EECS-343 Operating Systems
Lecture 4:
Scheduling

Steve Tarzia

Spring 2019

Northwestern

Announcements

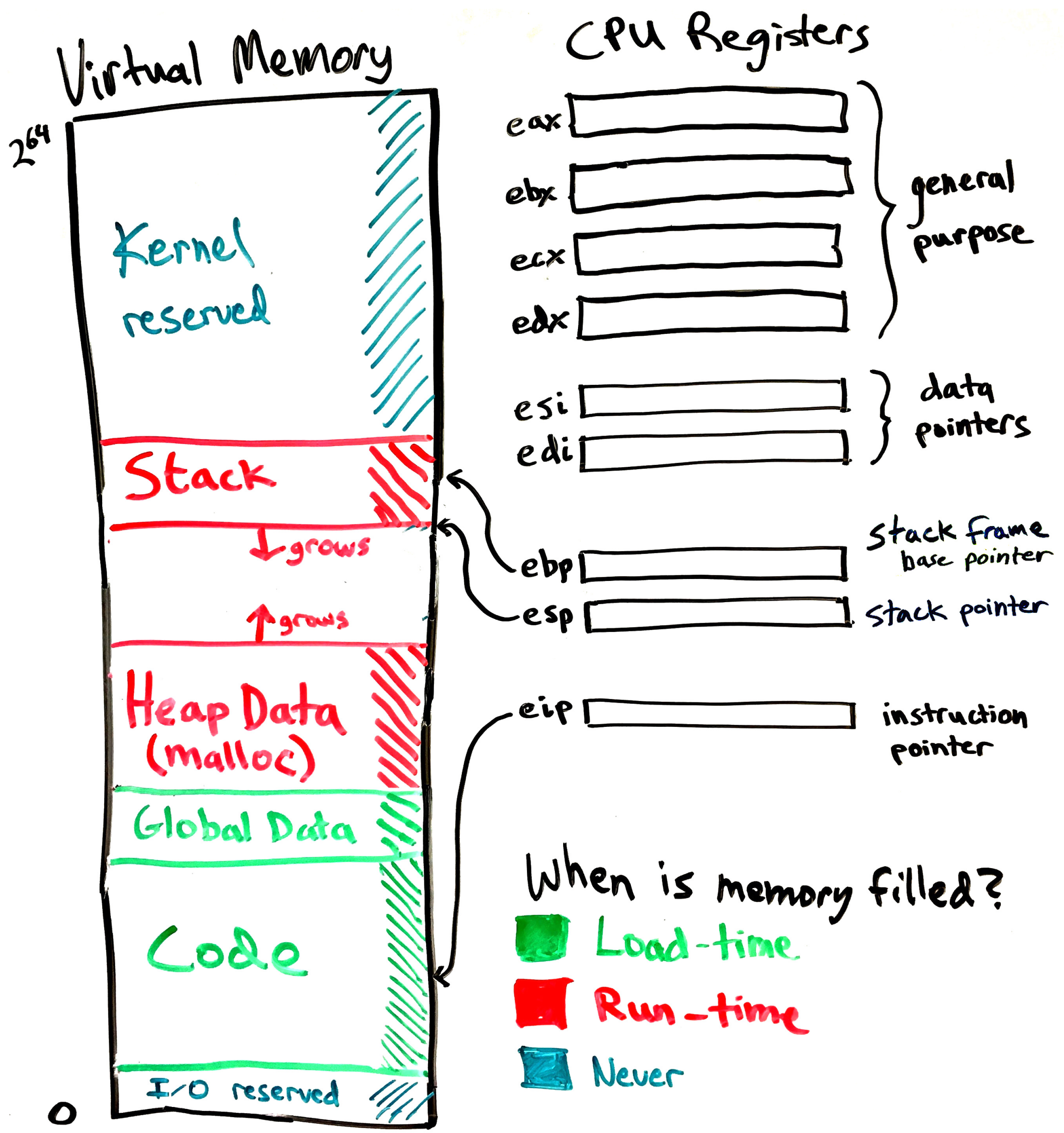
- Project 1 due on Monday
- HW1 due the next Monday (22nd).

Recap

- xv6 OS code is written for the Intel x86 CPU architecture, but…
- Linux supports 31 different CPU architectures
    - Low-level *mechanisms* are different on each arch.
    - High-level *policies* are the same for all.
- *Fork* syscall: run once, exits twice!
- *Nondeterminism* is when a program's output is unpredictable
- OS process scheduler can create *race conditions* in programs that rely on an interaction of multiple processes.
    - These are tricky to debug, because they are sensitive to timing (*Heisenbugs*).
- *Kernel panic* occurs when OS causes an exception and can't recover

Recap (continued)

- **fork** + **exec** runs a program.
  - fork duplicates the current process
  - exec copies code and global data from an executable file, and creates a new empty stack.
- Stack grows from high addresses down to lower.
  - Grows larger when a function is called.
  - Shrinks when a function returns.
- Heap is a block of memory managed by C's malloc & free.



Scheduling

- We have talked about the ***mechanisms*** for sharing the CPU:
    - Limited direct execution
    - User/kernel mode
    - Timer interrupts
    - System calls

- **Scheduling** is creating a ***policy*** for sharing the CPU:
    - Which process is chosen to run, and when?
    - When (if ever) are running processes preempted (interrupted)?

