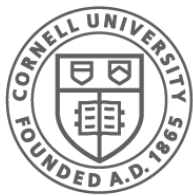


# CPU Scheduling

(Chapters 7-11)

CS 4410  
Operating Systems



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George,  
F.B. Schneider, E.G. Sirer, R. Van Renesse]

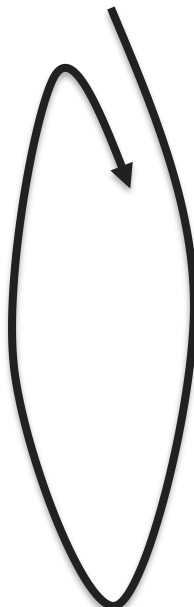
# Separating Mechanism and Policy

In this case:

- mechanism:
  - context switch between processes
- policy:
  - scheduling: which process to run next

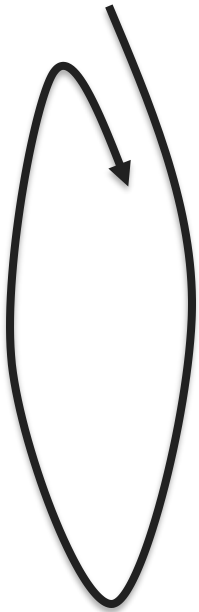
***An important principle in systems design***

# Kernel Operation (conceptual, simplified)

1. Initialize devices
  2. Initialize “first process”
  3. `while (TRUE) {`
    - while device interrupts pending
      - handle device interrupts
    - while system calls pending
      - handle system calls
    - **if run queue is non-empty**
      - **select process and switch to it**
    - otherwise
      - wait for device interrupt`}`
- 

# Kernel Operation (conceptual, simplified)

1. Initialize devices
2. Initialize “first process”
3. `while (TRUE) {`
  - while device interrupts pending
    - handle device interrupts
  - while system calls pending
    - handle system calls
  - **if run queue is non-empty**
    - **select process and switch to it**
  - otherwise
    - wait for device interrupt`}`



# The Problem

You're the cook at State Street Diner

- customers continuously enter and place orders 24 hours a day
- dishes take varying amounts to prepare

What is your *goal*?

- minimize average turnaround time?
- minimize maximum turnaround time?

Which *strategy* achieves your goal?

# Different goals

What if instead you are:

- the owner of an expensive container ship and have cargo across the world
- the head nurse managing the waiting room of the emergency room
- a student who has to do homework in various classes, hang out with other students, eat, and occasionally sleep

# Schedulers in the OS

- **CPU Scheduler** selects a process to run from the run queue
- **Disk Scheduler** selects next read/write operation
- **Network Scheduler** selects next packet to send or process
- **Page Replacement Scheduler** selects page to evict

Today we'll focus on **CPU Scheduling**

# Process Model

Processes switch between CPU & I/O bursts

CPU-bound processes: Long CPU bursts

matrix  
multiply



I/O-bound processes: Short CPU bursts

PowerPoint



We will call the CPU bursts “jobs”  
(aka *tasks*)



# Process Model

Processes switch between CPU & I/O bursts

CPU-bound processes: Long CPU bursts

matrix  
multiply



I/O-bound processes: Short CPU bursts

PowerPoint



Problems:

- When and how long are the jobs?
- Processes can change over time

# Job duration Prediction

- Based on the durations of the past jobs
- Use past as a predictor of the future

**No need to remember entire past history!**

Use *exponential moving average* (aka *low pass filter*):

