# Systems 1 – CS 6013

## Computer Architecture and Operating Systems

# Lecture 12: Scheduling (Part 2) and Inter-process Communication (IPC)

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024



*Adapted Ryan Stutsman's slides.

# Lecture 12 – Topics

- Scheduling
  - Note: Review book chapters 8, 9
- IPC – Inter-Process Communication

# Miscellaneous



- Reminders
  - Week 4 Vocabulary Assignment - Due Tonight
  - Week 5 Reading Assignment (Chapters 12-15)
- Lab after lecture today
  - It will help prepare you for the Shell assignment
- Emacs is cool… Emacs vs PowerPoint
  - What does a PowerPoint slide look like?

# Scheduling Basics



**Workloads:**
- Arrival Time
- Run Time

**Schedulers:**
- FIFO
- SJF
- STCF
- RR

**Metrics:**
- turnaround time
- response time

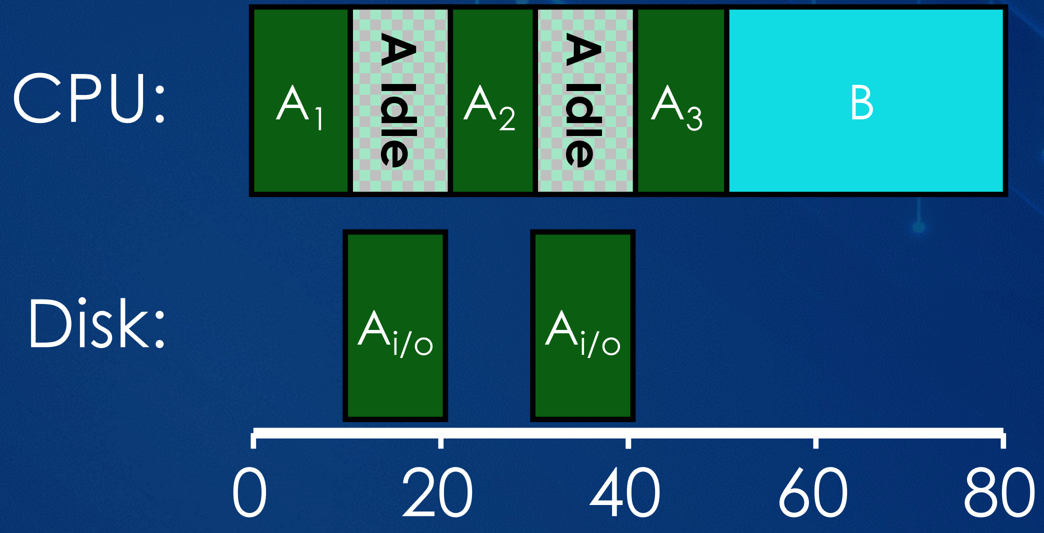
* First In, First Out; Shortest Job First; Shortest Time to Completion First; Round Robin

# Workload Assumptions



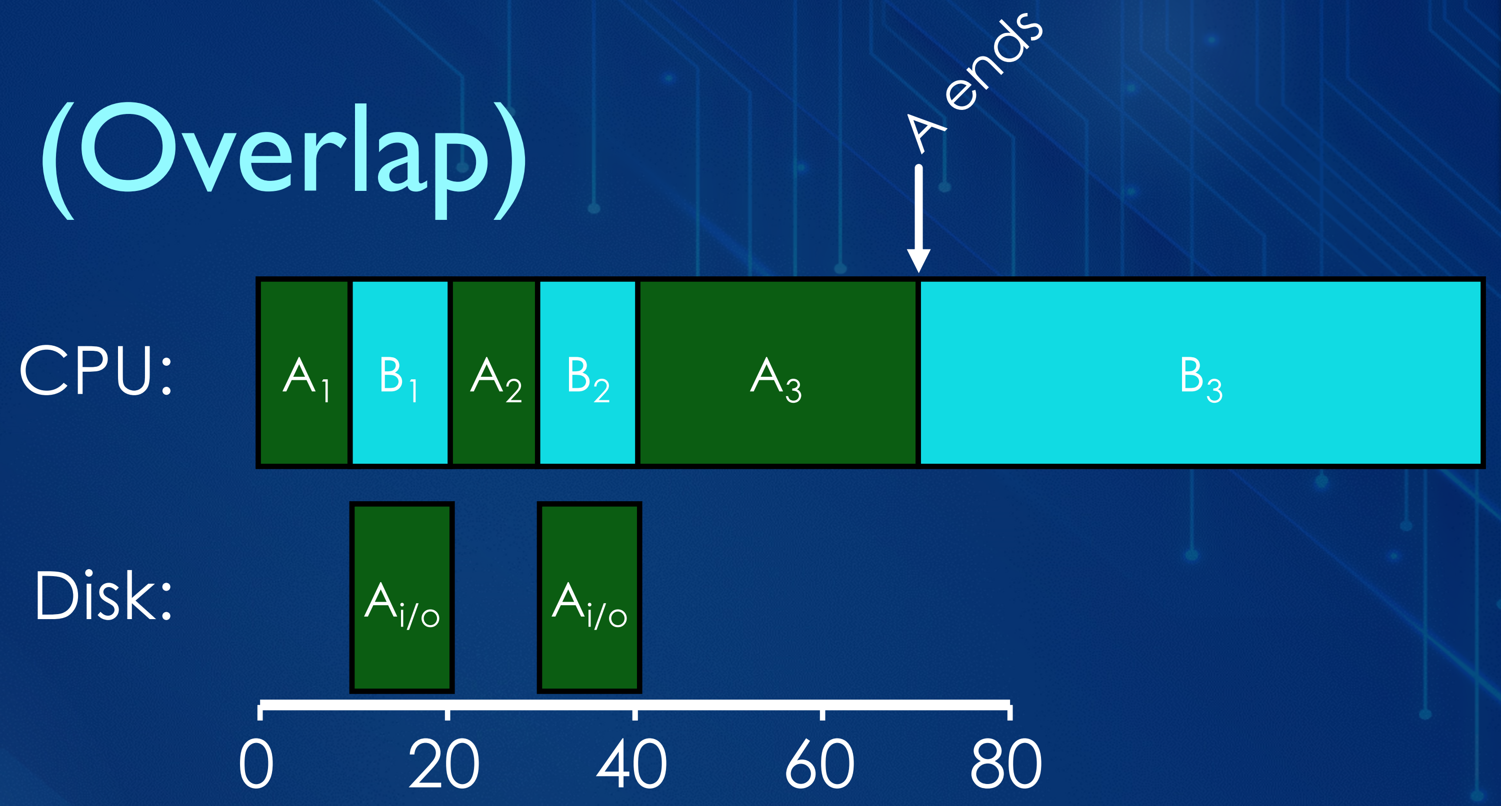
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. The run-time of each job is known

# What Happens if Scheduler is Not I/O Aware




- If the Scheduler is not aware of a Process doing I/O…
- Don't let Job A hold on to CPU while blocked waiting for disk

# I/O Aware (Overlap)




Treat Job A as 3 separate CPU bursts (sub-jobs)
When Job A completes I/O, "another" Job A is ready

Each CPU burst is shorter than Job B, so with STCF, Job A preempts Job B

# Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. ~~The run-time of each job is known~~

- We've used these assumptions to exam the basics of scheduling… But with modern systems, these assumptions do not apply, so…
- We need a smarter, fancier scheduler!

# MLFQ (Multi-Level Feedback Queue)



- Design Goal: general-purpose scheduling
- Must support two job types with distinct goals:
  - interactive programs care about response time
  - batch programs care about turnaround time
- Approach: multiple queues, each for a different priority level.
- Higher queue has a higher priority.
- If more than one job in one queue, use Round Robin (RR) on that queue.

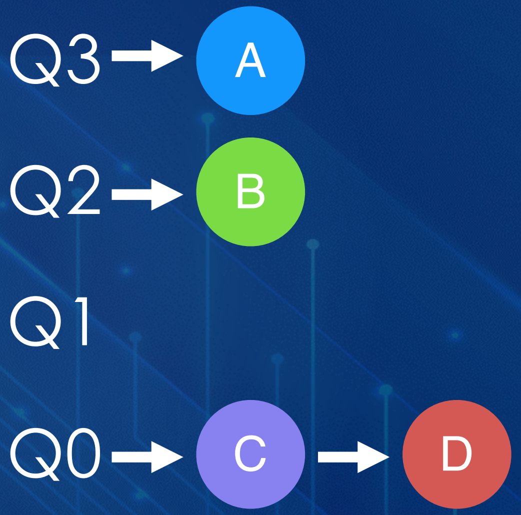
# Priorities

Q3 has the highest priority

Rule 1: If Priority(A) > Priority(B), A runs

Rule 2: If Priority(A) == Priority(B), A & B run in RR

More rules to come…




Can think of this as a "Multi-level Priority Queue"

How do we set priorities?

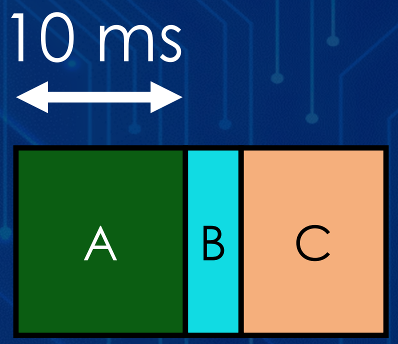
# History of Process' Past Behavior



- Use past behavior of process to predict future behavior (Heuristic)
  - Common technique in systems
- Assumption: Most processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

# More MLFQ Rules

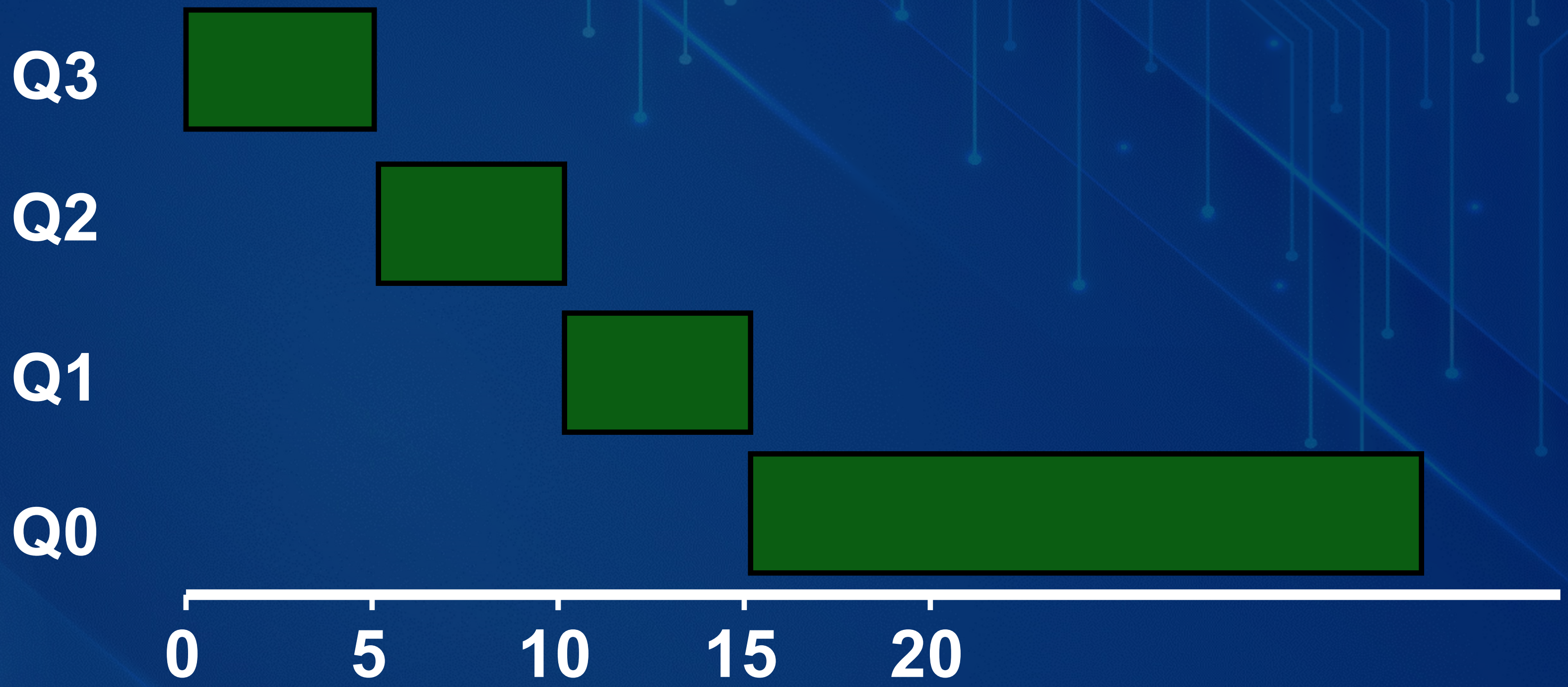





- Rule 1: If priority(A) > Priority(B),
  A runs
- Rule 2: If priority(A) == Priority(B),
  A & B run in RR
- Rule 3: Processes start at top priority (highest queue)
- Rule 4: If job uses its whole slice, demote process,
  - else leave at same level.
    - What does it mean if this case occurs?
      - Process is **not** doing I/O

# One Long Job – MLFQ Example

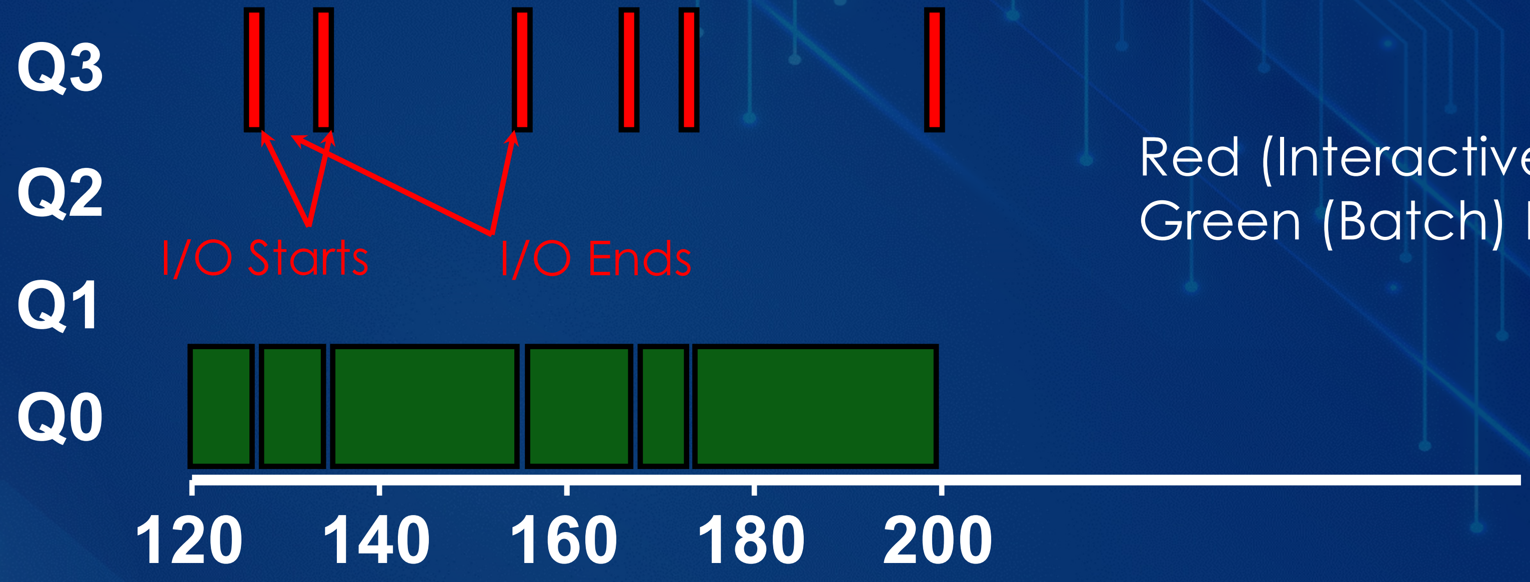





- Process uses entire time slice. What happens?
  - Drops in priority…
  - Ends up at the bottom of the priority queue…

# Example With Two Processes - MLFQ

Notes:
- *10 ms time slicing
- Q3 == top priority

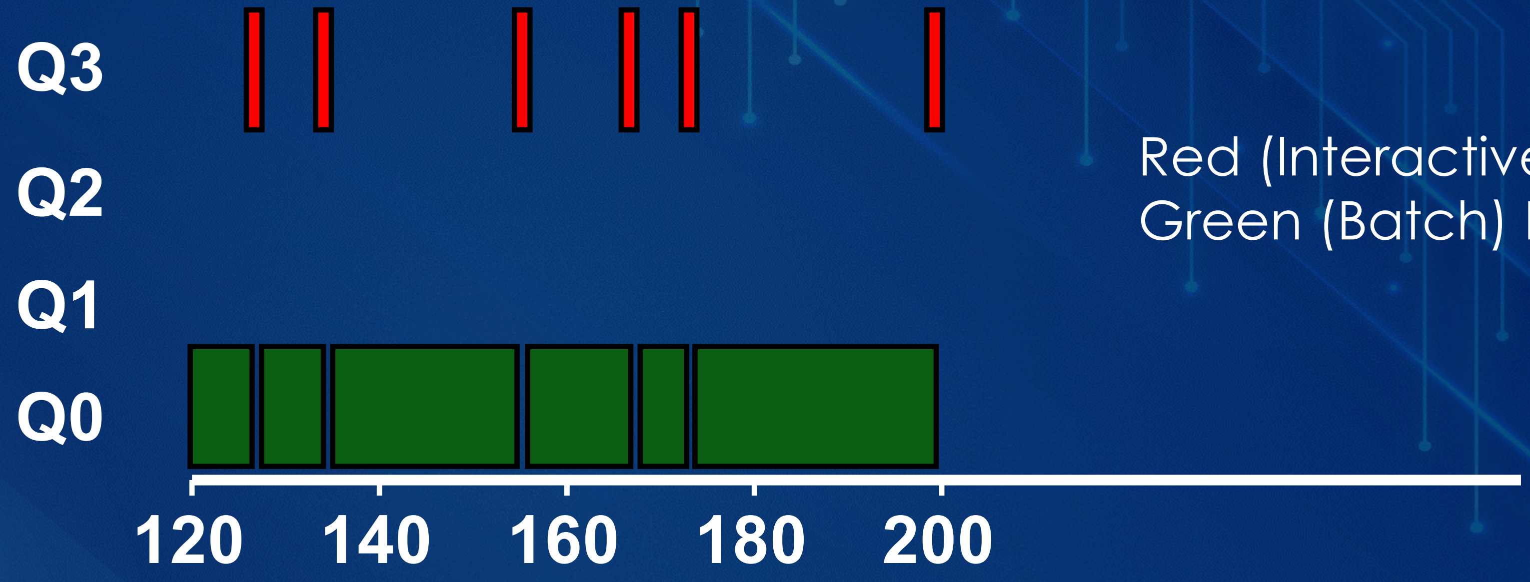



Red (Interactive) and Green (Batch) Process.

- Based on this diagram, what can you determine about the red / green processes?
  - Red is an interactive process (using I/O) - thus never uses its entire CPU time slice.
  - Interactive processes never use entire time slice, so are not demoted
  - Why isn't the red process allowed to use its entire time slice on the CPU?
    - Because it blocks on I/O and the Scheduler puts another process onto the CPU (while the red process waits for the I/O to finish.

# Problems with MLFQ?





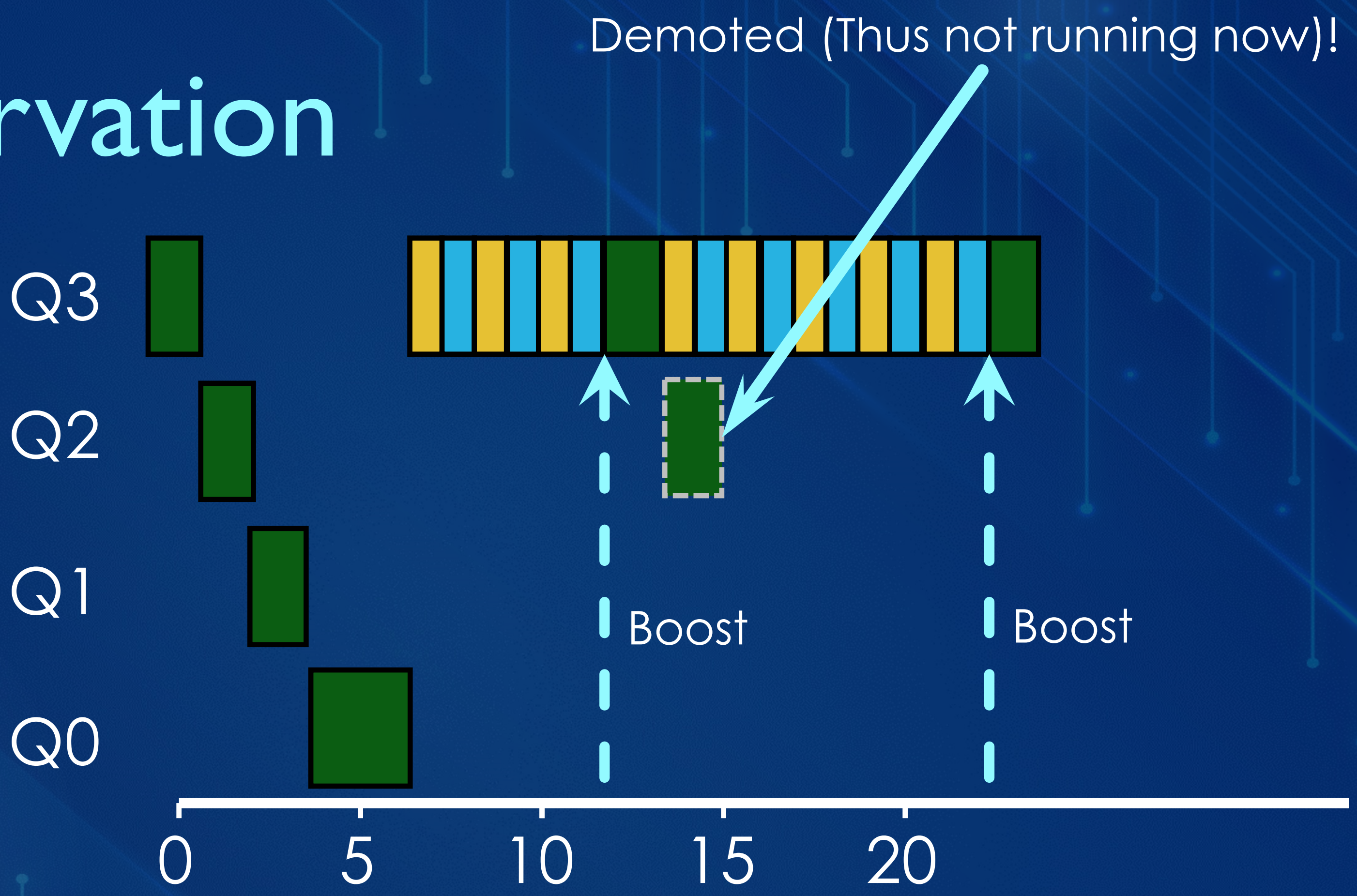

Red (Interactive) and Green (Batch) Process.

- Problems
  - Long running (ie: batch) jobs will never receive CPU time(**starvation**)
    - This is not apparent in the 2-process case, but imagine having 10s of "Red" processes…
  - Smart users could **game** the scheduler.
    - Just yield the CPU right before the time slice is about to end.

# Prevent Starvation

Interactive Processes:
Blue, Gold

CPU Intensive Process:
Green




Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs
(or all jobs that haven't been scheduled; aging)

# Prevent Gaming

- Job can yield just before time slice ends to retain CPU
- Fix: Account for job's total run time at priority level
  - Instead of just this time slice;
  - downgrade when threshold exceeded

# Updated MLFQ Rules



- Rule 1: If priority(A) > Priority(B), A runs
- Rule 2: If priority(A) == Priority(B), A & B run in RR
- Rule 3: Processes start at top priority (highest queue)
- Rule 4: Once a job uses allotment at given level, it moves down a queue (<u>prevents</u> gaming).
- Rule 5: After some time period, move (boost) all jobs to the topmost queue and repeat (<u>prevents</u> starvation).

# Lottery Scheduling



- Goal: proportional (fair) share, but allow for weights
- Approach:
  - Give processes lottery tickets
  - Whoever wins runs
  - More tickets → higher priority/share
- Amazingly simple to implement

# Lottery Scheduler Example



```
int counter = 0;
int winner = getrandom( 0, totaltickets );

node_t *current = head;
while( current ) { // Walk list to find winner
    counter += current->tickets;
    if (counter > winner){ break; }
    current = current->next;
}

// current gets to run
```

head → Job A (1) → Job B (1) → Job C (100) → Job D (200) → Job E (100) → null

# Scheduling Summary



- Understand goals (metrics) and workload, then design scheduler around that
- General purpose schedulers need to support processes with different goals
- Past behavior is good predictor of future behavior
- Random algorithms (lottery scheduling) can be simple to implement and can avoid corner cases.

# Interprocess Communication (IPC)

*adapted from slides by Scott Brandt at UC Santa Cruz and other general sources

# IPC Introduction



- Inter-Process Communication (IPC) enables processes to communicate with each other, to share information
  - **Pipes (half duplex)**
  - **FIFOs (named pipes)**
  - Stream pipes (full duplex)
  - Named stream pipes
  - Message queues
  - Semaphores
  - Shared Memory
  - Sockets (Can be on the same machine)
  - Signals

# Pipes



- What is a (Linux/Shell) Pipe?
  - A communication channel (between processes*)
  - A connection between two file descriptors**
- **File Descriptors
  - An integer that refers to the location in the array of open files.
  - stdin == 1, stdout == 2… What is #3?
    - stderr
  - Everything in Unix (Linux) is a file…
    - Keyboard, Disk, Monitor, etc
- dup2, pipe
  - Unix commands to setup and manipulate file descriptors
  - `read()` / `write()` / `cin` / `cout`

# Pipes



- Oldest (and perhaps simplest) form of UNIX IPC
- Half duplex
  - Data flows in only one direction
- Only usable between processes with a common ancestor
  - Usually parent-child
  - Also child-child

# Pipes on the command line



- Chain output of one command into the input of another.
- Syntax: `command1 | command2`
  - `ls | head`
  - `w | grep days`
- In Unix, all processes in a pipe are started "simultaneously".
- Pipes implement buffering under the hood if the read/write speeds of the two processes are different.

# Pipes (cont.)



- `#include <unistd.h>`
- `int pipe( int fildes[2] );`
  - How is the param `fildes` passed?
    - Reference… What?
    - It is actually `int * filedes`
  - `fildes[0]` is open for **reading** (remember – half duplex)
  - `fildes[1]` is open for **writing**
- The output of `fildes[1]` is the input for `fildes[0]`

# Understanding Pipes



- Within a single process
  - Data written to `fildes[1]` can be read on `fildes[0]`
  - Not very useful
- Typically used ***between*** processes.

# Pipes and fork()



- Create a pipe.
- Then fork().
- Parent and Child processes each have a copy of the pipe… however, it can only be used to send data in one direction.
  - Parent and Child must decide which one will write to the pipe, and which one will read from it.
  - Pipes are half-duplex.

# Pipes and fork()



- To send messages from parent to child:
  - Have parent close the read end of its pipe:
    - `close( filedes[0] );`
  - Have child close the write end of its pipe:
    - `close( filedes[1] );`
  - Use write() in the parent to write to the pipe. Write to `filedes[1]`.
  - Use read() in the child to read from the pipe. Read from `filedes[0]`.

# Summary



- IPC allows processes to communicate.
- One mechanism is via pipes.
- Pipes can be used programmatically via pipe() and fork() to allow IPC between parent and child.
- Pipes can be used on the command line to chain commands together without using intermediate files.

# ~ Fin ~