CSC 112: Computer Operating Systems
Lecture 3

Scheduling

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

Overview

- Metrics to evaluate scheduling algorithms
- Goals of Scheduling algorithms
- Different scheduling algorithms,
  - First in first out (FIFO)
  - Shortest job first (SFJ)
  - Shortest time-to-complete first (SCTF)
  - Round-robin (RR)

Scheduling

- **Scheduling** is an important problem in many contexts, e.g., logistics, airports, game schedule, etc.

- **Scheduling**: policies that OS employs to determine the execution order of ready processes

- **Scheduling algorithms/schedulers** have diverse objectives and demonstrate different effects on the system performance

Terminologies

- Scheduling metrics and goals:
  - **CPU utilization:** percentage of time CPU is busy executing jobs
  - **Throughput:** the number of processes completed in a given amount of time
  - **Turnaround time:** the time elapses between the arrival and competition of a process
    - $T_t = T_c - T_a$
  - **Waiting time：** the time a process spends in the ready queue
  - **Response Time：** the time elapses between the process' arrival and its first output
- Maximize: CPU utilization, throughput
- Minimize: Turnaround Time, waiting time, and response time

Workload Assumptions

- Each job runs for the same amount of time
- All jobs arrive at the same time
- All jobs only use the CPU (no I/O)
- Run-time of each job is known
- Once started, each job runs to completion (non-preemption)

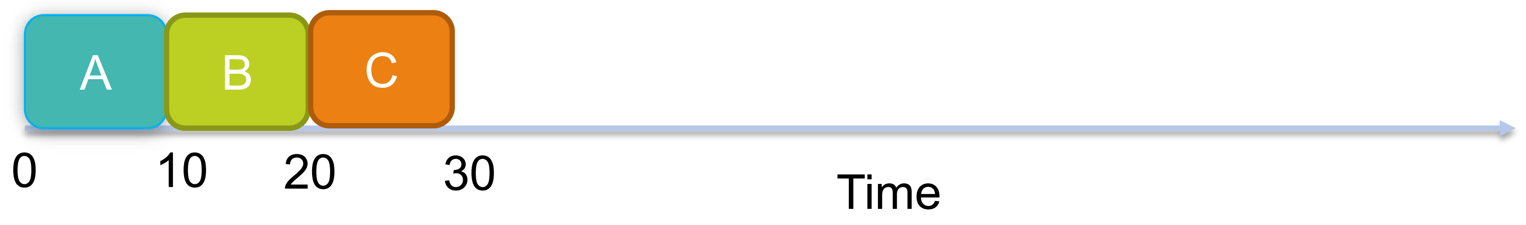
First In First Out (FIFO)

- A simple and basic scheduling algorithm, also called First Come First Served (FCFS)
- Jobs are executed in **arrival time order**

Job	Arrival time (ms)	Run time (ms)
A	0	10
B	0	10
C	0	10

- A first, B slightly later, and then C

Gantt Chart



First In First Out (FIFO)

Job	Arrival time (ms)	Run time (ms)	Finishing time (ms)
A	0	10	10
B	0^{+}	10	20
C	0^{++}	10	30

- $T_t = T_c - T_a$

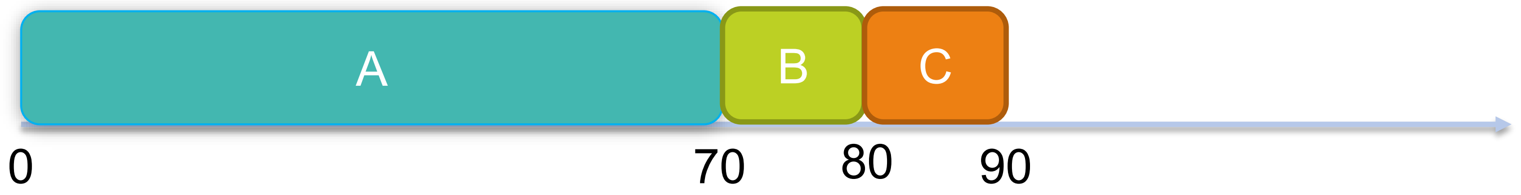
- **A: 10-0 = 10**
- **B: 20-0 = 20**
- **C: 30-0 = 30**
- **Average turnaround time = (10 + 20 + 30) / 3 = 20**

First In First Out (FIFO)

~~Each job runs for the same amount of time~~

Job	Arrival time (ms)	Run time (ms)	Finishing time (ms)
A	0	~~10~~ -> 70	
B	0^{+}	10	
C	0^{++}	10	90

Gantt Chart



Time

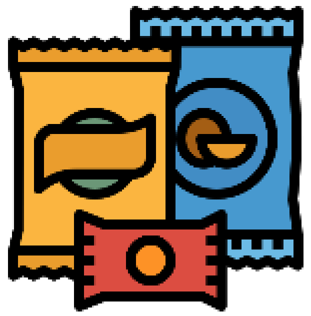
- Average turnaround time = (70 + 80 + 90) / 3 = 80

First In First Out (FIFO)

- Convoy effect
  - A number of jobs queued up behind one long job to finish execution, causing overall device and CPU utilization to be suboptimal.

Long job

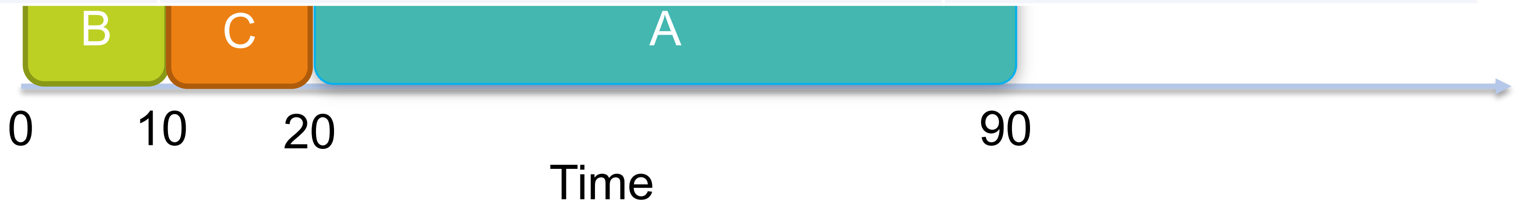
Short job


Short job

Shortest Job First (SJF)

- How to minimize average turnaround time? Run short jobs before long ones.
- SJF: jobs with the shortest execution time is scheduled first
  - Analogy: at supermarket checkout, Let people with few items jump the queue to cut in front of people with many items. It is not fair for each person, but more efficient from the supermarket (OS) perspective

Job	Arrival time (ms)	Run time (ms)	Finishing time (ms)
A	0	70	90
B	0	10	10
C	0	10	20
		Average Turnaround	40



- Average Turnaround time = (10 + 20 + 90) / 3 = 40 (vs. **80** for FIFO)

Shortest Job First (SJF)

