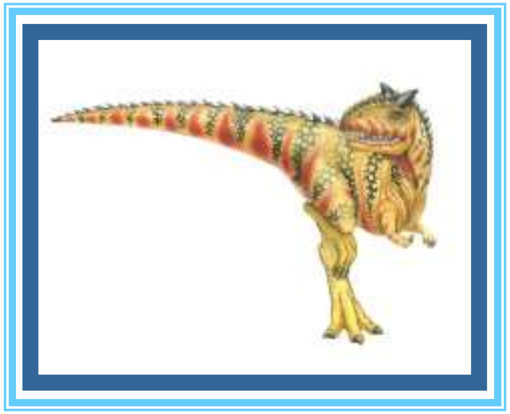
Chapter 6: CPU Scheduling



Background

- CPU scheduling is the basis of multi-programming operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive
- In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms

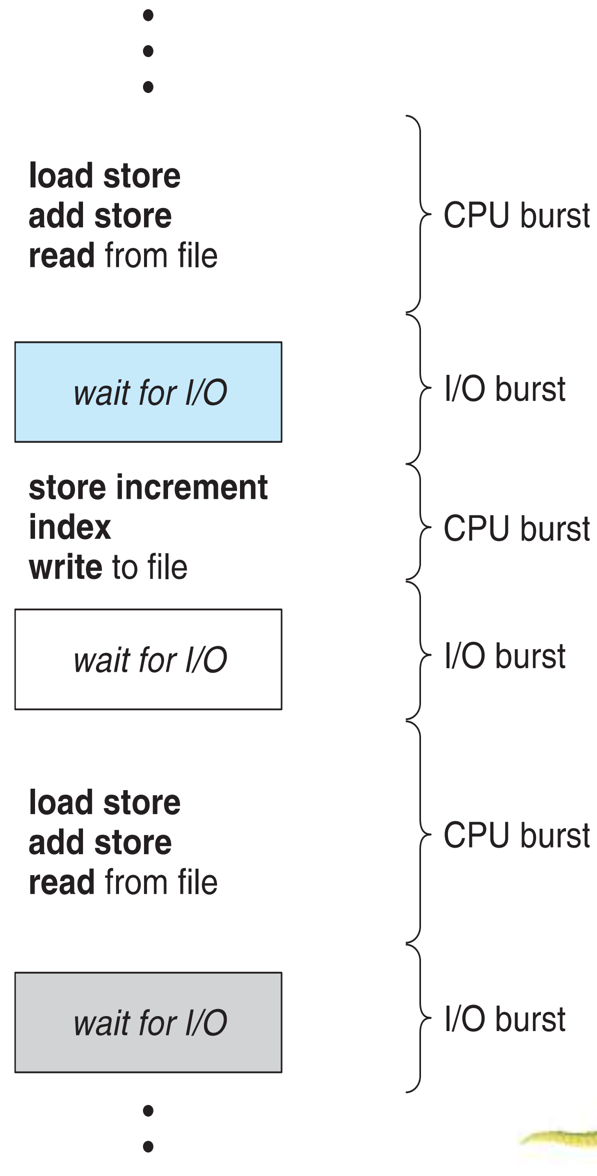
Background

- Review the life cycle of a process in a single-processor
  - A process is executed until it must wait, typically for the completion of some I/O request
  - In a simple computer system, CPU time is wasted!
- Objective of multiprogramming
  - Having some process running at all times to maximize CPU utilization
  - When one has to wait, takes the CPU away, gives to another

CPU scheduling

Basic Concepts

- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- Maximum CPU utilization obtained with multiprogramming
- CPU burst distribution is of main concern

