Lecture 07: Classical Scheduling

CS343 – Operating Systems

Branden Ghena – Fall 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Administriva

- PC Lab due Thursday

- Scheduling Lab should be out on Tuesday
  - Put it out early so you could get started before the exam

Midterm Exam 1

- Exam Details
  - In class, Tuesday October 22. Starts at 12:30 sharp. 80-minute exam
  - Covers all lectures through this week Thursday
    - (1. Introduction through 8. Scheduling: Real-Time & Modern)
  - You may bring ONE 8.5”x11” sheet of paper with notes on front and back
    - Handwritten, typeset, whatever you want
  - No calculators or other notes

- Review materials
  - Posted to Canvas homepage: practice problems + prior exams

Today's Goals

- Introduce the concept and challenges of scheduling
- Explore scheduling for batch and interactive systems
- Identify important metrics for measuring scheduler performance
- Examine several scheduling policies that target these metrics

Outline

- **Scheduling Overview**
- Scheduler Metrics
- Batch Systems
  1. First In First Out scheduling
  2. Shortest Job First scheduling
  3. Shortest Remaining Processing Time scheduling
- Interactive Systems
  1. Round Robin scheduling
  2. Multi-Level Feedback Queue scheduling

Lies your operating system always told you

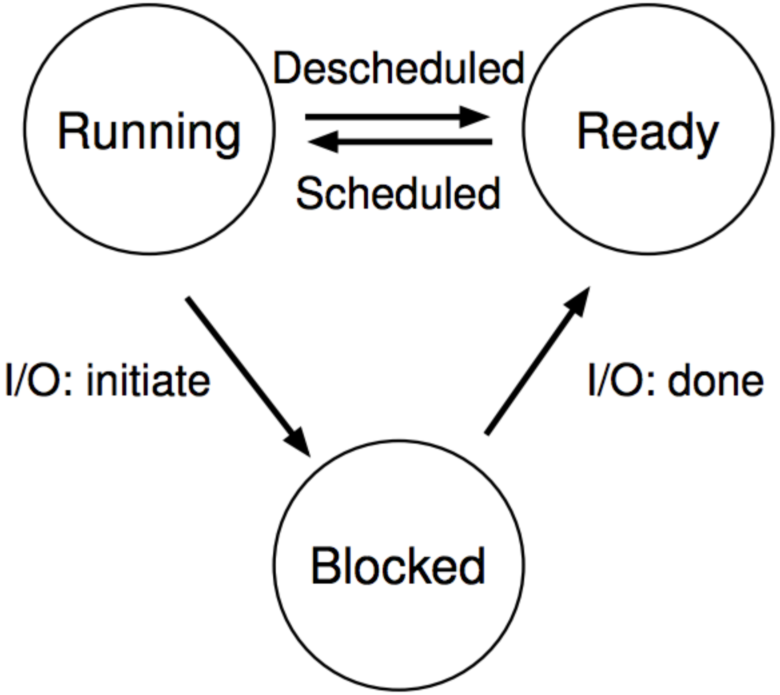
- "Every process on your computer gets to run at the same time!"
    - This is an *illusion*

- My desktop at home (running Windows)
    - Current load: 250 processes with 2987 threads

- So how does the magic work?

Processes don't run all the time

The three basic process states:

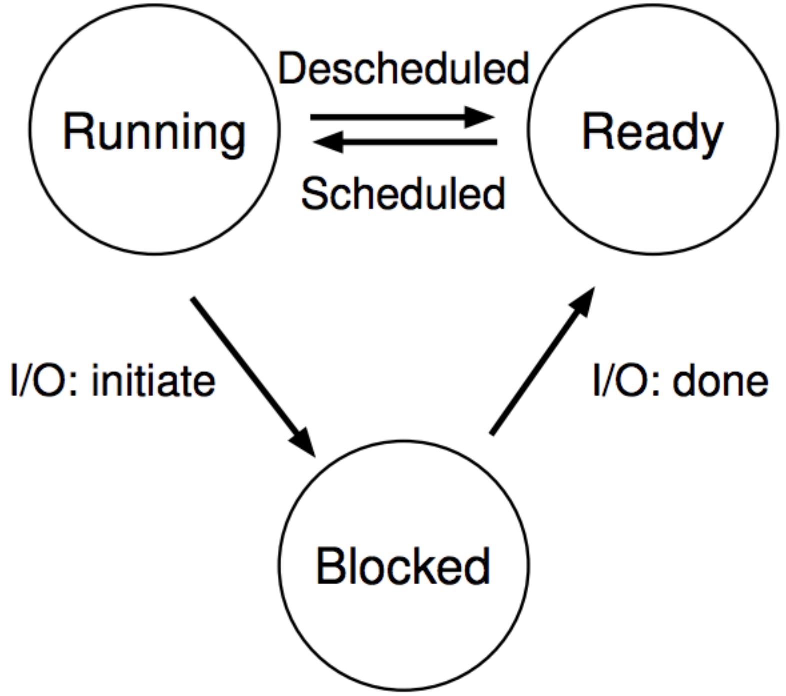


- OS ***schedules*** processes
  - Decides which of many competing processes to run.

- A ***blocked*** process is not ready to run and is waiting on I/O
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke, condvar/semaphore!
  - While waiting for results, the OS **blocks** the process, waiting to do more computation until the result is ready

Multiprogramming processes

The three basic process states:



- Even with a single processor, the OS can provide the illusion of many processes running simultaneously
  - And also use this opportunity to get more useful work done

- When one process is Blocked, OS can schedule a different process that is Ready
- OS can also swap between various Ready processes so they all make progress

Scheduling

- We know that multiple processes will be sharing the CPU
  - Possibly multiple threads in each process
  - Possibly multiple cores in the CPU

- Scheduling is creating a *policy* for sharing the CPU
  - Which process/thread is chosen to run, and when?
  - When (if ever) does the OS change which process is running?

Scheduling terminology

- Job - an execution unit handled by the scheduler (a.k.a. “task”)
  - Thread or process (doesn’t matter in this context)
  - Moves between Ready and Blocked queues

- Workload – set of jobs
  - Arrival time of each job
  - Run time of each job

When can the OS make scheduling decisions?

- Whenever the OS is actually running
  - i.e. after a context switch

- Possible triggers
  - System calls
    - Process/Thread creation/termination
    - I/O requests
    - Synchronization primitives (futex/condvar/semaphore)

  - Hardware events (interrupts)
    - I/O complete
    - Timer triggers

Goal of most schedulers: always have a job running

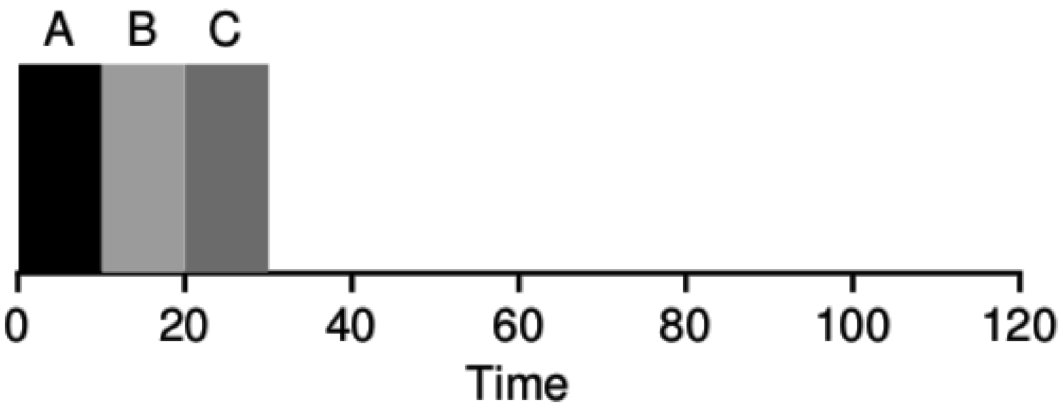
- The schedulers we look at in class are “work-conserving”
  - Always keeps scheduled resource busy if possible
  - When in doubt, make sure *some job* is running on the processor
    - Remember this for the lab and for exams!

- Counter-examples of “non-work-conserving” schedulers
  - Network I/O scheduling may rate-limit to avoid overloading network
  - Energy-limited systems may choose to run nothing to preserve energy

First scheduler: FIFO Scheduling

- First In, First Out (FIFO)
  - also known as First Come First Served (FCFS)

- Policy
  - First job to arrive gets scheduled first
  - Let a job continue until it is complete
  - Then schedule next remaining job with earliest arrival



Outline

- Scheduling Overview
- **Scheduler Metrics**

- Batch Systems
  1. First In First Out scheduling
  2. Shortest Job First scheduling
  3. Shortest Remaining Processing Time scheduling

- Interactive Systems
  1. Round Robin scheduling
  2. Multi-Level Feedback Queue scheduling

Metrics for systems

- Metric – standard for measuring something
  - Mathematical optimization: objective function
  - Economics: utility function

- For different computing scenarios, different metrics will be most important
  - Computing systems have different goals and uses
  - Performance metrics are often in conflict with each other

- Operating Systems are full of *tradeoffs*

A global scheduling metric

- Fairness
  - Each job should get a “fair” share of the processor

- Fair means different things of course
  - Could be “each job gets equal time”
  - Could be “each job starts in order it arrives”
  - Could be “each job is handled based on its priority”

- Scheduler should be fair with regards to the goals of the system it runs on

Other scheduling metrics

- Performance
  - How many jobs does the system complete?
  - How quickly are jobs completed?

- Responsiveness
  - How responsive does the system *feel* to users

- Energy use, types of jobs run, processor cores used, etc.