We'll begin with a simplified scheduling problem

Let's take ideas from Operations Research (process == "job")

Simplifying assumptions:

1. Jobs are the same length
2. No new jobs are added (they all are available at the beginning)
3. Jobs cannot be ***preempted*** (interrupted)
4. No I/O is done; it's just CPU work
5. Job length is known ahead of time
6. There is only one CPU
7. All processes have equal priority

Metrics

- A ***metric*** is a standard for measuring something
  - Like an "objective function" in mathematical optimization,
  - or a "utility function" in economics.
- We must choose a metric *before* designing a scheduling policy
  - Computing systems have many different goals and uses, so there are many competing performance metrics.
- Operating systems (and life) are full of *tradeoffs*

"fast acting" or *"long lasting"*?
Do I want to feel better now or later?

Average ***turnaround time*** is our first scheduling metric

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

- It's just the total time waited to finish the job, including both it's execution time and the time it was waiting before execution.
- *Average* turnaround time is computed across all processes.

First in, first out (FIFO)

- FIFO is the simplest scheduling policy.
- Just let a job run until it is done, then schedule the next job

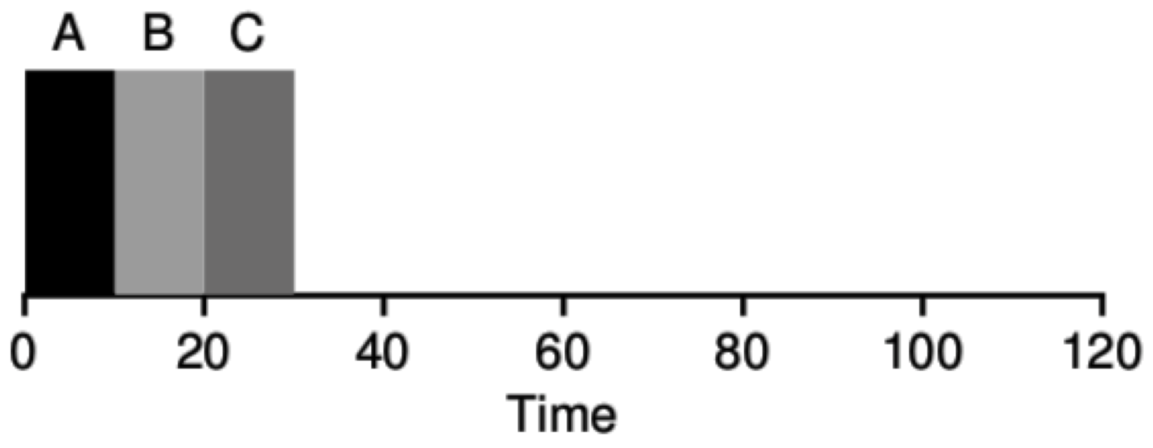


Figure 7.1: **FIFO Simple Example**

- Average turnaround time here is $(10 + 20 + 30)/3 =$ **20**

FIFO shortcomings

- FIFO is like a grocery store with one checkout line
- One big job can cause lots of jobs behind it to wait

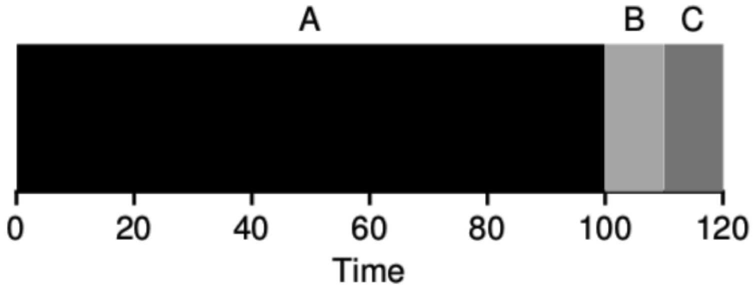


Figure 7.2: **Why FIFO Is Not That Great**

- Above, avg turnaround time = **110**
- ***Convoy effect*** – lots of small jobs getting stuck behind a big one

Photo from https://www.flickr.com/photos/countryluvinchix/4013902615

Shortest Job First (SJF)

- Start with the smallest jobs to minimize the number of *waiting* jobs
- Minimizing waiting will minimize average turnaround time

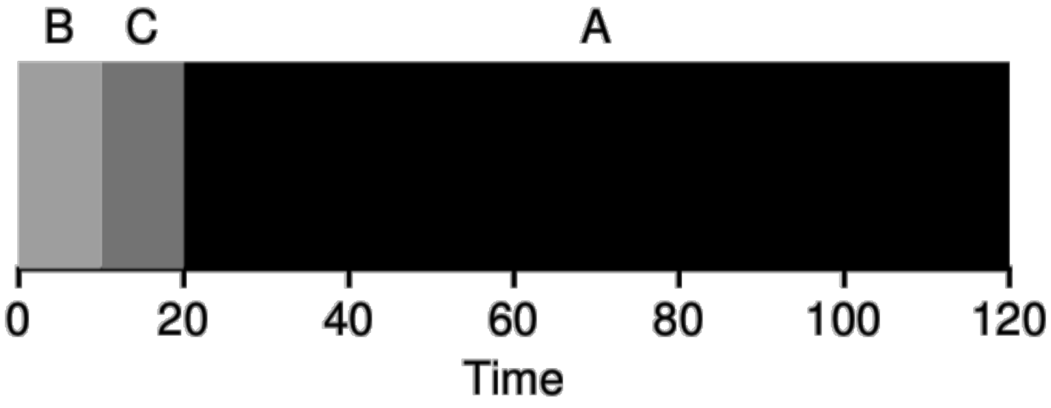


Figure 7.3: **SJF Simple Example**

- Above, average turnaround time = (10 + 20 + 120) / 3 = **50** 😆
  - Compare to 110 for FIFO

Let's get real

- Allow new jobs to be added ***after*** the start (drop assumption #2)
- Now, we can suffer from long waits even with Shortest Job First!

