Lecture 09

Process Description & Control

PCB · Process States · State Models · Context Switch · CPU Scheduling

SENG 21213 · Computer Architecture & Operating Systems

Learning Objectives — By the end of this lecture you should be able to:

- 1. List and explain all eight PCB field groups and justify why each is necessary
- 2. Draw the five-state and seven-state process models with all transitions and triggering events
- 3. Trace the seven-step context switch procedure identifying what is saved, where, and when
- 4. Apply FCFS, SJF, SRTF, Round-Robin, and Priority scheduling to a set of processes and compute average waiting and turnaround times
- 5. Explain why suspended states are needed and describe the conditions for swapping
- 6. Describe Stage 1 implementation requirements for the OS assignment

Section 1 · Process Control Block (PCB)

📖 Stallings Reference

- Chapter 3: Process Description and Control
- Section 3.1 — What Is a Process? Section 3.3 — Process Description

◆ PCB — Definition

- The Process Control Block (PCB) is the most important data structure in the OS. It is the OS's complete representation of a process — containing everything needed to suspend a running process and resume it later as if the interruption never occurred.
- When a process is created, a PCB is allocated and initialised. When the process terminates, the PCB is deallocated.
- The PCB is analogous to a student's complete academic record — everything about that student, accessible by anyone with the right permissions.

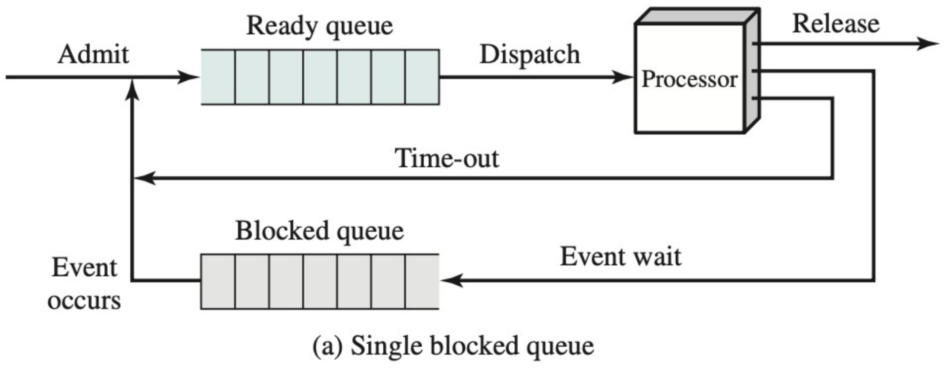


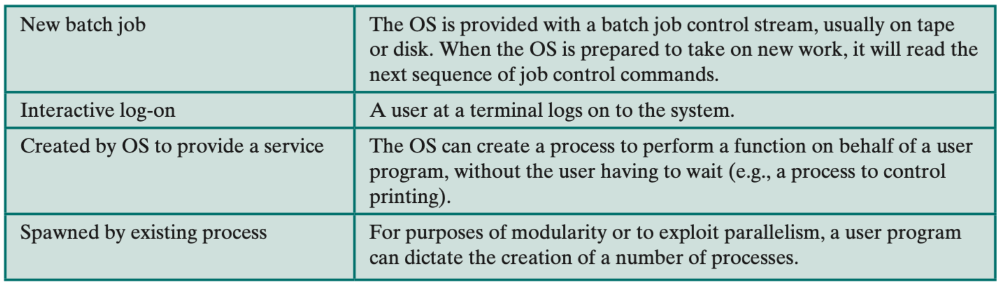
Figure: PCB structure showing the three field groups: identification, processor state, and process control information (Stallings Fig. 3.3)