Different systems have different important metrics

- Example: network server
  - Request for home page
  - Request for contact page

- Example: personal computer
  - Text editor that the user is actively interacting with
  - Compilation running in the background

- Example: autonomous vehicle
  - Image recognition algorithms
  - Radio

Different systems have different important metrics

- Example: network server – **Batch System**
  - Request for home page
  - Request for contact page

- Example: personal computer – **Interactive System**
  - Text editor that the user is actively interacting with
  - Compilation running in the background

- Example: autonomous vehicle – **Real-time System**
  - Image recognition algorithms
  - Radio

Outline

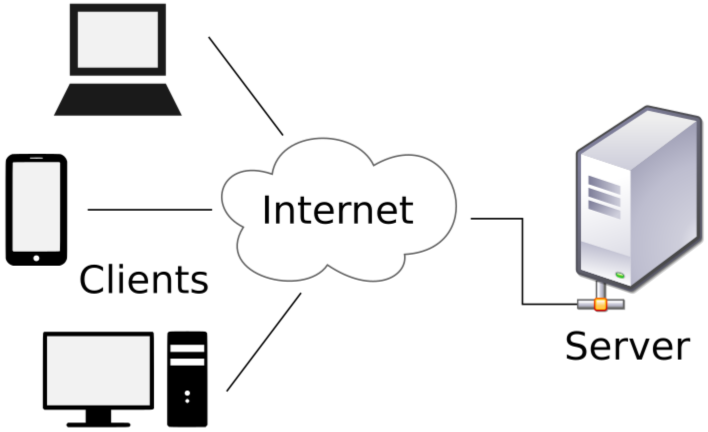
- Scheduling Overview
- Scheduler Metrics
- **Batch Systems**
  1. First In First Out scheduling
  2. Shortest Job First scheduling
  3. Shortest Remaining Processing Time scheduling
- Interactive Systems
  1. Round Robin scheduling
  2. Multi-Level Feedback Queue scheduling

What are batch systems?

- Systems designed to run a set of provided tasks
  - No direct interaction with users
  - Predominantly run-to-completion jobs

- Example: banking systems or payroll management

- Modern example: network servers
  - Tasks are serving requests
  - Multiple types of requests, each with known runtimes

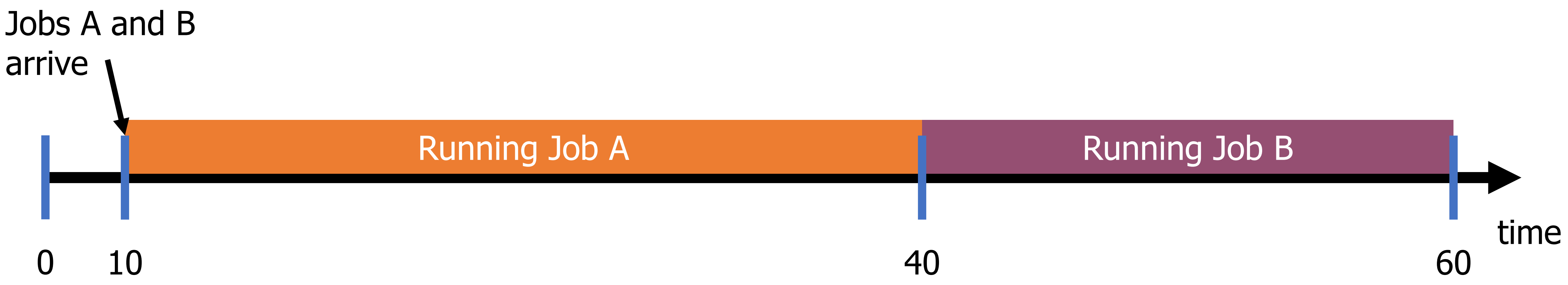


Metrics for batch systems

- Throughput
    - Jobs completed per unit time
    - Throughput = jobs_completed / total_duration
    - Higher is better

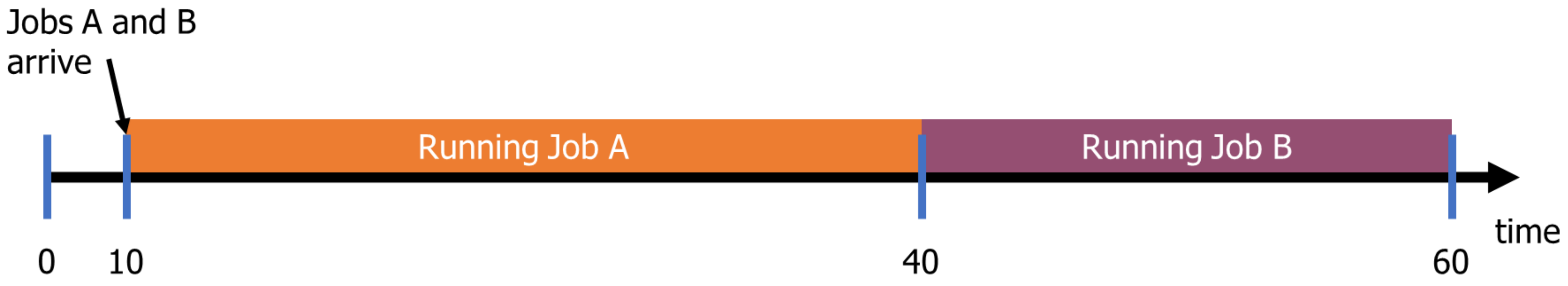
- Turnaround time
    - Duration from job arrival until job completion
    - $T_{turnaround} = T_{completion} - T_{arrival}$
    - Lower is better
    - Average turnaround time is computed across all jobs

Example: throughput and turnaround



- Job A
  - Arrival: 10
  - Completion: 40
  - Duration: 30
- Job B
  - Arrival: 10
  - Completion: 60
  - Duration: 20

Example: throughput and turnaround



Throughput = jobs_completed / total_duration

$T_{turnaround} = T_{completion} - T_{arrival}$

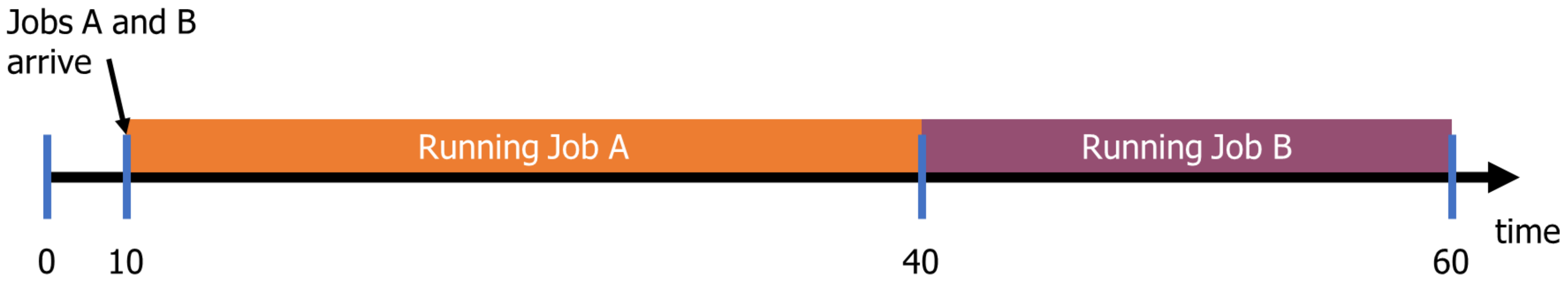
Throughput

Turnaround for A

Turnaround for B

Average Turnaround

Example: throughput and turnaround



Throughput = jobs_completed / total_duration

$T_{turnaround} = T_{completion} - T_{arrival}$

Throughput

2 jobs / 50 time = 0.04

Turnaround for A

40-10 = 30

Turnaround for B

60-10 = 50

Average Turnaround

(30+50)/2 = 40

Batch scheduler metric

- Which metric is most relevant to a batch system scheduler with a finite list of processes?
  - Throughput or Turnaround

- Throughput only cares about sum of durations of jobs
  - Throughput is the same no matter whether A or B goes first

- Turnaround accounts for delays in scheduling a job
  - Swapping A and B would result in better average turnaround

Turnaround for A	**Turnaround for B**	**Average Turnaround**
60-10 = 50	30-10 = 20	(50+20)/2 = 35

Schedulers for batch systems

1. First In First Out

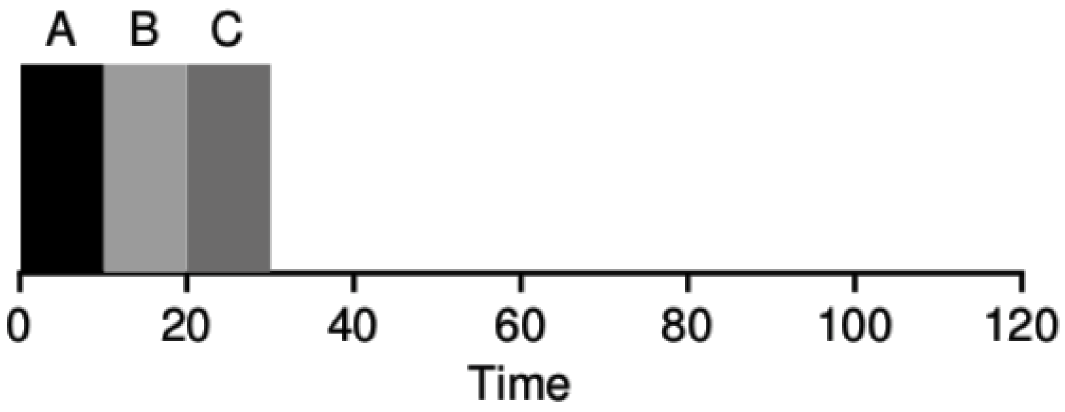
2. Shortest Job First

3. Preemptive Shortest Remaining Processing Time

1. FIFO Scheduling

- First In, First Out (FIFO)
  - assumption for now: all jobs arrive at time zero

