LAB NO:5 **Date:**

CPU SCHEDULING ALGORITHMS

Objectives:

1. Understand the different CPU scheduling algorithms.
2. Compute the turnaround time, response time and waiting time for each process.

1. Basic Concepts :

CPU scheduling is the basis of multi programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In a single-processor system, only one process can run at a time; others(if any) must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Scheduling Criteria & Optimization:

- CPU utilization – keep the CPU as busy as possible
    - Maximize CPU utilization
- Throughput – # of processes that complete their execution per time unit
    - Maximize throughput
- Turnaround time – amount of time to execute a particular process
    - Minimize turnaround time

- Waiting time – amount of time a process has been waiting in the ready queue
    - Minimize waiting time
- Response time – time from the submission of a request until the first response is produced (response time, is the time it takes to start responding, not the time it takes to output the response)
    - Minimize response time

CPU Scheduling algorithms:

(i) First-Come First Served (FCFS) Scheduling:

The process that requests the CPU first is allocated the CPU first.

(ii) Shortest-Job-First (SJF) Scheduling:

This algorithm associates with each process the length of its next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Two schemes:

- Non-preemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst.
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

(iii) Priority Scheduling:

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority. A smaller value means a higher priority.

Two schemes:

- Preemptive
- Non-preemptive

Note: SJF is a priority scheduling where priority is the predicted next CPU burst time.

(iv) Round-Robin (RR) Scheduling:

The RR scheduling is the Preemptive version of FCFS. In RR scheduling, each process

gets a small unit of CPU time (time quantum). Usually 10-100 ms. After quantum expires, the process is preempted and added to the end of the ready queue.

(v) Multilevel Queue (MQ) Scheduling:

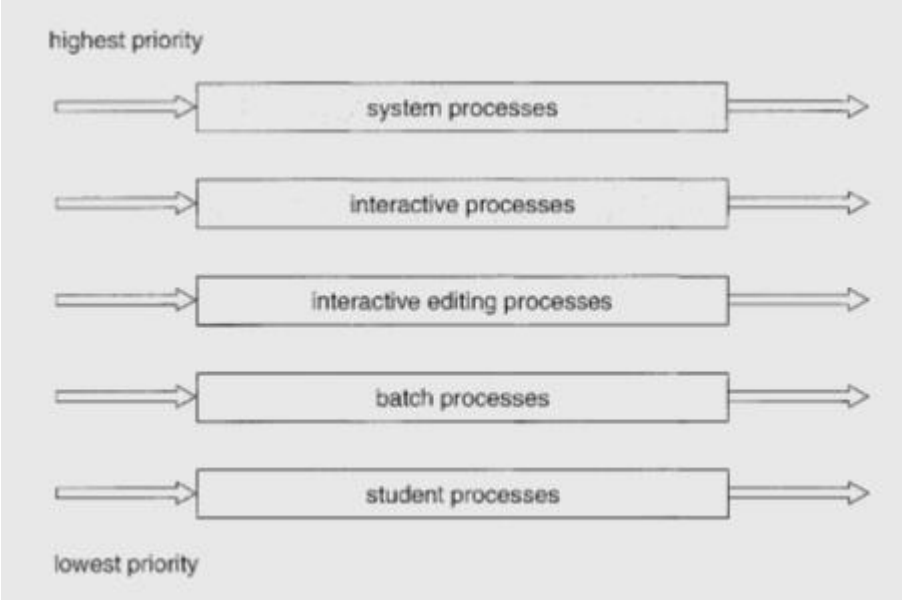


Figure 5.1: Multilevel queue scheduling

MQ scheduling is used when processes can be classified into groups. For example, **foreground** (interactive) processes and **background** (batch) processes. A MQ scheduling algorithm partitions the ready queue into several separate queues:

- foreground (interactive)
- background (batch)

Each process assigned to one queue based on its memory size, process priority, or process type. Each queue has its own scheduling algorithm

- foreground – RR
- background – FCFS

Scheduling must be done between the queues

- Fixed priority scheduling; (i.e., serve all from foreground then from background as shown Fig. 5.1).

- Time slice – each queue gets a certain portion of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS.

(vi) Multilevel Feedback Queue (MFQ) Scheduling:

In MQ scheduling algorithm processes do not move from one queue to the other. In contrast, the MFQ scheduling algorithm, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme 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**.**

Example of Multilevel Feedback Queue:

Consider multilevel feedback queue scheduler with three queues as shown in Fig. 5.2.

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

MFQ Scheduling

- A process queue in Q_0 *is given a time quantum of* 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the tail of queue Q_1.
- When Q_0 is empty, the process at the head of Q_1 is given a quantum of 16 milliseconds. If it does not complete, it is pre-empted and moved to queue Q_2 .