1.1 The Eight PCB Field Groups

Field Group	Contents	Why Needed
Process Identifier (PID)	Unique integer ID; secondary IDs: parent PID (PPID), user ID (UID), group ID (GID).	Cross-references all OS tables. Used by system calls like wait(), kill(), nice().
Processor State Information	All CPU registers saved here on context switch: PC (EIP), EFLAGS, EAX–EDX, ESI, EDI, ESP, EBP, FPU state.	These are the exact values needed to resume execution precisely where it was interrupted.
Process State	Current state: New, Ready, Running, Blocked, or Suspended (see models below).	Determines which queue the process belongs to and what the dispatcher may do with it.
Priority	Scheduling priority level. May be static (assigned at creation) or dynamic (adjusted based on behaviour).	The short-term scheduler uses this to decide which Ready process to run next.
Memory Pointers	Pointers to the process's code segment, data segment, heap, and shared memory blocks.	OS needs these to map the virtual address space when the process is dispatched.
Context Data	Complete snapshot of all processor registers at the most recent context switch.	Enables the process to resume execution transparently — registers restored from this.
I/O Status Information	List of pending I/O requests; assigned I/O devices; open file descriptor table (pointers into the global file table).	OS tracks what I/O the process is waiting for; needed to unblock it when I/O completes.
Accounting Information	CPU time used (total and this quantum); real time elapsed; memory high-water mark; page fault count; I/O operations.	Used for scheduling (CPU-bound vs. I/O-bound classification), billing, and performance analysis.

Section 2 · Process State Models

2.1 Two-State Model — Simplest Approach



New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Figure: Two-state model: Running and Not Running; all non-running processes in one queue (Stallings Fig. 3.5)

⚠️ Limitation of Two-State Model

- The 'Not Running' state conflates two fundamentally different situations:
- 1. Processes that are READY to execute — they are just waiting for the CPU to become available.
- 2. Processes that are BLOCKED — they cannot execute even if the CPU is available because they are waiting for I/O or a synchronisation event.
- The dispatcher cannot blindly pick the next process from the queue — it might pick a blocked one, wasting the dispatch. The five-state model solves this.

