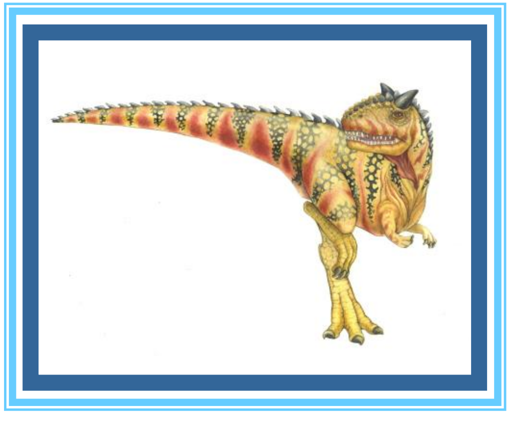
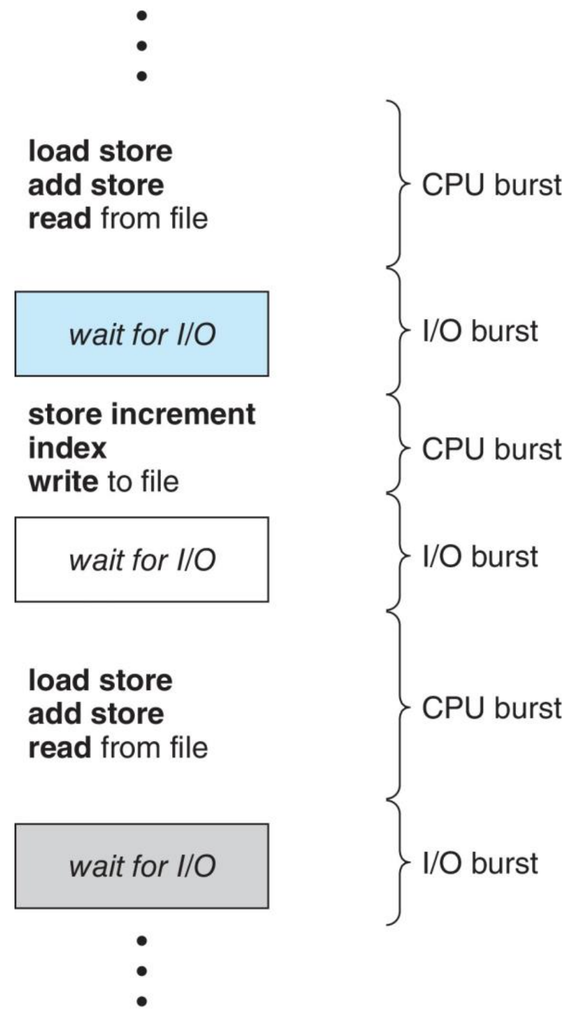
Chapter 5: CPU Scheduling



Basic Concepts

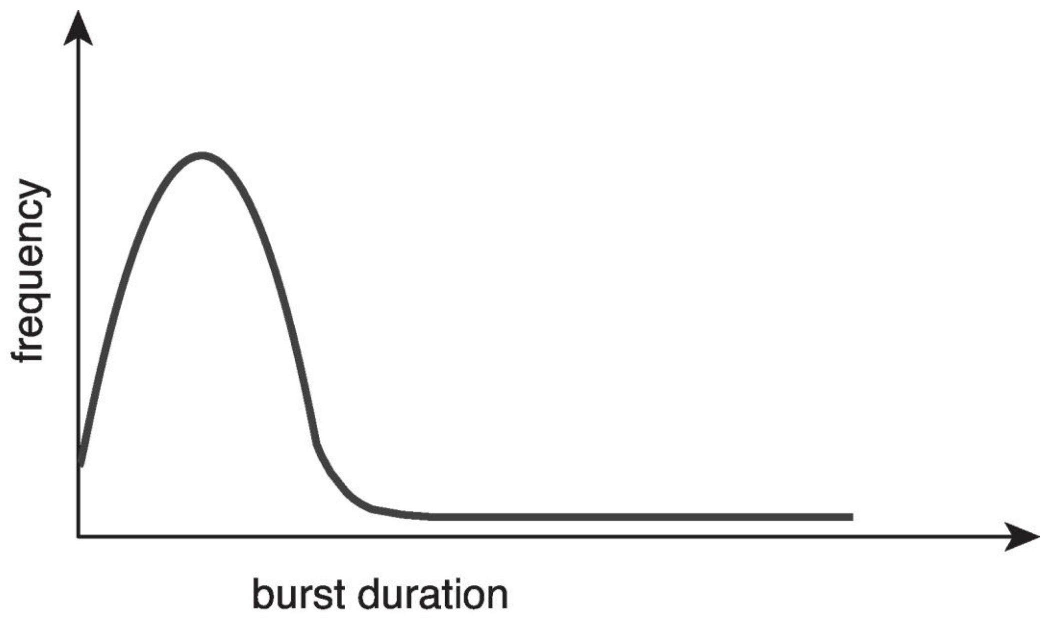
- Maximum CPU utilization obtained with multiprogramming
- Process execution consists of a **cycle** of **CPU burst** and **I/O burst**
- An **I/O-bound program** typically has many short CPU bursts
- A **CPU-bound program** typically have a few long CPU bursts