$t_n$  actual duration of  $n^{\text{th}}$  job

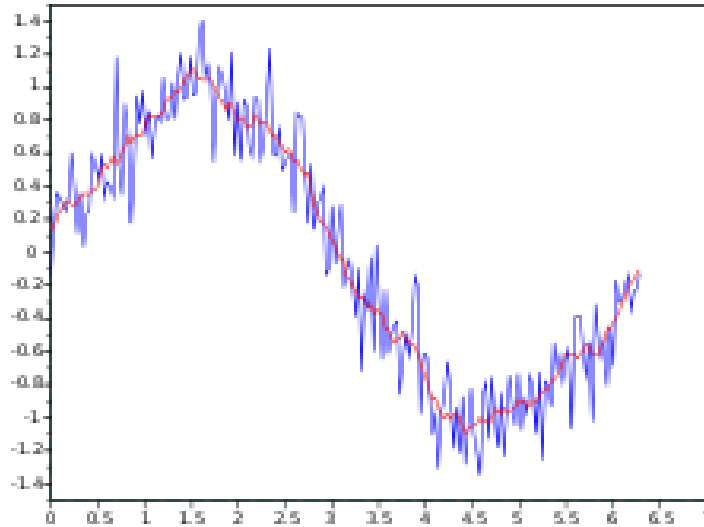
$\tau_n$  predicted duration of  $n^{\text{th}}$  job