2.2 Five-State Model — Standard OS Model

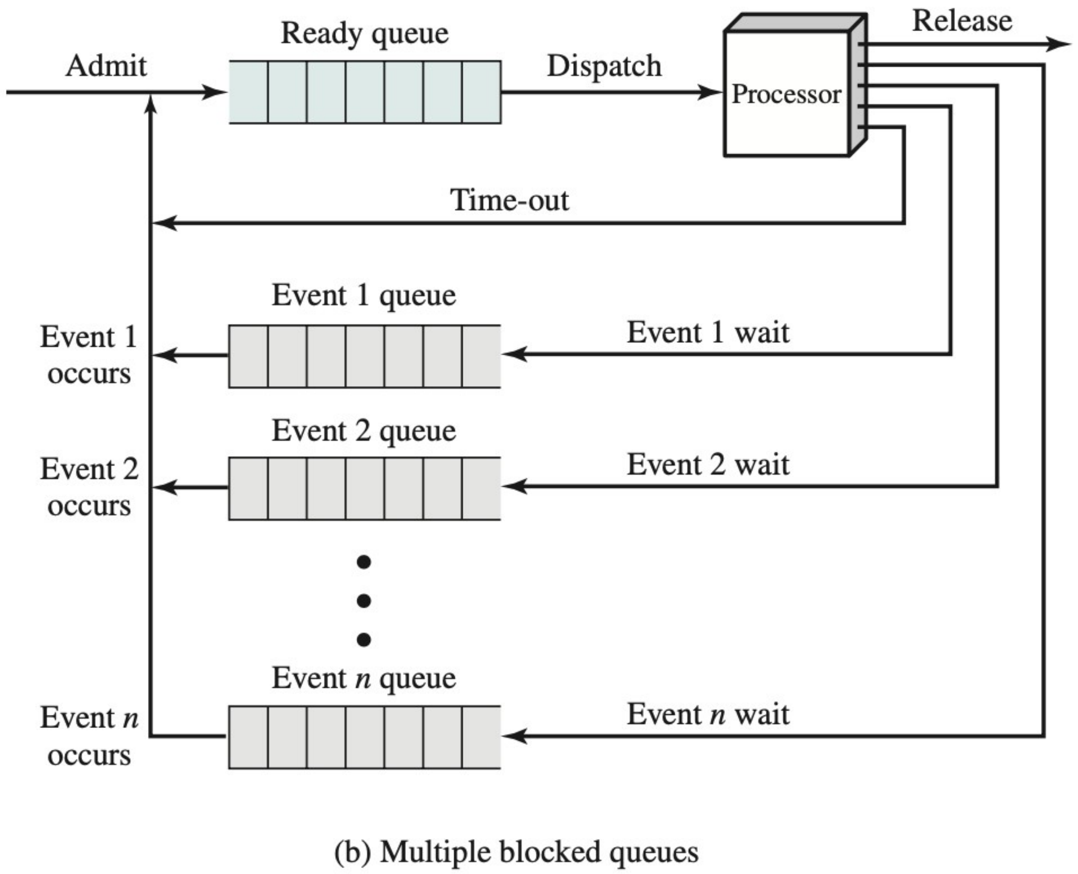


Figure: Five-state process model with all valid transitions (Stallings Fig. 3.6)

State	Description
New	Process being created — PCB allocated, resources assigned, but not yet admitted to the ready pool. Some systems limit active process count.
Ready	Process is in main memory with all resources available — waiting only for a CPU time slot. The ready queue holds all Ready processes.
Running	Currently executing on a CPU. On a single-processor system, exactly one process is Running at any time.
Blocked/ Waiting	Cannot proceed — waiting for an event (I/O completion, semaphore signal, child exit). Even if CPU is free, this process cannot use it.
Exit/Zombie	Process has finished (normal return or abort). PCB held briefly so the parent can collect the exit status. Then fully deallocated.

2.3 All Nine Transitions in the Five-State Model

Transition	Event That Causes It
Null → New	fork() or CreateProcess() system call creates a new process.
New → Ready	OS admits the process: memory allocated, PCB initialised. Added to ready queue.

Transition	Event That Causes It
Ready → Running	Short-term scheduler (dispatcher) selects this process. Context restored from PCB.
Running → Exit	process calls exit(), returns from main(), or is killed by a signal (e.g. SIGKILL).
Running → Ready	(1) Time quantum expires — preemption by clock interrupt. (2) Higher-priority process becomes ready — preemptive priority scheduling.
Running → Blocked	Process requests I/O (read, write), calls sem_wait(), or calls wait() for a child.
Blocked → Ready	I/O completes (interrupt from device); semaphore signalled; waited-for event occurs.
Ready → Exit	Parent terminates the child process before it ever ran (rare; e.g. parent calling kill() on child PID).
Blocked → Exit	Parent terminates a blocked child process.

2.4 Seven-State Model — Adding Swapping

When ALL processes in main memory are blocked, the CPU has nothing to do. The OS can SWAP (suspend) a blocked process to disk, freeing memory for a ready process.