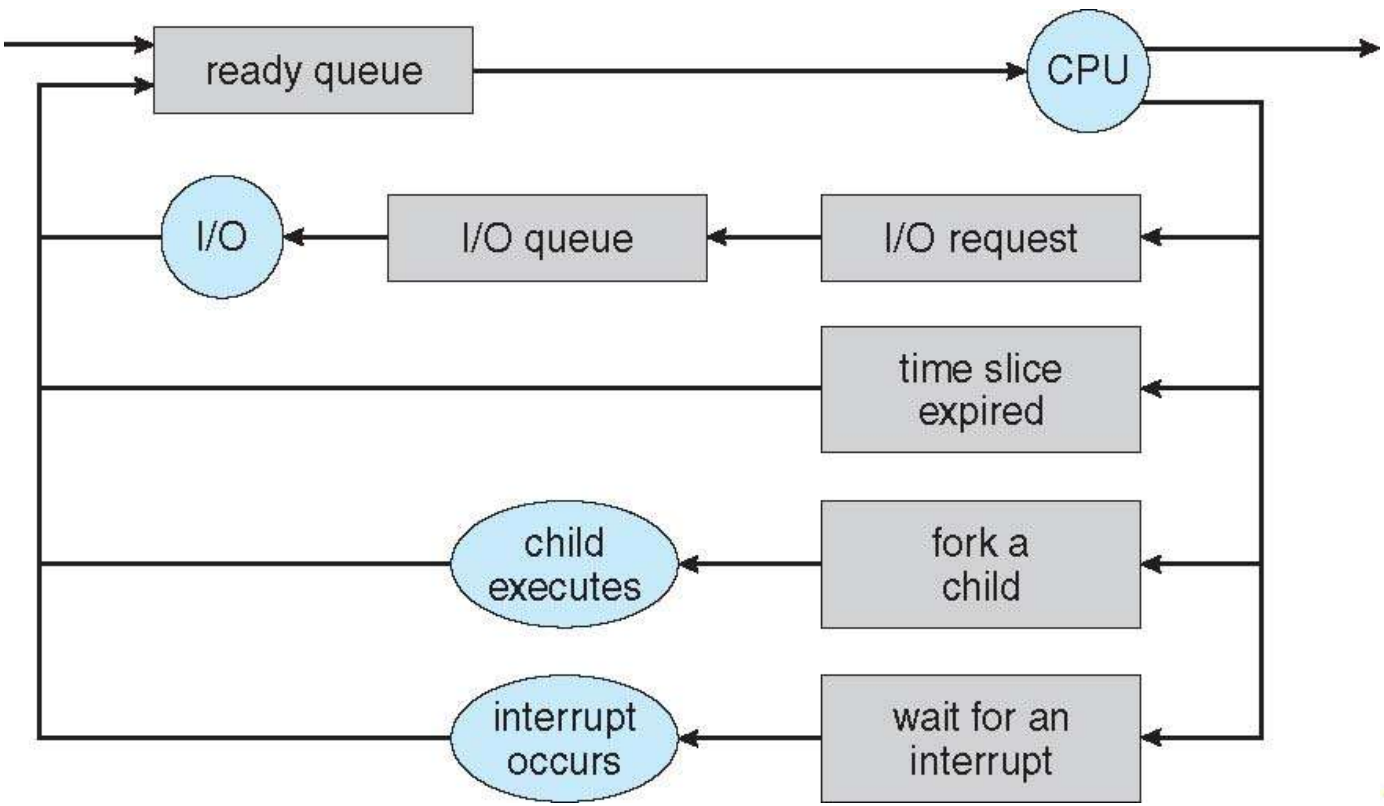


Review: Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked **frequently** (milliseconds)
  - It must be fast, because of the short time between executions
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked **infrequently** (seconds, minutes), may be slow
  - The long-term scheduler controls the **degree of multiprogramming**

CPU Scheduler

- **Short-term scheduler** (or **CPU scheduler**) – selects from among the processes in ready queue, and allocates CPU to one of them
  - Queue may be ordered in various ways
- When can process scheduling take place?



Scheduling Criteria

- What criteria make sense?
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process: from **time of submission** to **time of completion**
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - CPU scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue
- **Response time** – amount of time it takes from when a request was **submitted** until the **first response** is produced, **not output** (for time-sharing environment)

Difference between Turnaround time and Response time?

Scheduling Algorithm Optimization Criteria

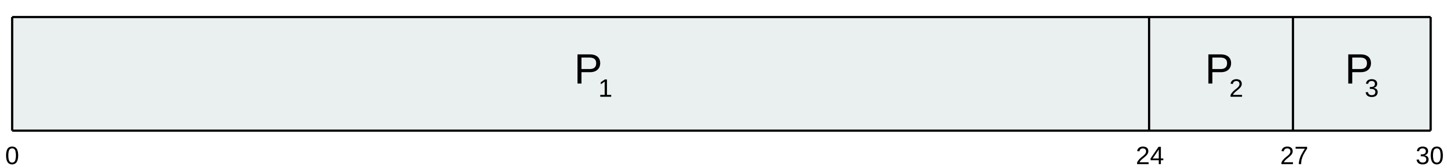
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come, First-Served (FCFS) Scheduling

FCFS: the process that requests the CPU first is allocated the CPU first

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- The average waiting time under the FCFS policy is often quite long
- The FCFS scheduling algorithm is non-preemptive

Thus, the average waiting time under the FCFS policy is generally **not minimal** and may vary substantially if the processes' CPU burst times vary greatly

Shortest-Job-First (SJF) Scheduling

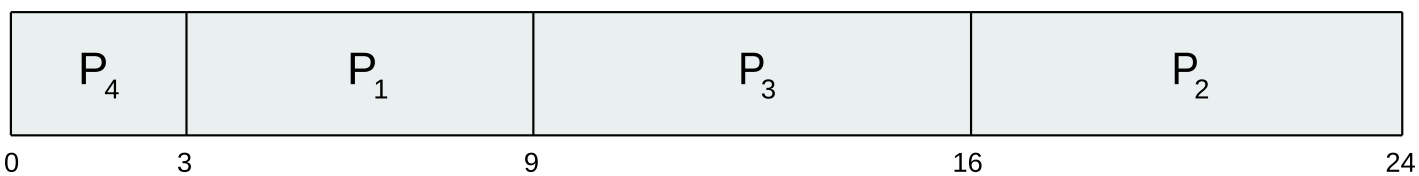
- Associate with each process **the length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time
  - When the CPU is available, it is assigned to the process that has the ***smallest next CPU burst***
  - See example in next slide

- Moving a short process before a long one decreases the waiting time of the short process more that it increases the waiting time of the long process. Consequently, the average waiting time decreases

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- If in FCFS, Average waiting time = (0 + 6 + 14 + 21) / 4 = 10.25

Shortest-Job-First (SJF) Scheduling

- SJF is optimal – gives **minimum average waiting** time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Shortest-remaining-time-first

- The SJF algorithm can be either preemptive or non-preemptive
  - Non-preemptive: Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O

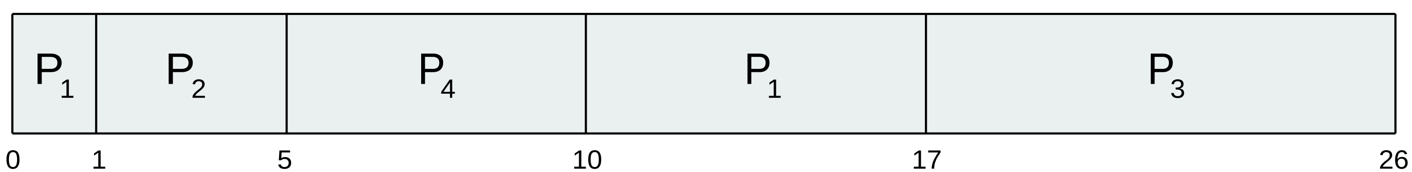
- **Preemptive SJF** scheduling is also called **shortest-remaining-time-first**
  - When a new process arrives at the ready queue while a previous process is still executing
  - The next CPU burst of the newly arrived process may be **shorter than what is left of the currently executing process**
  - A preemptive SJF algorithm will preempt the currently executing process

