CSIS 429 Operating Systems

Lecture 3: CPU Policy: Scheduling

September 14th 2020

Textbook chapters

- Read “Direct Execution” and “CPU Scheduling”

Intro	**Virtualization**		**Concurrency**	**Persistence**	**Appendices**
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Review: Understanding Operating Systems

- Operating system
  - Software that helps other programs control computer hardware and interact with users
- Application
  - Software program that provides service for computer user
  - Cannot act without “permission” from operating system

Review: CPU Virtualization

The goal of CPU virtualization is to run N processes on M CPUs. N >> M

Abstraction used: “process”

Each process “thinks” it has access to the entire machine.

To support the “process” abstraction, we need a lot of support in the OS kernel and CPU.

Review: Process State

Process == running program

A process has a “machine state”

- Memory – instructions and data == address space
- Registers – e.g. Program Counter, Stack Pointer
- I/O – open files
- Other OS resources

Review: OS controls program execution

Direct: allow a user process to run directly on hardware - "DOS" model. Can break:

- Security – can change files, settings, etc.
- Fairness – can use CPU forever
- Efficiency – may busy-wait for slow I/O

Solution: OS and hardware maintain some control over what processes can do.

Review: Controlling process execution

At system boot time, the OS has to set up trap handlers.

When processes (“jobs”) run:

- OS is involved during a sys call – privileged access
- OS is involved on timer interrupts to switch processes
- Processes “time-share” a single CPU

Q: What is a “context switch”? When is it used? Why?

Review: OS tracks program execution

- Each process is in one state at any single time:

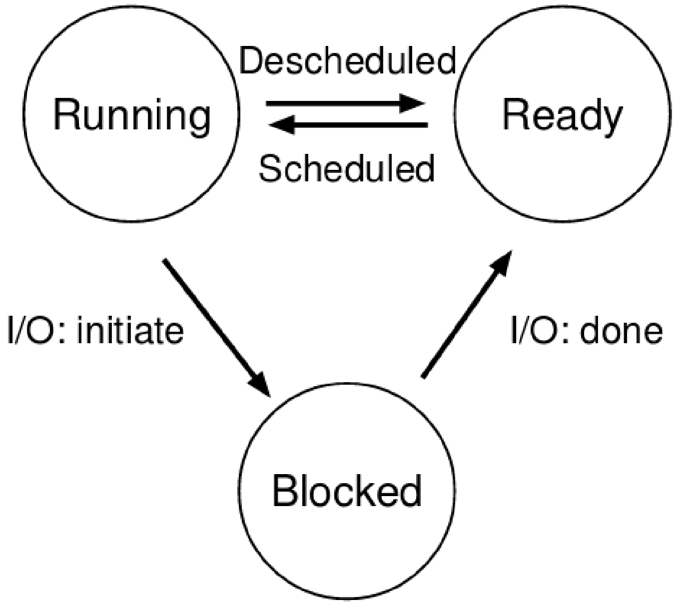


Figure 4.2: **Process: State Transitions**

Disk I/O:

HD: access time ~ seek time due to disk rotational latency = 60sec/10,000rpm ~ 6 ms.

SSD: seek time ~ 0.16 ms

How many instructions could be executed during these seek times?

Network I/O: depends on remote server – can be 0.1 to 10 ms

Controlling process execution schedule

Which process gets to run? → OS Scheduler

For now, we will assume that:

- a set of processes (“jobs”) all arrive at once.
- each uses CPU only – no I/O
- length of time each one needs is known in advance

We will evaluate different schedulers based on 1 metric:

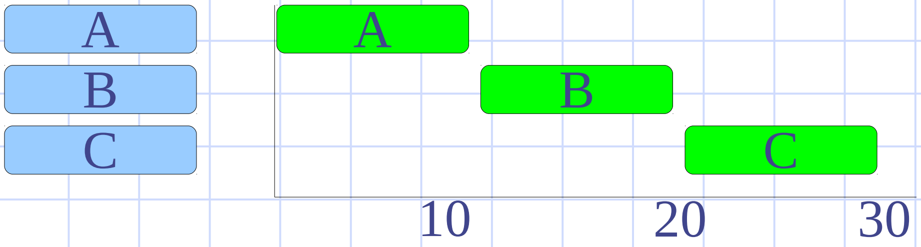
“turnaround” time for each job == $t_{completion} - t_{arrival}$

Scheduler #1: FIFO

FIFO == First in, first out == First come, first serve

Example: A, B, and C each require 10 time units.

