

**Department of Physics,
Computer Science & Engineering**

CPSC 410 – Operating Systems I

Virtualization: CPU Scheduling

Keith Perkins

Adapted from “CS 537 Introduction to Operating Systems” Arpaci-
Dusseau

CPU Virtualization:

- 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?

CPU Virtualization: Two Components

Dispatcher (Previous lecture)

Low-level mechanism

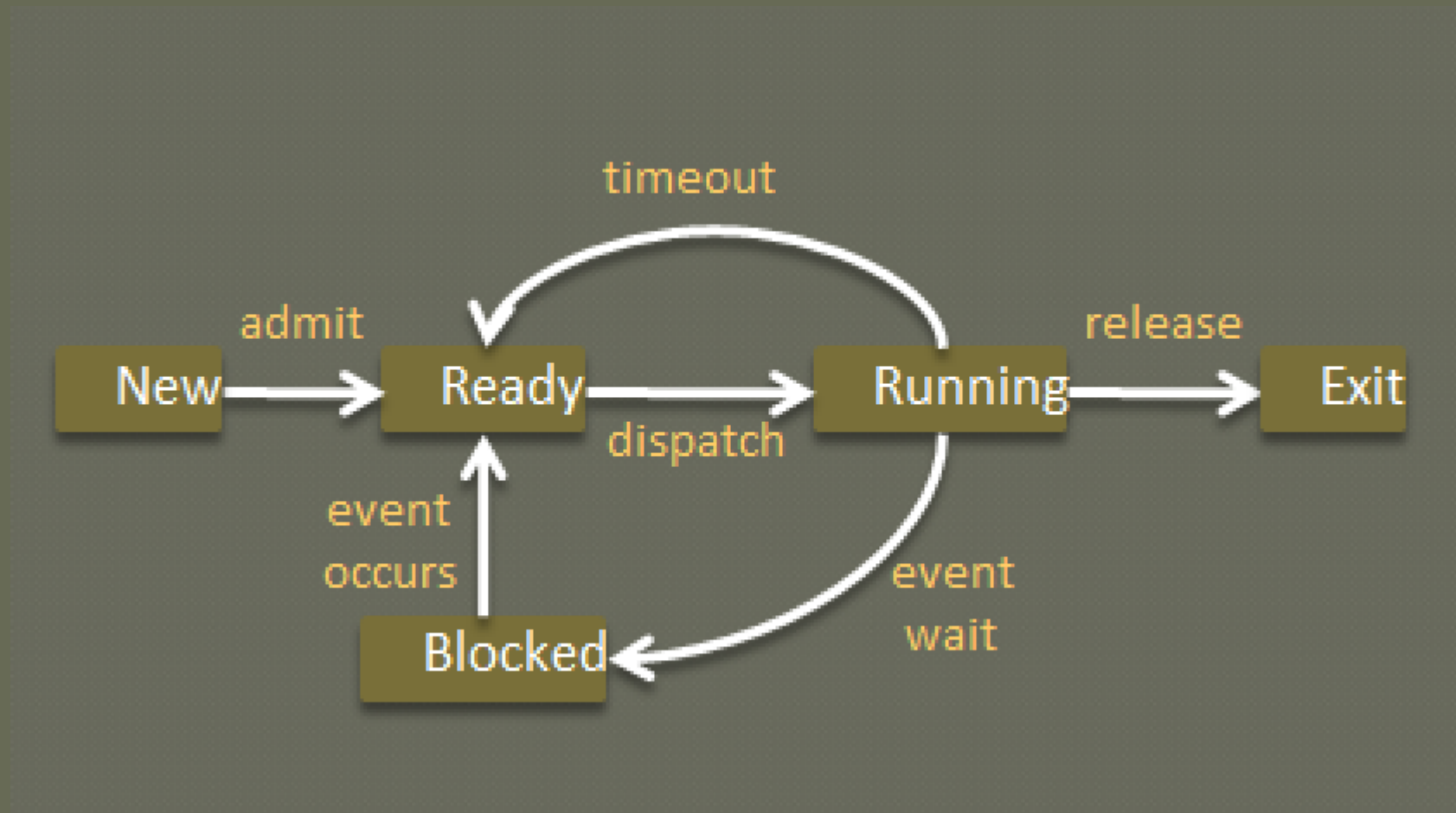
Performs context-switch

- ▢ Switch from user mode to kernel mode
- ▢ Save execution state (registers) of old process in Process Control Block (PCB)
- ▢ Insert process back in ready queue
- ▢ Load state of next process to run from 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: 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, so process moves between ready and blocked queues

Scheduler: logic that decides which ready job to run

Metric: measurement of scheduling quality

Scheduling Performance Metrics

Minimize turnaround time

Do not want to wait long for job to complete

$\text{Completion_time} - \text{arrival_time}$

Minimize response time

Schedule interactive jobs promptly so users see output quickly

$\text{Initial_schedule_time} - \text{arrival_time}$

Minimize waiting time

Do not want to spend much time in Ready queue

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?: $completion_time - arrival_time$

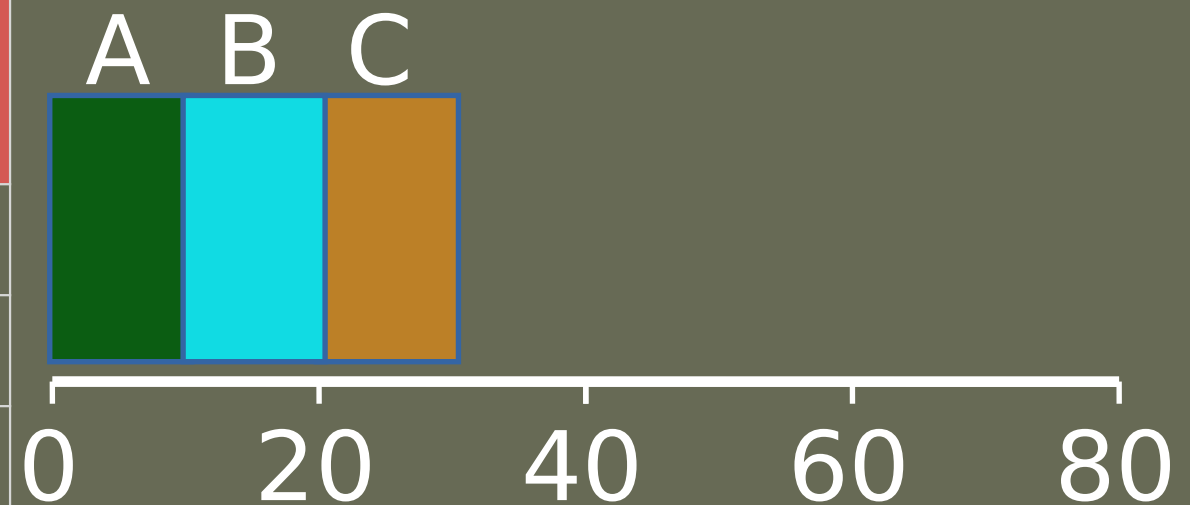
FIFO: Event Trace

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)

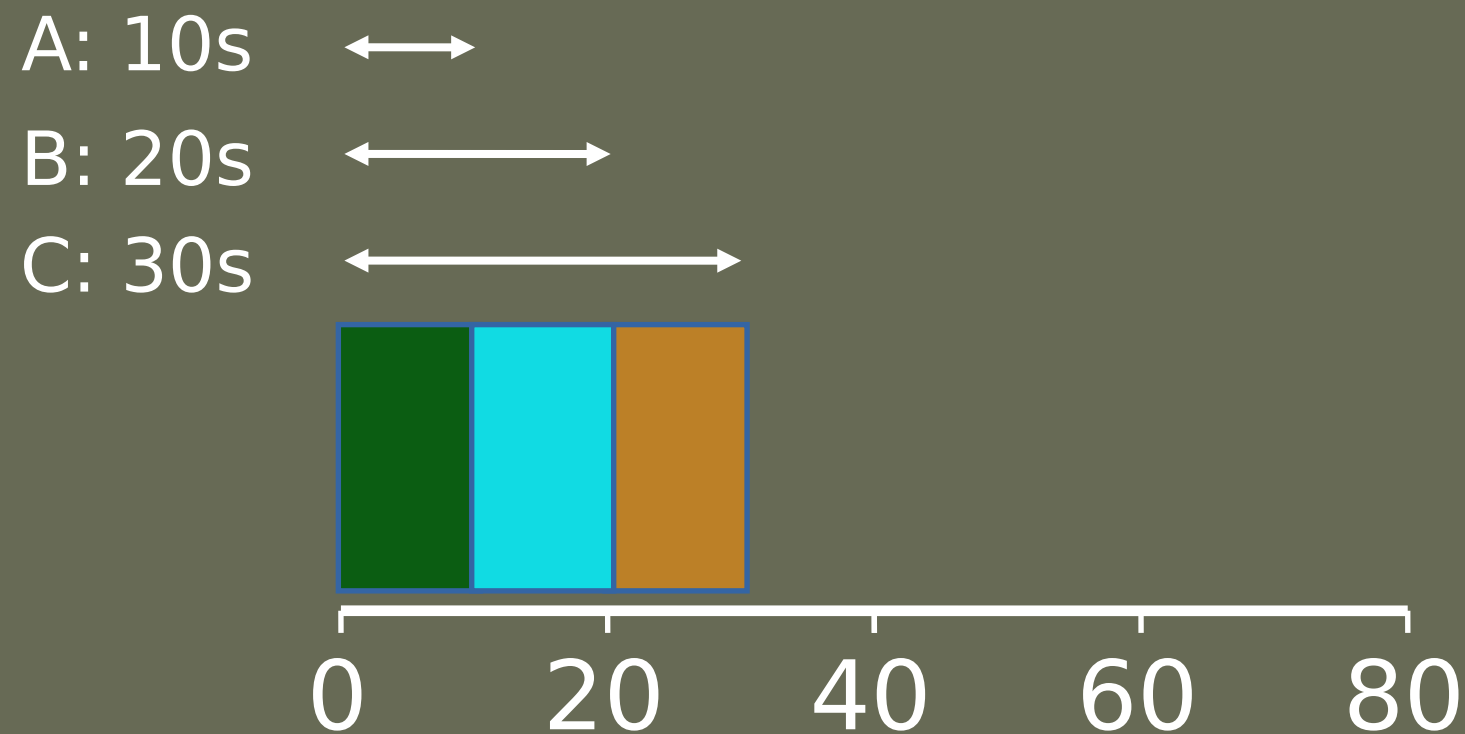
JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10