Histogram of CPU-burst Times

Large number of short CPU bursts

Small number of long CPU bursts



CPU Scheduler

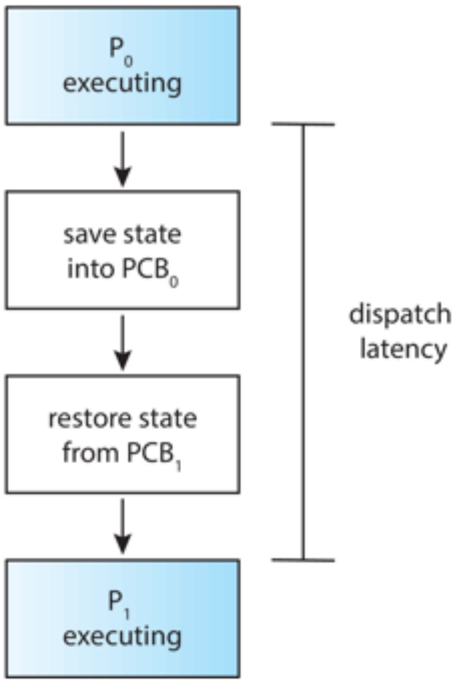
- **CPU scheduler** selects one of the processes in ready queue and allocates the CPU to that process
  - Ready queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g., I/O request)
  2. Switches from running to ready state (e.g., timer interrupt)
  3. Switches from waiting to ready (e.g., I/O completion)
  4. Terminates
- Scheduling is **nonpreemptive** when it takes place only under circumstances 1 and 4. Otherwise it is **preemptive**

Linux command
https://linux.die.net/man/8/vmstat

$vmstat 1 5
$cat proc/<pid>/status

Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the CPU 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

- **CPU utilization:** 0-100%, want to keep the CPU as busy as possible
- **Throughput:** # of processes that are completed per time unit
- **Turnaround time:** amount of time to execute a particular process, from time of submission to time of completion
- **Waiting time:** amount of time a process spends waiting in the ready queue
- **Response time:** time from the submission of a request until the first response is produced (i.e., time it takes to start responding)

- Optimization criteria
  - Maximize CPU utilization and throughput
  - Minimize average turnaround time, waiting time, and response time

CPU Scheduling Algorithms

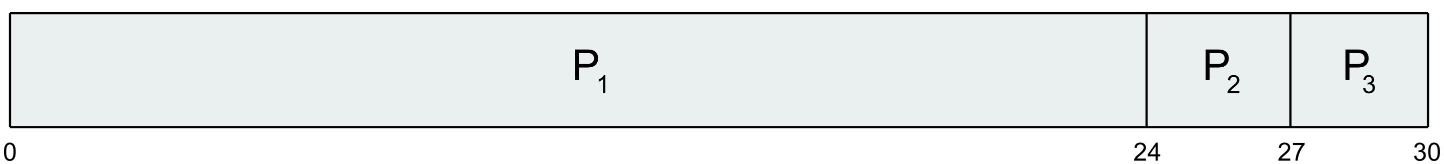
- First Come First Served
- Shortest Job First
- Shortest Remaining Time First
- Round Robin
- Priority Scheduling
- Priority Scheduling with Round-Robin

First-Come, First-Served (FCFS) Scheduling

- Can be implemented using a FIFO queue

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: $(0 + 24 + 27)/3 = 17$
- FCFS scheduling is nonpreemptive
- Animation 5.1

