Virtualization: CPU Scheduling

Kai Mast
CS 537
Fall 2022

RECAP: SCHEDULING MECHANISMS

Limited Direct Execution

- Special bit to designate user vs kernel mode
- Use system calls to access devices
- Use timer interrupts for OS to gain control to perform context switch

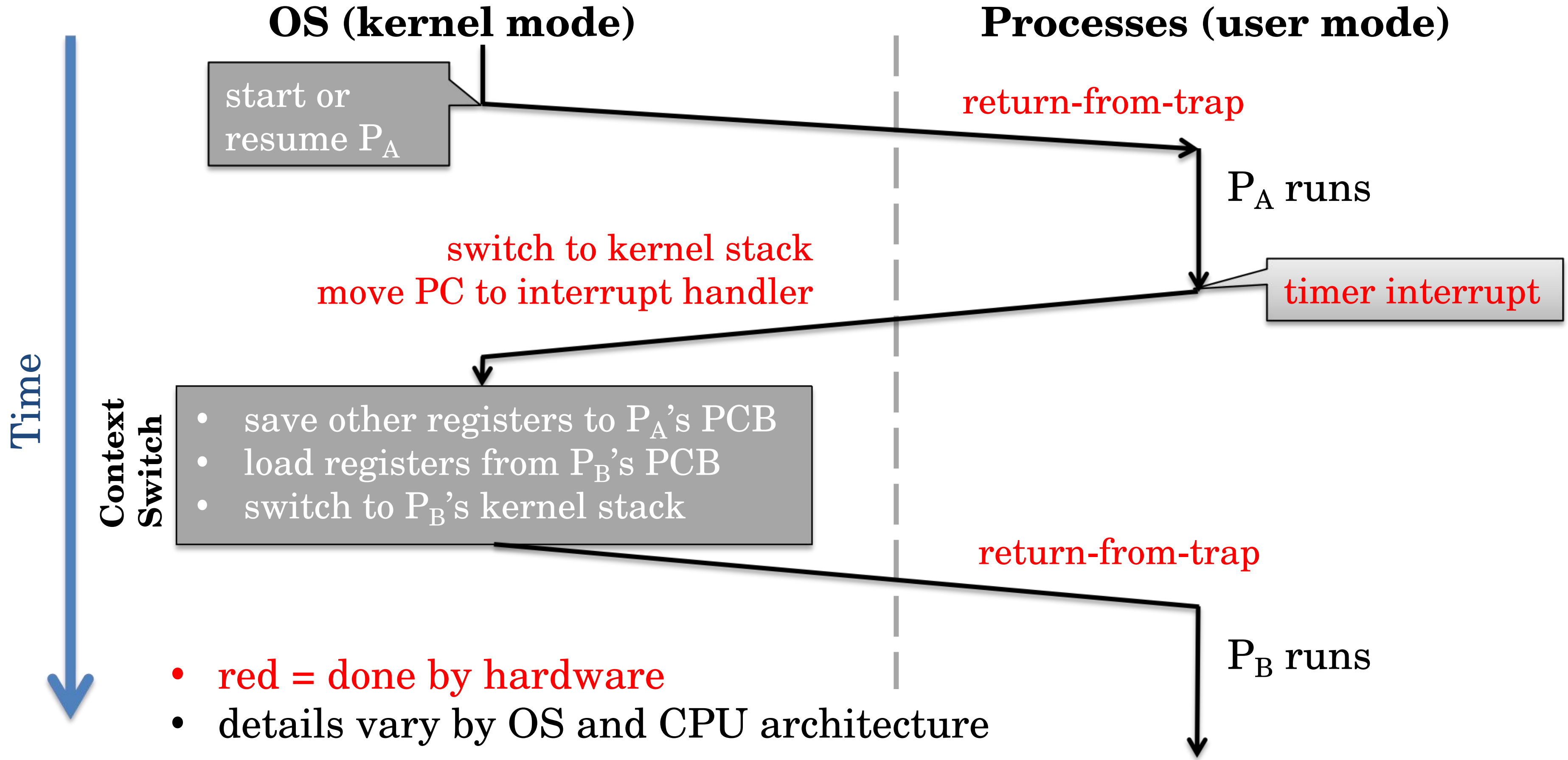
Process Control Block (PCB) or process table:

- Info saved in OS about every process, including register context when process not running

Kernel stack:

- Per process (fixed size); used as stack while executing in kernel mode;
- Save registers here from user process on system call

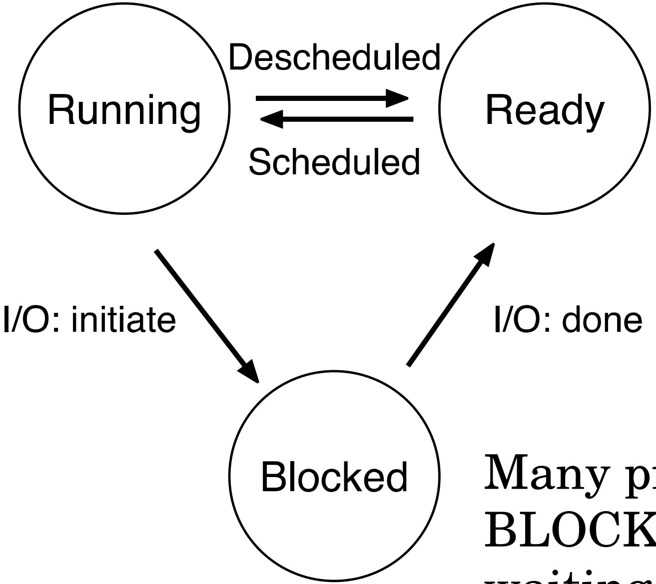
RECAP: CONTEXT SWITCHING



- red = done by hardware
- details vary by OS and CPU architecture

RECAP: PROCESS STATE TRANSITIONS

At most N processes are RUNNING (where N is the number of CPUs)



Many processes could be READY

Many processes could be BLOCKED,
waiting for I/O to complete

SCHEDULING TERMINOLOGY

Workload: set of **jobs** (arrival time, run_time)

Job: Current CPU burst of a process

- Process alternates between computation (CPU) and I/O
  - When process is waiting for I/O, blocked, cannot be scheduled (no job)
  - Process moves between ready and blocked queues