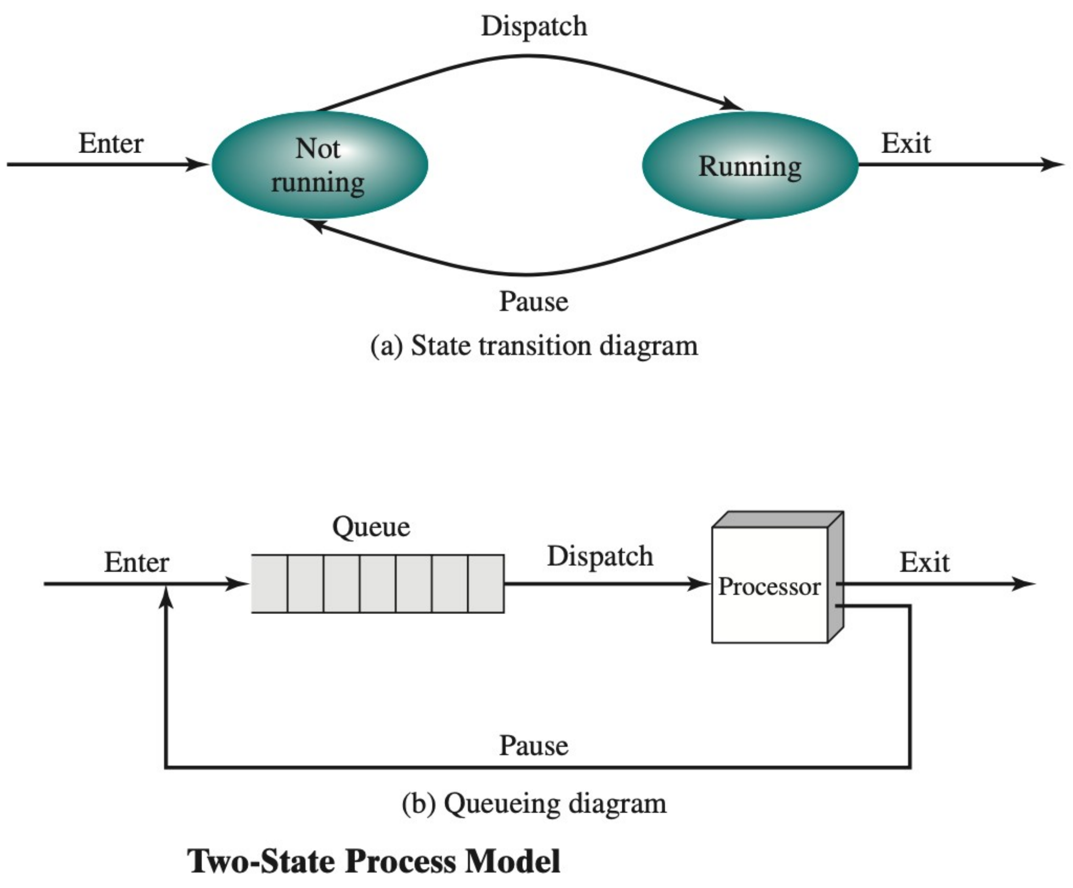


Figure: Seven-state model adding Blocked/Suspend and Ready/Suspend states with swapping transitions (Stallings Fig. 3.8)

New State	Description
Ready/Suspend	Process is on disk but logically ready — will run as soon as it is swapped back into memory.
Blocked/Suspend	Process is on disk and still waiting for an event. When event occurs: transitions to Ready/Suspend, not Ready (still on disk).
Transition	**Trigger**
Blocked → Blocked/Suspend	OS swaps out a blocked process to free memory for a more urgent ready process.
Blocked/Suspend → Ready/Suspend	Awaited event occurs (I/O completes for a suspended process).
Ready/Suspend → Ready	(1) No Ready processes in memory, OR (2) Ready/Suspend process has higher priority than all Ready processes.
Ready → Ready/Suspend	OS needs memory; this ready process is lower priority than incoming processes.

