Chapter 5: CPU Scheduling

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

A process is executed until it must wait, typically for the completion of some I/O request.

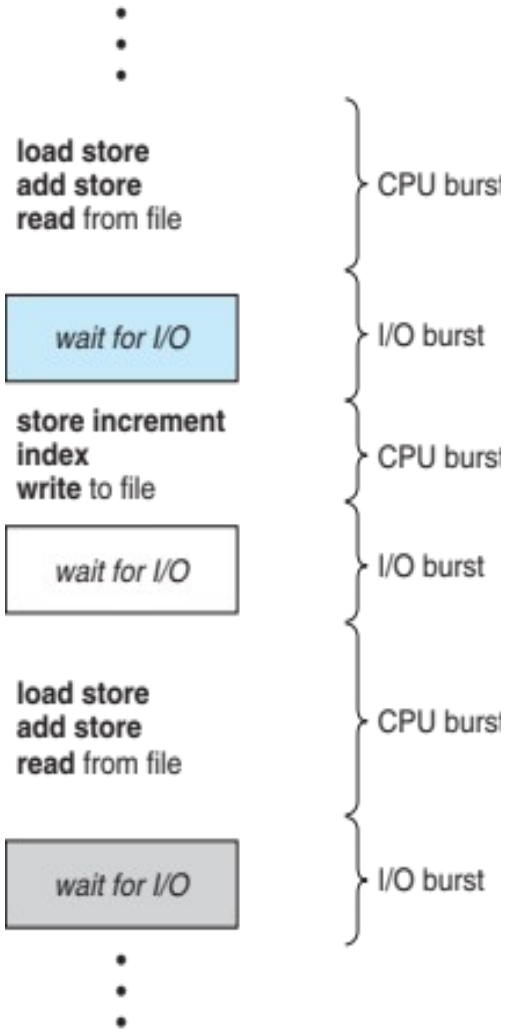
- In a system with a single CPU core, only one process can run at a time.
  - Others must wait until the CPU's core is free and can be reschedule
  - The CPU then just sits idle.
  - All this waiting time is wasted; no useful work is accomplished.

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
  - Several processes are kept in memory at one time.
  - When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process.

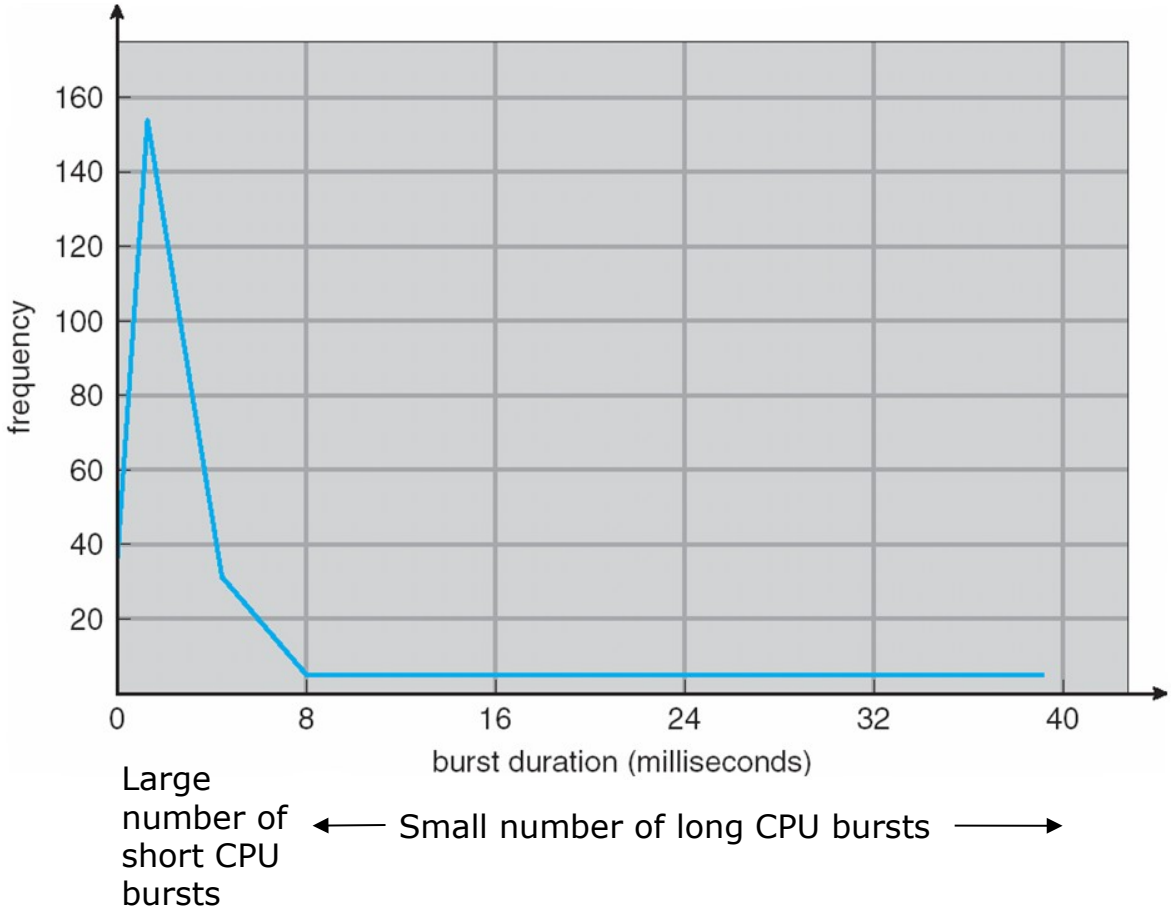
- Every time one process has to wait, another process can take over use of the CPU.
- On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