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Non-preemptive SJF would result in an average waiting time of 7.75 msec

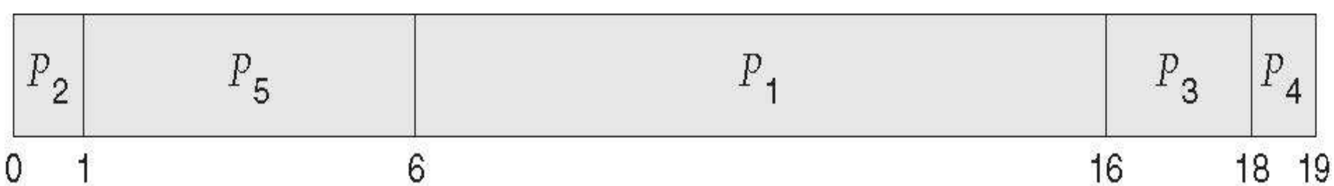
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - Preemptive
    - When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
    - A preemptive priority scheduling preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  - Non-preemptive
- SJF is priority scheduling where priority is the inverse of next CPU burst time
- See example in next slide

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Priority Scheduling

- Problem ≡ **Starvation** – low priority processes may never execute in preemptive case
- Solution ≡ **Aging** – gradually increasing the priority of processes that wait in the system for a long time

Round Robin (RR)

- Similar to FCFS, but **preemption** is added to enable the system to switch between processes
- Each process gets **a small unit of CPU time** (**time quantum** *q*), usually 10-100 milliseconds. **After this time has elapsed, the process is preempted and added to the end of the ready queue.**
- Treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, **sets a timer to interrupt after 1 time quantum**, and dispatches the process
- **Timer interrupts every quantum to schedule next process**

Round Robin (RR)

- One of two things will then happen
- If the process have a CPU burst of less than 1 time quantum
  - Release the CPU voluntarily
  - The scheduler will then proceed to the next process in the ready queue
- If the CPU burst of the currently running process is longer than 1 time quantum
  - The timer will go off and will cause an interrupt to the operating system.
  - A context switch will be executed, and the process will be put at the tail of the ready queue.
  - The CPU scheduler will then select the next process in the ready queue.

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrive at time 0, time quantum = 4
- The Gantt chart is:

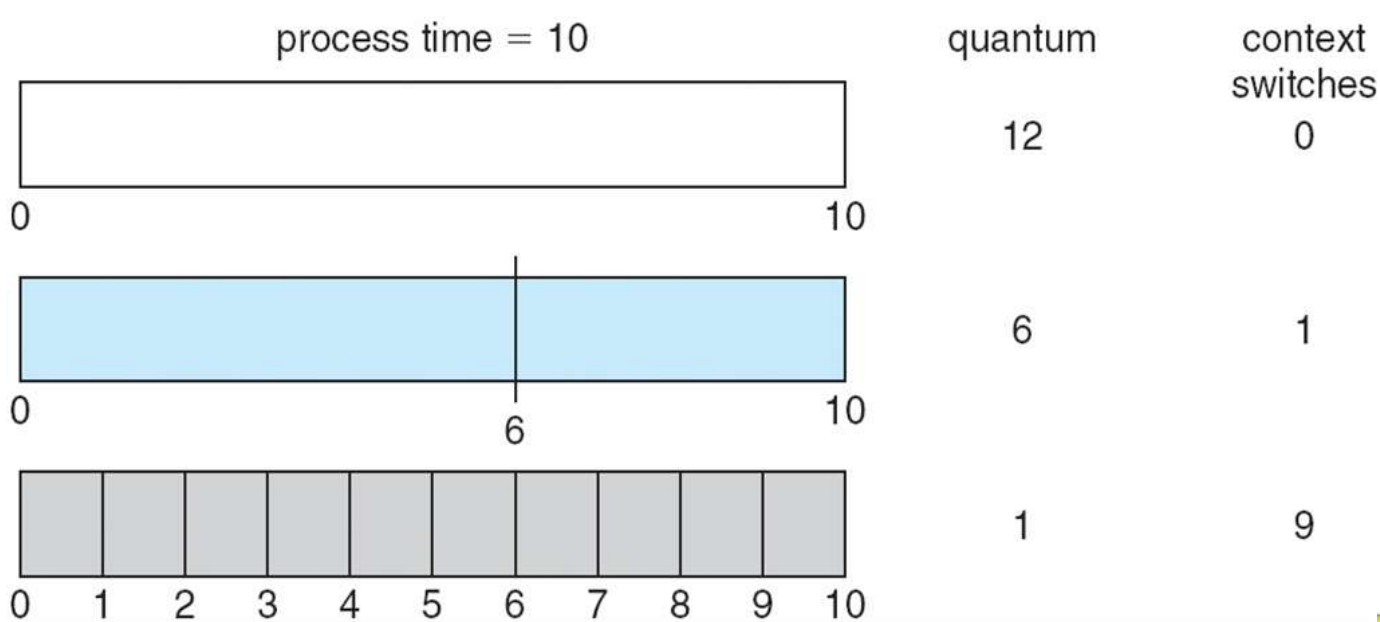
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Average waiting time: [(10-4) + (4-0) + (7-0)] / 3 = 17/3 = 5.66
- Typically, higher average turnaround than SJF, but better ***response***