SJF fail:

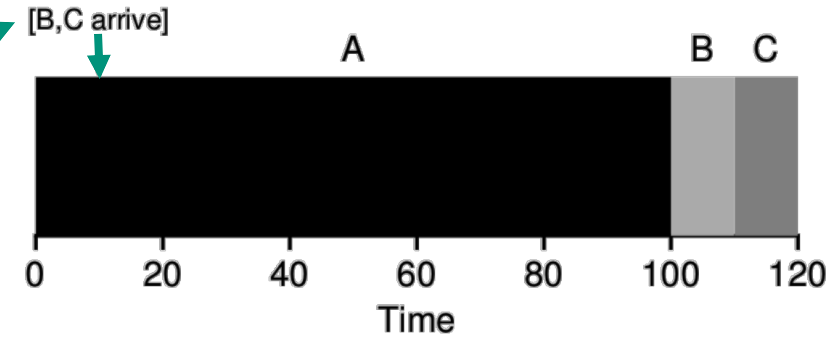


Figure 7.4: **SJF With Late Arrivals From B and C**

- If B & C arrive late, they will have to wait because we already scheduled job A, and jobs must finish once they start (assumption #3)
- Average turnaround time = (100 + (110-10) + (120-10))/3 = **103.3** 😞

Shortest Time-to-Completion First (STCF)

- Let's give our schedule the power to ***preempt*** jobs
  - Preemption is pausing a job to run another one (word "interrupt" was taken)
- Shortest Time-to-Completion First causes scheduler to:
  - reevaluate all the jobs when a new one arrives
  - schedule the job with the shortest remaining time

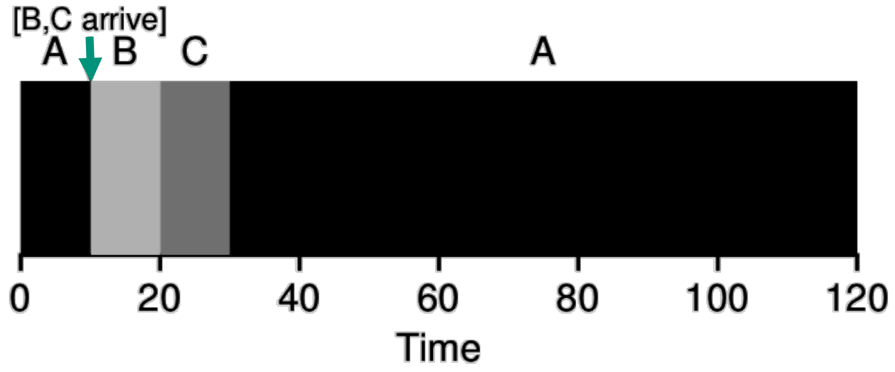


Figure 7.5: **STCF Simple Example**

- After B & C arrive, A is no longer the shortest time-to-completion job.
- Avg turnaround time = (120 + 10 + 20) / 3 = **50**

A different metric – ***response time***

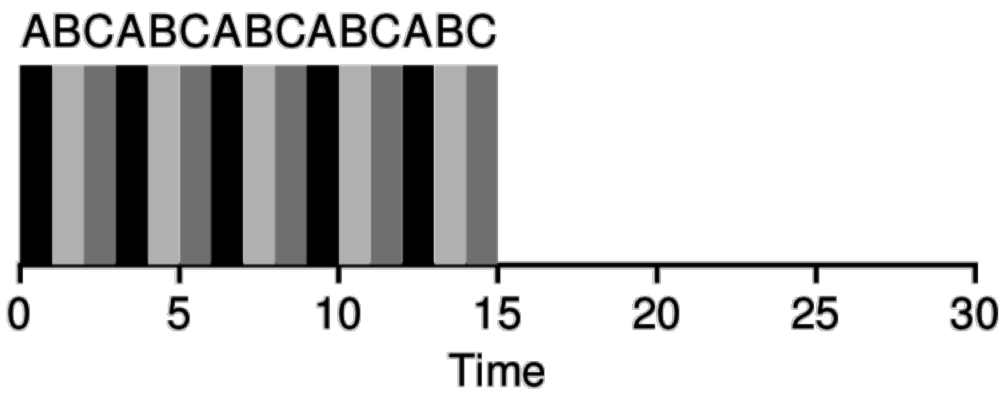
- STCF gives optimal avg turnaround time
- But long jobs may wait a long, long time and this may be undesirable
- ***Response time*** metric minimizes the time we wait for a job to ***start***:

$$T_{response} = T_{start} - T_{arrival}$$

- But we do **not** care how long it takes to *finish* a job
- This is good for interactive processes (GUI) which must quickly show that they are reacting to user inputs, but can service requests slowly

