Unit – 3 : CPU Scheduling & Deadlock

OS Unit | Exam Notes | Bullets + Examples

*Brought to you by **ZenYukti***

PART A — CPU SCHEDULING

1. Scheduling Concepts & Performance Criteria

- CPU Scheduling = deciding which process in the ready queue gets the CPU next
- Goal: maximise CPU utilisation; minimise waiting time and response time
- Scheduler runs when: process terminates, does I/O, time slice expires, or new process arrives

Performance Criteria

Criterion	Definition	Optimise?
CPU Utilisation	% of time CPU is busy doing useful work	Maximise (target 40–90%)
Throughput	Number of processes completed per unit time	Maximise
Turnaround Time	Total time from submission to completion	Minimise
Waiting Time	Time spent in the ready queue (not executing)	Minimise
Response Time	Time from request submission to first response	Minimise

- Turnaround Time = Completion Time − Arrival Time
- Waiting Time = Turnaround Time − Burst Time
- Response Time = Time of first CPU allocation − Arrival Time

2. Process States & Process Transition Diagram

Five Process States

State	Description
New	Process is being created; not yet admitted to ready queue
Ready	Process is in memory, waiting for CPU — fully prepared to run
Running	Process is currently executing on the CPU
Waiting/Blocked	Process waiting for I/O or event — cannot use CPU until done
Terminated	Process has finished execution; PCB being deallocated

Process Transition Diagram

- New → Ready: admitted by long-term scheduler (job scheduler)
- Ready → Running: selected by short-term scheduler (CPU scheduler)
- Running → Ready: preempted (time slice expired or higher-priority process arrives)
- Running → Waiting: process issues I/O request or waits for event
- Waiting → Ready: I/O completes or event occurs
- Running → Terminated: process finishes or is killed

> *Only Running→Waiting and Running→Ready are initiated by the process itself or the OS respectively. Waiting→Running never happens directly.*

3. Schedulers

Scheduler	Also Called	Frequency	Role
Long-Term	Job Scheduler	Infrequent (seconds/minutes)	Selects which jobs move from disk to ready queue; controls degree of multiprogramming
Short-Term	CPU Scheduler	Very frequent (ms)	Selects which ready process gets the CPU next; fastest scheduler
Medium-Term	Swapper	Moderate	Temporarily removes processes from memory (swap out) and reloads (swap in) to manage memory

- Preemptive scheduling: OS can forcibly take CPU from a running process
- Non-preemptive scheduling: process holds CPU until it finishes or does I/O voluntarily

4. Process Control Block (PCB)

- PCB = data structure maintained by OS for every process
- Also called Task Control Block (TCB)
- Stored in kernel memory; one PCB per process

PCB Contents

- Process Identification: PID, parent PID, user ID
- Process State: New / Ready / Running / Waiting / Terminated
- Program Counter (PC): address of next instruction to execute
- CPU Registers: all register values saved when process is preempted
- CPU Scheduling Info: priority, scheduling queue pointers
- Memory Management Info: page tables, segment tables, base/limit registers
- I/O Status Info: list of open files, I/O devices allocated
- Accounting Info: CPU time used, time limits, job number

> *Context switch = save current process's PCB + load next process's PCB. This is pure overhead — no useful work happens during it.*

5. Process Address Space

- Each process has its own virtual address space — isolated from others
- Divided into segments:
  - Text segment — executable code (read-only, shared between same-program processes)
  - Data segment — global and static variables; further split:
    - Initialised data (.data) — variables with initial values
    - Uninitialised data (.bss) — zero-initialised globals
  - Heap — dynamic memory (malloc/new); grows upward
  - Stack — function call frames, local variables, return addresses; grows downward
- Stack and Heap grow toward each other; OS detects overflow
- OS maintains memory maps via page tables / segment tables per process

6. Process Identification Information

- PID (Process ID) — unique integer identifying the process
- PPID (Parent PID) — PID of the process that created this one
- UID (User ID) — identifies which user owns the process

- GID (Group ID) — group the user belongs to
- Process Group ID — for signaling groups of related processes
- Session ID — group of process groups (e.g., a login session)
- UNIX: fork() creates child with new PID; child inherits parent's UID/GID

7. Threads and Their Management

- Thread = lightweight process; a unit of execution within a process
- A process can have multiple threads sharing the same address space

What Threads Share vs. Own

Shared (per Process)	Private (per Thread)
Code segment (text)	Thread ID
Data segment + Heap	Program Counter (PC)
Open files & sockets	Register set
Signal handlers	Stack (local variables, function calls)
Process ID, UID	State (Running/Waiting/etc.)

