CPU Scheduling

- With multiprogramming, 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 .
- 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

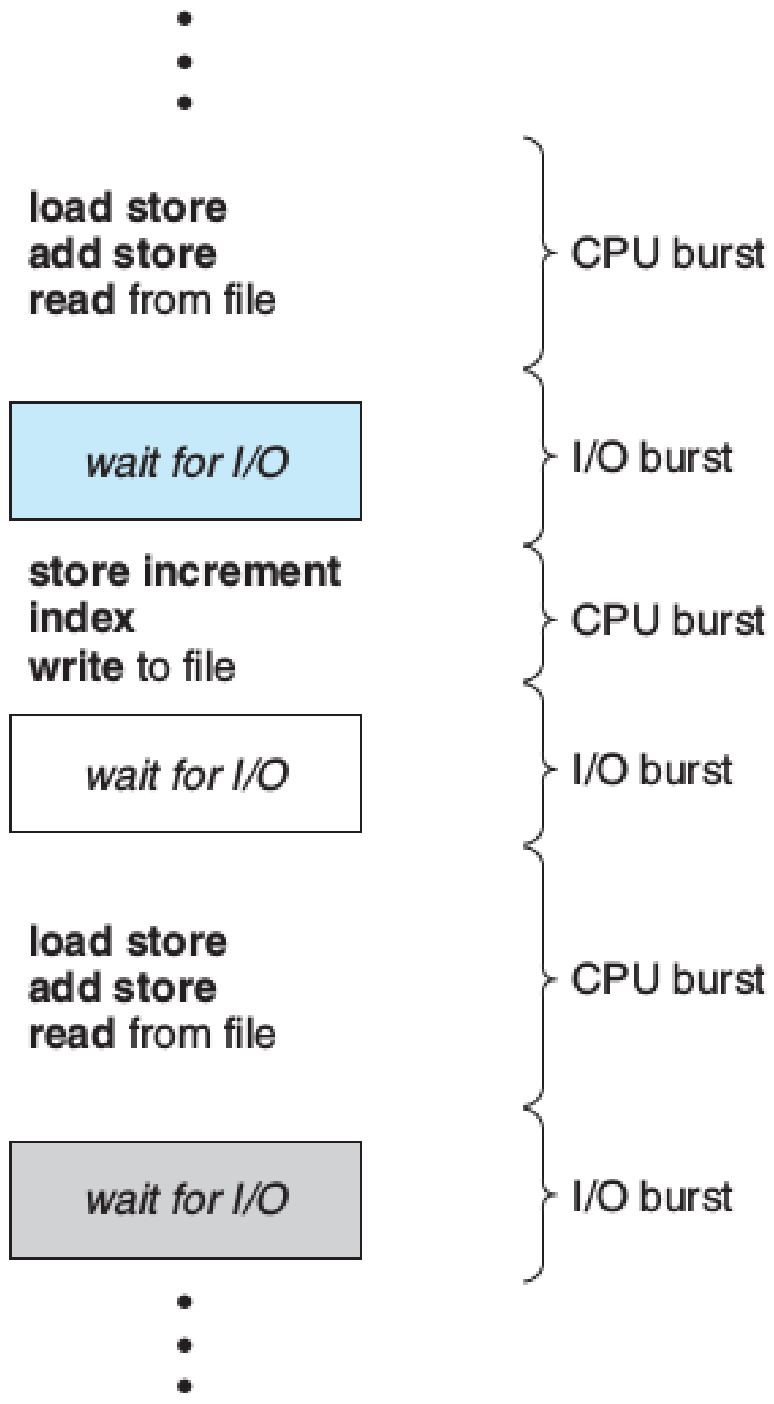


Figure 6.1 Alternating sequence of CPU and I/O bursts.

- 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.
- The records in the queues are generally process control blocks (PCB s) of the processes.

Pre-emptive 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

- 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.
- Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process 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.

- 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

- **CPU utilization**. We want to keep the CPU as busy as possible.
- **Throughput**. 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**. 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**. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, 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.

