



# Chapter 4. CPU Scheduling

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne.  
(2018). *Operating System Concepts* (10th ed.). Wiley.



# Contents

- 1. Basic Concepts
- 2. Scheduling Criteria
- 3. Scheduling Algorithms
- 4. Thread Scheduling



# 1. Basic Concepts

- In a system with a single CPU core:
  - Only one process can run at a time. Others must wait until the CPU core is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times → maximize CPU utilization.
  - When one process has to wait, OS takes the CPU away from that process and gives the CPU to another process.
    - On a multicore system, this concept of keeping the CPU busy is extended to all cores on the system.
  - CPU scheduling of this kind is a fundamental operating-system function.
    - CPU scheduling is central to operating-system design.

# 1. Basic Concepts

## 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:
    - CPU burst
    - I/O burst

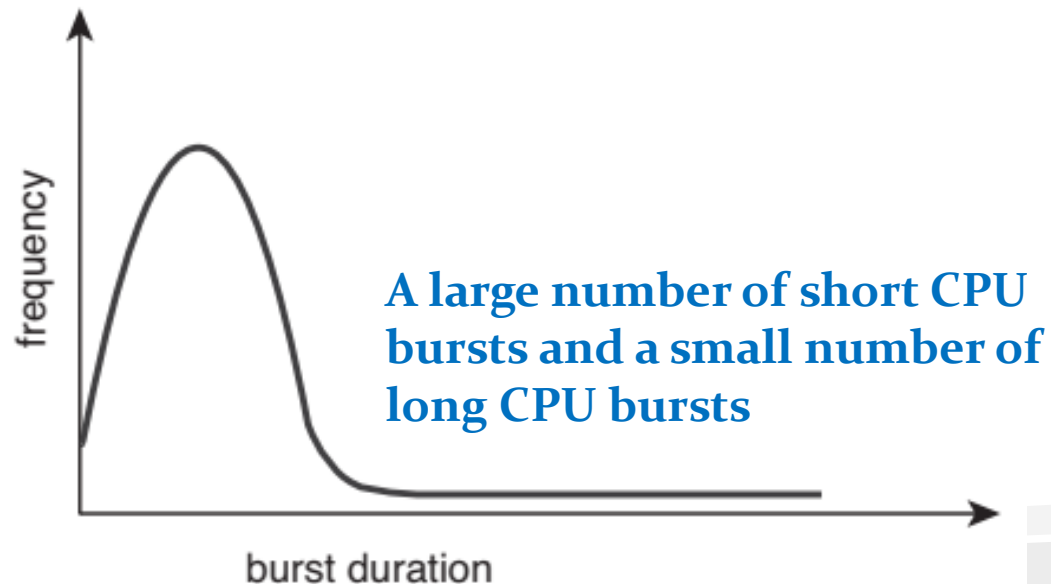


Figure 5.2 Histogram of CPU-burst durations.

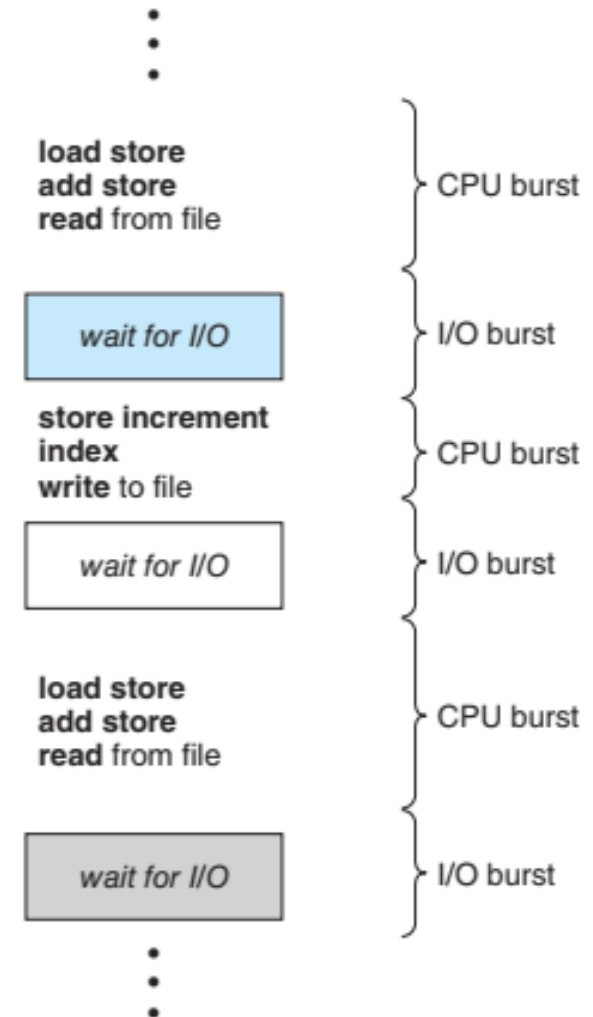


Figure 5.1 Alternating sequence of CPU and I/O bursts.