Time Quantum and Context Switch Time

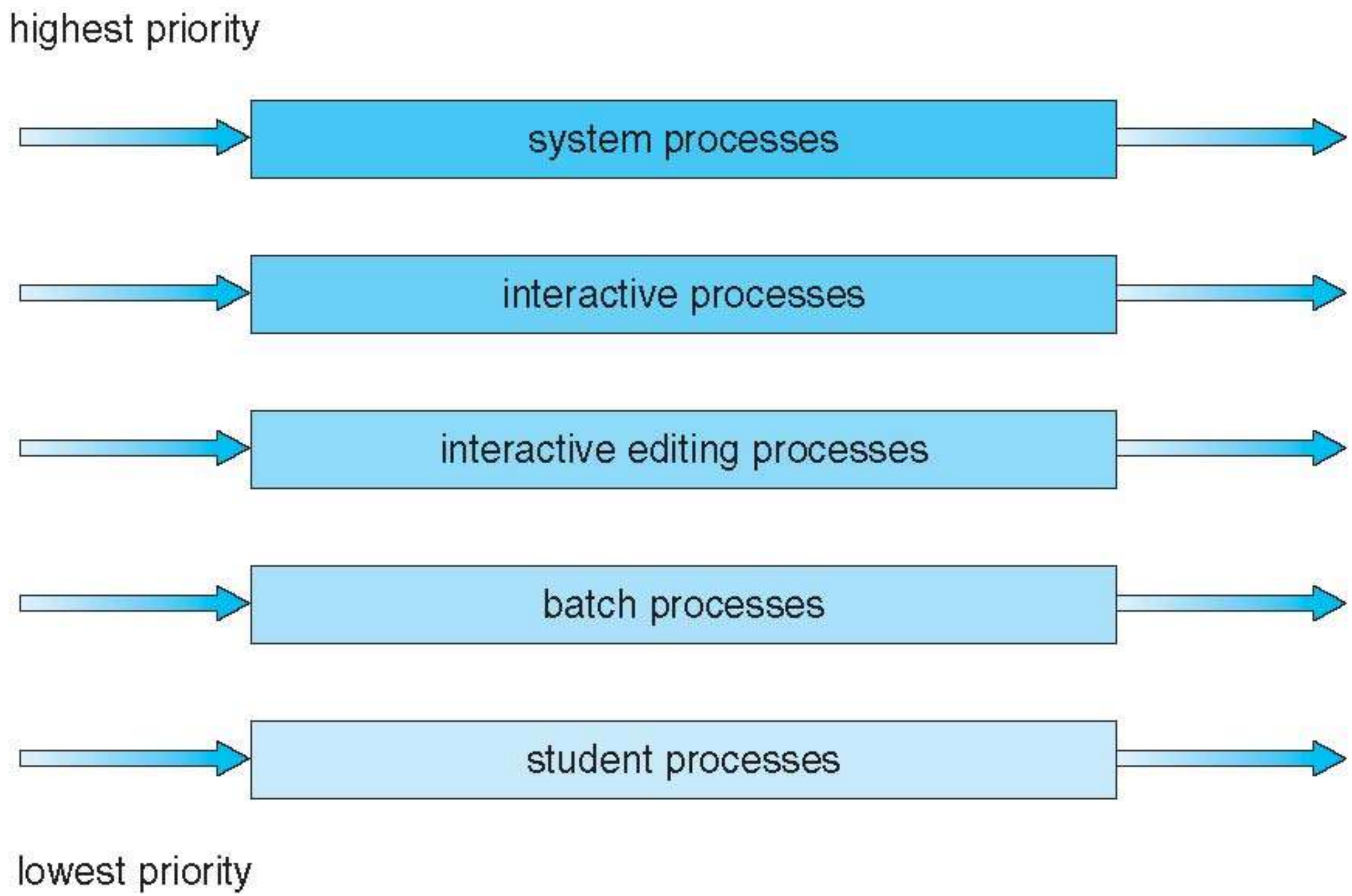
- The performance of the RR algorithm depends heavily on the size of the time quantum.
  - *q* large ⇒ FCFS
  - *q* small ⇒ results in a large number of context switch
  - *q* should be large compared to context switch time, otherwise overhead is too high



Multilevel Queue

- A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues
- The processes are permanently assigned to one queue,
  - generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm:
  - foreground (interactive) – RR
  - background (batch) – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation. (See Example)
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, and 20% to background in FCFS

Multilevel Queue Scheduling

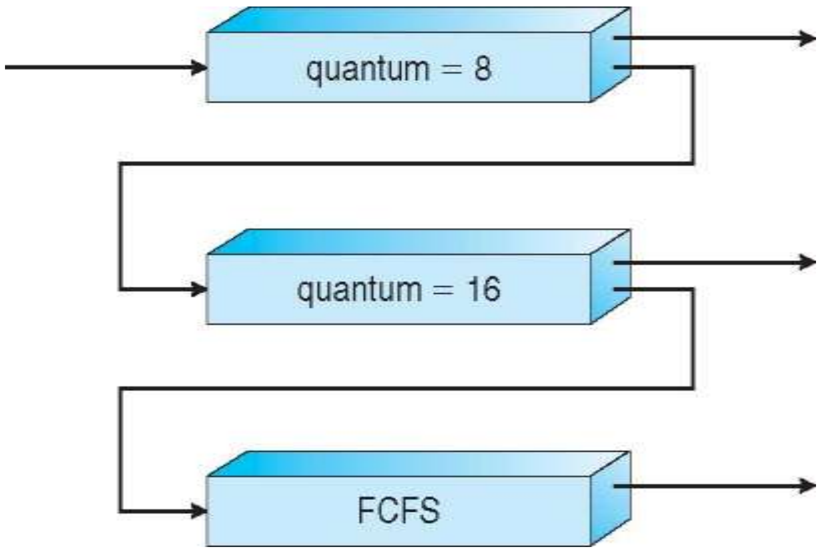


Multilevel Feedback Queue

- Allow a process to **move** between the various queues
- Idea: separate processes according to the characteristic of their CPU bursts
  - If a process uses too much CPU time, it will be moved to a lower-priority queue
  - Leaves I/O-bound and interactive processes in the higher-priority queues
  - In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue
    - This form **aging** prevents starvation