Gantt chart:

Illustrates how jobs are scheduled over time on a CPU

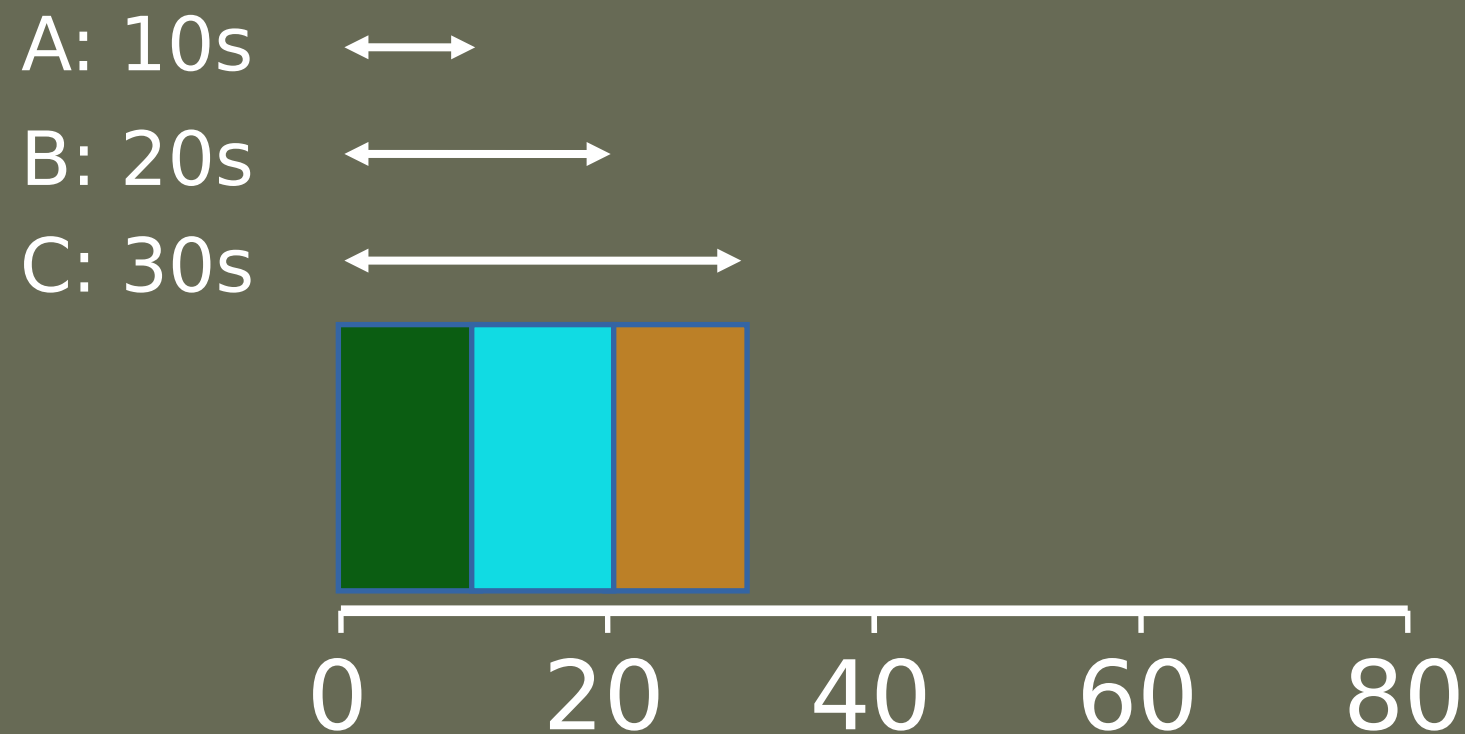
FIFO (IDENTICAL Jobs)



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:

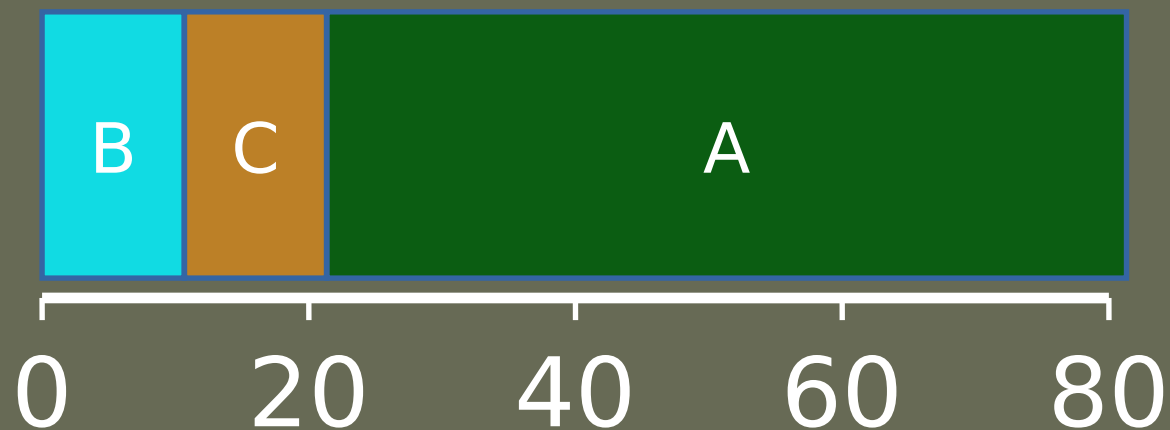
turnaround_time
response_time

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

Example: A Big Job Last

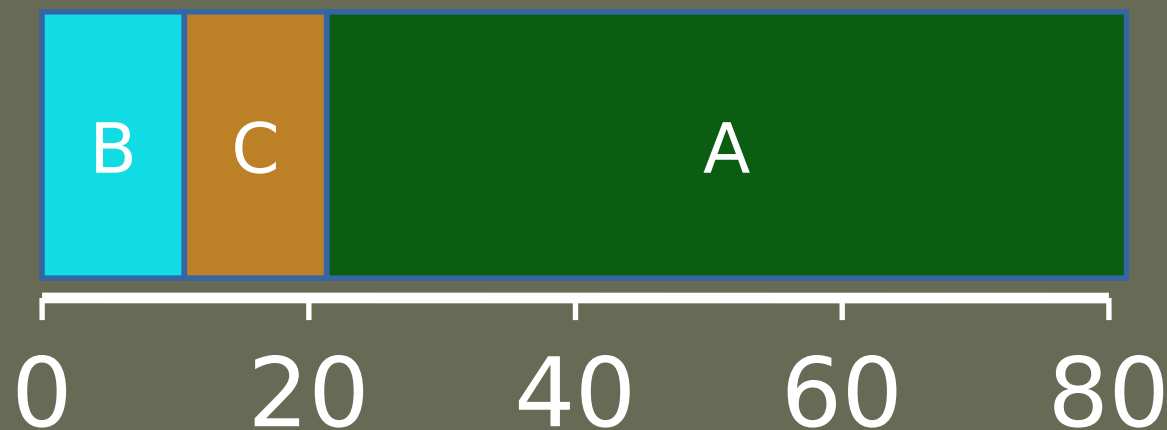
JOB	arrival_time (s)	run_time (s)
B	~0	10
C	~0	10
A	~0	60



What is the average turnaround time?

Example: A Big Job Last

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



What is the average turnaround time?

$$(10 + (10+10) + (10+10+60)) / 3 = 36.67s$$

Any Problematic Workloads for FIFO?

Workload: ?

Scheduler: FIFO

Metric: turnaround is high

Example: A Big Job First

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

What is the average turnaround time?

