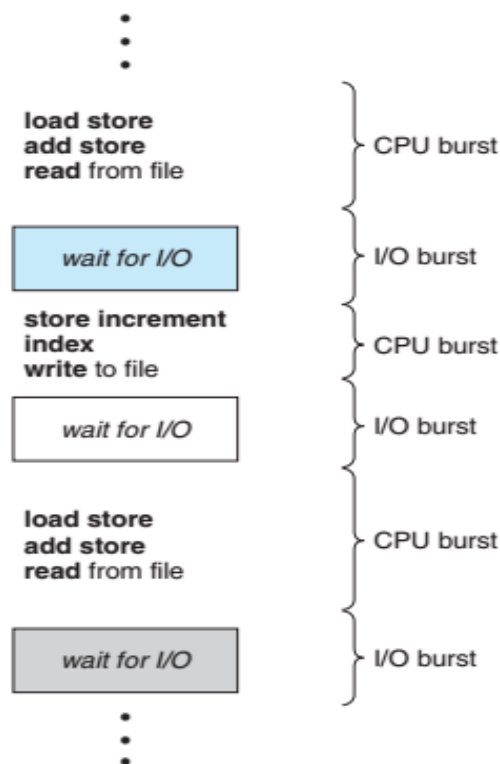


### CPU scheduling

- CPU scheduling is the basis of multiprogrammed 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; 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.
- 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.
- Scheduling of this kind is a fundamental operating-system function. 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.

### CPU-I/O Burst Cycle

process execution consists of a **cycle** of CPU execution and I/O 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, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.



Alternating sequence of CPU and I/O bursts.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm

### 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., a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. however, 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 one of the child processes)
  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 situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.
  - 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 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. This scheduling method was used by Microsoft Windows 3.x; Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
  - The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling.
  - Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

## 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:

- Switching context
- Switching to user mode
- 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

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

### ❖ CPU utilization:

We want to keep the CPU as busy as possible. Conceptually, 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).

### ❖ 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.

### ❖ Turnaround time:

From the point of view of 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/O.

### ❖ Waiting time:

The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. *Waiting time* is the sum of the periods spent waiting in the ready queue.

### ❖ 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, not the time it takes to output the response. The turnaround 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**.

**Scheduling Algorithms**

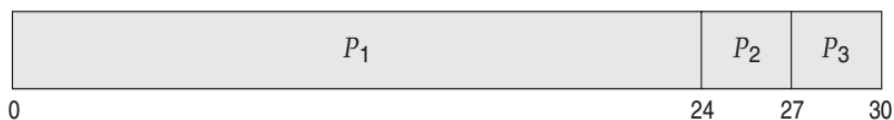
CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

**First-Come, First-Served Scheduling**

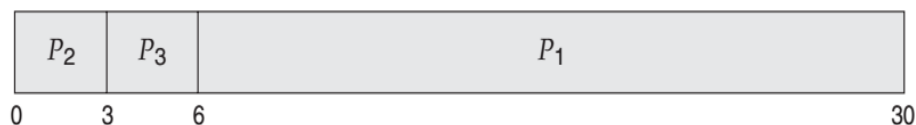
By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm**. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready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

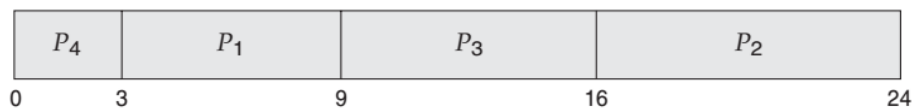
Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

**Shortest-Job-First Scheduling (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. Note that a 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. As an example of SJF scheduling, 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 10.25 milliseconds.

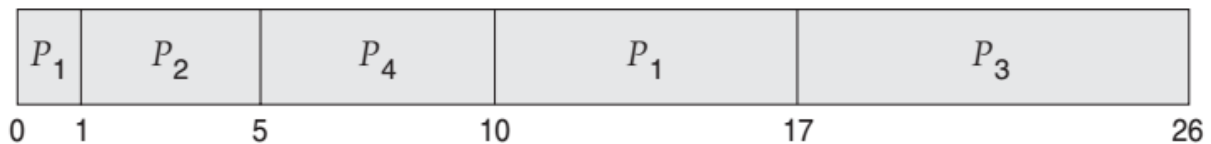
The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases.

The 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**.

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  $P_1$  is started at time 0, since it is the only process in the queue. Process  $P_2$  arrives at time 1. The remaining time for process  $P_1$  (7 milliseconds) is larger than the time required by process  $P_2$  (4 milliseconds), so process  $P_1$  is preempted, and process  $P_2$  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 7.75 milliseconds.