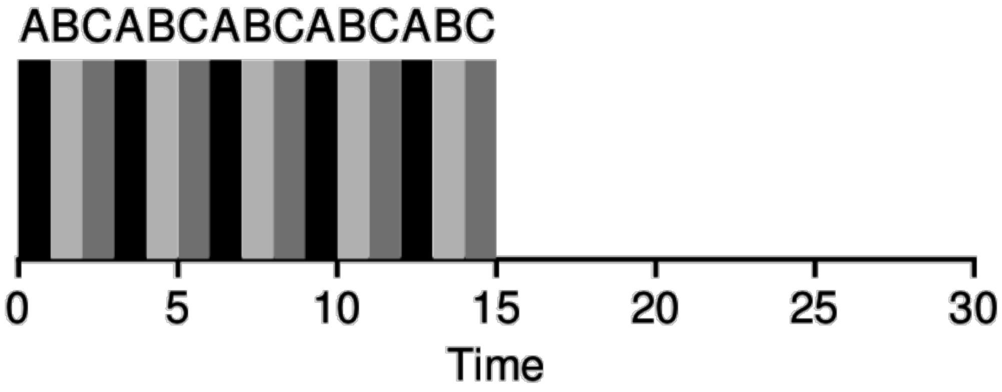
Round Robin optimizes response time

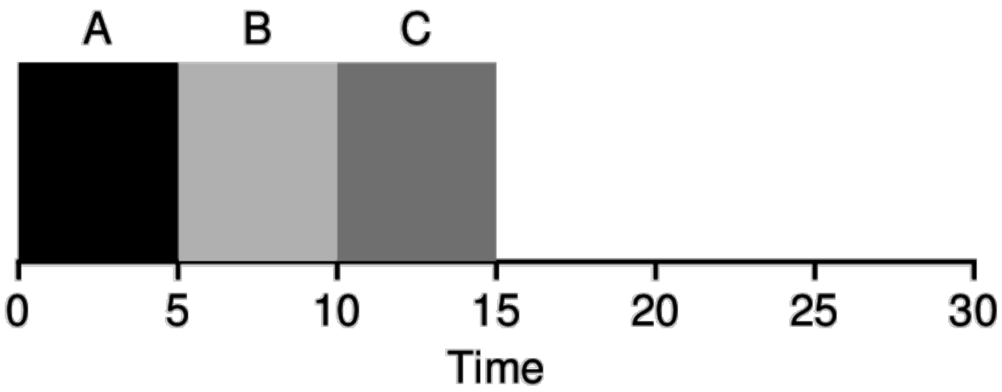
- ***Round Robin*** (RR) scheduling runs a job for a small *time slice*, then schedules the next job:



- Above, avg response time = (0 + 1 + 2) / 3 = **1**
  - In general, avg response time = (num_jobs – 1) * time_slice / num_jobs
- Smaller time slice means smaller response time

Different policies favor different metrics





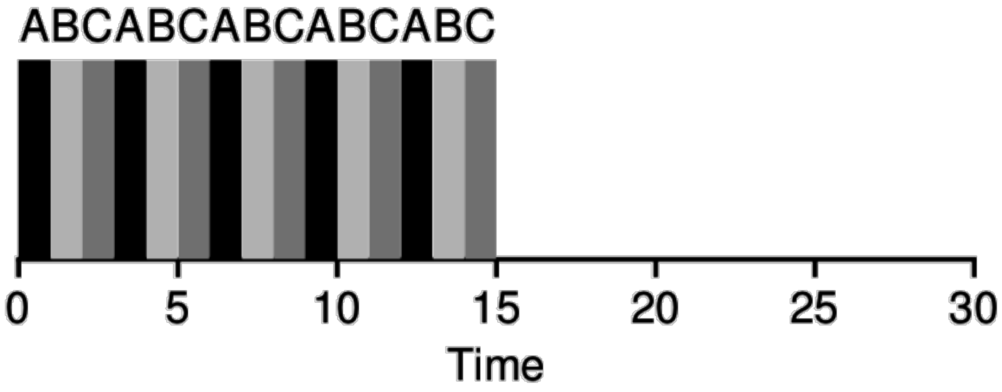
Round Robin scheduling:

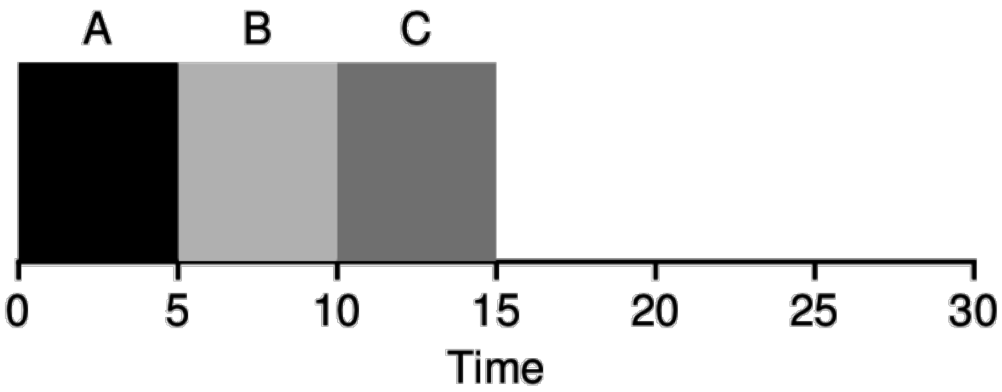
- Avg turnaround time = **14**
- Avg response time = **1**
- Context switches = **14**

Shortest Job first or ***STCF***:

- Avg turnaround time = **10**
- Avg response time = **5**
- Context switches = **2**

Time slice (a.k.a. time quantum) tradeoffs





Round Robin scheduling with time slice = 1:

- Avg response time = **1**
- Context switches = **14**

Round Robin scheduling with time slice = 5:

- Avg response time = **5**
- Context switches = **2**

Better response time vs. Less context switch overhead

Context switching overhead

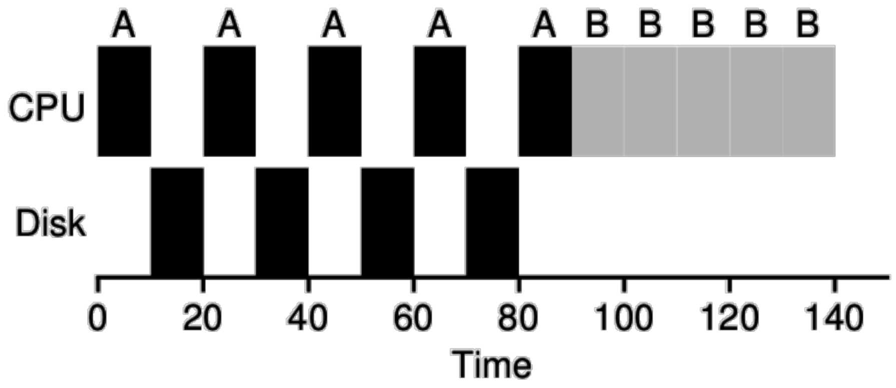
- We might expect context switches to be very quick because it just involves switching a few registers.
- However, there is a large cost in "warming" the CPU's ***caches***.
- Caches store copies of recently-used memory on the CPU itself
    - L1, L2, L3 memory cache
    - Translation Lookaside Buffer (TLB) is a cache of recent page mappings (it's a cache of the current page table)
    - Execution speed is often dominated by memory access, so this is important
- New process will use totally different physical memory locations, so all the cache data is useless to the new process.

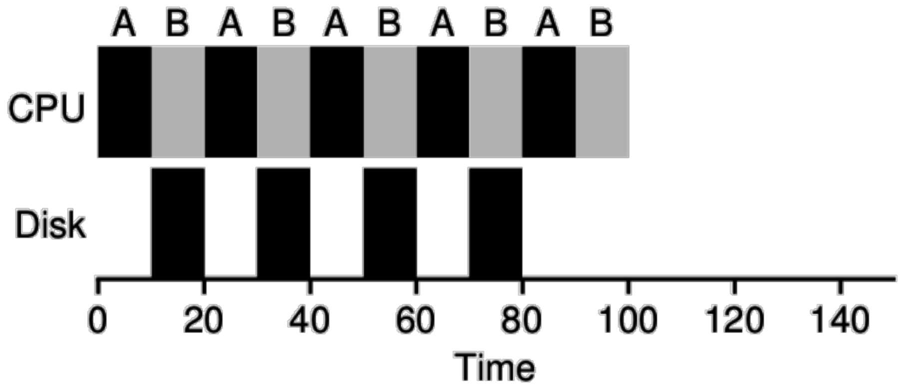
Intermission

"No, Thursday's out. How about never—is never good for you?"

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

- If process A does I/O every ten milliseconds and each I/O takes 10 ms, then the CPU is free during those I/Os:
- A is ***blocked*** during it's 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:
- Scheduler should favor processes that will do I/O soon because I/O frees the CPU and makes use of other hardware.





Blocked processes are actually making progress, but not using the CPU.

I/O bound and CPU bound processes

- We say a process is ***CPU bound*** if it needs lots of CPU time to progress
  - These processes have a lot of logic and math.
  - Usually in *running* or *ready* state
- A process is ***I/O bound*** if it needs to do lots of I/O to progress
  - These processes access disk, network, etc.
  - or they are ***interactive***, spending most of their time waiting for the next user input
    (from the keyboard, mouse, or touchscreen)
  - Usually in the *blocked* state