$\tau_{n+1}$  predicted duration of  $(n+1)^{\text{th}}$  job

$$\tau_{n+1} = \alpha \tau_n + (1 - \alpha) t_n$$

$0 \leq \alpha \leq 1$ ,  $\alpha$  determines weight placed on past behavior

# EMA examples



# Job Characteristics

## **Job Arrival**

- When the job was first submitted

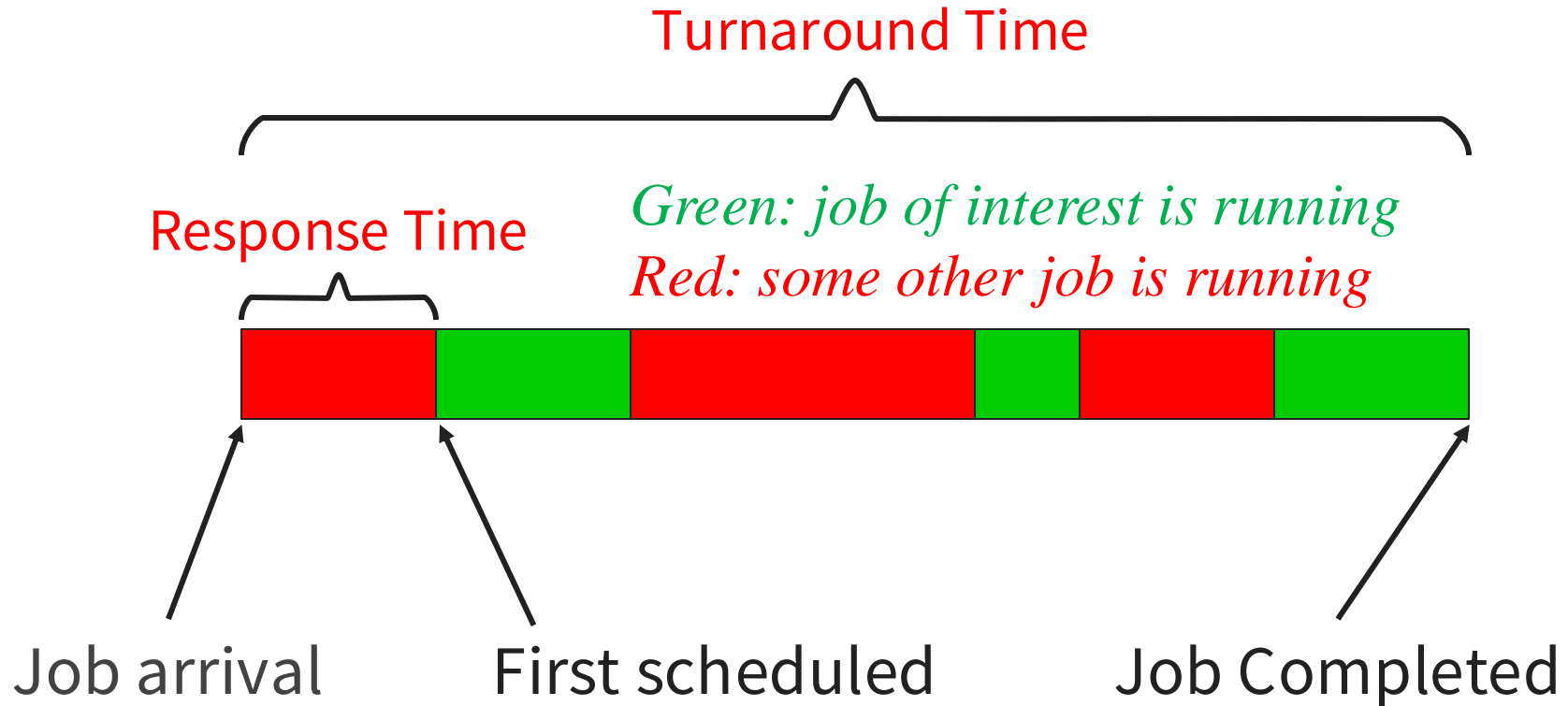
## **Job Execution time**

- Time needed to run the job without contention

## **Job Deadline**

- When the job must have completed. Think videos, car brakes, etc.

# Important Metrics of Scheduling



- *Execution Time*: sum of green periods
- *Waiting Time*: sum of red periods
- *Turnaround Time*: sum of both

# Performance Terminology

**Turnaround time:** How long?

- User-perceived time to complete some job

**Response time:** When does it start?

- User-perceived time before first output

**Waiting Time:** How much thumb-twiddling?

- Time on the run queue but not running

# More Performance Terminology

**Predictability:** How consistent?

- Low variance in turnaround time for repeated jobs

**Overhead:** How much useless work?

- Time lost due to switching between jobs

**Fairness:** How equal is performance?

- Equality in the resources given to each job

**Starvation:** How bad can it get?

- The lack of progress for one job, due to resources given to higher priority jobs

# The Perfect Scheduler

- Minimizes **response time** for each job
- Minimizes **turnaround time** for each job
- Provides **predictable performance**
- Maximizes **utilization**: keeps CPU and devices busy
- Is **work-conserving**
  - if there is a job that wants to run, there is a job running
- Meets all **deadlines**
- Is **starvation-free**: no job starves
- Is **envy-free**:
  - no job wants to switch its schedule with another job
- Has **zero overhead**

No such scheduler exists! 😞



# When does scheduler run?

## Non-preemptive

Job runs until it voluntarily yields CPU:

- process needs to wait (e.g., I/O or lock())
- process explicitly *yields*
- process terminates

## Preemptive

All of the above, plus:

- Timer and other interrupts
  - When jobs cannot be trusted to yield explicitly
- Incurs additional overhead

# What is the context switch overhead?

- Cost of saving registers
- Plus cost of scheduler determining the next process to run
- Plus cost of restoring registers

In addition, various caches may need to be flushed (L1, L2, L3, TLB, ...)

# Basic scheduling algorithms:

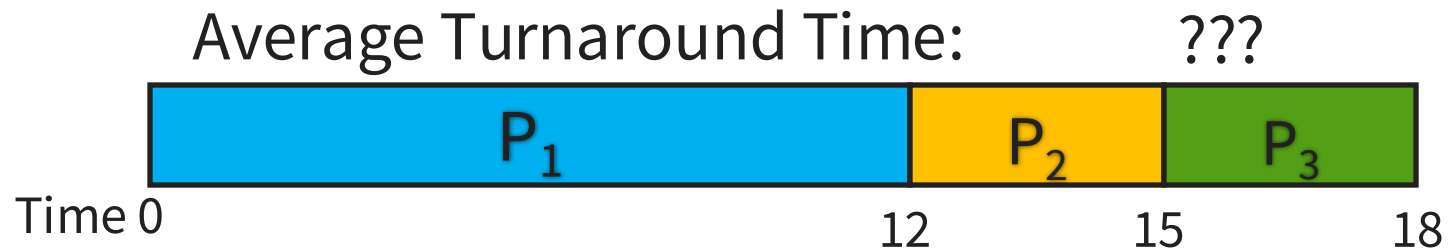
- First In First Out (FIFO)
  - aka First Come First Served (FCFS)
- Shortest Job First (SJF)
- Earliest Deadline First (EDF)
- Round Robin (RR)
- Shortest Remaining Time First (SRTF)