- Queued up in any order, we get total run time of 30 units
- And individual turnaround times of 10, 20, and 30



B & C have to wait!

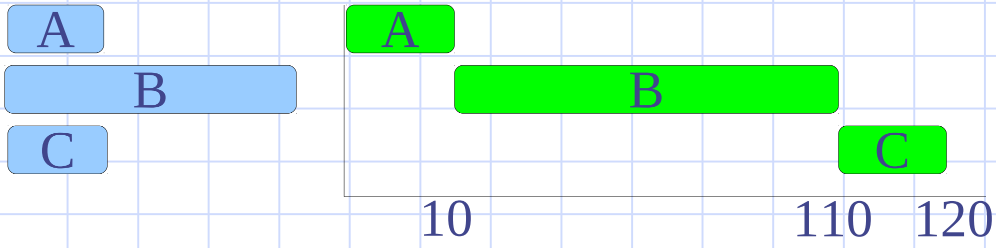
→ Average turnaround = 20

Scheduler #1: FIFO - First in, first out

Suppose jobs can have different run times

Example: A and C require 10 time units; B: 100

- Queued up in same order as before, we get total run time of 120 units
- And individual turnaround times of 10, 110, and 120



C really has to wait!

→ Average turnaround = **80**

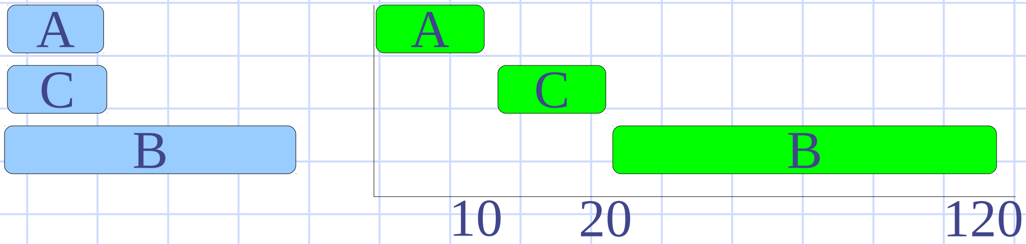
How can this be better?

Scheduler #2: Shortest Job First - SJF

Sort the jobs in increasing time requirements

Example: A and C require 10 time units; B: 100

- Queued up in SJF, we get total run time of 120 units
- And individual turnaround times of 10, 20, and 120



C doesn’t have to wait much

→ Average turnaround = **50**

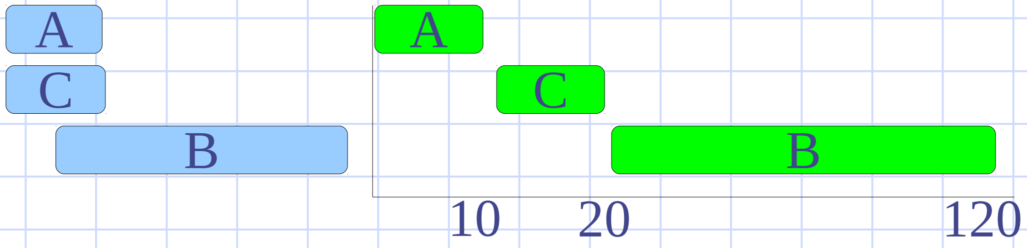
Better!

Allow jobs to come at any time

As the jobs come in, sort the remaining jobs in increasing time requirements

Example: A and C arrive at t=0 and require 10 time units; B arrives at t=5 and requires 100 time units

- Total run time of 120 units
- And individual turnaround times of 10, 20, and 115