CPU–I/O Burst Cycle

- Maximum CPU utilization obtained with multiprogramming
- 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**.
- 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.



Histogram of CPU-burst Times



- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound programming might have a few long CPU bursts.
- This distribution can be important when implementing a CPU-scheduling algorithm.

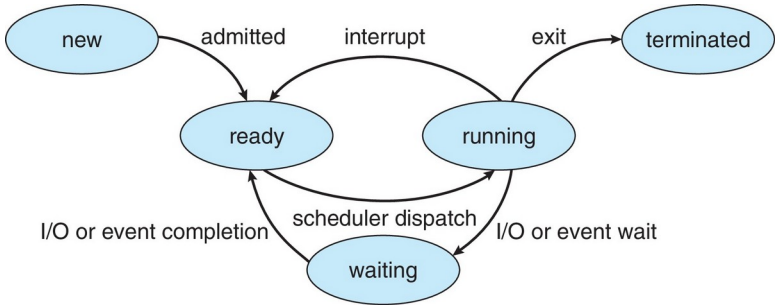
CPU Scheduler (short-term scheduler)

- 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.

- The **CPU scheduler** selects from among the processes in **ready queue**, and **allocates the a CPU** core to one of them
  - Queue may be ordered in various ways

Preemptive and Nonpreemptive Scheduling

- CPU **scheduling decisions may take place when a process**:
  1. Switches from running state to 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. Switches from running state to ready state
     - for example, when an interrupt occurs
  3. Switches from waiting to ready
     - for example, at completion of I/O
  4. Terminates



- Scheduling under 1 and 4 is **nonpreemptive. No choice in scheduling.**
- **Preemptive. Choice** for 2 and 3.
  - Access to shared data can result in race conditions
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Nonpreemptive Scheduling

Nonpreemptive scheduling

(Also known as **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 by requesting I/O.
  - 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.
- 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.
  - Nonpreemptive kernel-execution model is a poor one for supporting real-time computing, where tasks must complete execution within a given time frame.