# First In First Out (FIFO)

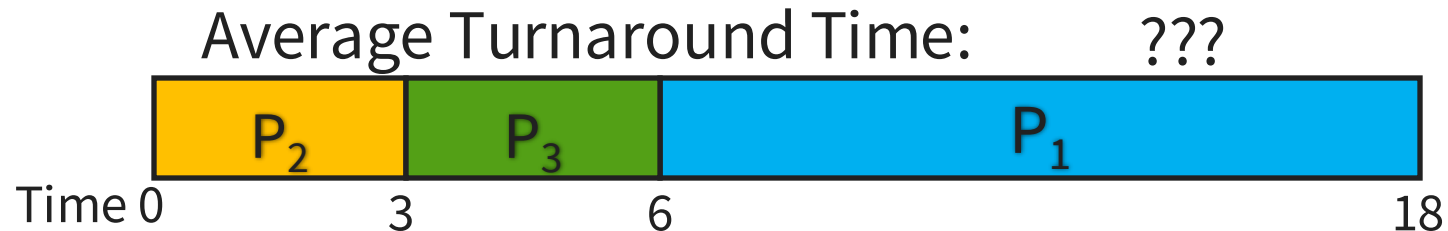
Jobs  $P_1$ ,  $P_2$ ,  $P_3$  with execution time 12, 3, 3

All arrive at the same time (so can be scheduled in any order)

Scenario 1: schedule order  $P_1$ ,  $P_2$ ,  $P_3$



Scenario 2: schedule order  $P_2$ ,  $P_3$ ,  $P_1$



# First In First Out (FIFO)

Jobs  $P_1, P_2, P_3$  with execution time 12, 3, 3

All arrive at the same time (so can be scheduled in any order)

Scenario 1: schedule order  $P_1, P_2, P_3$

Average Turnaround Time:  $(12+15+18)/3 = 15$



Scenario 2: schedule order  $P_2, P_3, P_1$

Average Turnaround Time: ???



# First In First Out (FIFO)

Jobs  $P_1, P_2, P_3$  with execution time 12, 3, 3

All arrive at the same time (so can be scheduled in any order)