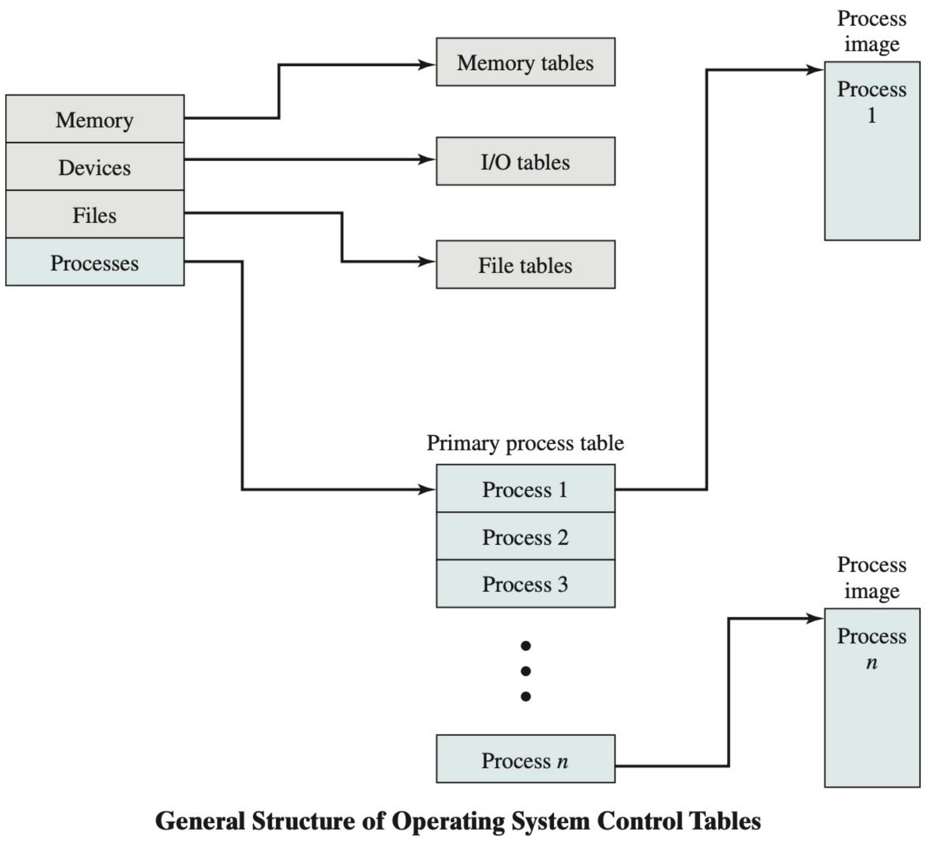


Figure: Queue representation of the seven-state model showing memory queues and suspend queues (Stallings Fig. 3.9)

Section 3 · Process Creation and Context Switching

3.1 Process Creation — Five Steps

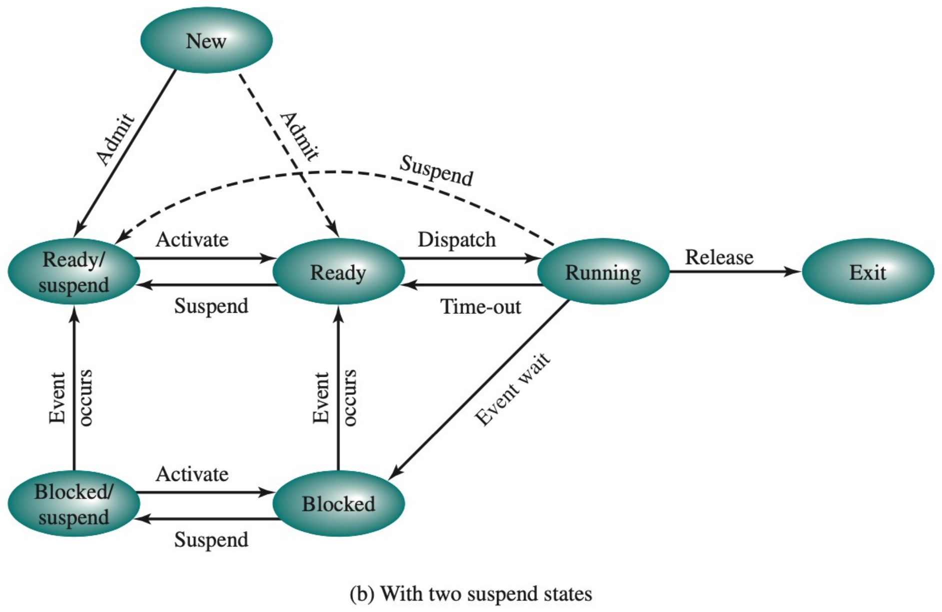


Figure: OS control structures created during process creation: process table entry, PCB, memory table entries (Stallings Fig. 3.11)

1. Assign a unique PID. Add a new entry to the primary process table pointing to the new PCB.
2. Allocate space for the process image: code segment, data segment, stack segment (grows downward), and PCB itself.
3. Initialise the PCB: PID, PPID, state=New, PC=entry_point, stack_pointer, default priority. Inheritance from parent: UID, GID, open file descriptors, signal handlers.
4. Set up scheduling linkages: change state to Ready; insert PCB pointer into the ready queue.
5. Create/expand supporting data structures: file descriptor table, signal handler table, virtual memory maps, security context.