- What is the average turnaround for this workload?
  - (10 + 20 + 30)/3 = 20

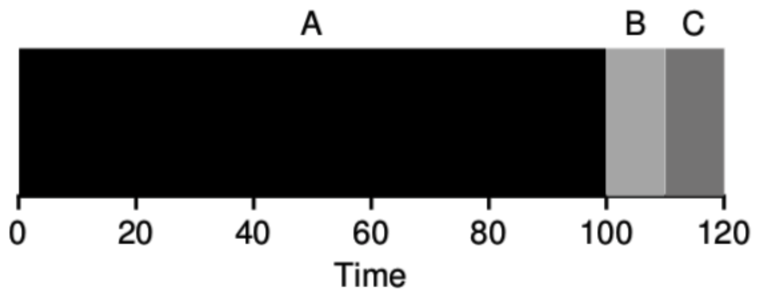


Check your understanding – FIFOs with different durations

- What is a problematic scenario for FIFO scheduling?
  - (consider job durations)

Check your understanding – FIFOs with different durations

- What is a problematic scenario for FIFO scheduling?
- One big job can cause lots of jobs behind it to wait
    - Convoy effect – lots of small jobs stuck behind one big job

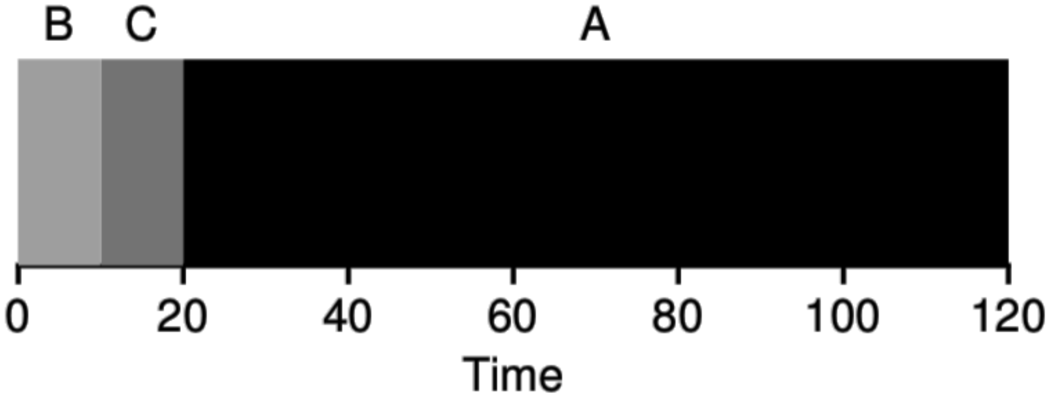


- Average turnaround time = (100+110+120)/3 = 110
    - Minimum average turnaround time = (10+20+120)/3 = 50

2. Shortest Job First

- Policy
  - Schedule the job with the smallest duration first
  - Let a job continue until it is complete
  - Then schedule next remaining job with smallest duration

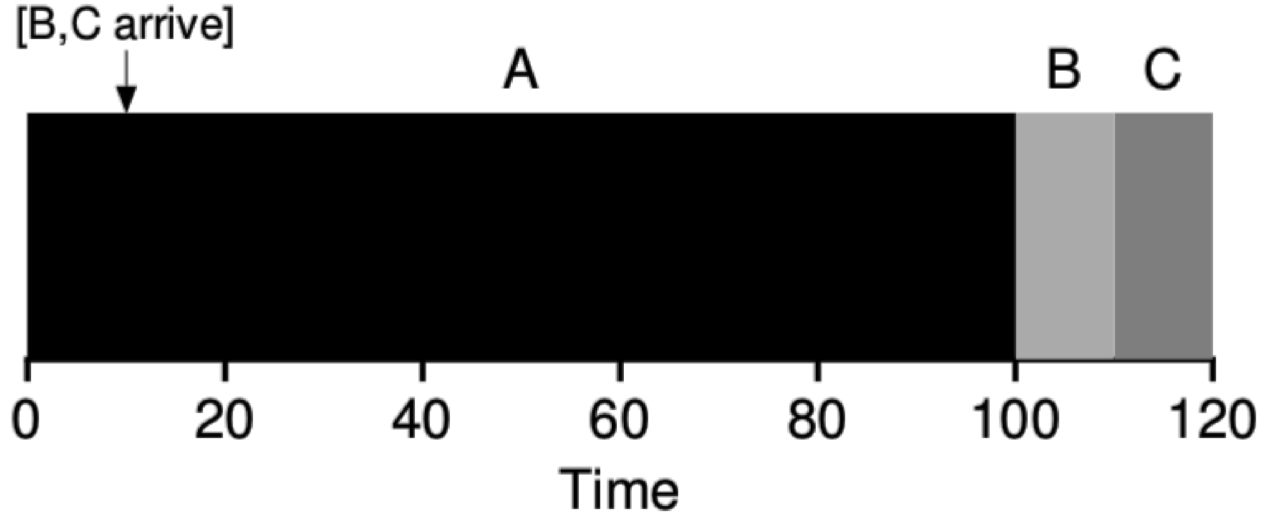
- Essentially: complete a job as soon as possible
  - Minimizes the number of waiting jobs, minimizing average turnaround



Average Turnaround
(10+20+120)/3 = 50

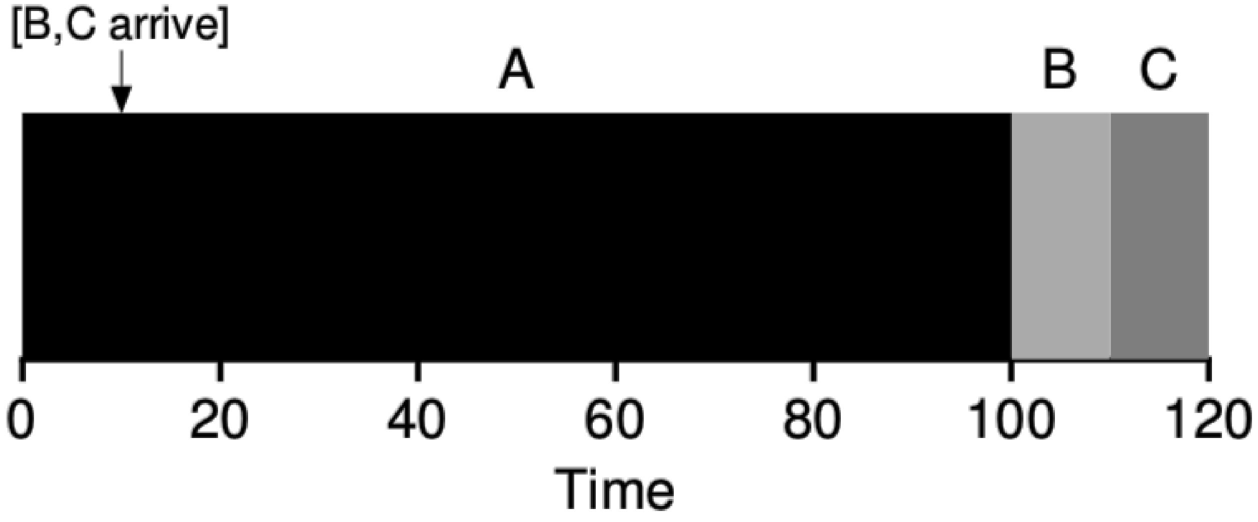
Shortest Job First can fail with late arrivals

- Scheduler's previously optimal decision could be invalidated by new job arrivals
  - If B and C arrive late, they will have to wait because A is already running



Check your understanding

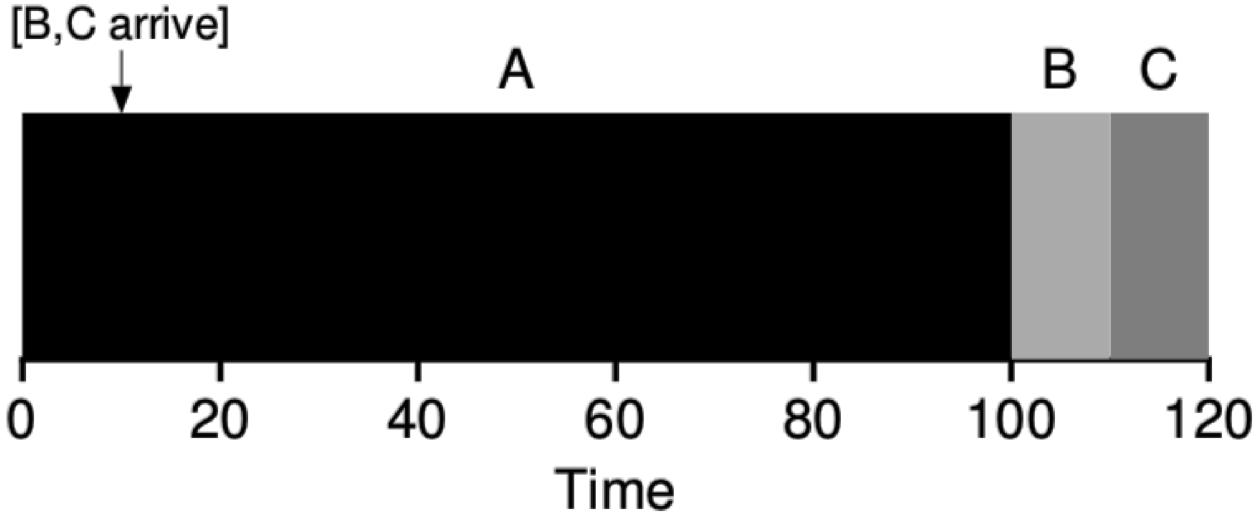
- What is the average turnaround time for this example?
  - B and C arrive at time 10