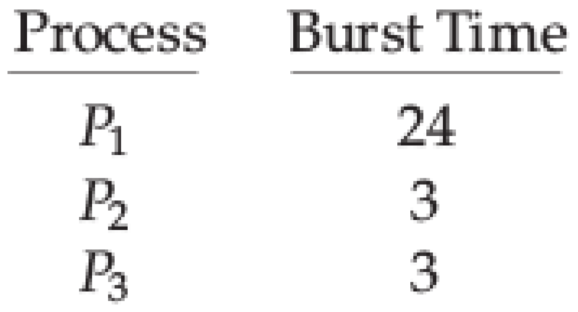
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling Algorithms

1. First-Come, First-Served Scheduling

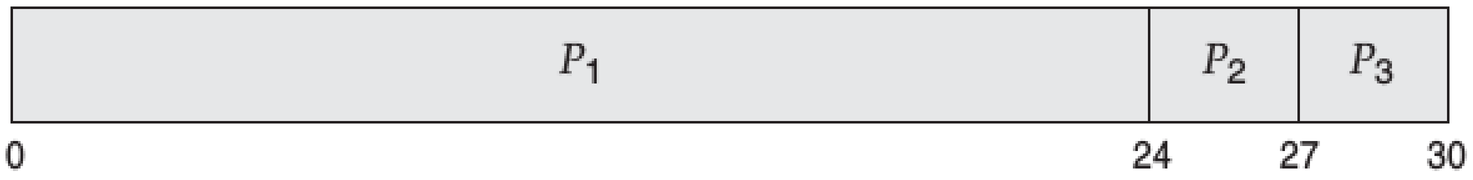
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

- The 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:

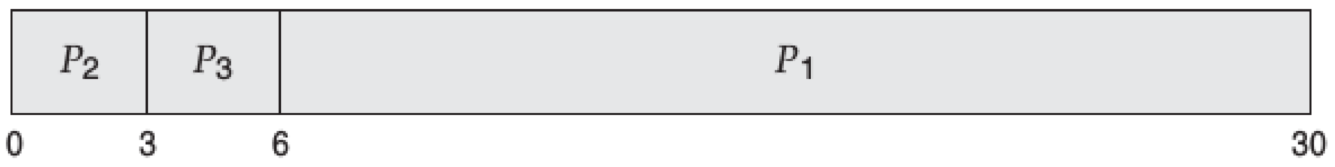


Process	Burst Time
P_1	24
P_2	3
P_3	3

- If the processes arrive in the order P1 , P2 , P3 , and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustr start and fi proce



- The waiting time is 0 milliseconds for process P1 , 24 milliseconds for process P2 , and 27 milliseconds for process P3 .
- Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2 , P3 , P1



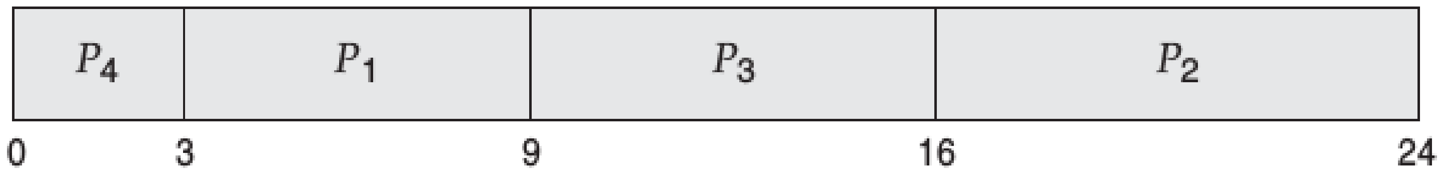
- The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.
- The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU , either by terminating or by requesting I/O .

- Assume we have one CPU -bound process and many I/O -bound processes. 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.
- 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.

2. Shortest-Job-First Scheduling

- 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.
- It is also called **shortest-next-CPU -burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

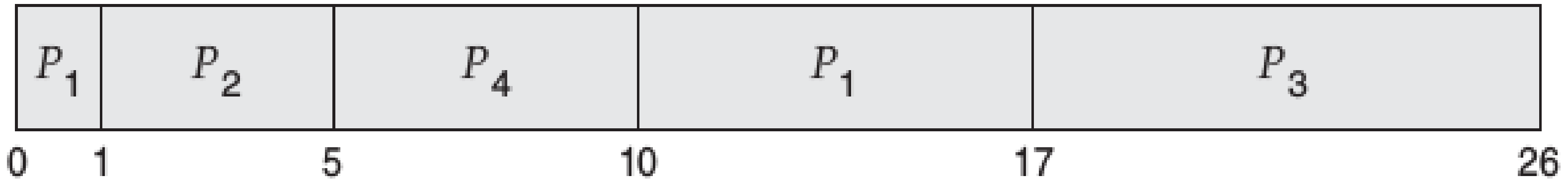
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- The waiting time is 3 milliseconds for process P1 , 16 milliseconds for process P2 , 9 milliseconds for process P3 , and 0 milliseconds for process P4 .
- 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.
- 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.

- The SJF algorithm can be either preemptive or nonpreemptive.
- 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.

