CPU scheduling

- 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 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

- The success of CPU scheduling depends on an observed property of processes:
- 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).
- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 6.2.
- The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU 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

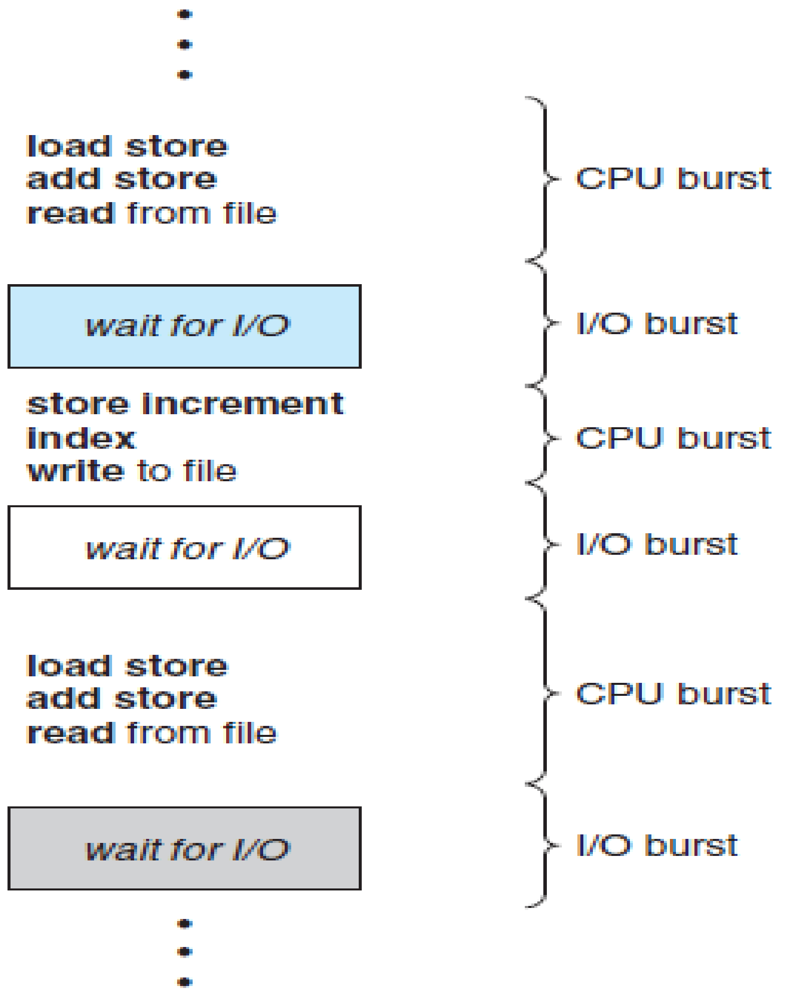


Figure 6.1 Alternating sequence of CPU and I/O bursts.