User-Level vs Kernel-Level Threads

- User-Level Threads (ULT): managed by user-space library (e.g., POSIX pthreads)
  - OS sees only one process; thread switch = no kernel involvement → fast
  - Blocking one thread blocks the whole process (OS doesn't know about others)
- Kernel-Level Threads (KLT): OS manages each thread directly
  - True parallelism on multi-core; one thread blocking doesn't block others
  - Thread creation/switch = syscall overhead → slower than ULT
- Hybrid (M:N model): M user threads mapped to N kernel threads — best of both

Thread Management Operations

- Create — spawn new thread within the process
- Join — wait for a thread to finish (synchronisation)
- Detach — thread runs independently; resources freed on completion
- Cancel — terminate another thread (safe points needed)
- Synchronisation: Mutex, Semaphore, Condition Variables to avoid race conditions

> *Threads share heap but have separate stacks. This is why global variables can cause race conditions but local variables are thread-safe.*

8. CPU Scheduling Algorithms

1. FCFS — First Come First Served

- Non-preemptive; processes served in arrival order (FIFO queue)
- Convoy Effect: short processes wait behind long ones → high average waiting time

- Example: P1(burst=24), P2(burst=3), P3(burst=3), all arrive at t=0
- Gantt: | P1(0-24) | P2(24-27) | P3(27-30) |
- Avg Waiting = (0 + 24 + 27) / 3 = 17 ms

2. SJF — Shortest Job First

- Non-preemptive; select process with shortest burst time next
- Optimal for minimising average waiting time (among non-preemptive)
- Problem: requires knowing burst time in advance — impractical; starvation of long processes

- Example: P1(bt=6,at=0), P2(bt=8,at=0), P3(bt=7,at=0), P4(bt=3,at=0)
- Order: P4(0-3) → P1(3-9) → P3(9-16) → P2(16-24)
- Avg Waiting = (3 + 16 + 9 + 0) / 4 = 7 ms (FCFS would give 10.25 ms)

3. SRTF — Shortest Remaining Time First (Preemptive SJF)

- Preemptive version of SJF: if new process has shorter remaining time → preempt current
- Best average waiting time overall; heavy context switch overhead

- Example: P1(at=0,bt=8), P2(at=1,bt=4), P3(at=2,bt=9), P4(at=3,bt=5)
- t=0: P1 starts. t=1: P2 arrives (rem=4 < P1 rem=7) → P1 preempted
- t=5: P2 done. t=5: check remainders: P1=7, P3=9, P4=5 → P4 runs
- t=10: P4 done → P1(rem=7) → t=17: P1 done → P3 → t=26 done
- Avg Waiting = (9 + 0 + 15 + 2) / 4 = 6.5 ms

4. Round Robin (RR)

- Preemptive; each process gets a fixed Time Quantum (TQ) then preempted
- Fair — no starvation; best for time-sharing systems
- Large TQ → behaves like FCFS. Small TQ → many context switches, high overhead
- Optimal TQ: 80% of CPU bursts should be shorter than the quantum

- Example: P1(bt=24), P2(bt=3), P3(bt=3), TQ=4
- Gantt: |P1(0-4)|P2(4-7)|P3(7-10)|P1(10-14)|P1(14-18)|P1(18-22)|P1(22-26)|P1(26-30)|
- Avg Waiting = (6 + 4 + 7) / 3 = 5.67 ms

5. Priority Scheduling

- Each process assigned a priority number; CPU given to highest priority
- Can be preemptive or non-preemptive
- Problem: Starvation — low-priority processes may never execute
- Solution: Aging — gradually increase priority of waiting processes over time
- SJF is a special case of priority scheduling (priority = inverse of burst time)