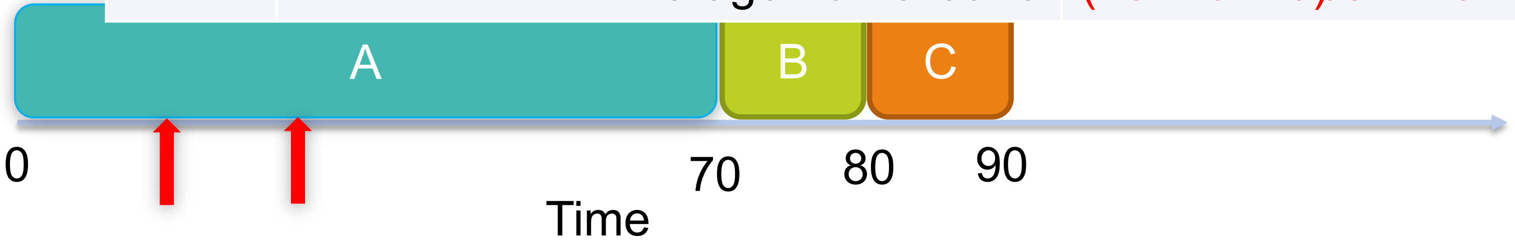
- SJF is an **optimal** scheduling algorithm in terms of **average waiting time** given these assumptions:
  - All jobs arrive at the same time
  - All jobs only use the CPU (no I/O)
  - Run-time of each job is known
  - Once started, each job runs to completion (non-preemption)

Shortest Job First (SJF)

- SJF is no longer optimal if not all jobs arrive at the same time:

~~All jobs arrive at the same time~~

Job	Arrival time (ms)	Run time (ms)	Finishing time (ms)
A	0	70	70
B	10	10	80
C	20	10	
		Average Turnaround	(70+70+70)/3 = 70



B and C are ready here, but cannot be scheduled due to non-preemption

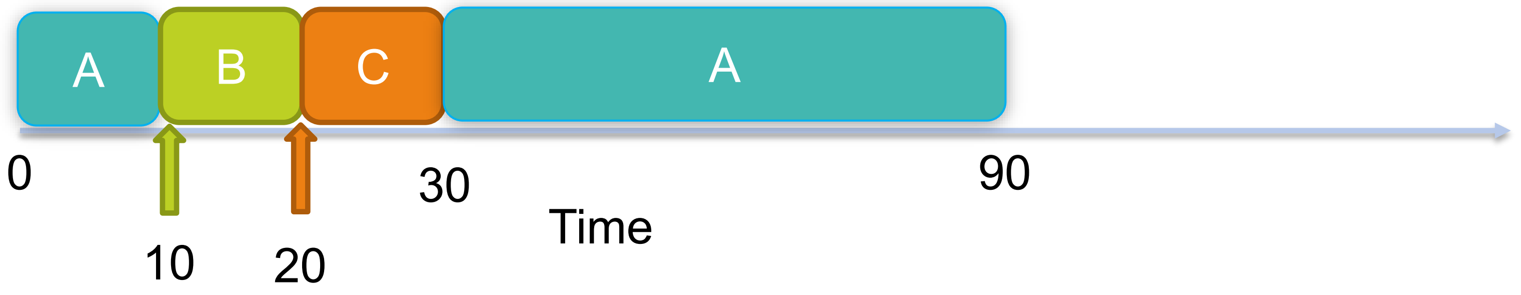
Shortest Time-to-Complete First (STCF)

~~Once started, each job runs to completion~~

- This assumption indicates the concept of **non-preemption**
- FIFO and SJF are both **non-preemptive** schedulers
- STCF policy: Always switch to jobs with the shortest completion time
- STCF is a **preemptive** scheduler
- STFC is optimal in terms of minimizing the average waiting time, if the assumptions hold.

Shortest Time-to-Complete First (STCF)

Job	Arrival time (ms)	Run time (ms)	Finishing time (ms)
A	0	70	90
B	10	10	20
C	20	10	30
		Average Turnaround	



- STCF may cause **starvation**
- How about other metric?

Response Time

- **Response time** refers to the time between a process's arrival in the ready queue and its first execution on the CPU. This metric is crucial for evaluating the responsiveness of the system, especially for interactive applications

Job	Arrival time (ms)	Run time (ms)
A	0	30
B	0^+	20
C	0^{++}	10

- FIFO
  - R = (0 + 30 + 50) / 3 = 26.7
- SJF
  - R = (0 + 10 + 30) / 3 = 13.3

- A better scheduler for response time? Round-Robin (RR)

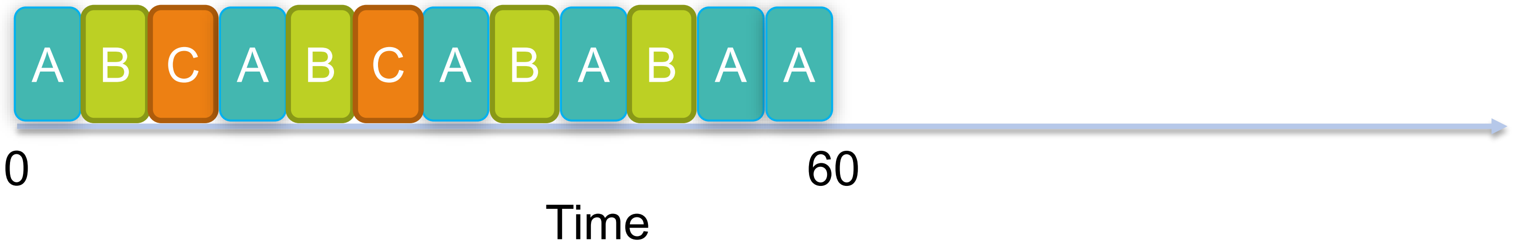
Round-Robin (RR)

- Concept of **time quantum/time slice/scheduling quantum:** a fixed and small amount of time units
- Each process executes for **a time slice**
- Switches to another one **regardless whether it has completed its execution or not**
- If the job has not yet completed its execution, the incomplete job is added to the tail of the ready queue, FIFO queue.

Round-Robin (RR)