# 1. Basic Concepts

## CPU Scheduler

- CPU scheduler
  - Whenever the CPU becomes idle, CPU scheduler selects one of processes in the ready queue to be executed (allocates the CPU to that process).
- Conceptually, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
  - A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

# 1. Basic Concepts

## Preemptive and Nonpreemptive Scheduling

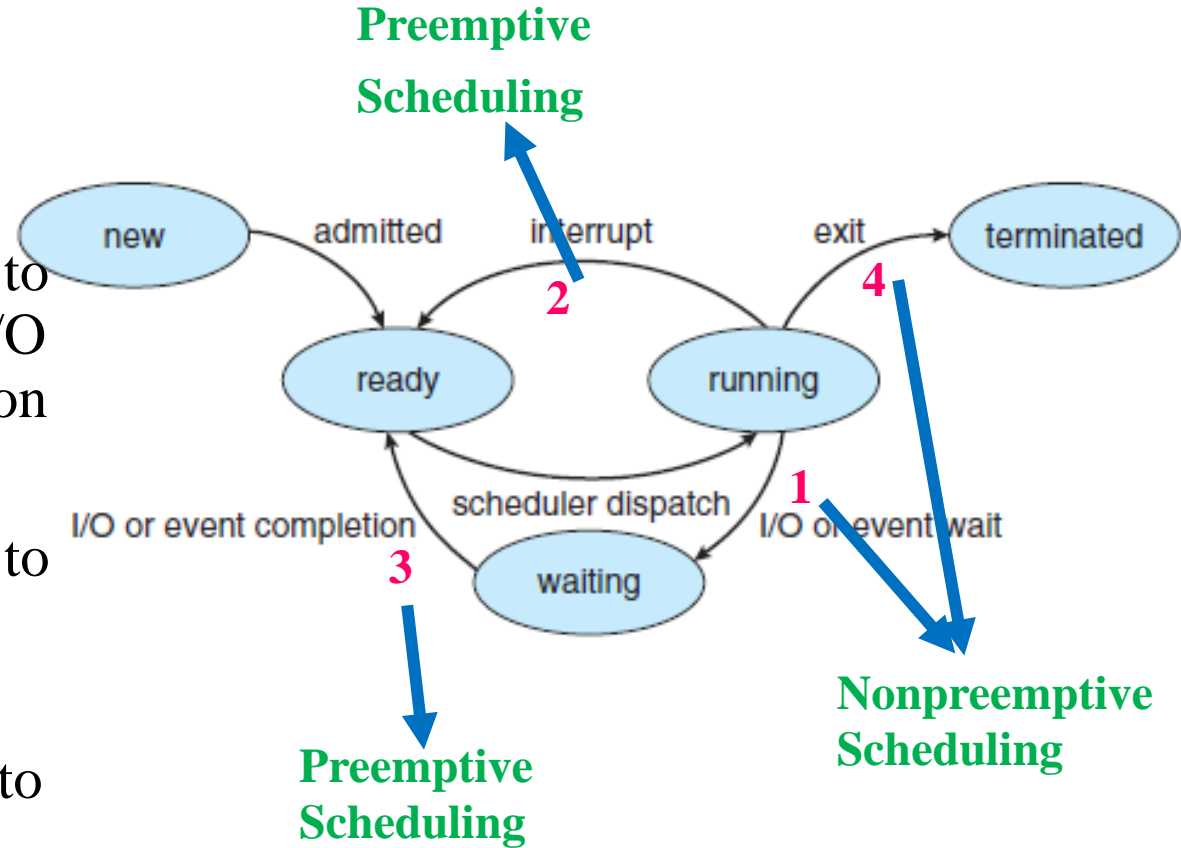
- CPU-scheduling decisions may take place under the following four circumstances:

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, then the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive** ( 2 and 3)



# 1. Basic Concepts

## Preemptive and Nonpreemptive Scheduling

- Nonpreemptive scheduling
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
  - A nonpreemptive kernel will wait for a system call to complete or for a process to block while waiting for I/O to complete 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, where tasks must complete execution within a given time frame.