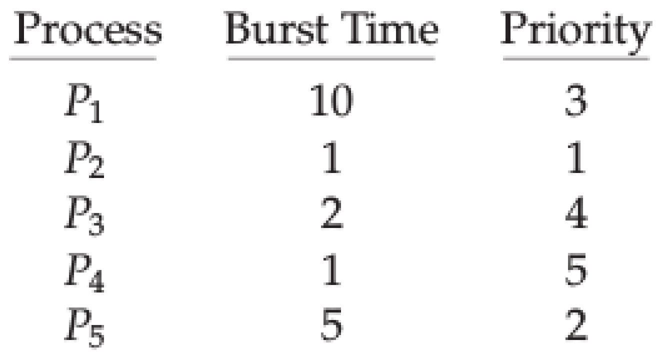
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



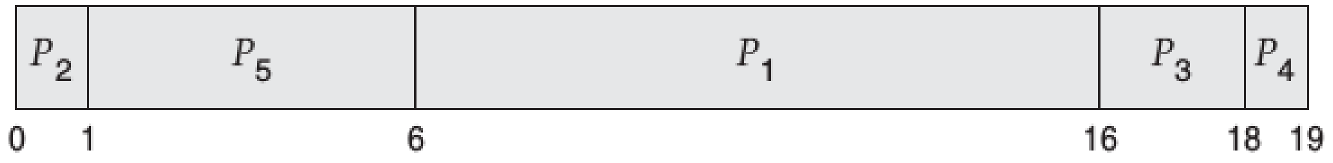
- Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- The average waiting 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 7.75 milliseconds.

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.
- For the following example, the average waiting time is 8.2 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



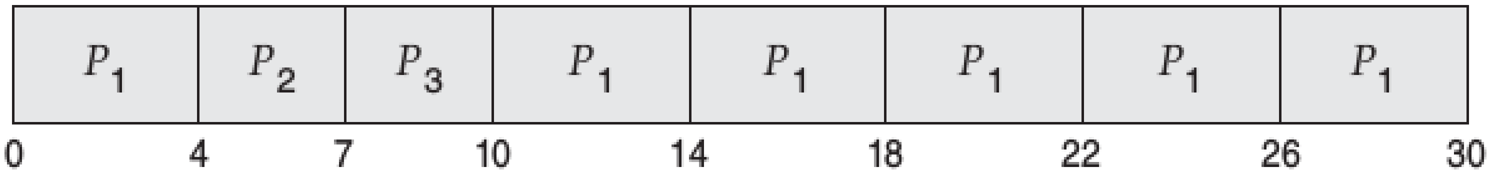
- Priorities can be defined either internally or externally.
- Internally defined priorities are time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
- External priorities are set by criteria outside the operating system, 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, often political, factors.
- 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 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.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

4. Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for time-sharing 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.
- The ready queue is treated as a circular queue.
- 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.
- 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.
- The average waiting time under the RR policy is often long.

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 . The RR scheduling algorithm is thus preemptive.
- 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.

- 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.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
- Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum.
- For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

- 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.
- 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, however, 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

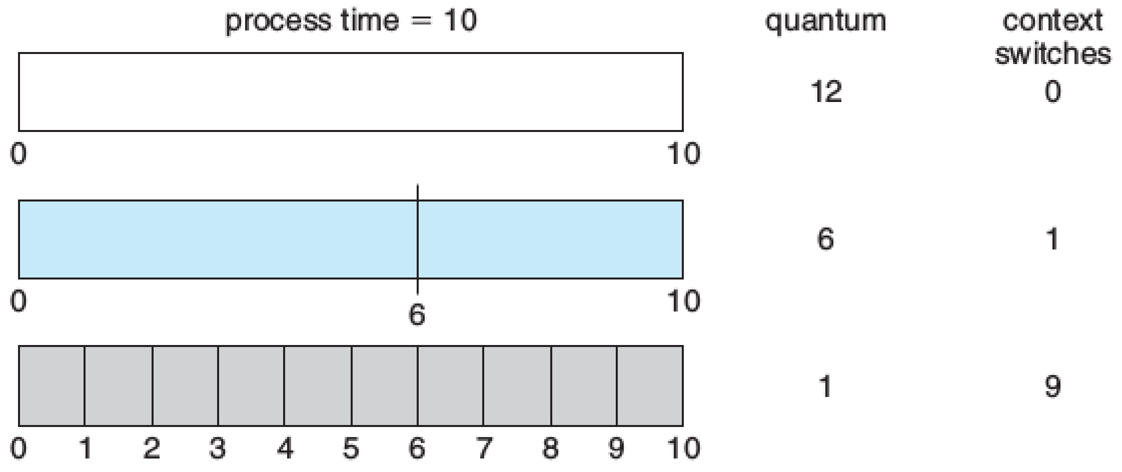


Figure 6.4 How a smaller time quantum increases context switches.