Scheduler: Decides which READY job to run

Metric: Measurement of scheduling quality

(POSSIBLE) PERFORMANCE METRICS

Minimize **turnaround time** (completion_time – arrival_time)

- Want job to be completed as soon as possible

Minimize **response time** (initial_schedule_time – arrival_time)

- Can't control how long job needs to run; minimize time before scheduled

Maximize **throughput** (jobs completed / second)

- Want many jobs to complete per unit of time

Maximize **resource utilization** (% time CPU busy)

- Keep expensive devices busy

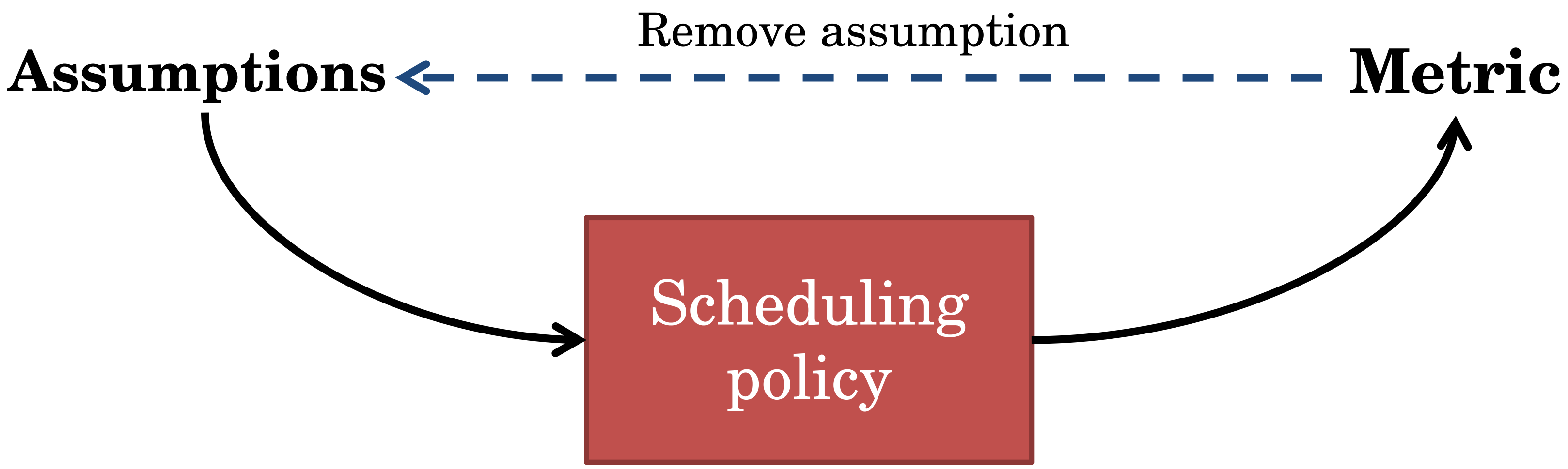
Minimize **overhead** (# of context switches and cache misses)

- Reduce number of context switches

Maximize **fairness** (variation of CPU time across jobs)

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

LECTURE FORMAT



We will gradually remove assumptions...

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
   (Oracle, perfect knowledge)

Metric 1: Turnaround Time

turnaround_time = *completion_time - arrival_time*

Example:

Process A arrives at time t = 10, finishes t = 30

Process B arrives at time t = 10, finishes t = 50

Turnaround time

A = 20

B = 40

Average = 30

FIFO / FCFS

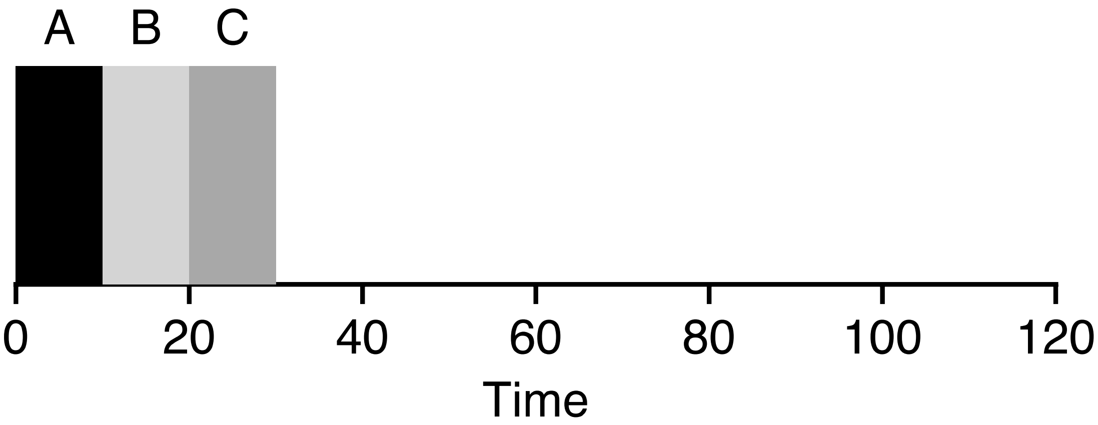
FIFO: First In, First Out

FCFS: First Come, First Served

Run jobs in *arrival_time* order (ties go to first job in list)

FIFO / FCFS

Job	Arrival Time (s)	Run Time (s)
A	~0	10
B	~0	10
C	~0	10



Gantt chart: Illustrate how jobs are scheduled over time

Average Turnaround Time ?