# 1. Basic Concepts

## Preemptive and Nonpreemptive Scheduling

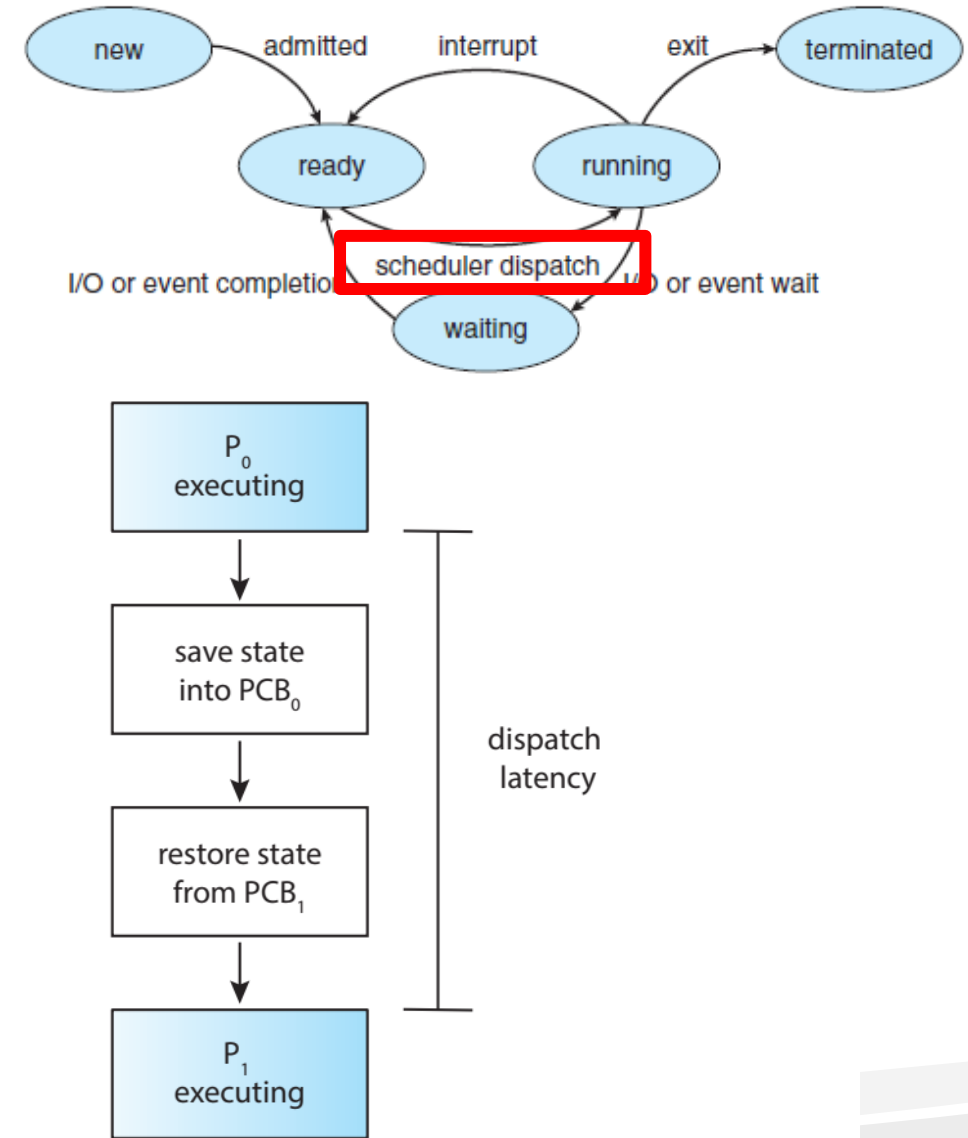
- Preemptive scheduling
  - All modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.
  - Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.
    - Consider the case of two processes that share 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.
  - During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O 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? Chaos ensues.



# 1. Basic Concepts

## Dispatcher

- The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This involves:
  - Switching context from one process to another
  - Switching to user mode
  - Jumping to the proper location in the user program to resume that program
- Dispatch latency: The time it takes for the dispatcher to stop one process and start another running.



**Figure 5.3** The role of the dispatcher.



# 1. Basic Concepts

## Dispatcher

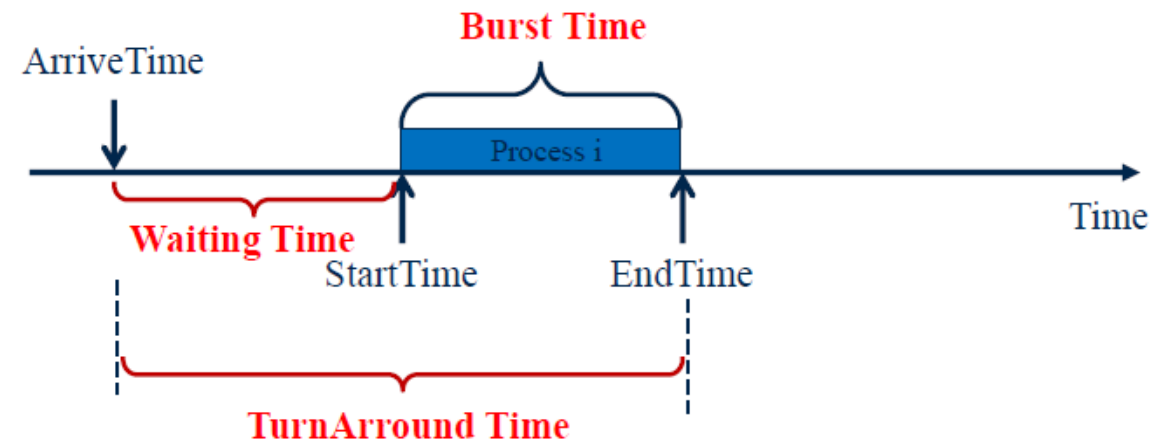
- We can use the **/proc** file system to determine the number of context switches for a given process.
- For example, the contents of the file **/proc/2166/** status will list various statistics for the process with pid = 2166.
- The command **\$cat /proc/2166/status** provides the following trimmed output:
  - **voluntary\_ctxt\_switches 150**
  - **nonvoluntary\_ctxt\_switches 8**
- A **voluntary context switch** occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.)
- A **nonvoluntary context switch** occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

## 2. Scheduling Criteria

- **CPU utilization** – Keep the CPU as busy as possible
- **Throughput** – Number of processes that complete their execution per time unit
- **Turnaround time** – Amount of time to execute a particular process
  - The sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O
- **Waiting time** – Amount of time a process has been waiting in the ready queue
- **Response time** – Amount of time it takes from when a request was submitted until the first response is produced

### Scheduling Metrics

- **Waiting Time** = Start Time – Arrive Time
- **Turnaround Time** = End Time – Arrive Time
- (or **Turnaround Time** = Waiting time + Burst time)