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

Problems with FCFS

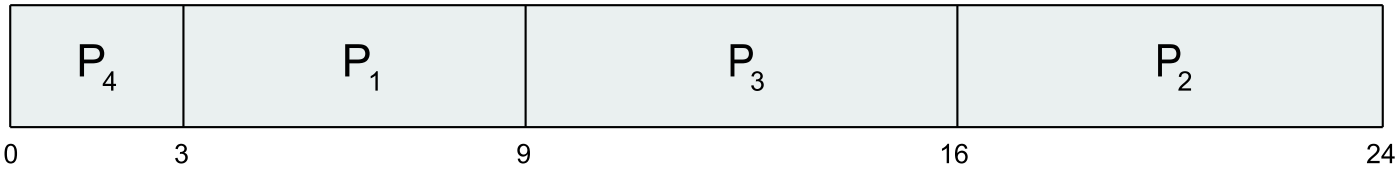
- Convoy effect: one big CPU bound process followed by several short I/O bound process
- Shorter processes wait for large process to release CPU, low CPU and device utilization
- non-preemptive, if there is a large CPU bound process lack of interaction for user

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Schedule the process with the shortest next CPU burst
  - Use FCFS scheduling to break ties

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
- What is the average waiting time if FCFS scheduling is used?

Problems with SJF

- How to know the burst time of next process in ready queue?
- What if a process arrives with lesser burst time than the one running?
- Preemptive/Non-preemptive

SJF is Optimal

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - Difficulty is knowing the length of the next CPU burst
  - Could predict the length of the next CPU burst and pick the process with the shortest predicted next CPU burst

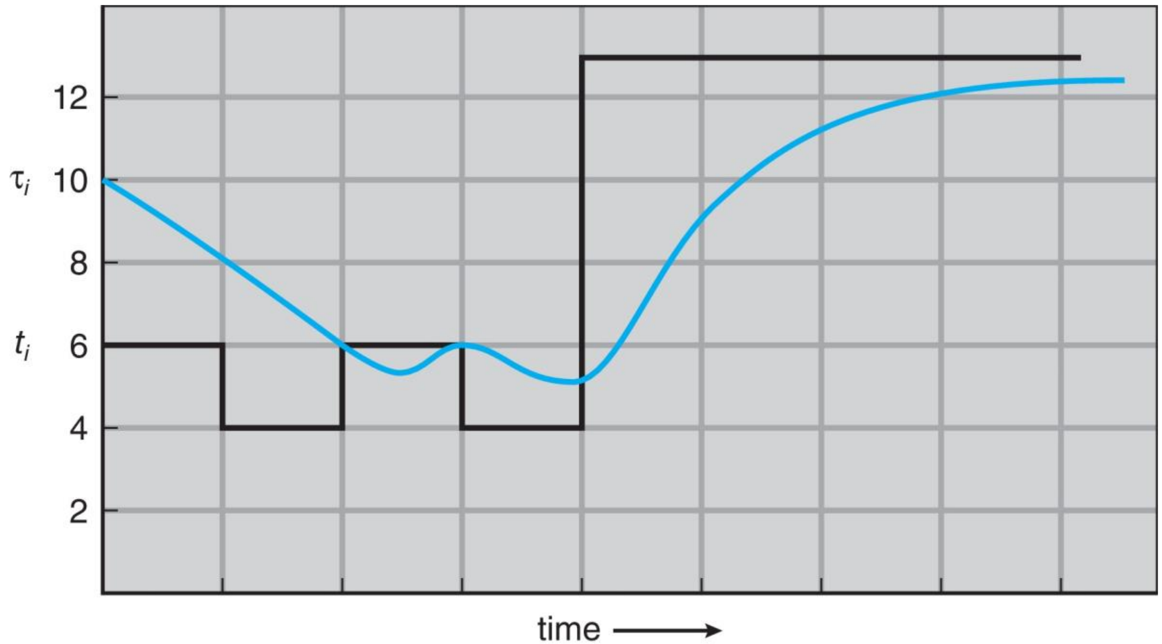
Predicting Length of Next CPU Burst

- Length of the next CPU burst should be similar to the previous ones
- Can predict the length of the next CPU burst using the length of previous CPU bursts, using exponential averaging

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

- Commonly, α set to ½
- Initial τ_0 can be a constant or overall system average

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	13	
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .
i	0	1	2	3	4	5	6	7	

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Most recent info has no effect
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

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

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

Shortest-Remaining-Time-First

- Preemptive version of SJF is called **Shortest-Remaining-Time-First**
  - If the next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, the currently executing 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

- *Preemptive* SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–26

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec
- What is the average waiting time for nonpreemptive SJF scheduling?