Example of Multilevel Feedback Queue

- Three queues:
  - ***Q_0*** – RR with time quantum 8 milliseconds
  - ***Q_1*** – RR time quantum 16 milliseconds
  - ***Q_2*** – FCFS



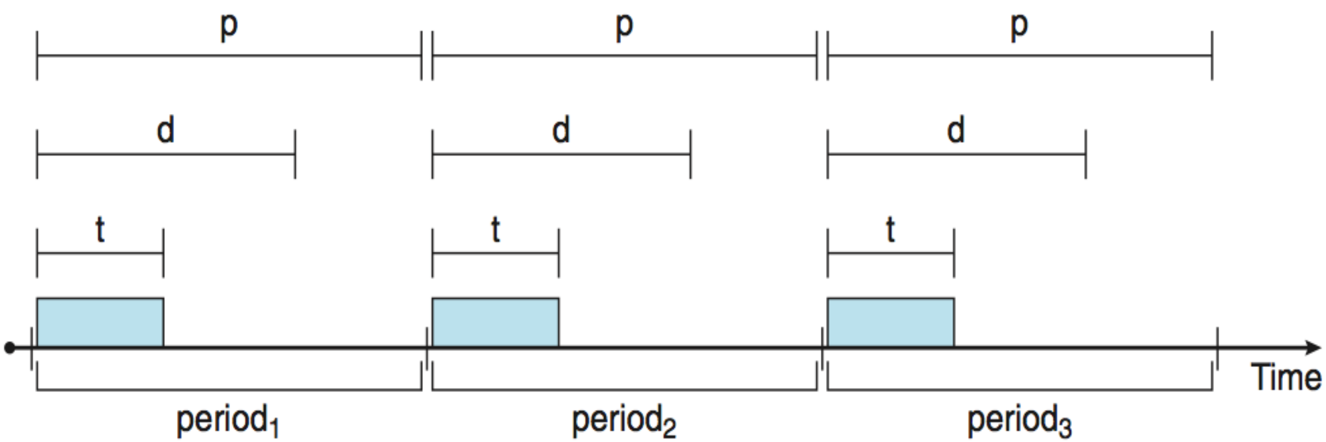
- Scheduling
  - A new job enters queue ***Q_0*** which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue ***Q_1***
  - If queue **Q_0** is empty, at ***Q_1*** job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue ***Q_2***

Priority-based Scheduling with preemption

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees **soft** real-time functionality
- A process currently running on the CPU will be **preempted** if a **higher-priority** process available to run
- For ***hard*** real-time must also provide ability to meet deadlines
- Deadline: by which it must be serviced by the CPU

Process Characteristics

- Processes have characteristics: **periodic------** process requires CPU at constant intervals
  - Has fixed processing time ***t***, deadline ***d***, period ***p***
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



Rate Monotonic Scheduling

- The rate-monotonic scheduling algorithm schedules **periodic** tasks using a **static priority policy** **with preemption**.
- If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process.

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority;
  - Longer periods = lower priority
  - Rationale: assign a higher priority to tasks that require the CPU more often

Example: Using Rate Monotonic Scheduling

- Suppose P1 is assigned a higher priority than P2 because the period of P1 is shorter than that of P2.

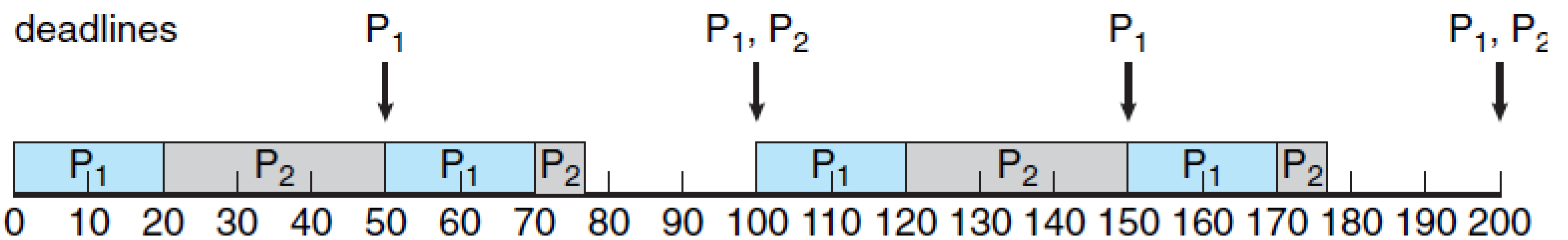


Figure 6.17 Rate-monotonic scheduling.

Deadline is satisfied!

Period: $P_1 = 50$; $P_2 = 100$;
Processing time: $t_1 = 20$; $t_2 = 35$;
Deadline: complete its CPU burst by the start of its next period.