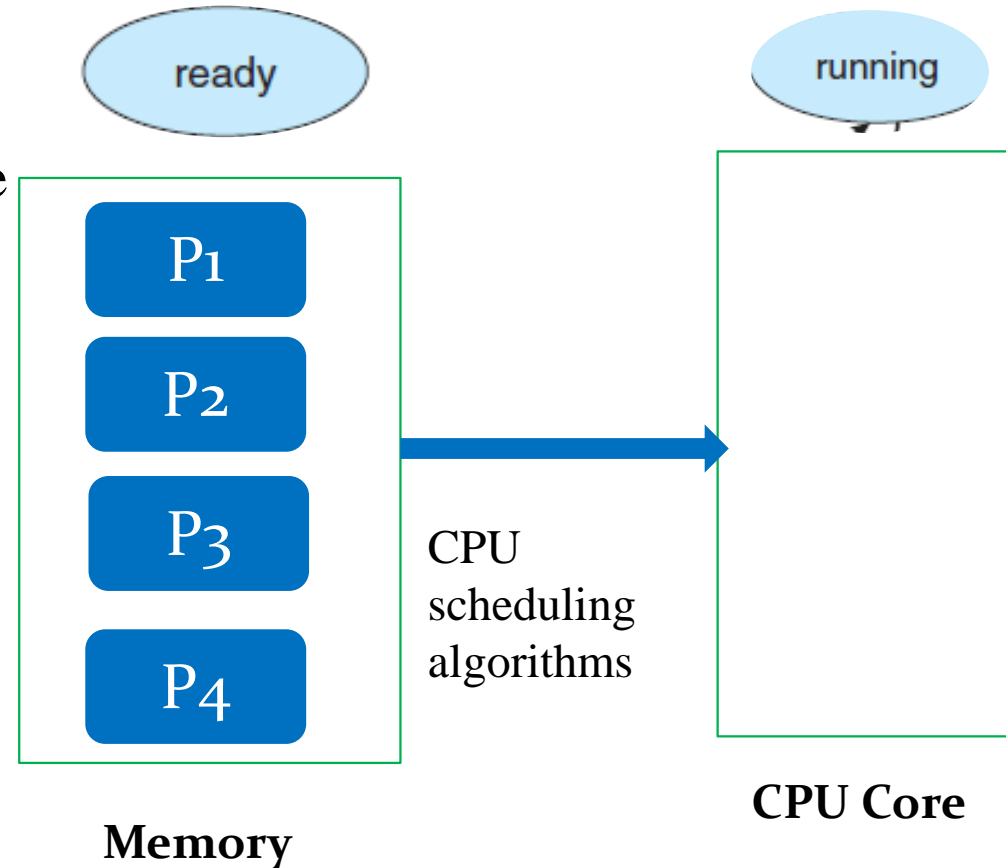


### Optimization Criteria :

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

### 3. Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core.
- There are many different CPU scheduling algorithms. These scheduling algorithms are discussed here in the context of only **one processing core** available.
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - Round-Robin Scheduling
  - Priority Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling





### 3. Scheduling Algorithms

## First-Come, First-Served (FCFS) Scheduling

- The simplest CPU-scheduling algorithm
- 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.

### 3. Scheduling Algorithms

#### First-Come, First-Served (FCFS) Scheduling

- Suppose that the processes arrive at time 0, in the order:  $P_1$ ,  $P_2$ ,  $P_3$ , with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The **Gantt Chart** for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

### 3. Scheduling Algorithms

## First-Come, First-Served (FCFS) Scheduling

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:

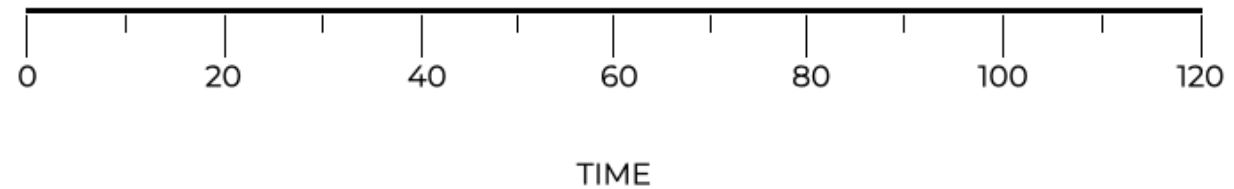


- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process

### 3. Scheduling Algorithms

## First-Come, First-Served (FCFS) Scheduling

- Key points
  - The job that arrives first is scheduled first.
  - A nonpreemptive algorithm.
  - Implemented using queue
  - Suffers from convoy effect.

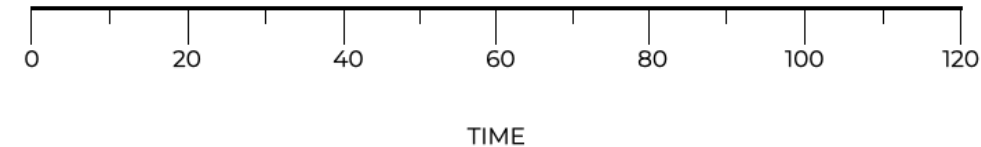




### 3. Scheduling Algorithms

## Shortest-Job-First (SJF) 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.
- This algorithm associates with each process the length of the process's next CPU burst.
  - 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.
- SJF is optimal, gives minimum average waiting time for a given set of processes.