Check your understanding

- What is the average turnaround time for this example?
  - B and C arrive at time 10

- Average turnaround = ((100-0) + (110-10) + (120-10))/3 = 103.33333

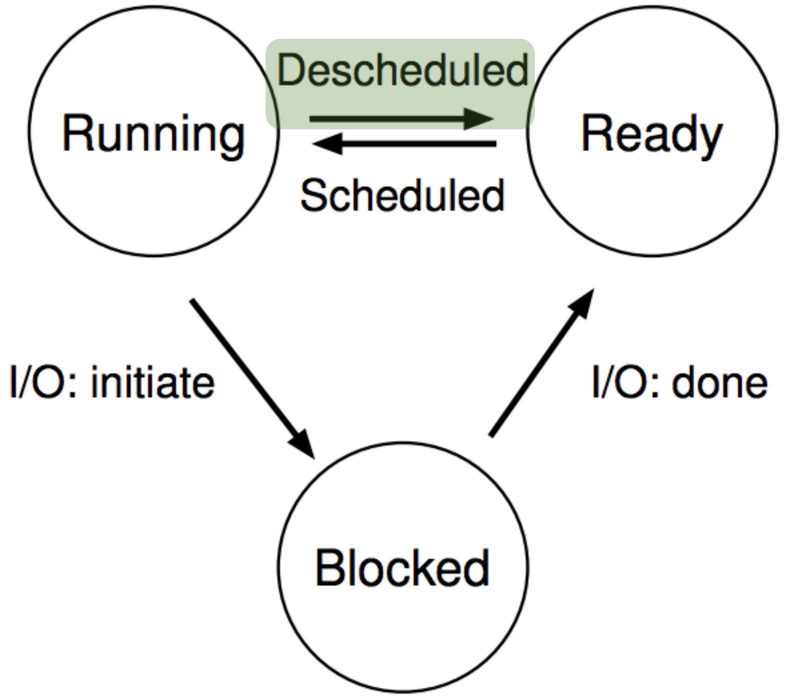


Preemption

- Let's add a new scheduler capability: preemption

- OS can "deschedule" jobs that are running

- This means it can make scheduling decisions more frequently
  - System calls
  - Interrupts
  - Timers



Context switching overhead

- Switching processes is expensive
  - Context switch to OS is on the order of 1 μs (1 millionth of a second)
  - Switching registers and CPU mode
- Memory is often the larger expense though
  - New process has different physical memory pages
  - Which means that caches have to be cleared
  - Caches will "warm up" as the process runs
  - Less of a penalty to threads (only stack changes)
- Alternative option: cooperative scheduling through yield()

3. Preemptive Shortest Remaining Processing Time

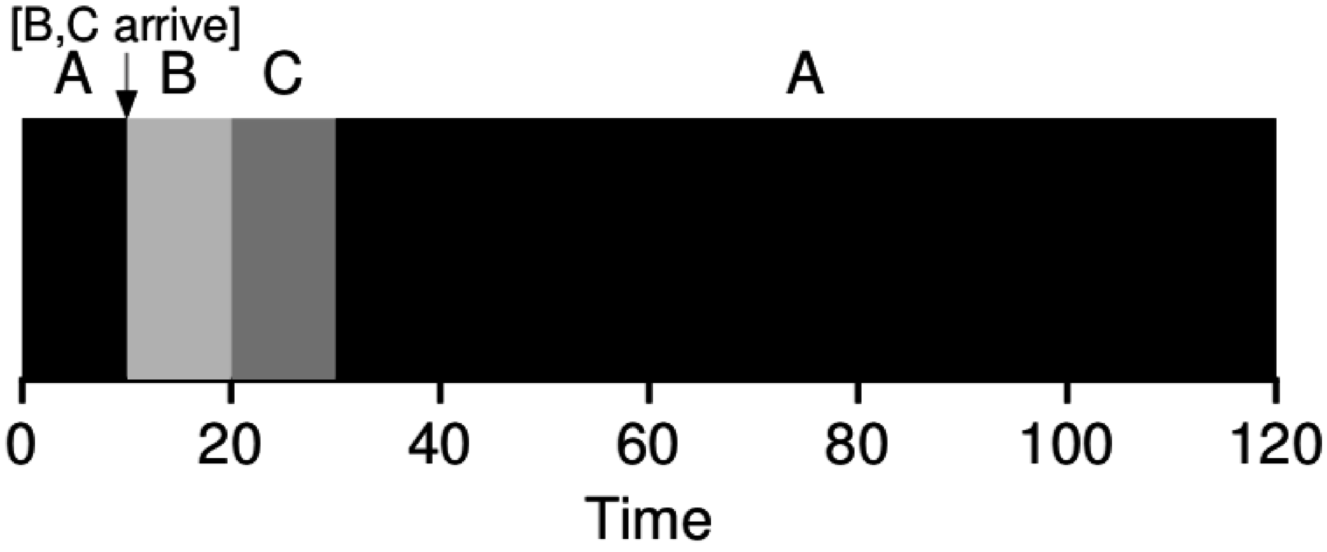
- Also known as Shortest Time-to-Completion First

- Policy
  - Schedule job with smallest duration first
  - Preempt a running job when new jobs arrive
  - Then schedule job with smallest remaining duration

- Essentially, reevaluate schedule when new information is gained

Shortest Remaining Processing Time example

- A is preempted when B and C arrive at time 10
- Scheduler chooses B as new shortest remaining time
  - B=10, C=10, A=100



Average Turnaround
(120+10+20)/3 = 50

Break + Starvation and scheduling

- Starvation can occur in schedulers
  - When one job will never actually get a chance to run

- We've discussed:
  - FIFO, Shortest Job First, and Shortest Remaining Processing Time
  - Which of these can exhibit starvation?

Break + Starvation and scheduling

- Starvation can occur in schedulers
  - When one job will never actually get a chance to run

- We've discussed:
  - FIFO, Shortest Job First, and Shortest Remaining Processing Time
  - Which of these can exhibit starvation?
    - Shortest Remaining Processing Time
    - Shortest Job First too if we allow new job arrivals (without preemption)

  - Arriving short tasks could lead a long task to never be scheduled

Outline

- Scheduling Overview
- Scheduler Metrics
- Batch Systems
    1. First In First Out scheduling
    2. Shortest Job First scheduling
    3. Shortest Remaining Processing Time scheduling
- **Interactive Systems**
    1. Round Robin scheduling
    2. Multi-Level Feedback Queue scheduling

What are interactive systems?

- Every computer you directly interact with
  - Desktops, laptops, smartphones
- Differences from batch systems
  - Humans are "in-the-loop"
    - Computer needs to feel responsive for programs they are using
  - **Many jobs have no predefined duration**
    - How long does Chrome run for?
- Still have some batch jobs though (background services)

Metric for interactive systems

- Response time
  - Time from arrival until the job **begins** execution
  - Doesn’t matter how long the job takes to run since it runs indefinitely
  - $T_{response} = T_{start} – T_{arrival}$

- Particularly useful for interactive processes
  - Need to quickly show that they are reacting to user inputs
  - Exact total run duration isn’t so important though

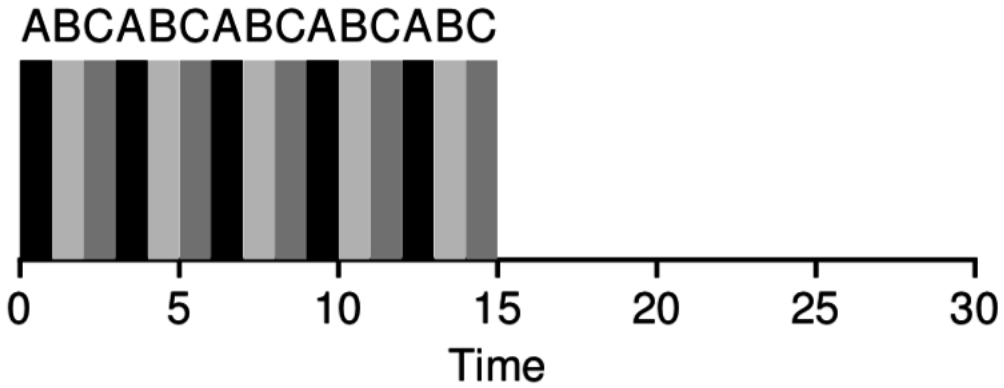
Schedulers for interactive systems

1. Round Robin

2. Multi-Level Feedback Queue

1. Round Robin

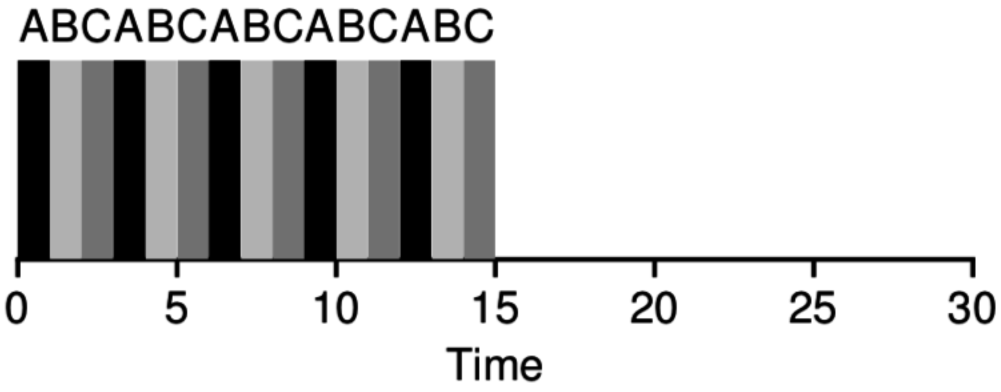
- Round Robin scheduling runs a job for a small *timeslice* (quanta), then schedules the next job