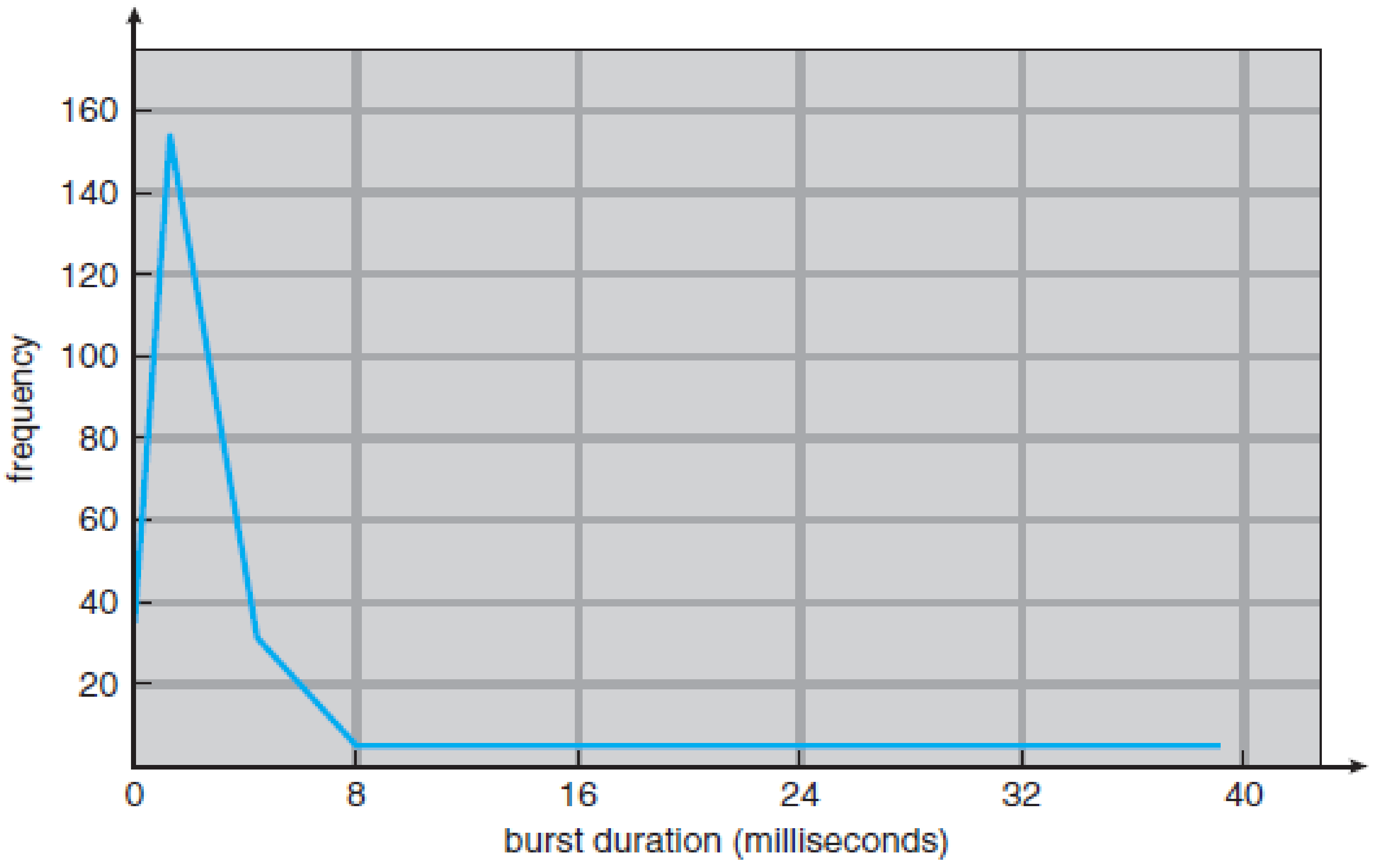


Figure 6.2 Histogram of CPU-burst durations.

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.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

- Under **nonpreemptive(cooperative) 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.
- 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.

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 loaded 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

Scheduling Criteria

- **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.
- In most cases, we optimize the average measure. However, under some circumstances, we prefer 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.

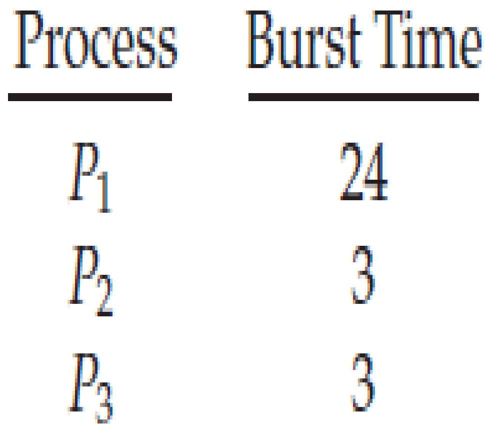
Scheduling Algorithms

- 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.

1.First-Come, First-Served Scheduling

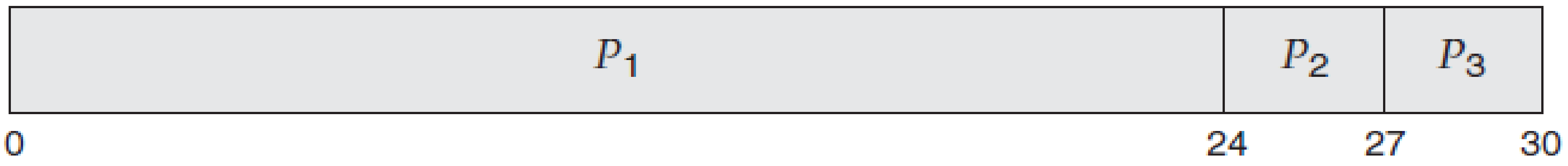
- the simplest CPU-scheduling algorithm is the first-come, first-served
- (FCFS) scheduling algorithm.
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready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