Job	Arrival time (ms)	Run time (ms)
A	0	30
B	0^+	20
C	0^{++}	10

- Assume that time slic



R = (0 + 5 + 10) / 3 = 5

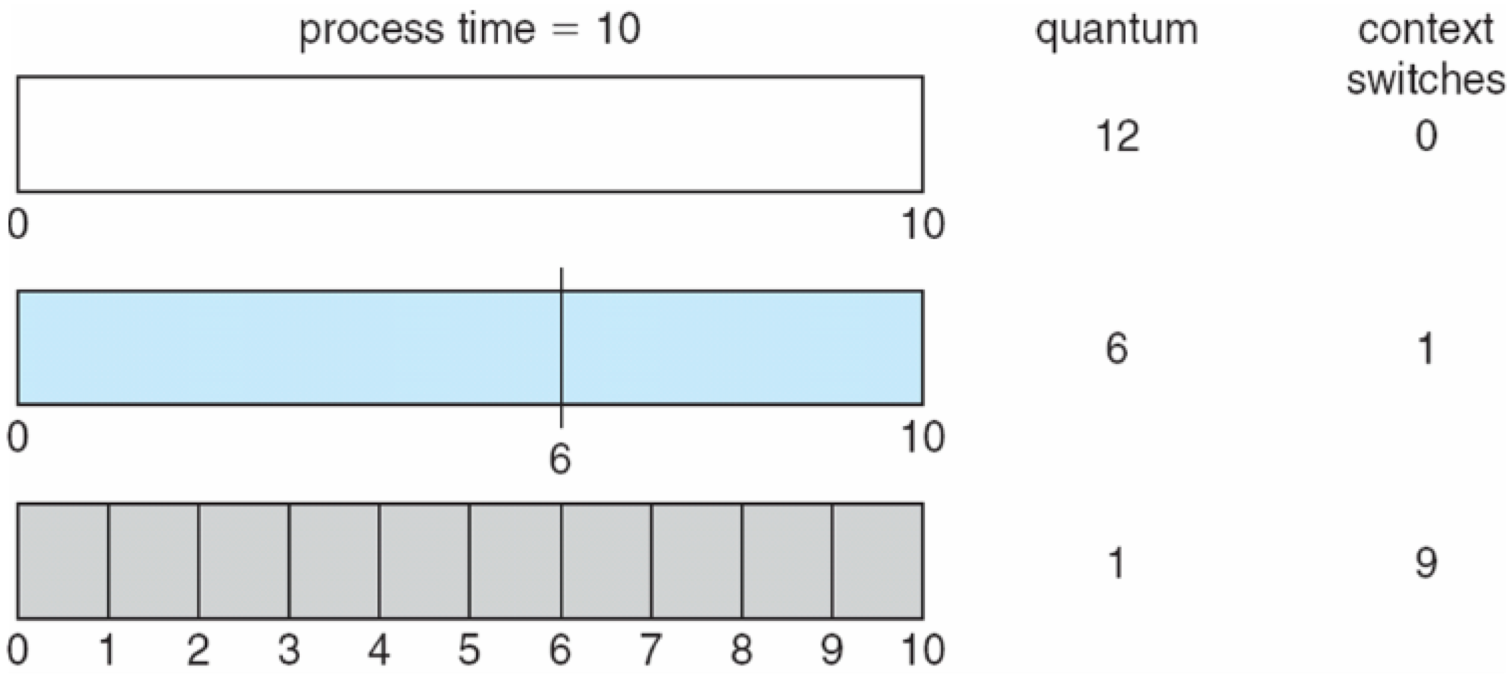
Round-Robin (RR)

Job	Arrival time (ms)	Run time (ms)	Time quantum
A	0	5	1
B	0	5	
C	0	5	

- RR
  - R = (0 + 1 + 2) / 3 = 1
- RR is a good scheduler in terms of short **response time**
- But poor in terms of **turnaround time**
  - FIFO: T = (5 + 10 + 15) / 3 = 10
  - SJF: T = (5 + 10 + 15) / 3 = 10 (same as FIFO since each job has the same run-time)
  - RR: T = (13 + 14 + 15) / 3 = **14**

Round-Robin (RR)

- If there are **n** processes in the ready queue and the time quantum is **q**, no process waits more than **(n-1)q** time units
- The selection of time quantum size should be carefully considered (usually 10-100 milliseconds)
  - Switching between processes incurs some overhead, i.e., context-switch time
  - Turnaround time depends on the size of time quantum



Round-Robin (RR)

- RR is a starvation-free scheduler
  - STCF may cause starvation: some processes may never get scheduled, e.g., processes with long execution times
- RR is fair, simple, and easy to implement and thus is used in modern OSs, such as Linux, Windows, and MacOS.

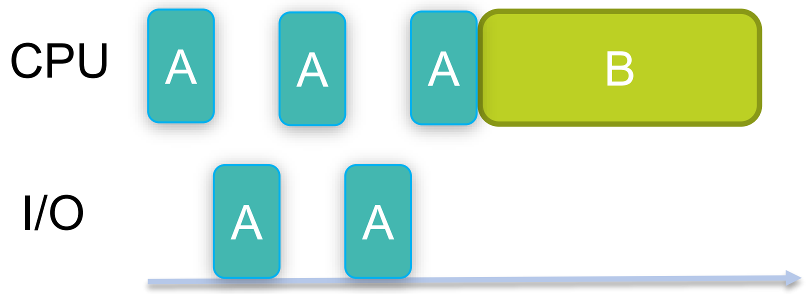
Summary

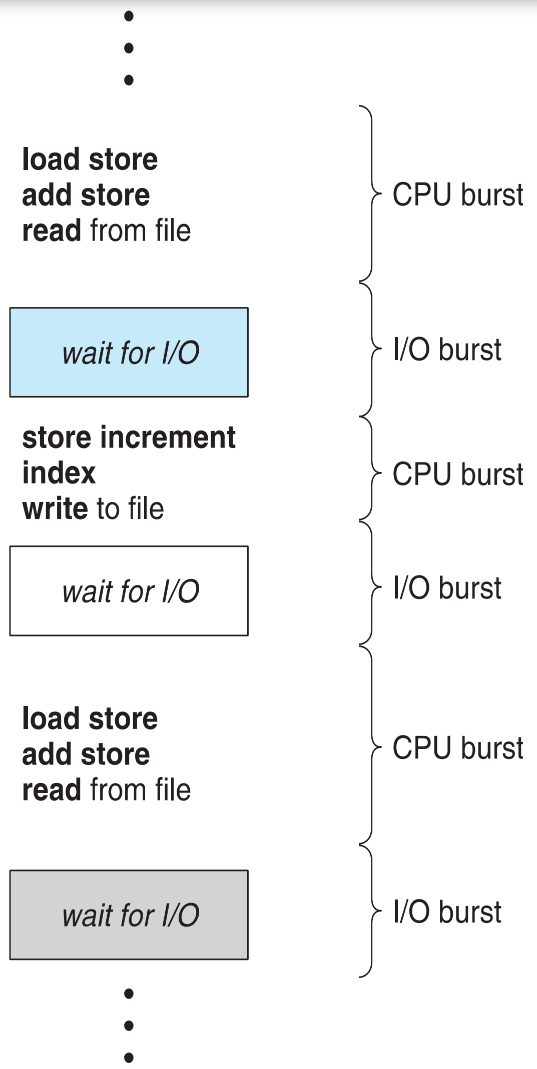
- There are different metrics to evaluate the performance of scheduling algorithms.
- FIFO is a simple scheduler, but suffers from the convoy effect
- SJF is an optimal scheduling algorithm in terms of minimizing average waiting time, if all jobs start simultaneously and know their execution times in prior
- STCF is an optimal scheduling algorithm in terms of minimizing average turnaround time, if all jobs know their execution times in prior
- RR is a better algorithm in terms of minimizing average response time

Incorporating I/O

~~All jobs only use the CPU (no I/O)~~

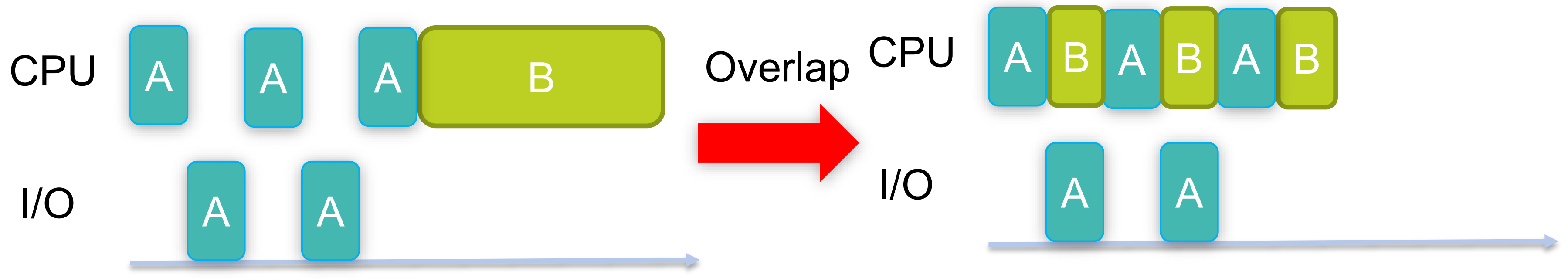
- Every program uses I/O
- Process execution consists of
  - CPU execution (CPU burst)
  - I/O wait (I/O burst)