Allow jobs to come at any time

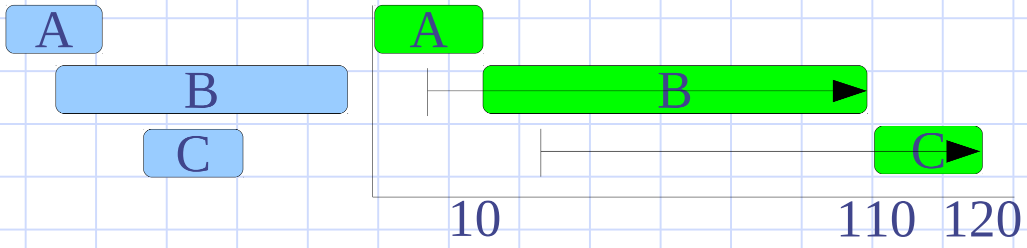
As the jobs come in, sort the remaining jobs in increasing time requirements

Example: A arrives at t=0 and requires 10 time units;

B @ t=5 requires 100 time units;

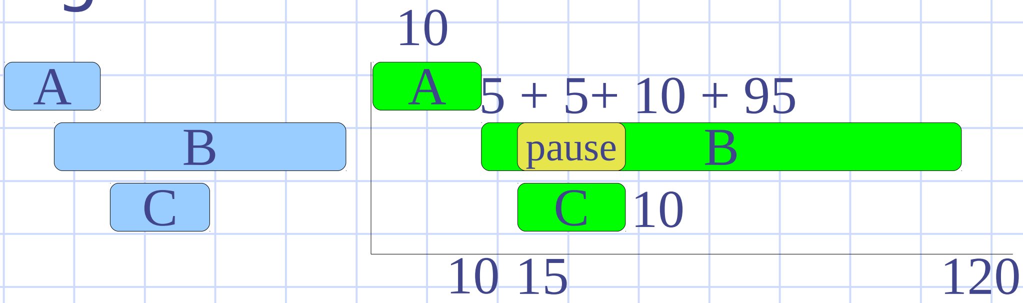
C @ t=15 requires 10 time units

- Individual turnaround times of 10, 105, and 105



Stop and start jobs

- We can see that we need to be able to "pause" a long job, run a small one to completion and restart the long one.



C doesn't have to wait; B does

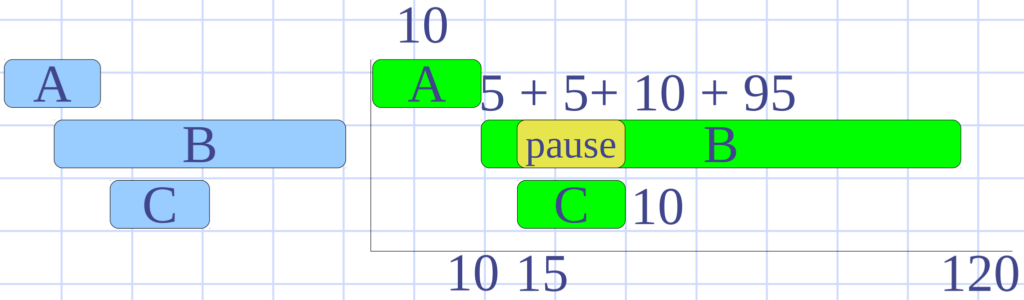
Turnaround = **135**

10 + 115 + 10

To "pause" a job is to "pre-empt" a running process, schedule another one to run instead and restart the waiting process.

Scheduler #3: Preemptive SJF (PSJF) or Shortest Time-to-Completion First (STCF)

If we can preempt and compute the shortest job every time we get a new one, we do better:



C doesn't have to wait; B does a bit

Turnaround = **135**

10 + 115 + 10

Response time

- For better interactive performance, we would want to use a different metric – response time: $t_{first\text{-}run} - t_{arrive}$

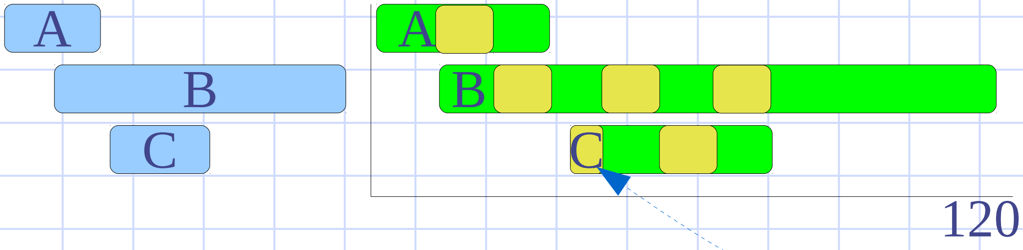
  i.e. how quickly can the system respond to each new input from a user?