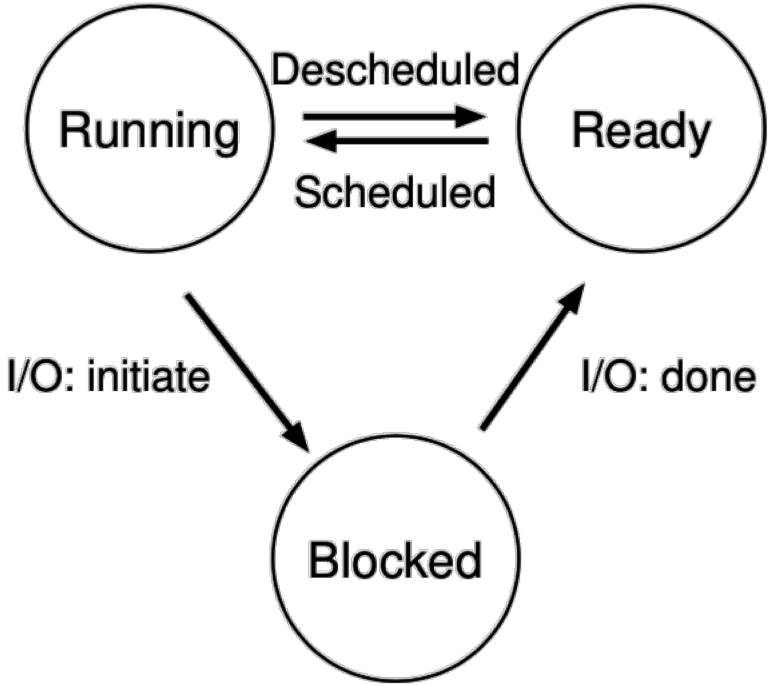


Figure 4.2: **Process: State Transitions**

Real OS Schedulers

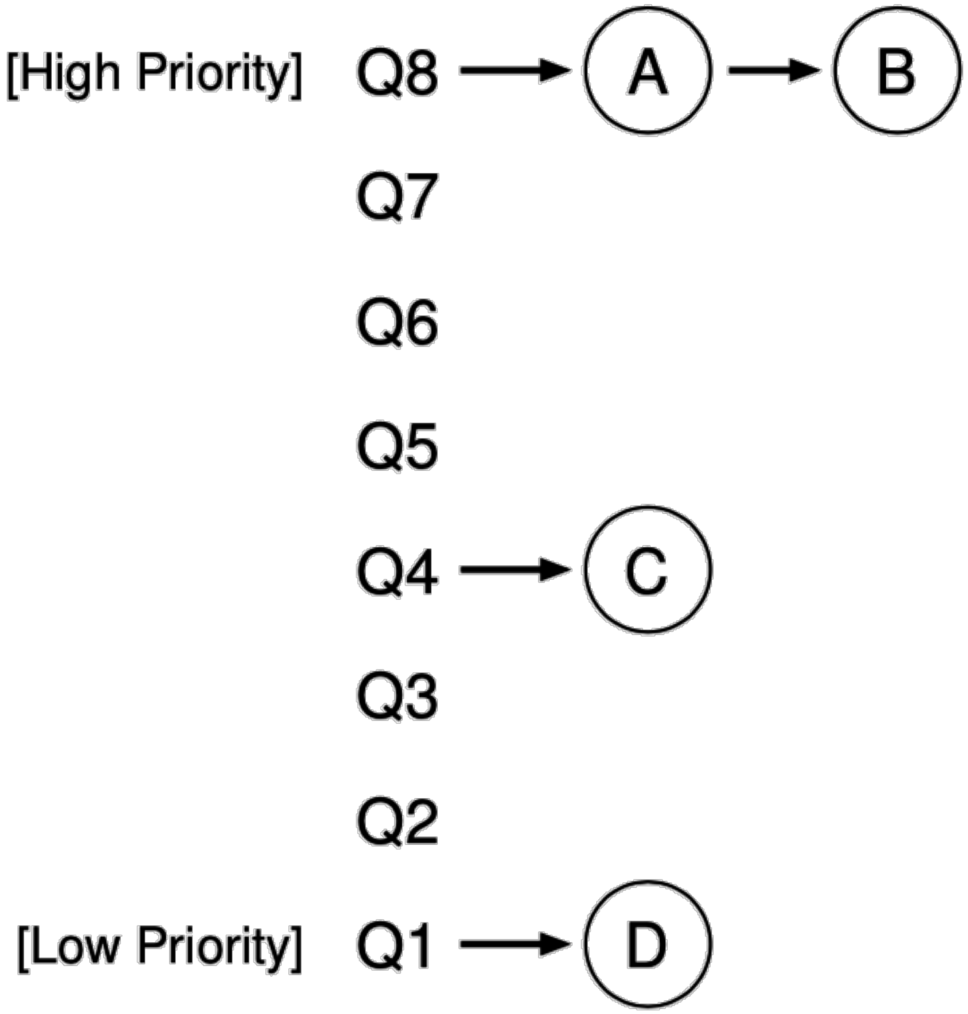
- In reality, we don't know the future behavior of processes
  - How long will a process run?
  - When will it perform I/O next?
- However, we can track past behavior and assume future will be similar

- Usually we want a policy that ***balances*** response time and turnaround time, and without too much context switching ***overhead***
- Interactive processes should usually be prioritized, because they will use little CPU, but make the system feel responsive.