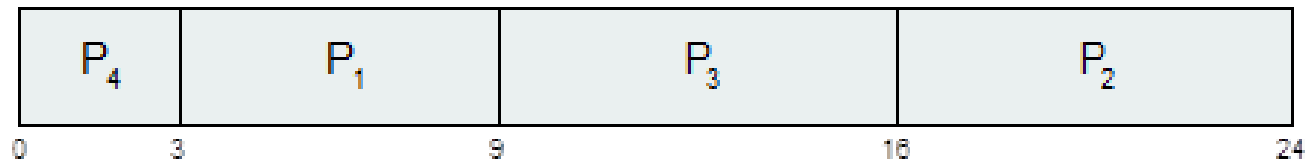


### 3. Scheduling Algorithms

## Shortest-Job-First (SJF) Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$



## 3. Scheduling Algorithms

### Shortest-Job-First (SJF) Scheduling

- Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst.
- Prediction of the length of the next CPU burst
  - We expect that the next CPU burst will be similar in length to the previous ones.
  - The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts.
  - Let  $t_n$  be the length of the  $n$ th CPU burst, and let  $\tau_{n+1}$  be our predicted value for the next CPU burst. Then, for  $\alpha$ ,  $0 \leq \alpha \leq 1$ , define

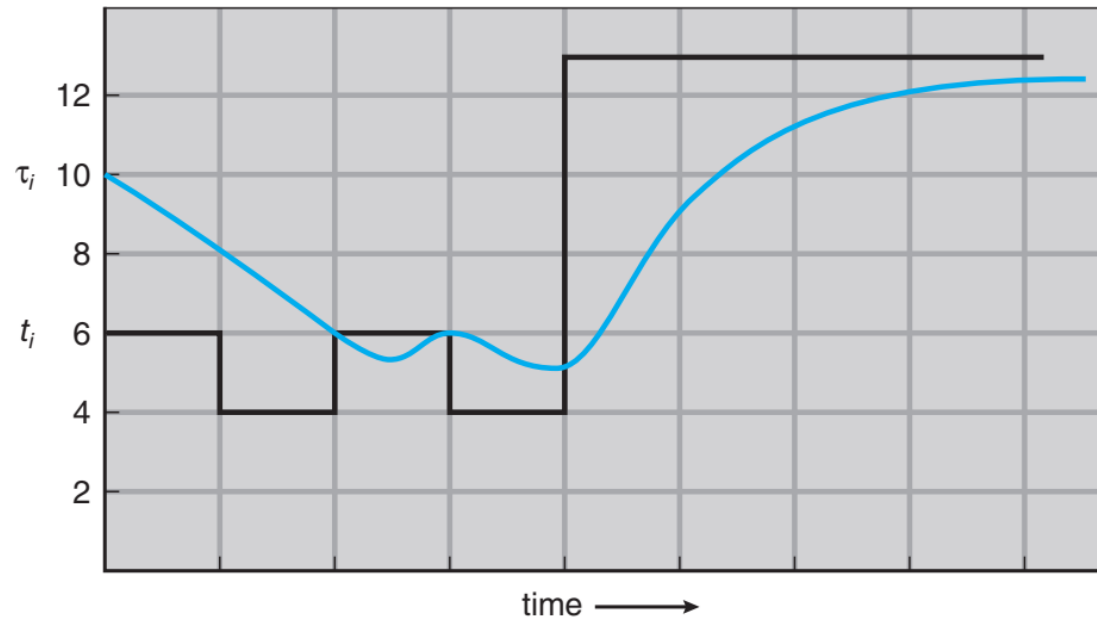
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

- More commonly,  $\alpha = 1/2$ , so recent history and past history are equally weighted.
- The initial  $\tau_0$  can be defined as a constant or as an overall system average.

### 3. Scheduling Algorithms

## Shortest-Job-First (SJF) Scheduling

- The following figure shows an exponential average with  $\alpha = 1/2$  and  $\tau_0 = 10$ .



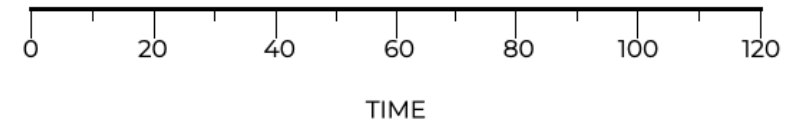
CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...	
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...

**Figure 5.4** Prediction of the length of the next CPU burst.

# 3. Scheduling Algorithms

## Shortest-Job-First (SJF) Scheduling

- 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.
    - Preemptive SJF scheduling is sometimes called **shortest-remaining time-first** scheduling.
  - A nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.



**Preemptive SJF Algorithm**

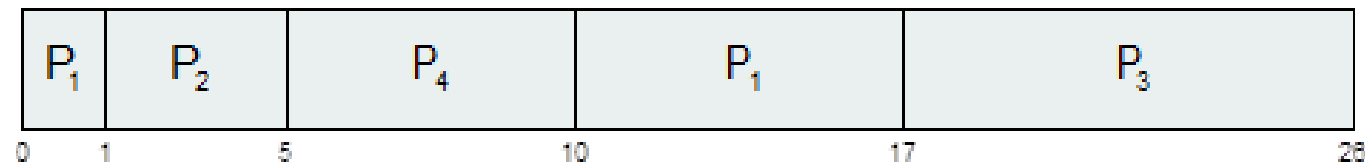
### 3. Scheduling Algorithms

## Shortest-Job-First (SJF) Scheduling

- Preemptive SJF algorithm
  - Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- *Preemptive SJF Gantt Chart*

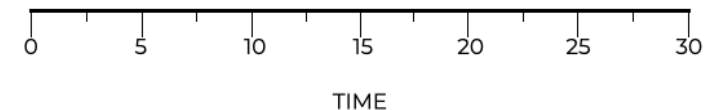


- Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$

### 3. Scheduling Algorithms

## Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm 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.
  - 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**.