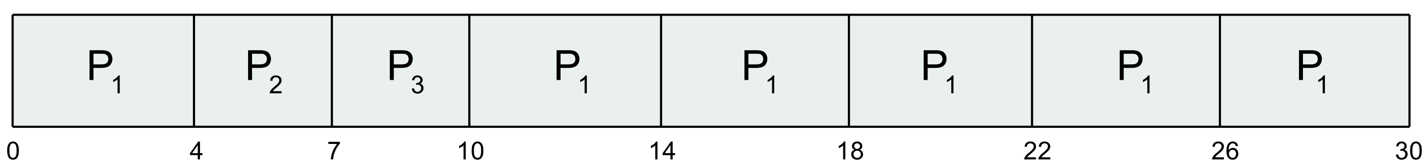
Round Robin (RR)

- RR is designed for multitasking systems to provide fast response time
- Each process gets a small unit of CPU time, called a **time quantum** or **time slice**, usually 10-100 milliseconds
  - After time quantum has elapsed, the process is preempted and added to the end of the ready queue
- RR is preemptive
- Burst time > q: process is interrupted (CPU scheduler starts a timer for q and dispatcher process in ready queue)
- Burst time < q: process releases CPU voluntarily
- n processes in ready queue, time quantum = q → 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

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



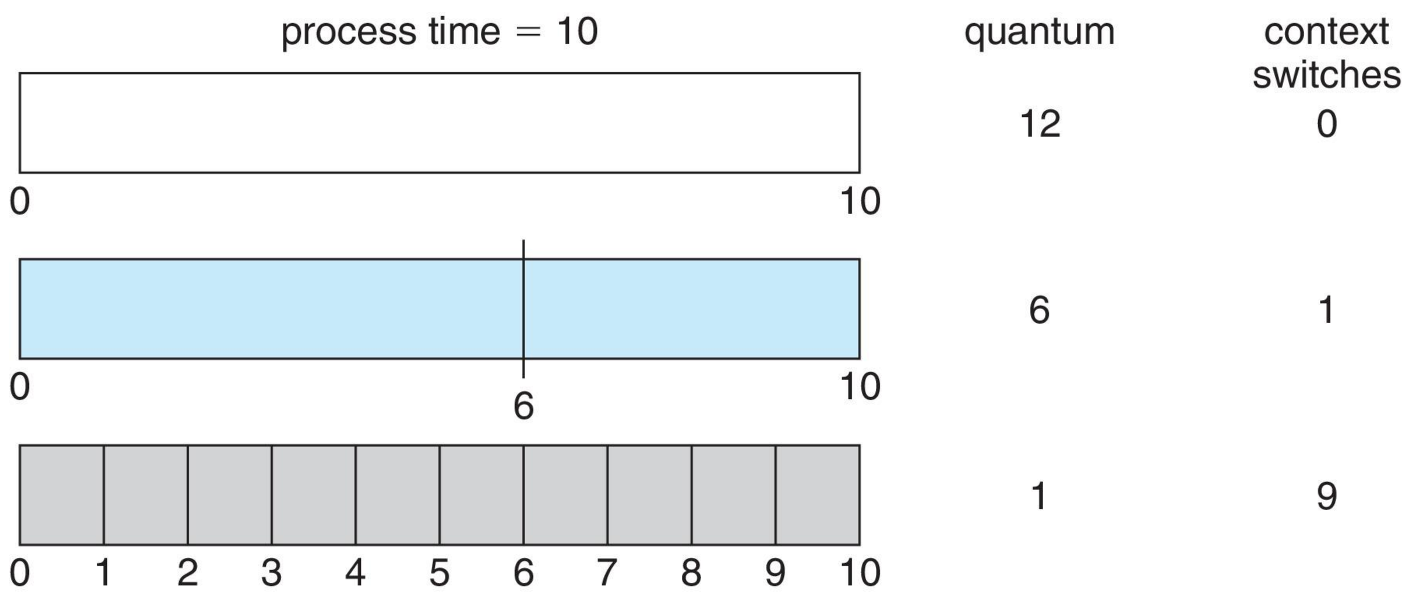
- Average turnaround time = (30+7+10)/3=15.67
- Average waiting time = ((10-4)+(4-0)+(7-0))/3 = 5.67 < (3-1)*3=6
- Typically, higher average turnaround than SJF, but better ***response time***

Problems with RR

- Choice of q can affect context switches and turn around time
- Choice of q affects waiting time
- Optimal q?