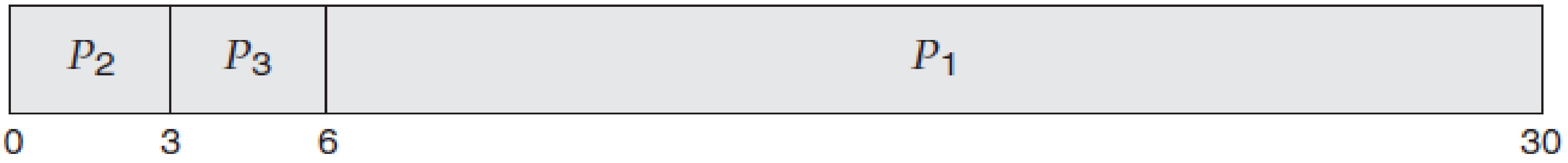


Process	Burst Time
P_1	24
P_2	3
P_3	3

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.

- 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. 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.
- 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.
- It would be disastrous to allow one process to keep the CPU for an extended period.

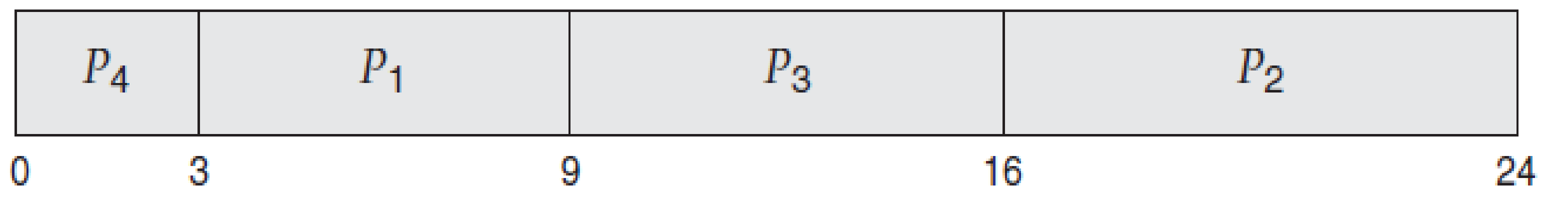
2.Shortest-Job-First Scheduling

- A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling** algorithm.
- 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.
- 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 real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- SJF scheduling is used frequently in long-term scheduling.
- 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.

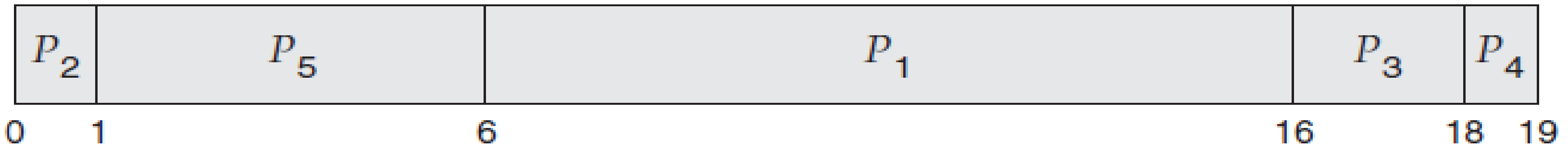
3.Priority Scheduling

- The SJF algorithm is a special case of the general **priority-scheduling algorithm.**
- 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.
- we discuss scheduling in terms of ***high priority and low priority.***
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, 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.
- **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 $P_1, P_2, \cdots, P_5$, 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.

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, 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 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 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.
- 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.
- In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process

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 from10 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.
- To implement RR scheduling, we again treat 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.

Round-Robin Scheduling

- 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.