- Xv6 uses a simple round-robin scheduler, but that's not realistic

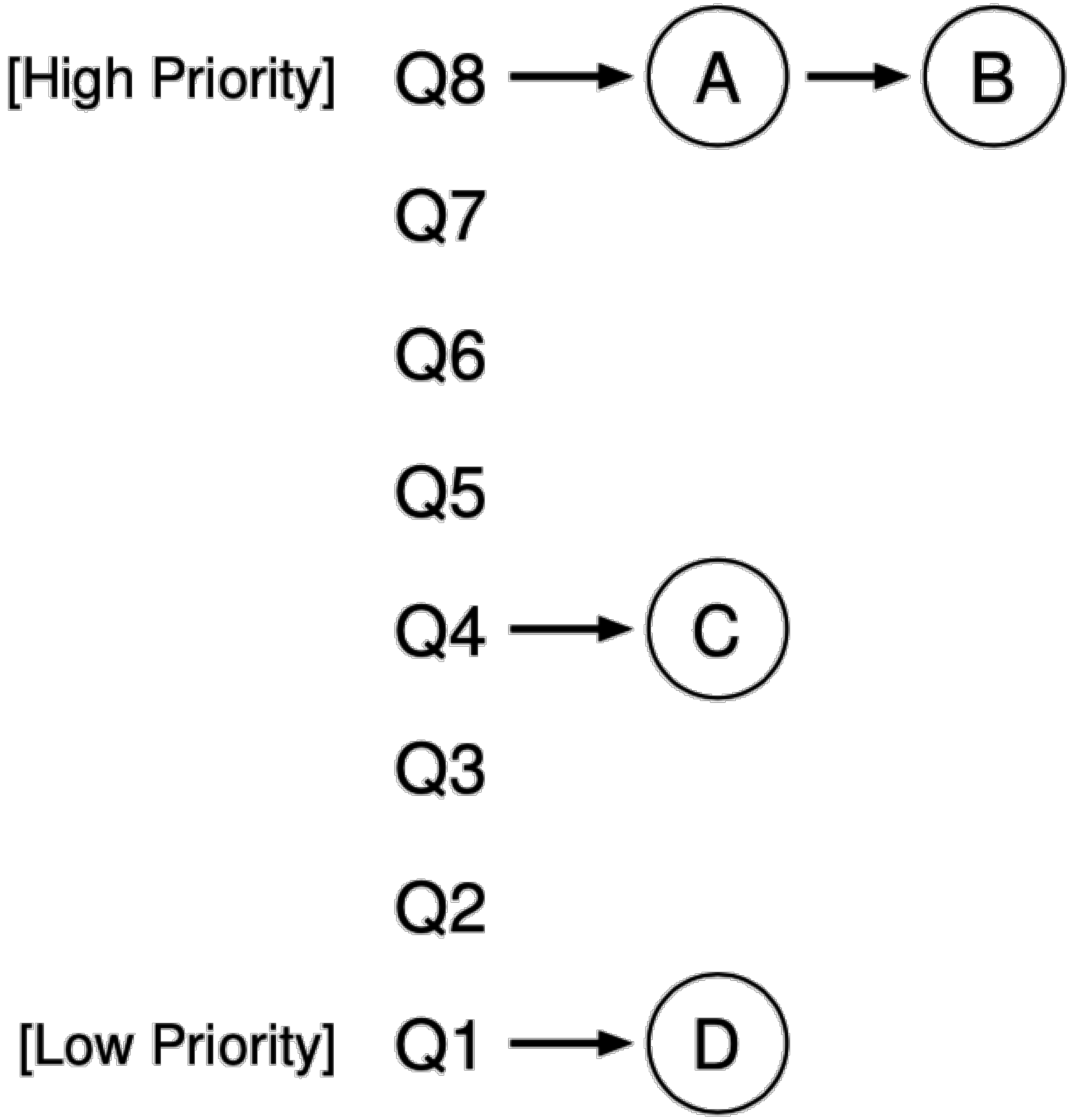
Multi-Level Feedback Queue (MLFQ)

- Several run queues, with varying priority
- Keep interactive jobs in ***high priority*** queues
- Processes at a given level are Round Robin scheduled
- Always run the highest priority processes
- Run lower-priority processes when all higher processes are blocked.
- Over time, processes ***lose*** and ***gain priority***
  - Each process has a CPU usage quota at a given level. When used up, it moves down one level.
  - Periodically reset by moving all processes up to highest priority.



MLFQ rules

1. If Priority(A) > Priority(B), A runs (B doesn't).
2. If Priority(A) = Priority(B), A & B run in RR.
3. When a job enters the system, it is placed at the highest priority (the topmost queue).
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).
5. After some time period S, move all the jobs in the system to the topmost queue.



MLFQ parameters

- Round-robin time slice
- Number of levels
- CPU time quota at each level
- Reset interval (*S*)

Note that we can use a formula to calculate a process' current priority level if we know the amount of CPU time used in the past ***S*** seconds.

Avoiding ***starvation***

- Low-priority processes are said to ***starve*** if they never are given a chance to run.
- MLFQ avoids starvation by periodically boosting all process' priorities (rule 5).