Preemptive scheduling

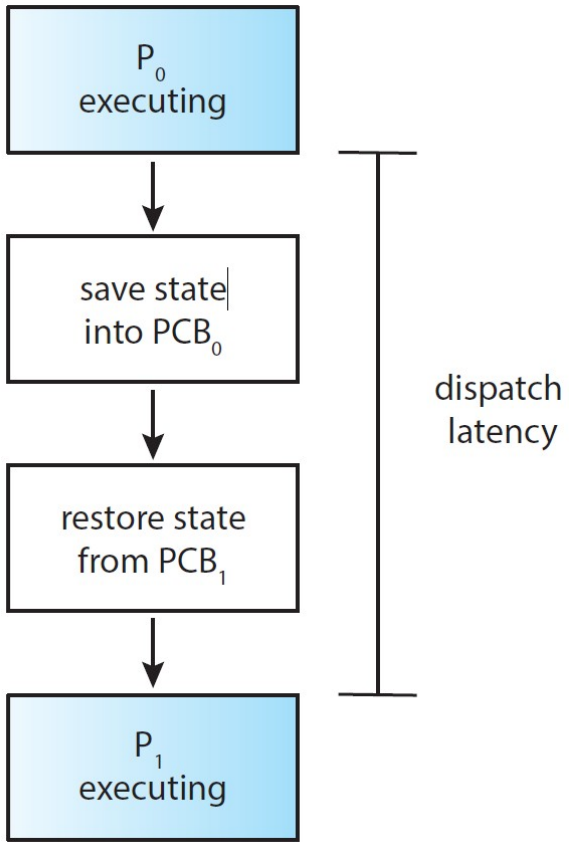
Preemptive scheduling:

- In computing, **preemption** is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time.
- Such changes of the executed task are known as **context switches**.
- It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the **power to preempt, or interrupt,** and **later resume,** other tasks in the system.
  - Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.

Virtually all modern OS including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

Different CPU-scheduling algorithms have different properties. The criteria for comparing CPU-scheduling algorithms include the following:

- **CPU utilization** – keep the CPU as busy as possible. Range from 0**%** to 100**%.**
- **Throughput** – **#** of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **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, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria - Max

Maximize:

- Maximize CPU utilization
  - We want to keep the CPU as busy as possible.
  - In a real system, CPU utilization should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

    The command to find CPU utilization in Linux, macOS and UNIX systems: `top`
- Maximize 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 Algorithm Optimization Criteria - Min

Minimize:

- Minimize 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**.
- Minimize waiting time
  - The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O.
  - **Waiting time** is the sum of the periods **spent waiting in the ready queue**.
- Minimize response time
  - The time from the submission of a request until the **first response** is **produced**.
  - 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.
  - Response time is the time it takes to start responding, not the time it takes to output the response. Generally limited by the speed of the output device.

Scheduling Algorithm Optimization Criteria

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

FCFS - First- Come, First-Served Scheduling

Nonpreemptive

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_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

- The code for FCFS scheduling is simple to write and understand.
- Implementation of the FCFS policy is 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.

FCFS Scheduling (Cont.)

- 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** – all other processes wait for the one big process to get off the CPU
- Consider one CPU-bound and many I/O-bound processes
- Troublesome for interactive systems

FCFS dynamic situation

Consider the performance of FCFS scheduling in a **dynamic** situation. Assume we have one CPU-bound process and many I/O-bound processes.

1. The CPU-bound process will get and hold the CPU.
2. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
3. While the processes wait in the ready queue, the I/O devices are idle.
4. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.
5. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
6. At this point, the CPU sits idle.
7. The CPU-bound process will then move back to the ready queue and be allocated the CPU.
8. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.
9. 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.

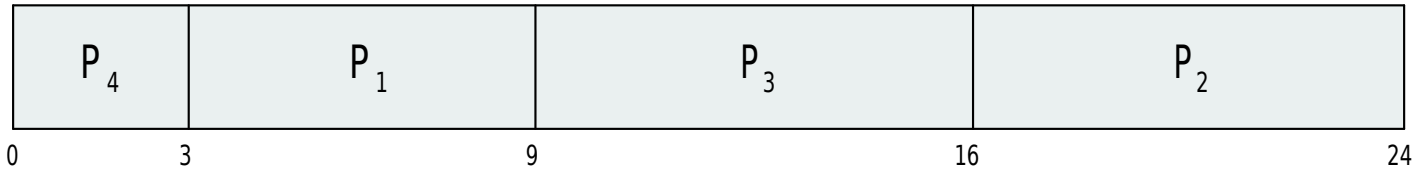
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The real difficulty is knowing the length of the next CPU request
  - Could ask the user
- **SJF can be preemptive or nonpreemptive**
- Preemptive version called **shortest-remaining-time-first**

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

Examples of Exponential Averaging

The parameter α controls the relative weight of recent and past history.

