جامعة ألأنبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم أنظمة شبكات الحاسوب

المرحلة الرابعه

Operating System

التدريسي: أ.م.د عمر منذر حسين

Overview

- In a single-processor system, only one process can run at a time. 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.
- Almost all computer resources are scheduled before use.
- The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.
- Process execution consists of a cycle of CPU execution and I/O wait.
- Process execution begins with a CPU burst.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O 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 operating system 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.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a **FIFO queue**, a **priority queue**, a **tree**, or simply an **unordered linked list**.

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).

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 **nonpreemptive** or cooperative. Otherwise, it is **preemptive**.
- Under **nonpreemptive** scheduling, once CPU allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the **waiting** state.
- This scheduling method (**nonpreemptive**) was used by Microsoft Windows 3.x.
- Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
- Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.

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.
- 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

- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- **CPU utilization**. 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 loaded system).
- **Throughput**. is the number of processes that are completed per time unit. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. The interval from the time of submission of a process to the time of completion. 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/O.
- **Waiting time**. is the sum of the periods spent waiting in the ready queue.
- **Response time**. is the time from the submission of a request until the first response is produced.

Scheduling Algorithms
1) First-Come, First-Served Scheduling

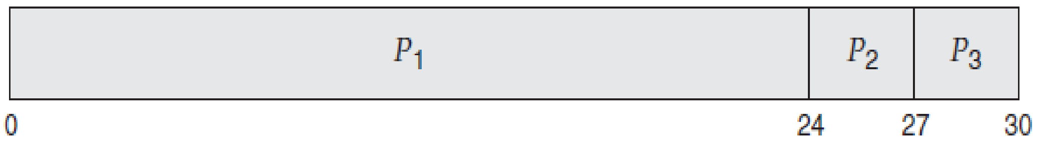
- By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters 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.

- **On the negative side**, the average waiting time under the FCFS policy is often quite long.

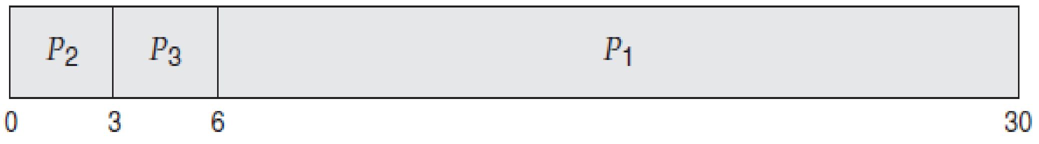
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

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, which 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, however, 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.

- **In addition, consider the performance of FCFS scheduling in a dynamic situation.**

- Assume we have one **CPU-bound** process and many **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.

❑ There is a **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.

❑ Note that the 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.

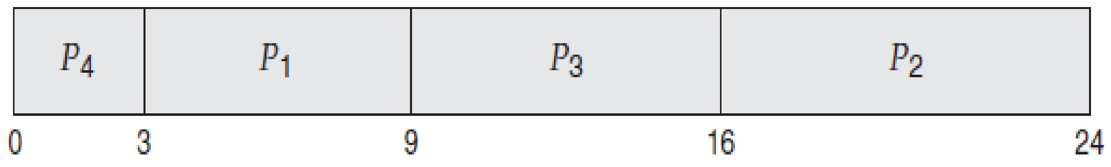
❑ The **FCFS algorithm is thus particularly troublesome for time-sharing systems**, where it is important that each user get a share of the CPU at regular intervals.

2) Shortest-Job-First Scheduling

- 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	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 10.25 milliseconds.

- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases 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.
- It cannot be implemented, 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.

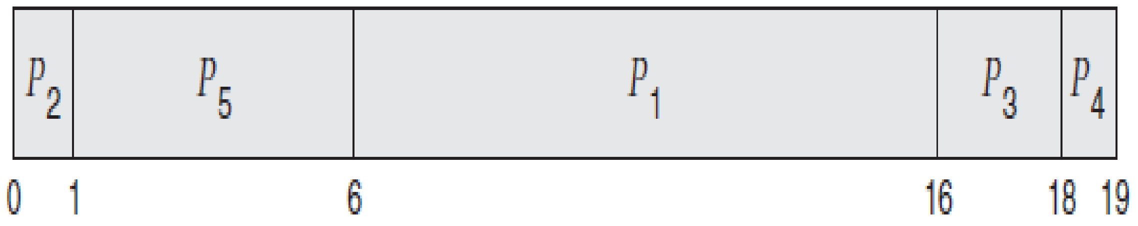
3) Priority Scheduling

- 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.

- In this text, 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

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



The average waiting time is 8.2 milliseconds.

- Priority scheduling can be either **preemptive** or **nonpreemptive**.
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

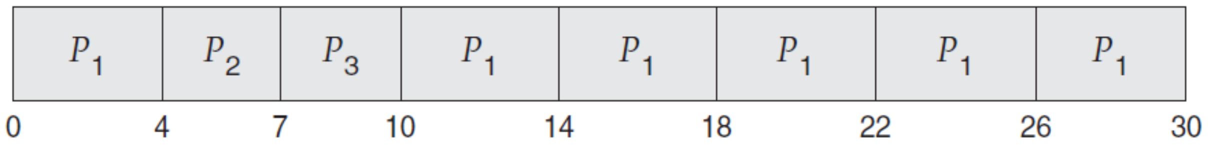
- A **preemptive** priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A **nonpreemptive** priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
- A process that is ready to run but waiting for the CPU can be considered blocked.
- 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** involves gradually increasing the priority of processes that wait in the system for a long time.

4) Round-Robin Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems.
- It is similar to 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.
- 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.
- 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 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.
- Process P2 does not need 4 milliseconds, so it quits before its time quantum expires.
- The CPU is then given to the next process, process P3. 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:



- Calculate the average waiting time for this schedule.
- 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 ms.
- 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.

- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of **context switches**.