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

2-MINUTE NEIGHBOR CHAT

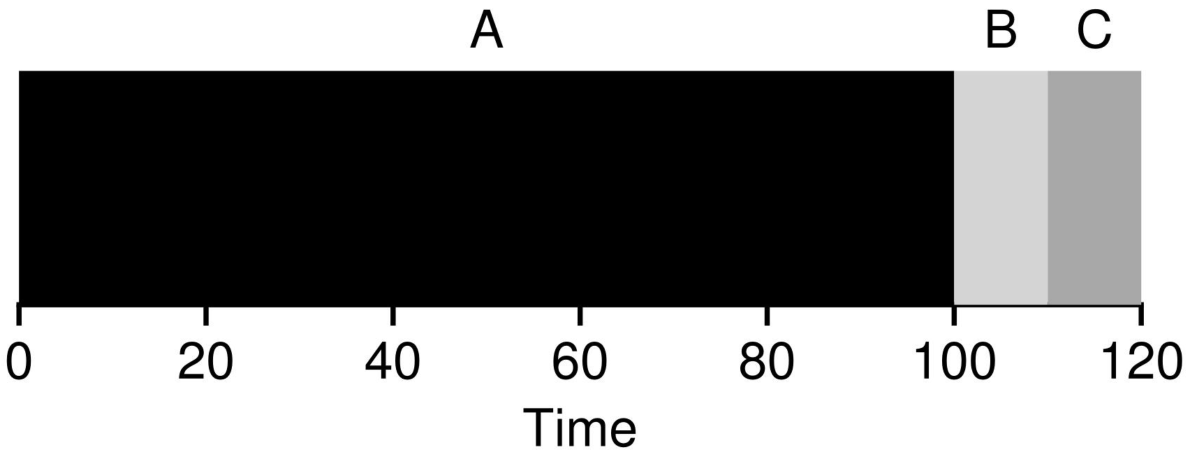
1. ~~Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

How will FIFO perform without this assumption ?

What scenarios can lead to bad performance?

LONG-RUNNING FIRST JOB

Job	Arrival Time (s)	Run Time (s)
A	~0	100
B	~0	10
C	~0	10



Average Turnaround Time?

(100 + 110 + 120)/ 3
= 110s

SCHEDULING PROBLEM: CONVOY EFFECT

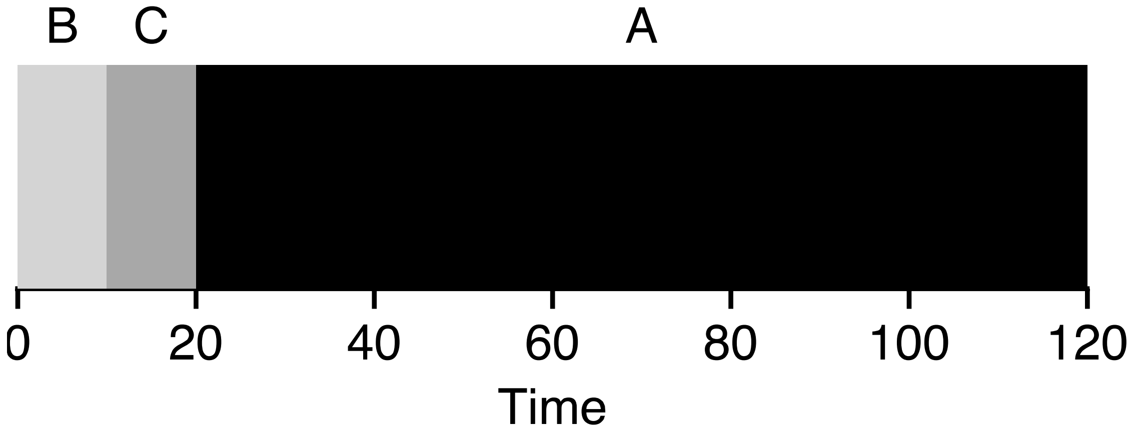
Turnaround time suffers when short jobs must wait for long jobs

Solution: A new scheduler

- SJF (Shortest Job First)
- Choose job with smallest run time!
- (Assume OS has perfect information…)

SHORTEST JOB FIRST (SJF)

Job	Arrival Time (s)	Run Time (s)
A	~0	100
B	~0	10
C	~0	10



Average Turnaround Time?

(10 + 20 + 120)/ 3

= 50s!

FIFO: 110s ?!

SHORTEST JOB FIRST (CONT.)

- SJF is provably optimal for minimizing average turnaround time (assuming no preemption)
- **Intuition:** Moving shorter job before longer job improves turnaround time of short job more than it harms the turnaround time of long jobs

ASSUMPTIONS

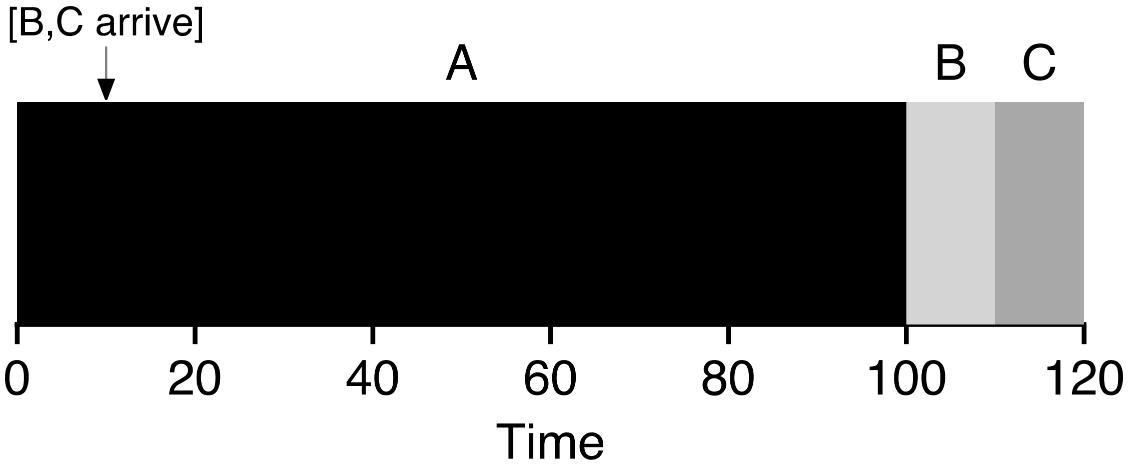
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

2-MINUTE NEIGHBOR CHAT

Job	Arrival Time (s)	Run Time (s)
A	0	100
B	~10	10
C	~10	10

Gantt Chart and Average Turnaround Time with SJF?

Job	Arrival Time (s)	Run Time (s)
A	0	100
B	~10	10
C	~10	10



Average Turnaround Time ?