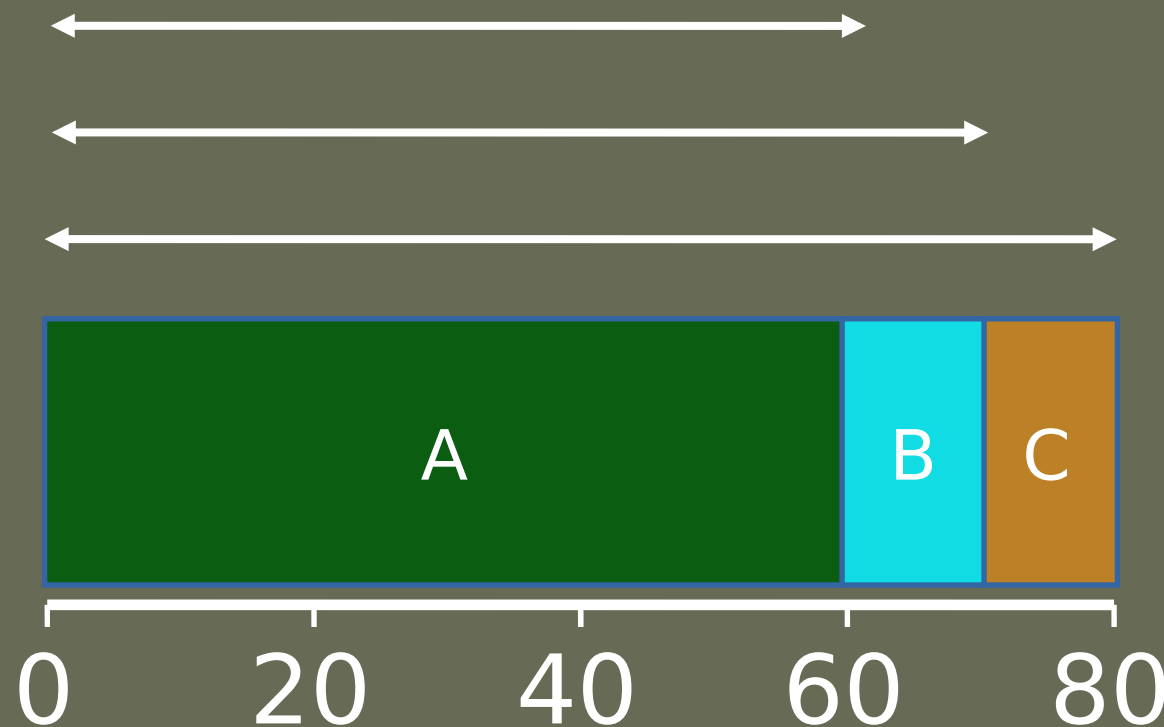
Example: Big First Job

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

A: 60s

B: 70s

C: 80s



Average turnaround time:

$$(60 + (10+60) + (10+70)) / 3 = 70s$$

Convoy Effect

When a long job early on slows all jobs



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)

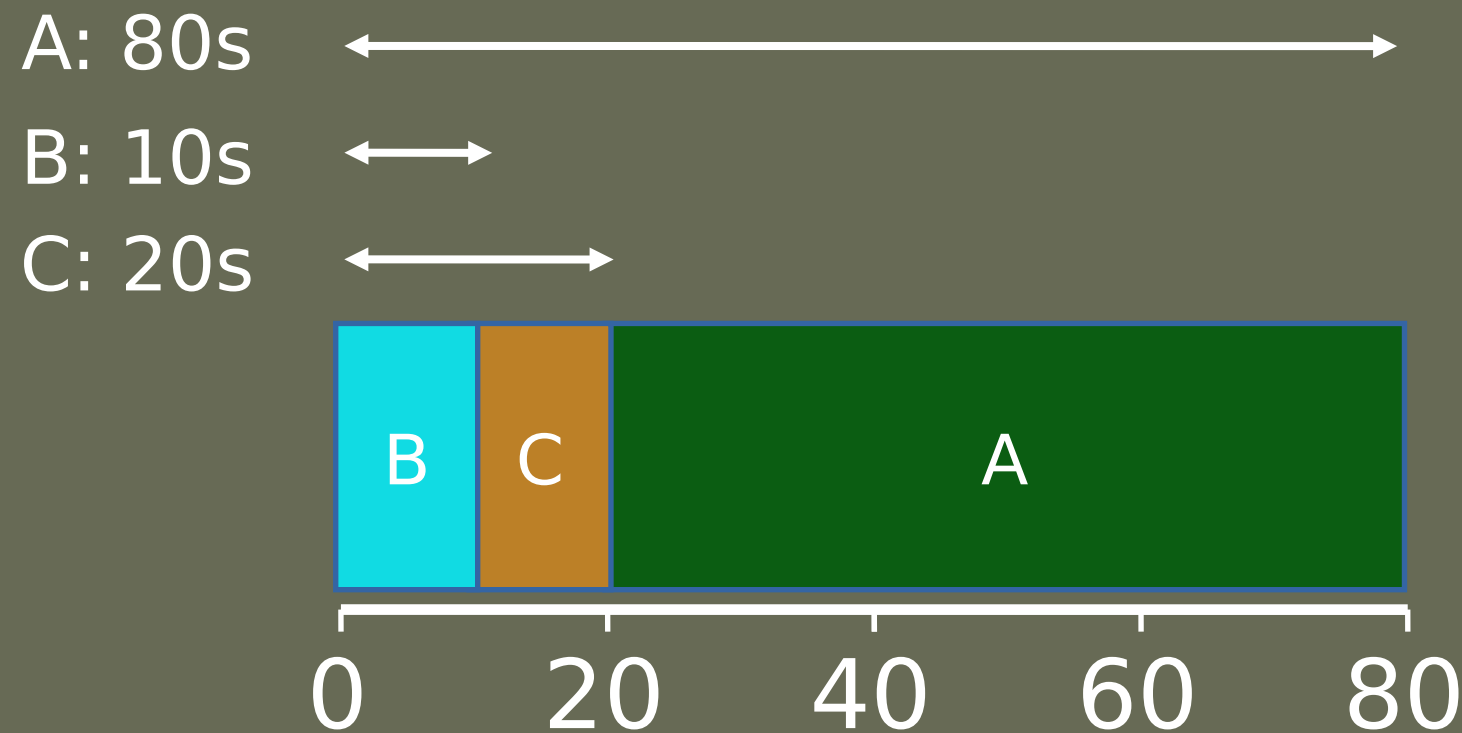
Choose job with smallest *run_time*

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?

$$(80 + 10 + 20) / 3 = \sim 36.7s$$

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

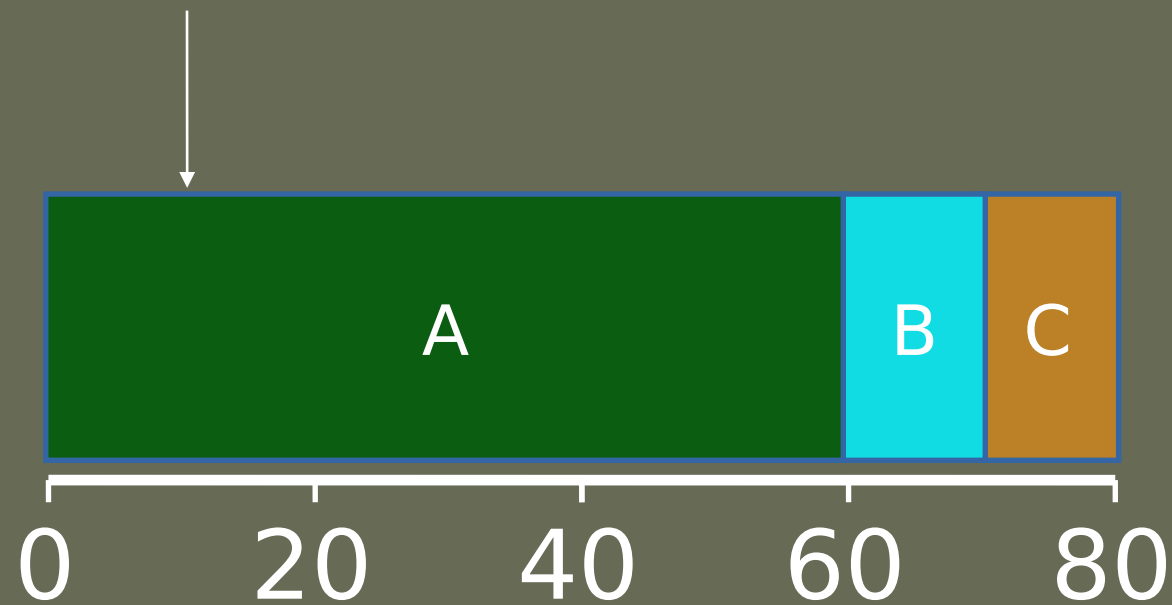
Shortest Job First (Arrival Time)

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

[B,C arrive]



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:

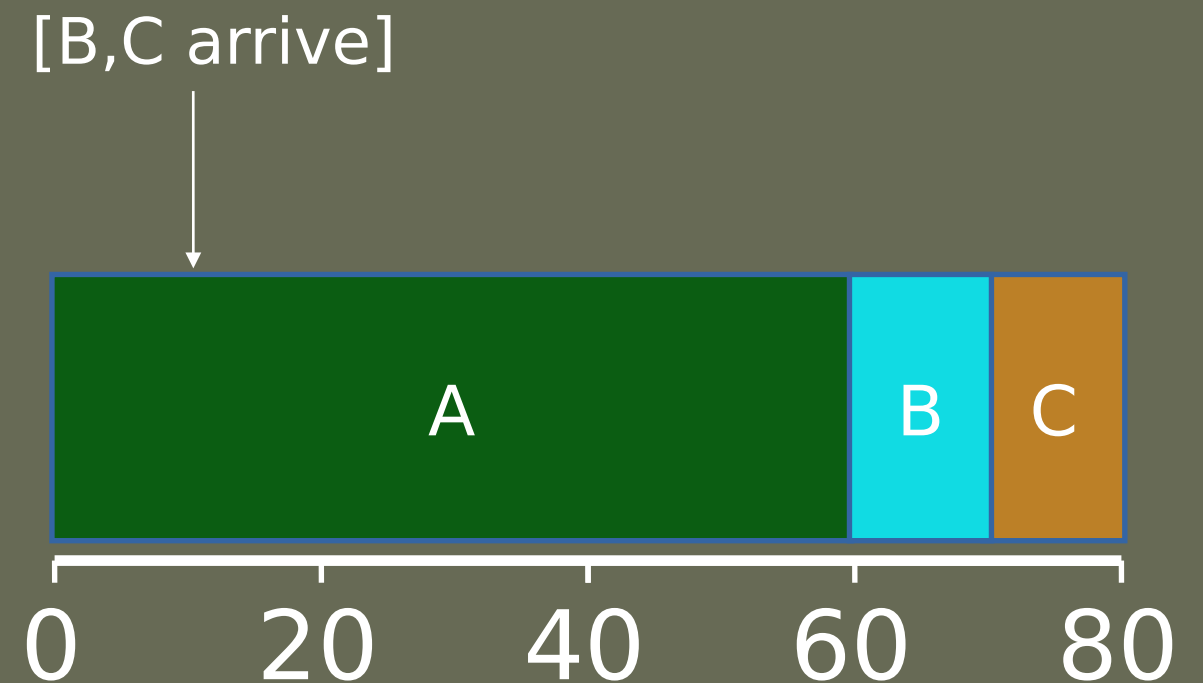
Preemptive: Potentially schedule different job at any point by taking CPU away from running job

STCF (Shortest Time-to-Completion First)

Always run job that will complete the quickest

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

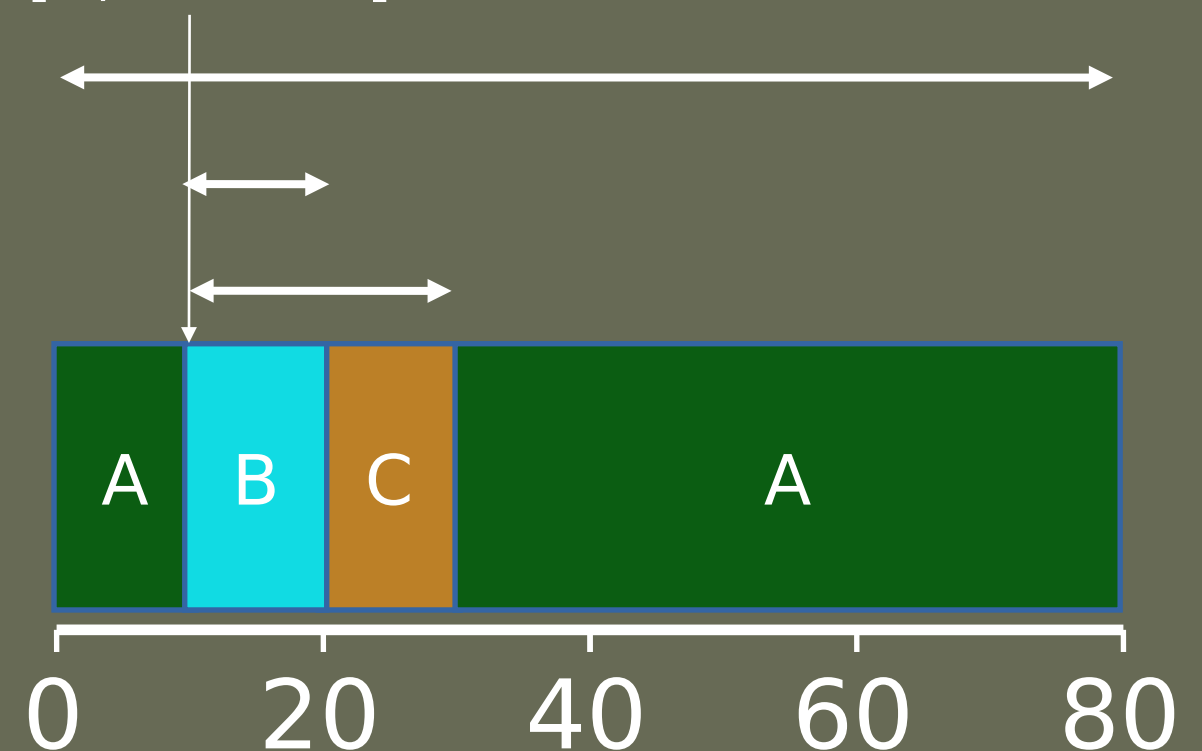
Turnaround time?

A: 80s

B: 10s

C: 20s

[B,C arrive]



Average turnaround time with STCF? **36.6s**

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 you care about when a job starts instead of when it finishes

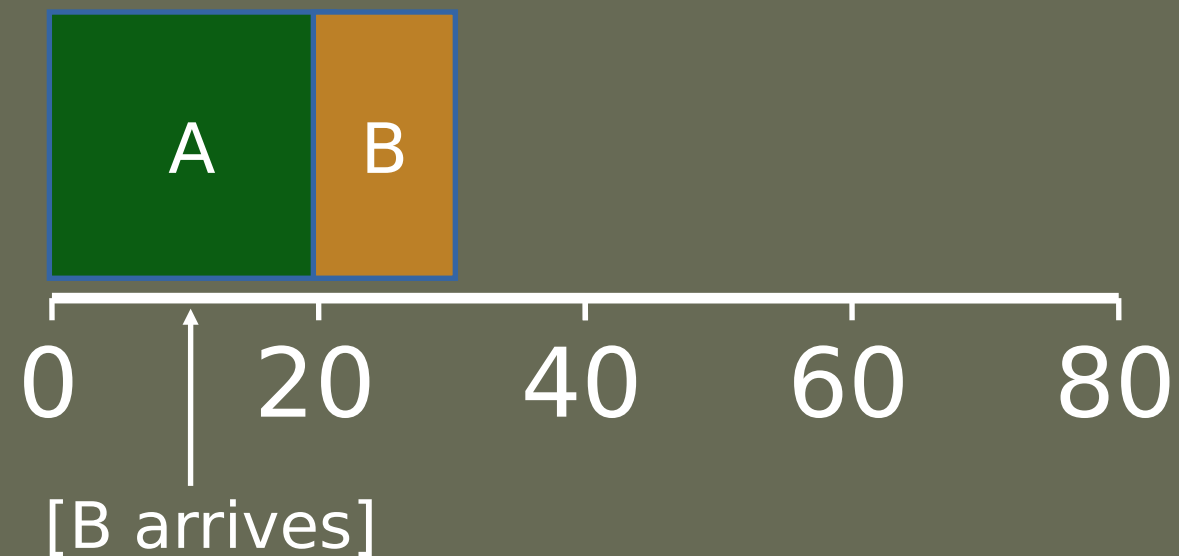
New metric:

$$\text{response_time} = \text{first_run_time} - \text{arrival_time}$$

Response vs. Turnaround

B's turnaround: 20s \longleftrightarrow

B's response: 10s \longleftrightarrow



Round-Robin Scheduler

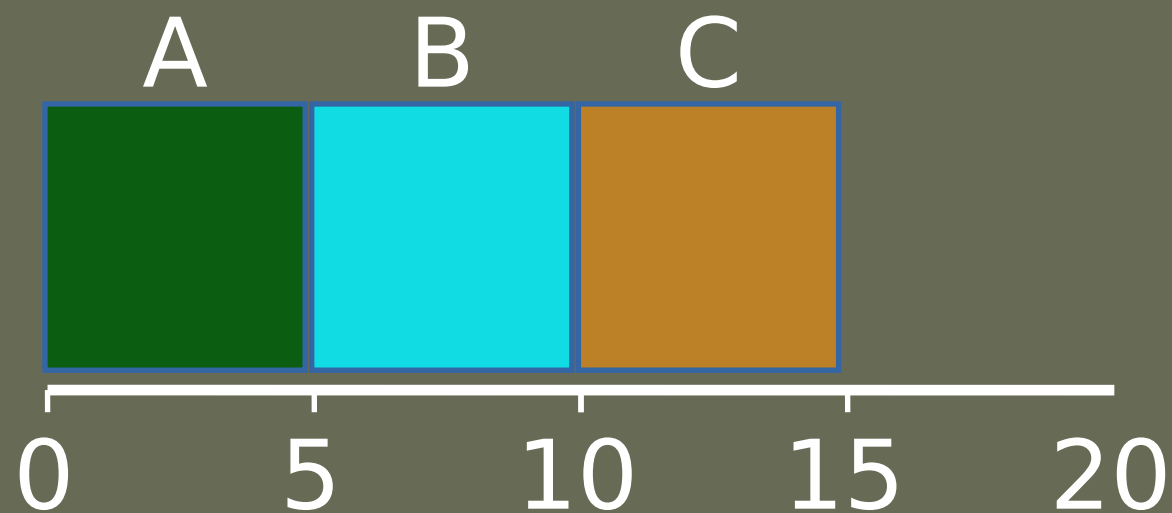
Prev schedulers:

FIFO, SJF, and STCF can have poor response time

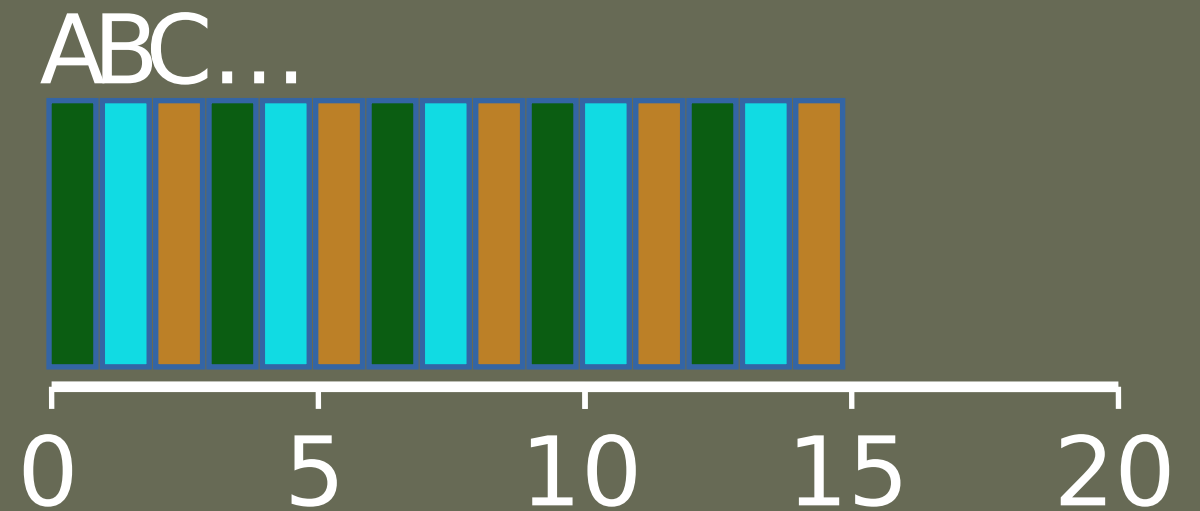
New scheduler: RR (Round Robin)

Alternate ready processes every fixed-length time-slice

FIFO vs RR



FIFO
 $(0+5+10)/3 = 5$

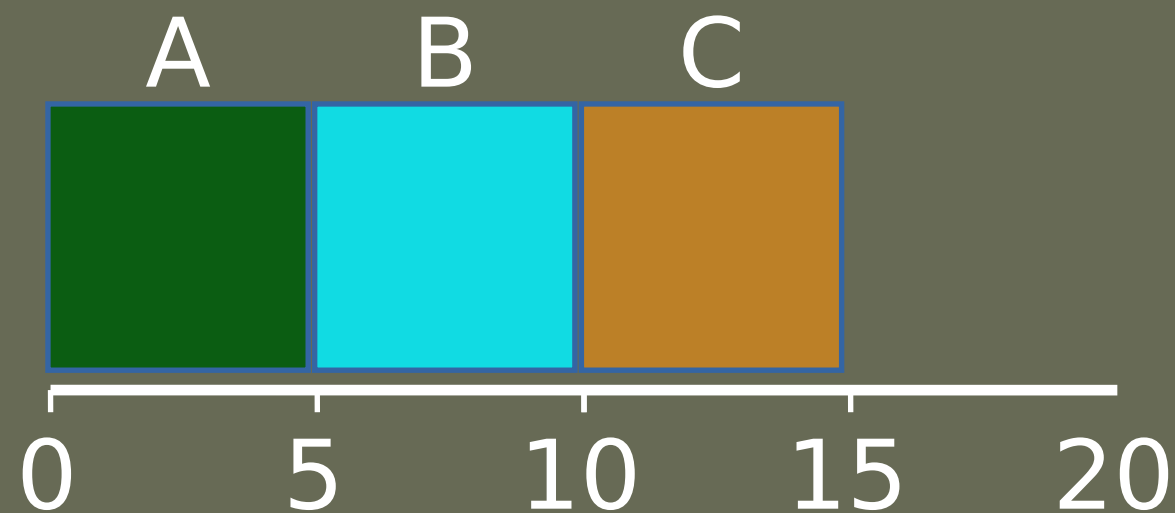


RR (time slice=1)
 $(0+1+2)/3 = 1$

What is the average response time (Assume all jobs arrive at time = 0s)?

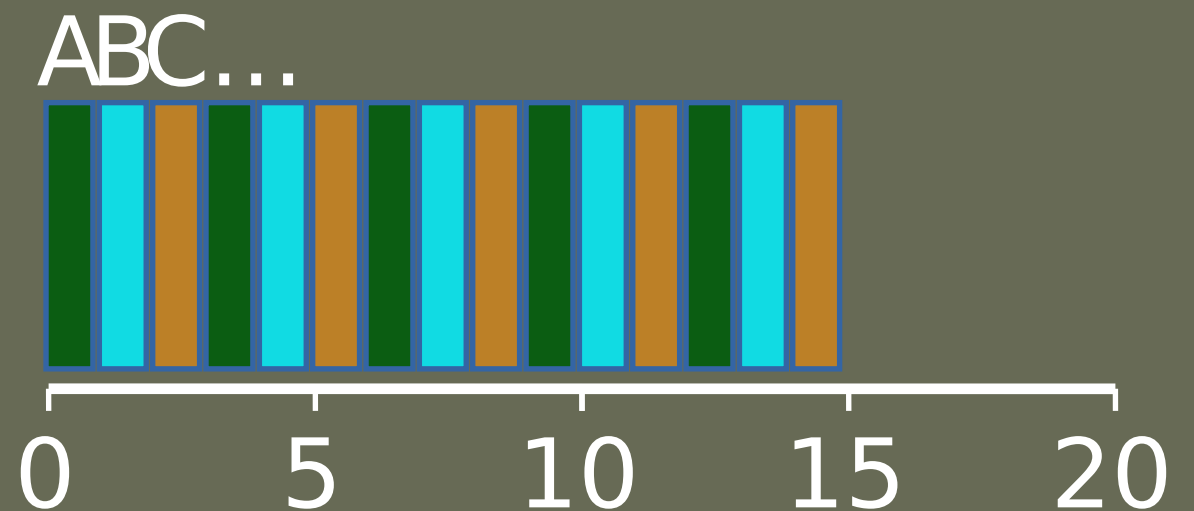
RR much more responsive, plus gives short jobs a chance to run and finish fast

FIFO vs RR



FIFO

$$(0+5)+(0+10)+(0+15))/3 = 10$$



RR (time slice=1)

$$(0+13)+(0+14)+(0+15))/3 = 14$$

What is the average turn-around time (Assume all jobs arrive at time = 0s)?

So average turn around time for RR is in general worse, especially for equal length jobs

Scheduling Basics

Workloads:

arrival_time
run_time

Schedulers:

FIFO
SJF
STCF
RR

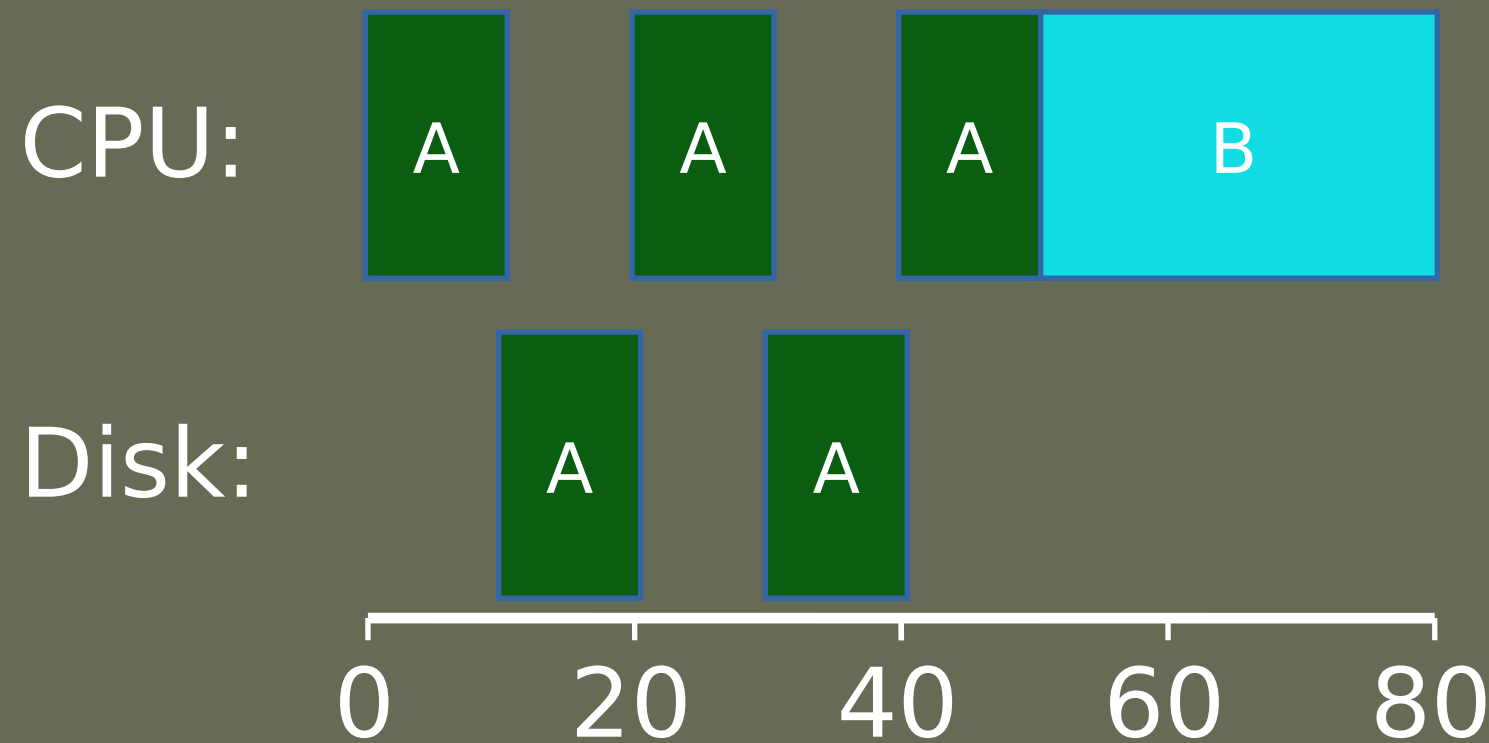
Metrics:

turnaround_time
response_time

Workload Assumptions

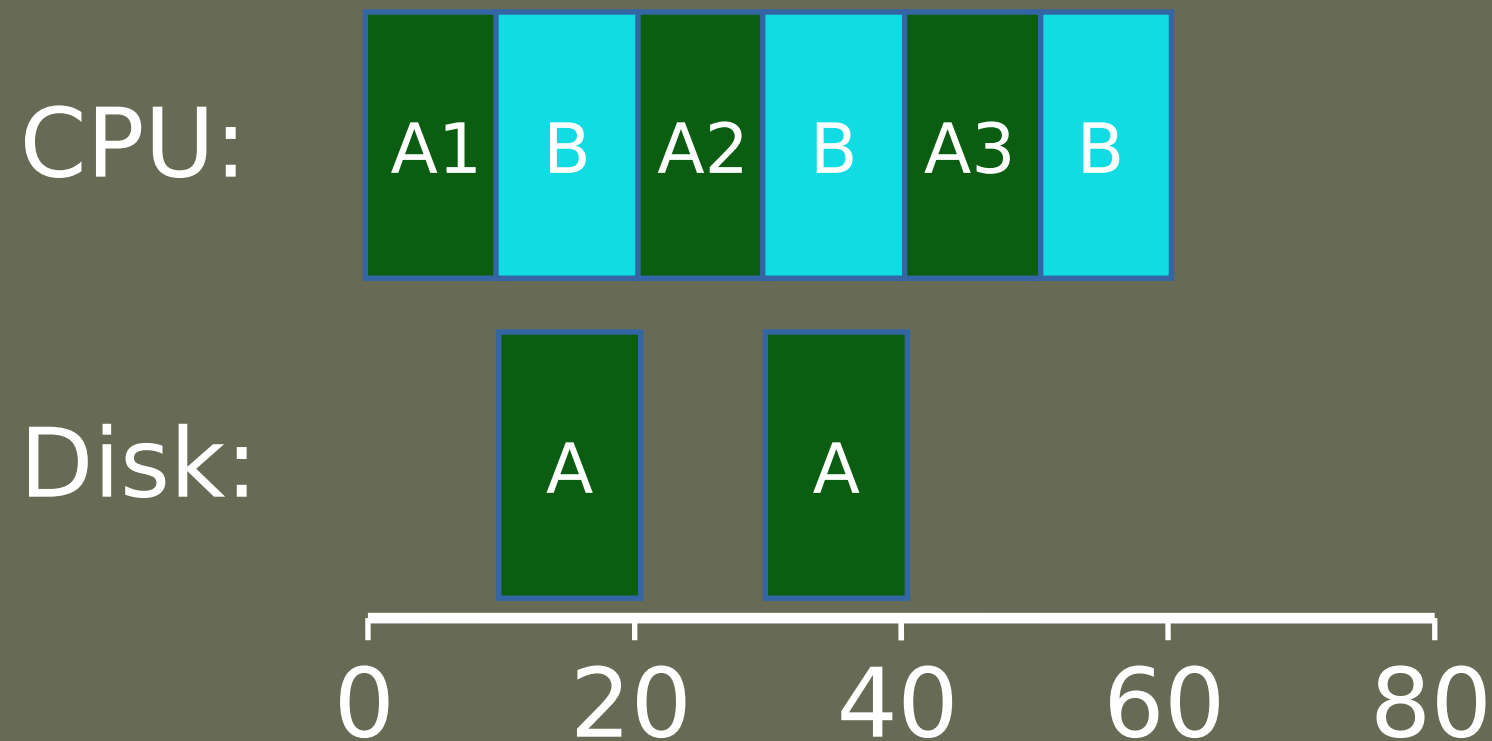
- ~~1. Each job runs for the same amount of time~~
- ~~2. All jobs arrive at the same time~~
- ~~3. All jobs only use the CPU (no I/O)~~
4. The run-time of each job is known

Not I/O Aware



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
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~~
(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

Priorities

Rule 1: If $\text{priority}(A) > \text{Priority}(B)$, A runs

Rule 2: If $\text{priority}(A) == \text{Priority}(B)$, A & B run in RR

Q3 → 

Q2 → 

Q1

Q0 →  → 

“Multi-level”

How to know how to set
priority?

Approach: use 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 $\text{priority}(A) > \text{Priority}(B)$, A runs

Rule 2: If $\text{priority}(A) == \text{Priority}(B)$, A & B run in RR

More rules:

Rule 3: Processes start at top priority

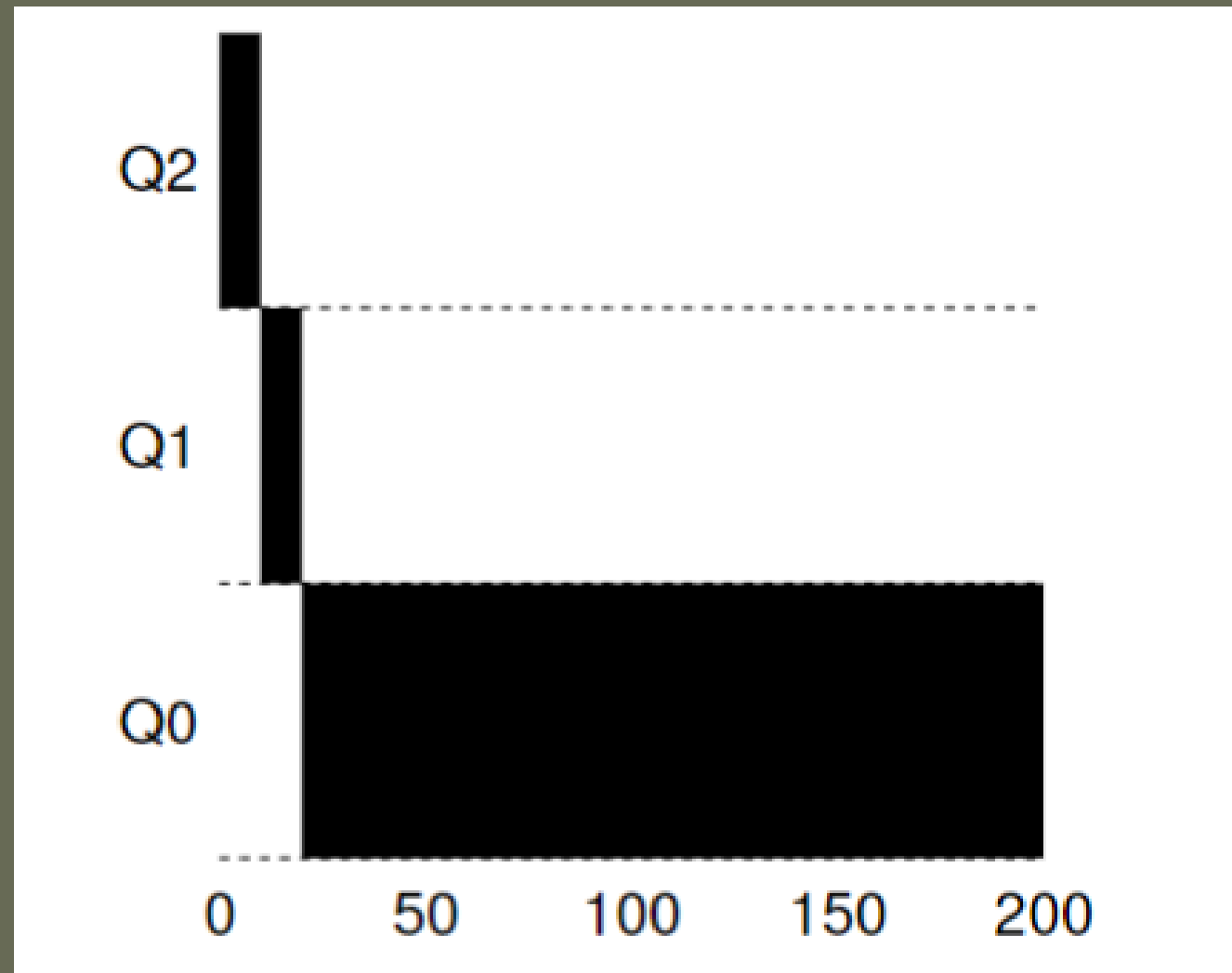
Rule 4: If job uses whole slice, demote process
(longer time slices at lower priorities)