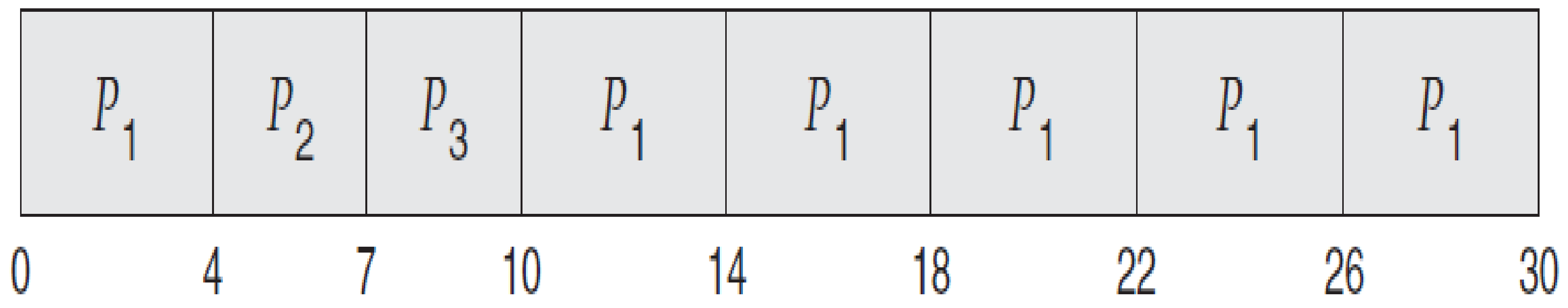
*The CPU scheduler will then select the next process in the ready queue.

- The average waiting time under the RR policy is often long.

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 P_1 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 P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds (10 - 4), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

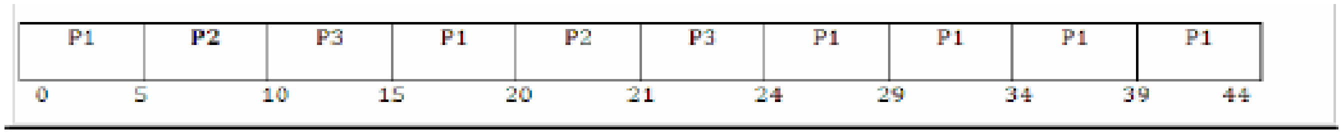
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

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.

- Important terms
- **Completion Time** It is the time at which any process completes its execution.
- **Turn Around Time** This mainly indicates the time Difference between completion time and arrival time. The Formula to calculate the same is:
- **Turn Around Time = Completion Time – Arrival Time**
- **Waiting Time(W.T):** It Indicates the time Difference between turn around time and burst time.
- And is calculated as
- **Waiting Time = Turn Around Time – Burst Time**
- If arrival time is not given for any problem statement then it is taken as 0 for all processes;
- if it is given then the problem can be solved accordingly.

example2

PROCESS	BURST TIME
P1	30
P2	6
P3	8



AVERAGE WAITING TIME :

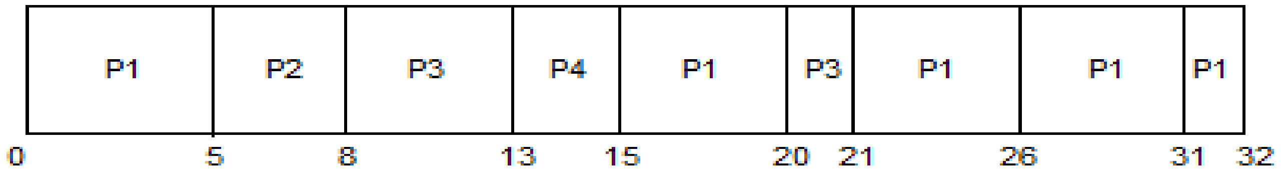
Waiting time for P1 => 0+(15-5)+(24-20) => 0+10+4 = 14
Waiting time for P2 => 5+(20-10) => 5+10 = 15
Waiting time for P3 => 10+(21-15) => 10+6 = 16
Average waiting time => (14+15+16)/3 = 15 ms.

AVERAGE TURN AROUND TIME :
FORMULA : Turn around time = waiting time + burst Time
Turn around time for P1 => 14+30 =44
Turn around time for P2 => 15+6 = 21
Turn around time for P3 => 16+8 = 24
Average turn around time => (44+21+24)/3 = 29.66 ms

example3

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

- The performance of the RR algorithm depends heavily on the size of the time quantum.
- At one extreme, 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 can result in a large number of context switches.
- for example, if 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).