(100 +
(110 -10) +
(120 – 10))/ 3
= 103.33s

PREEMPTIVE SCHEDULING

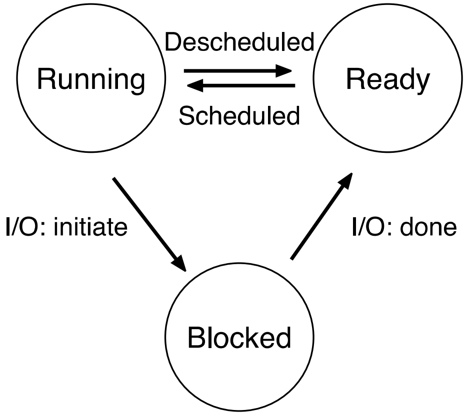
Previous schedulers:

- FIFO and SJF are non-preemptive (never deschedule a running process)
- Only schedule new job when previous job voluntarily relinquishes CPU (e.g., performs I/O or exits)

Preemptive: Schedule different job by taking CPU away from running job

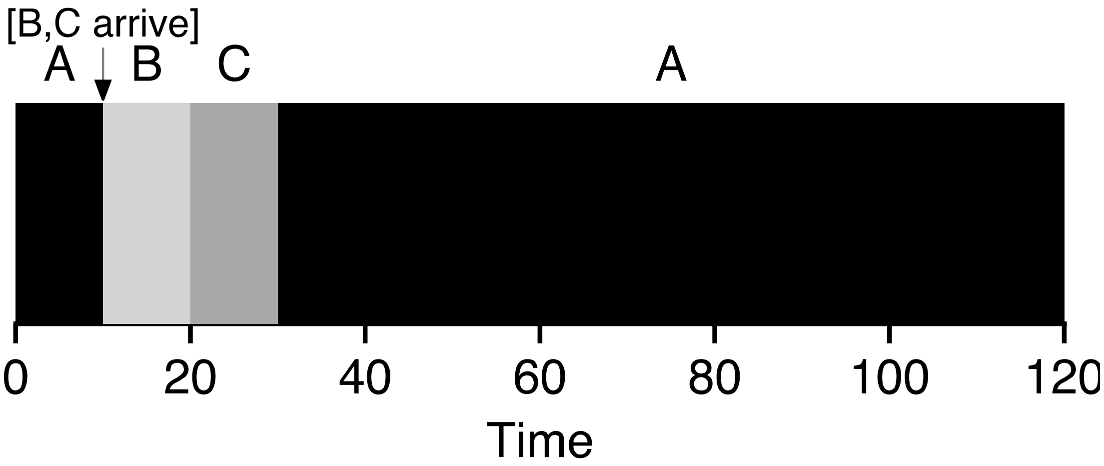
STCF (Shortest Time-to-Completion First)

- Always run job that will complete the quickest



PREEMPTIVE STCF (OR SCTF)

Job	Arrival Time (s)	Run Time (s)
A	0	100
B	~10	10
C	~10	10



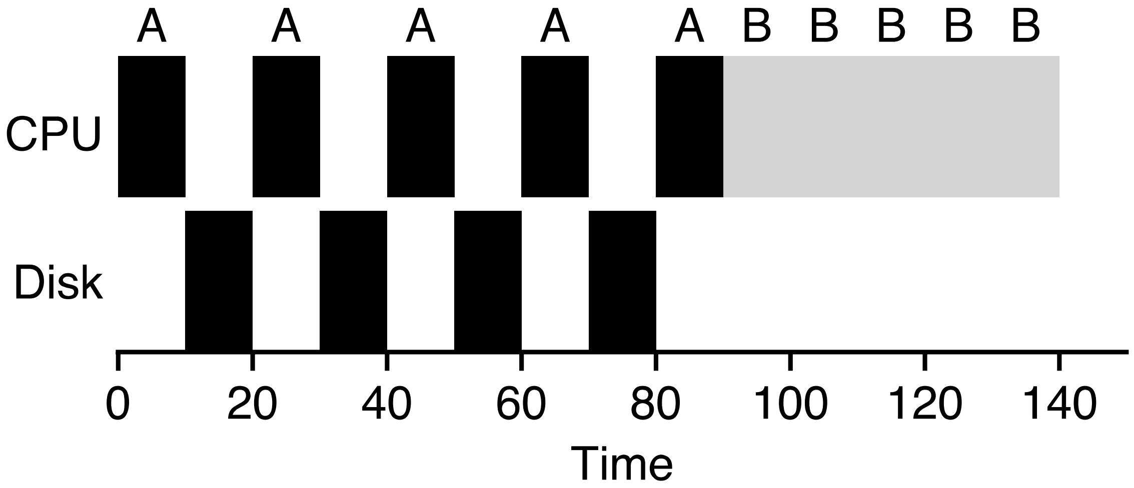
Average Turnaround Time

(10 + 20 + 120)/ 3
= 50s

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

NOT I/O AWARE

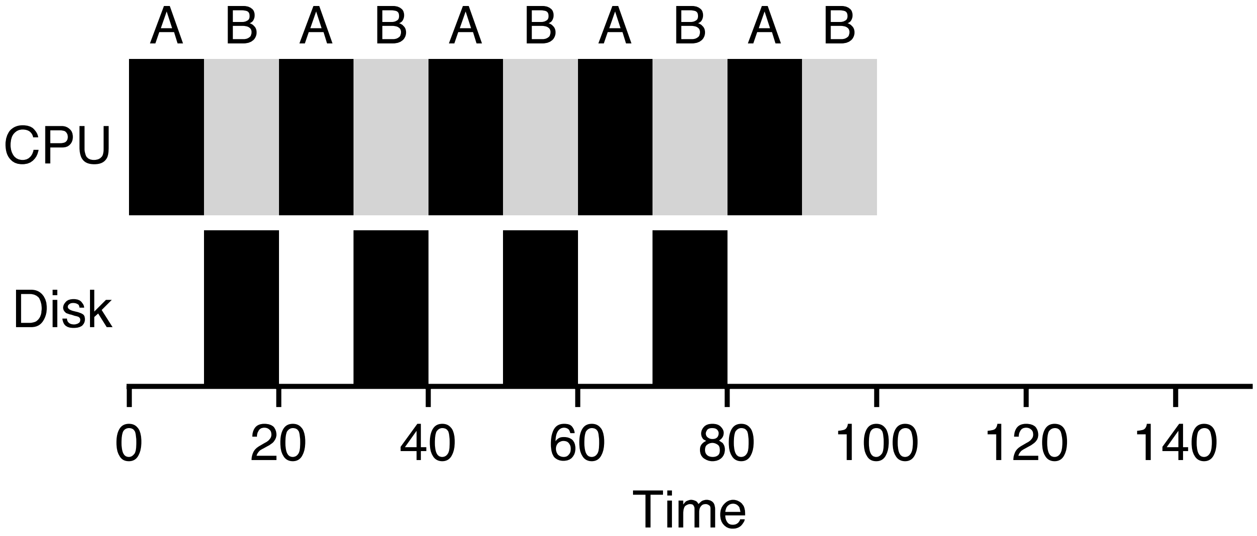


Job holds on to CPU while blocked on disk!