Rule 4 reasoning-

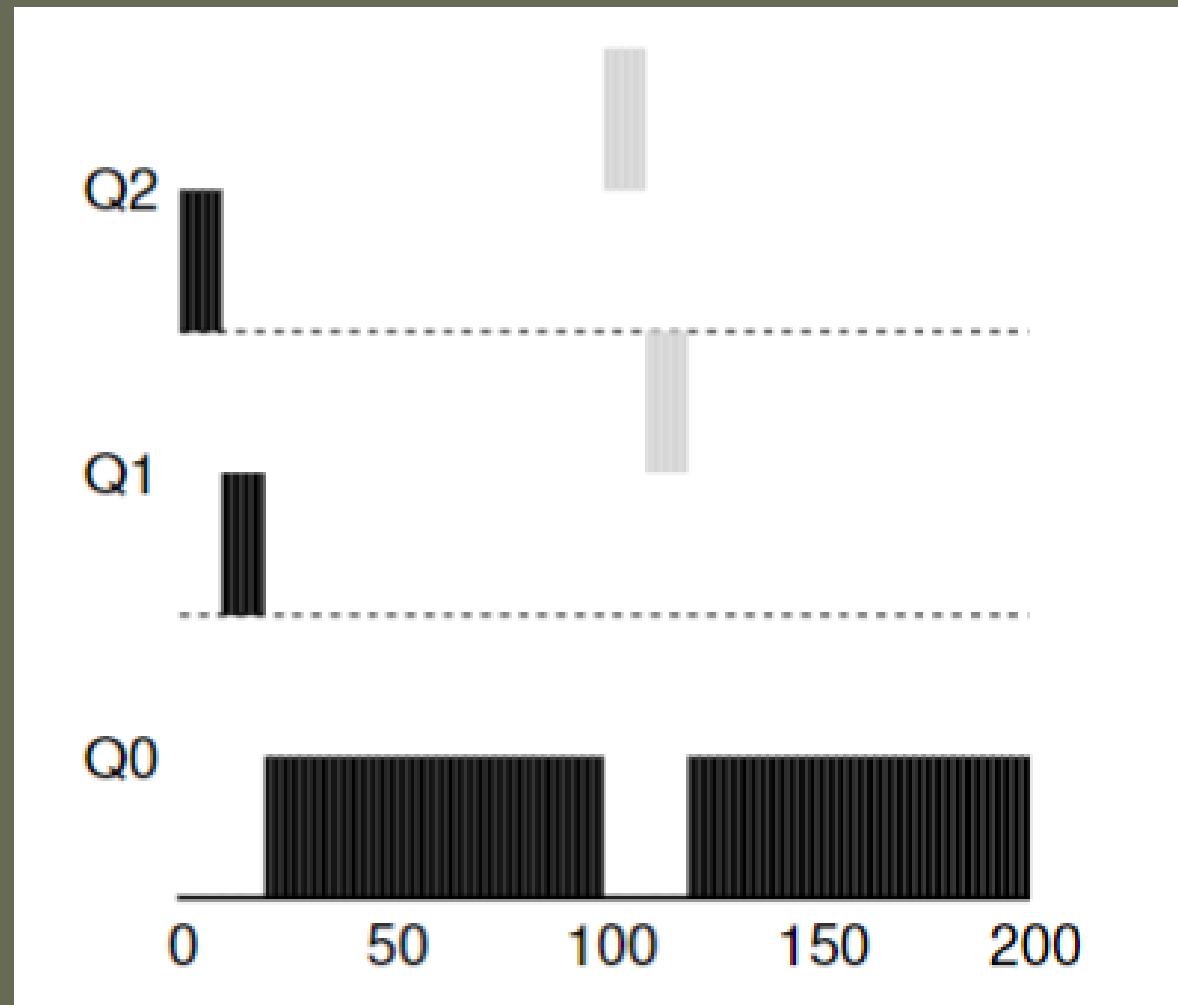
If job **DOES NOT** use all time slice its probably interacting with user, keep TS low to boost interactivity

If job **DOES** use all time slice, its likely not interacting with user, does not need to be as interactive, push to lower priority queue (which has a longer TS)

One Long Job (Example)