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

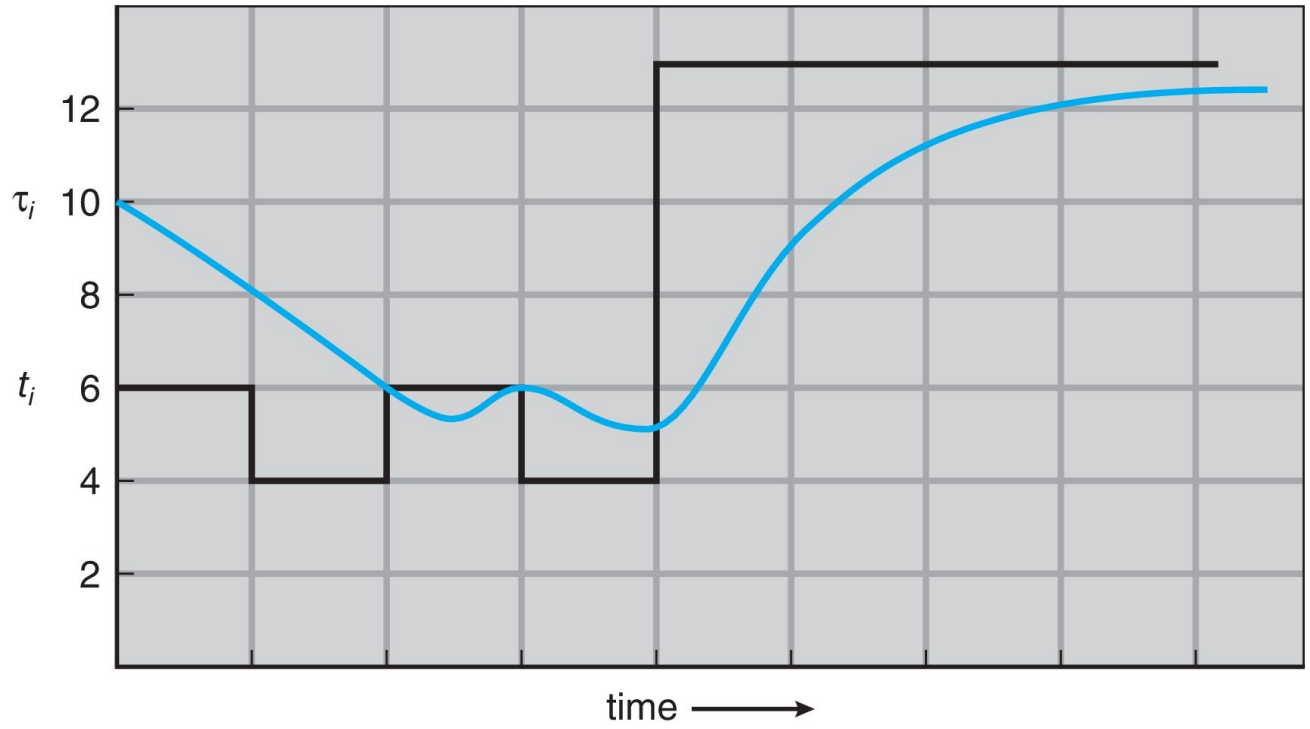
- if $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history has no effect.
- if $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst matters
- Commonly, α is set to ½ so recent history and past history are equally weighted.
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$$

Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst

Predicted value for the next CPU burst: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$.