3.2 Context Switch — Seven Steps

A context switch transfers the CPU from one process to another. It takes 1–100 µs depending on architecture (register count, TLB flush, cache effects).

6. Save context of the currently running process: PC, all registers (EAX–EBP, EFLAGS, FPU state) → store in current process's PCB.
7. Update the current PCB: change state to Ready (if preempted) or Blocked (if waiting for event); update accounting fields (CPU time used).
8. Move current PCB to appropriate queue: Ready queue (preemption) or Blocked-on-event-i queue.
9. Run the short-term scheduler: select next process from the Ready queue using the scheduling algorithm.
10. Update selected PCB: change state from Ready to Running; record dispatch time for accounting.
11. Update memory management: load CR3 with new process's page table base address (flushes TLB on x86). Update segment register limits if needed.

12. Restore context: load all registers from selected PCB (EAX, EBX, …, EBP, ESP, EIP, EFLAGS). CPU resumes execution at the restored EIP.

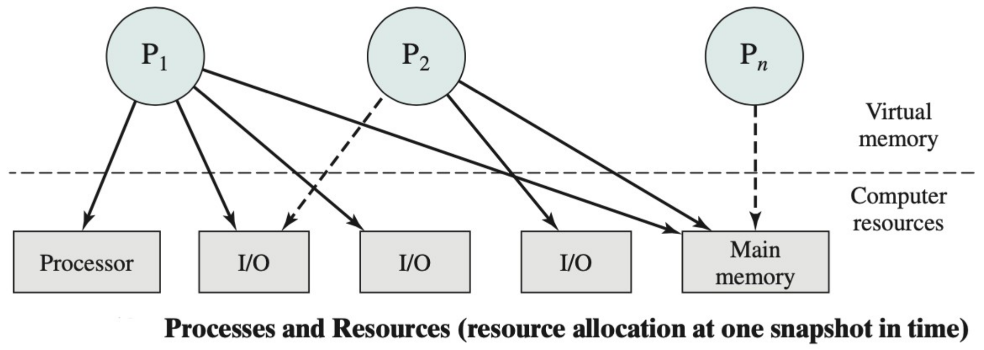


Figure: User mode and kernel mode execution — system calls and interrupts cause ring transitions (Stallings Fig. 3.12)

Section 4 · CPU Scheduling Algorithms

4.1 Scheduling Criteria

Criterion	Optimise For	Favours
CPU Utilisation	Maximise fraction of time CPU is executing useful work (target: 80–90%)	Long CPU bursts; few I/O waits
Throughput	Maximise completed processes per unit time	Short jobs; processes that finish quickly
Turnaround Time	Minimise time from submission to completion for each process	Short jobs; FCFS for single long jobs
Waiting Time	Minimise total time spent in the ready queue	Short burst lengths
Response Time	Minimise time from request to first response (interactive users)	Frequent short quanta; priority for interactive processes
Fairness	No process waits indefinitely (no starvation)	Ageing; round-robin with bounded waiting

4.2 Five Core Scheduling Algorithms

Algorithm	Selection Rule	Preemptive?	Problem	Best For
FCFS	Longest waiting process runs first	No	Convoy effect: short jobs stuck behind long ones	Simple batch
SJF	Shortest CPU burst next	No	Starvation of long jobs; needs burst	Minimise average wait time (theoretical optimal

Algorithm	Selection Rule	Preemptive?	Problem	Best For
			prediction	non-preemptive)
SRTF	Shortest REMAINING burst	Yes	Frequent preemption overhead; needs burst prediction	Optimal average wait time (preemptive)
Round Robin (RR)	Circular queue; each gets quantum Q (10–100 ms)	Yes	High context switch overhead if Q too small	Time-sharing; interactive; fair
Priority	Highest priority runs first; dynamic or static	Both	Starvation → fix with ageing (raise priority over time)	Real-time systems; mixed workloads