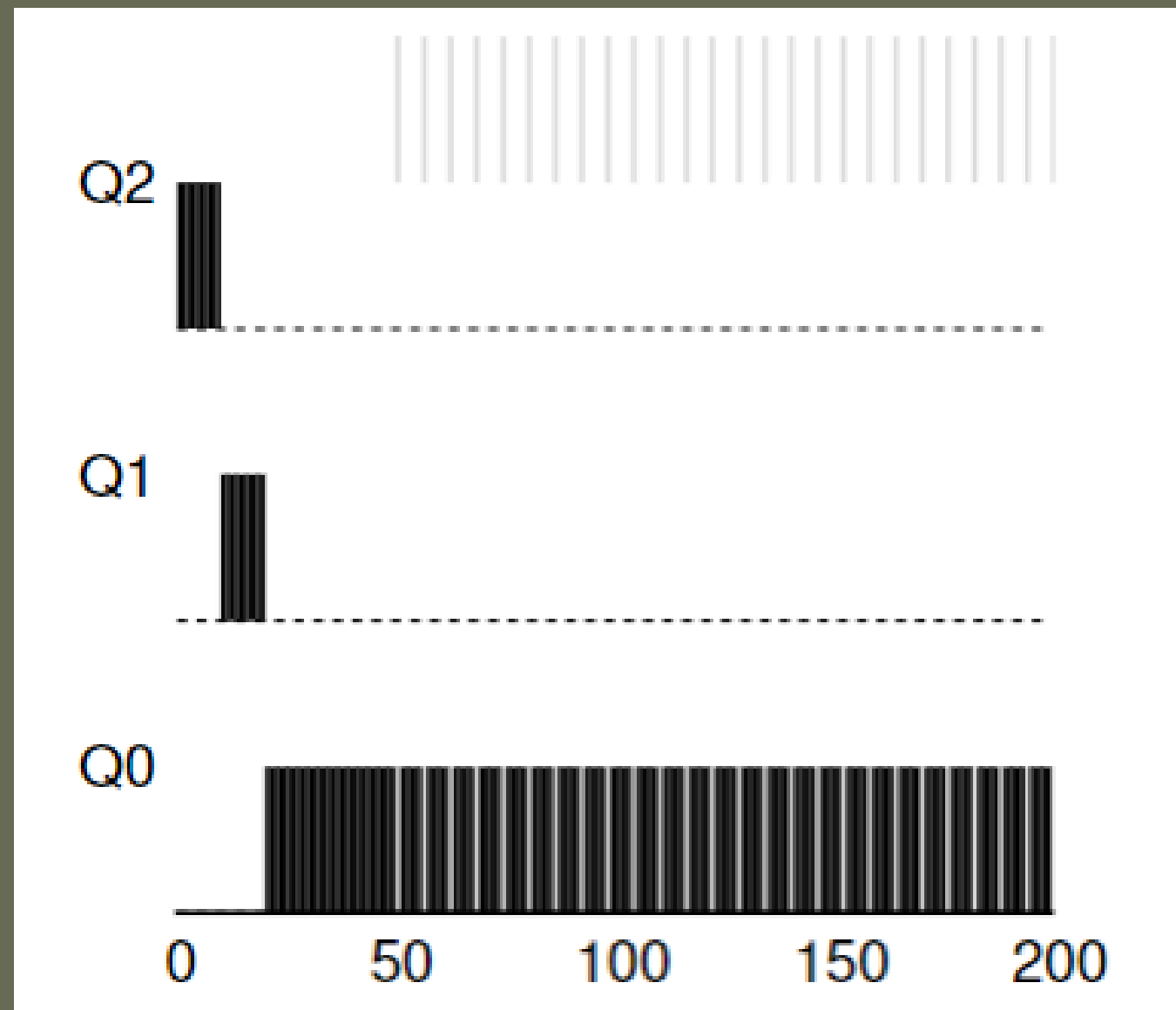


Short job



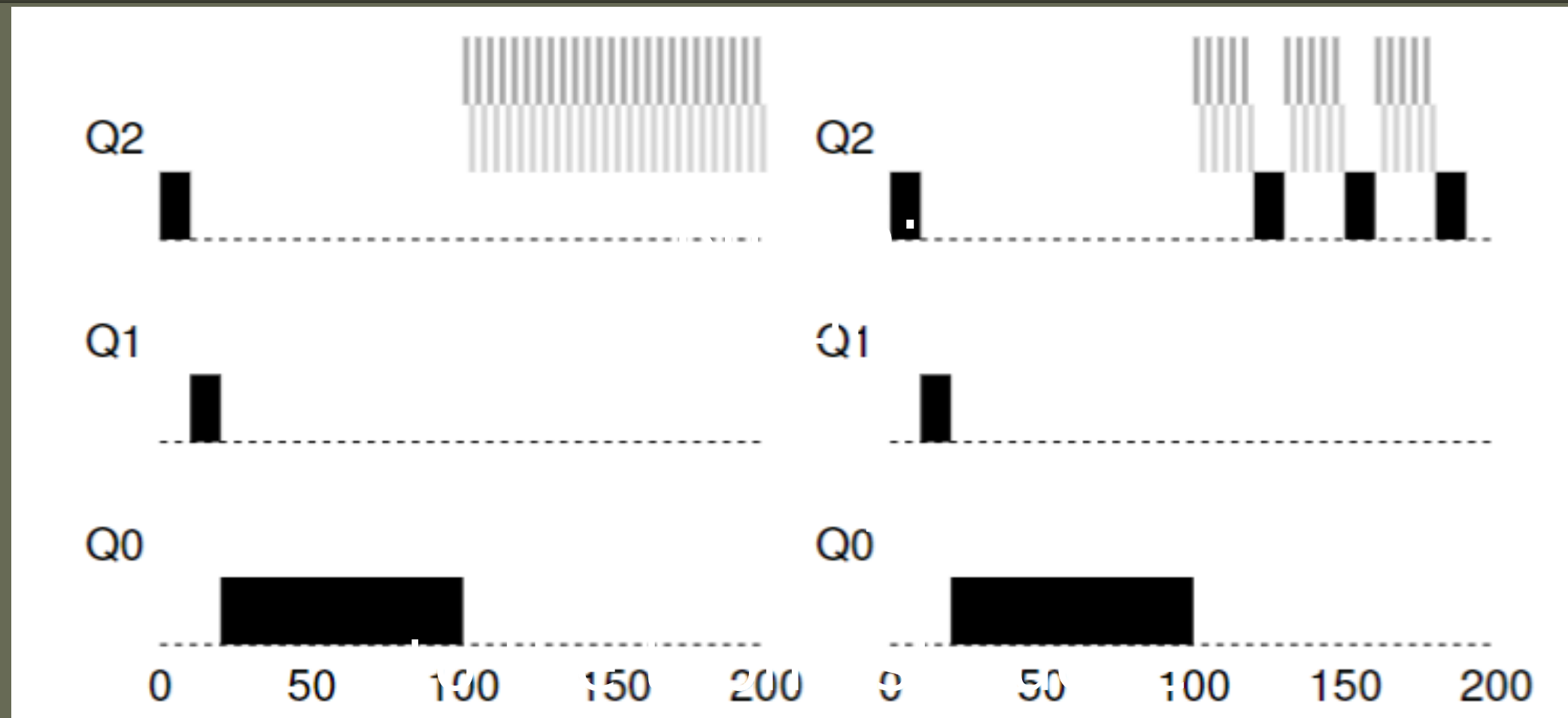
Short job starts in high priority Q2
Its lifetime run approximates SJF (gets all CPU time)

An Interactive Process Joins



Interactive process never uses entire time slice,
so never demoted, so stays in a very responsive
(short time slice queue)

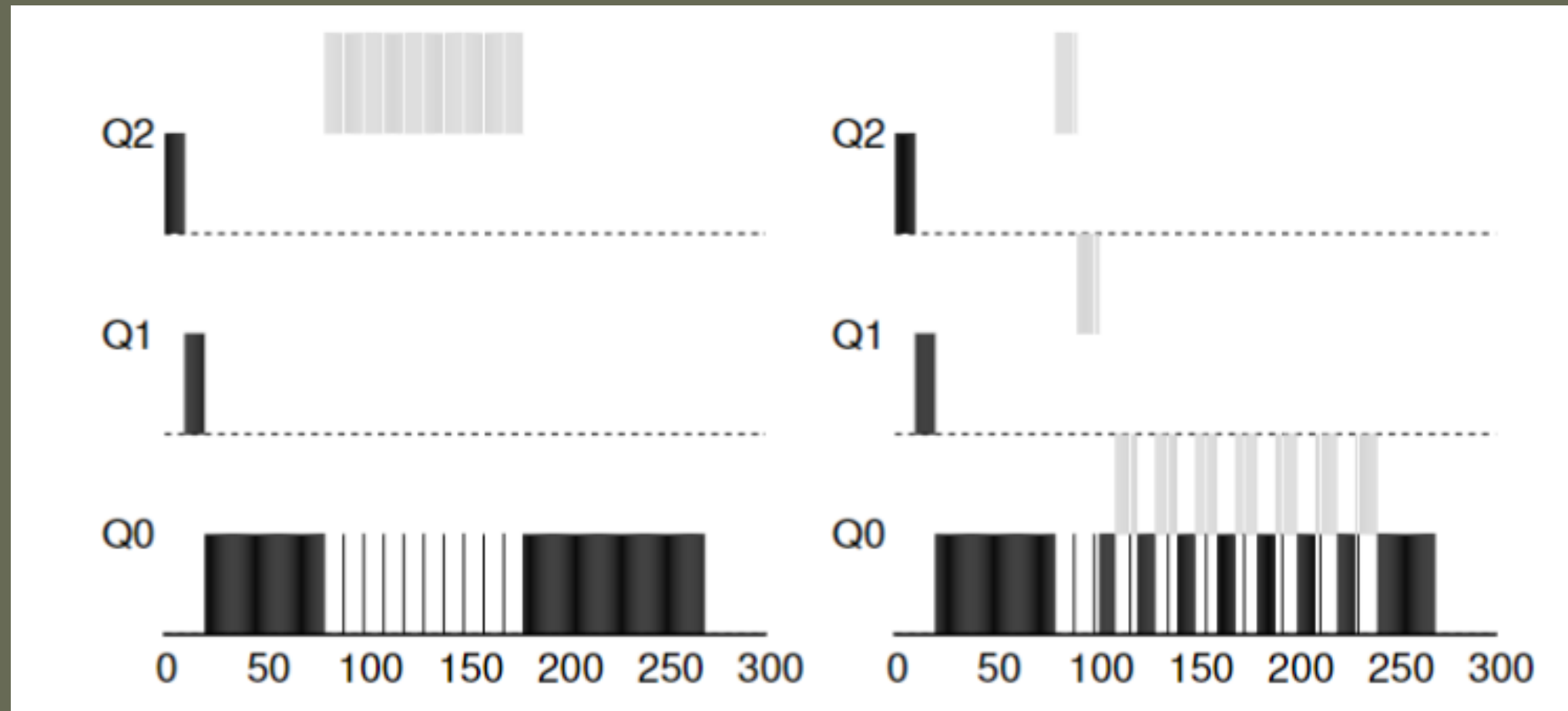
Starvation -low priority processes never run



Occasionally boost priority of low priority processes
So they get processor time or

Rule 5: After some time period S , move all the jobs in the system to the topmost queue (what value is S ?)

Gaming system – process uses up most of TS, then invokes IO call



Easy to cheat – so track cpu usage instead of I/O calls

Rule 4 redo: 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

MLFQ – the rules

Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).

Rule 2: If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in round-robin fashion using the time slice (quantum length) of the given queue.

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 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).

Rule 5: After some time period S , move all the jobs in the system to the topmost queue.

Lottery Scheduling

Goal: proportional (fair) share

Approach:

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

Simple to implement

Lottery Code

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//           get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Head- holds list of jobs, sorted by ticket numbers held

Lottery example

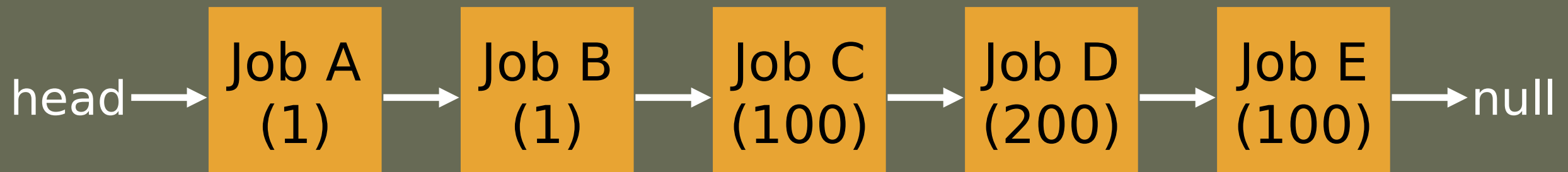
```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//           get a value, between 0 and the total # of tickets
int winner = getRandom(0, totaltickets);

// current: use this to walk through the list of jobs
current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Who runs if winner
is:

50
350
0



Other Lottery Ideas

Ticket Transfers

Ticket Currencies

Ticket Inflation

(read more in OSTEP)

Summary

Understand goals (metrics) and workload, then design scheduler around those

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 avoid corner cases.