Incorporating I/O

- Treating each CPU burst as a sub-job
  - Schedule a CPU burst
  - Initialize the subsequent I/O burst, when the CPU burst completes
  - Switch to another process



Run-time of each job is known

Priority-Based Scheduling

~~Run-time of each job is known~~

- A **priority** level (integer) is assigned to each process
  - Larger number indicates higher priority (typically)
  - FIFO, SJF, STCF are special priority-based scheduling algorithms
- The process with the highest priority is always scheduled
- Priority-Based Scheduling:
  - Preemptive or non-preemptive
- Different priority assignment methods
  - Internal or external
- Priority-based scheduling may suffer from **starvation** **(solution: aging)**

Multi-Level Feedback Queue (MLFQ)

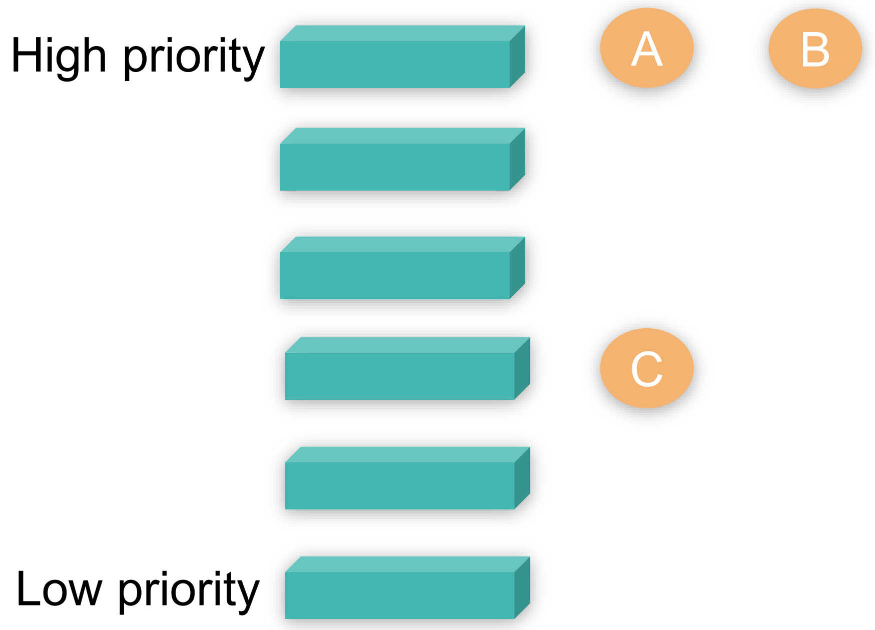
- SJF and STCF are good for turnaround time, but poor for response time
- In contrast, RR are good for response time, but terrible for turnaround time

- We need a scheduling algorithm that
  - Optimizes turnaround time for batch programs
  - Minimizes response time for interactive programs

- Multi-level Feedback Queue (MLFQ) combines
  - **Priority-based scheduling**
  - **RR**

Multi-Level Feedback Queue (MLFQ)

- MLFQ maintains a number of **queues (multi-level queue)**
  - Each queue has a different **priority level**
  - Jobs which are on the same queue have same priority
- Jobs are assigned to a queue



- **Rule 1**: If priority(A) > Priority(B), A runs
  - A&B are scheduled before C
- **Rule 2**: If priority(A) == Priority(B), A & B run with RR

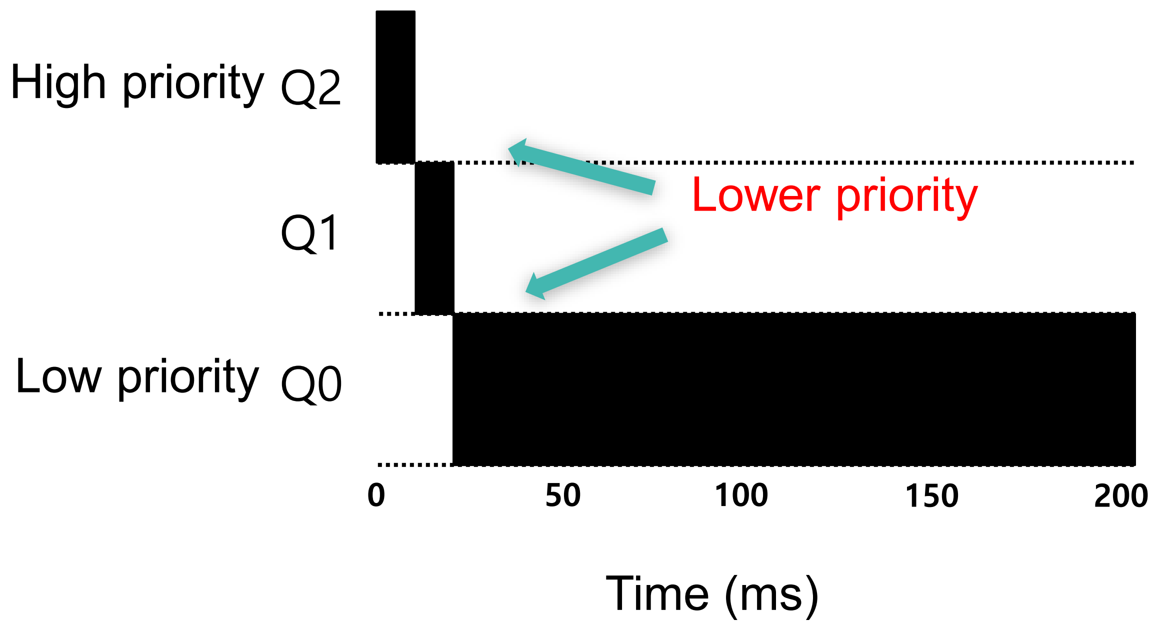
Multi-Level Feedback Queue (MLFQ)

- MLFQ **varies** the priority of a job instead of having a fixed priority
- MLFQ **varies** the priority of a job based on **its observed behavior**