Scenario 1: schedule order  $P_1, P_2, P_3$

Average Turnaround Time:  $(12+15+18)/3 = 15$



Scenario 2: schedule order  $P_2, P_3, P_1$

Average Turnaround Time:  $(3+6+18)/3 = 9$



# FIFO Roundup



- + Simple
- + Low-overhead
- + No Starvation



- Average turnaround time very sensitive to schedule order



- Not responsive to interactive jobs

*How to minimize average  
turnaround time?*

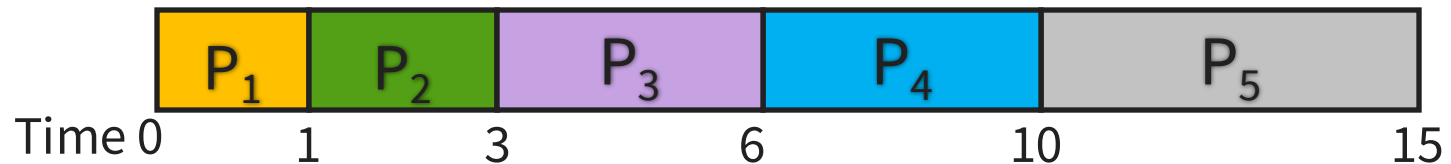


# Shortest Job First (SJF)

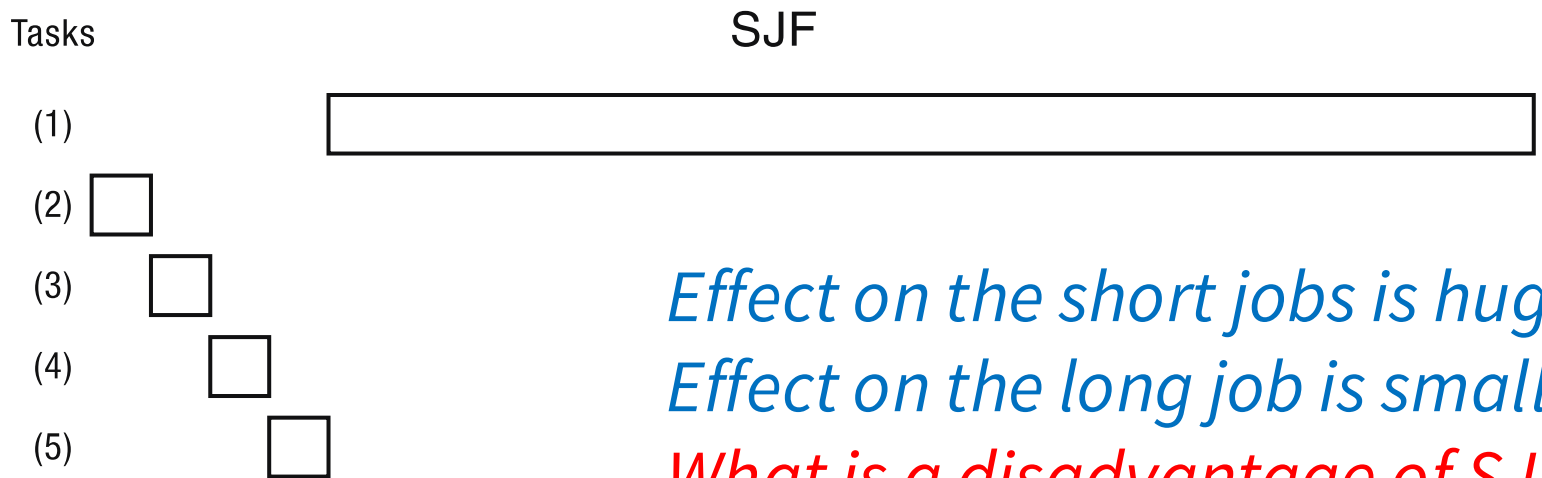
Schedule in order of execution time

Scenario : each job takes as long as its number

Average Turnaround Time:  $(1+3+6+10+15)/5 = 7$



# FIFO vs. SJF



*Effect on the short jobs is huge.  
Effect on the long job is small.  
What is a disadvantage of SJF?*

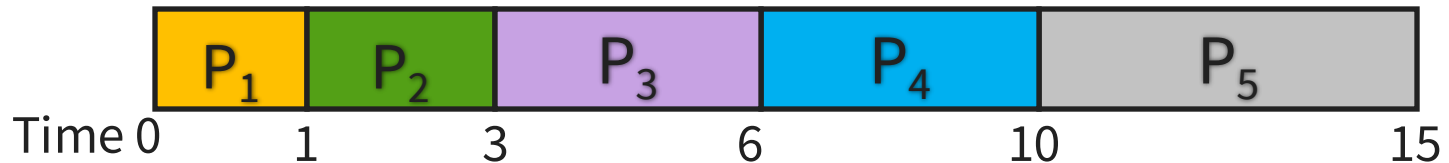
Time

# Shortest Job First (SJF)

Schedule in order of execution time

Scenario : each job takes as long as its number

Average Turnaround Time:  $(1+3+6+10+15)/5 = 7$



*Would another schedule improve avg turnaround time?*

# Informal proof of optimal turnaround time

- Let  $S$  be a schedule of a set of jobs
- Let  $j_1$  and  $j_2$  be two neighboring jobs in  $S$  so that  $j_1.\text{exe-time} > j_2.\text{exe-time}$
- Let  $S'$  be  $S$  with  $j_1$  and  $j_2$  switched around
  - *$S'$  has lower average turnaround time*
    - *because  $j_1$  will have the same turnaround time as  $j_2$  had before the switch while the turnaround time of  $j_2$  will be less than the one  $j_1$  had*
- Repeat until sorted (i.e., bubblesort)
- Resulting schedule is SJF

# SJF Roundup



+ Optimal average  
turnaround time



– Pessimial variance in  
turnaround time  
– Needs estimate of  
execution time



– Can starve long jobs

# Earliest Deadline First (EDF)

- Schedule in order of earliest deadline
- *If a schedule exists that meets all deadlines, EDF will generate such a schedule!*
- does not even need to know the execution times of the jobs

Why is that?

