

CS5460: Operating Systems

Lecture 5: Scheduling

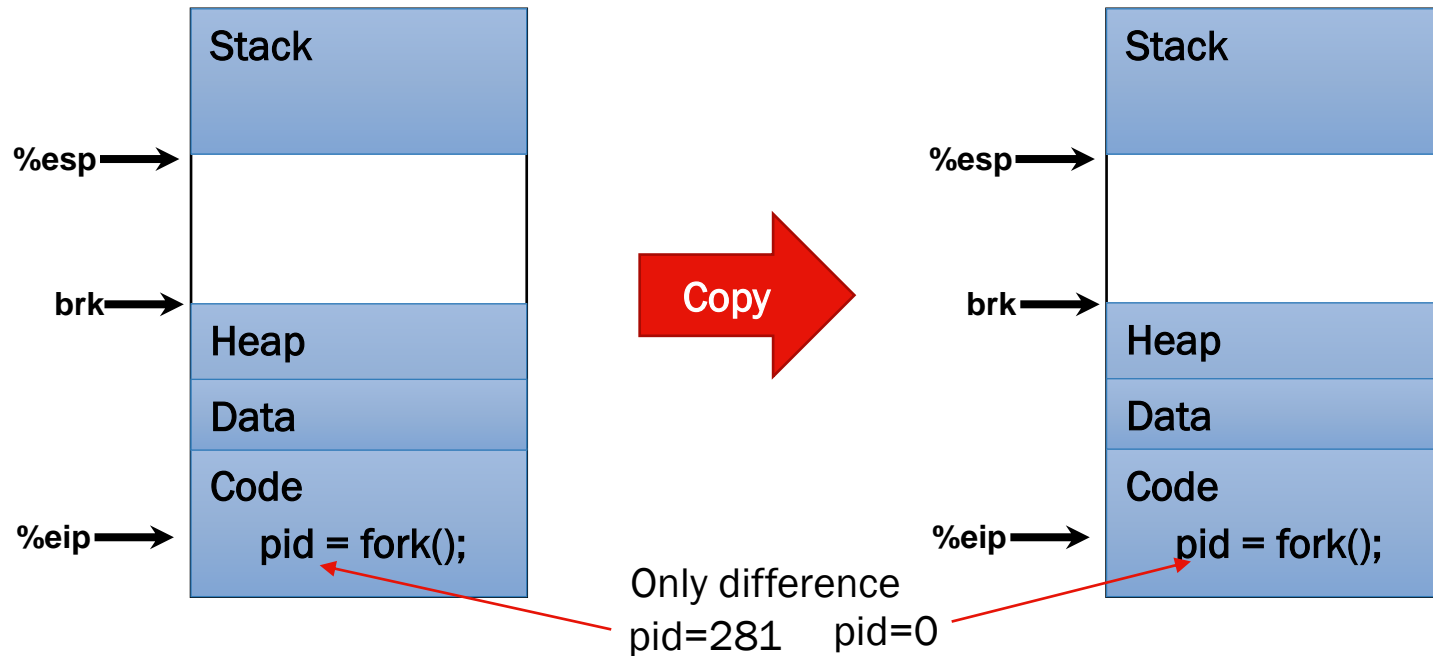
(Chapters 7, 8, 9)

Slide Credit: Andrea Arpaci-Dusseau

Assignment 1

- Due Feb 2, tonight before midnight!

Semantics of fork()



- fork(), exit(), and exec() are weird!
 - fork() returns twice – once in each process
 - exit() does not return at all
 - exec() usually “does not return”: replaces process’ program

fork() and wait()

```
int main(int argc, char *argv) {  
    printf("parent %d\n", (int)getpid());  
    pid_t rc = fork();  
    assert(rc >= 0)
```

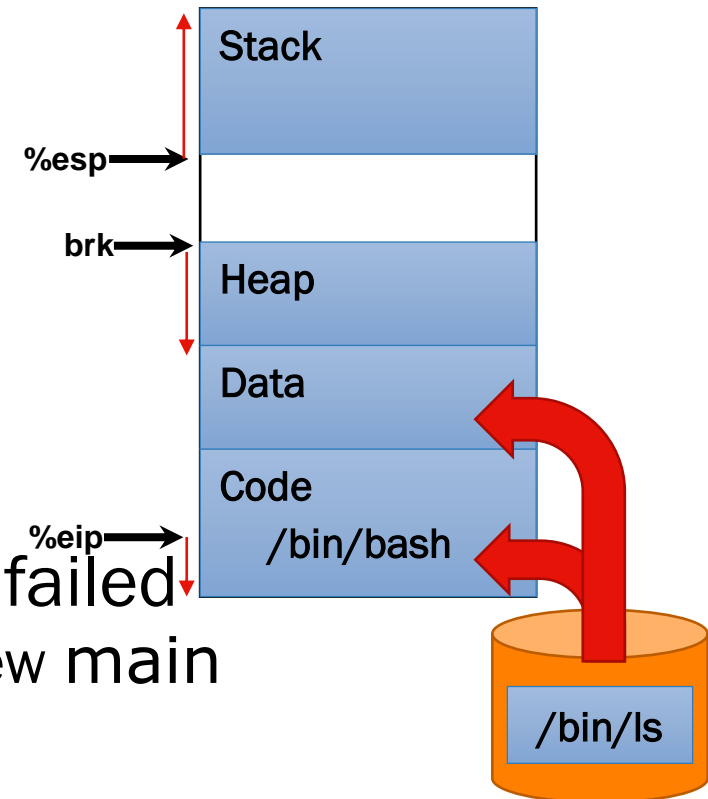
```
$ ./fork-wait  
parent 13037  
child 13039  
13037 is parent of 13039
```

```
    if (rc == 0) {                                // child  
        printf("child %d\n", (int)getpid());  
    } else {                                       // parent  
        wait(NULL);  
        printf("%d is parent of %d\n", (int)getpid(), rc);  
    }  
    return 0;  
}
```