Instead, treat Job A as multiple separate CPU bursts

I/O AWARE SCHEDULING

B is a long CPU-bound job



When Job A completes I/O, another Job A is ready

Each CPU burst of A is shorter than Job B; With SCTF, Job A preempts Job B

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~~**

WHAT IF JOB RUNTIME IS UNKNOWN?

For metric of average turnaround:

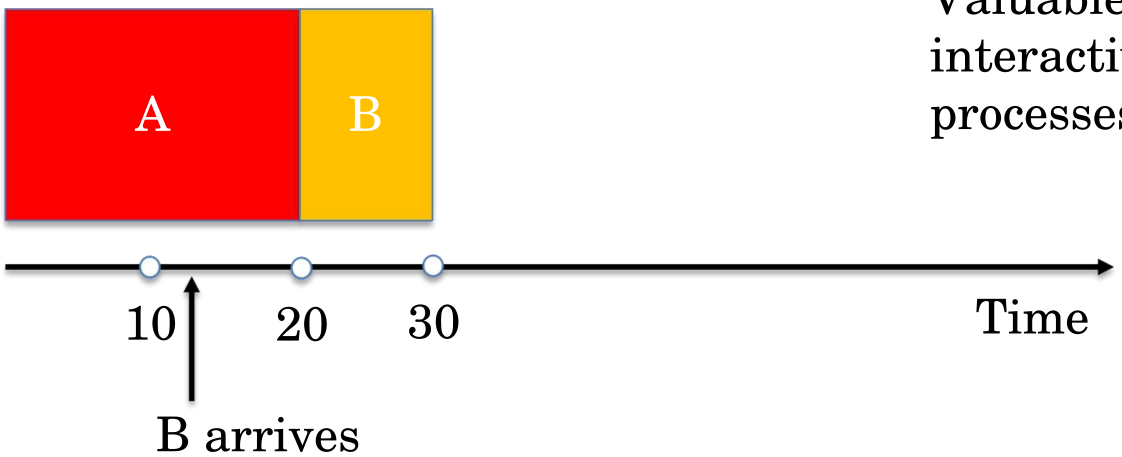
- If jobs have same length FIFO is fine
- If jobs have much different lengths SJF is much better

How can the OS get short jobs to complete first if OS does not know which ones are short?

METRIC 2: RESPONSE TIME

response_time = first_run_time - *arrival_time*

B's turnaround time = 20 seconds
B's response time = 10 seconds



Valuable for interactive processes

Round-Robin Scheduler

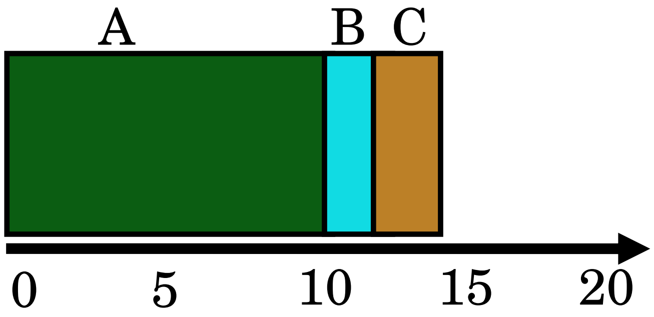
New scheduler: RR (Round Robin)

- Alternate ready processes for a fixed-length time-slice
- Preemptive

Short jobs will finish after fewer time-slices

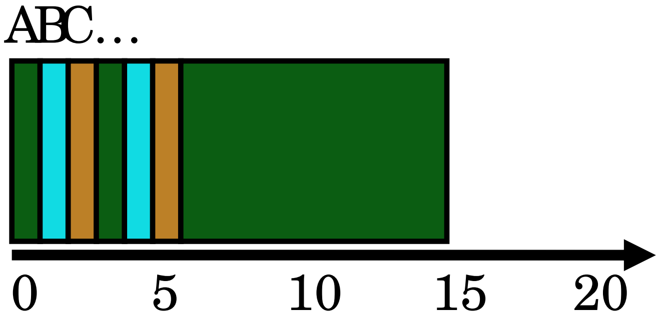
- Short jobs will finish sooner than long jobs

FIFO VS RR: JOBS DIFFERENT LENGTHS



A->10s B->12s C->14s

Avg. Turnaround Time?
(10+12+14)/3 = **12**

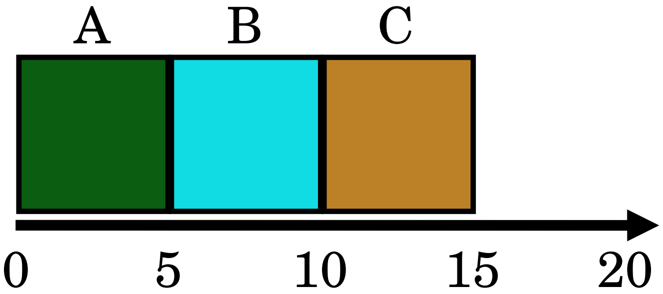


A->14s B->5s C->6s

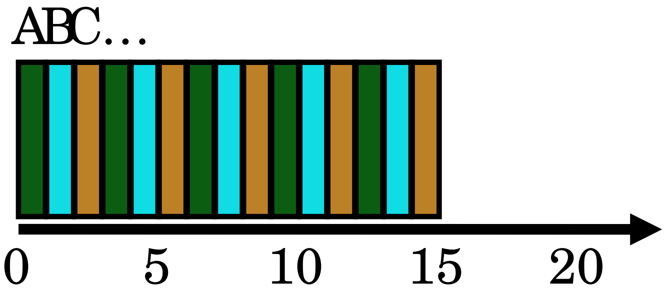
Avg. Turnaround Time?
(14+5+6)/3 = **8.3**

If know run-time of each job is unknown, RR gives short jobs a chance to run and finish fast

FIFO VS RR: JOBS SAME LENGTHS



Avg Turnaround Time?
(5+10+15)/3 = **10**



Avg Turnaround Time?
(13+14+15)/3 = **14**

When is RR worse than FIFO?
Average turn-around time with equal job lengths

TRADE-OFFS

Round robin:

- May increase turnaround time, decreases response time
- Potentially causes additional context switches

Tuning challenges:

- What is a good time slice for round robin?
- What is the overhead of context switching?