$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$
$\alpha = 1/2$
initial $\tau_0 = 10$
$t_0 = 6$
$\tau_1 = 0.5 * t_0 + (1\text{-}.5) * \tau_0$
$= .5*6 + .5*10 = 8$

t	CPU burst (t_i)		6	4	6	4	13	13	13	... t_n = actual CPU burst
τ	"guess" (τ_i)	10	8	6	6	5	9	11	12	... τ_{n+1} = predicted value for the next

Example of Shortest-remaining-time-first

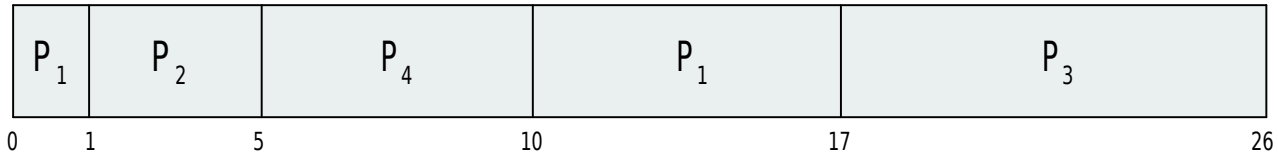
Preemptive

Now we add the concepts of varying arrival times and preemption to the analysis

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

Processes arrive at different times. Subtract the arrival time from each process

- ***Preemptive*** SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5msec
- TT = (17-0) + (5-1) + (26 – 2) + (10 -3) = 17 + 4 + 24 + 7
- WT = (17 – 8) 9 + (4 -4) 0 + (24 – 9) 15 + (7 – 5) 2

Priority Scheduling

Can be **Preemptive or Nonpreemptive**

- A priority number (integer) is associated with each process
- Some systems use low numbers to represent low priority; others use low numbers for high priority.

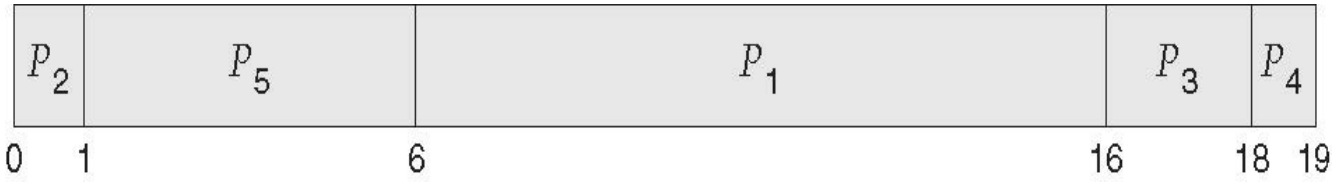
- The **CPU** is allocated to the **process with the highest priority** (assume smallest integer ≡ highest priority)
- **Equal-priority processes** are scheduled in **FCFS** order.

- SJF is a special case of the general priority-scheduling algorithm where priority is the inverse of predicted next CPU burst time. The larger the CPU burst, the lower the priority, and vice versa.

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Problems with priority scheduling

- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process
  - solution for indefinite blockage of low-priority processes
  - 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 periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

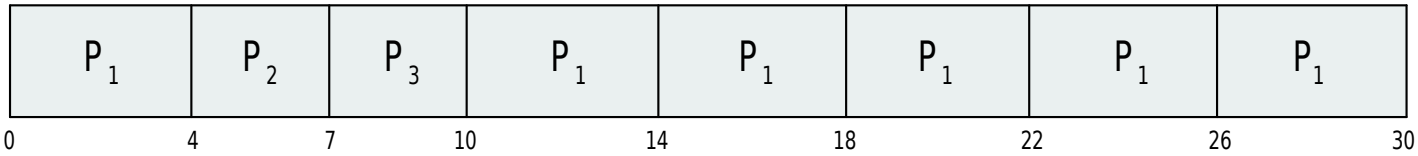
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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 at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