FIFO, SJF, PSJF, and STCF are good at turnaround times but are all “bad” at response times.

New approach → Round Robin Scheduler: give each process a time slice on the CPU.

Round Robin

Give each process a time slice on the CPU.



C cannot pre-empt B
Each process (B) gets to finish its timeslice …
… unless the process is done before a timeslice is over.

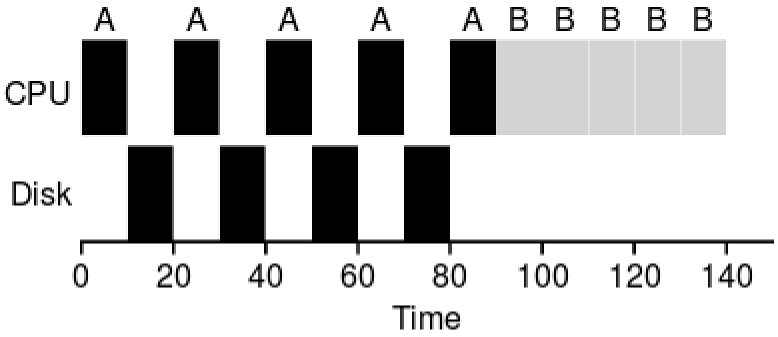
Low response time → better interactive performance

Turnaround time may not be so great

Allow slow I/O

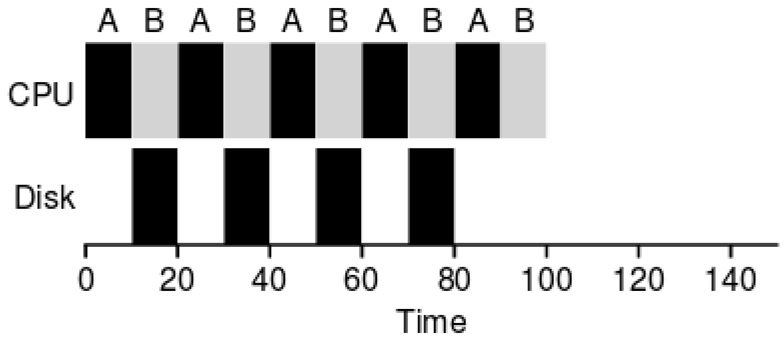
- Processes access slow I/O devices: disks, networks

- I/O can be very slow.



Overlap of I/O & CPU execution

When one process makes I/O request, allow another process to use CPU while the I/O is getting ready → "Overlap" of execution and I/O.



Results in better turnaround
 and better response times!

Lastly, don’t assume OS knows job times

- In most OSes, very little is known about how long a job will run.
- If we no longer assume the OS knows time requirements, we need some way to predict what processes will do next.

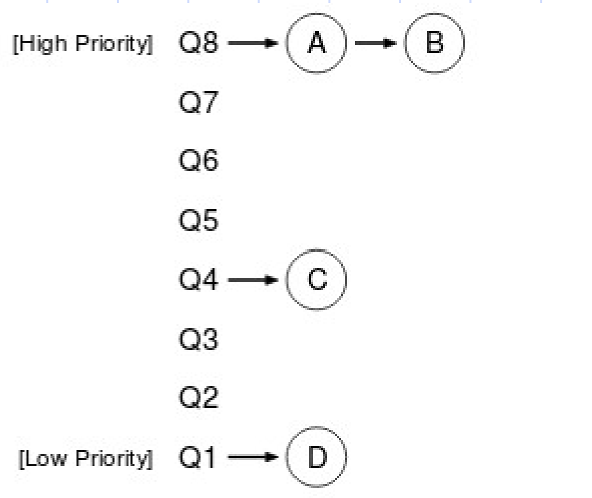
Best guess: history repeats itself → the near future will be like the recent past

→ Multi-level feedback queue Scheduler

Multi-level feedback queue Scheduler

- Set up many queues

- Each queue has a different priority level



Assume all processes are interactive and aim for good response times → use Round Robin scheduling at each priority level

Multi-level feedback queue (MLFQ) Scheduler

There are 2 main problems MLFQ tries to address:

- Optimize turnaround time
- Minimize response time

But do these without knowing how long each process will need to run!