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~~

MULTI-LEVEL FEEDBACK QUEUE (MLFQ)

MLFQ: A GENERAL PURPOSE SCHEDULER

- Widely used in practice
- Its inventor (Fernando Corbató) won the Turing Award

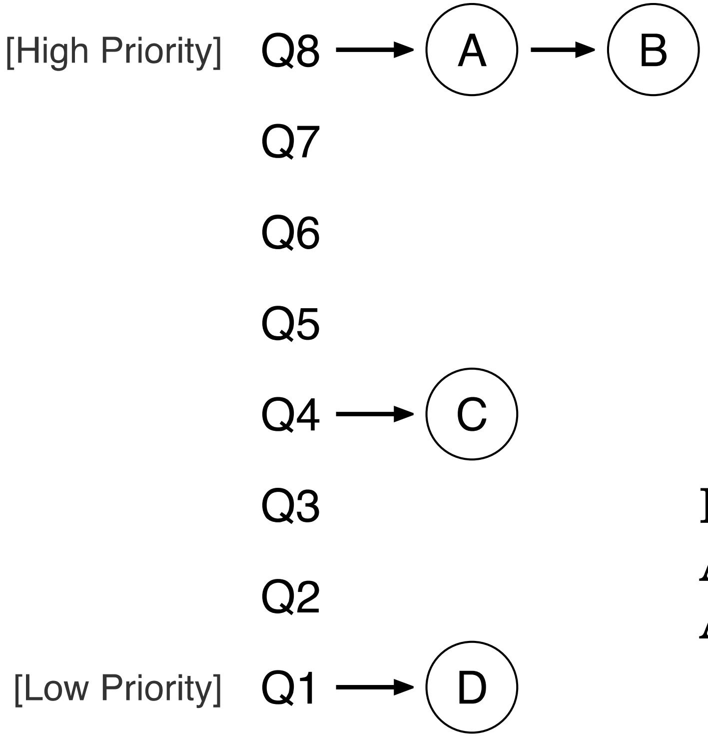
Must support two job types with distinct goals

- **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 level
- Can preempt them

MULTI-LEVEL PRIORITIES



"Multi-level" – Each level is a queue!

Rule 1: If priority(A) > Priority(C)
A runs

Rule 2: If priority(A) == Priority(B),
A & B run in Round-Robin

How to to set priority?
Approach 1: Static (no changes): nice command
Approach 2: Dynamic: Use history for feedback

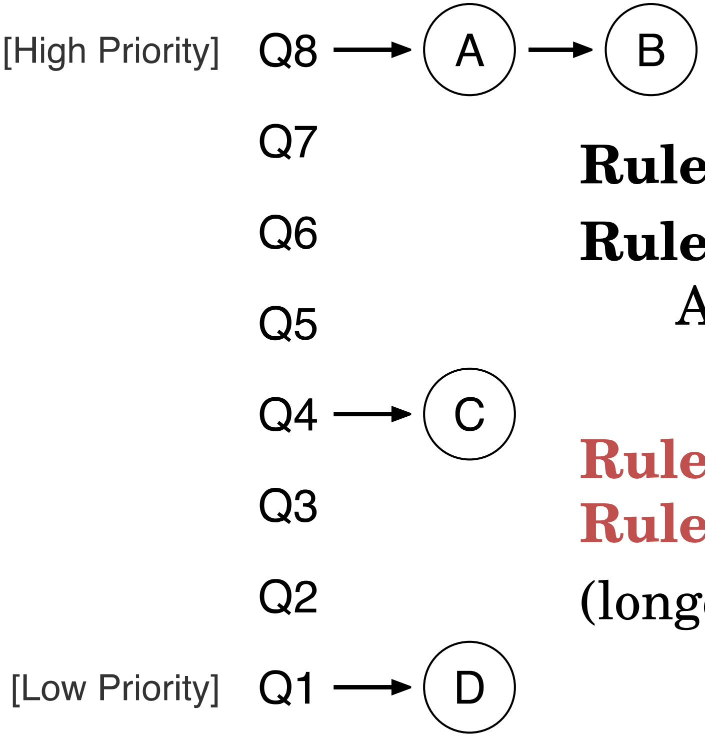
FEEDBACK: HISTORY

Approach:

- Use past behavior of process to predict future!
- Common approach in OS when don't have perfect knowledge

Guess how CPU burst (job) will behave based on past CPU bursts

MORE MLFQ RULES



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

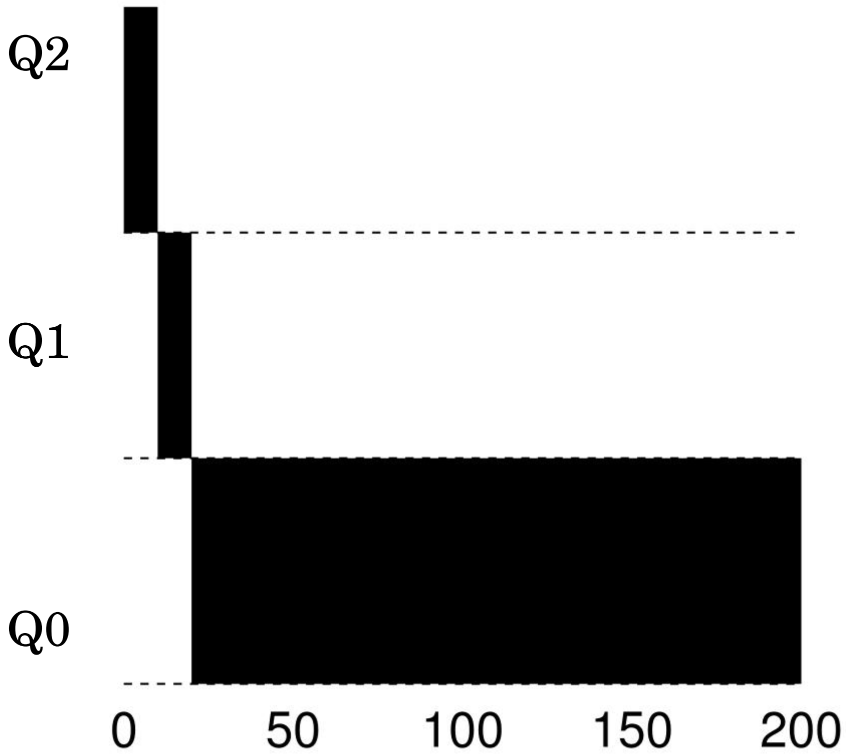
Rule 2: If priority(A) == Priority(B),
A & B run in Round-Robin