Time slice: 5



# 3. Scheduling Algorithms

## Round-Robin Scheduling

- Implementing RR scheduling:
  - The ready queue: 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.
- 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.
- Performance
  - $q$  large  $\rightarrow$  FCFS
  - $q$  small  $\rightarrow$  RR
- Note that  $q$  must be large with respect to context switch, otherwise overhead is too high.



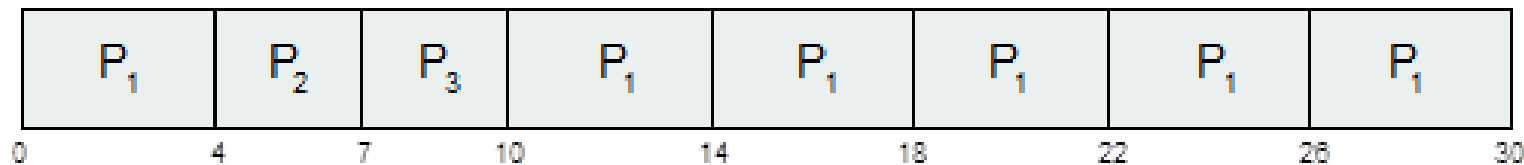
### 3. Scheduling Algorithms

## Round-Robin Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Time quantum of 4 milliseconds
- All processes arrive at time 0, in the order of  $P_1, P_2, P_3$

- The Gantt chart is:

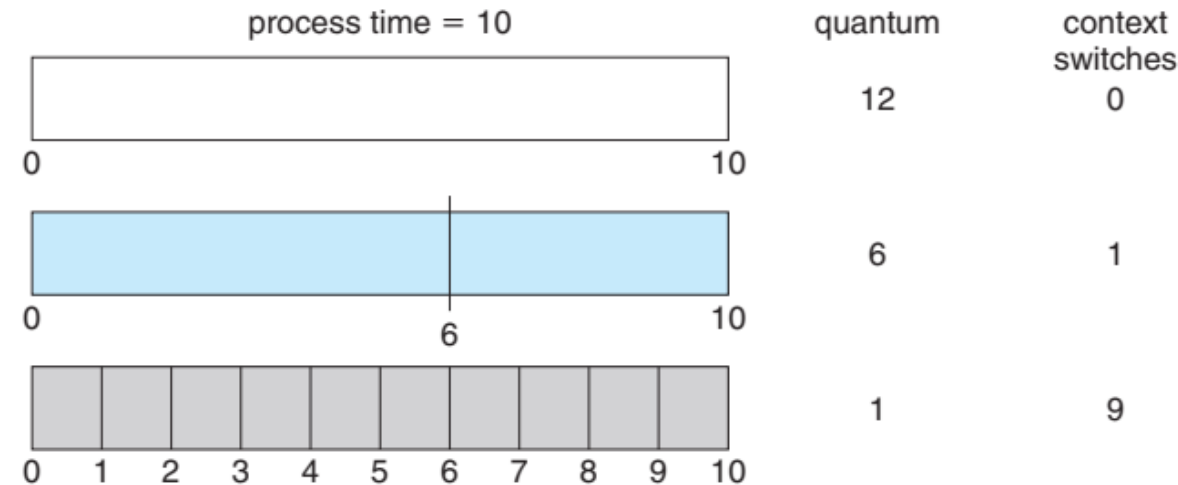


- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
  - $q$  usually 10 milliseconds to 100 milliseconds,
  - Context switch  $< 10$  microseconds

