

OPERATING SYSTEM

Study Notes for Engineering Students

Chapter 4 : Scheduling

Unit	Topic
4.1	Scheduling: objectives, concept, criteria, CPU and I/O burst cycle
4.2	Types of Scheduling: pre-emptive and non-pre-emptive
4.3	Scheduling Algorithms: FCFS, SJF, Round Robin, Priority
4.4	Other Scheduling: multilevel, multiprocessor, real-time
4.5	Deadlock: system model, necessary conditions, mutual exclusion, critical region
4.6	Deadlock Handling: prevention and avoidance
	15 Multiple Choice Questions with Answer Key
	10 Broad Questions

www.polynoteshub.co.in

4.1 Scheduling

A modern computer runs many programs at once, but a CPU can execute only one process at a time. **CPU scheduling** is the activity by which the operating system decides which of the processes waiting in memory should be given the CPU next. It is the basis of **multiprogramming**: when one process has to wait (for example for an I/O operation) the operating system takes the CPU away from it and gives it to another process, so the CPU is never left idle. Scheduling is one of the most fundamental functions of an operating system, because almost every computer resource is scheduled before use, and the CPU is the most important of them.

4.1.1 Concept of Scheduling

The processes that are ready to run are kept in the **ready queue**. The part of the operating system that selects one process from the ready queue is called the **CPU scheduler** (short-term scheduler). The ready queue is not necessarily a FIFO queue: depending on the algorithm it may be a FIFO queue, a priority queue, a tree or simply an unordered list. The records in the queue are the **Process Control Blocks (PCBs)** of the waiting processes.

Basic terms used in scheduling

Term	Meaning
Process	A program in execution; the unit of work that is scheduled.
Arrival time (AT)	The time at which a process enters the ready queue.
Burst time (BT)	The total CPU time a process needs to complete its execution.
Completion time (CT)	The time at which the process finishes execution.
Turnaround time (TAT)	Total time from arrival to completion. $TAT = CT - AT$.
Waiting time (WT)	Total time spent waiting in the ready queue. $WT = TAT - BT$.
Response time (RT)	Time from arrival until the process gets the CPU for the first time.
Time quantum	The fixed slice of CPU time given to a process in Round Robin scheduling.

4.1.2 Objectives of Scheduling

A good scheduling policy tries to achieve the following goals at the same time:

- **Keep the CPU busy** as much of the time as possible (maximum utilization).
- **Fairness**: every process should get a fair share of the CPU and no process should starve.
- **Maximize throughput**: complete as many processes as possible per unit time.
- **Minimize turnaround, waiting and response time** so that users see good service.
- **Balance resource use**: keep I/O devices as well as the CPU busy.
- **Enforce priorities** so that important (for example system or real-time) processes get the CPU first.
- **Predictability and low overhead**: behave consistently and spend little time in the scheduler itself.

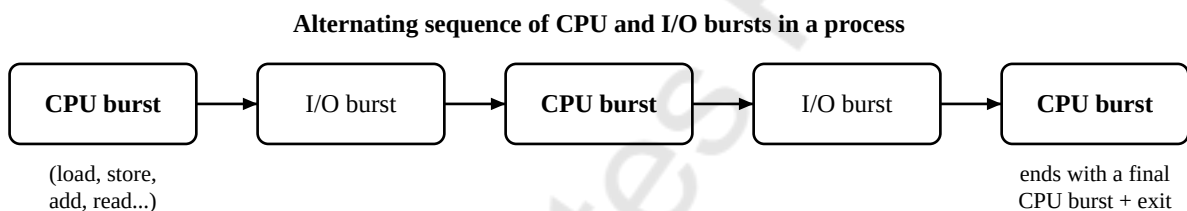
4.1.3 Scheduling Criteria

Different algorithms favour different classes of processes, so criteria are needed to compare them. The commonly used criteria are:

Criterion	Meaning	Aim
CPU utilization	Percentage of time the CPU is busy. In a real system it ranges from about 40% (lightly loaded) to 90% (heavily loaded).	Maximize
Throughput	Number of processes completed per unit time.	Maximize
Turnaround time	Interval from submission of a process to its completion (includes waiting in memory, ready queue, CPU and I/O).	Minimize
Waiting time	Sum of the periods spent waiting in the ready queue. The scheduling algorithm affects only this time, not the burst time.	Minimize
Response time	Time from submission of a request until the first response is produced (important in interactive systems).	Minimize

4.1.4 CPU and I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: **process execution consists of a cycle of CPU execution and I/O wait**. A process begins with a **CPU burst**, which is followed by an **I/O burst**, then another CPU burst, then another I/O burst and so on. The last CPU burst ends with a system request to terminate the process.



Execution starts with a CPU burst, alternates, and finishes with a CPU burst that requests termination.

Fig. 4.1 : CPU and I/O burst cycle

When the frequency of CPU burst durations is plotted, the curve is typically exponential or hyper-exponential: there are a large number of **short CPU bursts** and a small number of **long CPU bursts**. This observation helps in choosing a scheduling algorithm.

Feature	I/O-bound process	CPU-bound process
CPU bursts	Many, very short	Few, very long
I/O activity	Heavy, spends most time waiting for I/O	Little, spends most time computing
Example	Text editor, database query, web browser	Scientific simulation, video encoding
Scheduling need	Should get the CPU quickly so that I/O devices stay busy	Can be given longer slices; should not block short jobs

4.1.5 Process States

During its life a process moves between several states. **New**: the process is being created. **Ready**: it is waiting in the ready queue for the CPU. **Running**: its instructions are being executed. **Waiting** (blocked): it is waiting for an I/O completion or an event. **Terminated**: it has finished execution. Scheduling decisions are taken at the transitions shown in Fig. 4.2.