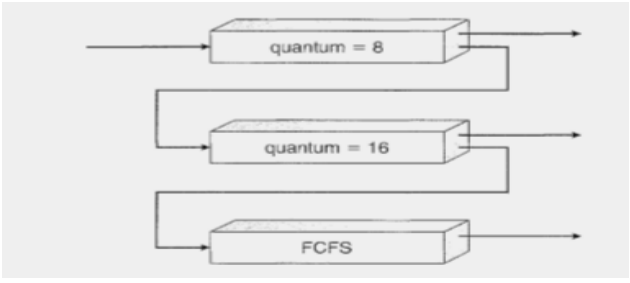


Figure 5.2: Multilevel feedback queues

2. Problem Description and Algorithm :

2.1 Round Robin Scheduling (RR):

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to (First Come First Serve) FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. To implement RR scheduling, we keep the ready queue as a FIFO queue o£ processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the firstprocess from the ready queue, sets a timer to interrupt after 1 time quantum, and dis- patches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, 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: Time quantum=3

Process	Arrival Time	Execution Time
P1	0	8
P2	5	4
P3	3	9
P4	7	16

P1	P3	P1	P2	P3	P4	P1	P2	P3	P4

0 3 6 9 12 15 18 20 21 24 37

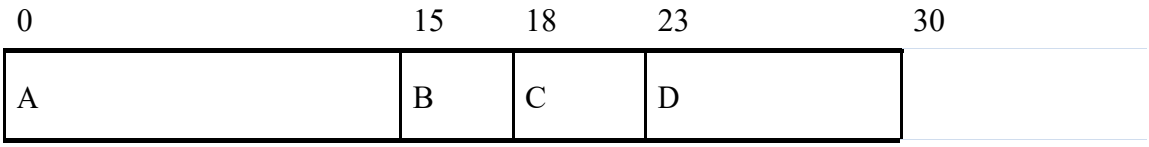
Average waiting time: (12+12+12+14)/4 = 12.5.

2.2 Shortest Job First (SJF):

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Process	Arrival time	Burst time
A	0	15
B	5	3
C	8	5
D	10	7

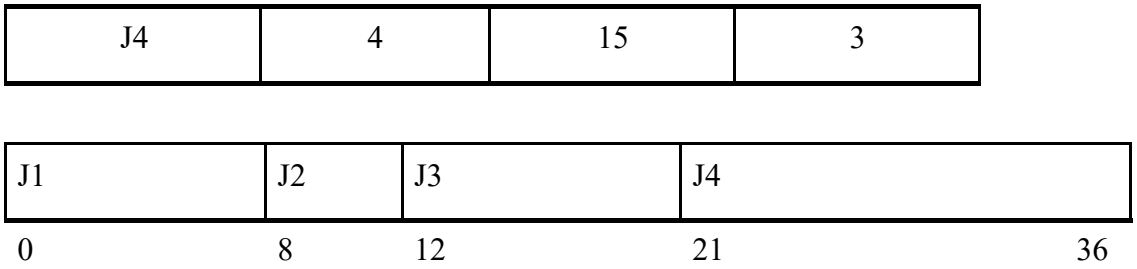
SJF Waiting time A= 0, B=10, C=10, D=13. TT A=15, B=13, C=15, D=20.



2.3 Priority Scheduling:

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Process	Arrival Time	Execution Time	Priority
J1	0	8	1
J2	2	4	2
J3	9	9	2



Average waiting time: (0+3+17)/3 = 6.67.

1. Introduction to Real Time Systems (RTSs)

A real-time system is a computer system that requires not only that the computing results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed even if correct-may be of no real value. To illustrate, consider an autonomous robot that delivers mail in an office complex. If its vision-control system identifies a wall after the robot has walked into it, despite correctly identifying the wall, the system has not met its requirement. Contrast this timing requirement with the much less strict demands of other systems. In an interactive desktop computer system, it is desirable to provide a quick response time to the interactive user, but it is not mandatory to do so. Some systems -such as a batch-processing system-may have no timing requirements whatsoever.

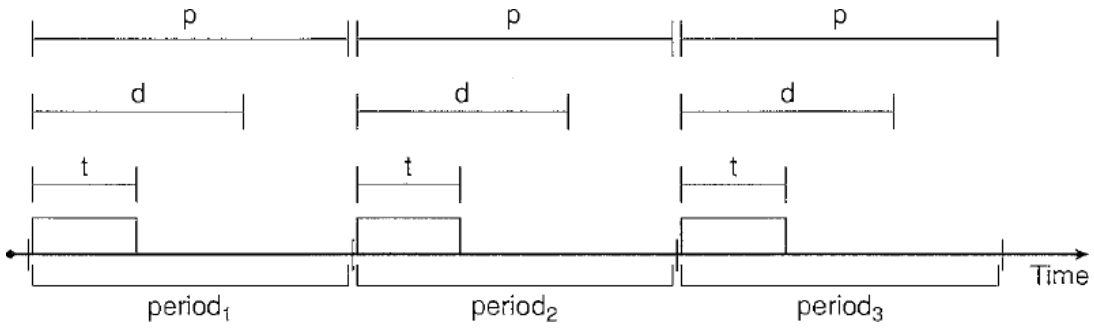
Types of RTSs

Real-time computing is of two types: hard and soft. A hard real-time system has the most stringent requirements, guaranteeing that critical realtime tasks be completed within their deadlines. Safety-critical systems are typically hard real-time systems. A soft real-time system is less restrictive, simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes. Many commercial operating systems-as well as Linux-provide soft real-time support.