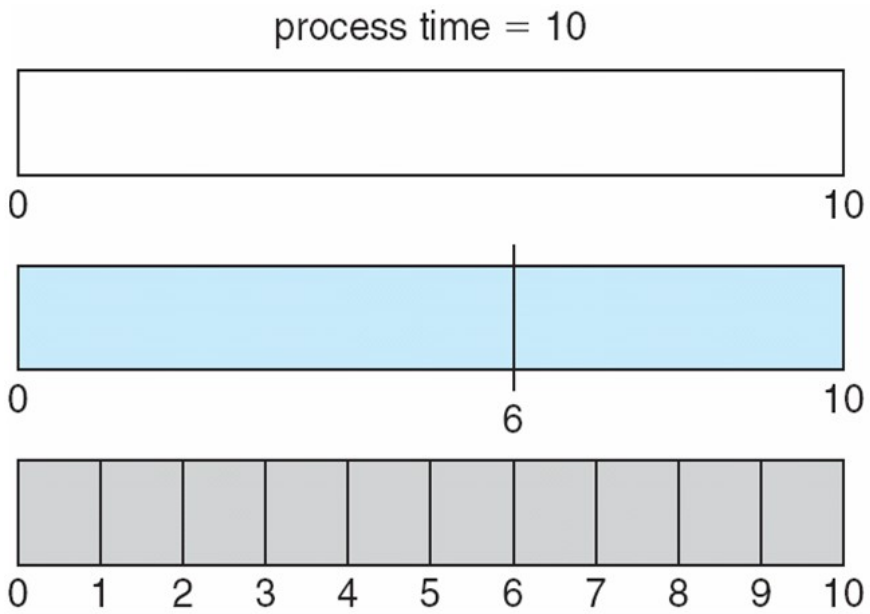
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



  - *P*1 gets the first 4ms. Since it requires another 24-4 = 20ms, it is preempted after the first time quantum, and the CPU is given to the next process
  - *Next, P2 gets 4ms. P*2 needs only 3ms & it exits after 3ms. Same with P3.
  - *P*1 waits for 6ms (10 - 4), *P*2 waits for 4ms, and *P*3 waits for 7ms.
  - Thus, the average waiting time is 17/3 = 5.66 milliseconds.
- Typically, higher average turnaround than SJF, but **better *response***
- q should be large compared to context switch time
- q usually 10milliseconds to 100ms, context switch < 10 microseconds

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

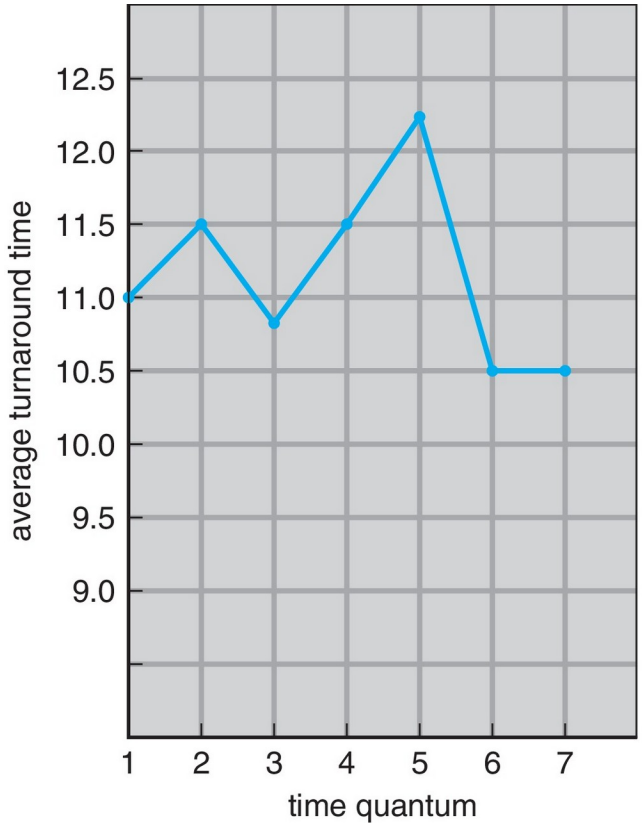
The performance of the RR algorithm depends heavily on the size of the time quantum.