Example: Without Using Rate Monotonic Scheduling

- Suppose P_2 is assigned a higher priority than P_1

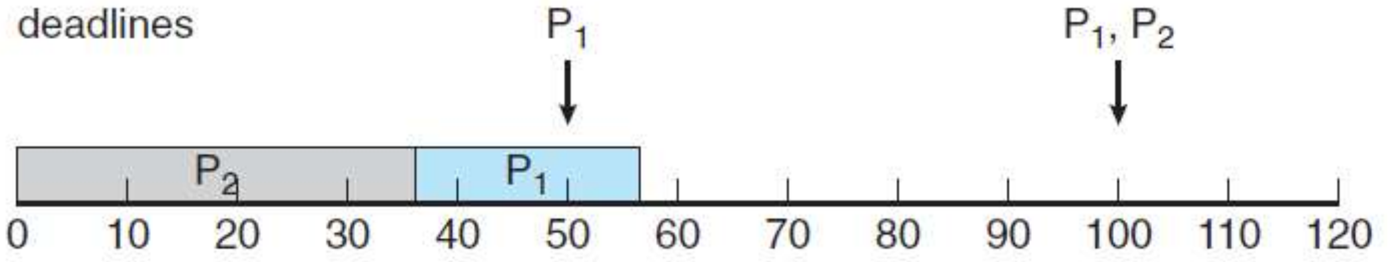


Figure 6.16 Scheduling of tasks when P_2 has a higher priority than P_1.

P1 misses its first deadline at time 50

Period: P_1 = 50; P_2 = 100;
Processing time: t_1 = 20; t_2 = 35;
Deadline: complete its CPU burst by the start of its next period.

Earliest Deadline First Scheduling (EDF)

- Priorities are **dynamically** assigned according to **deadlines**:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority

- What the difference between rate-monotonic scheduling and EDF?
  - In EDF, priorities may have to be adjusted to reflect the deadline of the newly runnable process.
  - However, in rate-monotonic scheduling, priorities are fixed

Earliest Deadline First Scheduling (EDF) (cont.)

- Example: Earliest Deadline First Scheduling (EDF)

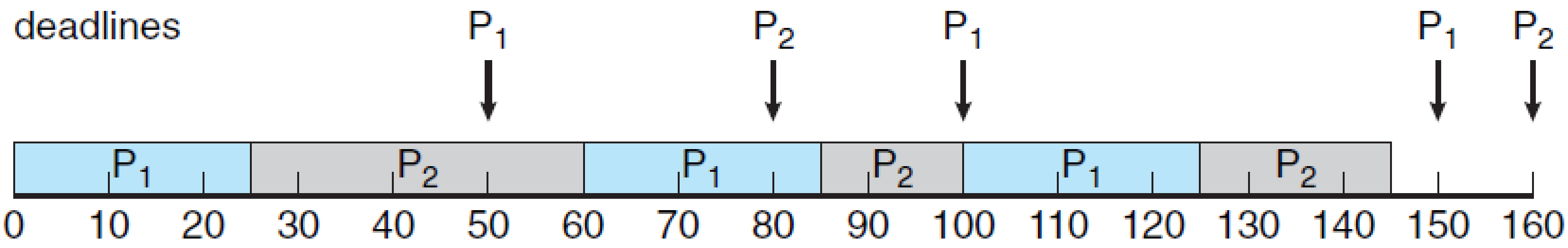


Figure 6.19 Earliest-deadline-first scheduling.

- At time 50, P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100).
- P2 is preempted at time 100 because P1 has an earlier deadline (time 150) than P2 (time 160).

Period: $P_1 = 50$; $P_2 = 80$;
Processing time: $t_1 = 25$; $t_2 = 35$;
Deadline: complete its CPU burst by the start of its next period.

Problem 01 - Question

Consider three process, all arriving **at time zero**, with total execution time of **10, 20 and 30** units respectively.

Each process spends the first **20%** of execution time doing **I/O**, the next **70%** of time doing **computation**, and the last **10%** of time doing **I/O** again.

The operating system uses a **shortest-remaining-time-first** scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst.

Assume that all I/O operations can be **overlapped** as much as possible.

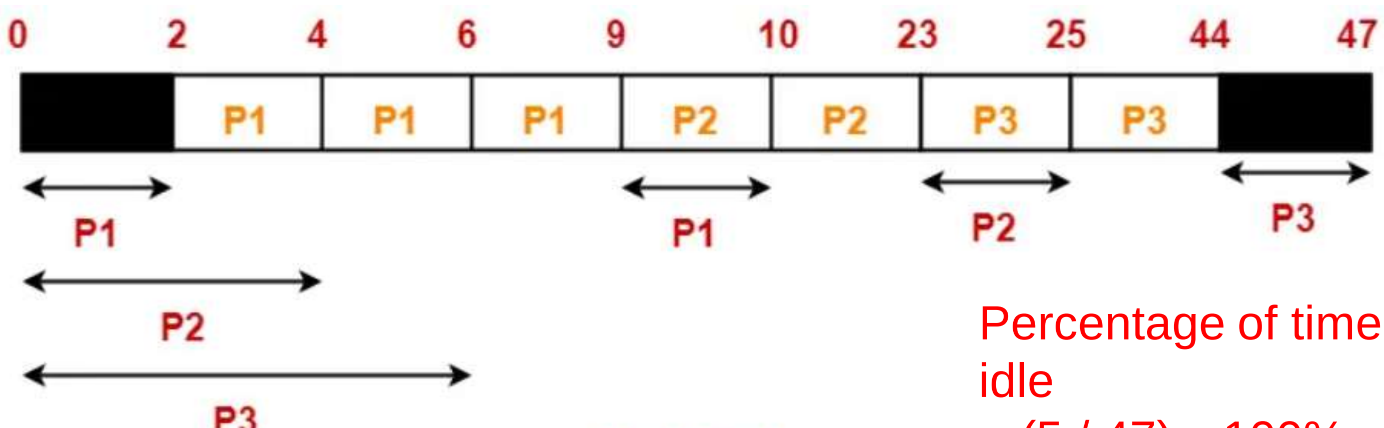
For what percentage of does the CPU remain idle?

Problem 01 - Solution

According to question, we have

	Total Burst Time	I/O Burst	CPU Burst	I/O Burst
Process P1	10	2	7	1
Process P2	20	4	14	2
Process P3	30	6	21	3

The scheduling algorithm used is Shortest-Remaining-Time-First:



Gantt Chart

Percentage of time CPU remains idle

$= (5 / 47) \times 100\%$

$= 10.638\%$

Problem 02 - Question

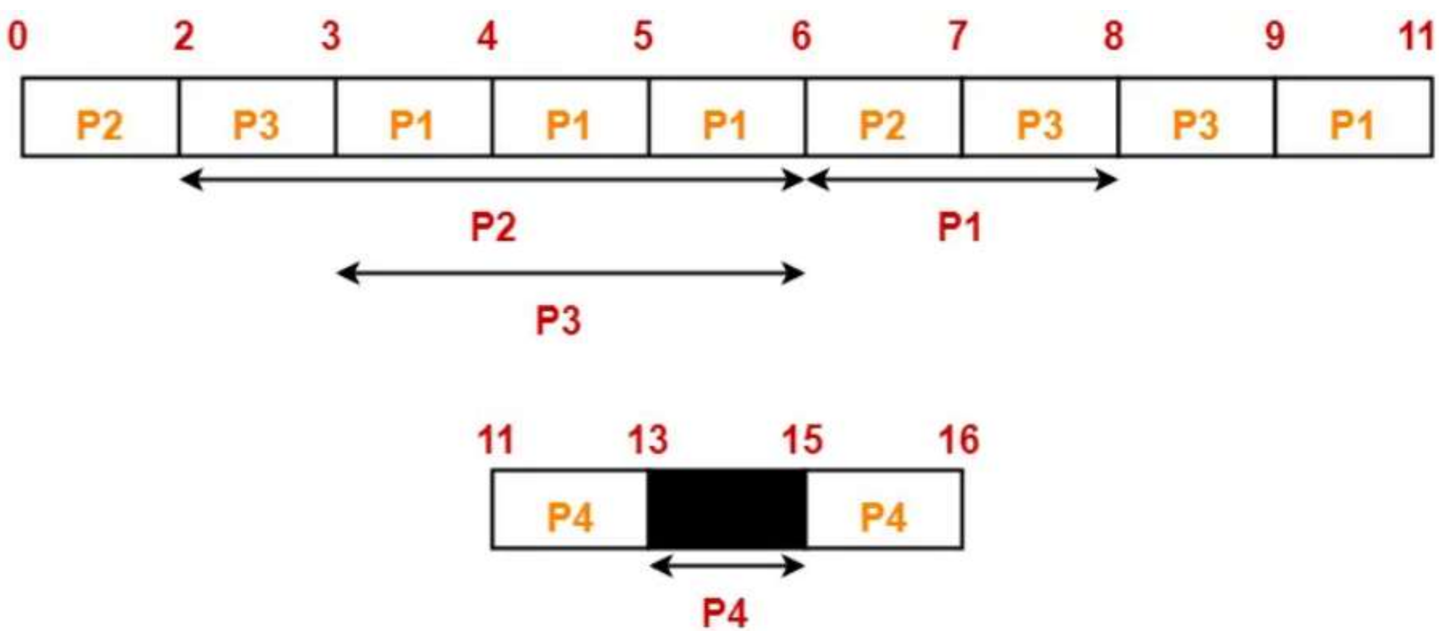
Consider the set of 4 processes whose arrival time and burst time are given below

Process No.	Arrival Time	Burst Time		
		CPU Burst	I/O Burst	CPU Burst
P1	0	3	2	2
P2	0	2	4	1
P3	2	1	3	2
P4	5	2	2	1

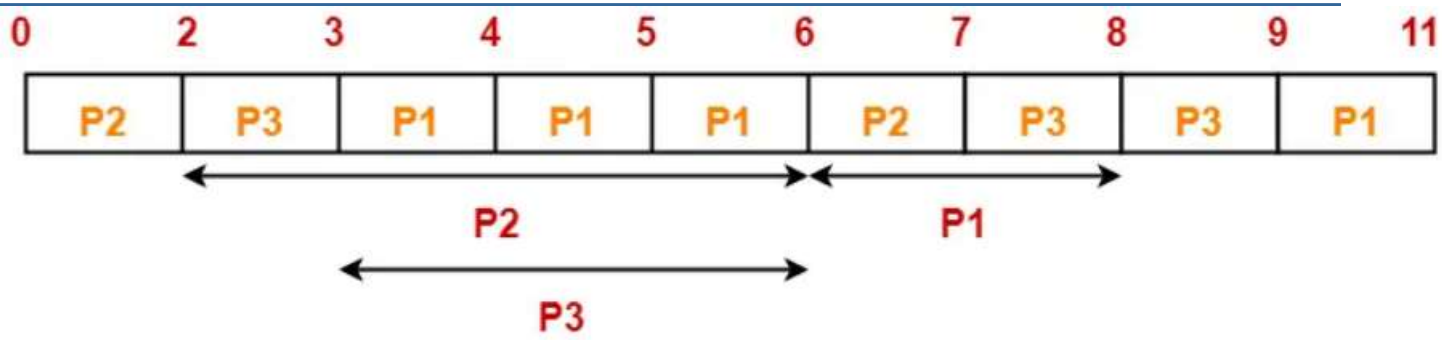
If the CPU scheduling policy is **Shortest Remaining Time First**, calculate the **average waiting time** and **average turn around time**.

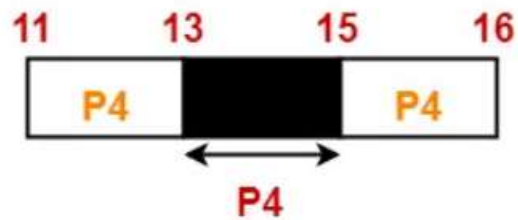
Problem 02 - Solution

Process No.	Arrival Time	Burst Time		
		CPU Burst	I/O Burst	CPU Burst
P1	0	3	2	2
P2	0	2	4	1
P3	2	1	3	2
P4	5	2	2	1



Problem 02 - Solution





Turn Around time = Exit time – Arrival time
Waiting time = Turn Around time – Burst time

Average Turn Around time = (11 + 7 + 7 + 11) / 4 = 36 / 4 = 9
Average waiting time = (6 + 4 + 4 + 8) / 4 = 22 / 5 = 4.4

Process Id	Exit time	Turn Around time	Waiting time
P1	11	11 – 0 = 11	11 – (3+2) = 6
P2	7	7 – 0 = 7	7 – (2+1) = 4
P3	9	9 – 2 = 7	7 – (1+2) = 4
P4	16	16 – 5 = 11	11 – (2+1) = 8

Problem 03 - Question

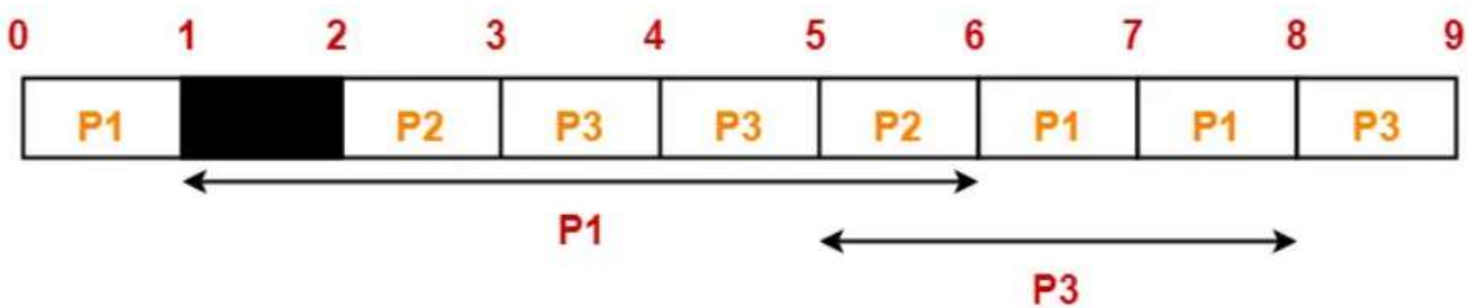
Consider the set of 4 processes whose arrival time and burst time are given below

Process No.	Arrival Time	Priority	Burst Time		
			CPU Burst	I/O Burst	CPU Burst
P1	0	2	1	5	3
P2	2	3	3	3	1
P3	3	1	2	3	1

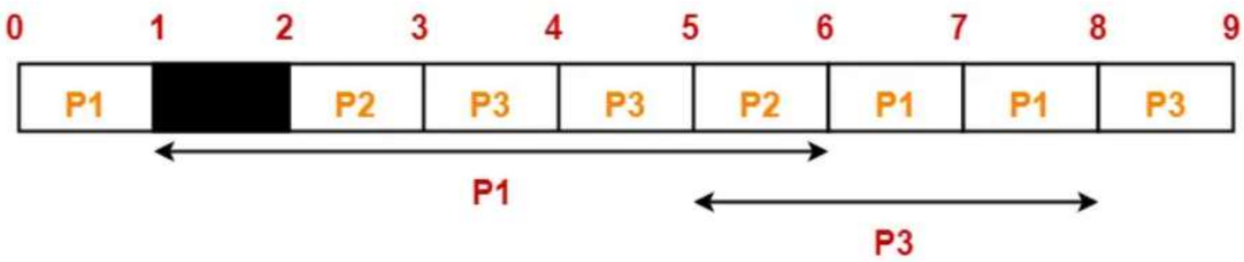
If the CPU scheduling policy is Priority Scheduling, calculate the average waiting time and average turn around time. (Lower number means higher priority)

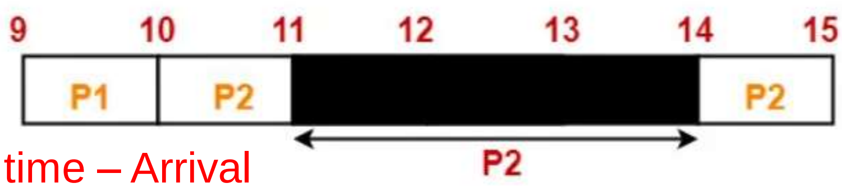
Problem 03 - Solution

Process No.	Arrival Time	Priority	Burst Time		
			CPU Burst	I/O Burst	CPU Burst
P1	0	2	1	5	3
P2	2	3	3	3	1
P3	3	1	2	3	1



Problem 03 - Solution





Turn Around time = Exit time – Arrival time

Waiting time = Turn Around time – Burst time

Average Turn Around time = (10 + 13 + 6) / 3 = 29 / 3 = 9.67

Average waiting time = (6 + 9 + 3) / 3 = 18 / 3 = 6

Process Id	Exit time	Turn Around time	Waiting time
P1	10	10 – 0 = 10	10 – (1+3) = 6
P2	15	15 – 2 = 13	13 – (3+1) = 9
P3	9	9 – 3 = 6	6 – (2+1) = 3

End of Chapter 6