Rule 3: Processes start at top priority

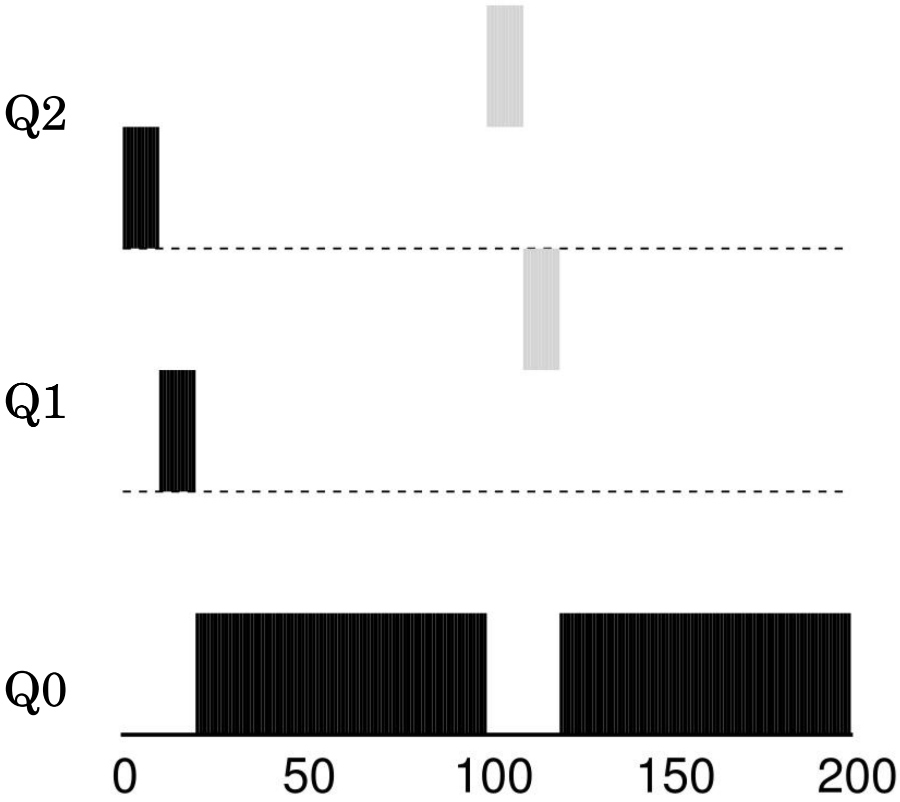
Rule 4: If job uses whole slice, demote process

(longer time slices at lower priorities)

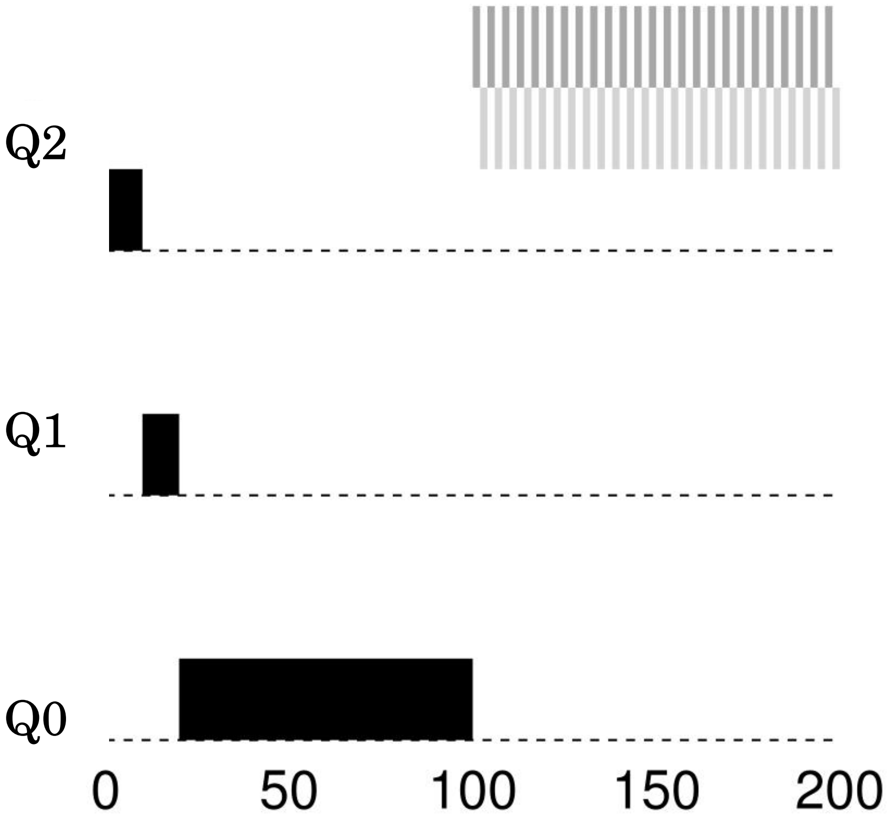
EXAMPLE: ONE LONG JOB



INTERACTIVE PROCESS JOINS



MLFQ PROBLEMS?