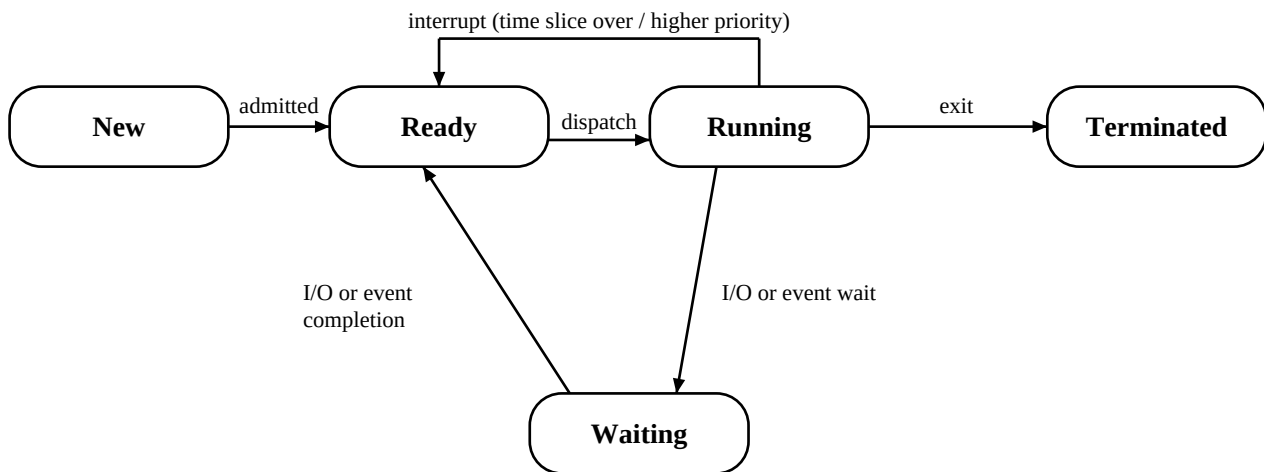


Fig. 4.2 : Process state diagram

4.1.6 Types of Schedulers and the Dispatcher

Scheduler	Also called	Function	Frequency
Long-term	Job scheduler	Selects processes from the job pool on disk and loads them into memory. Controls the degree of multiprogramming and the mix of I/O-bound and CPU-bound jobs.	Very low (seconds to minutes)
Short-term	CPU scheduler	Selects one process from the ready queue and allocates the CPU to it.	Very high (every few milliseconds), so it must be fast
Medium-term	Swapper	Swaps processes out of memory and later back in to reduce the degree of multiprogramming.	Intermediate

The **dispatcher** is the module that gives control of the CPU to the process selected by the short-term scheduler. Its work includes (i) **switching context**, (ii) **switching to user mode**, and (iii) **jumping to the proper location** in the user program to restart it. The time the dispatcher takes to stop one process and start another is called **dispatch latency**; it should be as small as possible.

4.2 Types of Scheduling

CPU-scheduling decisions may take place under four circumstances:

- 1 When a process switches from the **running** state to the **waiting** state (for example an I/O request).
- 2 When a process switches from the **running** state to the **ready** state (for example when an interrupt occurs).
- 3 When a process switches from the **waiting** state to the **ready** state (for example at completion of I/O).
- 4 When a process **terminates**.

For circumstances 1 and 4 there is no choice: a new process must be selected. When scheduling takes place **only** under 1 and 4 the scheme is **non-pre-emptive**. When it also takes place under 2 and 3 it is **pre-emptive**.

4.2.1 Non-Pre-emptive Scheduling

Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by **terminating** or by **switching to the waiting state**. The operating system never forcibly removes the running process. This scheme was used by Windows 3.x and is simple to implement because it needs no special timer hardware. Examples: FCFS, non-pre-emptive SJF and non-pre-emptive priority scheduling.

4.2.2 Pre-emptive Scheduling

The running process can be **interrupted and moved back to the ready queue** if a higher-priority process arrives or its time slice expires. The CPU is taken away from the process even though it has not finished its burst. Windows 95 onwards and all versions of Mac OS X and UNIX use pre-emptive scheduling. Examples: Round Robin, Shortest Remaining Time First (pre-emptive SJF) and pre-emptive priority scheduling.

4.2.3 Comparison

Basis	Non-pre-emptive	Pre-emptive
CPU release	Process keeps the CPU until it terminates or blocks	CPU can be taken away before the burst is complete
Response to urgent jobs	Poor; a short or urgent job may wait behind a long job	Good; urgent jobs get the CPU quickly
Overhead	Low, few context switches	Higher, more context switches
Hardware need	No timer needed	Timer interrupt required
Shared-data problem	Rare, as the process is never interrupted mid-way	Possible race conditions; needs synchronization
Starvation	Long jobs can delay all others (convoy effect)	Low-priority processes may starve
Examples	FCFS, SJF, Priority (non-pre-emptive)	RR, SRTF, Priority (pre-emptive)

Issues with pre-emption. Pre-emption can cause **race conditions** if two processes share data and one is pre-empted while updating it. Pre-emption also affects the **operating-system kernel**: if the kernel is pre-empted while modifying its own data structures (such as I/O queues) the structures may be left inconsistent. Hence, either the kernel waits for a system call to complete or an I/O block to occur before a context switch (non-pre-emptive kernel), or it protects critical data with locks and interrupt disabling (pre-emptive kernel).

4.3 Scheduling Algorithms

A scheduling algorithm decides which process in the ready queue is allocated the CPU. In the examples below the processes are shown with their burst times in milliseconds (ms), and a **Gantt chart** (a bar chart showing the schedule with start and end times) is used to represent the result.

4.3.1 First Come First Served (FCFS)

FCFS is the simplest algorithm. The process that **requests the CPU first is allocated the CPU first**. It is implemented with a FIFO queue: a new process is added at the tail, and the CPU is given to the process at the head. It is **non-pre-emptive**.

Example 1. Three processes arrive at time 0 in the order P1, P2, P3.

Process	Burst time (ms)
P1	24
P2	3
P3	3

Gantt chart : FCFS (order P1, P2, P3)



Waiting times: P1 = 0, P2 = 24, P3 = 27. Average waiting time = $(0 + 24 + 27) / 3 = 17$ ms. If the processes had arrived in the order P2, P3, P1, the Gantt chart would be P2 (0-3), P3 (3-6), P1 (6-30) and the average waiting time would drop to $(6 + 0 + 3) / 3 = 3$ ms. The average waiting time under FCFS is therefore generally not minimal and varies greatly with the arrival order.

Advantages and disadvantages

Advantages	Disadvantages
Very simple to understand and implement	Average waiting time is often long
Fair in the sense that requests are served in order of arrival	Convoy effect: short processes wait behind one long CPU-bound process
No starvation, every process eventually runs	Non-pre-emptive; not suitable for time-sharing systems

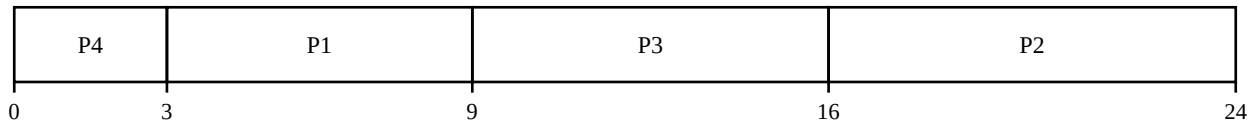
4.3.2 Shortest Job First (SJF)

SJF associates with each process the length of its **next CPU burst**. When the CPU is free it is assigned to the process with the **smallest next burst**; ties are broken by FCFS. SJF is **provably optimal**: it gives the **minimum average waiting time** for a given set of processes, because moving a short process before a long one reduces the waiting time of the short process more than it increases that of the long one.

