PROCESS SCHEDULING

- In a single-processor system, only one process can run at a time; any others 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.
- Several processes are kept in memory at one time.
- When one process has to wait, OS 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-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/0 wait.
- Processes alternate between these two states.
- Process execution begins with a **CPU burst**.
- That is followed by an **I/O burst**, which is followed by another CPU burst and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution

CPU Scheduler

- Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler** (or **CPU scheduler**).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- The ready queue is not necessarily a first-in, first-out (FIFO) queue.
- A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- All the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally process control blocks (PCBs) of the processes.

Preemptive Scheduling

- CPU-scheduling decisions may take place under the following four circumstances:

    1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

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

- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive** or **cooperative**. Otherwise, it is **preemptive**.
- So scheduling scheme can be of 2 types: **Preemptive & Non Preemptive**
- Non-preemptive scheduling: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- **Non-preemptive scheduling is also called cooperative scheduling**
- Preemptive scheduling: A running process may be preempted with a higher priority process
- **Preemptive scheduling may lead to data inconsistency (race condition).**
- Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations,

we need new mechanisms to coordinate access to shared data

- **Preemption also affects the design of the OS kernel.**
- During the processing of a system call, the kernel may involve changing important kernel data (for eg, I/0 queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure?
- Conflicts may occur. Certain OS deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch.
- This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state.
- Unfortunately, this kernel-execution model is a poor one for supporting real-time computing

Dispatcher

- Another component involved in the CPU-scheduling function is the dispatcher.
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves the following:
    1. Switching context
    2. Switching to user mode

3. Jumping to the proper location in the user program to restart that program

- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

SCHEDULING CRITERIA

- Different CPU-scheduling algorithms have different properties
- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- These criteria are used for comparison and judgment which algorithm is to be best.
- The criteria are:

1. **CPU utilization**: We want to keep the CPU as busy as possible. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. **Throughput**: If the CPU is busy executing processes, then work is being done. One measure of work is the **number of processes that are completed per time unit, called *throughput***. For long processes, this rate

may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time:** About a particular process, the important criterion is how long it takes to execute that process. **The interval from the time of submission of a process to the time of completion is the *turnaround time.*** Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0.
4. **Waiting time**: *Waiting time* is the sum of the periods spent waiting in the ready queue.
5. **Response time**: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the **time from the submission of a request until the first response is produced. This measure, called *response time,*** is the time it takes to start responding. The response time is generally limited by the speed of the output device.

- **It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.**
- In most cases, we optimize the **average** measure. However, under some circumstances, it is desirable to

optimize the minimum or maximum values rather than the average.

- For example, to guarantee that all users get good service, we may want to **minimize** the **maximum** response time.
- It is more important to **minimize** the ***variance*** in the response time than to minimize the average response time.
- A system with reasonable and *predictable* response time may be considered more desirable than a system that is faster on the average but is highly **variable**.

SCHEDULING ALGORITHMS

1. First-Come, First-Served Scheduling (FCFS)
2. Shortest-Job-First Scheduling (SJF)
3. Priority Scheduling
4. Round-Robin Scheduling (RR)

First-Come, First-Served Scheduling (FCFS)

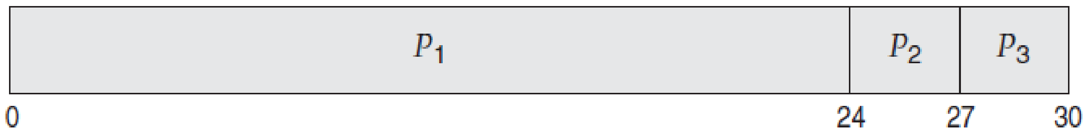
- Simplest scheduling algorithm
- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is

allocated to the process at the head of the queue. The running process is then removed from the queue.

- The code for FCFS scheduling is simple to write and understand.
- On the negative side, the average waiting time under the FCFS policy is often quite long.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

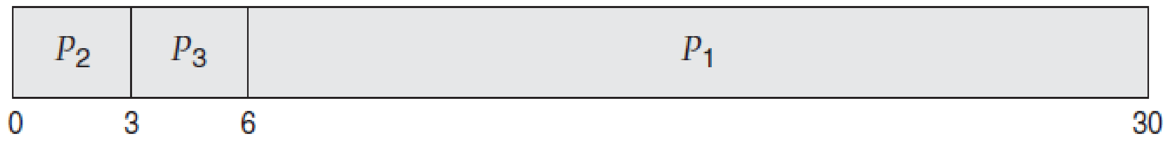
Process	Burst Time
P_1	24
P_2	3
P_3	3

- If the processes arrive in the order *P*1, *P*2, *P*3, and are served in FCFS order, we get the result shown in the following **Gantt chart**
- **Gantt chart** is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



- The waiting time is 0 milliseconds for process *P*1, 24 milliseconds for process *P*2, and 27 milliseconds for process *P*3.

- Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
- If the processes arrive in the order *P*2, *P*3, *P*1, the results will be as shown in the following Gantt chart:



- The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial.
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.
- Assume we have one big CPU-bound process and many small I/O-bound processes.
- As the processes flow around the system, the following scenario may result.
- The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
- While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move-back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process

will then move back to the ready queue and be allocated the CPU.

- Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. This is called **convoy effect** as all the other processes wait for the one big process to get off the CPU.
- This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
- FCFS scheduling algorithm is nonpreemptive.
- 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.
- The FCFS algorithm is thus troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.
- It would be disastrous to allow one process to keep the CPU for an extended period.

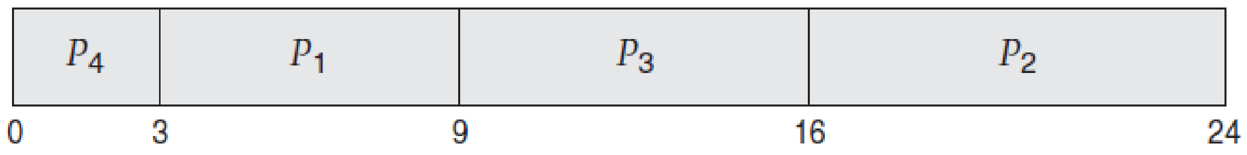
Shortest-Job-First Scheduling (SJF)

- This algorithm depends on 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.

- More appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



- The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4.
- Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.
- By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be …… milliseconds.

<<<<HOME WORK>>>

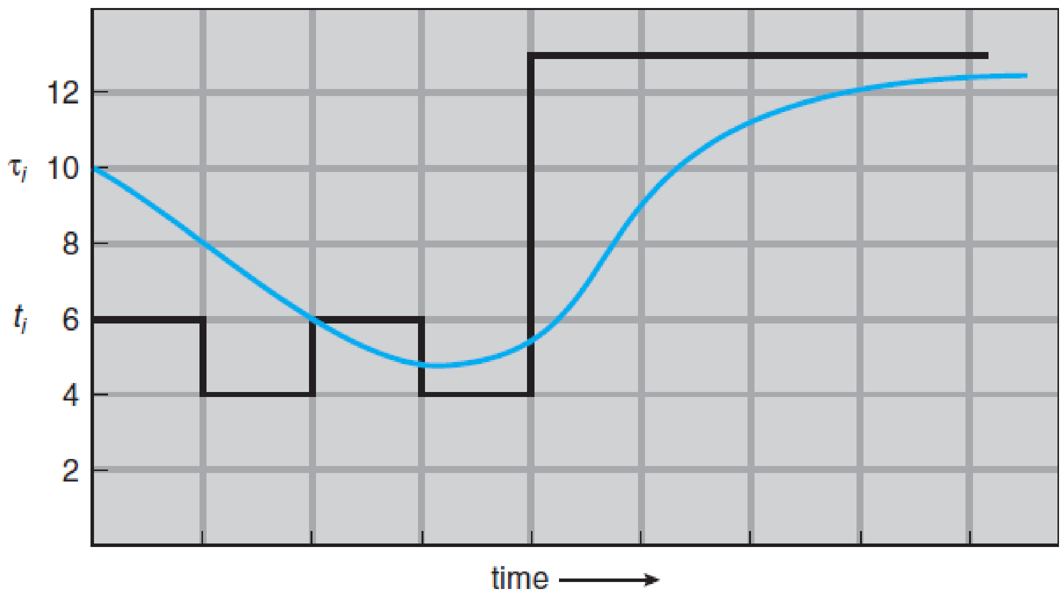
- The SJF scheduling algorithm is optimal, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.
- With short-term scheduling, there is no way to know the length of the next CPU burst.
- One approach to this problem is to try to approximate SJF scheduling.
- We may not know the length of the next CPU burst, but we may be able to predict its value.
- We expect that the next CPU burst will be similar in length to the previous ones.
- By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
- The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts.