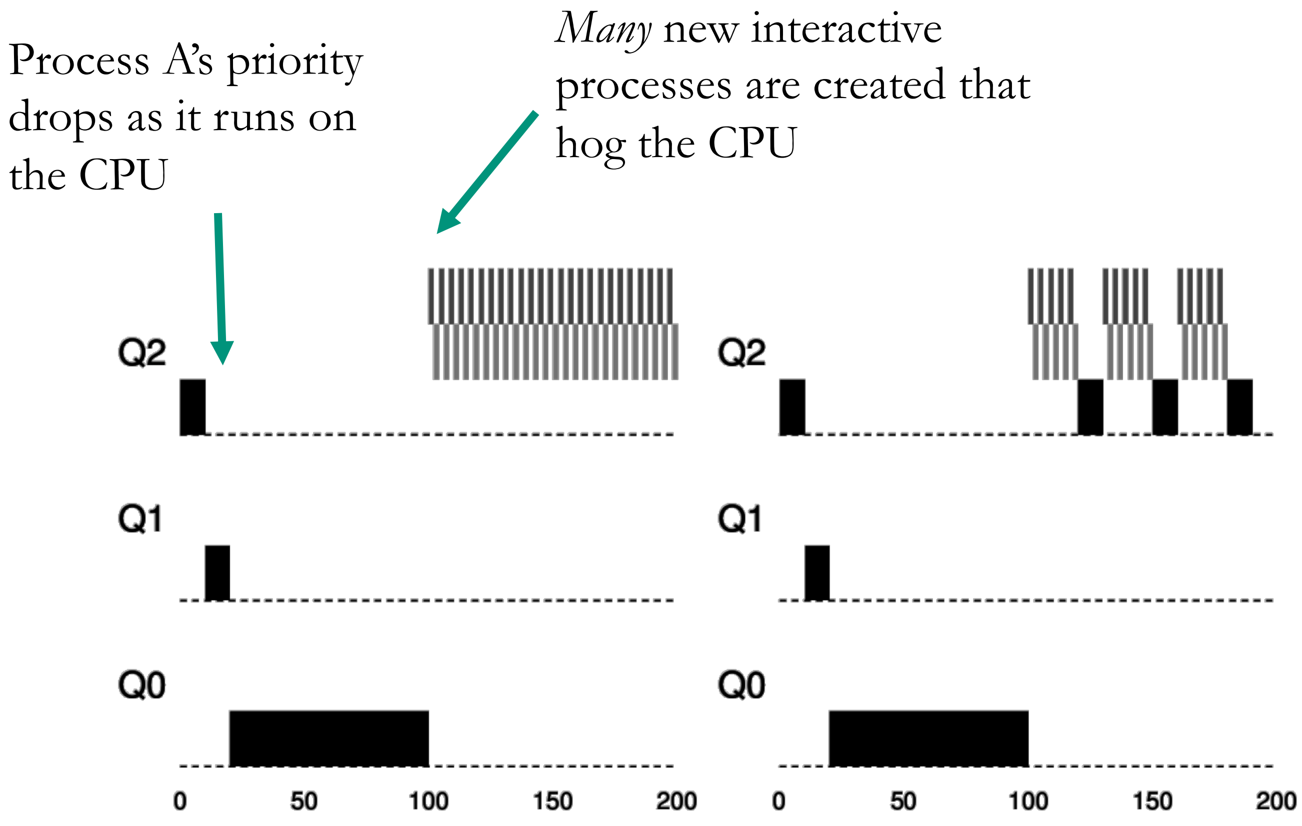


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

*This diagram shows a simplified version of MLFQ

An MLFQ optimization

- Lower priority processes are CPU-bound, not interactive, so we can use longer time slices (quanta) to minimize context switches:

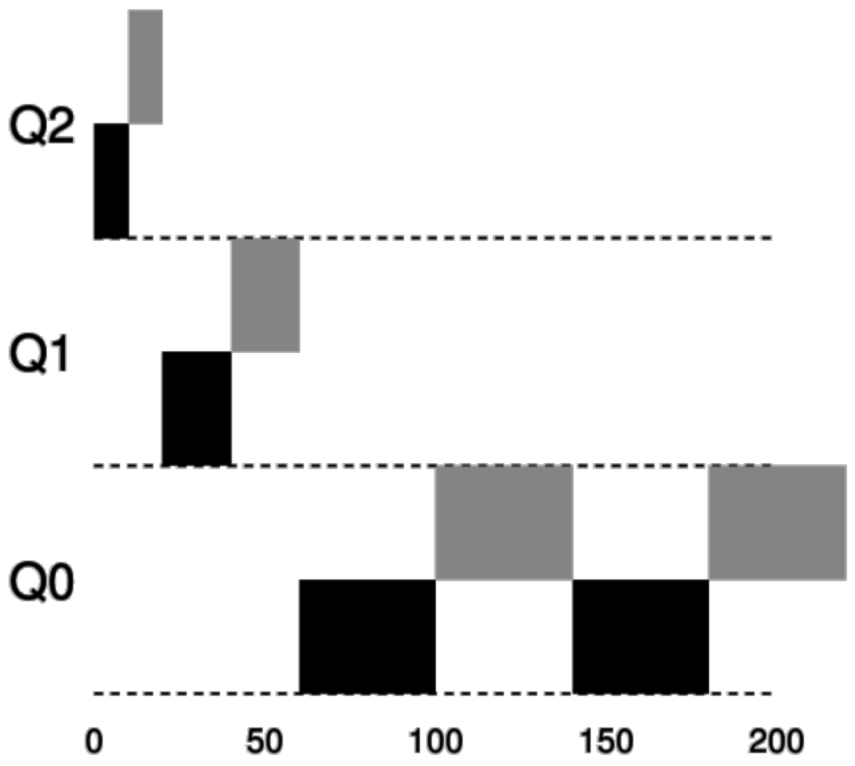


Figure 8.7: **Lower Priority, Longer Quanta**

User-defined process priority

- MLFQs are designed to automatically favor the right processes
  - But sometimes it makes sense to give the OS some scheduling hints
- Most OSes also have a way for users to specify a process' "priority"
- Eg., `nice` command on Unix
- User-specified priority can change the MLFQ behavior
- For example, if the user marks a process as "low priority" then
  - MLFQ may reset it to one of the middle levels instead of the top level.
  - May give it a smaller CPU quota at each level
- The OS may also treat system (root) processes with higher priority

Context switch mechanisms revisited

- Recall that OS takes over when an interrupt occurs
- At this time, it can use its scheduling algorithm to determine which process should run next.
    - Can return to the same process, or
    - Can ***context switch*** to a different process
- Programmable **timer** should be set to the scheduling **time slice** (or a multiple of it) to give the OS scheduler an opportunity to run.

Recap

- Defined two conflicting metrics: ***turnaround time*** and ***response time***
  - Cannot optimize both – must tradeoff, or balance, the two
- Optimized by ***shortest job first*** and ***round robin***, respectively
- Context switching overhead is due to the CPU caches
  - CPU keeps most recently used data in nearby caches, so it's more efficient to let an ongoing process continue.
- ***I/O-blocked*** processes make progress without using the CPU
  - We should prioritize I/O-bound processes
- ***Multi-Level Feedback Queues*** are often used in real OS schedulers
  - Prioritizes "polite" processes that use little CPU time when scheduled
  - CPU-bound processes squander their time quotas and lose priority