Key to MLFQ: by monitoring each process, MLFQ “learns” the characteristics of each process

Predicts future behavior from past performance

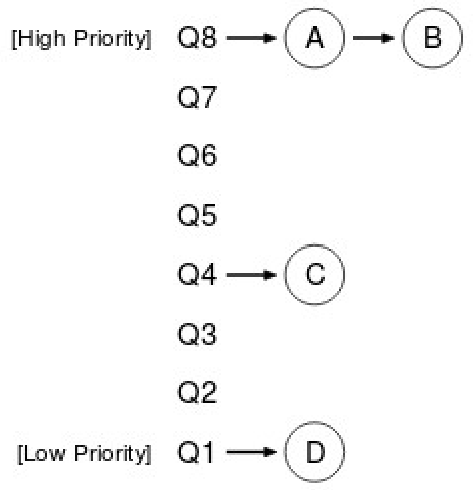
MLFQ rules

If we had to decide which of two jobs, X & Y, to run:

Rule #1: if Priority(X) > Priority(Y), X runs, Y has to wait

Rule #2: if Priority(X) = Priority(Y), X & Y are in RR

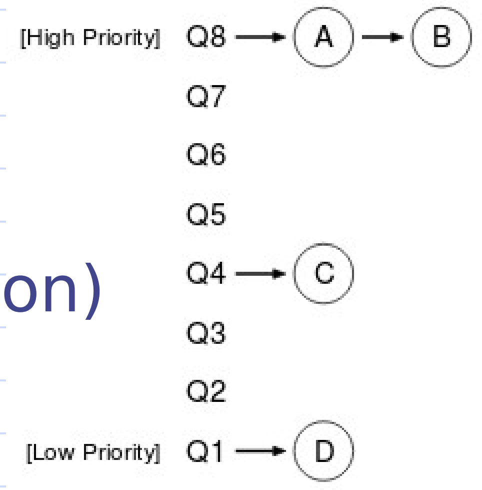
In this example, A and B run,
C and D have to wait.



Key: How can we change priorities over time so that the scheduler is "fair" to all?

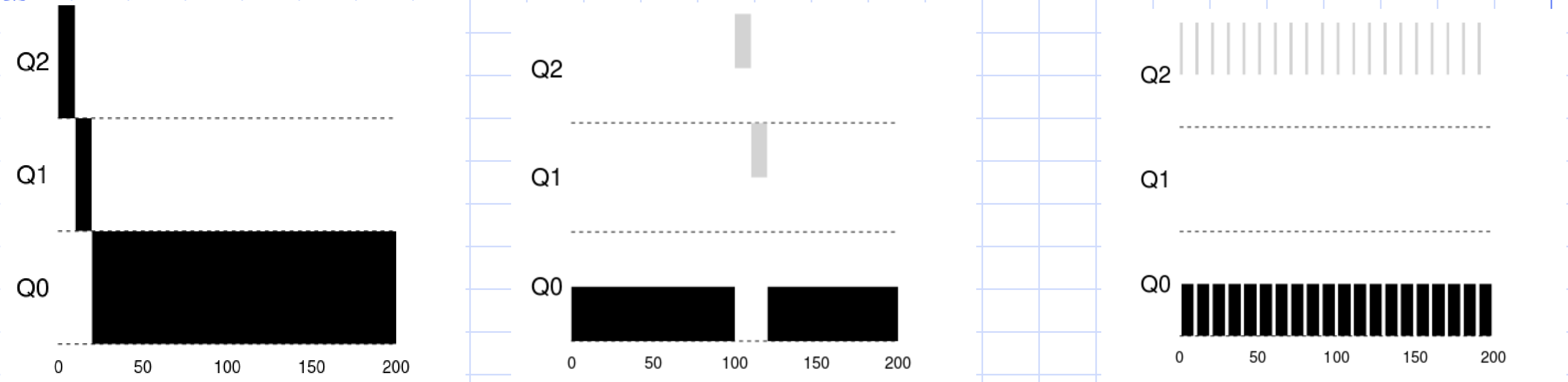
MLFQ rules for priority changes (Attempt 1)