- We can define the exponential

average with the following formula. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\tau_n.$$

The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 6.3 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

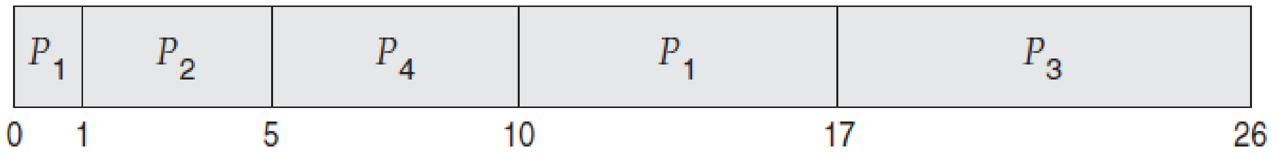
Figure 6.3 Prediction of the length of the next CPU burst.

- The SJF algorithm can be either preemptive or nonpreemptive.

- The choice arises 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, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling. **(SRTF)**
- As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

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

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



- Process $P1$ is started at time 0, since it is the only process in the queue. Process $P2$ arrives at time 1. The remaining time for process $P1$ (7 milliseconds) is larger than the time required by process $P2$ (4 milliseconds), so process $P1$ is preempted, and process $P2$ is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds.
- Nonpreemptive SJF scheduling would result in an average waiting time of …….. milliseconds.

<<<<HOME WORK>>>>

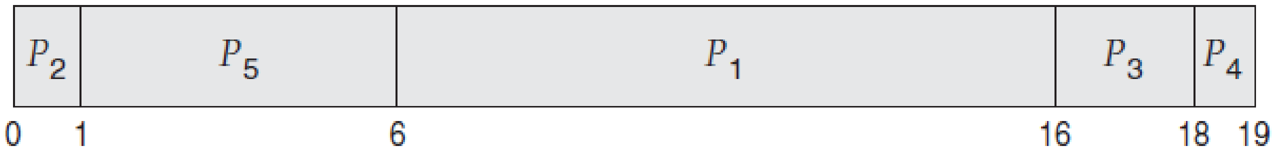
<u>PRIORITY SCHEDULING</u>

- 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.
- The SJF algorithm is a special case of the general priority scheduling algorithm.
- 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.
- There is no general agreement on whether 0 is the highest or lowest priority.

- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- Here we assume that low numbers represent high priority.
- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, *P2,* · · ·, *P5,* with the length of the CPU burst given in milliseconds:

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

Solution:



Average waiting time is (6+0+16+18+1)/5 = **8.2 mS**

- Priorities can be defined either internally or externally.
- Example for internal priorities: time limits, memory requirements, the number of open files, and the ratio of average I/0 burst to average CPU burst etc.
- External priorities are set by criteria outside the OS, such as the importance of the process, the type and amount of

funds being paid for computer use, the department sponsoring the work, and other political factors etc.

- Priority scheduling can be either preemptive or non-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 algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Drawback

- A major problem with priority scheduling algorithms is **indefinite blocking, or starvation.**
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- **A solution to the problem of indefinite blockage of low-priority processes is aging.**
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

- For eg. We can increase the priority of a waiting process by 1 in every 15 minutes. Eventually, even a process with lowest priority would have the highest priority in the system and would be executed.

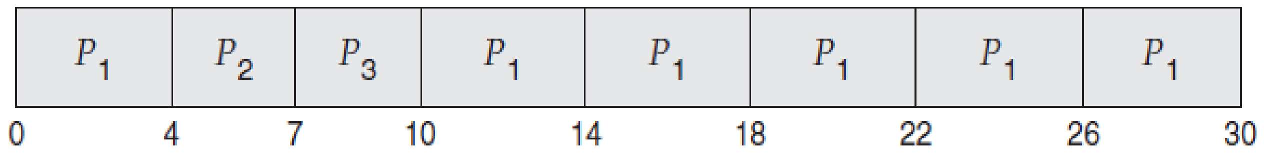
ROUND ROBIN SCHEDULING (RR)

- The **round-robin (RR) scheduling algorithm** is designed especially for time sharing systems.
- A small unit of time, called a **time quantum or time slice**, is defined.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- We keep 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.
- 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 OS. 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.
- **The average waiting time under the RR policy is often long**.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a **time quantum of 4 milliseconds**, then process P1 gets the first 4milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process
- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



- P1 waits for 6 milliseconds (10- 4), *P2* waits for 4 milliseconds, and *P3* waits for 7 milliseconds. Thus, the average waiting time is 17/3 = **5.66** milliseconds.
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. **The RR scheduling algorithm is thus preemptive.**
- **The performance of the RR algorithm depends heavily on the size of the time quantum.**
- **If the time quantum is extremely large, the RR policy is the same as the FCFS policy.**
- **If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing.**
- Also we need to consider the **effect of context switching** on the performance of RR scheduling. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

- We want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.