### 3. Scheduling Algorithms

## Time Quantum and Context Switch Time

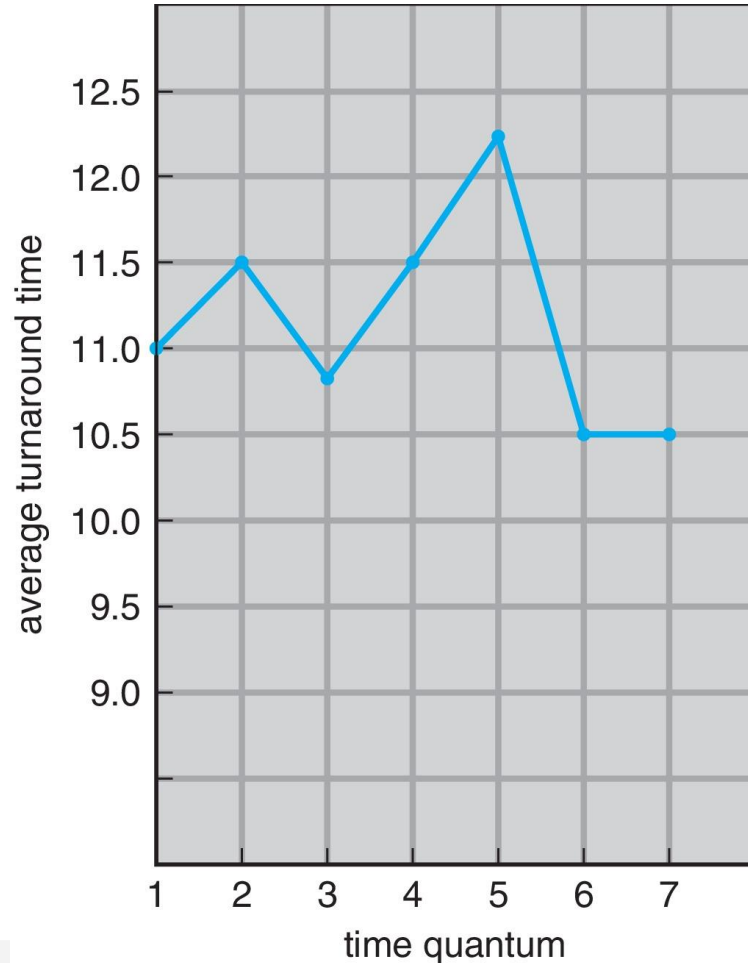
- For example, we have only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly
- $q$  should be large compared to context switch time
  - $q$  is usually from 10ms to 100ms, context switch  $< 10 \mu s$



**Figure 5.5** How a smaller time quantum increases context switches.

### 3. Scheduling Algorithms

## Turnaround Time Varies with the Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

- Turnaround time also depends on the size of the time quantum.
- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
- 80% of CPU bursts should be shorter than  $q$



## 3. Scheduling Algorithms

### Priority Scheduling

- A priority is associated with each process.
  - Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
  - There is no general agreement on whether 0 is the highest or lowest priority. In this text, we assume that low numbers represent high priority.
- 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.

### 3. Scheduling Algorithms

## Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

➤ All processes arrive at time 0.

#### ■ Priority scheduling Gantt Chart



#### ■ Average waiting time = 8.2



## 3. Scheduling Algorithms

### Priority Scheduling

- Priority Scheduling can be either preemptive or nonpreemptive.
  - 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.
  - Nonpreemptive: The 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.

### 3. Scheduling Algorithms


## Priority Scheduling with Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin.

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

Gantt Chart with time quantum = 2





### 3. Scheduling Algorithms

## Multilevel Queue Scheduling

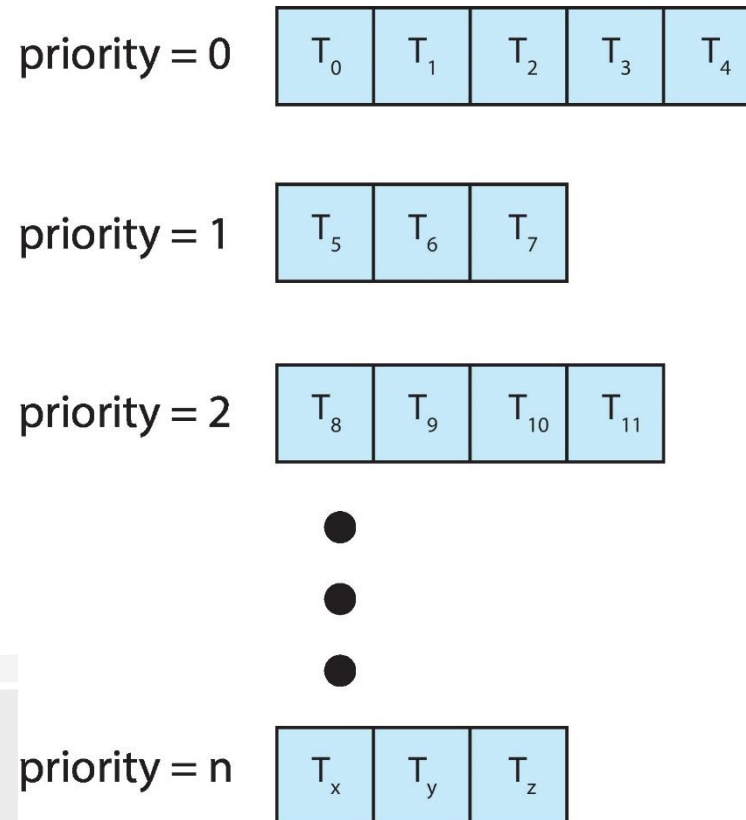
- The ready queue consists of multiple queues.
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues



### 3. Scheduling Algorithms

## Multilevel Queue Scheduling – Priority Based

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue

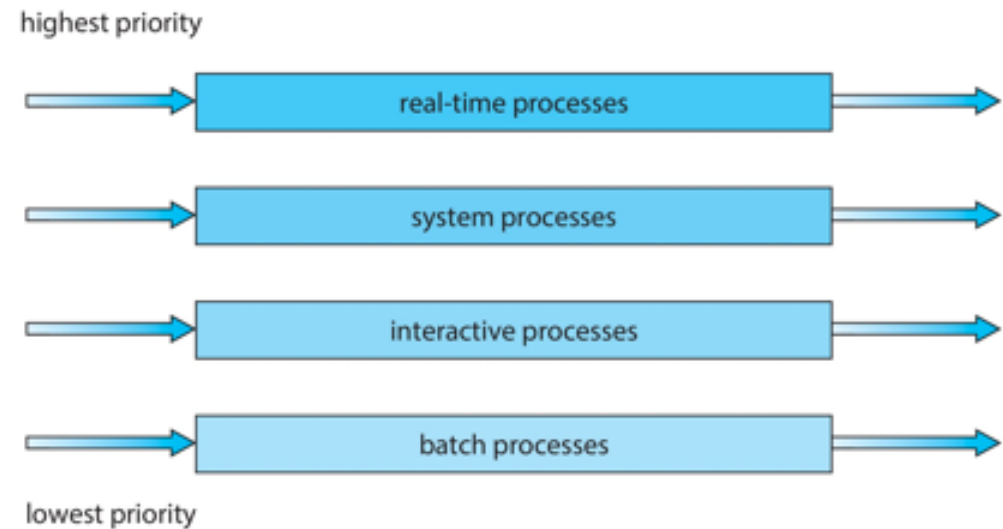


### 3. Scheduling Algorithms

## Multilevel Queue Scheduling – Process Type Based

- Prioritization based upon process type
- For example, a common division is made between
  - Foreground Process (Interactive process)
  - Background Process (Batch process)
- The foreground queue might be scheduled by an RR algorithm
- The background queue is scheduled by an FCFS algorithm.
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation
  - Time slice – each queue gets a certain amount of CPU time which it can schedule among its processes; i.e., 80% to foreground in RR and 20% to background in FCFS

Prioritization based upon process type





# 3. Scheduling Algorithms

## Multilevel Feedback Queue Scheduling

- Multilevel Feedback Queue allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts.
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - Leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue.

# 3. Scheduling Algorithms

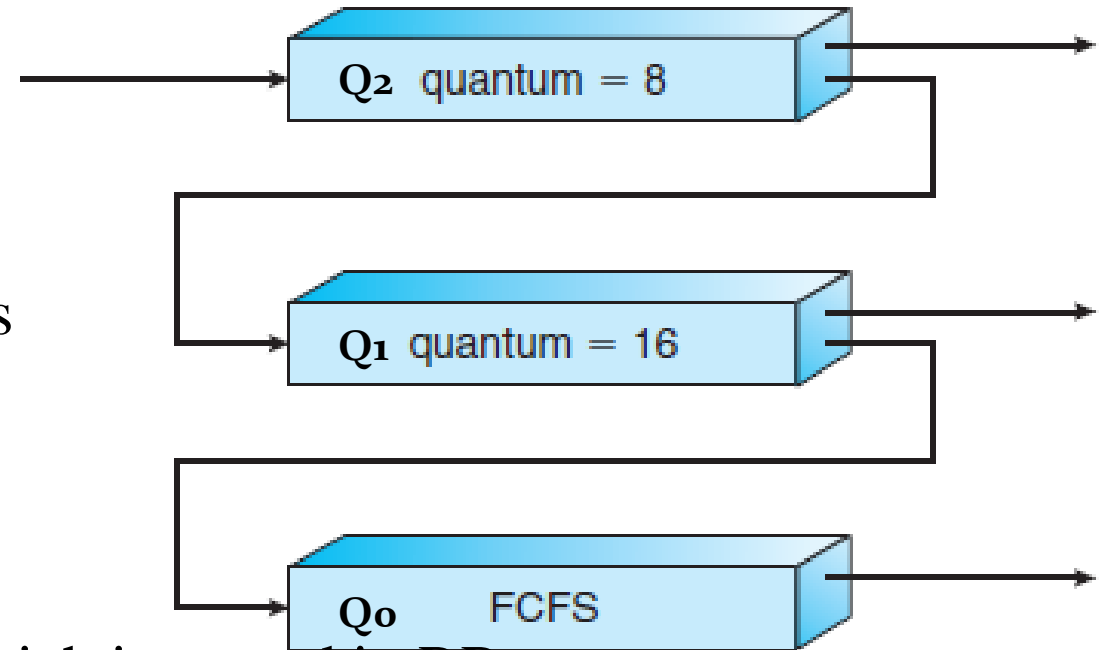
## Multilevel Feedback Queue Scheduling

- Three queues:

- $Q_2$  – RR with time quantum 8 ms
- $Q_1$  – RR with time quantum 16 ms
- $Q_0$  – FCFS

- Scheduling

- A new process enters queue  $Q_2$  which is served in RR
  - When it gains CPU, the process receives 8 milliseconds
  - If it does not finish in 8 milliseconds, the process is moved to queue  $Q_1$
- At  $Q_1$  job is again served in RR and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue  $Q_0$





## 4. Thread Scheduling

- On most modern operating systems it is kernel-level threads—not processes—that are being scheduled by the operating system.
- User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread
  - This mapping may be indirect and may use a lightweight process (LWP).
- One distinction between user-level and kernel-level threads lies in how they are scheduled.
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP.
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU **is system-contention scope (SCS)** – competition among all threads in system
  - Systems using the one-to-one model, such as Windows and Linux, schedule threads using only SCS.



## 4. Thread Scheduling

### Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
  - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow `PTHREAD_SCOPE_SYSTEM`



## 4. Thread Scheduling

### Pthread Scheduling

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```



## 4. Thread Scheduling

### Pthread Scheduling

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```