Chapter 6: CPU Scheduling

Chapter 6: CPU Scheduling

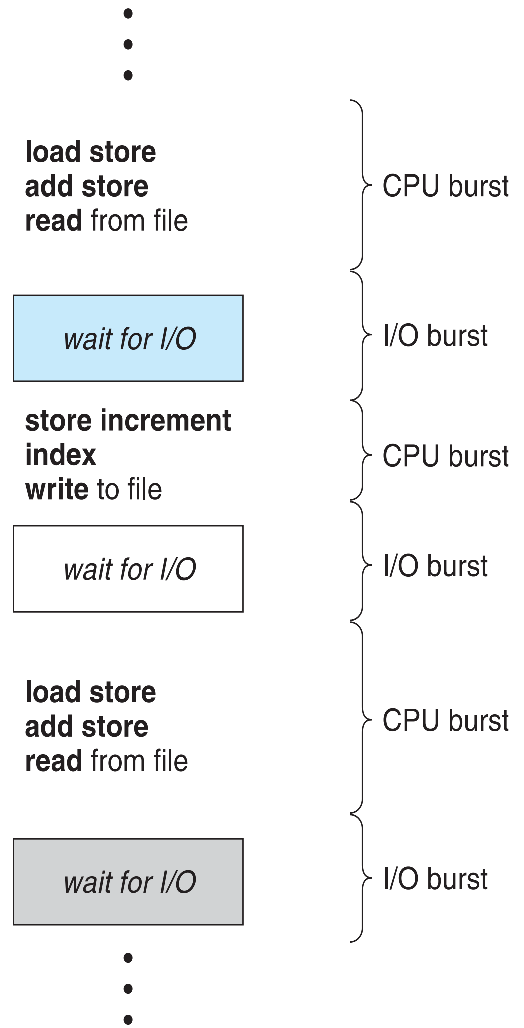
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Algorithm Evaluation

Objectives

- To introduce *CPU scheduling*, which is the basis for multi-programmed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

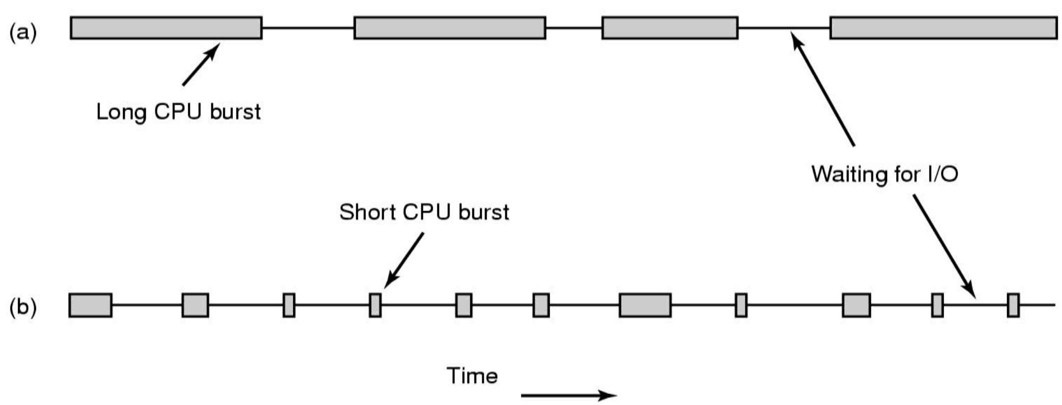
Basic Concepts

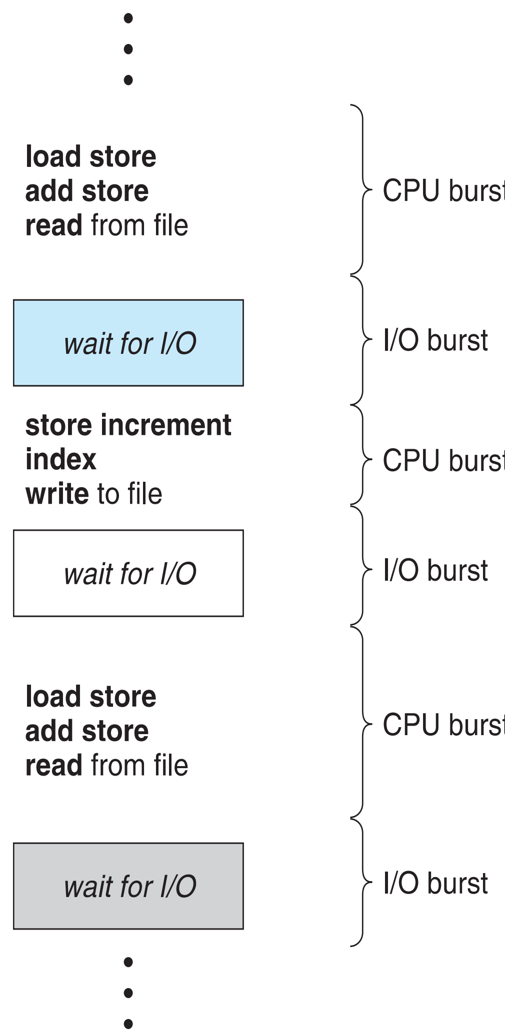
- **Objective of multiprogramming:** Achieve maximum CPU utilization
  - CPU always running a process
  - No Idle CPU
- A process runs in CPU bursts and I/O bursts
  - Run instructions (CPU Burst)
  - Wait for I/O (I/O Burst)



Basic Concepts

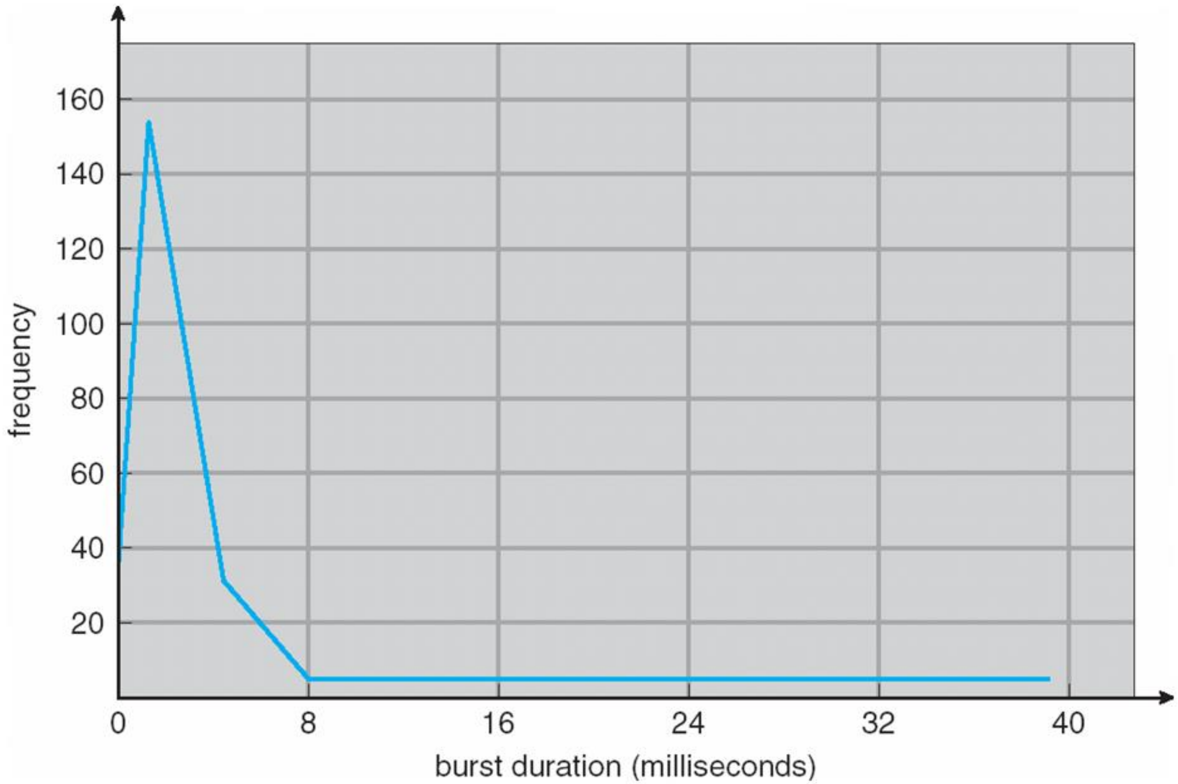
- CPU–I/O Burst Cycle:
  - Process execution consists of a **cycle** of *CPU execution* and *I/O wait*
  - When a process is waiting, then assign the CPU to another process
  - See Process Scheduler on Chap-3
- **CPU burst** followed by **I/O burst**





