in this presentation (Slide 6) are original worked examples created for this presentation, not sourced statistics.