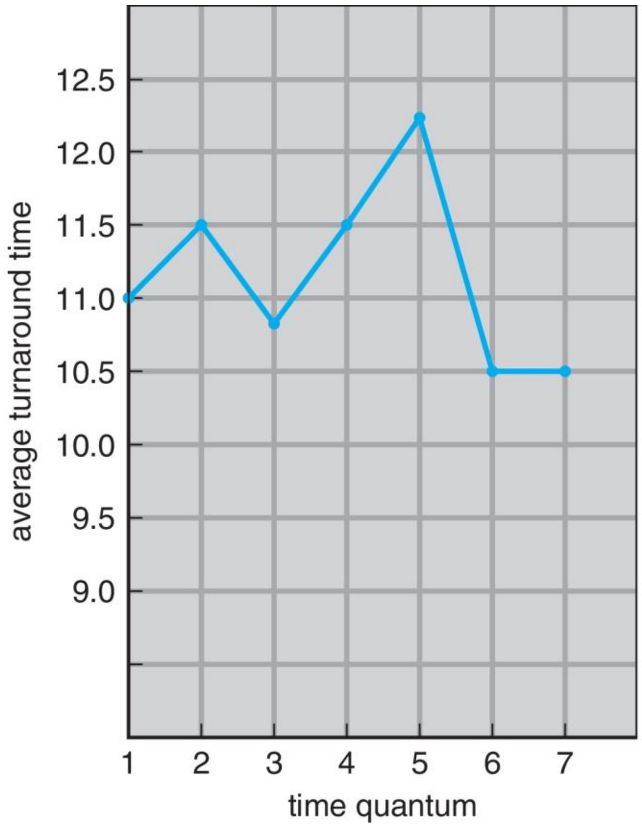
Time Quantum and Context Switch Time

- How big should be q?
  - q too large ⇒ RR same as FCFS
  - q too small ⇒ large number of context switches, high overhead



- q must be large with respect to context switch time
  - q usually 10ms to 100ms, context switch time typically < 10μs

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of thumb: 80% of CPU bursts should be shorter than q.

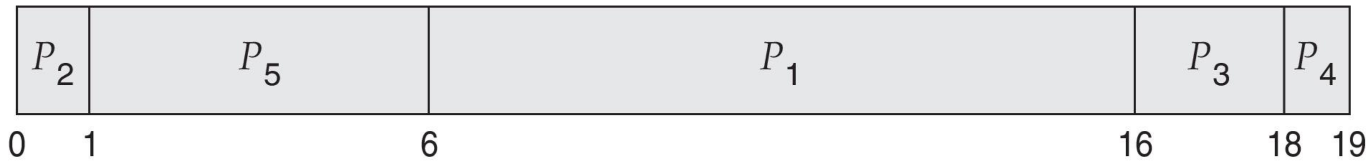
Priority Scheduling

- A priority number (integer) is associated with each process
  - We assume smallest integer ≡ highest priority
- CPU is allocated to the process with the highest priority
- Can be preemptive or nonpreemptive
  - Preemptive: preempt the currently running process if the priority of newly arrived process is higher than the priority of the currently running process
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

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 = (6+0+16+18+1)/5=8.2

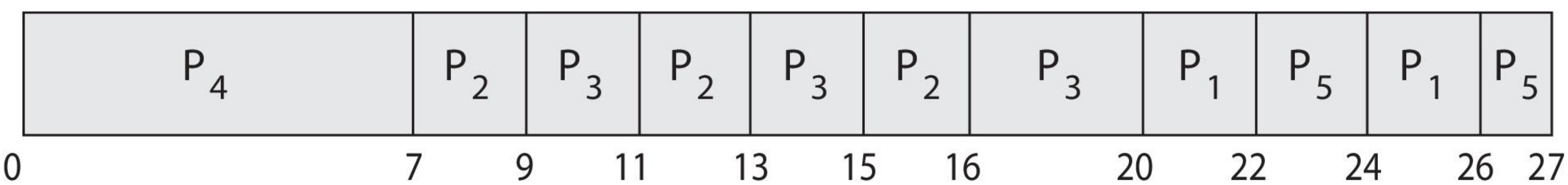
Problem of Priority Scheduling

- **Starvation** – low priority processes may never execute
- Solution:
  - **Aging** – as time progresses increase the priority of the processes that wait in the ready queue
    - E.g., increase the priority of a waiting process by 1 every second
  - Combine with RR

Priority Scheduling with Round-Robin

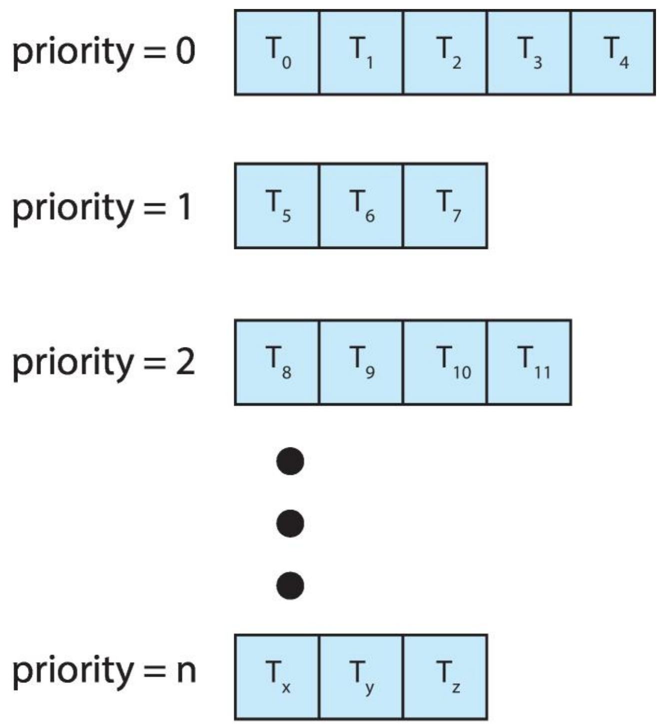
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority, and processes with the same priority run round-robin
- Gantt Chart with 2ms time quantum



Multilevel Queue Scheduling (1)

- Have separate queues for each priority
- Schedule the process in the highest-priority queue



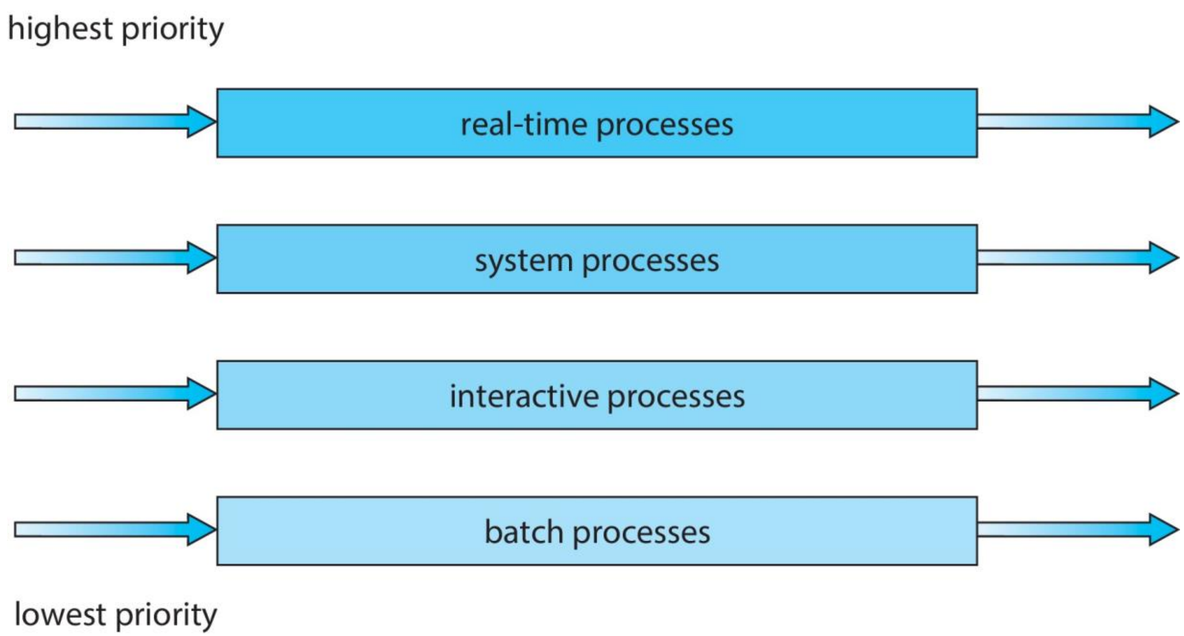
Separate queues for each priority

Multilevel Queue Scheduling (2)