**What is the
output without
wait()?**

exec(): Run Another Program

- Can't write entire system in one program!
 - Need to replace process with another program
- Loads program from filesystem
 - Replaces code, data segments
 - Put argv on stack; reset %esp
 - Release heap memory
 - Reset %eip to main
- exec() only returns to caller if failed
 - Otherwise process is now in a new main



Why Separate fork and exec?

- Lots of parameters on creating a process
 - Shell may want to
 - redirect output of children
 - change child environment
 - change child working directory
 - run child as a different user
- Hard to create simple, expressive-enough API
- Separation allows policy to be expressed in parent's program but in child's process
- Tradeoff: child may inherit things it doesn't need (or shouldn't have)

Example: Output Redirection

```
pid_t rc = fork();

if (rc == 0) {                                // child
    close(STDOUT_FILENO);                      // close fd 1
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU); // new fd 1

    const char *myargs[3];
    myargs[0] = "wc";
    myargs[1] = "p4.c";
    myargs[2] = NULL;
    execvp(myargs[0], myargs); // runs "wc p4.c > p4.output"
} else {                                       // parent
    wait(NULL);
}
```

Termination: `exit()`, `kill()`

- When process dies, OS reclaims resources
 - Record exit status in PCB
 - Close files, sockets
 - Free memory
 - Free (nearly) all kernel structures
- Process terminates with `exit()`
- Process terminates another with `kill()`

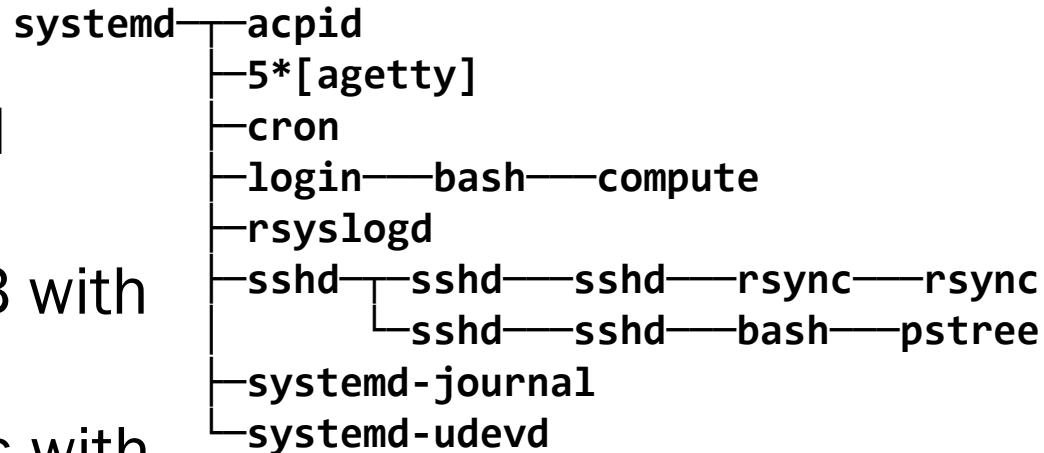
```
int main(int argc, char* argv[]) {  
    pid_t pid = fork();  
    if (pid == 0) {  
        sleep(10);  
        printf("Child exiting!\n");  
        exit(0);  
    } else {  
        sleep(5);  
        if (kill(pid, SIGKILL) != -1)  
            printf("Sent kill!\n");  
    }  
}
```


Orphans and Zombies

- Parent `wait()` on child returns status
- Must keep around PCB with status after child exit
- **Zombie**: exited process with uncollected status
- Parent exits before child?

Orphaned

- `init` adopts orphans
- Collects and discards status of reparented children after exit
- Useful for “daemons” (`nohup`)



Important Terms and Ideas

- Process, programs
- Process Control Blocks
- syscall, user/kernel mode
- Dispatcher, context switch
- Process State Machine
- New (Embryo), Ready, Running, Blocked, Terminated (Zombie)
- fork(), wait(), exec(), exit(), kill()
- Orphans, zombies, init

CPU Virtualization:

2 Components

Dispatcher (Previously)

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

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
- 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

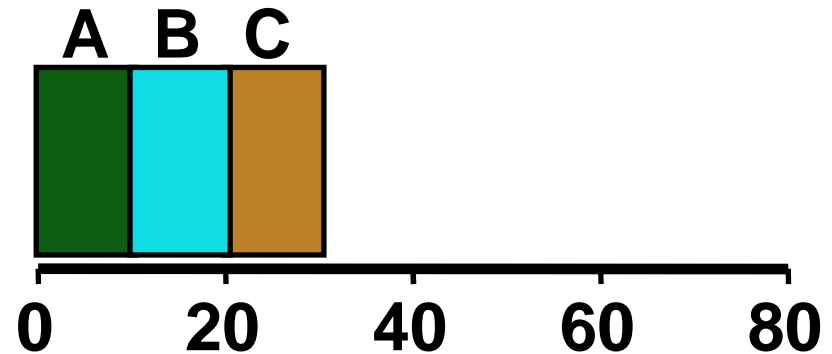
What is our turnaround? $completion_time - arrival_time$

FIFO: Event Trace

JOB	arrival_time (s)	run_time (s)	Time	
A	~0	10	0	A arrives
B	~0	10	0	B arrives
C	~0	10	0	C arrives
			0	run A
			10	complete A
			10	run B
			20	complete B
			20	run C
			30	complete C

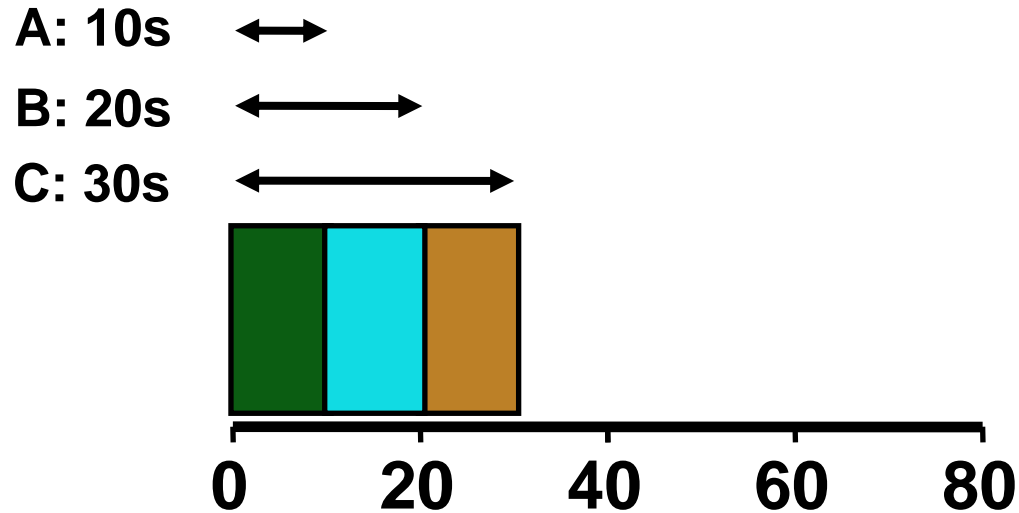
FIFO (Identical Jobs)

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?

$$\text{turnaround_time} = \text{completion_time} - \text{arrival_time}$$

$$(10 + 20 + 30) / 3 = 20\text{s}$$

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

Problematic Workloads for FIFO?

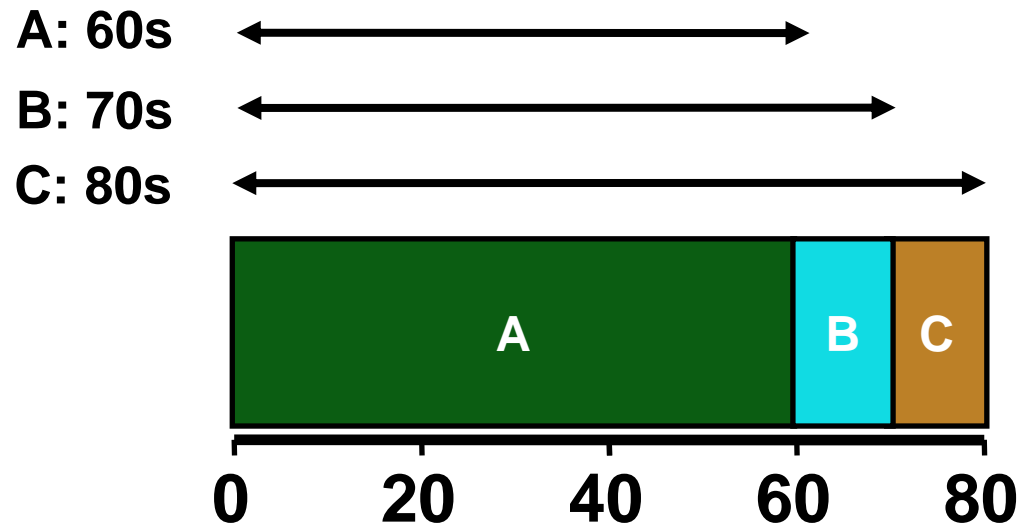
Workload: ?

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



Average turnaround time: **70 s**

Convoy Effect



Passing the Tractor

Problem with Previous Scheduler:

FIFO: Turnaround time suffers when short jobs wait behind long jobs

New scheduler:

SJF (Shortest Job First)

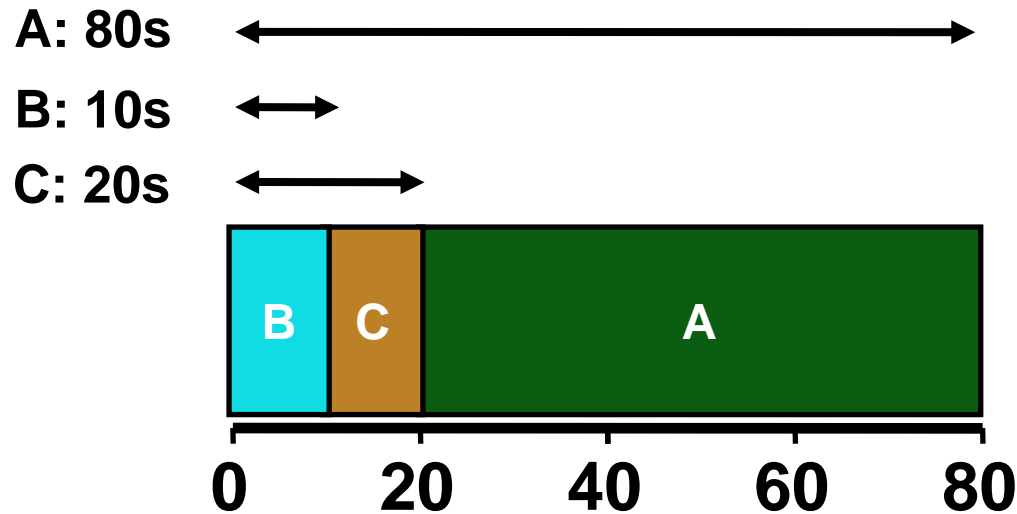
Choose job with smallest *run_time*

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?

$$(80 + 10 + 20) / 3 = \sim 36.7s \quad \text{Average turnaround with FIFO: 70s}$$

SJF optimal in minimizing turnaround time (when no preemption)

Shorter job before longer job improves turnaround time of short more than it harms turnaround time of long

Scheduling Basics

Workloads:

arrival_time

run_time

Schedulers:

FIFO

SJF

STCF

RR

Metrics:

turnaround_time

response_time

Workload Assumptions

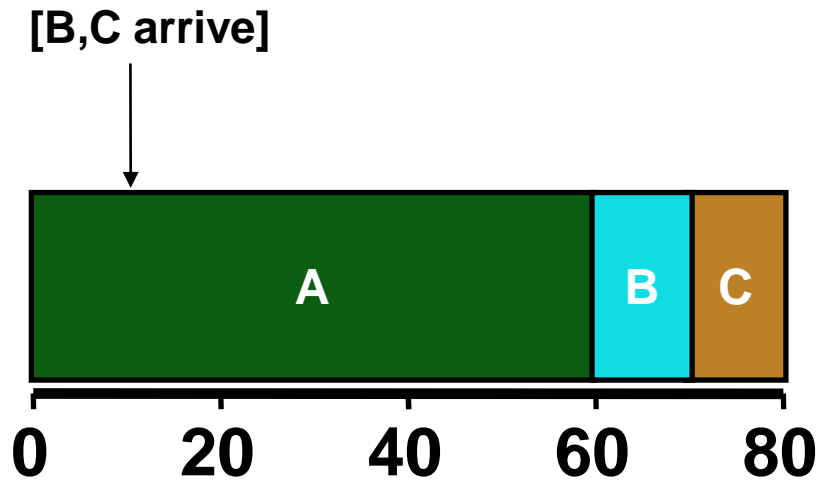
- ~~1. Each job runs for the same amount of time~~
- ~~2. All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. The run-time of each job is known

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

Previous 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

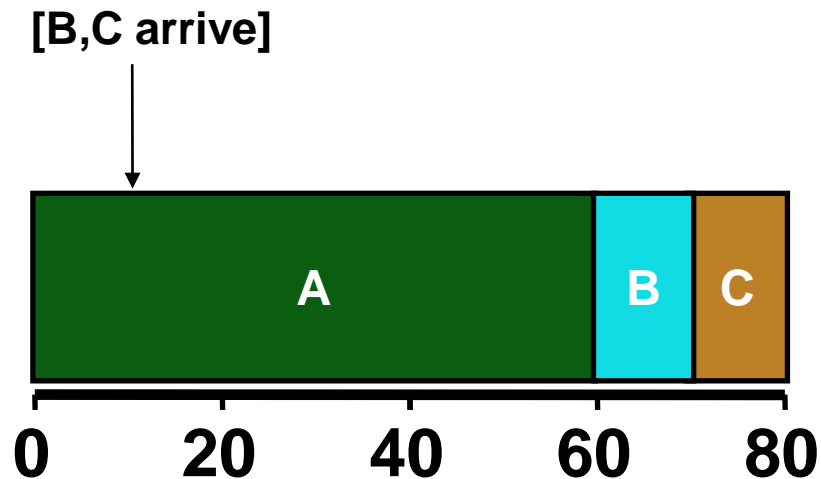
Non-Preemptive: SJF

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
---	----	----

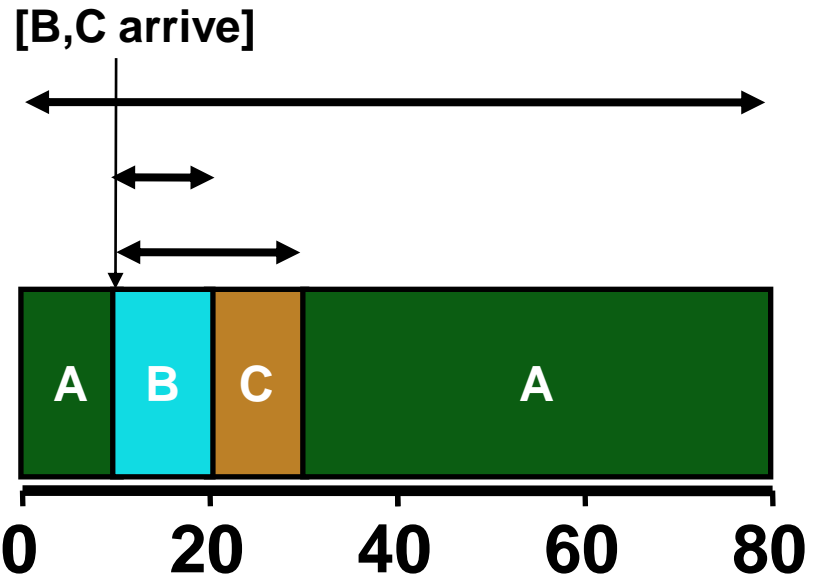
B	~10	10
---	-----	----

C	~10	10
---	-----	----

A: 80s

B: 10s

C: 20s



Average turnaround time with STCF? 36.7

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

SCTF okay for batch systems, but for time sharing systems when a job completes is less important

Sometimes care about when job starts instead of when it finishes

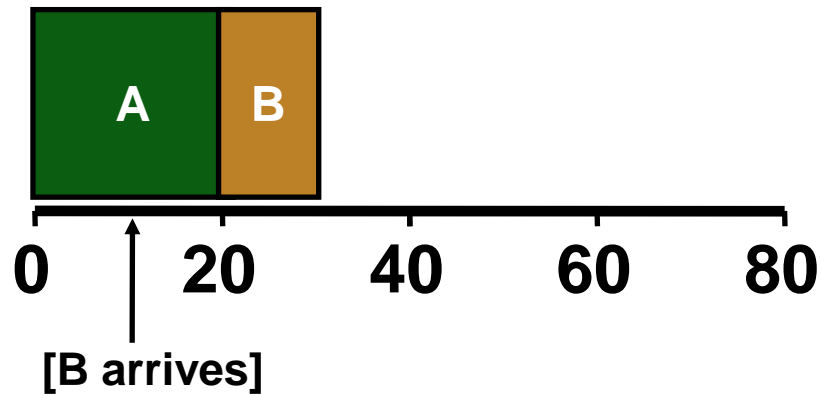
New metric:

$$response_time = first_run_time - arrival_time$$

Response vs. Turnaround

B's turnaround: 20s \longleftrightarrow

B's response: 10s \longleftrightarrow



Round-Robin Scheduler

Previous schedulers:

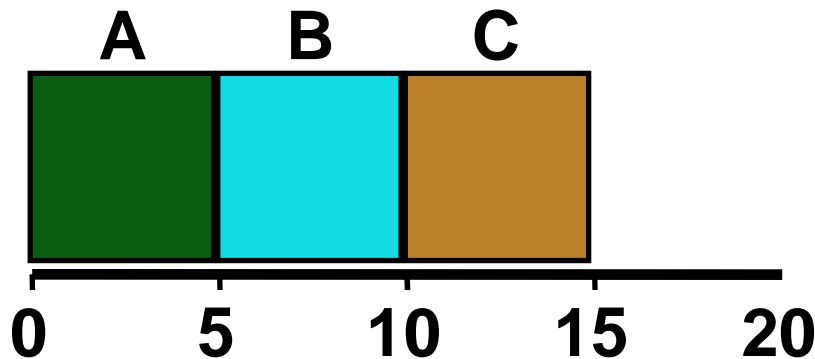
FIFO, SJF, and STCF can have poor response time

New scheduler: RR (Round Robin)

Alternate ready processes

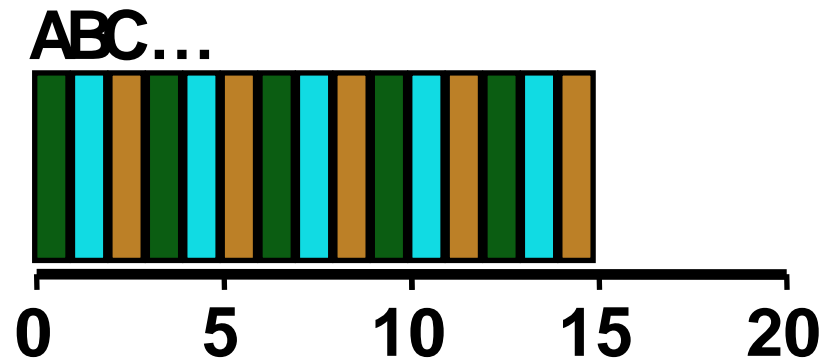
Switch after fixed-length **time-slice** (or **quantum**)

FIFO vs Round-Robin



Avg Response Time?

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



Avg Response Time?

$$(0+1+2)/3 = 1$$

In what way is RR worse?

Avg turn-around time with equal job lengths is horrible

Other reasons why RR could be better?

If run-times unknown, short jobs get chance to run, finish fast

Fair

Scheduling Basics

Workloads:

arrival_time

run_time

Schedulers:

FIFO

SJF

STCF

RR

Metrics:

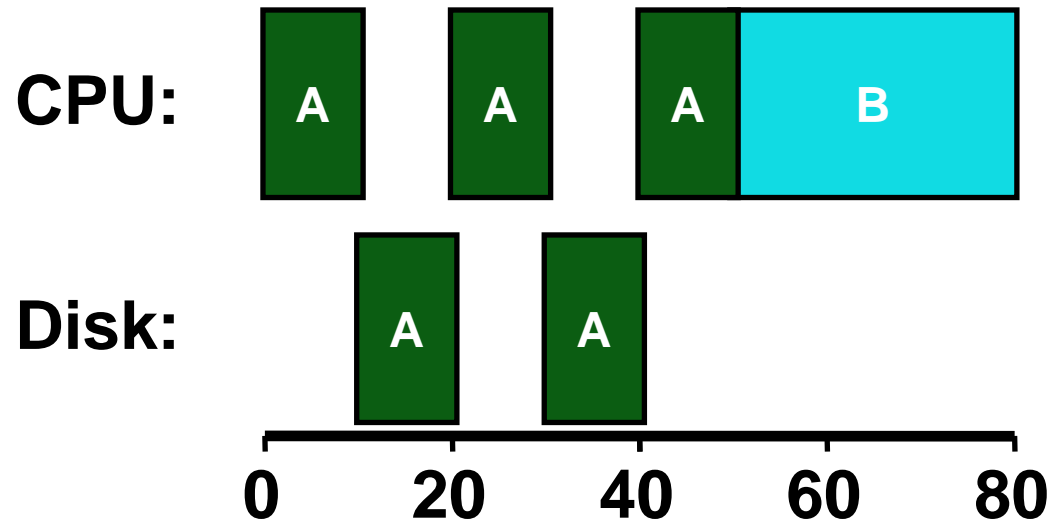
turnaround_time

response_time

Workload Assumptions

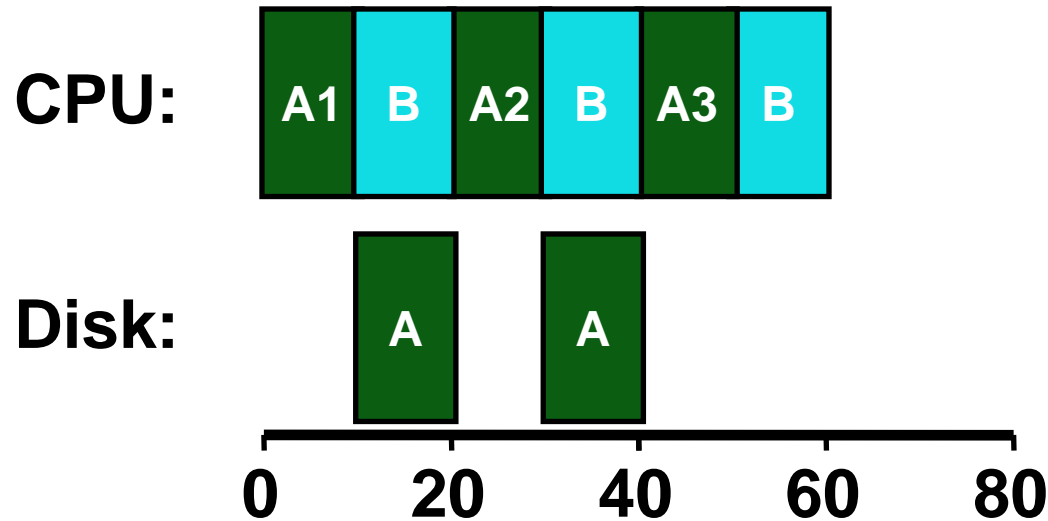
- ~~1. Each job runs for the same amount of time~~
- ~~2. All jobs arrive at the same time~~
- ~~3. All jobs only use the CPU (no I/O)~~
4. The run-time of each job is known

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** (sub-jobs)
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF,
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 → 

“Multi-level”

Q2 → 

How do we set priorities?

Q1

Q0 →  → 

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

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

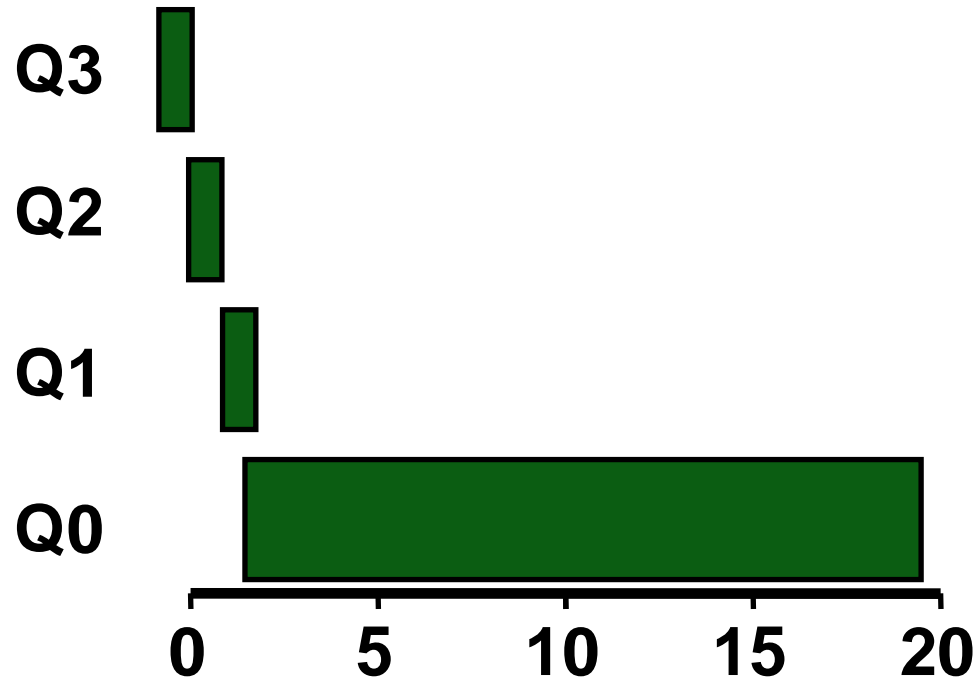
More rules:

Rule 3: Processes start at top priority

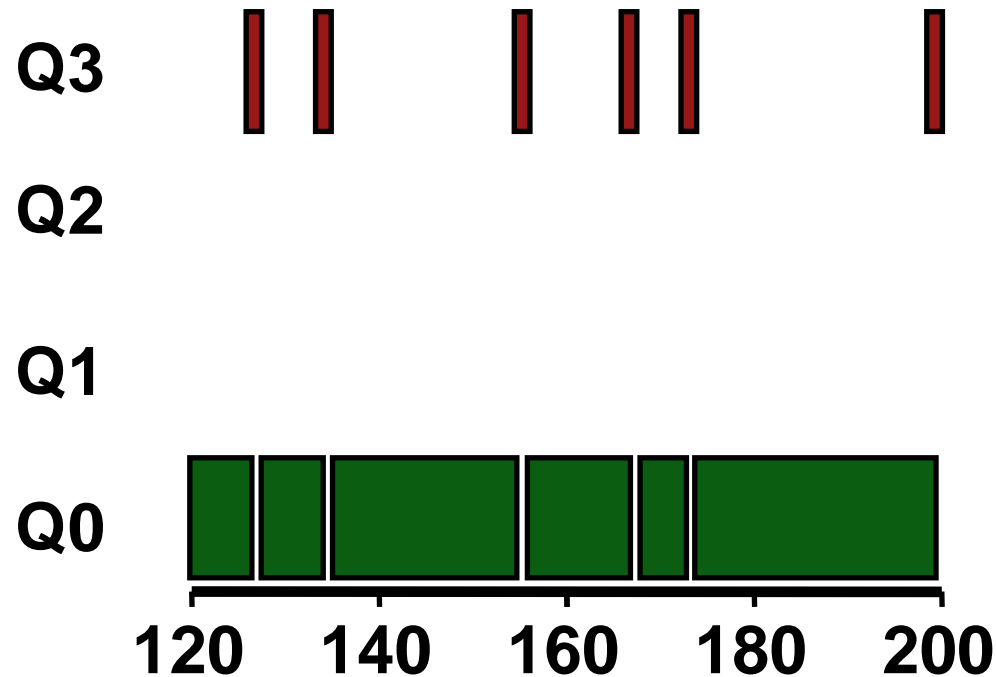
Rule 4: If job uses whole slice, demote process,
else leave at same level

(longer time slices at lower priorities)

One Long Job (Example)

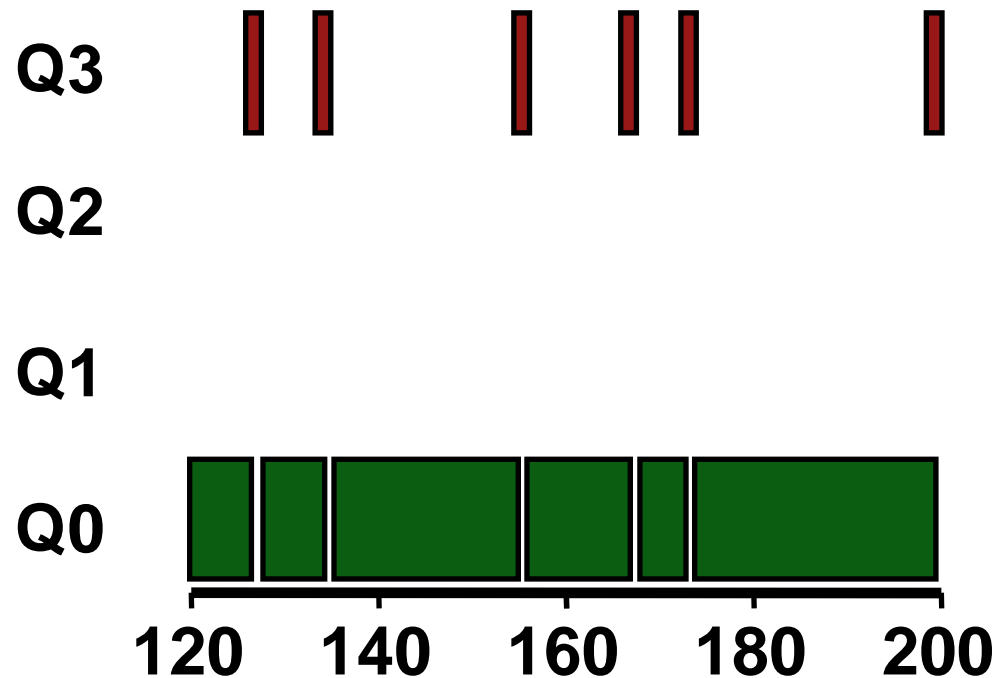


An Interactive Process Joins



Interactive process never uses entire time slice, so never demoted

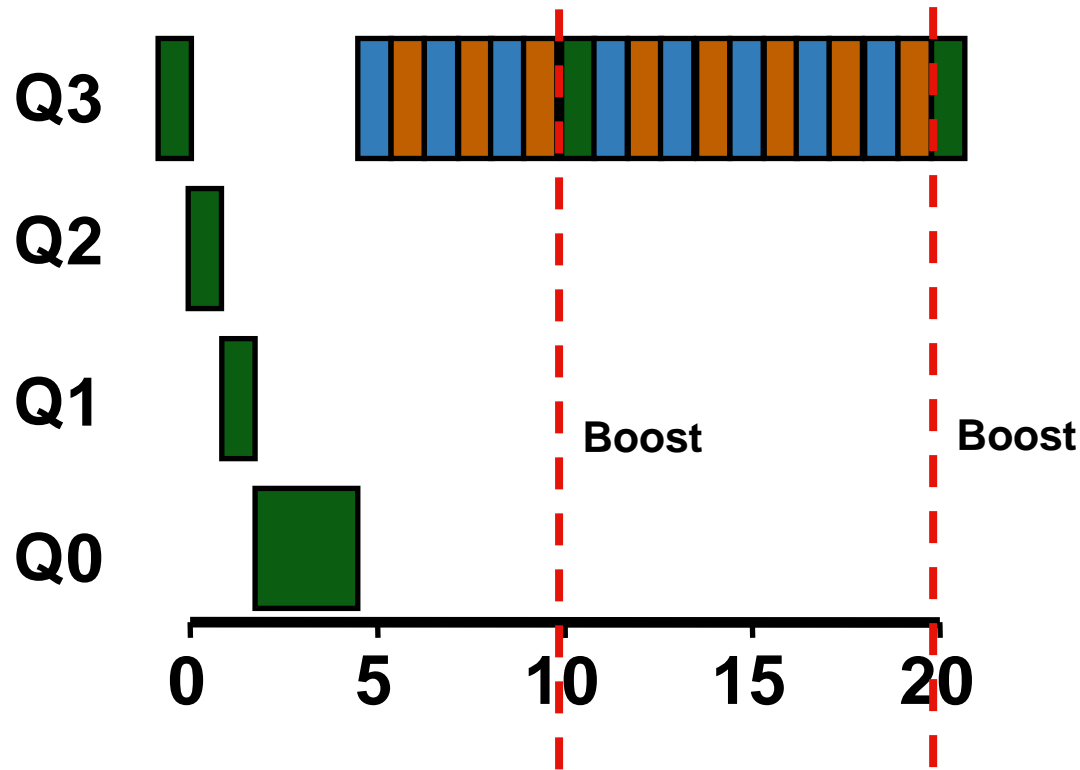
Problems with MLFQ?



Problems

- Unforgiving + **starvation**
- Gaming the system

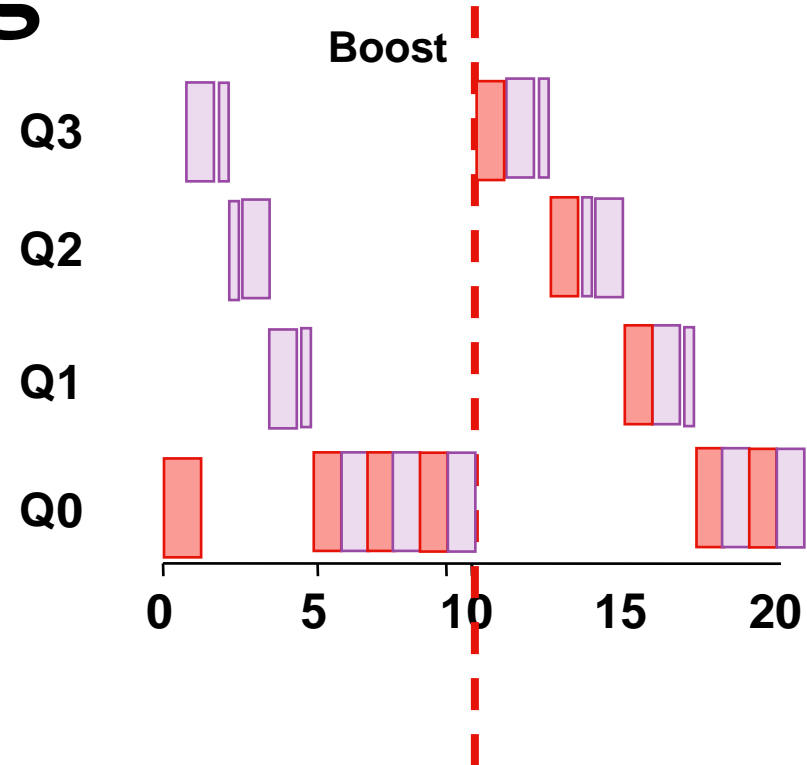
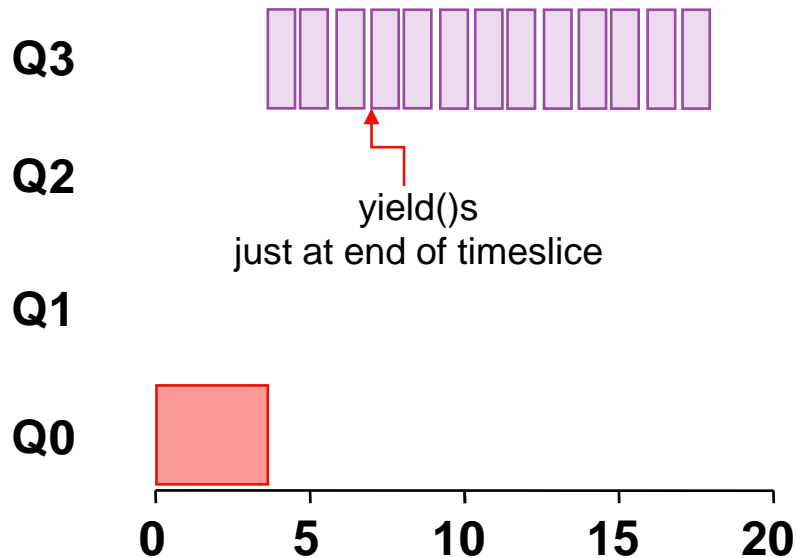
Prevent Starvation



Problem: Low priority job may never get scheduled
and processes may switch between CPU and I/O phases

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

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 example

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

Sum of tickets of all
ready processes

```
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