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

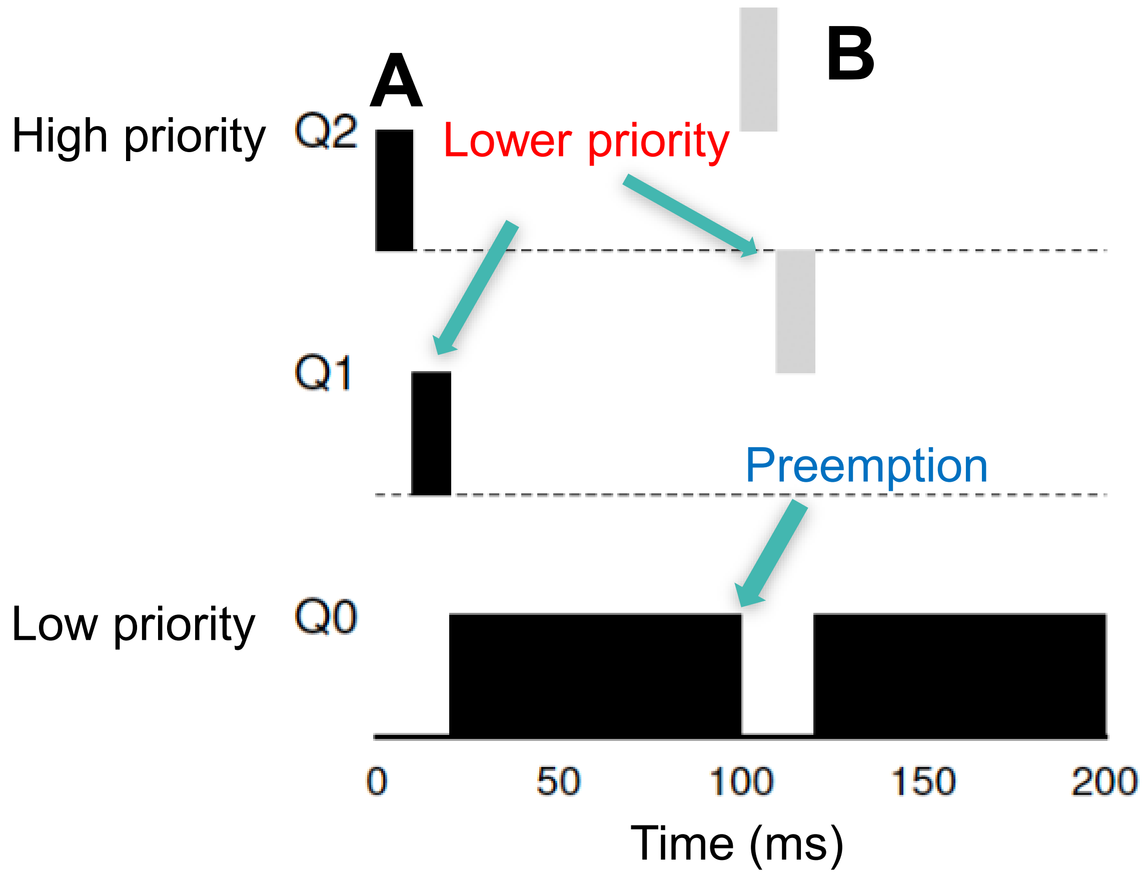
Multi-Level Feedback Queue (MLFQ)

- Rule 3 and 4a (starts at the highest, and lower if not finished)
- A long running job; Time slice is 10ms



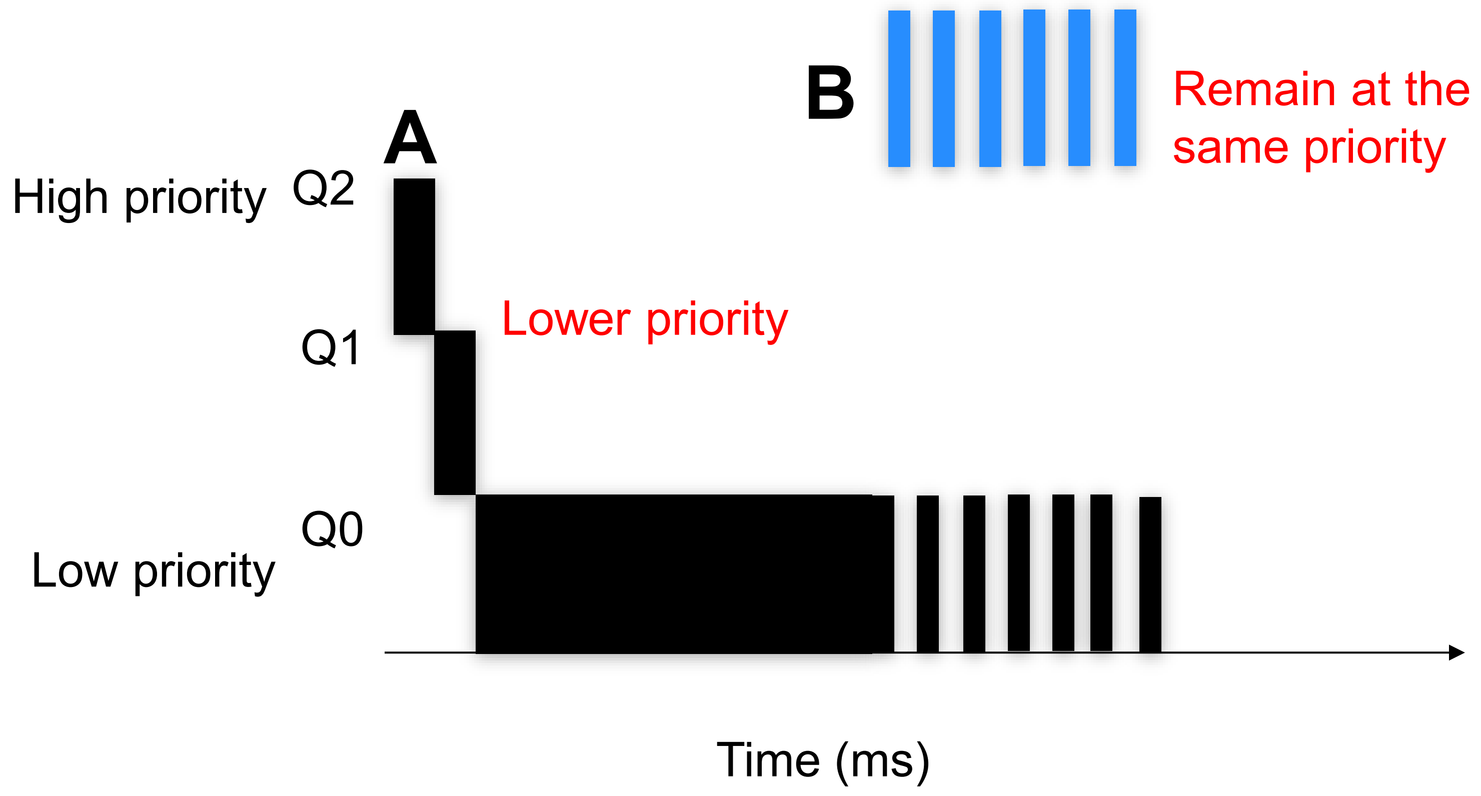
Multi-Level Feedback Queue (MLFQ)

- **Job A**: A long-running CPU-intensive job
- **Job B**: A short-running interactive job (20ms runtime), arrives at 100ms