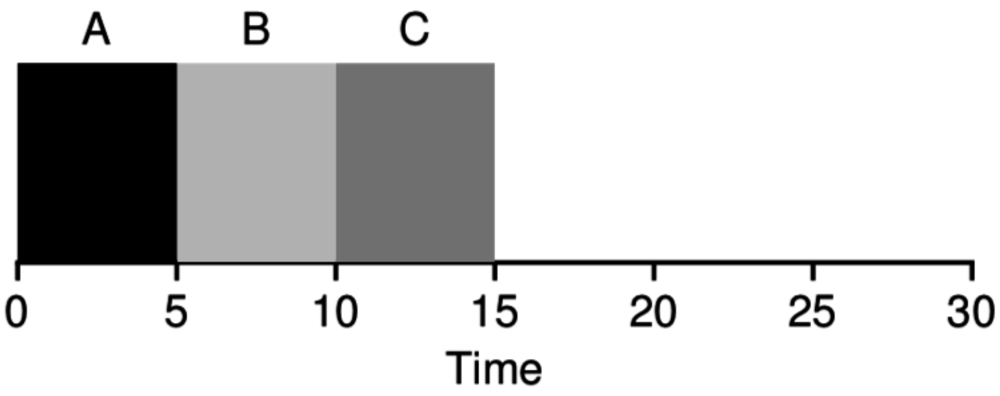


- If all jobs arrive at time 0
  - Average response time = (0 + 1 + 2)/3 = 1

- Smaller timeslice means smaller response time

Different policies favor different metrics





Round Robin scheduling:

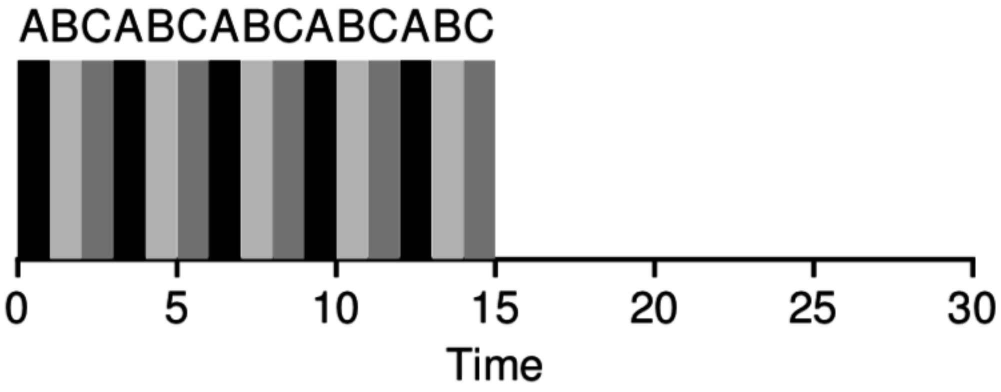
- Avg turnaround time = **14**
- Avg response time = **1**

Shortest Job first or ***SRPT***:

- Avg turnaround time = **10**
- Avg response time = **5**

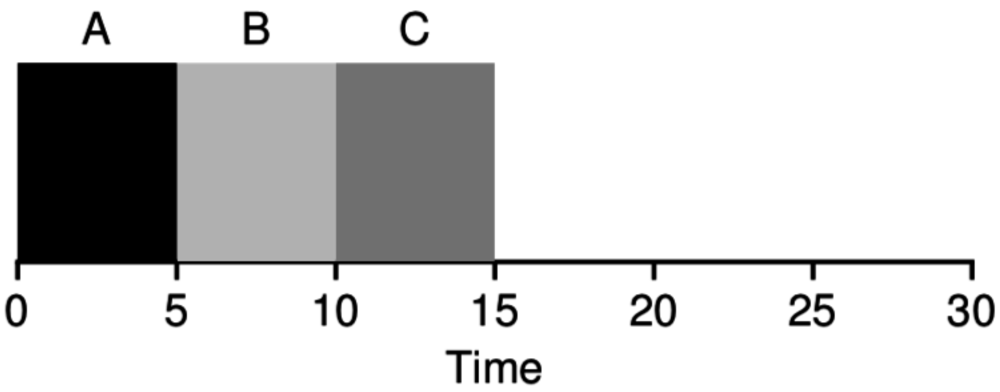
Better response time versus Better turnaround time

Remember, context switches are not free



Round Robin scheduling:

- Context switches = **14**



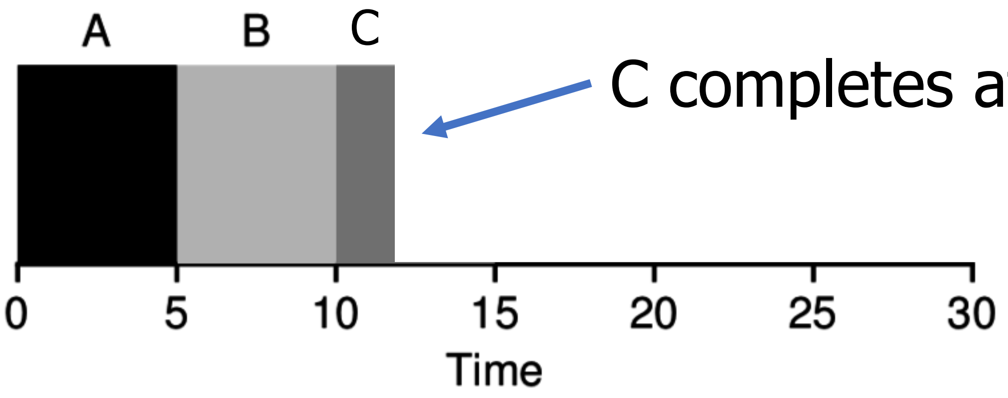
Shortest Job first or ***SRPT***:

- Context switches = **2**

- In a real OS, Round Robin would take an extra ~12 µs
  - Plus more time lost with cold caches…
- Timeslice must be **much** greater than context switch time
  - Usually timeslice is ~1 ms and context switch is ~1 µs

Handling a round-robin edge case

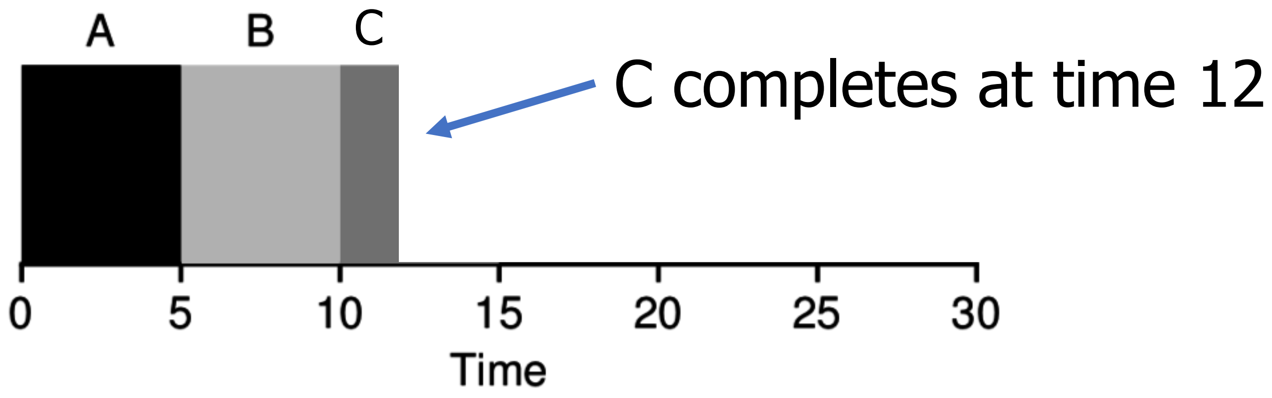
Assume quantum (timeslice duration) is 5



C completes at time 12

- What should the scheduler do?
  1. Schedule nothing for the rest of the timeslice
  2. Schedule a new job for the rest of the timeslice
  3. Schedule a new job with a new, full timeslice

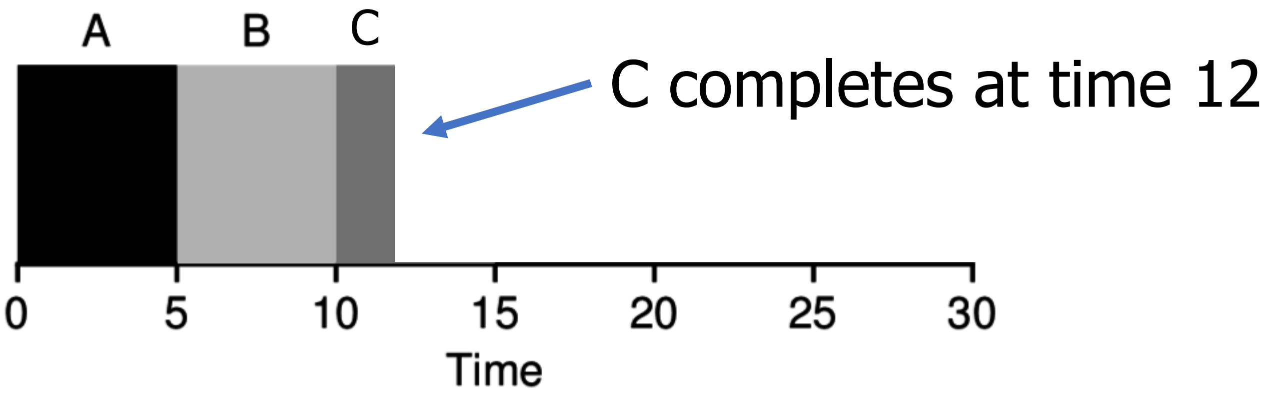
Handling a round-robin edge case



Assume quantum (timeslice duration) is 5

- What should the scheduler do?
  1. ~~Schedule nothing for the rest of the timeslice~~ **Not work-conserving**
  2. Schedule a new job for the rest of the timeslice
  3. Schedule a new job with a new, full timeslice