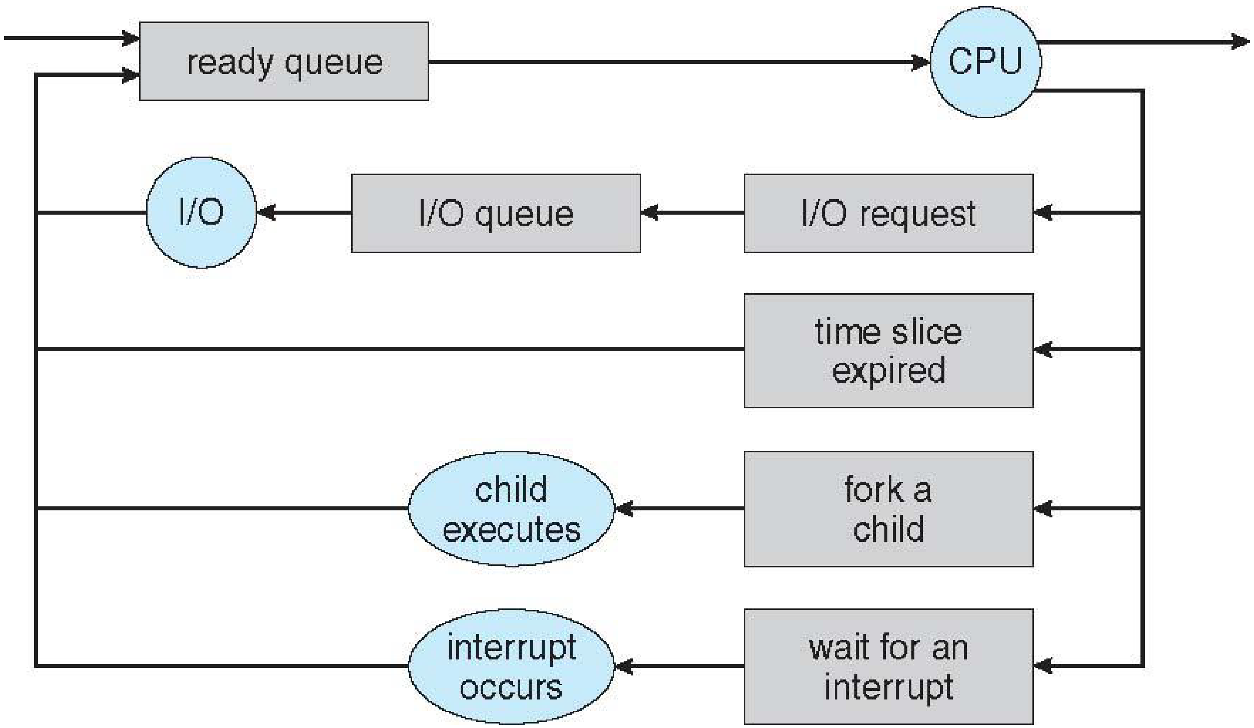
Histogram of CPU-burst Times

- CPU burst distribution is of main concern
- CPU bursts vary from process to process, but an extensive study shows frequency patterns similar to the below diagram:
- Scheduling is aided by knowing the length of these bursts



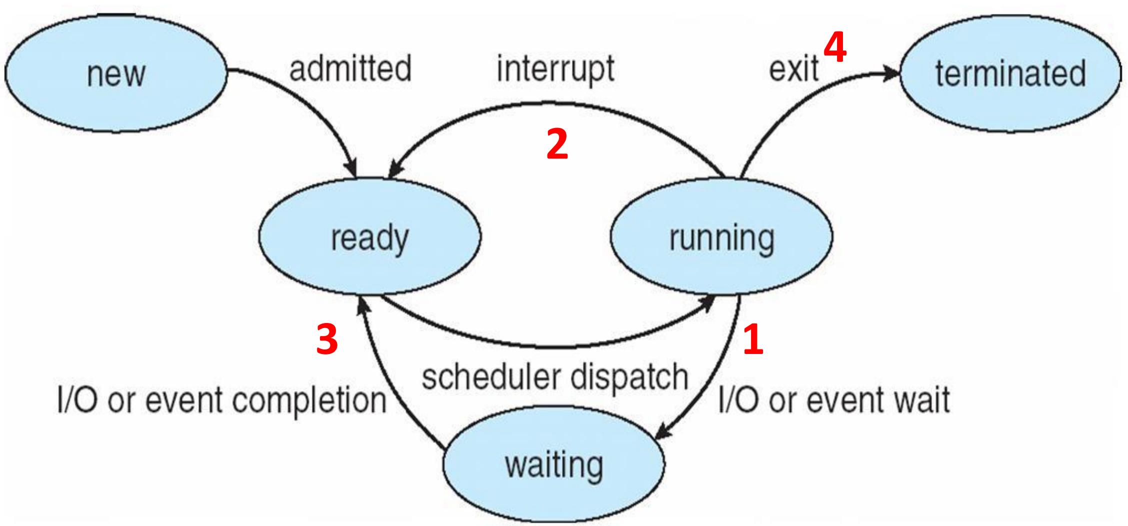
CPU Scheduler Queues

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways: FIFO, LIFO, Random, Priority, … etc
- Types of scheduling
  - **Preemptive scheduling** - Running process may be interrupted and moved to the Ready queue
  - **Non-preemptive scheduling:** - once a process is running, it continues to execute until it terminates or blocks for I/O



Scheduling Decision Modes

- The scheduler runs when it needs to select a process to run on the processor. The **scheduling decision mode** specifies the times at which the *selection of a process to run* is made. This decision can take place at the following times:
  1. Switches from running to waiting state; ex: as result of I/O request or wait()
  2. Switches from running to ready state; ex: when an interrupt occurs
  3. Switches from waiting to ready; at completion of I/O. or because a new process was created and was placed in the queue.
  4. Terminates