6. Multilevel Queue Scheduling

- Ready queue split into multiple separate queues by process type
- Common queues: System > Interactive > Batch
- Each queue has its own scheduling algorithm (e.g., RR for interactive, FCFS for batch)
- Fixed priority among queues — lower queues only run if higher queues are empty
- No movement between queues — process belongs to one queue for its lifetime

7. Multilevel Feedback Queue (MLFQ)

- Like Multilevel Queue but processes CAN move between queues
- New process enters highest-priority queue; if it uses full quantum → demoted one level
- Process that waits too long → promoted (aging) to prevent starvation
- Most flexible and widely used in practice (e.g., Linux, Windows task scheduler)

- Parameters: number of queues, scheduling per queue, promotion/demotion rules, entry queue

Algorithm	Preemptive?	Starvation?	Best For
FCFS	No	No	Batch systems; simple
SJF	No	Yes	Minimise avg waiting (theory)
SRTF	Yes	Yes	Optimal avg wait; impractical
Round Robin	Yes	No	Time-sharing, fairness
Priority	Both	Yes	Real-time, importance-based
MLFQ	Yes	No (aging)	General-purpose OS

9. Multiprocessor Scheduling

- Multiple CPUs share the workload — more complex than single-CPU scheduling

Approaches

- Asymmetric Multiprocessing (AMP): one master CPU runs the OS; others run user processes
  - Simple; master can become bottleneck
- Symmetric Multiprocessing (SMP): each CPU schedules itself from shared/private ready queue
  - All modern OS use this (Linux, Windows)

Key Issues

- Processor Affinity: keep process on same CPU to exploit warm cache
  - Soft affinity — OS tries but may migrate; Hard affinity — process is pinned
- Load Balancing: distribute work evenly across CPUs
  - Push migration: overloaded CPU pushes tasks to idle CPUs
  - Pull migration: idle CPU pulls tasks from busy CPUs
- Cache Affinity vs Load Balance conflict: migrating process invalidates cache entries
- NUMA (Non-Uniform Memory Access): memory access time differs by CPU-to-memory distance
  - NUMA-aware schedulers prefer to run process near its memory

---

PART B — DEADLOCK

10. System Model

- System has finite resources: CPU cycles, memory, files, I/O devices, semaphores
- Resources categorised as: Reusable (CPU, memory) vs Consumable (signals, messages)
- Process resource usage cycle: Request → Use → Release
- Deadlock = set of processes each waiting for a resource held by another process in the set
- No process can proceed — permanent block unless OS intervenes

Resource Allocation Graph (RAG)

- Directed graph: Processes = circles, Resources = rectangles, Instances = dots inside
- Request edge: P → R (process requesting resource)
- Assignment edge: R → P (resource instance assigned to process)
- If graph has NO cycle → no deadlock
- If graph has a cycle AND each resource has only ONE instance → deadlock exists
- If graph has a cycle AND resources have multiple instances → deadlock POSSIBLE (not certain)

11. Deadlock Characterisation

- Deadlock requires ALL FOUR conditions simultaneously (Coffman, 1971):

Condition	Meaning	Example
Mutual Exclusion	At least one resource must be non-shareable; only one process at a time	Printer held exclusively by one process
Hold and Wait	Process holding resources can request more without releasing current ones	P1 holds mutex1 and waits for mutex2
No Preemption	OS cannot forcibly take a resource from a process holding it	Cannot take lock away from a running process
Circular Wait	A circular chain of processes each waiting for a resource held by the next	P1→P2→P3→P1: each holds what the next needs

> *Remove ANY ONE condition → no deadlock possible. This is the basis of all prevention strategies.*

12. Deadlock Prevention

- Ensure at least one of the four conditions never holds — by design

1. Negate Mutual Exclusion

- Make resources shareable (e.g., read-only files) — cannot work for inherently exclusive resources

2. Negate Hold and Wait

- Approach A: Process must request ALL resources before starting — all or nothing
  - Problem: low utilisation; process may wait forever (starvation)
- Approach B: Release all held resources before requesting new ones
  - Problem: process loses progress made so far