Handling a round-robin edge case



Assume quantum (timeslice duration) is 5

- What should the scheduler do?
  1. ~~Schedule nothing for the rest of the timeslice~~ Not work-conserving
  2. ~~Schedule a new job for the rest of the timeslice~~ **Not fair**
  3. Schedule a new job with a new, full timeslice **Correct!**

Timeslices are attached to jobs

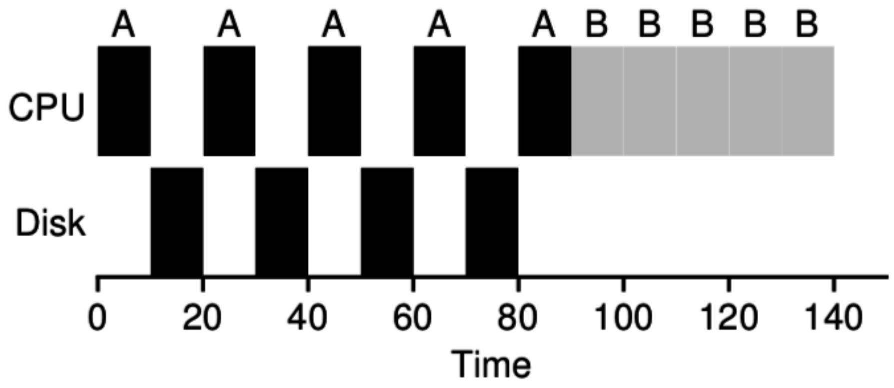
- Each *job* gets its own timeslice duration

- Jobs may use less than their entire timeslice voluntarily
  - They could complete
  - They could become blocked
  - They could decide to yield

- The scheduler, however, should always provide a full timeslice
  - In previous example: runtime of one job shouldn't affect another job

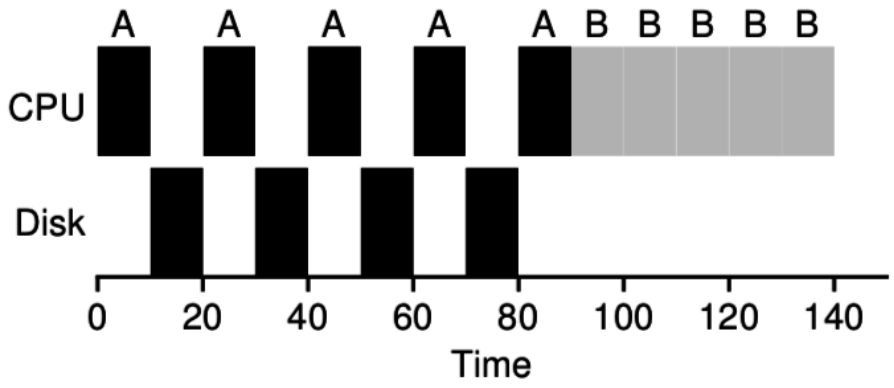
I/O creates scheduling ***overlap*** opportunities

- Job A does I/O every ten milliseconds and each I/O takes 10 ms:
- A is **blocked** during its I/O.
  - It's just waiting for data from the disk
  - But it does not need the CPU



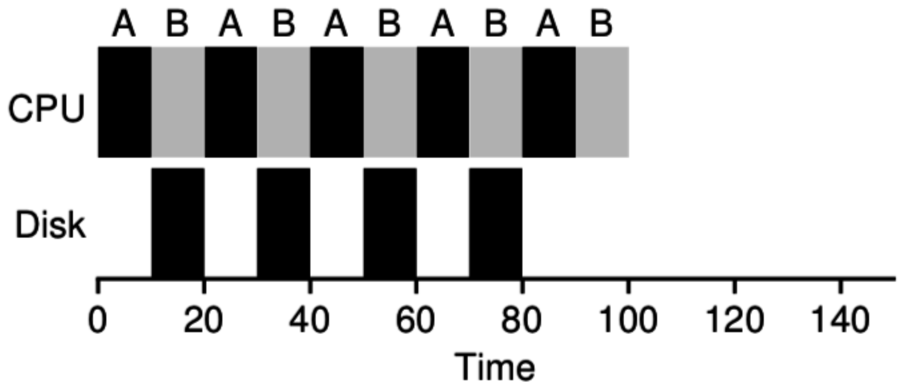
I/O creates scheduling ***overlap*** opportunities

- Job A does I/O every ten milliseconds and each I/O takes 10 ms:



- A is **blocked** during its I/O.
  - It's just waiting for data from the disk
  - But it does not need the CPU

- We can schedule another job during process A's I/O
  - Once a job is blocked, the scheduler can immediately move to the next job!



Jobs can be I/O-bound or CPU-bound

- CPU-bound process
  - Lots of computation between each I/O request
  - Actually needs to do computation on a processor
  - Example: doing matrix math

- I/O-bound process
  - Very little computation between each I/O request
  - Just needs a processor to figure out its next I/O request
  - Example: searching a file system for a file name

Scheduling goal: I/O-bound before CPU-bound

- First maximize I/O
  - Run the I/O-bound jobs as quickly as possible,
  - So they can send next I/O request,
  - And our disks, network cards, etc. are maximally used

- Then fill up the processor(s)
  - Lots of room for multiprogramming between the I/O requests
  - Blocked jobs are still “progressing” as their I/O is fetched

Scheduling goal: I/O-bound before CPU-bound

- First maximize I/O
  - Run the I/O-bound jobs as quickly as possible,
  - So they can send next I/O request,
  - And our disks, network cards, etc. are maximally used

- Then fill up the processor(s)
  - Lots of room for multiprogramming between the I/O requests
  - Blocked jobs are still “progressing” as their I/O is fetched

- But how do you know when a job is going to use I/O?
  - Can’t know the future
  - Can track past behavior of the job

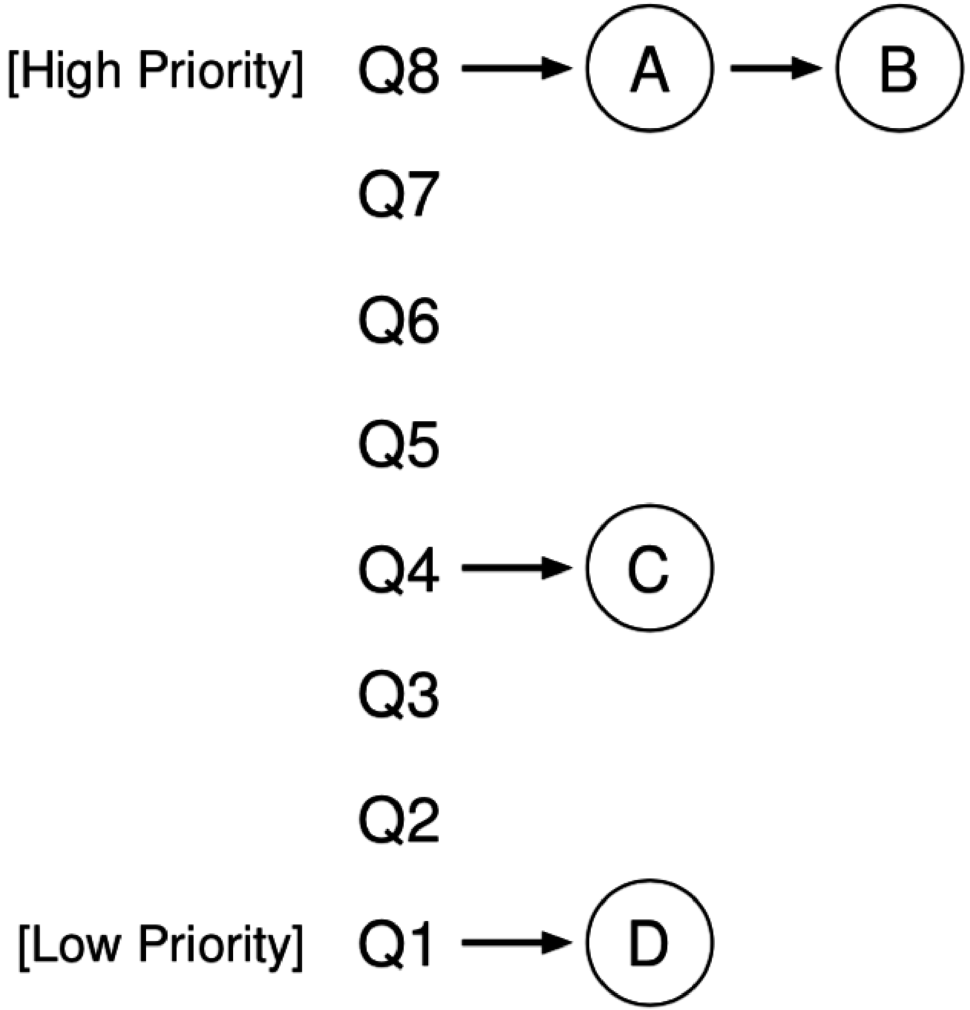
2. Multi-Level Feedback Queue (MLFQ)