Process Characteristics

Certain characteristics of the processes are as follows: First, the processes are considered periodic. That is, they require the CPU at constant intervals (periods). Each periodic process has a fixed processing timet once it acquires the CPU, a deadline *d* by which time it must be serviced by the CPU, and a period *p.* The relationship of the processing time, the deadline, and the period can be expressed as $0 <= t <= d <= p$. The rate of a periodic task

is 1 / p. The Figure below illustrates the execution of a periodic process over time. Schedulers can take advantage of this relationship and assign priorities according to the deadline or rate requirements of a periodic process.



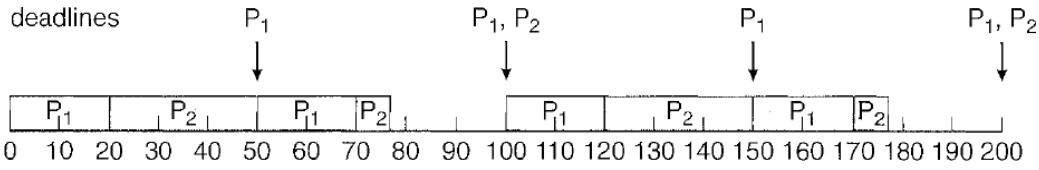
2. Rate-Monotonic Scheduling

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. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes P1 and P2. The periods for P1 and P2 are 50 and 100, respectively-that is, Pl =50 and P2 = 100. The processil1.g times are t1 = 20 for P1 and t2 = 35 for P2 . The deadline for each process requires that it complete its CPU burst by the start of its next period. We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process Pi as the ratio of its burst to its period -ti I Pi -the CPU utilization of P1 is 20/50 = 0.40 and that of P2 is 35/100 = 0.35, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Example:

Suppose we use rate-monotonic scheduling, in which we assign P1 a higher priority than P2, since the period of P1 is shorter than that of P2.



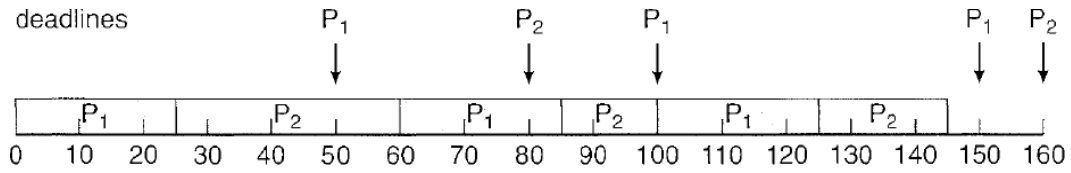
P1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P2 starts running at this point and runs until time 50. At this time, it is preempted by P1, although it still has 5 milliseconds remaining in its CPU burst. P1 completes its CPU burst at time 70, at which point the scheduler resumes P2. P2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P1 is scheduled again. Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

3. Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly rmmable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

Example:

P1 has values of p1 = 50 and t1 = 25 and that P2 has values of p2 = 80 and t2 = 35. The EDF scheduling of these processes is shown in Figure below.



Process P1 has the earliest deadline, so its initial priority is higher than that of process P2

• Process P2 begins rmming at the end of the CPU burst for P1. However, whereas rate-monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running. P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100). Thus, both P1 and P2 meet their first deadlines. Process P1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100.

P2 begins rum1ing at this point only to be preempted by P1 at the start of its next period at time 100. P2 is preempted because P1 has an earlier deadline (time 150) than P2 (time 160). At time 125, P1 completes its CPU burst and P2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process a1mom1ce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal-theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

Lab Exercise

1. Consider the following set of processes, with length of the CPU burst given in milliseconds:

Process	Arrival time	Burst time	*Priority*
P1	0	60	3
P2	3	30	2
P3	4	40	1
P4	9	10	4

Write a menu driven C/C++ program to simulate the following CPU scheduling algorithms. Display Gantt chart showing the order of execution of each process. Compute waiting time and turnaround time for each process. Hence, compute average waiting time and average turnaround time.

(i) FCFS (ii) SRTF (iii) Round-Robin (quantum = 10) iv) non-preemptive priority (higher the number higher the priority)

2. Write a C program to simulate multi-level feedback queue scheduling algorithm.
3. Write a C program to simulate Earliest-Deadline-First scheduling for real time systems.

Additional Exercises

1. Write C/C++ program to implement FCFS. (Assuming all the processes arrive at the same time)
2. Write a C/C++ program to implement SJF, where the arrival time is different for the processes.
3. Write a C/C++ program to simulate Rate-Monotonic scheduling for real time systems.