- Need rules for changing priorities of jobs



- Rule 3: When a new job starts, put it at the highest level (but no preemption)
- Rule 4a: Each time the job runs for a timeslice, reduce its priority
- Rule 4b: If a job is interrupted before its timeslice is up, no priority change

Result of attempt 1 for MLFQ priority rules



Long-running Job Over Time

Along Came An Interactive Job

Mixed I/O-intensive and CPU-intensive

Seems to work:

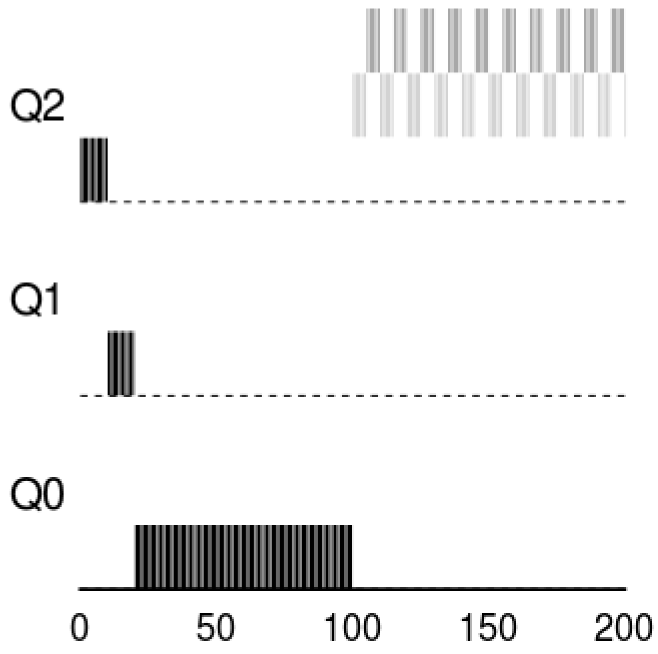
- Long running jobs move to lowest priority
- Short jobs get to run to completion even with high I/O

Problem with MLFQ priority rules attempt 1

One or more interactive jobs could consume all the CPU time

→ long running jobs may not get any CPU time == “starvation”

Can fool the scheduler: a process can do some I/O just before the end of a timeslice to keep its priority.

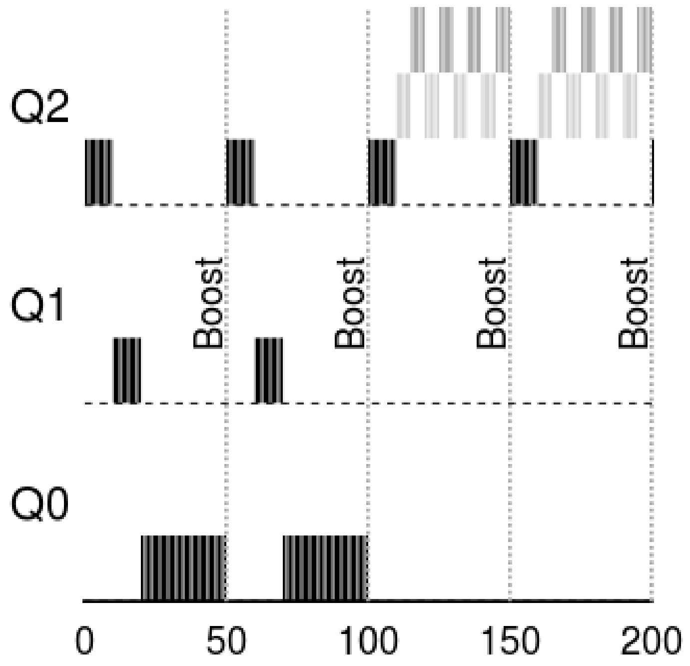


Fix: Attempt #2 MLFQ priority rules

To avoid starvation:

Rule #5: After some period of time, S, boost all jobs to highest priority

A CPU-bound process that goes into an interactive phase will also get to be more responsive when boosted.



Fix “bad” jobs: Attempt #3

To prevent a “bad” process that does I/O to maintain its priority fix Rule #4:

Keep track of CPU usage, when a process gets to run for (a total of) its timeslice, reduce its priority (regardless of how it got to run).

MLFQ “works” - it is used in Windows, Linux, BSD, Solaris, etc.