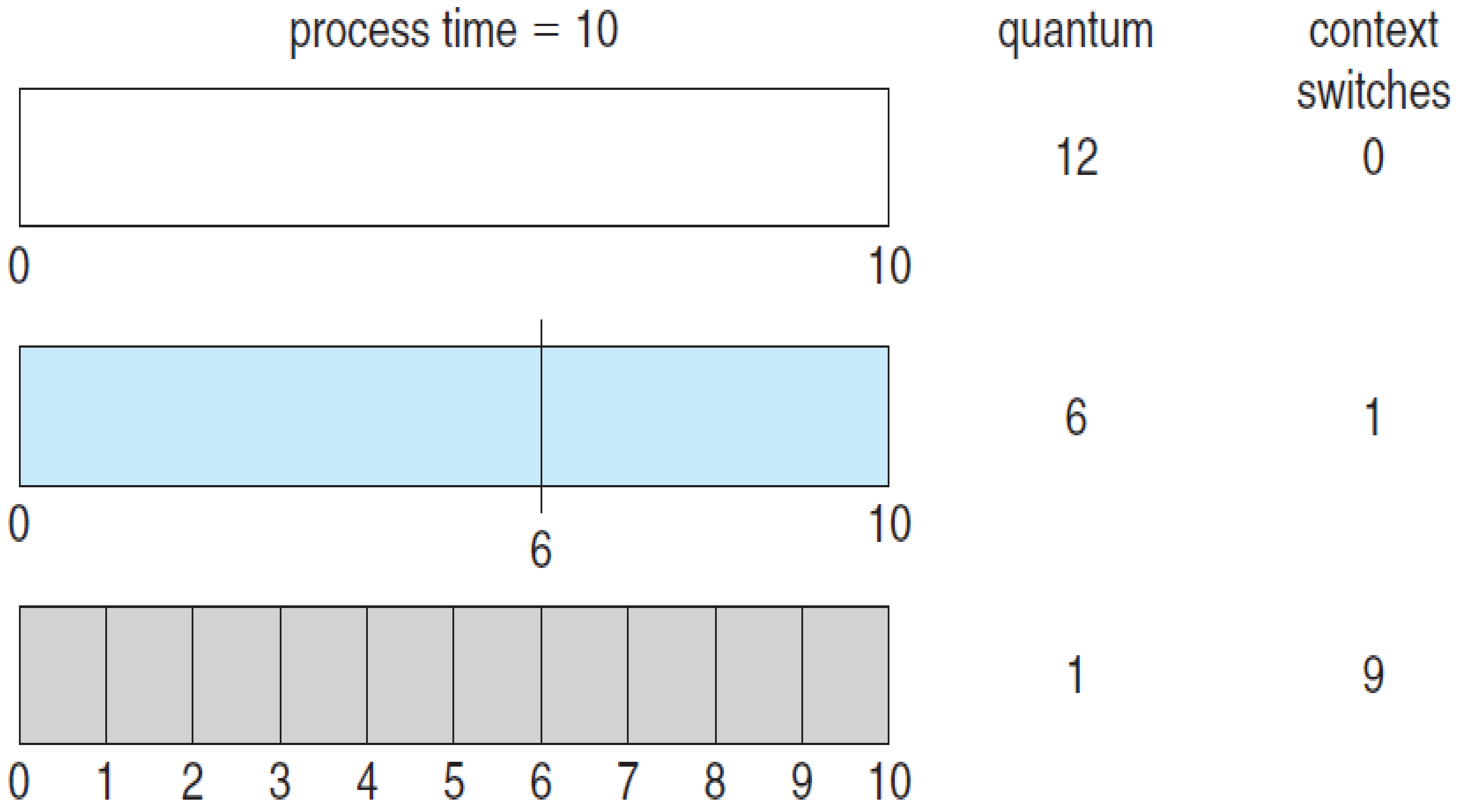


Figure 6.4 How a smaller time quantum increases context switches.

Multiple-Processor Scheduling

- If multiple CPUs are available, **load sharing** becomes possible—but scheduling problems become correspondingly more complex.
- Dealing with systems in which the processors are identical—homogeneous —in terms of their functionality.
- We can then use any available processor to run any process in the queue.
- even with homogeneous multiprocessors, there are sometimes limitations on scheduling.
- Consider a system with an I/O device attached to a private bus of one processor.
- Processes that wish to use that device must be scheduled to run on that processor
- **Approaches to Multiple-Processor Scheduling**
- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server.

- The other processors execute only user code.
- This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing
- A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling.
- All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- Regardless scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute
- if we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
- We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.
- Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.