- At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches.
- General rule: 80% of CPU bursts should be shorter than q

RR - turnaround time and time quantum

- Turnaround time also depends on the size of the time quantum.
- The average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
- For example, given three processes of 10 time units each:
  - A quantum of 1 time unit, the average turnaround time is 29.
  - If the time quantum is 10, however, the average turnaround time drops to 20.
- If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

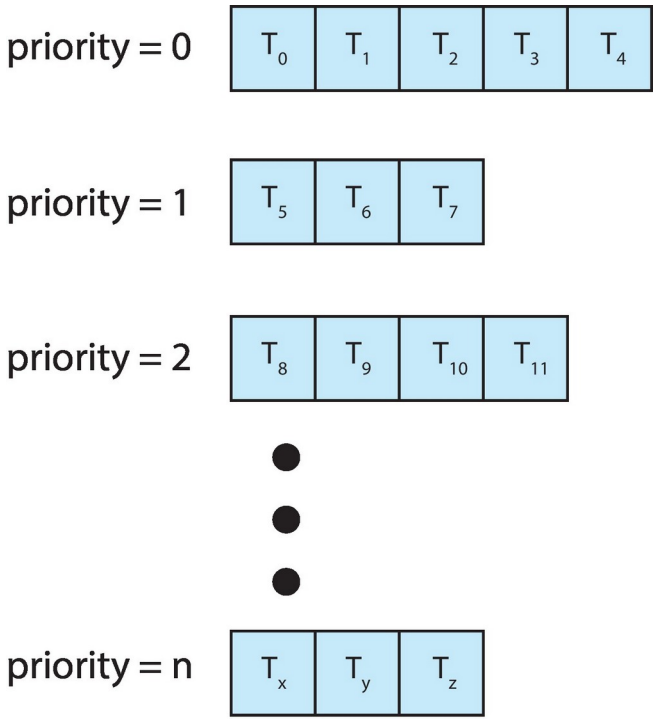
As we can see, the average turnaround time does not necessarily improve as q size increases.

Quantum = 1

P1|P2|P3|P4|P1|P2|P4|P1|P2|P4|P1|P4|P1|P4|P1|P4|P4|

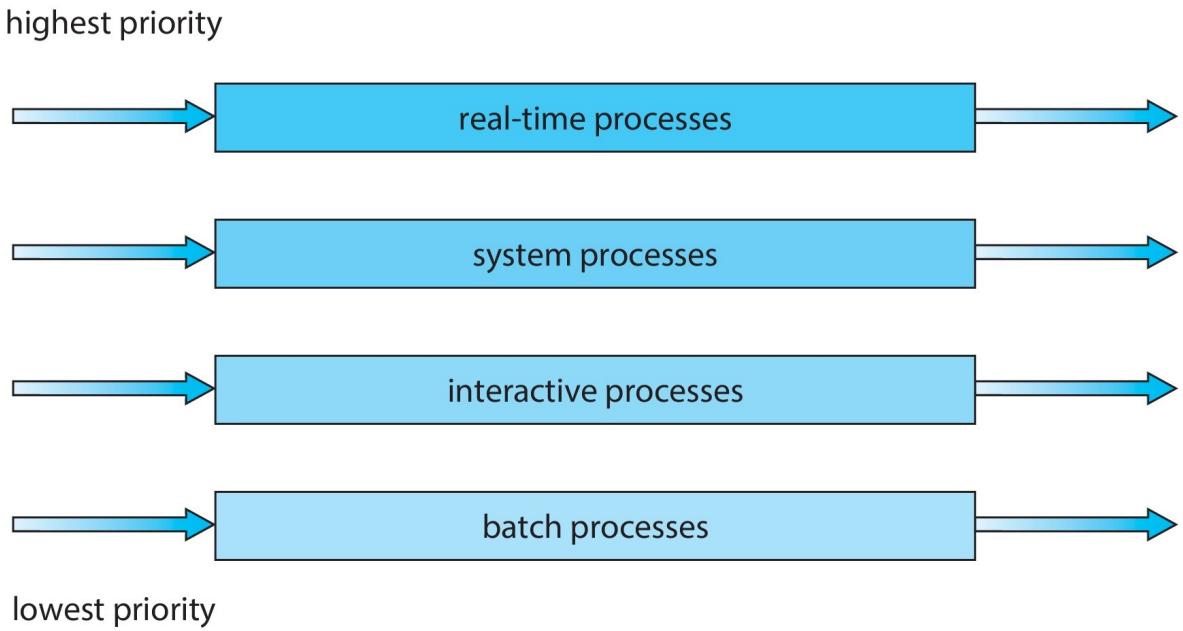
Multilevel Queue

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



Multilevel queue

Prioritization based upon process type:



- Each queue has absolute priority over lower-priority queues.
- No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty.
- If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Multilevel Queue

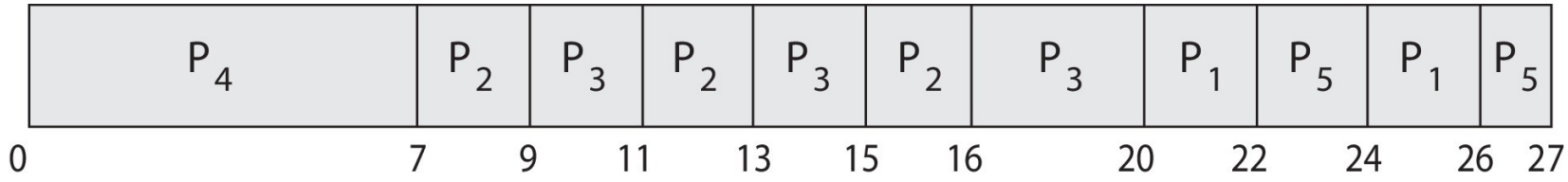
- A priority is assigned statically to each process and a process remains in the same queue for the duration of its runtime
- A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process. These types of processes have different response-time requirements.
  - **foreground** (interactive) may be RR
  - **background** (batch) may be FCFS
- 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 amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS
- Works well when priority scheduling is combined with round-robin.

Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

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

- Gantt Chart wit 2 ms time quantum



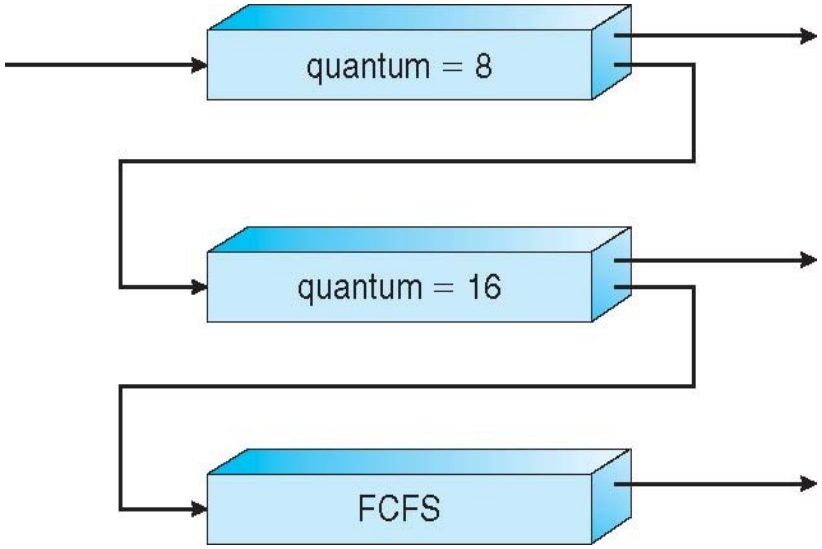
Multilevel Feedback Queue

- 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.
- I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues.
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of **aging** prevents starvation.

- Example: An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process thatwaits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



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

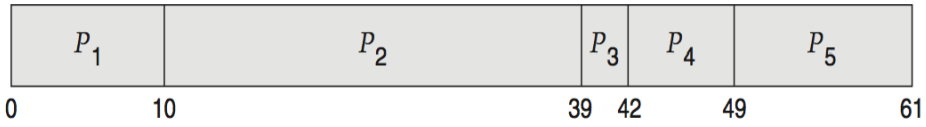
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

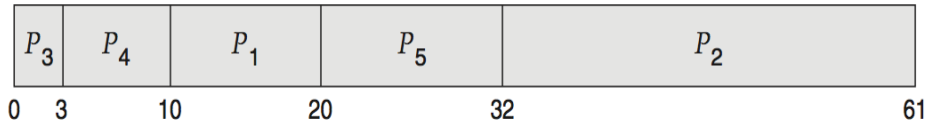
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms: