CPU Virtualization: Scheduling

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- Steps
  - Load specified code and data into memory; Create empty call stack
  - Create and initialize PCB (make it look like context-switch)
  - Put process on ready list
- Advantages: No wasted work (compared to option 2)
- Disadvantages: Difficult to express all possible options for setup, complex
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Process Creation

Option 2: Clone an existing process and change it

- Example: Unix fork() and exec()
  - Fork(): Clones the calling process
  - Exec(char *file): Overlays file image on calling process
- Fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process? Yes!
- Exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

Unix Process Creation

Fork/exec crucial to how the user's shell is implemented!

```
While (1) {
  Char *cmd = getcmd();
  Int retval = fork();
  If (retval == 0) {
       // This is the child process
       // Setup the child's process environment here
       // E.g., where is standard I/O, how to handle signals?
       exec(cmd);
       // exec does not return if it succeeds
       printf("ERROR: Could not execute %s\n", cmd);
       exit(1);
  } else {
       // This is the parent process; Wait for child to finish
       int pid = retval;
       wait(pid);
  }
}
```

Scheduling

Questions answered in this lecture:

What are different scheduling policies, such as: FCFS, SJF, STCF, RR and MLFQ?

What type of workload performs well with each scheduler?

What scheduler does Linux currently use?

https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Chapters 7-10

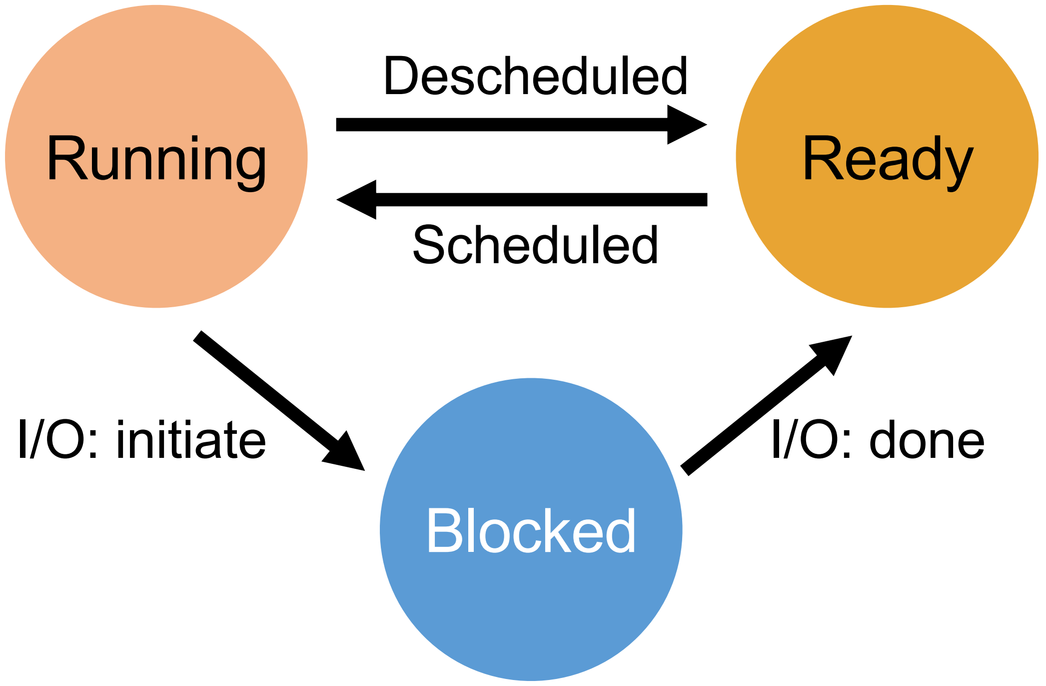
CPU Virtualization: Two Components

Dispatcher (Previous lecture)

- Low-level mechanism
- Performs context-switch
  - Switch from user mode to kernel mode
  - Save execution state (registers) of old process in k-stack, PCB
  - Insert PCB in ready queue
  - Load state of next process from k-stack, PCB to registers
  - Switch from kernel to user mode
  - Jump to instruction in new user process

- Scheduler (Today)
  - Policy to determine which process gets CPU when

Review: Process State Transitions



How to transition? (“mechanism”)
When to transition? (“policy”)

Vocabulary

Workload: set of **job** descriptions (arrival time, run_time)

- Job: View as current CPU burst of a process
- Process alternates between CPU and I/O
  process moves between ready and blocked queues

Scheduler: logic that decides which ready job to run

Metric: measurement of quality of schedule

Scheduling Performance Metrics

Minimize turnaround time

- Do not want to wait long for job to complete
- **Completion_time – arrival_time**

Minimize response time

- Schedule interactive jobs promptly so users see output quickly
- **Initial_schedule_time – arrival_time**

Maximize throughput

- Want many jobs to complete per unit of time

Maximize resource utilization

- Keep expensive devices busy

Minimize overhead

- Reduce number of context switches

Maximize fairness

- All jobs get same amount of CPU over some time interval

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. Run-time of each job is known

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Example: workload, scheduler, metric

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

FIFO: First In, First Out
- also called FCFS (first come first served)
- run jobs in *arrival_time* order

What is our turnaround? *completion_time - arrival_time*

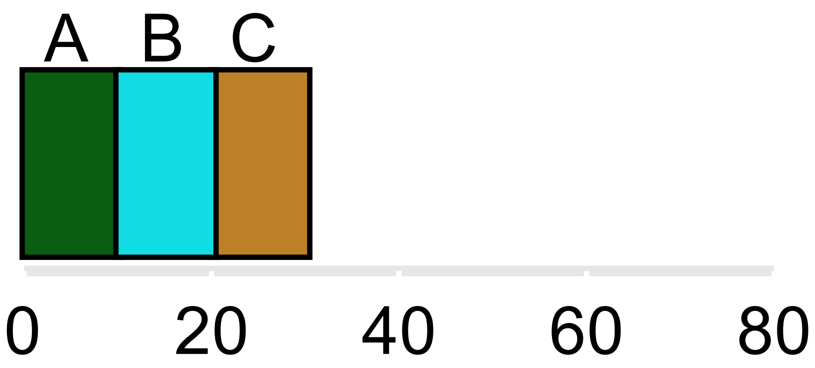
FIFO: Event Trace

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

Time	Event
0	A arrives
0	B arrives
0	C arrives
0	run A
10	complete A
10	run B
20	complete B
20	run C
30	complete C

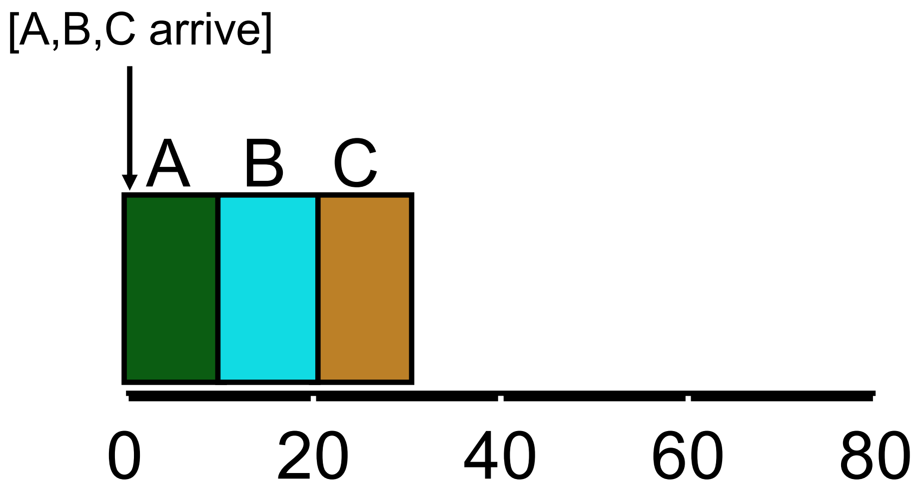
FIFO (Identical JOBS)

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10



Gantt chart:
Illustrates how jobs are scheduled over time on a CPU

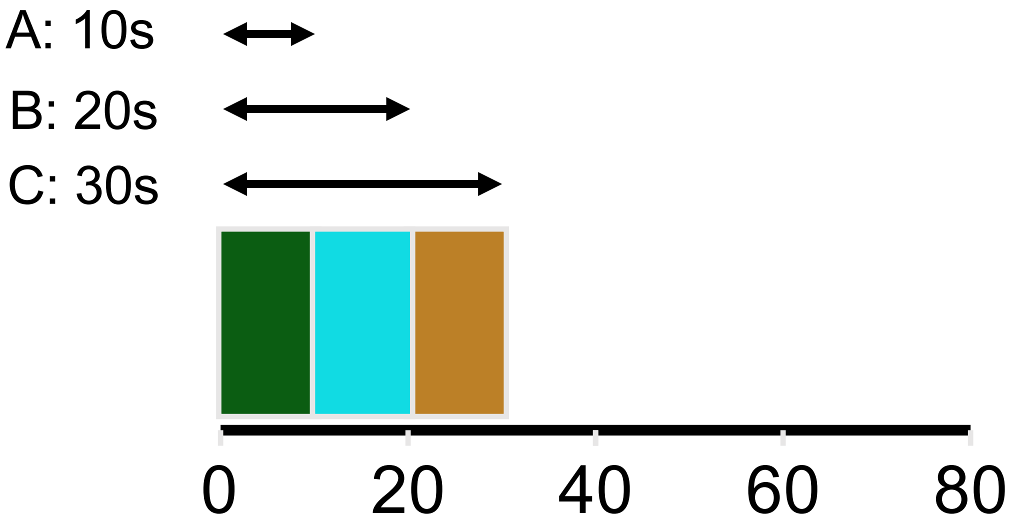
FIFO (IDENTICAL JOBS)



What is the average turnaround time?

Def: *turnaround_time* = *completion_time* - *arrival_time*

FIFO (IDENTICAL Jobs)



What is the average turnaround time?

Def: *turnaround_time* = *completion_time* - *arrival_time*

(10 + 20 + 30) / 3 = **20s**

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

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

Any Problematic Workloads for FIFO?

Workload: ?

Scheduler: FIFO

Metric: turnaround is high

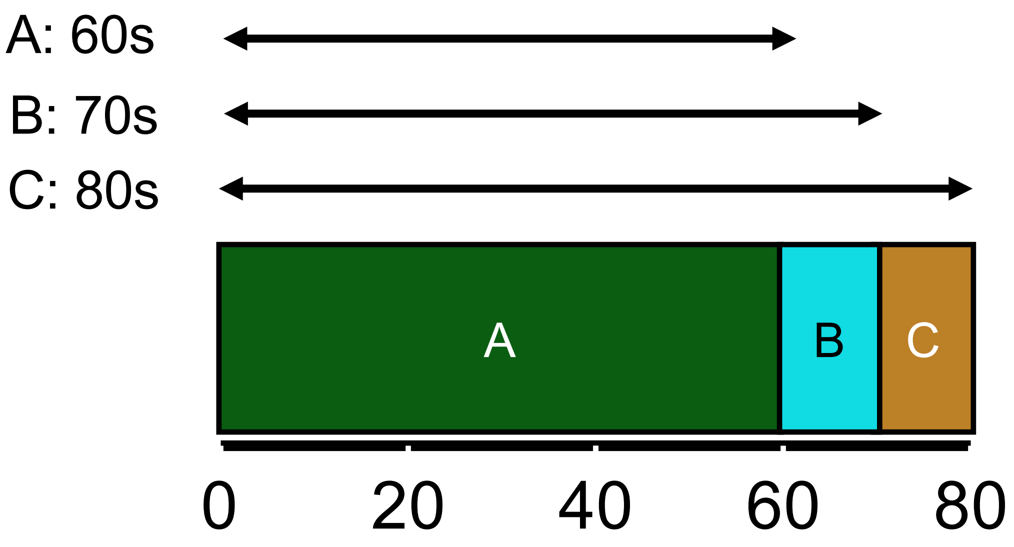
Example: Big First Job

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

Draw Gantt chart for this workload and policy…
What is the average turnaround time?

Example: Big First Job

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10



Average turnaround time: **70s**

Convoy Effect

Passing the Tractor

Problem with Previous Scheduler:

FIFO: Turnaround time can suffer when short jobs must wait for long jobs

New scheduler:

SJF (Shortest Job First)

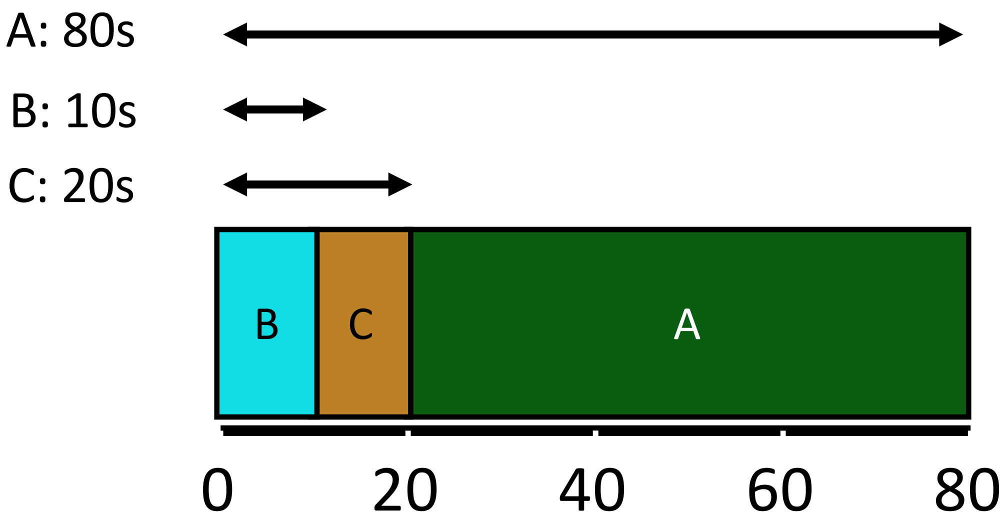
Choose job with smallest *run_time*

Shortest Job First

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

What is the average turnaround time with SJF?

SJF Turnaround Time



What is the average turnaround time with SJF?

(80 + 10 + 20) / 3 = **~36.7s**

Average turnaround with FIFO: 70s

For minimizing average turnaround time (with no preemption):
SJF is provably optimal
Moving shorter job before longer job improves turnaround time of short job more than it harms turnaround time of long job

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

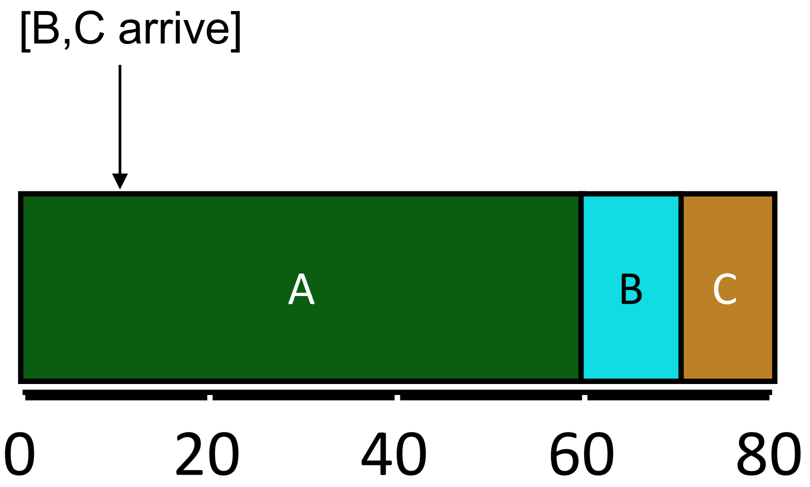
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. The run-time of each job is known

Shortest Job First (Arrival Time)

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time with SJF?

Stuck Behind a Tractor Again



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time?

(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

Preemptive Scheduling

Prev schedulers:

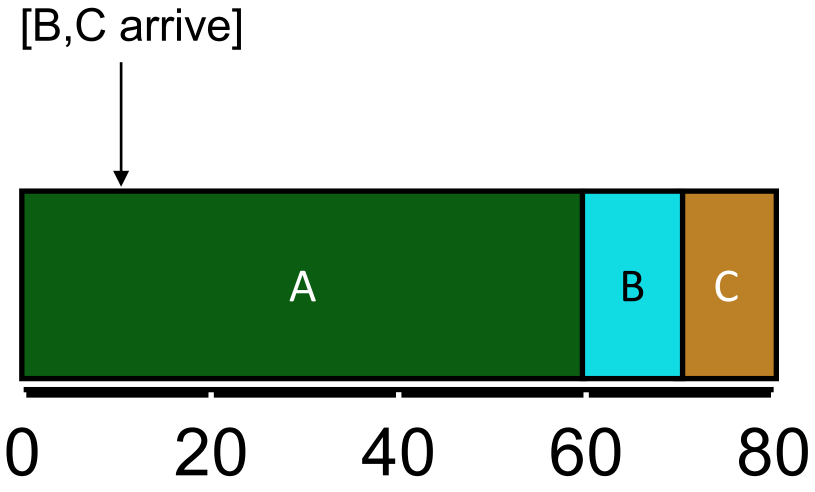
- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)

New scheduler:

- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
- Always run job that will complete the quickest
  - (That job may change over time)

NON-PREEMPTIVE: SJF

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

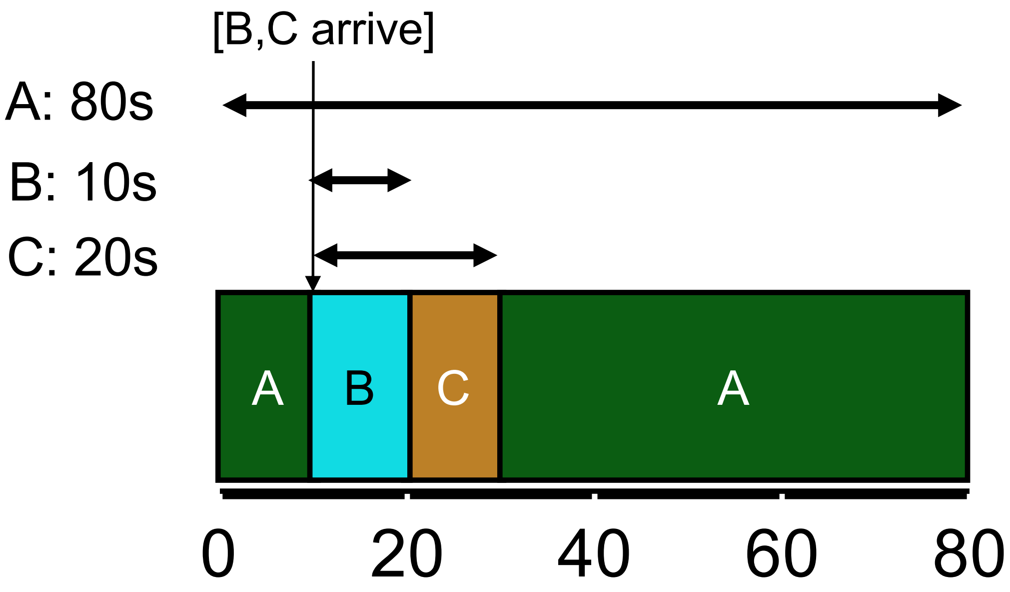


Average turnaround time:

(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

PREEMPTIVE: STCF

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10



Average turnaround time with STCF?

36.6

Average turnaround time with SJF: **63.3s**

Scheduling Basics

Workloads:
arrival_time
run_time

Schedulers:
FIFO
SJF
STCF
RR

Metrics:
turnaround_time
response_time

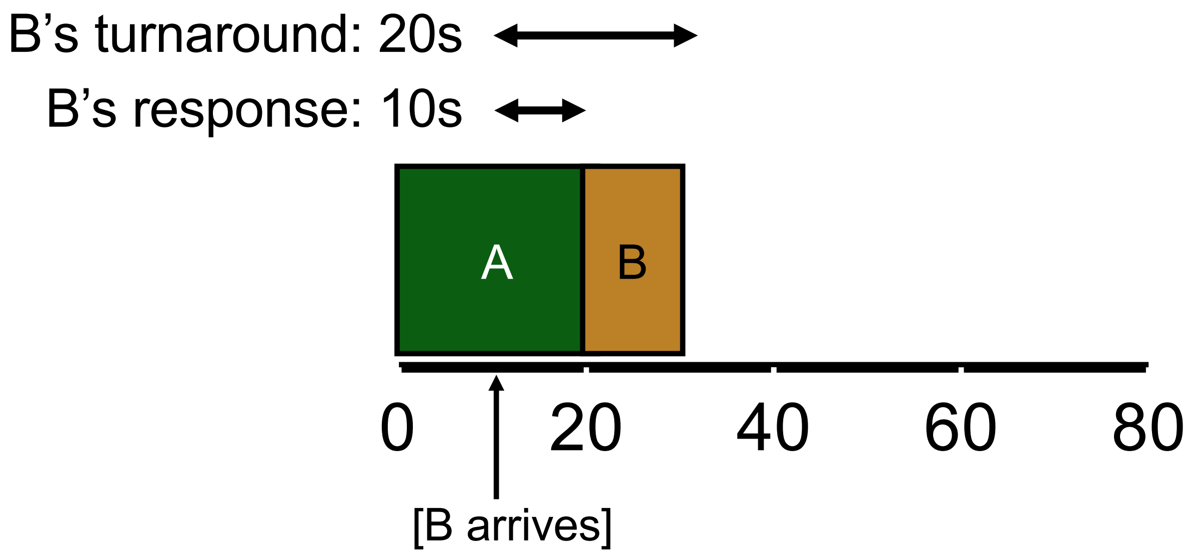
Response Time

Sometimes we care about when job starts instead of when it finishes

New metric:

response_time = *first_run_time* - *arrival_time*

Response vs. Turnaround



Round-Robin Scheduler

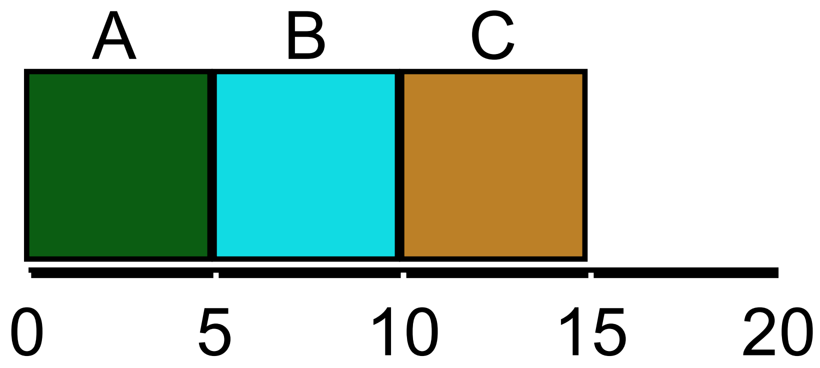
Prev schedulers:

FIFO, SJF, and STCF can have poor response time

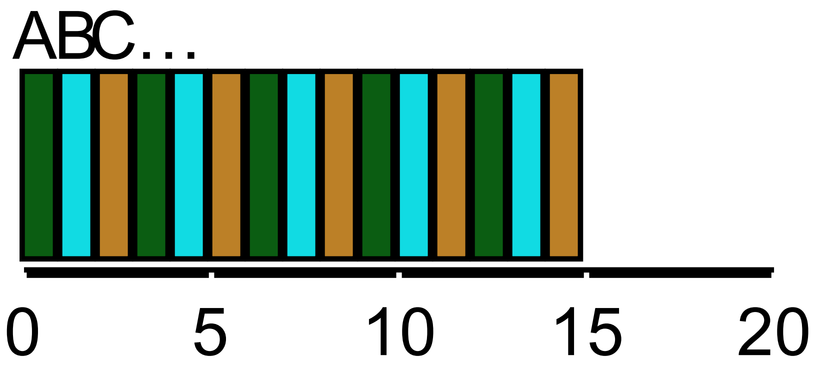
New scheduler: RR (Round Robin)

Alternate ready processes every fixed-length time-slice

FIFO vs RR



Avg Response Time?
(0+5+10)/3 = **5**



Avg Response Time?
(0+1+2)/3 = **1**

In what way is RR worse?
Ave. turn-around time with equal job lengths is horrible

Other reasons why RR could be better?
If don't know run-time of each job, gives short jobs a chance to run and finish fast

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Review- 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~~

   (need smarter, fancier scheduler)

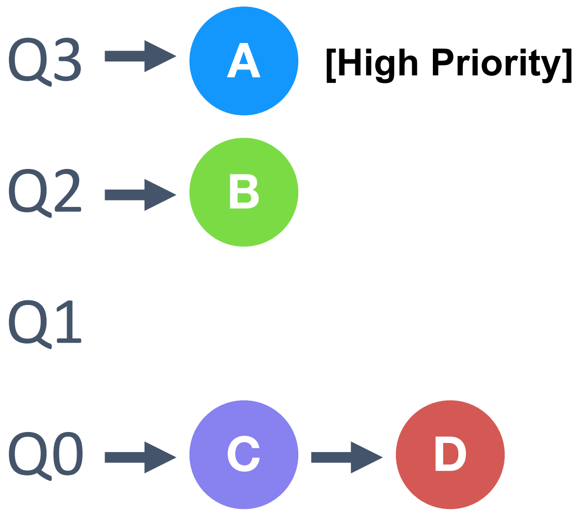
MLFQ
(Multi-Level Feedback Queue)

- 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 levels of round-robin
  - each level has higher priority than lower levels and preempts them
- MLFQ has a number of distinct queues.
- Each queue is assigned a different priority level.

Priorities

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

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



"Multi-level"

How to know how to set priority?

Approach 1: nice
Approach 2: history "feedback"

History

- Use past behavior of process to predict future behavior
  - Common technique in systems
- 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

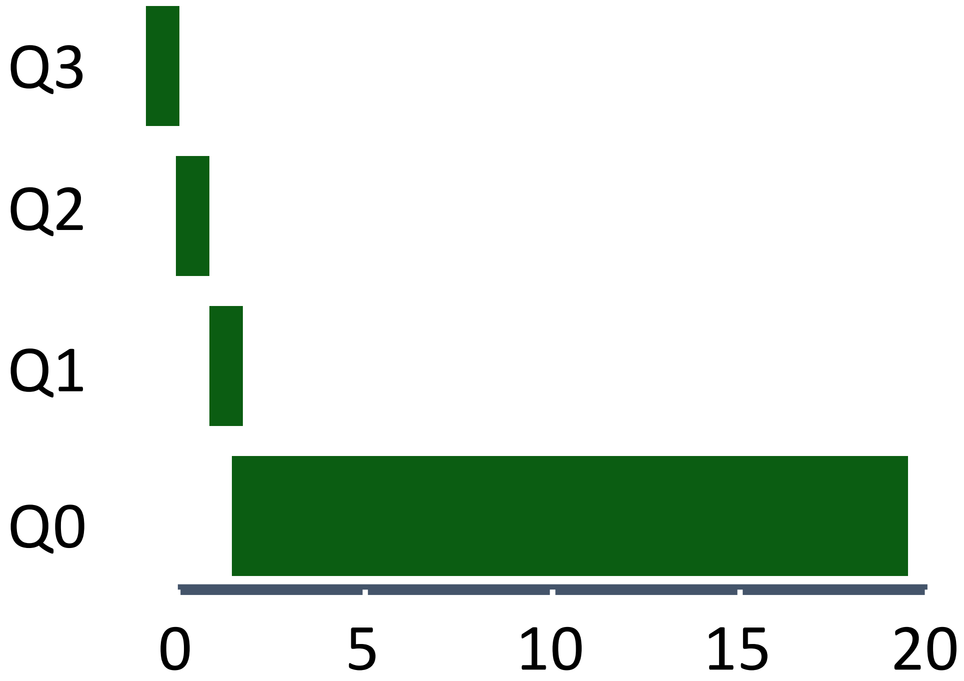
Rule 1: If priority(A) > Priority(B), A runs

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

More rules:

- Rule 3: Processes start at top priority
- Rule 4: If job uses whole slice, demote process (longer time slices at lower priorities)

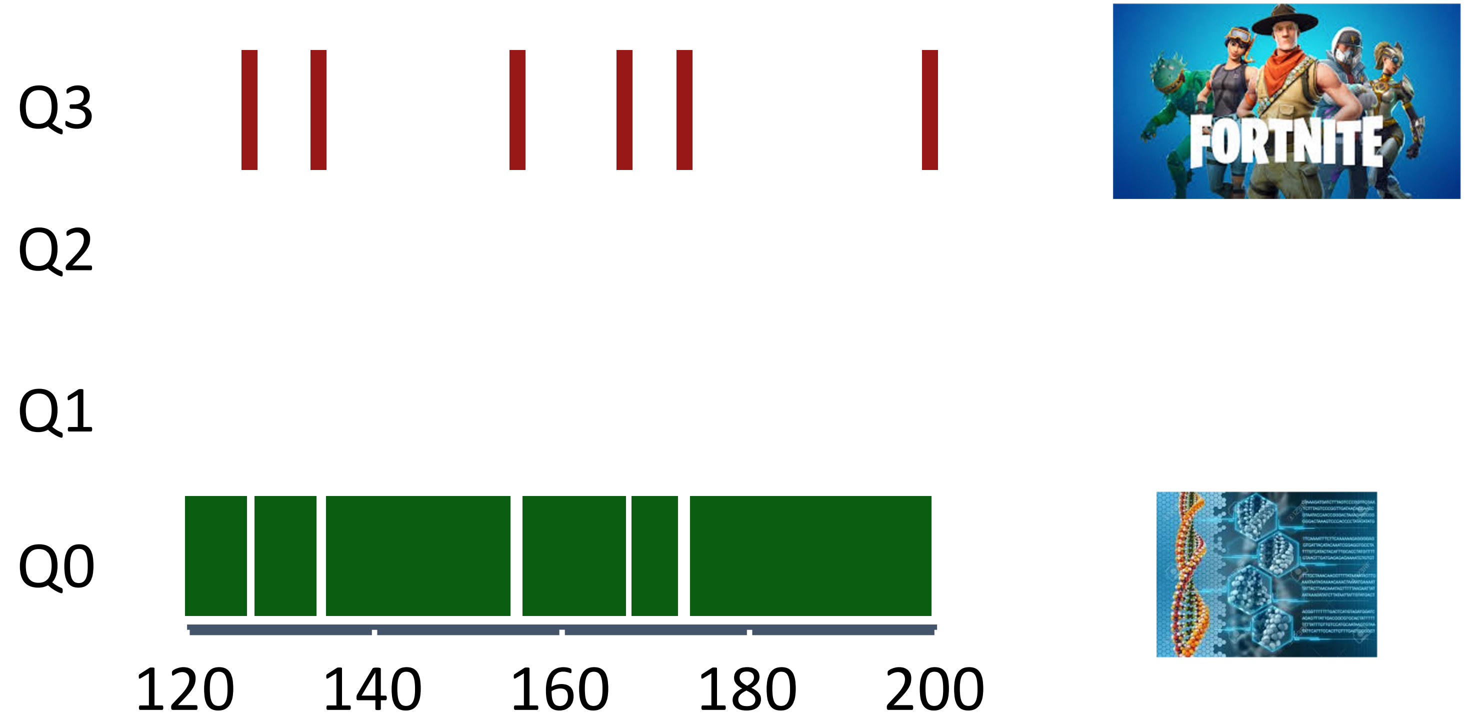
One Long Job (Example)



A four-queue scheduler with time slice 10ms

Long batch job – DNA analysis

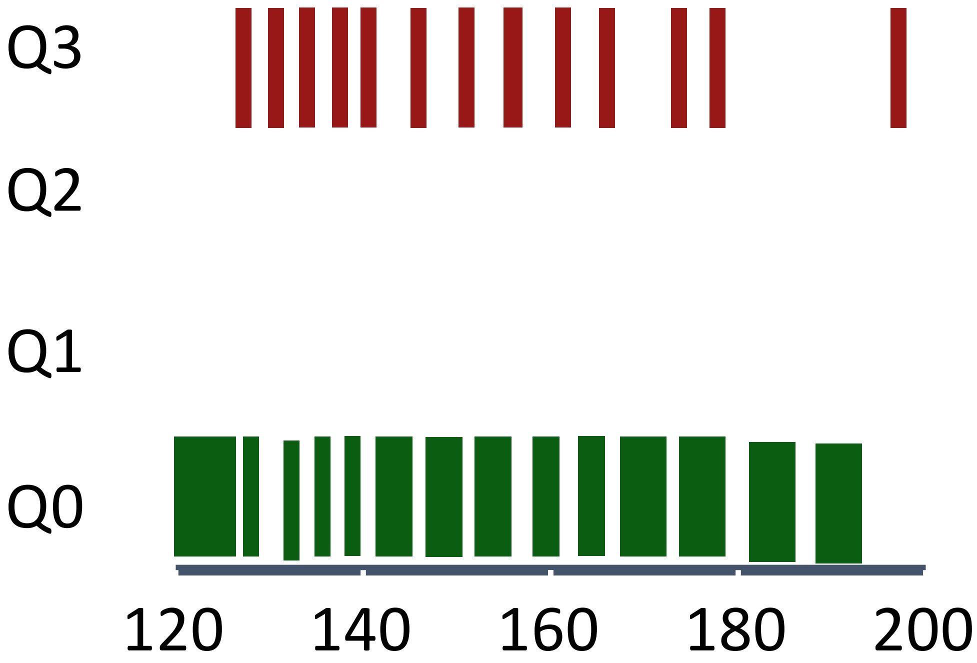
An Interactive Process Joins



Interactive job performs quick operation and does an I/O

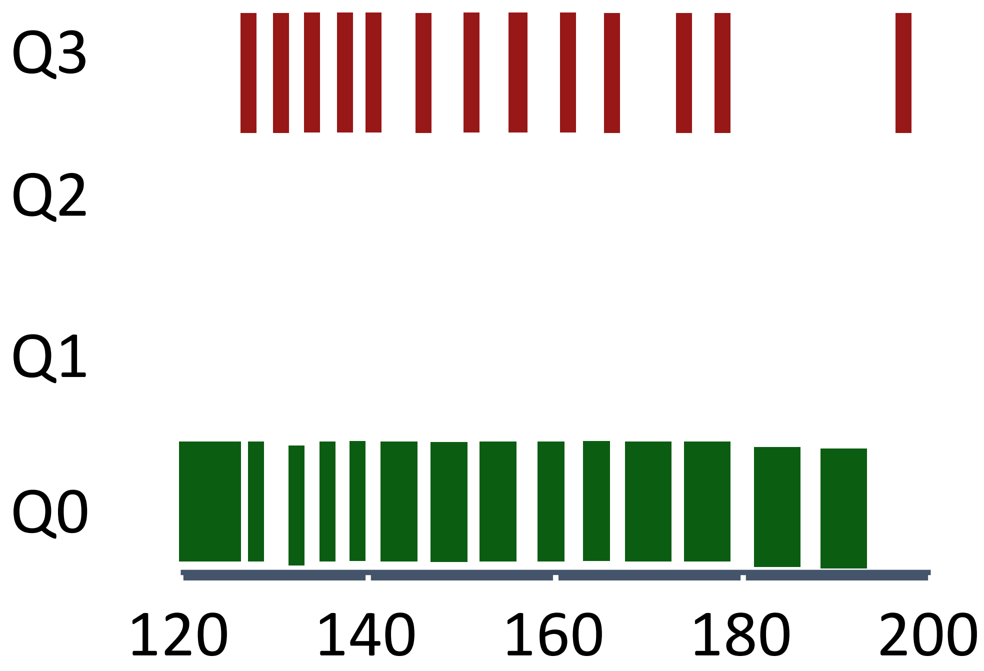
Interactive process never uses entire time slice, so never demoted

Problems with MLFQ?



Problems
- unforgiving + starvation
- gaming the system

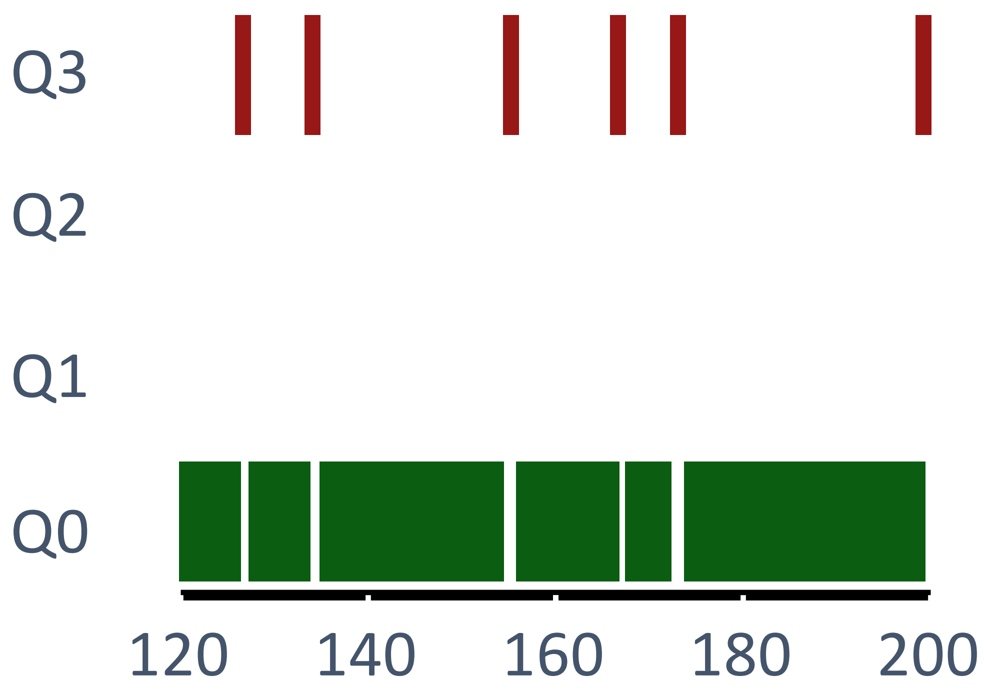
Problems with MLFQ?



Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs (or all jobs that haven’t been scheduled)

Prevent Gaming the Schedule



Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

Fix: Account for job's total run time at priority level (instead of just this time slice); downgrade when exceed threshold

Lottery Scheduling

Goal: proportional (fair) share
Sometimes we just care about fairly sharing the CPU.

Fair-share scheduler

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

Approach:

- give processes lottery tickets
- whoever wins runs
- higher priority => more tickets

Amazingly simple to implement

Lottery Scheduling

- Tickets
  - Represent the share of a resource that a process should receive
  - Percent of tickets represents its share of the system resource in question.

- Example
  - 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

Lottery Scheduling

- The scheduler picks a winning ticket.
  - Load the state of that *winning process* and runs it.
- 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

Intuition:
The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Lottery Code

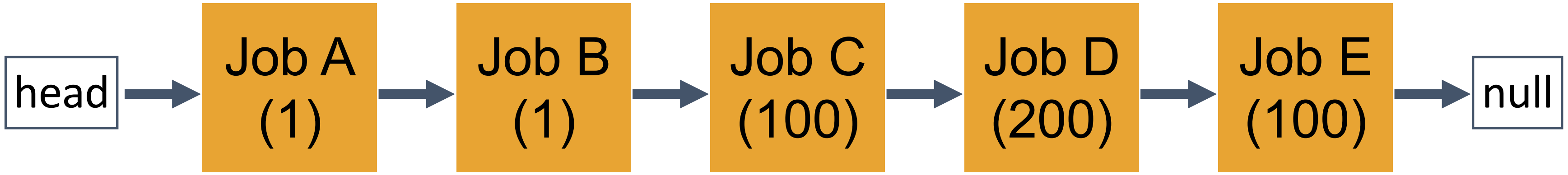
```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
        counter += current->tickets;
        if (counter > winner) break;
        current = current->next;
}
// current is the winner
```

Lottery example

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while(current) {
            counter += current->tickets;
            if (counter > winner) break;
            current = current->next;
}
// current gets to run
```

Who runs if winner is:

- 50
- 350
- 0



Other Lottery Ideas

Ticket Transfers

Ticket Currencies

Ticket Inflation

(read more in OSTEP)

Can make lottery scheduling deterministically fair, too