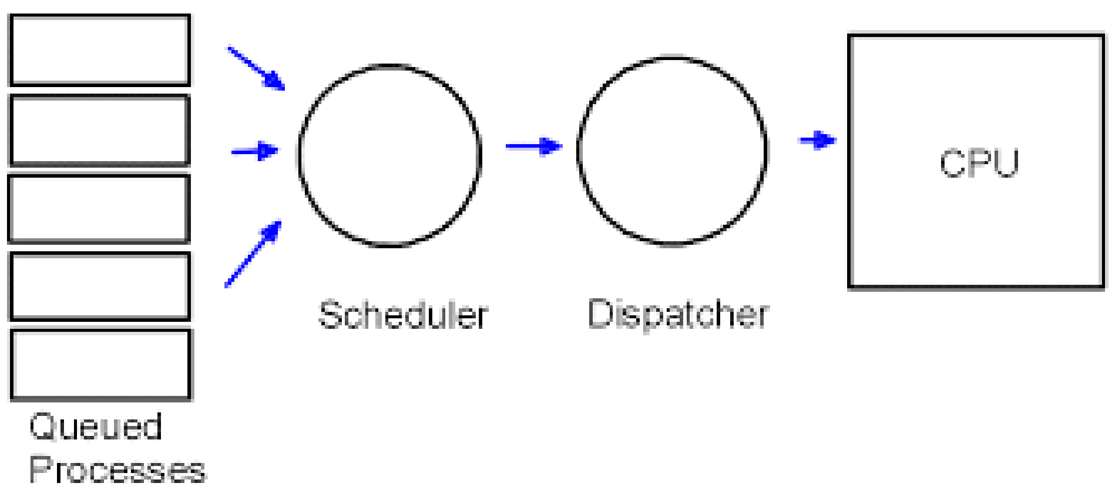


CPU Scheduler

- If a scheduling decision is made **only under conditions 1 and 4**, the scheduling is called **nonpreemptive scheduling**.
  - Process keeps the CPU until it either terminates or switches to waiting state
  - No choice in terms of scheduling; new process must be selected for CPU
- In circumstances 2 and 3, the OS scheduler has a choice:
  - it can either allow the current process to continue running, or it could step in and put the current process to sleep and select a different process to run. The latter operation is called "**preempting**" the current process and scheduling a new one to run.

The Dispatcher

- The **dispatcher** is the kernel routine that performs the context switch.
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - *switching context* from the currently running process to the new one.
  - *switching to user mode* by changing the processor mode to user mode
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
  - Dispatcher is invoked during every process switch; hence it should be as fast as possible



Scheduling Criteria

- **Scheduling criteria** are the objectives of scheduling algorithms.
- **CPU utilization** – keep the CPU as busy as possible
  - Ranges from 40% to 90% i.e., from light to heavy loaded
  - Scheduling algorithm optimization criteria: Max CPU utilization
- **Throughput** – # of processes that completing per time unit
  - Ranges from 10 processes/second to 1 process/hour
  - Scheduling algorithm optimization criteria: Max throughput
- **Turnaround time** – the time from the submission of a job until it completes;
  - Sum of times spent in job pool + ready queue + CPU execution + doing I/O
  - Scheduling algorithm optimization criteria: Min turnaround time
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - Sum of times spent waiting in the ready queue
  - Scheduling algorithm optimization criteria: Min waiting time
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output
  - The time it takes to start responding to the user; in an interactive system
  - Scheduling algorithm optimization criteria: Min response time

Scheduling Algorithm Optimization Criteria

- It is desirable that we:
- **Maximize:**
  - CPU utilization
  - throughput
- **Minimize:**
  - turnaround time
  - waiting time
  - response time

Scheduling Algorithms

- First come first serve
- Shortest Job First
- Shortest Remaining Time First
- Round Robin
- Multilevel Queue Scheduling
- Multilevel Feedback Queueing

First- Come, First-Served (FCFS) Scheduling

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.

First- Come, First-Served (FCFS) Scheduling

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:[includes start and finish time of each process]

Throughput: 3 jobs/30 seconds = 0.1 jobs/second

- Turnaround Time: P1 : 24, P2 : 27, P3 : 30
  - Average TT: (24 + 27 + 30)/3 = 27
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
- The simplest scheduling algorithm but usually very bad average waiting time

FCFS Scheduling (Cont.)

Now suppose that the processes arrive in the order:

P_2 , P_3 , P_1, (instead of P_1,P_2,P_3) where Burst times, for P_2 = 3, P_3=3 and P_1 = 24.

- The Gantt chart for the schedule is: ??

- **Throughput:** ?
- **Turnaround time:** ?
- **Average TT:** ?
- **Waiting time** for $P_1 = ?; P_2 = ?; P_3 = ?$
- **Average waiting time:** ?

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1 , where Burst times, for P_2 = 3, P_3=3 and P_1 = 24.

- The Gantt chart for the schedule is:

- **Throughput:** 3 jobs / 30 sec = 0.1 jobs/sec
- **Turnaround time:** P_1 = 30, P_2 = 3, P_3 = 6
- **Average TT:** (30 + 3 + 6)/3 = 13 → much less than 27
- **Waiting time** for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- **Average waiting time:** (6 + 0 + 3)/3 = 3; 3 instead of 17, substantial reduction in wait time
- Much better than previous case
- **Lesson:** scheduling algorithm can reduce TT
  - Minimizing waiting time can improve TT
- Can a scheduling algorithm improve throughput?
  - Yes, if jobs require both computation and I/O

FCFS Scheduling: Convoy Effect (Cont.)

- **Convoy effect** - short processes maybe stuck behind long processes
  - CPU-bound processes hold CPU while I/O-bounds wait in ready queue for the CPU
    - I/O devices are idle until CPU released… then CPU is idle.. Then … this repeats
  - Consider one long CPU-bound and many I/O-bound processes
    - I/O-bound processes spend most of the time waiting for CPU-bound to release CPU
    - Result in lower CPU and device utilization
- **FCFS is nonpreemptive:** process holds CPU until termination or I/O request

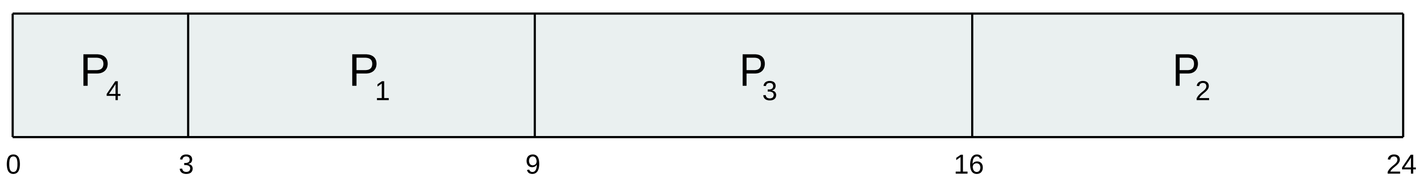
Shortest-Job-First (SJF) Scheduling

- The preceding examples showed that when short processes are scheduled before long processes, the average waiting times are smaller than if they were to follow after it.
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - Assign the CPU to the process that has the smallest next CPU burst
  - What if two process have the same next CPU burst length? FCFS breaks the tie
- Better term: Shortest-Next-CPU-Burst-Scheduling (SNCB) algorithm
  - But most books use SJF
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user to estimate their processes' time limits; for job scheduling
    - CPU scheduling can use these time limits as estimates of CPU burst lengths

