Chap 7, 8: Scheduling

Introduction

- **Multiprogramming**
  - Multiple processes in the system with one or more processors
  - Increases processor utilization by organizing processes so that the processor always has one to execute
  - Resource management
    - Resources for time sharing
      - Multiple processes use a resource in a time-shared manner
      - Processor
      - Process scheduling: Allocates processor time slots to processes
    - Resources for space sharing
      - Partition a resource and let each process use the partitions
      - Memory

Goals of Scheduling

- **Goals of process scheduling**
  - Improving system performance
- **Typical performance indices**
  - Turnaround time: amount of time to execute a particular process
    - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Response time: amount of time it takes to start responding
    - $T_{response} = T_{firstrun} - T_{arrival}$
  - Throughput: number of processes completed per time unit
  - Fairness
  - Utilization: Percentage of time that the resource is busy during a given interval
  - Predictability
  - Etc
- Each system selects a scheduling policy with the consideration on the performance indices for its application domain

Scheduling Policies

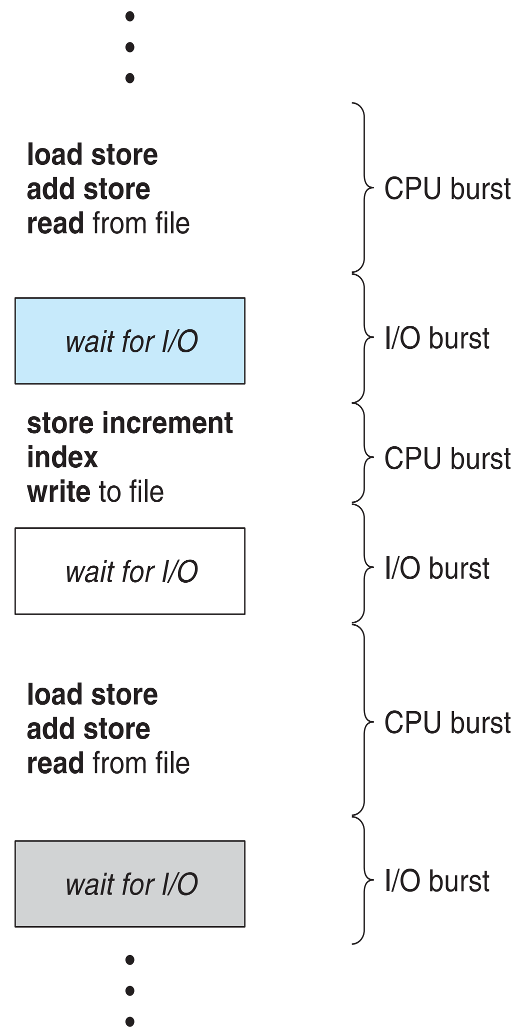
- **Preemptive/non-preemptive scheduling**
  - Preemptive scheduling
    - CPU may be preempted to another process independent of the intention of the running process
      - Flexibility, adaptability, performance improvements
    - For time-sharing systems and real-time systems
    - Incurs a cost associated with access to shared data
      → [Process synchronization]
    - Affects the design of operating system kernel
      - Kernel data integrity and consistency
      - Preemptible kernel
    - High context switching overhead

Scheduling Policies

- **Preemptive/non-preemptive scheduling**
  - Non-preemptive scheduling
    - Process uses the CPU until it voluntarily releases it (eg. for system call)
    - No preemption
    - Pros
      - Low context switch overhead
    - Cons
      - Frequent priority inversions
      - May result in longer mean response time

Terminologies

- **CPU burst vs. I/O burst**
  - Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst
    - Each cycle of CPU execution
  - I/O burst
    - Each cycle of I/O wait
  - Burst time is an important factor(criteria) for scheduling algorithms



Scheduling Schemes

- **FIFO, FCFS (First-Come First Service)**
- **SJF (Shortest Job First)**
- **STCF (Shortest Time-to-Completion First)**
- **RR**
- **Priority**
- **MLFQ**

Scheduling Schemes

- **FCFS(First-Come-First-Service) scheduling**
  - Non-preemptive scheduling
  - Scheduling criteria
    - Arrival time (at the ready queue)
    - Faster arrival time process first
  - High resource utilization
  - Adequate for batch systems, not for interactive systems
  - Disadvantages
    - Convoy effect
      - short process behind long process
      - Consider one CPU-bound and many I/O-bound processes
    - Longer mean response time

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:

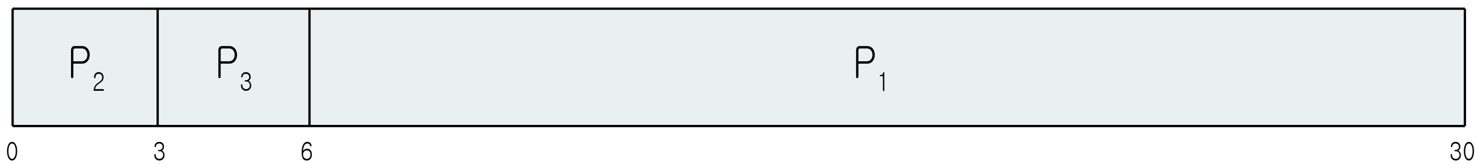
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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**

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**
  - Non-preemptive scheduling
  - Scheduling criteria
    - Burst time
    - Shortest next CPU burst time first scheduling
  - Pros
    - Gives minimum average waiting time for a given set of processes
    - Minimizes the number of processes in the system
      - Reduces the size of the ready queue
      - Reduces the overall space requirements
    - Fast responses to many processes

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**
  - Cons
    - Starvation, indefinite postponement(blocking)
      - Long burst-time processes
      - Can be solved by aging
    - No way to know the length of the next CPU burst for each process
      - It is necessary to have a scheme for burst time estimation
      - Estimation by exponential average

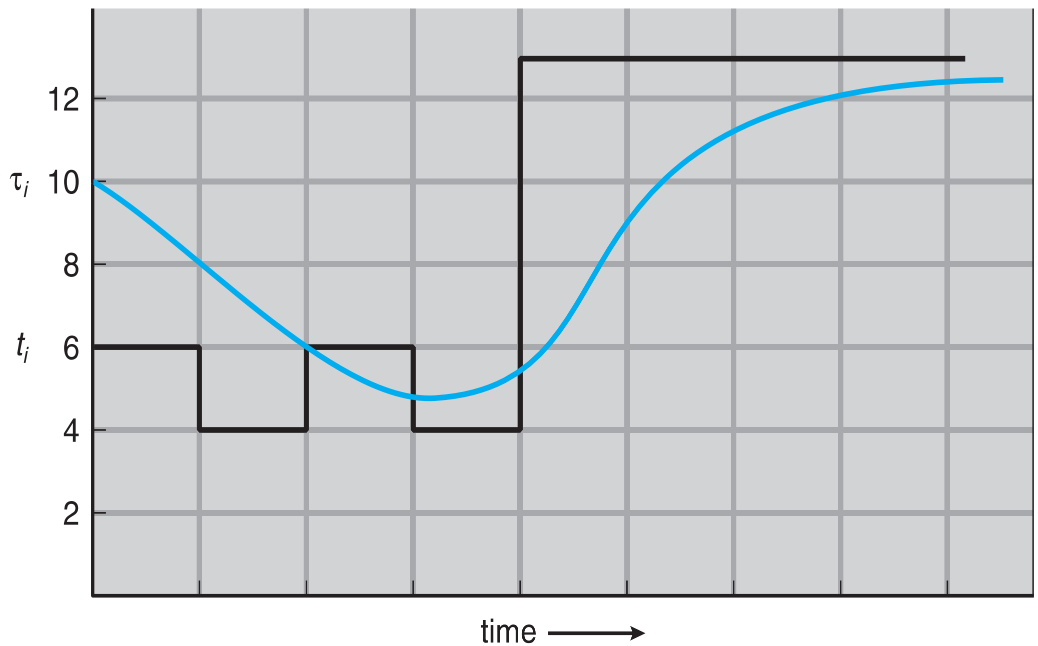
1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

Commonly, α set to ½

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**

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



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

Scheduling Schemes

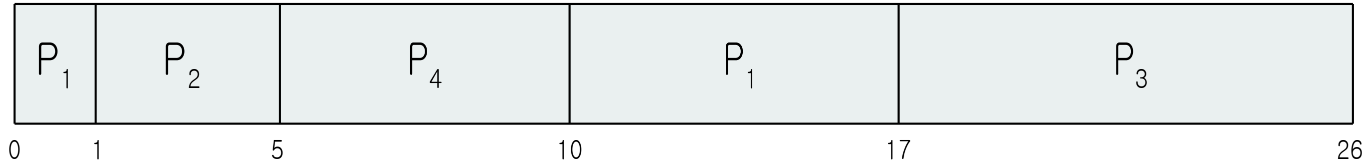
- **STCF (Shortest Time-to-Completion First) scheduling**
  - Variation of SJF scheduling (preemptive SJF)
  - Preemptive scheduling
    - Preempt current running process when another process with shorter remaining CPU burst time arrives at the ready queue
  - Cons
    - Burst time estimation overhead as in SPN
    - Overhead for tracing remaining burst time
    - High context switching overhead

Example of STCF

- 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



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

A New Metric: Response Time

- At time-shared machines, users would sit at a terminal and demand interactive performance from the system.
- **Response time:** the time from when the job arrives in a system to the first time it is scheduled
  - $T_{response} = T_{firstrun} - T_{arrival}$

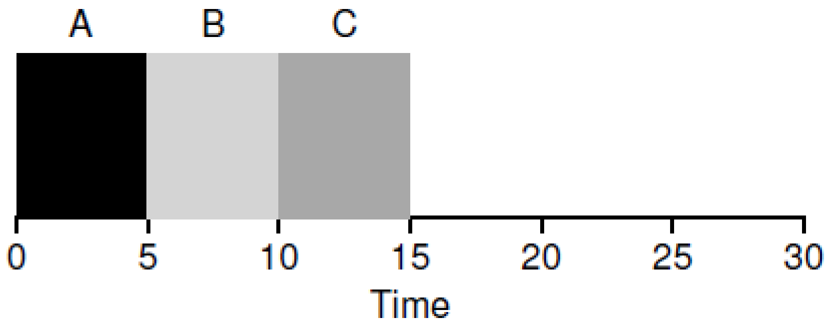


Figure 7.6: **SJF Again (Bad for Response Time)**

Scheduling Schemes

- **RR (Round-Robin) scheduling**
  - Preemptive scheduling
  - Scheduling criteria
    - Arrival time (at the ready queue)
    - Faster arrival time process first
  - Time slice (scheduling quantum) for each process
    - System parameter
    - The (running) process that has exhausted his time slice releases the CPU and goes to the ready state (timer runout)
      - Prevents monopoly of the CPU by a process
  - High context switching overhead due to preemptions
  - Adequate for interactive/time-sharing system

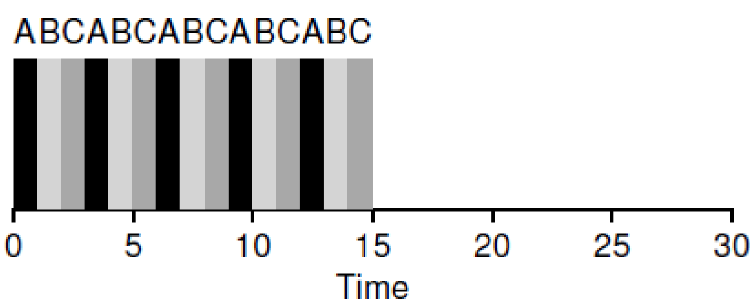


Figure 7.7: **Round Robin (Good for Response Time)**

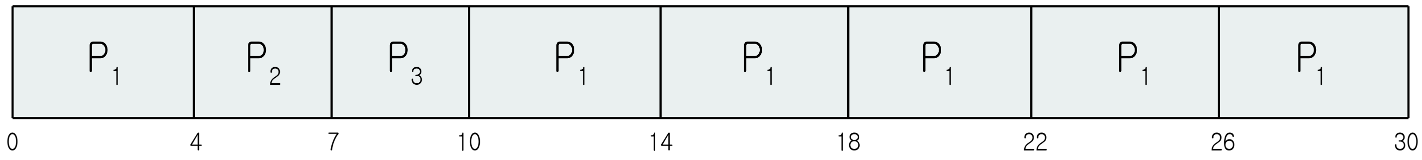
Scheduling Schemes

- **RR (Round-Robin) scheduling**
  - Performance of the RR scheme depends heavily on the size of the time slice
    - Very large (infinite) time slice → FCFS
    - Very small time slice → processor sharing
      - Appears to the users as though each of the n processes has its own processor running at 1/n the speed of the real processor
      - Better response time
      - High context switching cost
        - OS actions of saving and restoring a few registers
        - H/W flush: Cache, TLB, branch predictor
    - Deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.

Example of RR with Time Quantum = 4

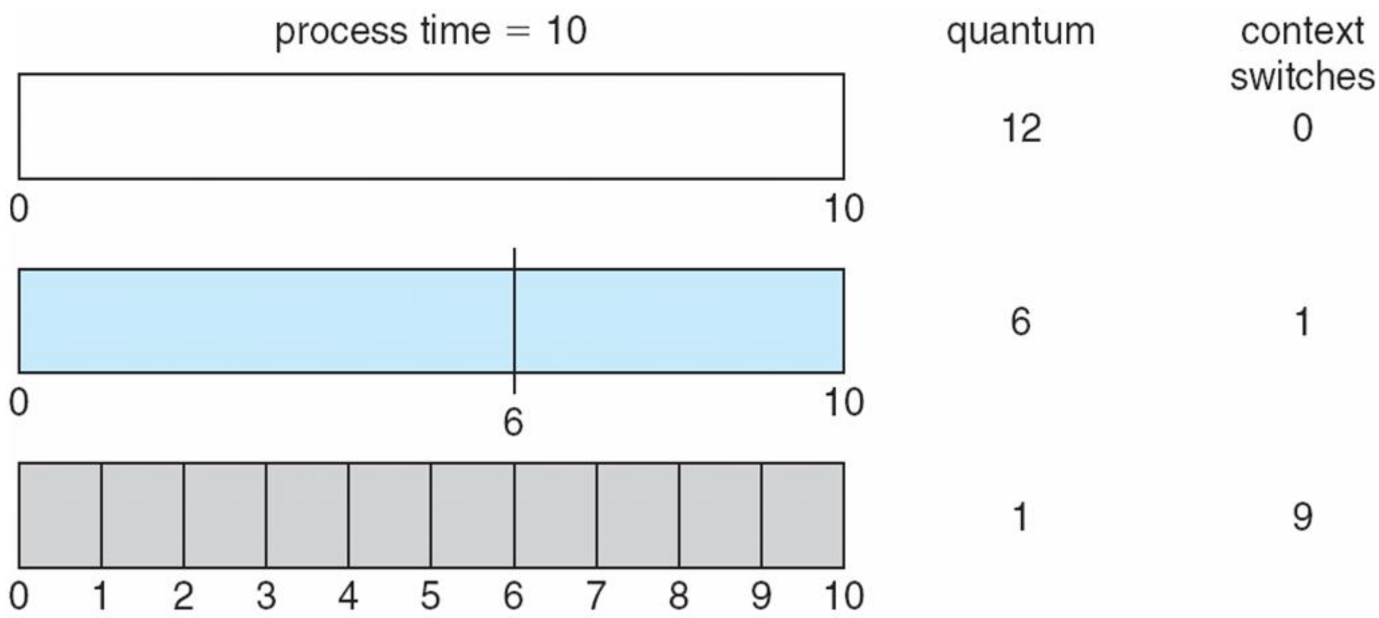
Process	Burst Time
P_1	24
P_2	3
P_3	3

- **The Gantt chart is:**

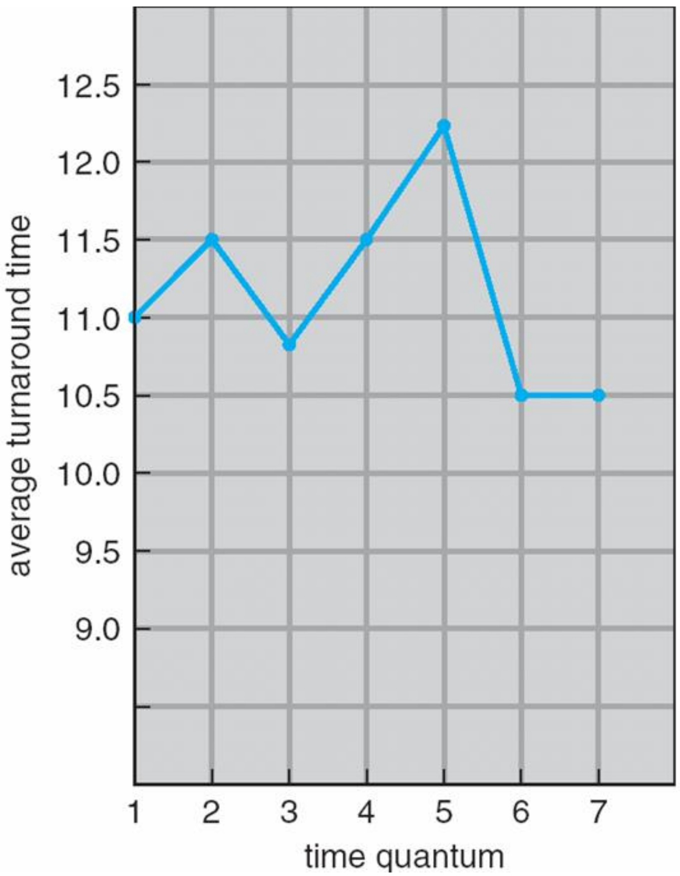


- **Typically, higher average turnaround than SJF, but better *response***
  - *RR is indeed one of the worst policies if turnaround time is our metric*
- **q should be large compared to context switch time**
- **q usually 10ms to 100ms, context switch < 10 usec**

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Incorporating I/O

- When a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is blocked waiting for I/O completion
- When the I/O completes, an interrupt is raised, and the OS runs and moves the blocked process back to the ready state.

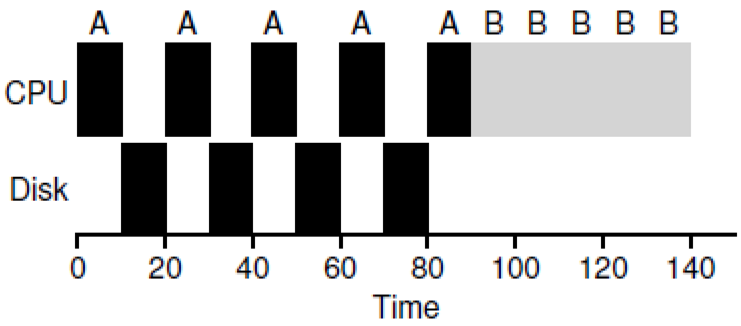


Figure 7.8: **Poor Use of Resources**

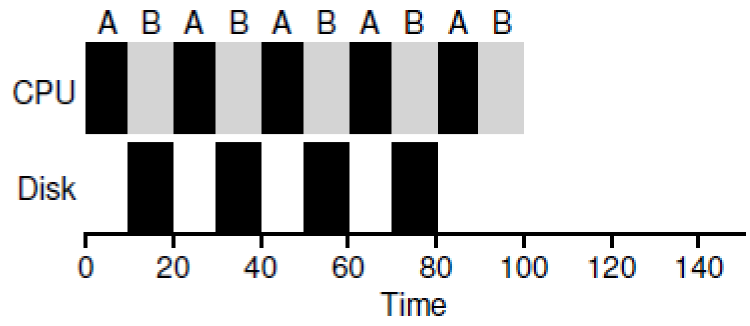


Figure 7.9: **Overlap Allows Better Use of Resources**

Treat each CPU burst as a job

Scheduling Schemes

- **Priority scheduling**
  - Scheduling criteria
    - Process priority
    - Tie breaking: FCFS
  - Priority range is different for each system
  - Mapping from the numerical value of the priority to the priority level is different for each system
  - Can be either preemptive or non-preemptive
  - Major problem
    - Starvation
    - Solution
      - **Aging** – as time progresses increase the priority of the process

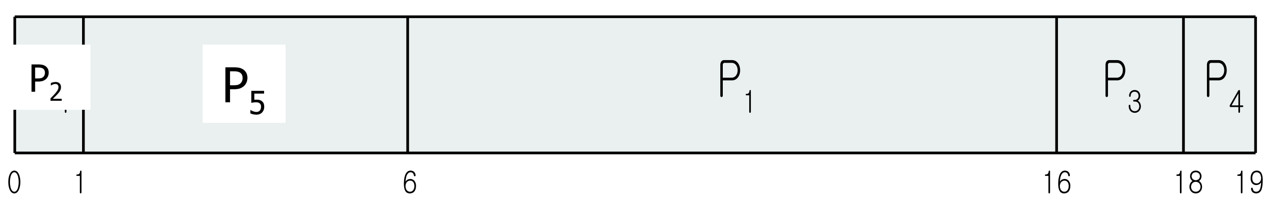
Scheduling Policies

- **Priority**
  - Classification
    - Static priority (external priority)
      - Decided at process creation time and fixed during execution of the process
      - Not adaptable to system environments
      - Simple, low-overhead
    - Dynamic priority (internal priority)
      - Initial priority at process creation time
      - May vary as the state of the system and processes changes
      - Adaptable to system environments
      - Complex, high overhead doe to priority adjustment

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

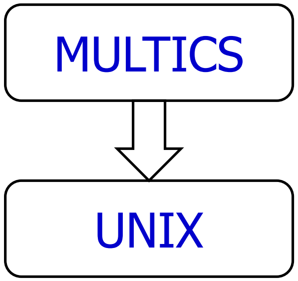
- **Priority scheduling Gantt Chart**



- **Average waiting time = 8.2 msec**

MLFQ (Multi-Level Feedback Queue)

- First described by Corbato et al. in 1962 in Compatible Time-Sharing System (CTSS) and Multics



Corbato, MIT 1965
(Turing Award 1990)

Ken Thompson
Dennis Ritchie
Bell Lab 1973
(Turing Award 1983)

- To optimize turnaround time
  - running shorter jobs first
  - Problem: SJF/STCF cannot know how long a job will run for
- To be responsive to interactive users
  - Round Robin
  - Problem: RR is terrible for turnaround time.

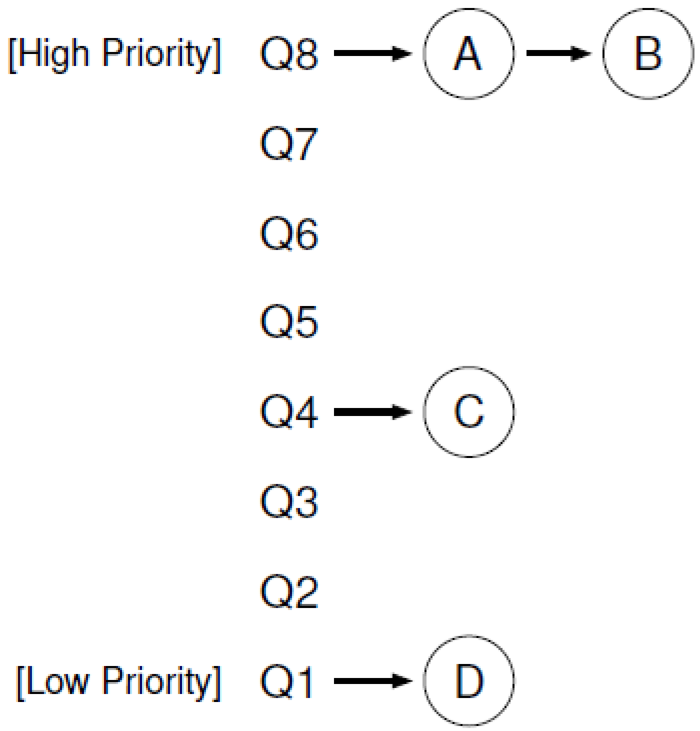
- Our problem
  - Given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals?

  ➔ learn from the past to predict the future

MLFQ: Basic Rules

- Multiple separate ready queues, each assigned a different priority.
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.

- Interactive process
  - Repeatedly relinquishes the CPU while waiting for input
  - High priority
- Batch process (CPU-bound)
  - Uses the CPU intensively for long periods of time
  - Low Priority



Attempt #1: How To Change Priority

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.

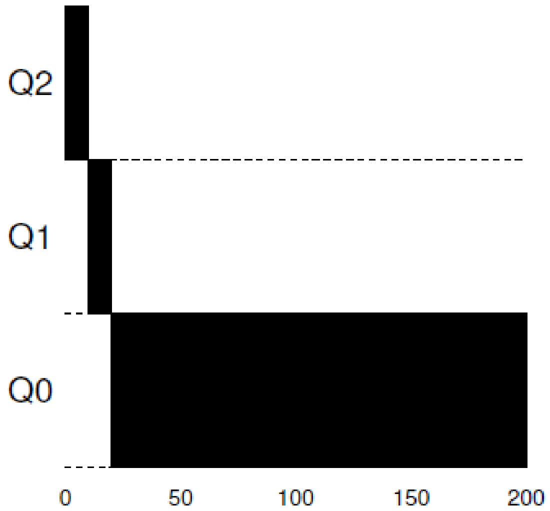


Figure 8.2: **Long-running Job Over Time**

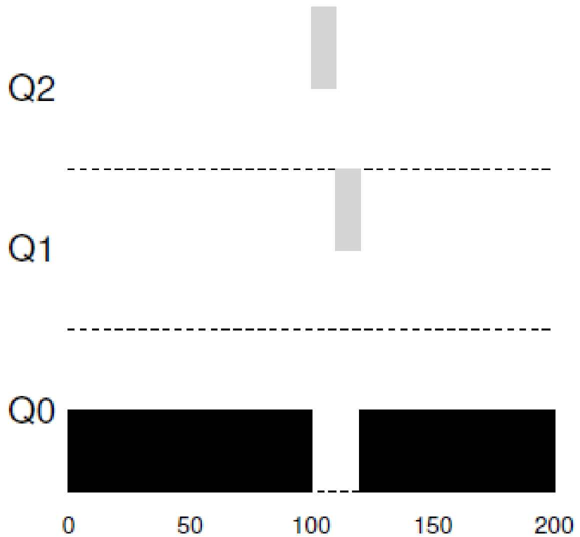


Figure 8.3: **Along Came An Interactive Job**

Problems With Our Current MLFQ

- **Starvation**
  - if there are "too many" interactive jobs in the system, long-running jobs will never receive any CPU time (they starve).
  - ➔ Need Priority Boost
- **Gaming the scheduler**
  - a smart user could rewrite their program
  - before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU
- Program may **change its behavior** over time

Attempt #2: The Priority Boost

- **Rule 5:** After some time period *S*, move all the jobs in the system to the topmost queue.
  - Prevent starvation and detect the change of behavior
- **Aging** is also a choice
  - Processes that have long waiting time moves up in the queue hierarchy

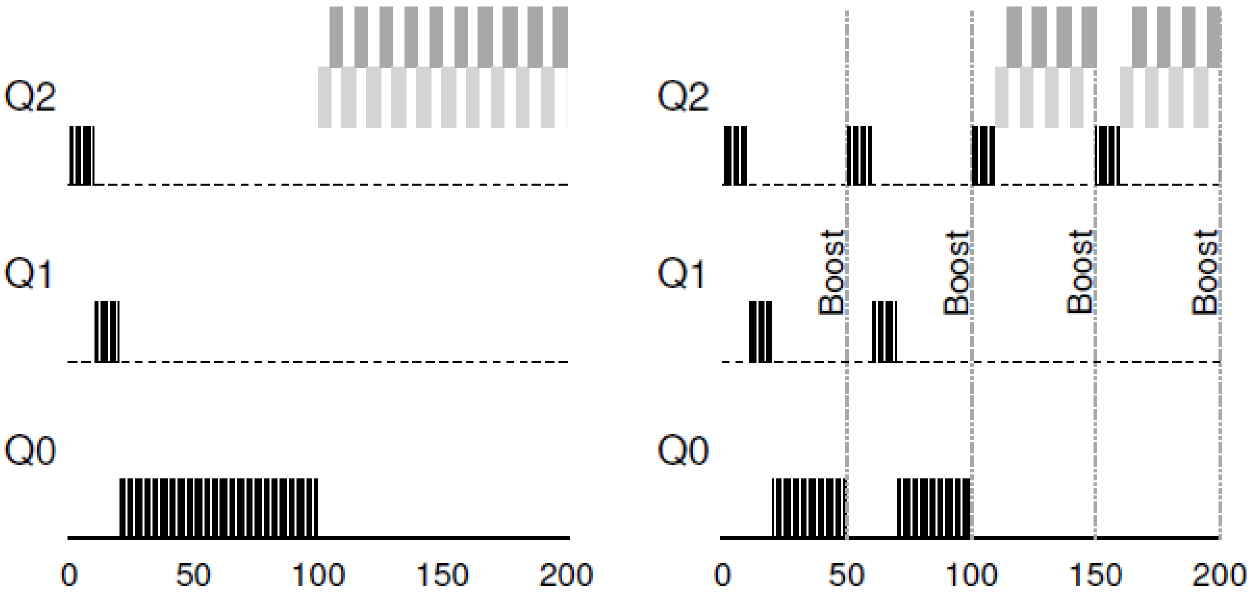


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

Attempt #3: Better Accounting

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
  - Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track

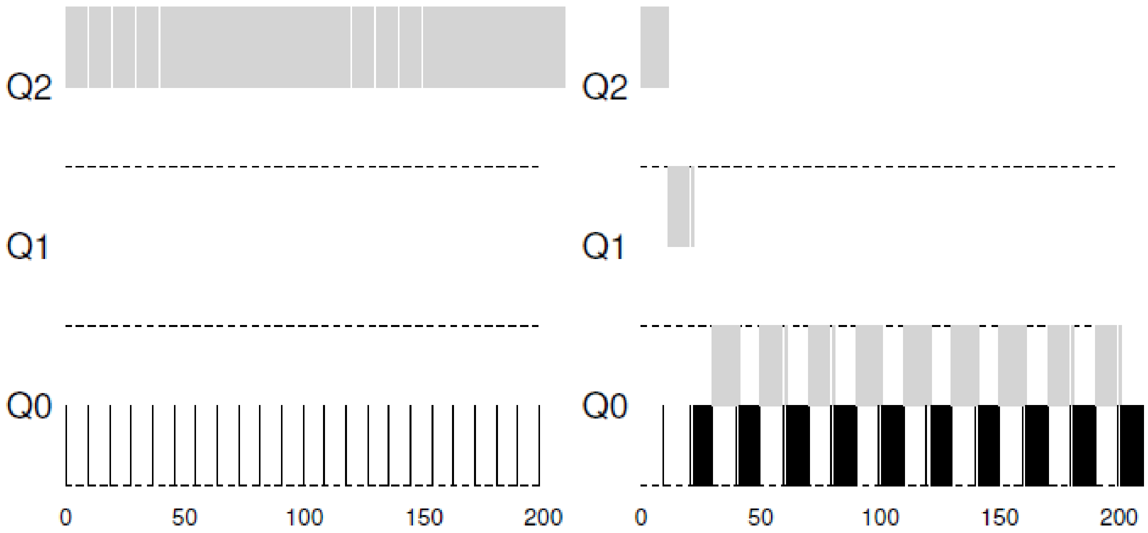
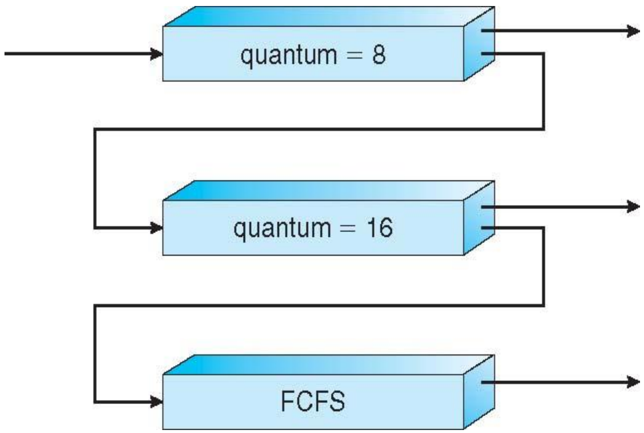


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

Attempt #4: Different Time Slice

- **Three queues:**
  - Q_0 – RR with time quantum 8ms
  - Q_1 – RR with time quantum 16ms
  - Q_2 – FCFS
- **Scheduling**
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8ms
    - If it does not finish in 8ms, job is moved to queue Q_1
  - At Q_1 job receives additional 16ms
    - If it still does not complete, it is preempted and moved to queue Q_2



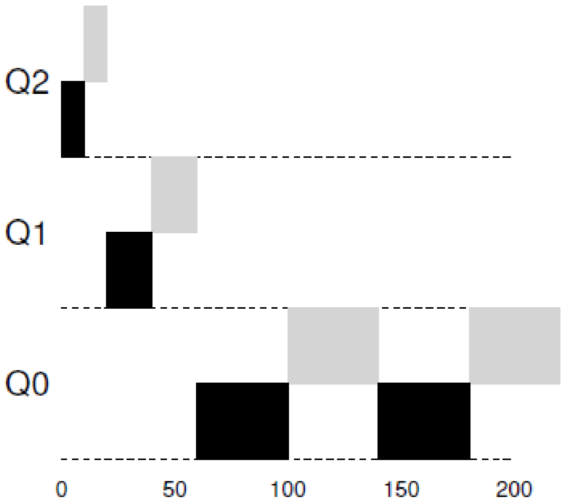


Figure 8.7: Lower Priority, Longer Quanta

Parameters for MLFQ scheduling

- The number of queues
- The scheduling algorithm for each queue
- The time slice of each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

- Easy Configuration
  - Provides a set of tables that determine exactly how the priority of a process is altered, how long each time slice is, and how often to boost the priority of a job (Solaris)
  - Uses a formula to calculate the current priority level of a job (FreeBSD)