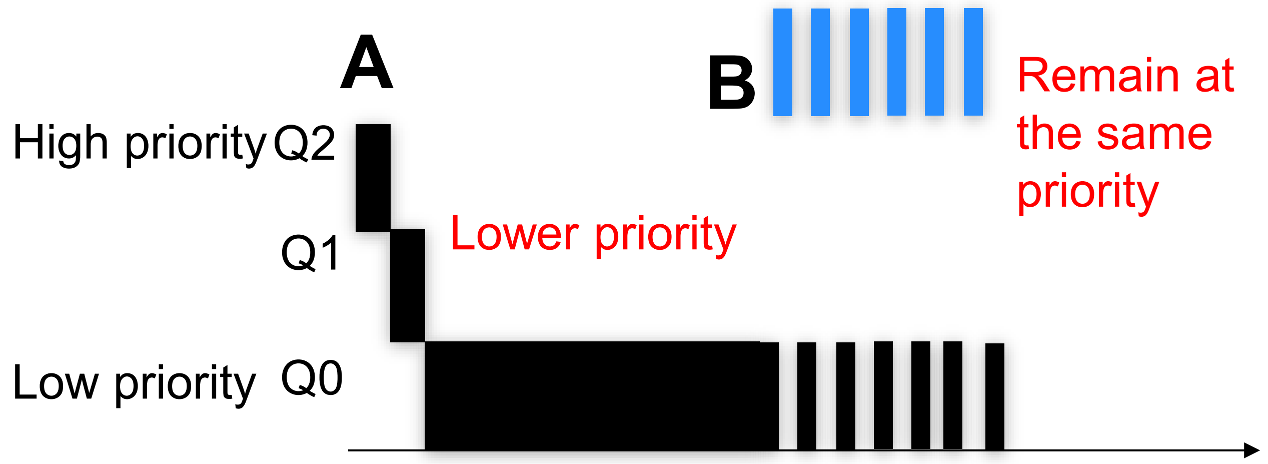
Multi-Level Feedback Queue (MLFQ)

- Rule 4b (stays at the same priority);
- **Job A**: A long-running CPU-intensive job
- **Job B**: An interactive job that only needs CPU for 5ms before I/O



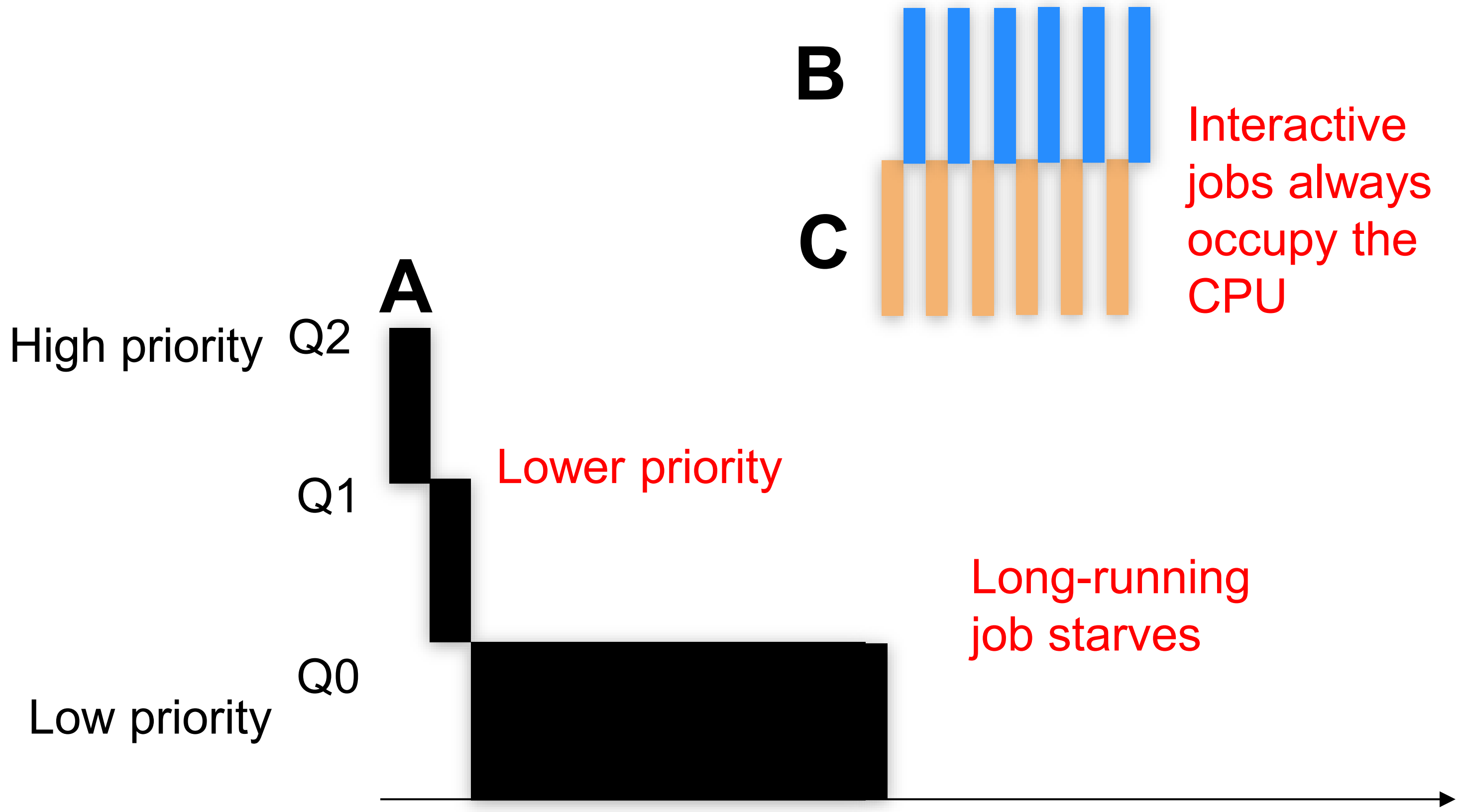
Multi-Level Feedback Queue (MLFQ)

- What are the problems of the current MLFQ?
  - Starvation
  - Processes may “game the scheduler” to get more resource share
  - Changed behavior over time



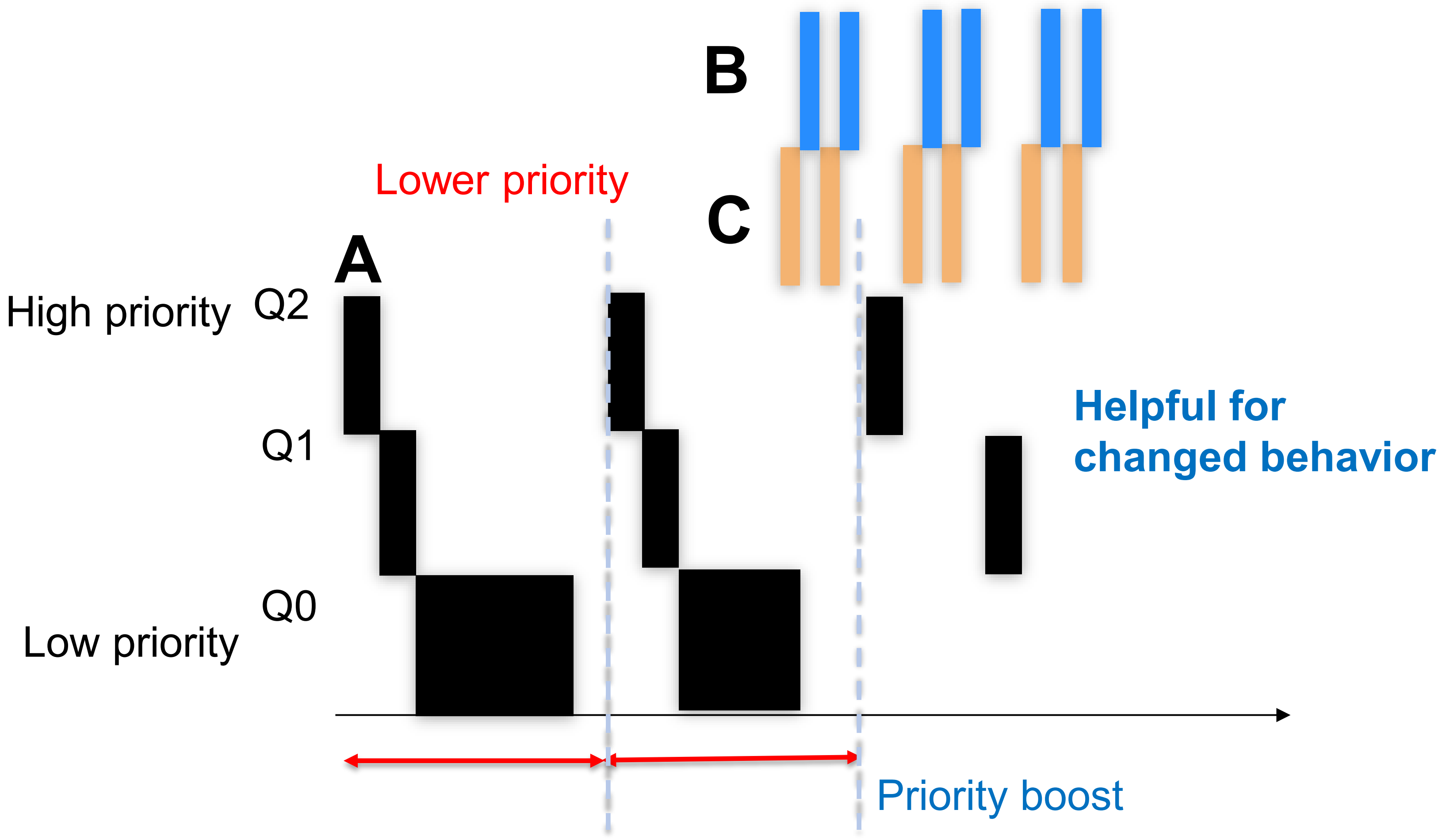
Multi-Level Feedback Queue (MLFQ)

- Starvation



Multi-Level Feedback Queue (MLFQ)

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.



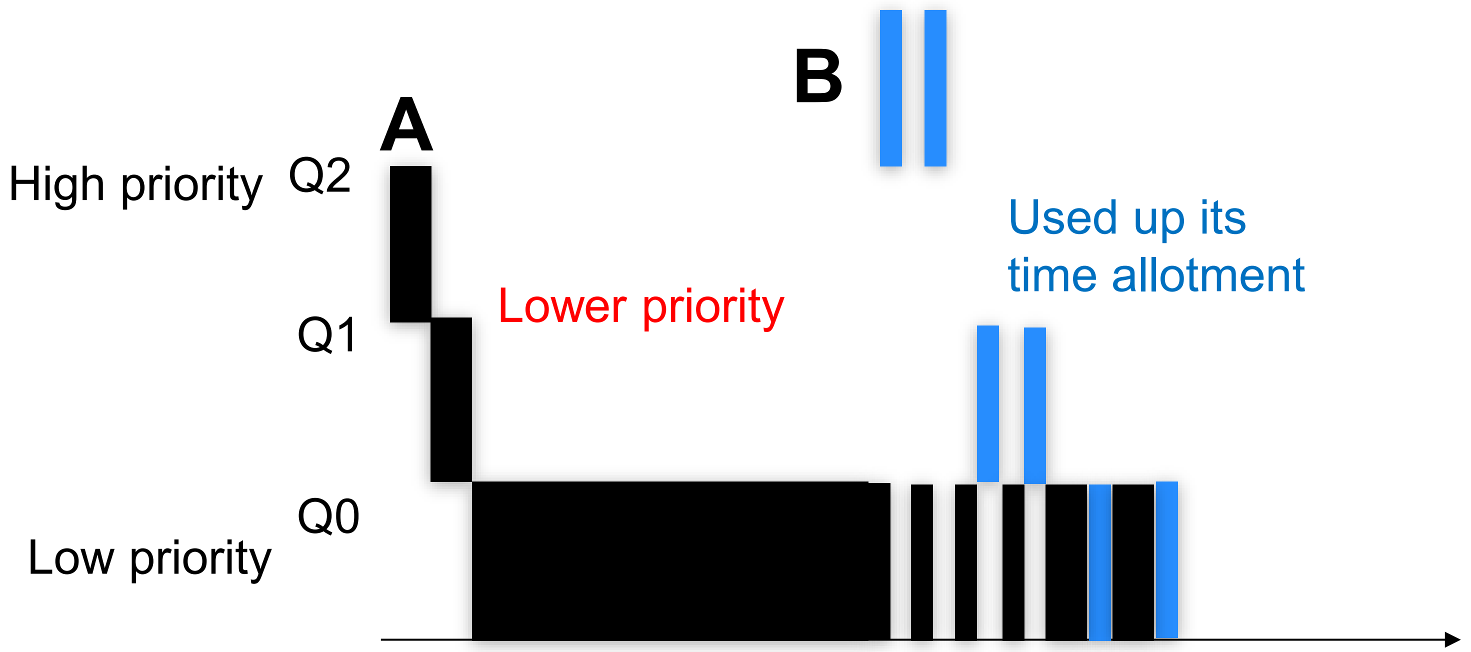
Multi-Level Feedback Queue (MLFQ)

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

↓

- **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).

Multi-Level Feedback Queue (MLFQ)

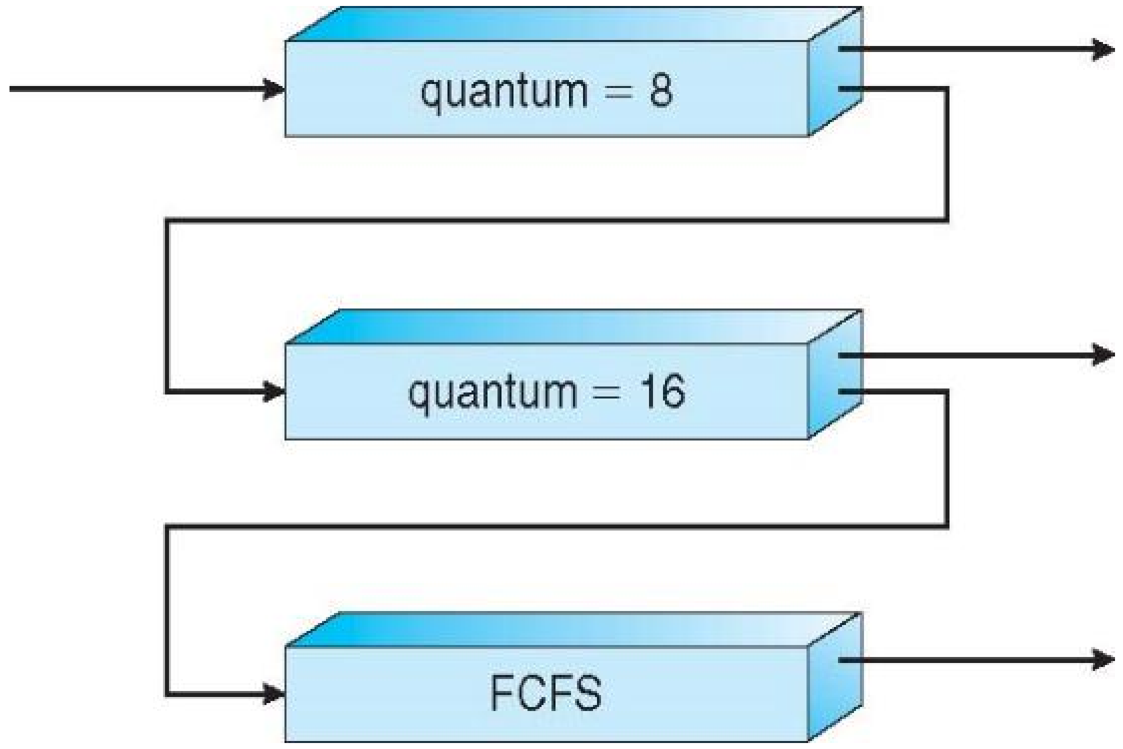


Multi-Level Feedback Queue (MLFQ)

- The refined set of MLFQ rules:
  - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn’t).
  - **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
  - **Rule 3:** When a job enters the system, it is placed at the highest priority.
  - **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 on queue).
  - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
- Benefit of MLFQ
  - It does not require prior knowledge on the CPU usage of a process.

Multi-Level Feedback Queue (MLFQ)

- MLFQ scheduler is defined by the following parameters
  - Number of queues
  - Time quantum of each queue
  - How often should priority be boosted?
  - Scheduling algorithms for each queue
  - etc.
- High priority queue:
  - Interactive processes
  - Response time
- Low priority queue:
  - Batch processes (CPU-intensive)
  - Turnaround time



Proportional Share Scheduling

- **Fair-share** scheduler
  - Guarantee that each job obtain *a certain percentage* of CPU time.
  - Not optimized for turnaround or response time

- **Lottery scheduling**
  - Based on the concept of **tickets**
    - The percentage of tickets denotes the share of a resource for a process

  - There are two processes, A and B.
    - Process A has 75 tickets → receive 75% of the CPU
    - Process B has 25 tickets → receive 25% of the CPU

Proportional Share Scheduling

- A probabilistic way to implement lottery scheduling
  - Time slice (like in RR)
  - Scheduler knows how many tickets exist
  - Scheduler picks a winning tickets from the ticket pool for each time slice

- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

A ≈ 74% **B ≈ 26%**

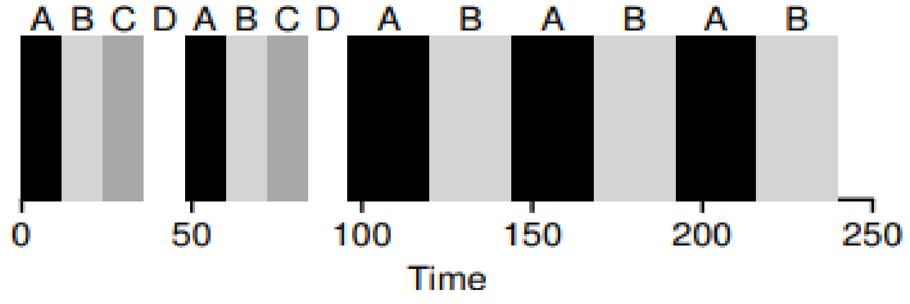
Completely Fair Scheduling (CFS)

- Completely Fair Scheduling (CFS)
  - Choose the process with lowest execution time: ***vruntime.***
  - Run the process for a ***time slice***.
  - The current CPU scheduler in Linux (since 2.6.23, 2007)
  - Non-fixed time slice.
    - CFS assigns process`s time slice a proportion of the processor.
  - **Priority**
    - Enables control over priority by using **nice value**.
  - Efficient data structure.
    - Use **red-black** tree for efficient search, insertion and deletion of a process.

Completely Fair Scheduling (CFS)

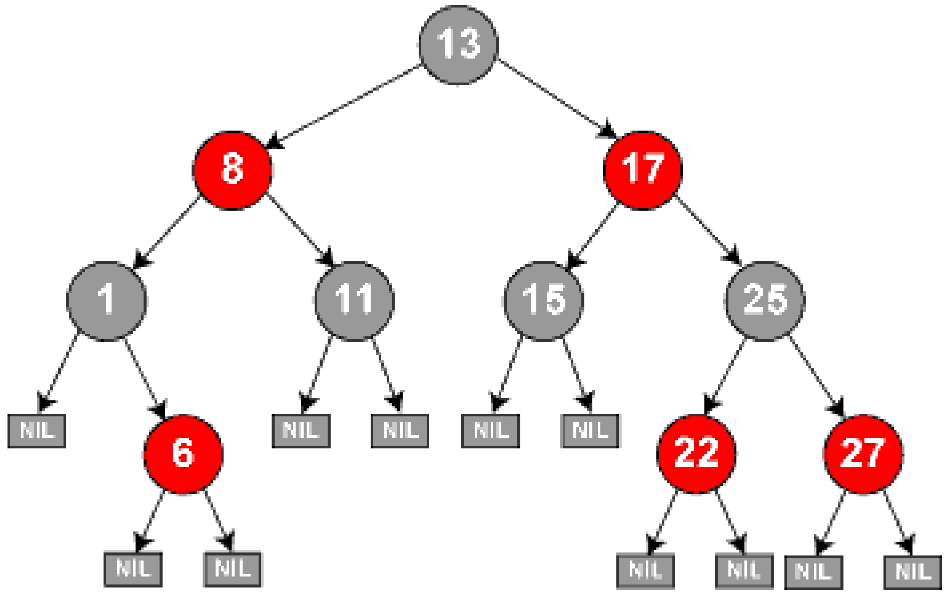
- **Virtual runtime** (vruntime)
  - Denotes how long the process has been executing.
  - Per-process variable
  - Increases in **proportion with physical (real) time** when it runs.
  - CFS will pick the process with the **lowest vruntime** to run next.
- sched_latency
  - Used to determine how long a process should run before considering a switch.
  - A typical value is 48 (milliseconds)
  - process`s time slice = sched_latency / (the number of process)
    - N = 4, time slice = 12msec
    - N = 2, time slice = 24 msec
    - What if N = infinite, set the

    minimum time slice value to 6 ms.



Completely Fair Scheduling (CFS)

- CFS deploys a red-black tree
  - Balanced binary search tree
  - Ordered by **vruntime** as key
  - **Complexity:**
    - insertion, deletion, update -> **O(logN)**
    - find min -> **O(1)**



Completely Fair Scheduling (CFS)

- How does CFS deal with I/O and sleep
  - When a process issues an I/O, it turns into sleep state. Then it will **not** accumulate **vruntime**
  - As soon as the long-sleep process is waked up, it has a small vruntime, i.e., a high priority. As a result, it is likely to monopolize CPU for a long while.
  - **Solution:** Set the vruntime of a wake-up process to the minimum value found in the RB-tree

- How CFS boosts interactivity:
  - I/O-bound (Interactive) processes typically have shorter CPU bursts and thus will have a low vruntime – higher priority