3. Allow Preemption

- If a process can't get a resource, preempt its held resources and restart later
- Works for CPU, memory — not for printers, tape drives (state cannot be saved easily)

4. Negate Circular Wait

- Impose total ordering on resource types; processes must request in increasing order only
- If P1 holds R1 and wants R2, and R1 < R2, no circular wait can form
- Most practical prevention strategy — widely used in OS design

> *Ordering resources prevents circular wait: always acquire locks in the same global order.*

13. Deadlock Avoidance

- OS knows in advance the maximum resources each process may need
- Before granting a request, OS checks: will this lead to an UNSAFE state?
- Safe State = there exists a safe sequence where all processes can finish
- Unsafe State ≠ deadlock, but deadlock MIGHT occur

Banker's Algorithm (Dijkstra)

- Works when: multiple instances of each resource type, and max needs known in advance
- Data structures (n processes, m resource types):
  - Available[m] — number of available instances of each resource
  - Max[n][m] — max demand of each process for each resource

- Allocation[n][m] — currently allocated to each process
- Need[n][m] = Max − Allocation — still needed by each process

Safety Algorithm

```
Work = Available; Finish[i] = false for all i
Find i such that: Finish[i]=false AND Need[i] <= Work
If found: Work += Allocation[i]; Finish[i] = true; repeat
If all Finish[i] = true → SAFE STATE; else → UNSAFE
```

Resource-Request Algorithm

```
If Request[i] > Need[i] → error (exceeded max)
If Request[i] > Available → wait
Tentatively allocate; run Safety Algorithm
If safe → grant; else → rollback and wait
```

Banker's Algorithm Example

- 3 resources (A=10, B=5, C=7 total). 5 processes P0–P4

Process	Alloc A B C	Max A B C	Need A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

- Available = Total − ΣAlloc = (10,5,7) − (7,2,5) = (3,3,2)
- Safe sequence: P1→P3→P4→P2→P0 ✓ (system is in safe state)

14. Deadlock Detection

- OS does NOT prevent or avoid — lets deadlock occur, then detects and recovers
- Uses a detection algorithm periodically or when CPU utilisation drops sharply

Single Instance — Wait-For Graph

- Simplified RAG: remove resource nodes; draw P_i → P_j if P_i waits for a resource held by P_j
- If wait-for graph has a cycle → DEADLOCK DETECTED
- Cycle detection: DFS on the wait-for graph; $O(n^2)$

Multiple Instances — Detection Algorithm

- Similar structure to Banker's Algorithm safety check, but no Max/Need required
- Data: Available, Allocation, Request (current pending requests)

```
Work = Available; Finish[i] = false if Allocation[i]≠0, else true
Find i: Finish[i]=false AND Request[i] <= Work
If found: Work += Allocation[i]; Finish[i]=true; repeat
Any Finish[i] = false → process i is DEADLOCKED
```

When to Run Detection?

- Every resource request — overhead too high

- Periodically (e.g., every hour) — may allow long deadlock to persist
- When CPU utilisation drops below a threshold — practical compromise

15. Recovery from Deadlock

Option 1 — Process Termination

- Abort all deadlocked processes — guaranteed fix but loses all work
- Abort one at a time, run detection after each — fewer losses but more overhead
- Selection criteria for victim: priority, burst time used, resources held, number of victims

Option 2 — Resource Preemption

- Forcibly take resources from one process and give to another
- Three issues to handle:
  - Selecting a victim: minimise cost (time wasted, restart cost)
  - Rollback: return preempted process to safe state; full rollback or partial
  - Starvation: ensure same process is not always chosen as victim (count rollbacks)

Comparison: Prevention vs Avoidance vs Detection

Strategy	Knowledge Needed	Overhead	Utilisation	Key Mechanism
Prevention	None	Low (static rules)	Low (restrictive)	Break one of 4 conditions by design
Avoidance	Max resource needs	Medium (per request)	Medium	Banker's Algorithm — safe state check
Detection	None	Low-Medium (periodic)	High (no restriction)	Wait-for graph / detection algorithm + recovery

> *Deadlock prevention = restrictive but simple. Avoidance = flexible but needs future info. Detection = most permissive but requires recovery.*

END OF NOTES