- General purpose scheduler to support multiple goals
  - Good response time for interactive jobs
  - Good turnaround time for batch jobs
  - Achieves this by prioritizing I/O bound jobs over CPU bound jobs

- Policy
  - Automatically attach priority to jobs:
    - Interactive, I/O bound jobs should be highest priority
    - CPU bound, batch jobs should be lowest priority
    - Apply different round robin timeslices to each priority level

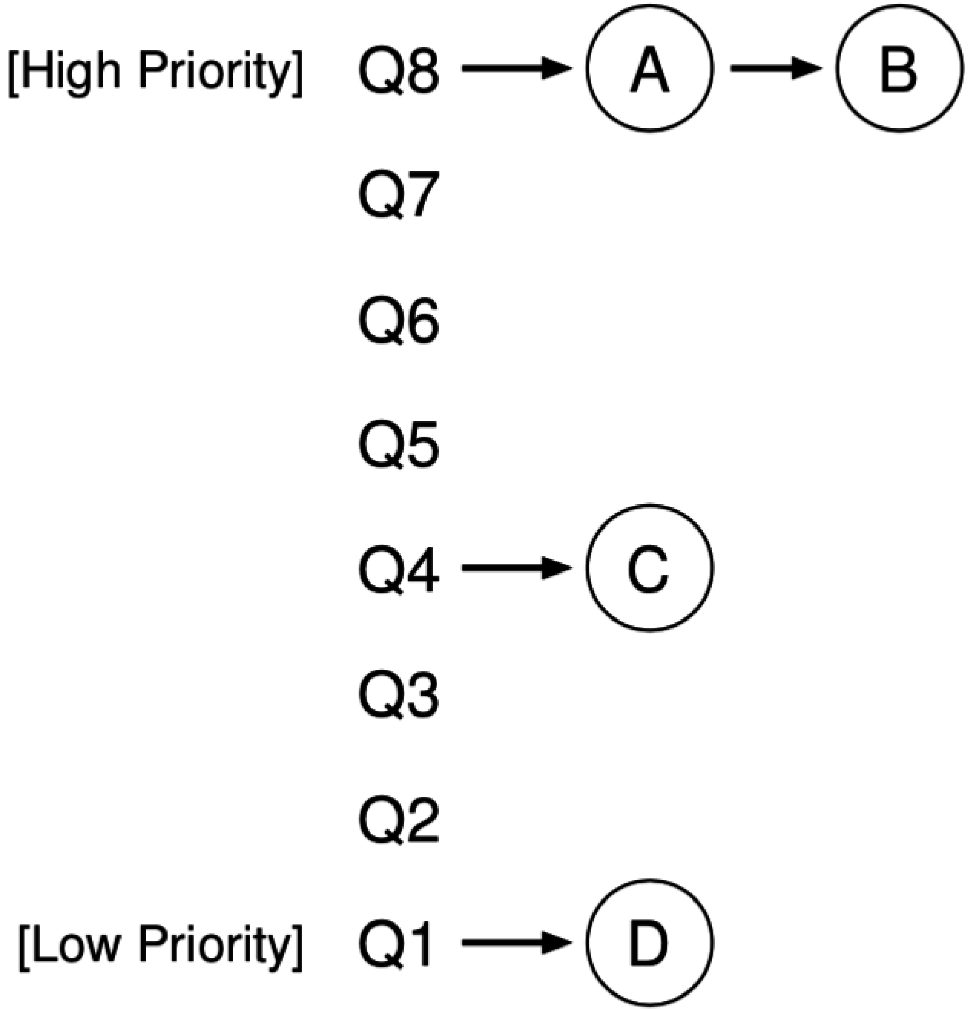
Multi-Level Feedback Queue Details

- Run highest priority level available
  - Round robin among jobs there
- When all jobs at a level are blocked on I/O
  - Move down to next lower level
- Long running jobs lose priority
  - Set a processor usage limit at a given level
  - When used up, demote job one level

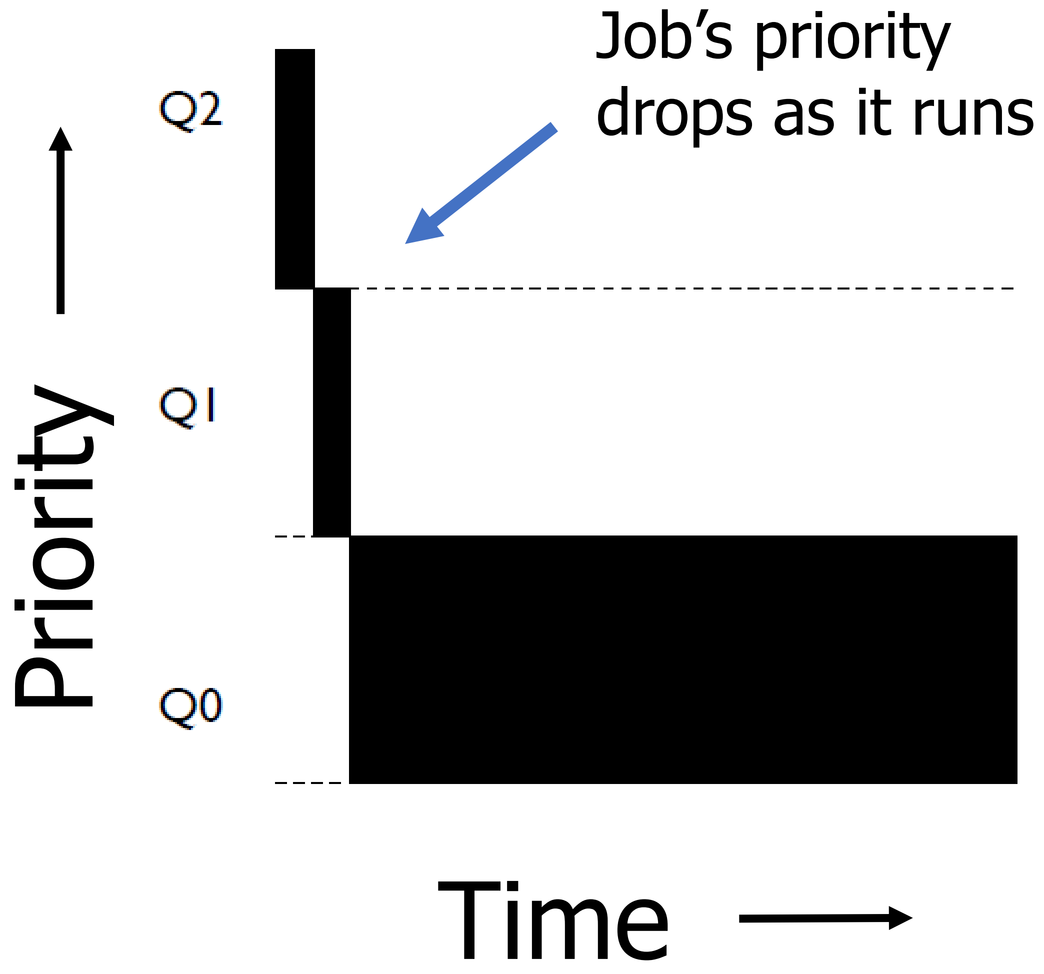


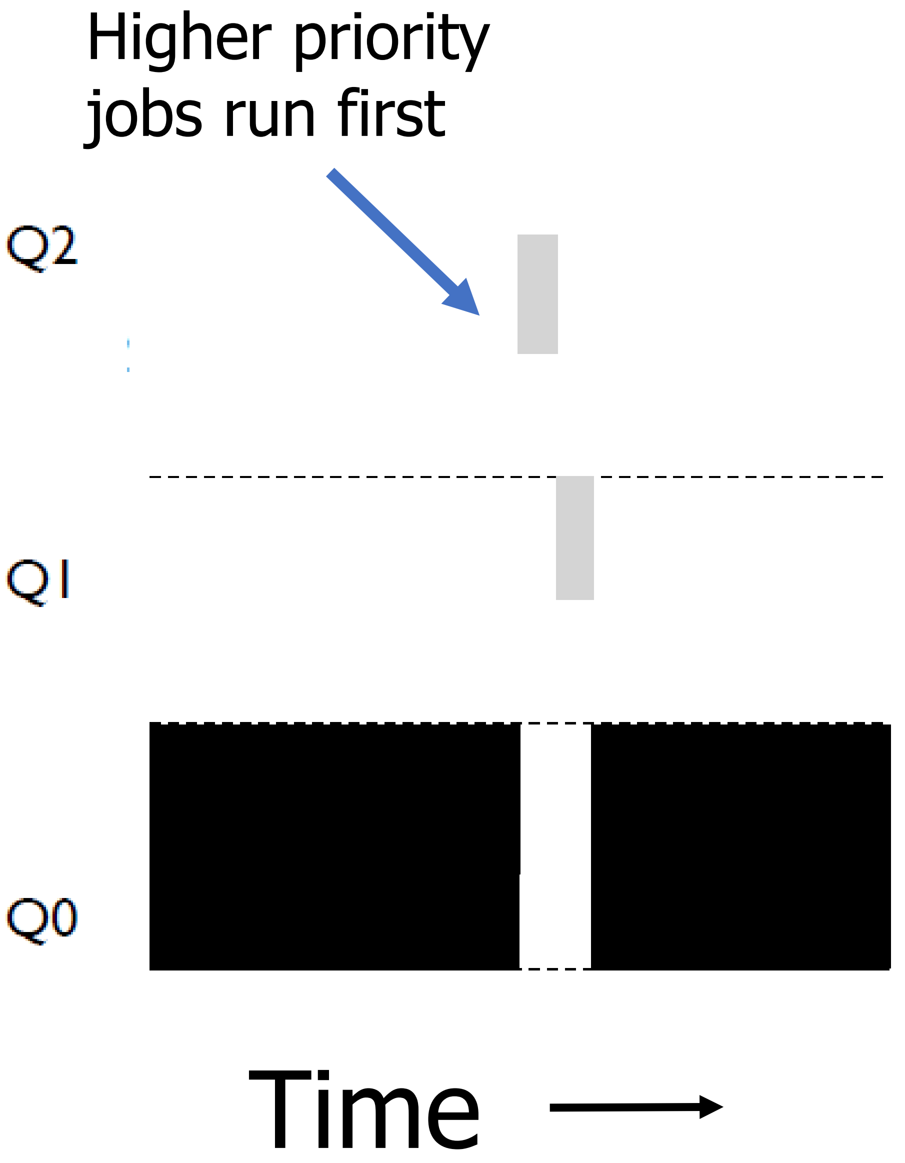
MLFQ Rules

1. If Priority($\mathbf{J_1}$) > Priority($\mathbf{J_2}$), $\mathbf{J_1}$ runs
2. If Priority($\mathbf{J_1}$) = Priority($\mathbf{J_2}$), $\mathbf{J_1}$ and $\mathbf{J_2}$ run in Round Robin
3. Jobs start at top priority
4. When a job uses its time quota for a level, demote it one level
5. Every **S** seconds, reset priority of all jobs to top