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
  - With FCFS it would be 10.25 time units with this order P1 P2 P3 P4
- Moving a short process before a long one decreases its waiting time more than it increases the long process's waiting time. Thus, decrease of average waiting time

Determining Length of Next CPU Burst

- Burst time can be estimated using lengths of past bursts: next = average of all past bursts
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using ***exponential weighted moving average***; **next = average of (past actual + past estimate)**

1. T_n = actual length of n^{th} CPU burst
2. S_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$\boldsymbol{S_{n+1} = (1-\alpha)\, S_n + \alpha\, T_n}$$

- Relative weight of recent history and past history. α = ½ usually but anything in the range [0, 1] is acceptable
- Commonly, α set to ½

- Note the difference between English alphabet t (pronounced TEE) and Greek letter τ (pronounced tau)

Prediction of the Length of the Next CPU Burst

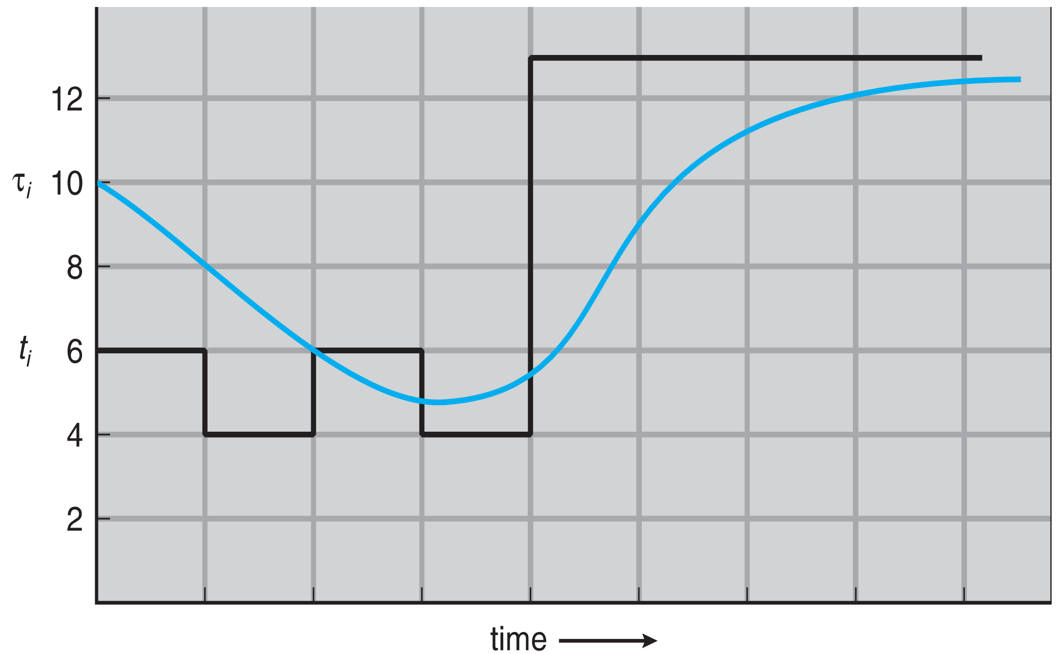
How to find next estimated burst, if initial estimate is $S_0 = 10$ and $T_0 = 6$:

$S_{(1)} = 0.5 \text{ x } 10 + 0.5 \text{ x } 6 = 3 + 5 = 8.$

$S_{(2)} = 0.5 \text{ x } 8 + 0.5 \text{ x } 4 = 4 + 2 = 6$

$S_{(3)} = ?$

$S_{(4)} = ?$



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- When α =0, then the next estimated burst $\boldsymbol{S_{n+1} = (1 - \alpha) S_n + \alpha T_n}$ becomes
  - $S_{n+1} = S_n$
  - Recent history does not count
- When α =1, then the next estimated burst $\boldsymbol{S_{n+1} = (1 - \alpha) S_n + \alpha T_n}$ becomes
  - $S_{n+1} = T_n$
  - Only the actual last CPU burst counts

Example of Shortest-Remaining-Time-First (SRTF)

- **Preemptive** version of SJF is called **shortest-remaining-time-first**
  - The next burst of new process may be shorter than that left of current process
    - Currently running process will be preempted
- Now we add the concepts of varying arrival times and preemption to the analysis

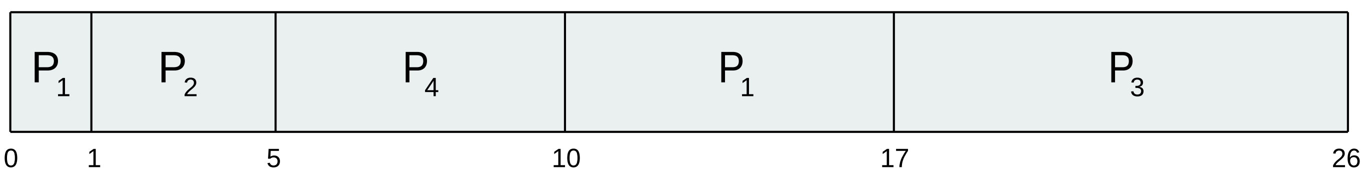
Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Draw Gantt chart for above mentioned processes:

Example of Shortest-remaining-time-first (SRTF)

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

- *Preemptive* SJF Gantt Chart



Waiting Time=Finish Time−Arrival Time−Burst Time

- **Waiting time** for $P_1 = (10\text{-}1)$; $P_2 = (1\text{-}1)$; $P_3 = (17\text{-}2)$, $P_4 = (5\text{-}3)$
- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3))]/4 = 26/4 = 6.5 msec
    - Nonpreemptive SJF will result in 7.75 time units

SJF Limitations

- SJF doesn't always minimize average Turnaround Time (time from entering the system till completion)
  - Only minimizes waiting time
- Can lead to unfairness or starvation → newly arriving tasks have shorter bursts
- In practice, can't actually predict the future
- But can estimate CPU burst length based on past

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - We assume that smallest integer ≡ highest priority
    - Internally defined priority: based on criteria within OS. Ex: memory needs
    - Externally defined priority: based on criteria outside OS. Ex: paid process
  - Preemptive; a running process is preempted by a new higher priority process
  - Nonpreemptive: running process holds CPU until termination or I/O request
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

Priority Scheduling and Starvation

- it is possible for a process with low priority to wait indefinitely to run because a stream of higher priority processes keeps arriving. This is called **starvation**.
- Solution ≡ This problem can be overcome by using *dynamic priorities*, in which a process's priority changes over time.

- **Aging** – as time progresses increase the priority of the process waiting in the ready queue increases
  - **Example:** do

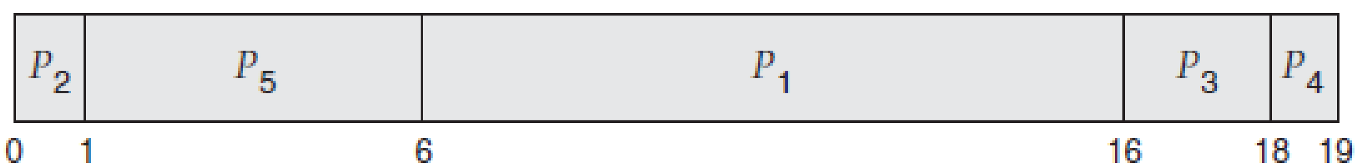
    priority = priority – 1 //increase priority

    every 15 minutes
- Conversely, processes that spend too much time on the processor can have their priorities reduced.