Example 2 (non-pre-emptive SJF). All processes arrive at time 0.

Process	Burst time (ms)
P1	6
P2	8
P3	7
P4	3

Gantt chart : non-pre-emptive SJF



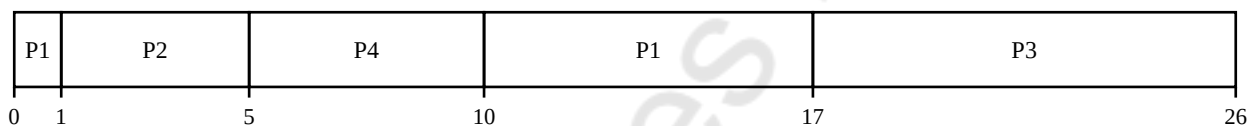
Waiting times: P4 = 0, P1 = 3, P3 = 9, P2 = 16. Average waiting time = $(0 + 3 + 9 + 16) / 4 = 7$ ms. (FCFS on the same data would give 10.25 ms.)

Pre-emptive SJF : Shortest Remaining Time First (SRTF)

If a new process arrives with a CPU burst shorter than the **remaining** time of the running process, the running process is pre-empted. Consider the following processes.

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart : SRTF



Process	Completion	Turnaround (CT - AT)	Waiting (TAT - BT)
P1	17	17	9
P2	5	4	0
P3	26	24	15
P4	10	7	2

Average waiting time = $(9 + 0 + 15 + 2) / 4 = 6.5$ ms. Average turnaround time = $(17 + 4 + 24 + 7) / 4 = 13$ ms.

Predicting the next CPU burst

The real difficulty with SJF is that the length of the next CPU burst is **not known in advance**. It can only be **estimated** from the lengths of previous bursts using **exponential averaging**: $\tau_{n+1} = \alpha \times t_n + (1 - \alpha) \times \tau_n$, where t_n is the actual length of the n-th burst, τ_n is the previous prediction (and τ_{n+1} the new prediction) and α ($0 \leq \alpha \leq 1$) controls the weight of recent history. With $\alpha = 0$ recent history has no effect; with $\alpha = 1$ only the last burst counts; a common choice is $\alpha = 1/2$.

Because of this, SJF cannot be implemented exactly at the level of short-term CPU scheduling; it is mainly used in long-term (job) scheduling where users can state a time limit. A major drawback of SJF is **starvation** of long jobs when short jobs keep arriving.

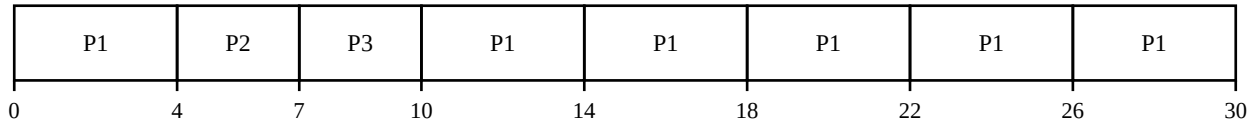
4.3.3 Round Robin (RR)

Round Robin is designed specially for **time-sharing systems**. It is similar to FCFS but with **pre-emption**. A small unit of time called the **time quantum** (or time slice, usually 10 to 100 ms) is defined. The ready queue is treated as a **circular queue**; the scheduler goes around it and allocates the CPU to each process for up to one time quantum. If a process finishes in less than one quantum it releases the CPU voluntarily; otherwise the timer interrupts it and it is put at the tail of the ready queue.

Example 3. Time quantum = 4 ms, all processes arrive at time 0.

Process	Burst time (ms)
P1	24
P2	3
P3	3

Gantt chart : Round Robin (q = 4)



Completion times: P1 = 30, P2 = 7, P3 = 10. Waiting times = TAT - BT: P1 = 30 - 24 = 6, P2 = 7 - 3 = 4, P3 = 10 - 3 = 7. Average waiting time = $17 / 3 = 5.66$ ms. Note how the short processes P2 and P3 now finish quickly, which is not the case with FCFS (average 17 ms).

Effect of the time quantum

Time quantum	Behaviour
Very large (larger than the longest burst)	RR degenerates into FCFS.
Very small	Many context switches; the overhead becomes large and CPU efficiency falls. The system behaves like processor sharing.
Properly chosen	About 80% of CPU bursts should be shorter than the quantum. The quantum should be large compared with the context-switch time (which is usually below 10 microseconds).

With n processes in the ready queue and quantum q , no process waits more than $(n - 1) \times q$ time units before getting its next turn. RR gives good **response time** but generally a higher average turnaround time than SJF.

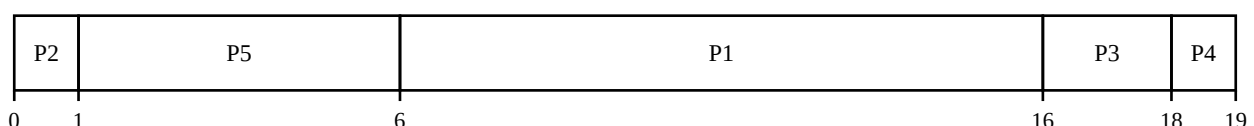
4.3.4 Priority Scheduling

A **priority** (an integer) is associated with every process and the CPU is allocated to the process with the **highest priority**. Equal-priority processes are scheduled in FCFS order. In these notes a **smaller number means higher priority** (some systems use the opposite convention). SJF is itself a priority algorithm where the priority is the inverse of the predicted next CPU burst. Priority scheduling can be **pre-emptive** or **non-pre-emptive**. Priorities may be **internal** (set by the OS using time limits, memory needs, number of open files, I/O-to-CPU burst ratio) or **external** (set by importance of the user, department or payment).

Example 4 (non-pre-emptive priority). All processes arrive at time 0.

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

Gantt chart : Priority scheduling



Waiting times: $P_2 = 0$, $P_5 = 1$, $P_1 = 6$, $P_3 = 16$, $P_4 = 18$. Average waiting time = $(0 + 1 + 6 + 16 + 18) / 5 = 8.2$ ms.

Problem and solution: starvation and aging

The major problem is **indefinite blocking or starvation**: a low-priority process may wait forever while higher-priority processes keep arriving. (A well-known rumour says that when the IBM 7094 at MIT was shut down in 1973 a low-priority job submitted in 1967 had still not run.) The solution is **aging**: the priority of a process that has waited a long time is **gradually increased** (for example by 1 every 15 minutes), so that it eventually reaches the highest priority and runs.

4.3.5 Comparison of the Algorithms

Algorithm	Type	Advantage	Disadvantage	Starvation?
FCFS	Non-pre-emptive	Simple, fair by arrival order	Convoy effect, long waiting	No
SJF	Both (SRTF is pre-emptive)	Minimum average waiting time	Burst length must be predicted	Yes
Round Robin	Pre-emptive	Good response time, fair share	Quantum choice critical; overhead	No
Priority	Both	Supports important jobs	Low-priority jobs may starve	Yes (fixed by aging)

4.4 Other Scheduling

4.4.1 Multilevel Queue Scheduling

When processes can be easily classified into groups with different needs (for example **foreground/interactive** and **background/batch** processes), the ready queue is **partitioned into several separate queues**. Each process is **permanently assigned** to one queue according to its type, priority or memory size. Each queue has **its own scheduling algorithm**: for example the foreground queue may use RR while the background queue uses FCFS.

Scheduling is also required **between the queues**, which is commonly done in one of two ways:

- **Fixed-priority pre-emptive scheduling**: a lower queue runs only when all higher queues are empty. A newly arriving process in a higher queue pre-empts the running process of a lower queue. Lower queues may starve.
- **Time slicing among queues**: each queue gets a fixed share of CPU time, for example 80% to the foreground queue (RR) and 20% to the background queue (FCFS).

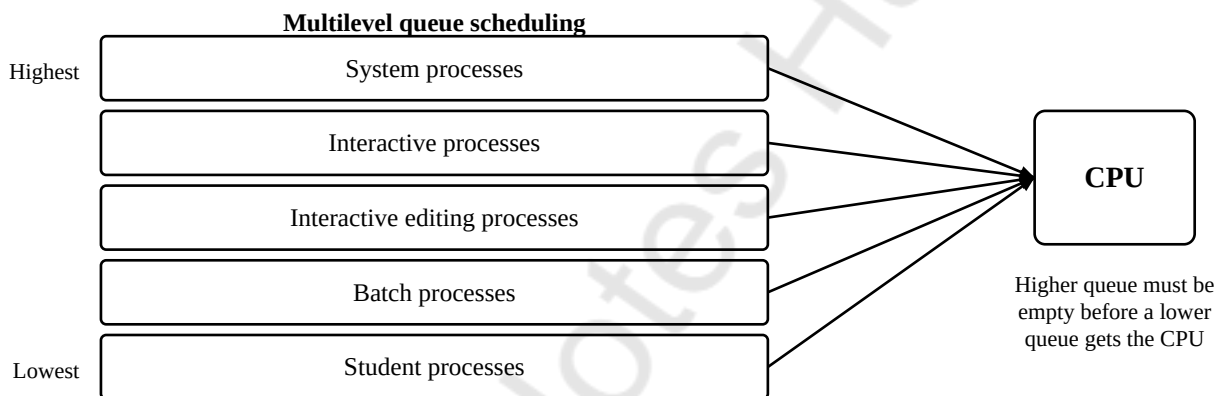


Fig. 4.3 : Multilevel queue scheduling

4.4.2 Multilevel Feedback Queue Scheduling

In a multilevel queue a process can never change queue, which is inflexible. The **multilevel feedback queue** allows a process to **move between queues**. The idea is to separate processes according to the characteristics of their CPU bursts: a process that uses too much CPU time is **moved to a lower-priority queue**, while an I/O-bound or interactive process stays in the higher-priority queues. A process that waits too long in a low-priority queue may be moved to a higher queue (**aging**), which prevents starvation.

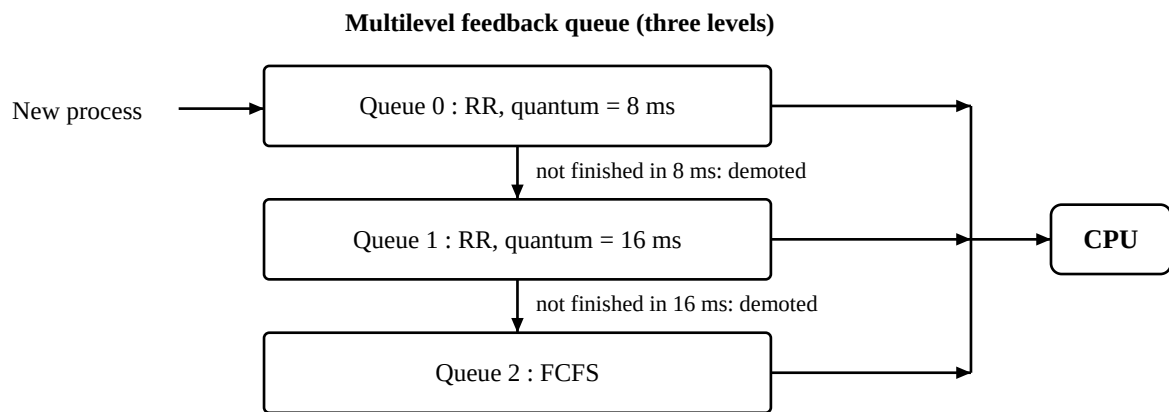


Fig. 4.4 : Multilevel feedback queue with three queues

In the three-queue example of Fig. 4.4 a new process enters Queue 0 and gets 8 ms. If it does not finish it moves to Queue 1 and gets 16 ms more; if it still does not finish it is placed in Queue 2 and served FCFS. Queue 1 runs only when Queue 0 is empty and Queue 2 only when Queues 0 and 1 are empty. The scheduler is defined by these parameters:

- number of queues;
- scheduling algorithm for each queue;
- method used to decide when to **upgrade** a process to a higher queue;
- method used to decide when to **demote** a process to a lower queue;
- method used to decide which queue a new process enters.

It is the most general scheme and can be configured for a specific system, but it is also the **most complex** algorithm to implement because choosing the best parameters is difficult.

4.4.3 Multiple-Processor Scheduling

When more than one CPU is available, **load sharing** becomes possible but scheduling becomes more complex. We consider systems in which the processors are **homogeneous** (identical in functionality), so any process can run on any processor.

Approaches

Approach	Description	Remark
Asymmetric multiprocessing (AMP)	One processor, the master server , handles all scheduling decisions, I/O processing and other system activity. The other processors execute only user code.	Simple, since only one processor accesses system data structures; the master may become a bottleneck.
Symmetric multiprocessing (SMP)	Each processor is self-scheduling. All processes may be in a common ready queue or each processor may have its own private queue.	Used by Windows, Linux and macOS. Needs care so that two processors do not pick the same process.

Issues in SMP scheduling

- **Processor affinity:** a process prefers to keep running on the same processor because its data is already in that processor's cache. *Soft affinity* means the OS tries but does not guarantee this; *hard affinity* means the process may specify that it must not migrate.
- **Load balancing:** keeps the workload evenly distributed so that no processor is idle while others are overloaded. *Push migration:* a task checks the load and pushes tasks from overloaded to idle processors. *Pull migration:* an idle processor pulls a waiting task from a busy processor. Load balancing often works against processor affinity.

- **Multicore processors:** several cores on one chip; a core may stall on a memory access, so multithreaded cores switch to another hardware thread during the stall.

4.4.4 Real-Time Scheduling

A **real-time system** is one in which the correctness of a result depends not only on its logical correctness but also on the **time at which it is produced**. Examples are air-traffic control, flight and medical systems, industrial robots and multimedia. Real-time systems are of two kinds:

Feature	Hard real-time	Soft real-time
Deadline	Must be met strictly; a missed deadline is a system failure	Critical tasks get priority, but a missed deadline is tolerable
Guarantee	Task must be serviced within its guaranteed time	Only favoured over non-critical tasks
Consequence of failure	May be disastrous	Degraded quality of service
Examples	Pacemaker, anti-lock brakes, nuclear plant control	Video streaming, online gaming, multimedia

Requirements and features

- **Pre-emptive, priority-based scheduling** is essential: the highest-priority real-time task must get the CPU at once. Soft real-time systems (e.g. Windows, Linux) give real-time tasks the highest priority but do not guarantee deadlines.
- **Low interrupt latency:** the time from the arrival of an interrupt to the start of its service routine must be minimal.
- **Low dispatch latency:** the time the dispatcher takes to stop one process and start another. Needs a pre-emptive kernel and quick release of resources held by low-priority processes.
- Real-time tasks are often **periodic**: a task needs the CPU at constant intervals (period p), with processing time t and deadline d , where $0 \leq t \leq d \leq p$.

Real-time scheduling algorithms

Algorithm	Idea	Remark
Rate Monotonic (RM)	Static priority: the task with the shorter period gets the higher priority ; pre-emptive.	Optimal among static-priority algorithms. CPU utilization is bounded: for N tasks it guarantees deadlines if utilization is below $N(2^{1/N} - 1)$; about 69% for large N .
Earliest Deadline First (EDF)	Dynamic priority: the task with the earliest deadline gets the highest priority at that moment.	Theoretically optimal; can reach 100% CPU utilization, but priorities must be recalculated when new tasks arrive.

4.5 Deadlock

In a multiprogramming environment several processes may compete for a limited number of resources. A process requests resources and, if they are not available at that time, it enters the waiting state. Sometimes a waiting process can never again change state because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. A simple example is a narrow bridge: two cars meet in the middle and neither can move forward unless the other backs up. Another example is two processes each holding one tape drive and each waiting for the tape drive held by the other.

Definition. A set of processes is in a **deadlock** state when every process in the set is waiting for an event (such as the release of a resource) that can be caused only by another process in the same set.

4.5.1 System Model

A system consists of a finite number of **resources** distributed among competing processes. The resources are partitioned into **types** (or classes), each having a number of identical **instances**. Examples: CPU cycles, memory space, files, printers, tape drives and I/O devices. If a process requests an instance of a type, allocating *any* instance of that type satisfies the request. Resources are of two kinds: **physical** (printers, drives, memory) and **logical** (files, semaphores, monitors).

Under the normal mode of operation a process uses a resource in the following sequence only:

- 1 **Request:** the process requests the resource. If it cannot be granted immediately the process must wait.
- 2 **Use:** the process operates on the resource (for example prints on the printer).
- 3 **Release:** the process releases the resource.

Requests and releases are made through **system calls** such as request()/release() for devices, open()/close() for files and allocate()/free() for memory, or through the wait() and signal() operations of semaphores. The operating system keeps a table recording whether each resource is free or allocated and, if allocated, to which process; it also maintains a queue of processes waiting for each resource.

4.5.2 Necessary Conditions for Deadlock

A deadlock can arise **only if all four** of the following conditions hold simultaneously in a system (Coffman conditions). If any one of them is absent, a deadlock cannot occur.

Condition	Explanation
1. Mutual exclusion	At least one resource must be held in a non-sharable mode: only one process at a time can use it. Another process requesting it must wait until it is released.
2. Hold and wait	A process must be holding at least one resource and waiting to acquire additional resources that are currently held by other processes.
3. No pre-emption	Resources cannot be forcibly taken away; a resource can be released only voluntarily by the process holding it after it has finished its task.
4. Circular wait	A set of waiting processes {P ₀ , P ₁ , ..., P _n } must exist such that P ₀ waits for a resource held by P ₁ , P ₁ for one held by P ₂ , ..., P _{n-1} for one held by P _n , and P _n waits for one held by P ₀ .

The conditions are not completely independent: the circular-wait condition implies the hold-and-wait condition. They are, however, treated separately because each suggests a different way of preventing deadlock (Section 4.6).

4.5.3 Resource Allocation Graph (RAG)

Deadlocks can be described precisely with a directed graph called the **system resource-allocation graph**. The vertices are divided into the set of **processes** $P = \{P_1, P_2, \dots, P_n\}$ (drawn as circles) and the set of **resource types** $R = \{R_1, R_2, \dots, R_m\}$ (drawn as rectangles, with a dot for each instance). There are two kinds of edges:

- **Request edge** $P_i \rightarrow R_j$: process P_i has requested an instance of R_j and is waiting for it.
- **Assignment edge** $R_j \rightarrow P_i$: an instance of R_j has been allocated to P_i .

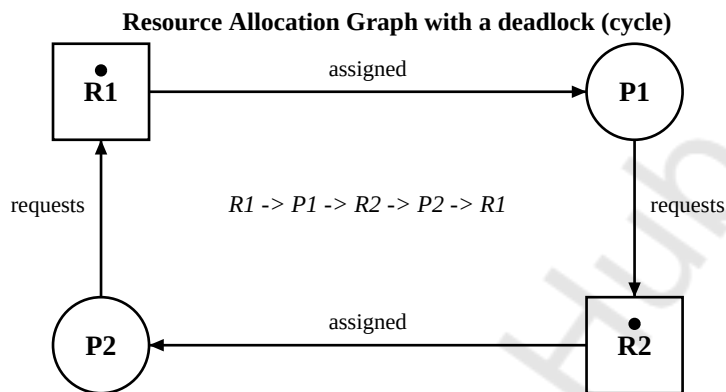


Fig. 4.5 : Resource allocation graph containing a cycle (deadlock)

Situation in the graph	Conclusion
No cycle in the graph	No deadlock exists in the system.
Cycle and one instance per resource type	A cycle means a deadlock (necessary and sufficient).
Cycle and several instances per resource type	A cycle is a necessary but not sufficient condition; a deadlock may or may not exist (a process outside the cycle may release an instance and break it).

4.5.4 Mutual Exclusion and Critical Region

Mutual exclusion means that when one process is using a shared resource or shared data, no other process is allowed to use it at the same time. It is required for non-sharable resources (a printer, a tape drive, a write-locked file) and is the first of the four deadlock conditions. Sharable resources, such as a file opened read-only, do not need mutually exclusive access and never take part in a deadlock.

A **critical region** (also called the **critical section**) is the segment of code of a process in which it accesses **shared variables, files or other shared resources**. The critical-section problem is to design a protocol so that **when one process is executing in its critical section, no other process is allowed to execute in its critical section**. The general structure of a process is:

```
do {
    [ entry section ]    /* request permission to enter */
        CRITICAL SECTION
    [ exit section ]     /* release permission */
        REMAINDER SECTION
} while (true);
```

A solution to the critical-section problem must satisfy three requirements:

- **Mutual exclusion:** if a process is executing in its critical section, no other process can be executing in its critical section.
- **Progress:** if no process is in its critical section and some wish to enter, only processes not in their remainder sections may decide who enters next, and this decision cannot be postponed indefinitely.
- **Bounded waiting:** there is a limit on the number of times other processes may enter their critical sections after a process has requested entry and before that request is granted.

Mutual exclusion is enforced with tools such as **semaphores**, **mutex locks**, monitors and Peterson's solution. A **semaphore** S is an integer variable accessed only through the atomic operations **wait(S)** and **signal(S)**. If the semaphores are used carelessly, a deadlock results. Consider two semaphores P and Q, both initialized to 1:

Process P0	Process P1
wait(P); wait(Q); ... critical section ... signal(P); signal(Q);	wait(Q); wait(P); ... critical section ... signal(Q); signal(P);

If P0 executes wait(P) and then P1 executes wait(Q), P0 will wait for Q held by P1 and P1 will wait for P held by P0. Neither can continue: a deadlock has occurred. **Starvation** is a related problem (indefinite blocking), but unlike deadlock a starving process is not part of a circular wait and may eventually run.

Basis	Deadlock	Starvation
Meaning	Set of processes blocked forever, each waiting for another in the set	A process waits indefinitely because others are always preferred
Resources	Held by the blocked processes themselves	Held by or given to other, running processes
Needed conditions	All four Coffman conditions	No such conditions; due to unfair policy
Solution	Prevention, avoidance, detection and recovery	Aging, fair scheduling

4.6 Deadlock Handling

There are three general ways to deal with the deadlock problem:

- **Use a protocol to prevent or avoid deadlock**, ensuring that the system never enters a deadlocked state (deadlock prevention or deadlock avoidance).
- **Allow the system to enter a deadlocked state**, detect it and recover (deadlock detection and recovery).
- **Ignore the problem** and pretend that deadlocks never occur. This is the **ostrich algorithm**, used by most operating systems including UNIX and Windows, because deadlocks are rare and prevention is costly; the application programmer must deal with them.

Prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. **Avoidance** requires that the operating system is given in advance additional information about the resources a process will request and use during its life, and with this it decides for each request whether to grant or delay it.

4.6.1 Deadlock Prevention

By making sure that **at least one of the four necessary conditions cannot hold**, deadlock can be prevented.

Condition attacked	Method	Drawbacks
Mutual exclusion	Make resources sharable wherever possible (e.g. read-only files). Use spooling so that only the spooler daemon uses the printer.	Some resources are intrinsically non-sharable (a printer cannot be shared by two jobs at once), so this condition cannot generally be denied.
Hold and wait	(a) Require a process to request and be allocated all its resources before it begins execution , or (b) allow a process to request resources only when it holds none (it must release all current resources first).	Low resource utilization (resources are held but unused for long periods); starvation is possible for a process needing popular resources; the full requirement may not be known in advance.
No pre-emption	If a process holding some resources requests another that cannot be allocated, all resources it currently holds are pre-empted (released) and it restarts when it regains the old and new resources together. Or, take resources from a waiting process.	Applicable only to resources whose state can be saved and restored easily (CPU registers, memory); not to printers or tape drives.
Circular wait	Impose a total ordering of all resource types, e.g. tape drive = 1, disk = 5, printer = 12. Each process must request resources in increasing order of number only.	Practically the most useful method. Drawback: the ordering may not suit all programs and may reduce resource utilization.

Why circular wait is prevented by ordering. Suppose a circular wait exists among $P_0, \dots, P_n$ with P_i holding R_i and requesting $R_{(i+1)}$. Since each process requests only higher-numbered resources than it holds, $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$, which is impossible. Hence no circular wait can occur.

4.6.2 Deadlock Avoidance

Prevention methods are simple but may lead to **low device utilization and reduced throughput**. Avoidance is an alternative: it needs **a priori information** about how resources will be used. The simplest and most useful model requires that each process **declares the maximum number of resources of each type that it may need**. The deadlock-avoidance algorithm then dynamically examines the **resource-allocation state** (the number of available and allocated resources and the maximum demands of the processes) to ensure that a circular-wait condition can never arise.

Safe and unsafe states

A state is **safe** if the system can allocate resources to each process (up to its maximum) in **some order** and still avoid a deadlock. Formally, a state is safe if there exists a **safe sequence** of all the processes $\langle P_1, P_2, \dots, P_n \rangle$ such that for each P_i the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all P_j with $j < i$. If no such sequence exists the state is **unsafe**.