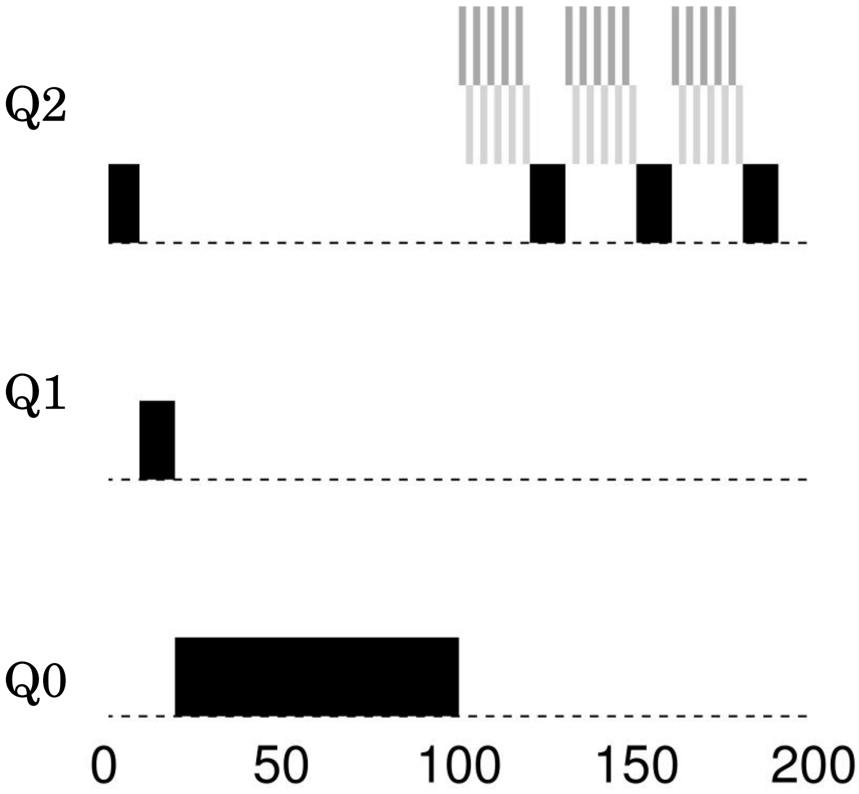


- Two (or more) short jobs arrive
- Each uses CPU for short burst
- Each stays at high priority

What is the problem with this schedule ?

Low priority batch job may never get scheduled

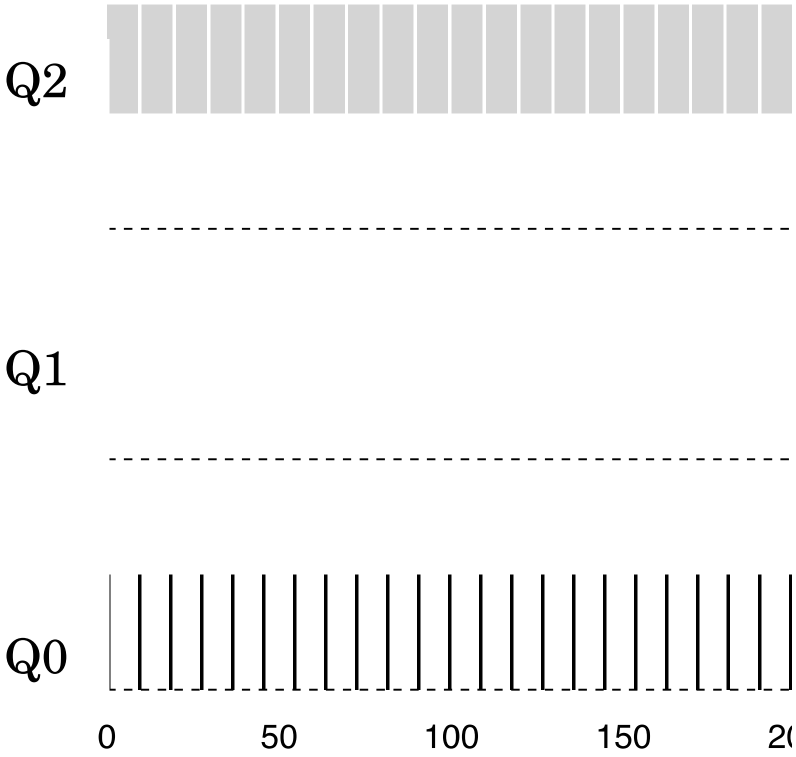
AVOIDING STARVATION



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

Open Question: How frequently do we boost jobs?

GAMING THE SCHEDULER?



Job could trick scheduler by not using entire time-slice (doing I/O just before time-slice end)

Possible solution

- Account for total run time at priority
- Downgrade when threshold is exceeded

OTHER SCHEDULERS: LOTTERY SCHEDULING

Other environments might require different types of schedules, e.g., purchasing fixed amount of cloud resources

Goal: proportional (fair) share

Approach:

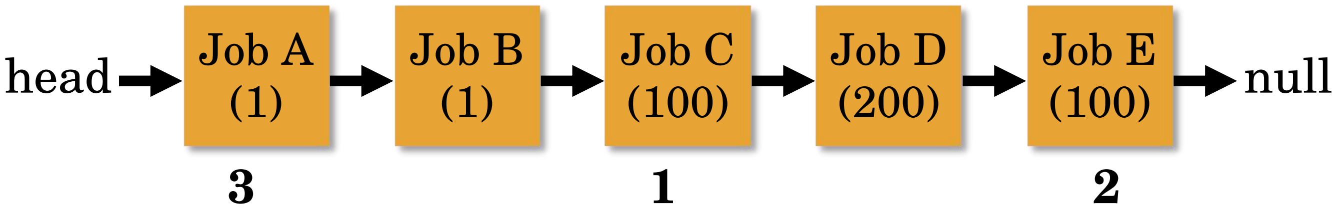
- give processes lottery tickets
- whoever wins runs
- higher priority => more tickets

Amazingly simple to implement

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;
}
if(current) run(current);
```

Who runs if winner is
Round 1: 50
Round 2: 350
Round 3: 0



SUMMARY

- No ideal scheduler exists for every workload and metric
  - Understand goals, the revelant metrics, and workload
  - Design scheduler (parameters) around that
- General purpose schedulers need to support processes with different goals
- Past behavior is a good predictor of future behavior?