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 ($P_2 > P_5 > P_1 > P_3 > P_4$)



- Waiting time for P1 = (16-10); P2 = (1-1); P3 = (18-2), P4 = (19-1) , P5 = (6-5)
- Average waiting time = $(0+1+6+16+18) = 41/5 =$ 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum or time slice** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
  - Ready queue is treated as a circular 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.

Round Robin (RR)

- Timer interrupts every quantum to schedule next process
- Performance depends heavily on the size of the time quantum
  - *If q very large $\Rightarrow$ RR scheduling = FCFS scheduling*
  - *If q very small $\Rightarrow$ q must be large with respect to context switch,*
    - Otherwise overhead of number of context switches will be is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

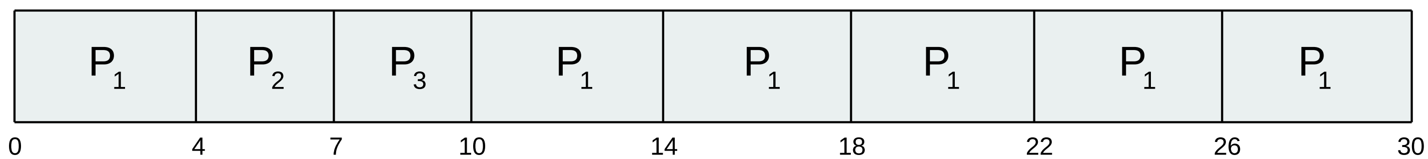
- The Gantt chart when time quantum is 4 seconds:

- Average Waiting Time = ?.

Example of RR with Time Quantum = 4

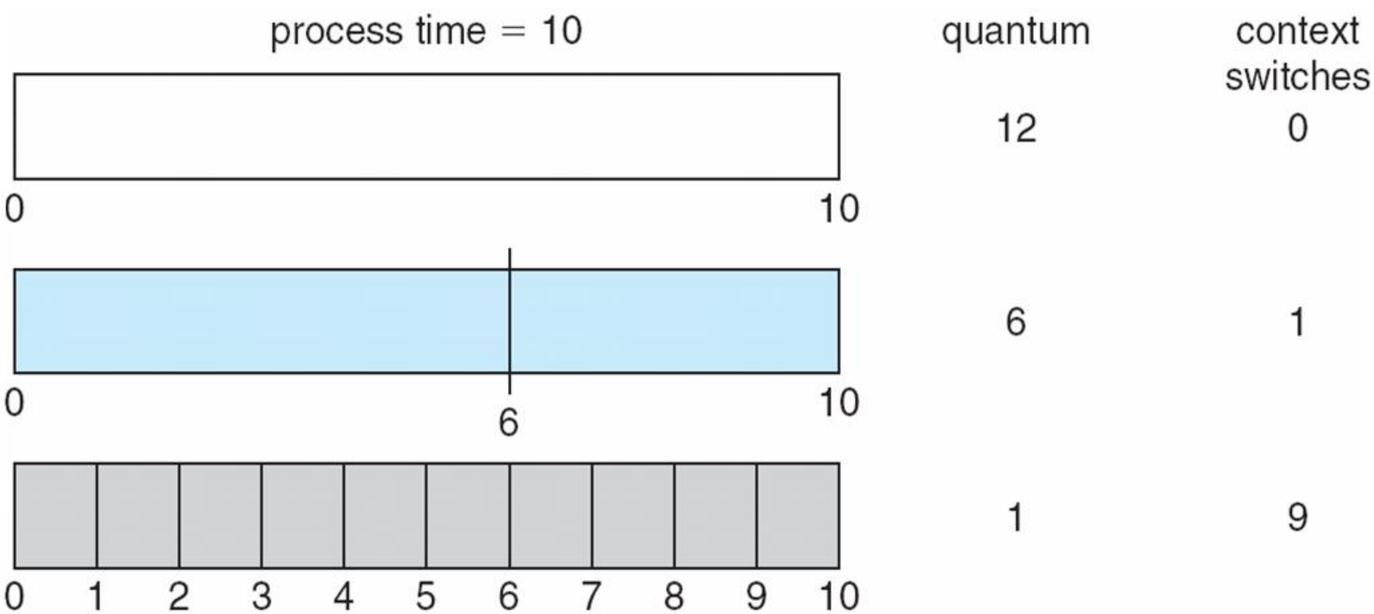
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart when time quantum is 4 seconds long:

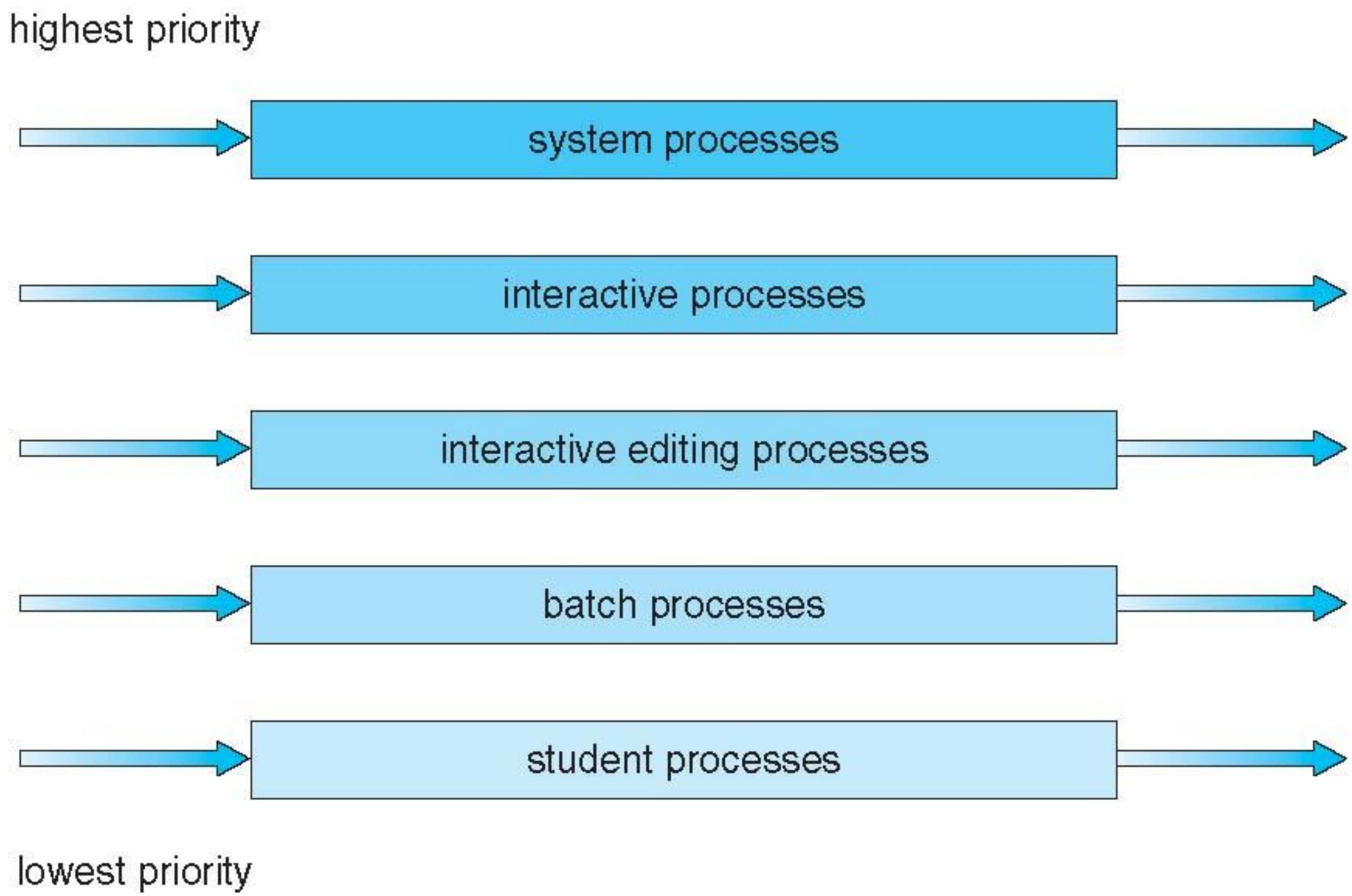


- We assume the arrival time for each process is 0.
- **Waiting time** for P_1 = (10-4); P_2 = 4; P_3 = 7,
- Average Waiting Time = [(10-4)+4+7]/3 = 5.66milliseconds.
- Average waiting time under RR scheduling is often long
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Multilevel Queue Scheduling



Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues, e.g: two queues containing
  - **foreground** (interactive) processes
    - May have externally defined priority over background processes
  - **background** (runs without active user interaction)
- Process permanently in a given queue; no move to a different queue

  (different than ***multilevel feedback queue***, will be covered soon)

Multilevel Queue Scheduling

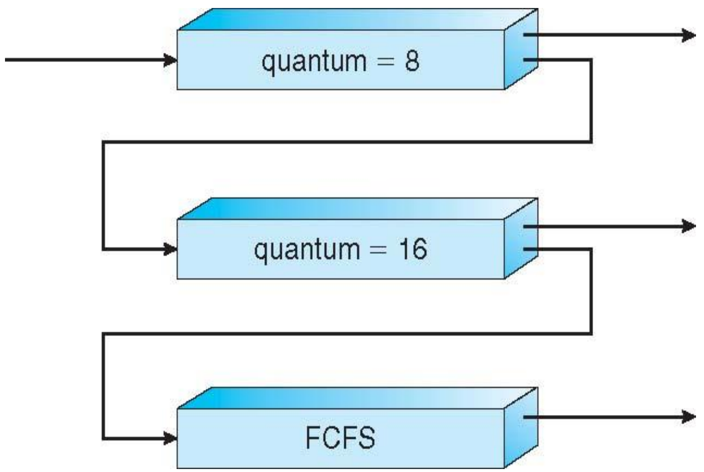
- Each queue has its own scheduling algorithm:
  - foreground – RR or …
  - background – FCFS or …
- Scheduling must be done between the queues:
  - **Fixed priority scheduling;** (i.e., serve all from foreground first then from background). Possibility of starvation.
  - **Time slice** – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - And 20% to background in FCFS

Multilevel Feedback Queue

- A process can move between the various queues; ***aging*** can be implemented this way
- Demote → Move heavy foreground CPU-bound process to the background queue
- Upgrade → Move starving background process to the foreground queue

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
    - Highest priority. Preempts Q1 and Q2 proc's
  - Q_1 – RR time quantum 16 milliseconds
    - Medium priority. Preempts processes in Q2
  - Q_2 – FCFS
    - Lowest priority



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

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Algorithm Evaluation

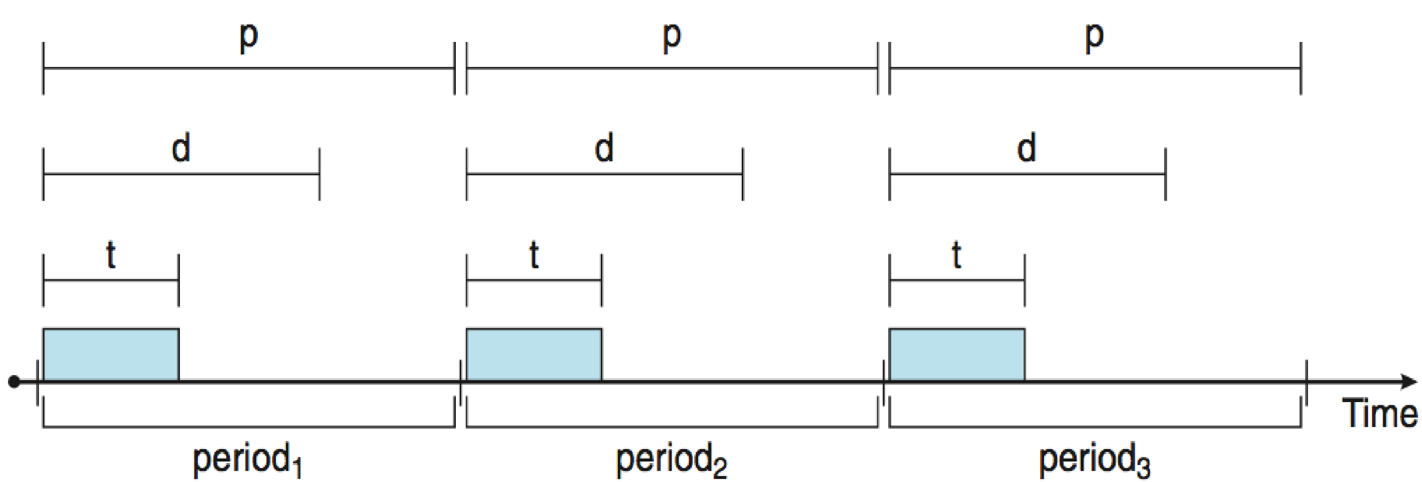
Real-Time CPU Scheduling

- A real-time operating system is for ***real-time applications*** that processes data and events that have critically defined time constraints. For example patient monitoring in ICU, air traffic control, robot control in automated factory.
- In a real-time system the correctness of the system behavior depends not only the logical results of the computations, but also on the time at which these results are produced.
- **Soft real-time systems** – In these type of systems, an associated deadline is with the soft-real time task is desirable but not mandatory, i.e., missing a deadline is acceptable.
- **Hard real-time systems** – These type of OS strictly adhere to the ***deadline*** associated with the tasks. Missing on a deadline can have catastrophic affects.