- A **safe state is not a deadlocked state**.
- A **deadlocked state is an unsafe state**.
- An **unsafe state may lead to a deadlock**, but not necessarily; it only means that the OS can no longer guarantee that deadlock will be avoided.
- Avoidance therefore means: **never let the system move from a safe state to an unsafe state**.

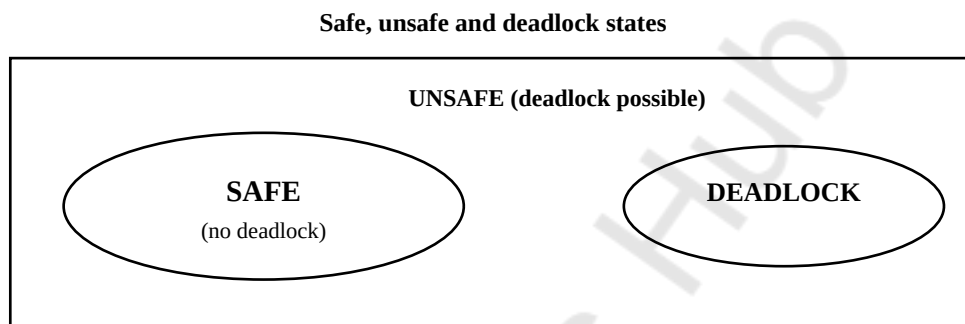


Fig. 4.6 : Relation between safe, unsafe and deadlock states

A. Resource-allocation-graph algorithm (single instance per resource)

A third kind of edge is added to the RAG: the **claim edge** $P_i \rightarrow R_j$ (drawn as a dashed line), which shows that P_i **may request** R_j in the future. When P_i actually requests R_j the claim edge is converted into a request edge; when the resource is allocated the request edge becomes an assignment edge; after release it becomes a claim edge again. All claim edges must be declared before the process starts. A request is granted **only if converting the request edge into an assignment edge does not form a cycle** in the graph (cycle detection takes $O(n^2)$ time). If a cycle would form, the allocation would leave the system in an unsafe state and the process must wait.

B. Banker's Algorithm (several instances per resource type)

The resource-allocation-graph method does not work when a resource type has multiple instances. For this case the **Banker's algorithm** (Dijkstra) is used. The name comes from banking: a bank never allocates its cash in such a way that it can no longer satisfy the needs of all its customers. When a new process enters the system it must declare its maximum need, which may not exceed the total resources. When a process requests resources the system checks whether granting them leaves the system in a safe state; if so they are granted, otherwise the process must wait.

Data structures (n = number of processes, m = number of resource types)

Structure	Size	Meaning
Available	vector of length m	$Available[j] = k$ means k instances of resource type R_j are free.
Max	$n \times m$ matrix	$Max[i][j]$ = maximum demand of process P_i for resource R_j .
Allocation	$n \times m$ matrix	$Allocation[i][j]$ = number of instances of R_j currently allocated to P_i .
Need	$n \times m$ matrix	$Need[i][j] = Max[i][j] - Allocation[i][j]$, the remaining requirement.

Safety algorithm

- 1 Let **Work** = Available and **Finish[i] = false** for all i.
- 2 Find a process i such that **Finish[i] = false** and **Need_i <= Work**. If none exists, go to step 4.
- 3 Set **Work = Work + Allocation_i** and **Finish[i] = true**; go to step 2.
- 4 If **Finish[i] = true for all i**, the system is in a safe state; otherwise it is unsafe.

Resource-request algorithm (process P_i requests Request_i)

- 1 If Request_i <= Need_i go to step 2; otherwise raise an error, as the process has exceeded its declared maximum.
- 2 If Request_i <= Available go to step 3; otherwise P_i must **wait**, since resources are not available.
- 3 Pretend to allocate: Available = Available - Request_i; Allocation_i = Allocation_i + Request_i; Need_i = Need_i - Request_i. If the resulting state is **safe** the allocation is made permanent; if **unsafe** P_i must wait and the old state is restored.

Worked example

A system has five processes P0 to P4 and three resource types A (10 instances), B (5 instances) and C (7 instances). The snapshot at time T0 is:

Process	Allocation A B C	Max A B C	Available A B C	Need = Max - Allocation A B C
P0	0 1 0	7 5 3	3 3 2	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

Is the system in a safe state? Work = (3, 3, 2). The trace of the safety algorithm is shown below.

Step	Process chosen	Condition checked (Need <= Work)	New Work = Work + Allocation
1	P1	(1,2,2) <= (3,3,2) true	(3,3,2) + (2,0,0) = (5,3,2)
2	P3	(0,1,1) <= (5,3,2) true	(5,3,2) + (2,1,1) = (7,4,3)
3	P4	(4,3,1) <= (7,4,3) true	(7,4,3) + (0,0,2) = (7,4,5)
4	P2	(6,0,0) <= (7,4,5) true	(7,4,5) + (3,0,2) = (10,4,7)
5	P0	(7,4,3) <= (10,4,7) true	(10,4,7) + (0,1,0) = (10,5,7)

All processes can finish, so the system is in a **safe state** with the **safe sequence** <P1, P3, P4, P2, P0>. (Other safe sequences may exist.)

Can the request (1, 0, 2) by P1 be granted? Check: Request <= Need (1,0,2 <= 1,2,2) true; Request <= Available (1,0,2 <= 3,3,2) true. Pretend to allocate: Available = (2,3,0); Allocation of P1 = (3,0,2); Need of P1 = (0,2,0). Running the safety algorithm on the new state gives the safe sequence <P1, P3, P4, P0, P2>, so the request **can be granted immediately**. By contrast, in that new state a request (3,3,0) by P4 cannot be granted because the resources are not available (Request > Available), and a request (0,2,0) by P0 cannot be granted even though resources are available, because the resulting state would be unsafe.

4.6.3 Prevention versus Avoidance

Basis	Deadlock prevention	Deadlock avoidance
Approach	Ensures at least one necessary condition can never hold	Dynamically checks each request so the system stays in a safe state
Information needed	None in advance	Maximum resource needs of every process in advance
Resource utilization	Low	Better than prevention
Cost / overhead	Restrictions are applied at design time	Run-time overhead of checking safety on every request
Examples	Ordering of resources, request-all-at-start	Banker's algorithm, RAG with claim edges

Quick Revision Points

- Scheduling is the basis of multiprogramming; the short-term scheduler picks from the ready queue and the dispatcher hands over the CPU.
- Non-pre-emptive: CPU released only on termination or waiting. Pre-emptive: CPU can be taken away.
- FCFS: simple, convoy effect. SJF: optimal average waiting time, but burst unknown, starvation. RR: time-sharing, quantum selection is crucial. Priority: starvation, solved by aging.
- Multilevel queue: fixed queues. Multilevel feedback queue: processes move between queues. SMP: self-scheduling processors, affinity and load balancing. RM and EDF are real-time algorithms.
- Deadlock needs all four: mutual exclusion, hold and wait, no pre-emption, circular wait.
- Cycle in a RAG: deadlock for single-instance resources; only a possibility for multiple instances.
- Prevention denies one of the four conditions; avoidance (Banker's algorithm) keeps the system in a safe state.

Multiple Choice Questions

Choose the correct option for each question. The answer key is given at the end of this section.

Q1. Which of the following schedulers is also known as the CPU scheduler?

- (a) Long-term scheduler
- (b) Short-term scheduler
- (c) Medium-term scheduler
- (d) Job scheduler

Q2. The module that gives control of the CPU to the process selected by the short-term scheduler is called the:

- (a) Loader
- (b) Swapper
- (c) Dispatcher
- (d) Compiler

Q3. Which scheduling algorithm suffers from the convoy effect?

- (a) Round Robin
- (b) FCFS
- (c) Multilevel feedback queue
- (d) Earliest deadline first

Q4. Which algorithm gives the minimum average waiting time for a given set of processes?

- (a) FCFS
- (b) Round Robin
- (c) SJF
- (d) Priority scheduling

Q5. If the time quantum in Round Robin scheduling is made very large, the algorithm becomes the same as:

- (a) SJF
- (b) FCFS
- (c) Priority scheduling
- (d) Multilevel queue

Q6. Starvation of low-priority processes in priority scheduling is solved by:

- (a) Aging
- (b) Pre-emption
- (c) Spooling
- (d) Swapping

Q7. Three processes P1, P2, P3 with burst times 24, 3 and 3 ms arrive at time 0 in that order. The average waiting time under FCFS is:

- (a) 3 ms
- (b) 10 ms
- (c) 17 ms
- (d) 27 ms

Q8. Four processes with burst times 6, 8, 7 and 3 ms arrive together. The average waiting time under non-pre-emptive SJF is:

- (a) 5 ms
- (b) 7 ms
- (c) 10.25 ms
- (d) 16 ms

Q9. In a multilevel feedback queue, a process that uses too much CPU time is:

- (a) Terminated
- (b) Moved to a lower-priority queue
- (c) Moved to a higher-priority queue
- (d) Kept in the same queue forever

Q10. In symmetric multiprocessing (SMP):

- (a) Only the master processor schedules
- (b) Each processor is self-scheduling
- (c) Only one processor runs user code
- (d) Processors cannot share memory

Q11. In Rate Monotonic real-time scheduling, the highest priority is given to the task with the:

- (a) Longest period
- (b) Shortest period
- (c) Latest deadline
- (d) Longest burst

Q12. Which of the following is NOT a necessary condition for deadlock?

- (a) Mutual exclusion
- (b) Hold and wait
- (c) No pre-emption
- (d) Starvation

Q13. In a resource allocation graph where every resource type has a single instance, a cycle indicates:

- (a) No deadlock
- (b) A deadlock
- (c) A safe state
- (d) Starvation only

Q14. Requiring a process to request all its resources before it starts execution prevents which deadlock condition?

- (a) Mutual exclusion
- (b) Hold and wait
- (c) No pre-emption
- (d) Circular wait

Q15. The Banker's algorithm is used for:

- (a) Deadlock prevention
- (b) Deadlock avoidance
- (c) Process synchronization only
- (d) CPU scheduling

Answer Key

Q	Ans	Q	Ans	Q	Ans
1	b	6	a	11	b
2	c	7	c	12	d
3	b	8	b	13	b
4	c	9	b	14	b
5	b	10	b	15	b

Short explanations. Q1: the short-term scheduler selects from the ready queue. Q3: one long process delays all short ones behind it. Q4: SJF is provably optimal for average waiting time. Q5: with a huge quantum every process finishes in its first turn. Q7: $(0 + 24 + 27) / 3 = 17$. Q8: order P4, P1, P3, P2 gives $(0 + 3 + 9 + 16) / 4 = 7$. Q12: starvation is a different problem, not a Coffman condition. Q13: single instance, so a cycle is both necessary and sufficient. Q14: the process then never holds one resource while waiting for another.

Broad Questions

- 1 Define CPU scheduling. Explain the objectives of scheduling and describe the different scheduling criteria used to compare algorithms.
- 2 Explain the CPU and I/O burst cycle with a neat diagram. How does the distribution of CPU bursts help in selecting a scheduling algorithm?
- 3 Differentiate between pre-emptive and non-pre-emptive scheduling. Mention the four circumstances in which CPU-scheduling decisions are taken, with suitable examples of algorithms of each type.
- 4 Describe the First Come First Served and Shortest Job First scheduling algorithms. For processes P1, P2, P3, P4 with burst times 7, 4, 1, 4 ms arriving at time 0, draw Gantt charts and calculate the average waiting time for FCFS and non-pre-emptive SJF.
- 5 Explain Round Robin scheduling. For processes with burst times 10, 5 and 8 ms and time quantum 3 ms, draw the Gantt chart and calculate the average waiting and turnaround time. Discuss the effect of the time quantum on performance.
- 6 Explain priority scheduling. What is starvation? How does aging solve it? Compare pre-emptive and non-pre-emptive priority scheduling.
- 7 Describe multilevel queue and multilevel feedback queue scheduling with neat diagrams. State the parameters that define a multilevel feedback queue scheduler.
- 8 Discuss multiprocessor scheduling and real-time scheduling. Explain asymmetric and symmetric multiprocessing, processor affinity, load balancing, hard and soft real-time systems, and the Rate Monotonic and Earliest Deadline First algorithms.
- 9 Define deadlock. Explain the system model and the four necessary conditions for deadlock. Describe the resource allocation graph and show how it can be used to detect a deadlock. What is a critical region and how is mutual exclusion related to it?
- 10 Explain deadlock prevention and deadlock avoidance. What are safe and unsafe states? Describe the Banker's algorithm with its data structures, and use it to find a safe sequence for a system with five processes and three resource types of your own choice.

- End of Chapter 4 -