4.3 Round-Robin Worked Example (Q = 4 ms)

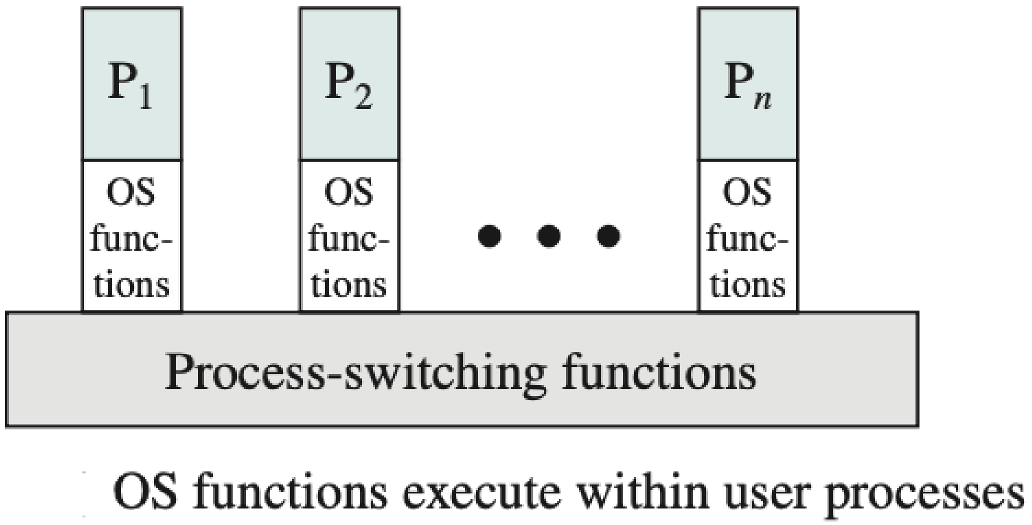

P1
OS func- tions
P2
OS func- tions
Pn
OS func- tions
Process-switching functions
OS functions execute within user processes

Figure: Round-Robin scheduling Gantt chart with quantum Q=4: P1(10ms), P2(6ms), P3(4ms) — all arrive at t=0 (Stallings Fig. 9.5)

★ Round-Robin Calculation

- Processes: P1(burst=10), P2(burst=6), P3(burst=4). Quantum Q=4. All arrive at t=0.
- Schedule: P1(0–4) → P2(4–8) → P3(8–12) → P1(12–16) → P2(16–18) → P1(18–20)
- Completion times: P3=12, P2=18, P1=20
- Turnaround: P3=12, P2=18, P1=20 → Average = (12+18+20)/3 = 16.7 ms
- Waiting time: P3=8, P2=12, P1=10 → Average = (8+12+10)/3 = 10 ms
- Smaller Q → more preemptions → shorter response time but more context switch overhead

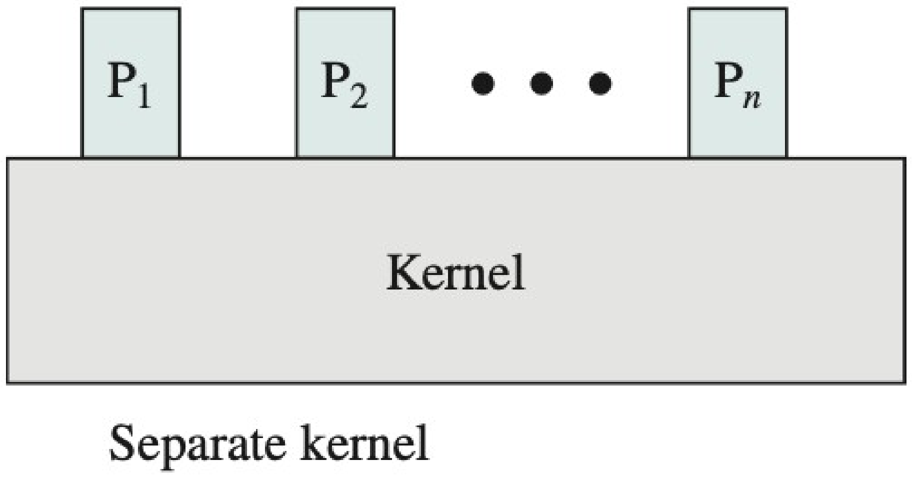


Figure: Comparison of FCFS, SJF, and RR scheduling for the same workload (Stallings Fig. 9.6)

Section 5 · OS Assignment — Stage 1

★ Milestone 1: Process Table + Round-Robin Scheduler

- process.h — typedef struct { int pid; int state; uint32_t esp; uint32_t eip; int priority; char name[32]; } pcb_t;
- States: PROC_READY=0, PROC_RUNNING=1, PROC_BLOCKED=2, PROC_ZOMBIE=3
- process.c — int create_process(char *name, void (*entry)(), int priority): allocate stack (4KB), init PCB, set eip=entry, state=READY, add to process_table[].
- scheduler.c — void schedule(): save context of current process into its PCB (esp←ESP); find next READY process (round-robin); restore context from next PCB (ESP←esp).
- switch.asm — context_switch(pcb_t *cur, pcb_t *next): PUSHAD; MOV [cur+esp_offset], ESP; MOV ESP, [next+esp_offset]; POPAD; RET
- irq.c — Program PIT (8253/8254) via ports 0x40–0x43 for ~100 Hz. IRQ0 handler calls schedule().
- Test: 'ps' shell command lists all PCBs with PID, name, state, priority.

Section 6 · Practice Exercises

✎ Exercise 9.1 [12 marks] — Five-State Model

- (a) [4] Draw the five-state process model. Label all states and all valid transitions with their triggering event. Include the Null → New and Ready → Exit transitions that are often omitted.
- (b) [4] A process P1 is currently in the Running state. For each event below, state the new state of P1 and which transition occurred: (i) P1 calls read() on a disk file (I/O takes 15 ms). (ii) The 50 ms time quantum expires. (iii) P1 calls pthread_mutex_lock() on a held mutex. (iv) The disk I/O from event (i) completes (interrupt received from disk controller).
- (c) [4] Explain why the Blocked → Ready transition is triggered by an interrupt handler (ISR), not

by the scheduler. Trace exactly what happens in the ISR when a disk I/O completes: (i) which data structure is checked? (ii) which process PCB is updated? (iii) to which queue is the process moved?

✎ Exercise 9.2 [14 marks] — CPU Scheduling

- (a) [6] Five processes arrive with these CPU burst times and arrival times: P1(8ms,t=0), P2(4ms,t=1), P3(9ms,t=2), P4(5ms,t=3), P5(2ms,t=4). Draw a Gantt chart and calculate average turnaround time and average waiting time for: (i) FCFS, (ii) SJF (non-preemptive), (iii) SRTF (preemptive SJF). Show all working.
- (b) [4] Using the same 5 processes, apply Round-Robin with Q=3 ms. Draw the Gantt chart. Calculate: (i) average turnaround time, (ii) average waiting time. (iii) How many context switches occur? Is RR better or worse than SJF for these processes?
- (c) [4] Priority scheduling with ageing: P1(priority=1, burst=10), P2(priority=5, burst=3), P3(priority=3, burst=7). P3 has been waiting for 20 ms; ageing adds 1 priority per 5 ms of wait. (i) What is P3's current effective priority? (ii) In which order do the processes execute? (iii) Without ageing, which process could starve? Explain.

✎ Exercise 9.3 [9 marks] — Context Switch

- (a) [4] Describe in detail what information is saved when the OS context-switches from Process A (currently running) to Process B (currently ready). For each item saved, explain: (i) WHERE it is saved, (ii) WHY it must be saved, (iii) WHEN it is restored.
- (b) [3] A context switch on an x86 processor takes 5 µs. Round-Robin scheduling uses Q=50 ms. What percentage of CPU time is spent on context switches? If Q is reduced to 5 ms, what percentage is now spent on context switches? Comment on the trade-off.
- (c) [2] Why does a context switch also require updating the CR3 register (page table base) on x86? What happens to the TLB when CR3 is loaded with a new value, and why does this add latency to context switching?