Priority-based Scheduling

- Real-time OS responds immediately to a real-time process when it requests CPU
  - For real-time scheduling, scheduler must support preemptive, priority-based scheduling
- For hard real-time, it must also provide ability to meet deadlines
- Processes have new characteristics:
  - **periodic** processes require CPU at constant intervals
    - Has processing time ***t***, deadline ***d***, period ***p***
    - $0 \le t \le d \le p$
    - **Rate** of periodic task is $1/p$

Admission-control: process announces its requirements, then scheduler admits the process if it can complete



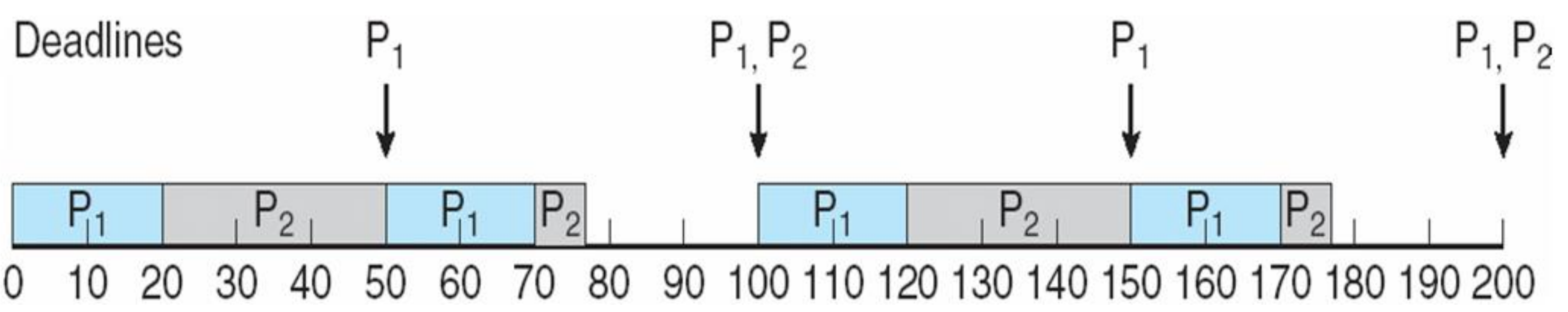
1. Rate Monotonic Scheduling

- ***Static priority*** policy with preemption
  - A priority is assigned based on the inverse of its period ***p***
    - Shorter periods = higher priority; Longer periods = lower priority
    - Assumes processing time ***t*** is the same for each CPU burst
    - Rationale: higher priority to tasks that require the CPU more often

- Example:
  - Process P_1 : t_1 = 20, d_1 = complete CPU burst by start of next period, p_1 = 50.
  - Process P_2 : t_2 = 35, d_2 = complete CPU burst by start of next period, p_2 = 100.

1. Rate Monotonic Scheduling

- Example:
  - Process P_1 : t_1 = 20, d_1 = complete CPU burst by start of next period, p_1 = 50.
  - Process P_2 : t_2 = 35, d_2 = complete CPU burst by start of next period, p_2 = 100.
  - P_1 is assigned a higher priority than P_2. Why?
  - CPU utilization = t_i / p_i. Thus total CPU utilization = 20/50 + 35/100 = 75%
  - In this example both processes meet the deadlines successfully



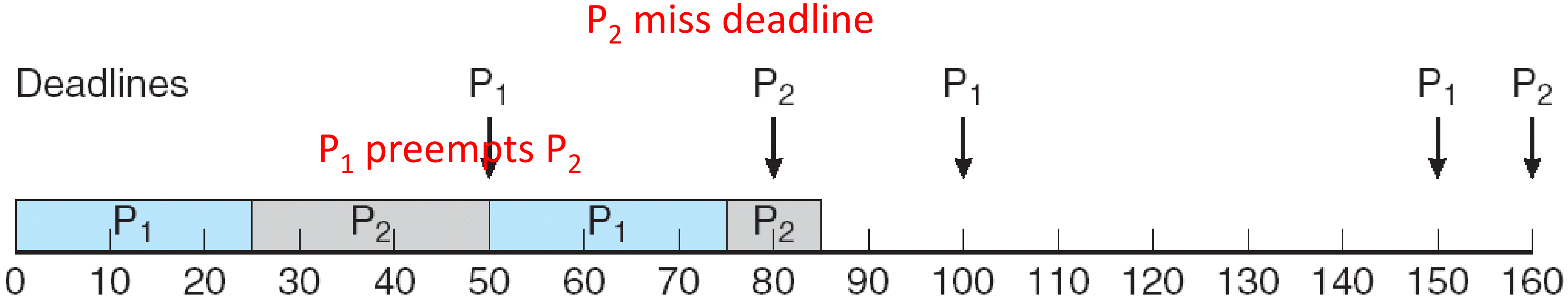
Problem with Rate Monotonic Scheduling

- Let's change process P_1 processing time from 20 to 25

- Process P_1 : t_1 = 25, d_1 = complete CPU burst by start of next period, p_1 = 50.
- Process P_2 : t_2 = 35, d_2 = complete CPU burst by start of next period, p_2 = 80.
- Total CPU utilization = 25/50 + 35/80 = 94% (in previous example U = 75%)
- P1 is assigned a higher priority than P2. Why?

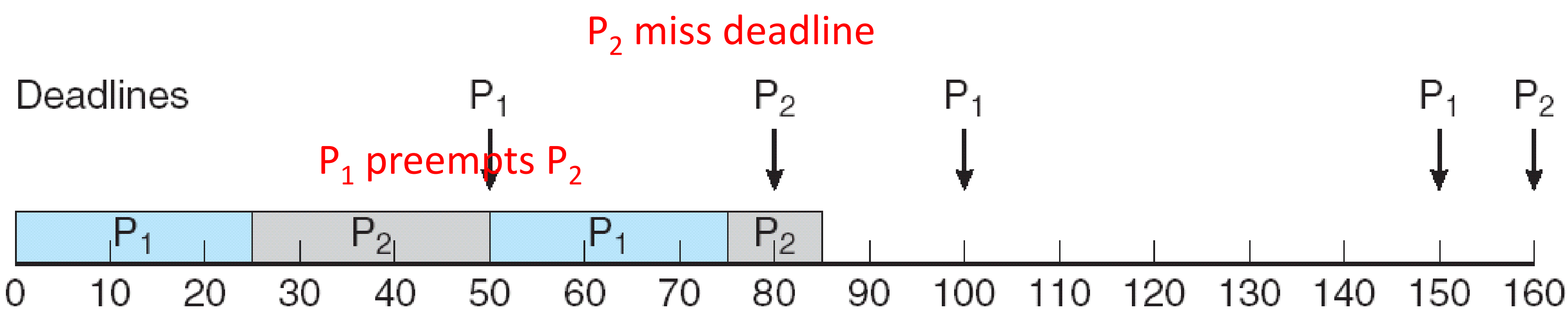
Problem with Rate Monotonic Scheduling: Missed Deadlines

- Let's change process P_1 processing time from 20 to 25 and period to 80
- Process P_1 : t_1 = 25, d_1 = complete CPU burst by start of next period, p_1 = 50.
- Process P_2 : t_2 = 35, d_2 = complete CPU burst by start of next period, p_2 = 80.
- Total CPU utilization = 25/50 + 35/80 = 94% (in previous example U = 75%)
- P1 is assigned a higher priority than P2. Why?
- **P2 has missed the deadline for completion of its CPU burst at time 80**
- **Rate Monotonic Scheduling has limitation.**



Problem with Rate Monotonic Scheduling: Missed Deadlines

- Rate Monotonic Scheduling has limitation.
- Total CPU utilization = 25/50 + 35/80 = 94%
- Reason - bounded CPU utilization; worst case for N processes: $B = N(2^{1/N} - 1)$
  - Bounded at B = 83% for N = 2 processes
  - Theory: cannot guarantee that processes can be scheduled to meet their deadlines if actual CPU utilization > Bound B from ***UB test***
    - 94% > 83% in the example above



Class Exercise

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

- What is the utilization of the system in the above scenario?
- According to UB test, what is the sufficient condition under which the system is schedulable?
- Is the system schedulable??

Class Exercise

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

- What is the utilization of the system in the above scenario?
  - 0.725
- According to UB test, what is the sufficient condition under which the system is schedulable?
  - 0.7797
- Is the system schedulable??
  - Yes: 0.725 < 0.7797

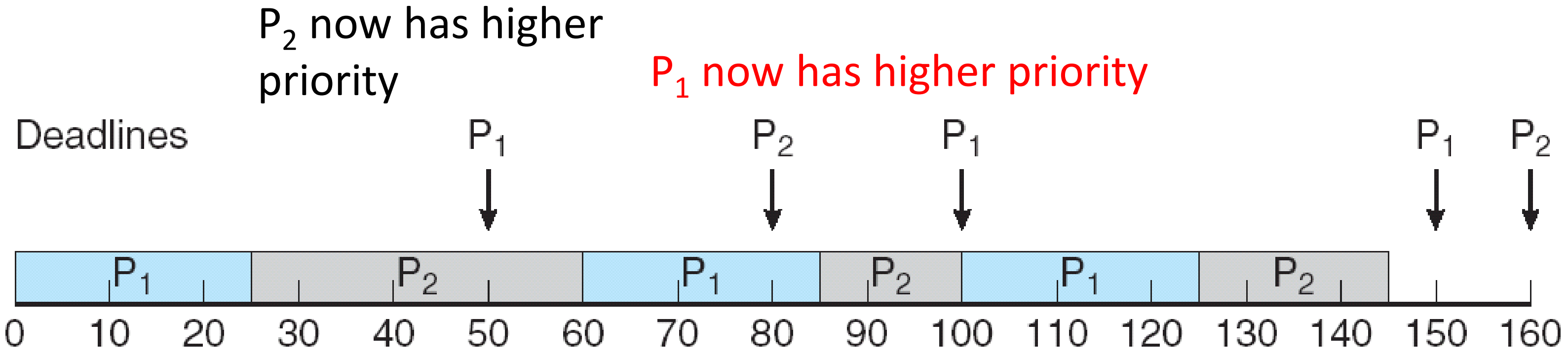
2. Earliest Deadline First Scheduling (EDF) (Solution)

- *Dynamic priority* policy with preemption
  - Priorities are dynamically assigned according to deadlines:
    - **Earlier deadline = higher priority; later deadline = lower priority**
  - New runnable process must announce its deadline requirements to scheduler
    - Scheduler will adjust current priorities accordingly
  - Process need not be periodic
  - CPU burst time need not be constant

2. Earliest Deadline First Scheduling (EDF) (Solution)

- *Example:*

- Process P_1 : t_1 = 25, d_1 = complete CPU burst by start of next period, p_1 = 50.
- Process P_2 : t_2 = 35, d_2 = complete CPU burst by start of next period, p_2 = 80.
- P_1 is initially assigned a higher priority than P_2 since $d_1 = 50 \le d_2 = 80$
- But later P_2 is assigned higher priority than P_1.



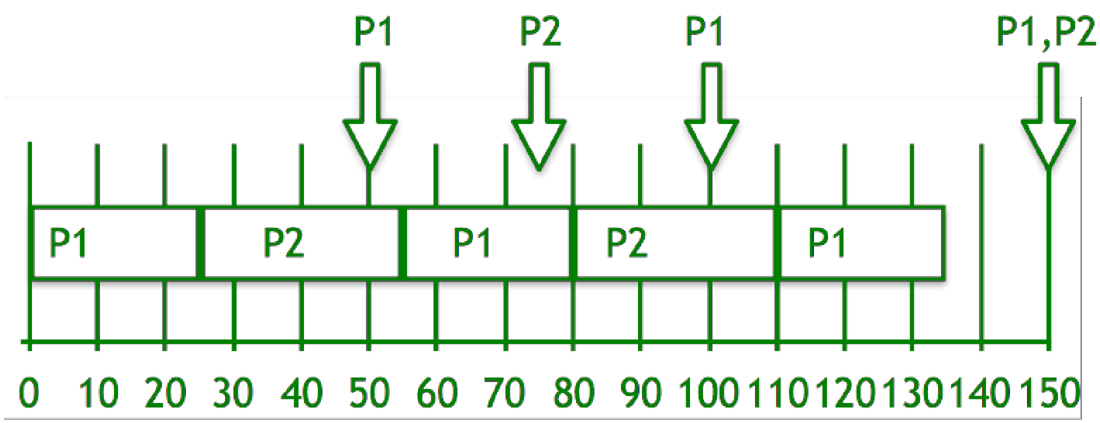
2. Earliest Deadline First Scheduling (EDF) (Solution)

- *Exercise:*

- Process P_1 : t_1 = 25, d_1 = complete CPU burst by start of next period, p_1 = 50.
- Process P_2 : t_2 = 30, d_2 = complete CPU burst by start of next period, p_2 = 75.
-

2. Earliest Deadline First Scheduling (EDF) (Solution)

- *Exercise:*
- Process P_1 : t_1 = 25, d_1 = complete CPU burst by start of next period, p_1 = 50.
- Process P_2 : t_2 = 30, d_2 = complete CPU burst by start of next period, p_2 = 75.
- P_1 is initially assigned a higher priority than P_2 since $d_1 = 50 \le d_2 = 75$
- But later P_2 is assigned higher priority than P_1.
- Priorities are dynamically changed.



3. Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time
- Example: three processes A, B and C, with T = 100 shares
  - A, B and C assigned each 50, 15 and 20 shares, respectively
  - Thus A will have 50% of total processor times, and so on with B and C
- Scheduler must use admission-control policy:
- **Admission-control**: process announces its requirements, then scheduler admits the process if it can complete it on time, or, reject it if it cannot be serviced
  - Admit a request only if sufficient shares are available
  - Example: If new process D needs 30 share then scheduler will deny him CPU
    - Current total share is 50 + 15 + 20 = 85
    - Only a new process with share < 15 can be scheduled

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Algorithm Evaluation

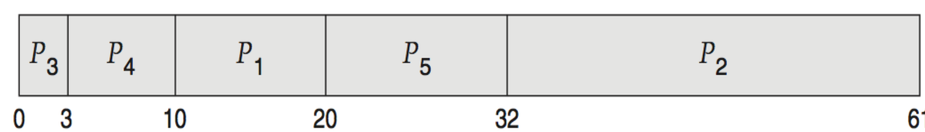
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria (e.g., throughput, utilization etc.), 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:

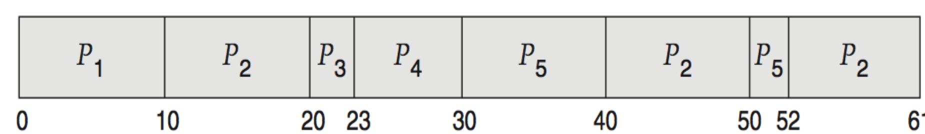
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- FCFS is 28ms:

- Non-preemptive SFJ is 13ms:



- RR is 23ms:



(2/4) Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
- Computer system described as *network of servers*, each with queue of waiting processes
    - Commonly exponential, and described by mean
    - Knowing arrival rates and service rates
    - Computes utilization, average queue length, average wait time, etc

(3/4) Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically

(4/4) Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary

End of Chapter 6