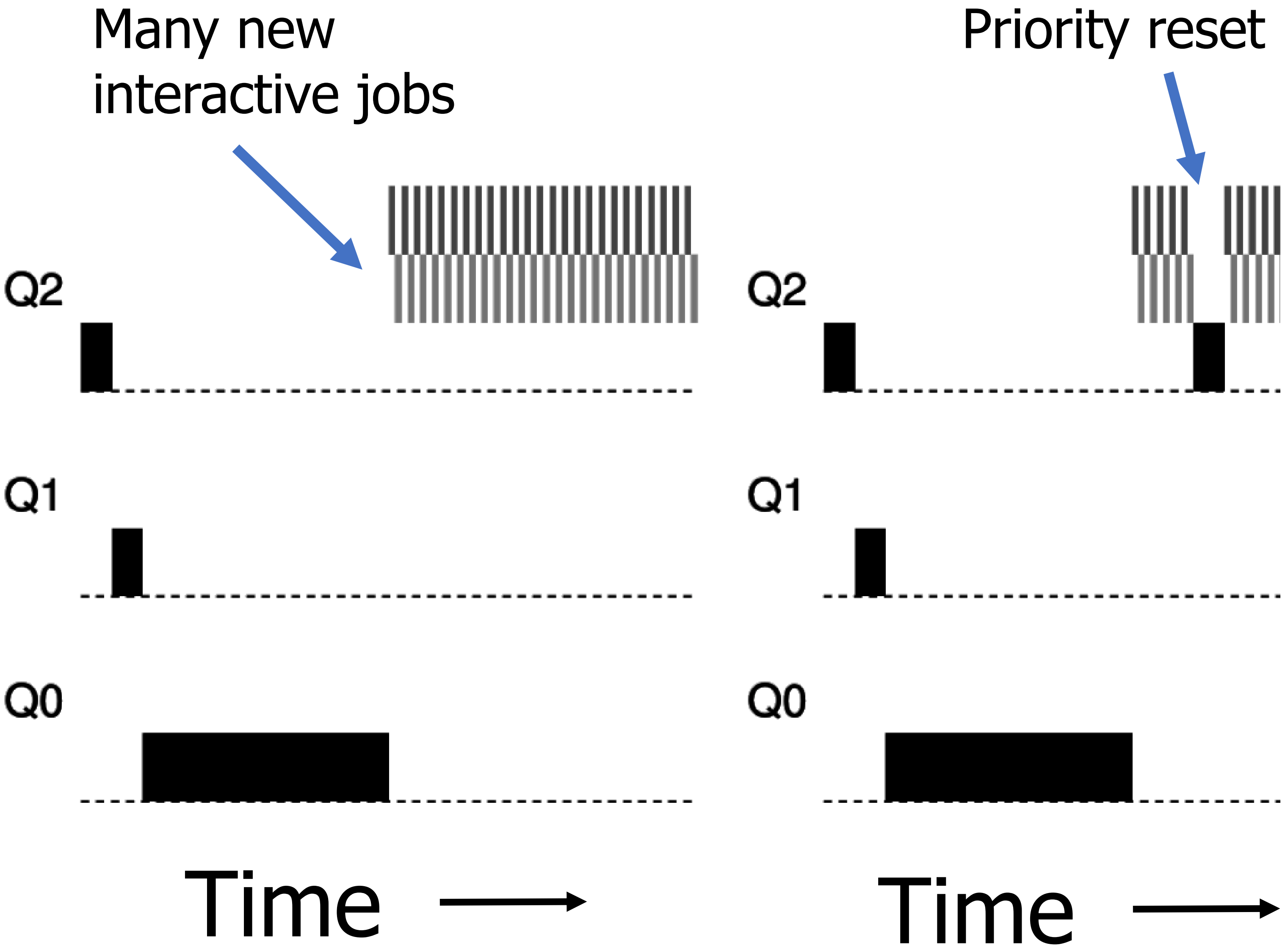
MLFQ Example





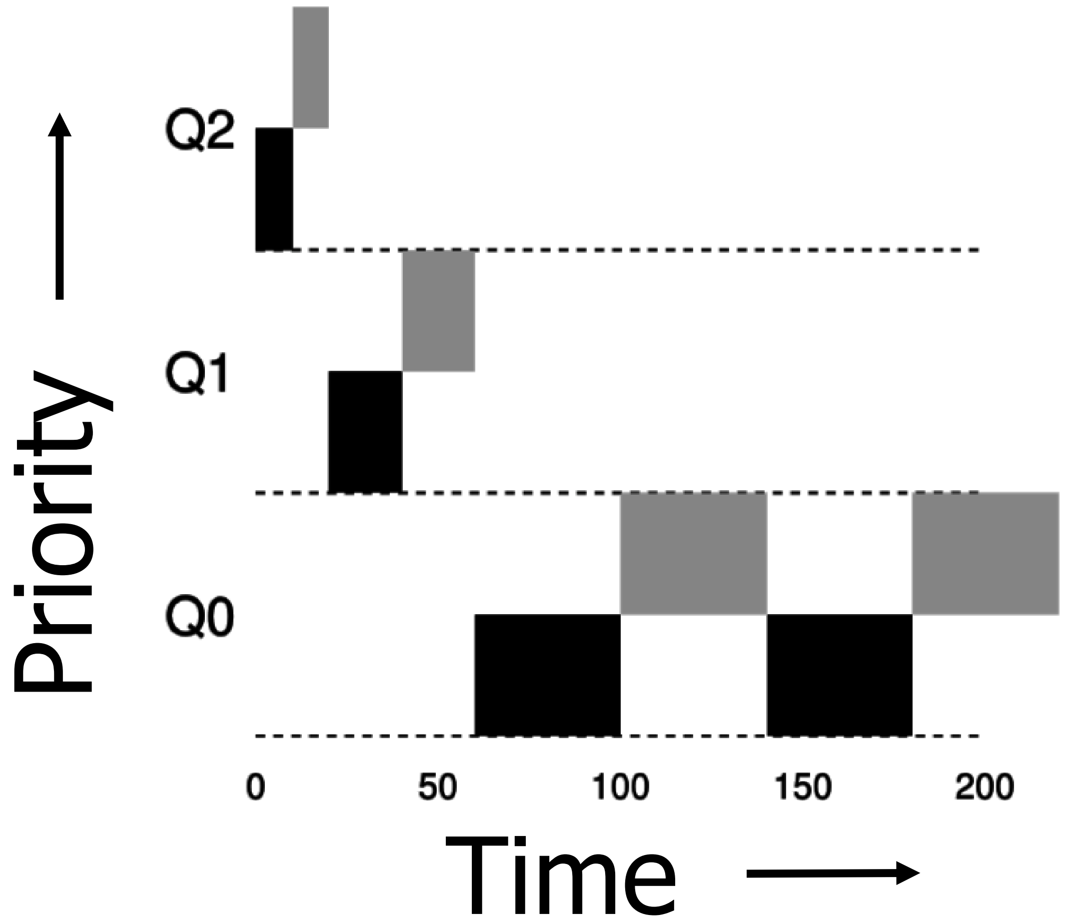
MLFQ avoids starvation with periodic priority reset

- Low priority jobs could starve if there are enough interactive jobs
- MLFQ avoids starvation by periodically resetting priorities



Change timeslices to optimize response and turnaround

- Lower priority jobs are CPU bound, not interactive
  - So we can use longer timeslices to minimize context switches



MLFQ parameters

- Every MLFQ implementation needs to choose a bunch of parameters
  - How many queues/priority levels?
  - When does a job get demoted in priority?
  - How often to reset priority for everything?
  - How large is the timeslice at each priority level?

MLFQ in the wild

- The embedded OS I work on has an MLFQ scheduler!
  - https://github.com/tock/tock/blob/master/kernel/src/scheduler/mlfq.rs
- How many queues/priority levels?
  - Three
- When does a job get demoted in priority?
  - If it ever uses its whole timeslice without blocking
- How often to reset priority for everything?
  - Every five seconds
- How large is the timeslice at each priority level?
  - 10 ms, 20 ms, 50 ms

Outline

- Scheduling Overview
- Scheduler Metrics
- Batch Systems
  1. First In First Out scheduling
  2. Shortest Job First scheduling
  3. Shortest Remaining Processing Time scheduling
- Interactive Systems
  1. Round Robin scheduling
  2. Multi-Level Feedback Queue scheduling