- Can partition processes into separate queues based on process type
  - e.g. **foreground** (interactive) queue and **background** (batch) queue
- Each queue can have its own scheduling algorithm
  - e.g., foreground queue uses RR, background queue uses FCFS
- Scheduling among the queues
  - Fixed-priority preemptive scheduling
    - e.g., foreground queue has absolute priority over background queue
  - Time-slice among the queues
    - e.g., foreground queue gets 80% CPU time and background queue gets 20% CPU time

Multilevel Queue Scheduling Example

- Prioritization based upon process type

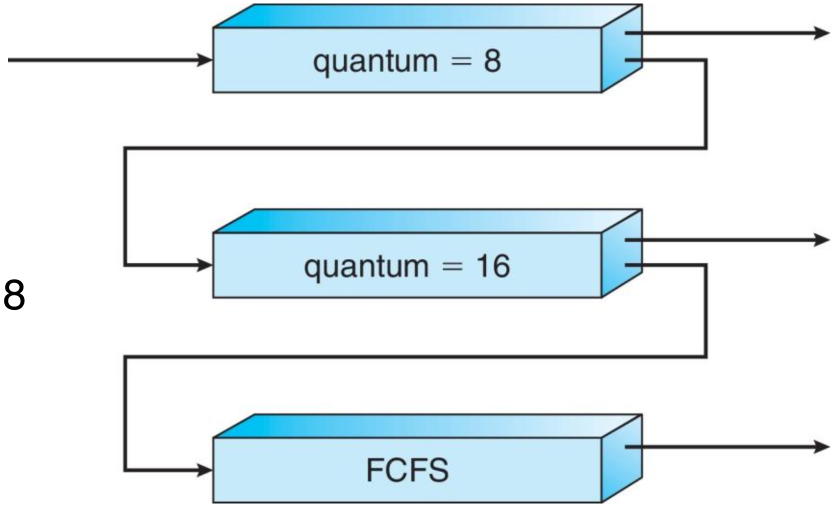


Multilevel Feedback Queue Scheduling

- A process can move between the various queues
  - If a process uses too much CPU time, it is moved to a lower-priority queue
    - This leaves I/O-bound and interactive processes in higher-priority queues
  - A process that waits too long in a lower-priority queue is moved to a higher-priority queue; this implements aging
- 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

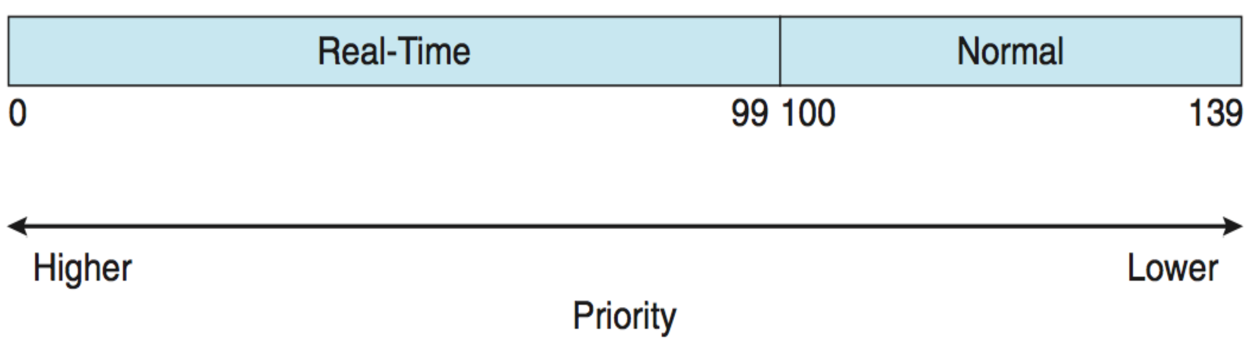
Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 ms
  - Q_1 – RR with time quantum 16 ms
  - Q_2 – FCFS
  - Q_0 highest priority, Q_2 lowest priority
- Scheduling
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8 ms
    - If it does not finish in 8 ms, job is moved to queue Q_1
  - At Q_1 job receives 16 additional ms
    - If it still does not complete, it is moved to queue Q_2
- Does this algorithm favor CPU-bound processes or I/O-bound processes?



Linux Scheduling in Version 2.6.23 +

- Priority-based, preemptive scheduling of tasks
  - Scheduler picks highest priority task in highest scheduling class
  - Lower priority value is higher priority
- Two scheduling classes
  - Real-time class: FIFO (tasks run to completion) or RR (tasks run until they exhaust a time slice)
    - Priority range 0-99, static priority
  - Normal class: **Completely Fair Scheduler** (**CFS**)
    - Each task assigned a nice value between -20 (maps to priority 100) and +19 (maps to priority 139)
    - Default nice value is 0, can be changed by user



Completely Fair Scheduler (CFS)

- CFS scheduler assigns a proportion of CPU time (i.e., a time slice) to each task
- Proportions of CPU time are allocated from a **target latency** (default is 20ms) – interval of time during which every runnable task should run at least once
  - Each runnable task gets a 1/N slice of the target latency where N is the number of runnable tasks
- Nice value is used to weight the 1/N slice
  - Lower nice value receives a larger time slice
  - Larger nice value receives a smaller time slice

CFS (continued)

- CFS scheduler records how long each task has run by maintaining per task **virtual run time** in variable **vruntime**
- If a process has run for t ms, then

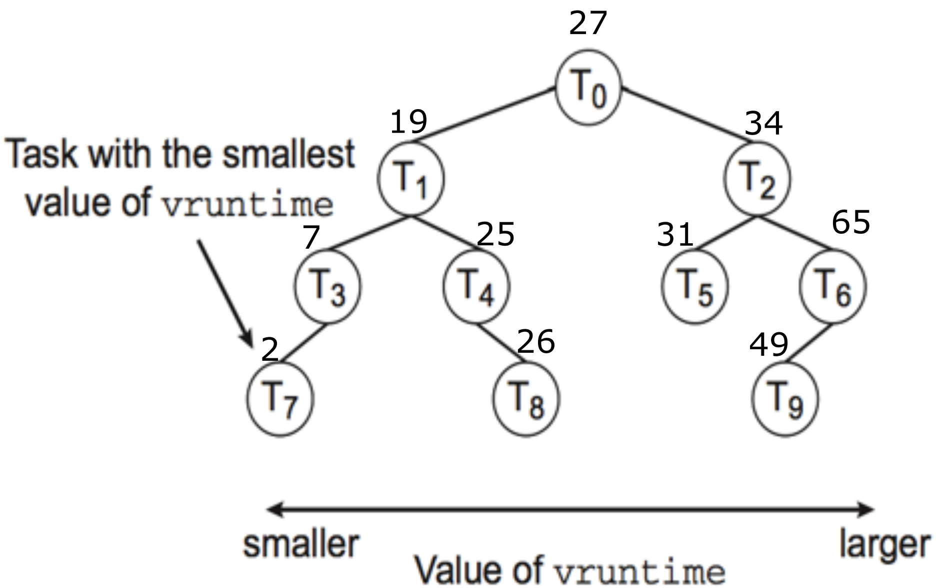
  vruntime += t * (weight based on nice value of process)

  - nice value = 0: weight =1 and virtual run time = actual run time
  - nice value < 0: weight <1 and virtual run time < actual run time
  - nice value > 0: weight >1 and virtual run time > actual run time

- When current running task uses up its time slice, scheduler picks task with lowest **vruntime** value

CFS Implementation

- Runnable tasks are organized as a red-black tree (a self-balancing binary search tree) based on **vruntime**
  - Search, insert, and delete operations take O(log N) time, where N is the number of nodes in tree
- The min **vruntime** task is the leftmost node on tree
  - Finding the leftmost node requires O(log N) time
  - Scheduler caches the leftmost node, so determining which task to run next requires constant time



I/O and CPU Bound Processes

- I/O bound processes should get higher priority compared to CPU bound processes
- CFS achieves this efficiently
  - I/O-bound processes have small CPU bursts, therefore will have a low vruntime. They would appear towards the left of the tree. Thus are given higher priorities