# Informal proof

- Let  $S$  be a schedule of a set of jobs that meets all deadlines
- Let  $j_1$  and  $j_2$  be two neighboring jobs in  $S$  so that  $j_1.\text{deadline} > j_2.\text{deadline}$
- Let  $S'$  be  $S$  with  $j_1$  and  $j_2$  switched
  - *$S'$  also meets all deadlines*
- Repeat until sorted (i.e., bubblesort)
  - Resulting schedule is EDF

# EDF Roundup



- + Meets deadlines if possible
- + Free of starvation



- Does not optimize other metrics



- Cannot decide when to run jobs without deadlines



# Generalization: Priority Scheduling

- Assign a number to each job and schedule jobs in (increasing) order
  - Can implement any scheduling policy
    - e.g., reduces to SJF if  $\tau_n$  is used as priority
- ↑  
estimate of execution time

# Priority Inversion

- Problem: some high priority process is waiting for some low priority process
  - maybe low priority process has a lock on some resource
- Solution: High priority process (needing lock) temporarily donates priority to lower priority process (with lock)

“Priority Inheritance”

# Avoiding Starvation

- Two approaches:
  1. improve job's priority with time (*aging*)
    - FIFO and EDF do this implicitly
  2. select jobs *randomly* weighted by priority

# Round Robin (RR)

- Each job allowed to run for a *quantum*
  - quantum = some configured period of time
  - Improves response time!
- Context is switched (*at the latest*) at the end of the quantum
  - **Preemption!!**
- Next job is the one on the run queue that hasn't run for the longest amount of time

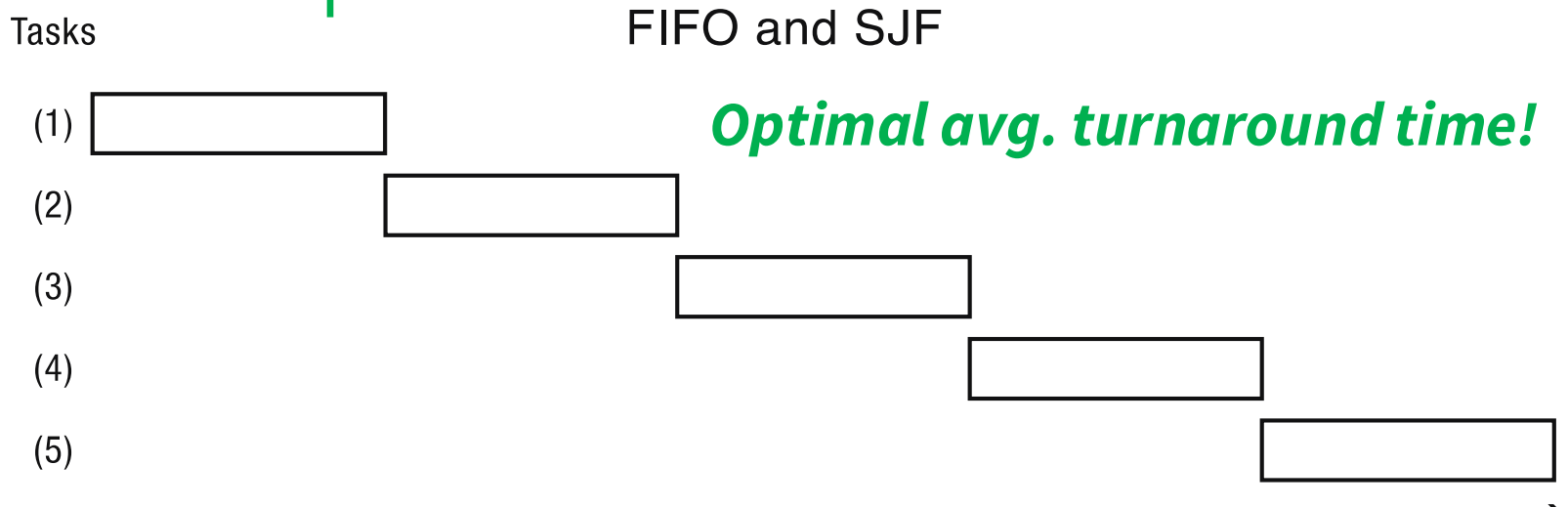
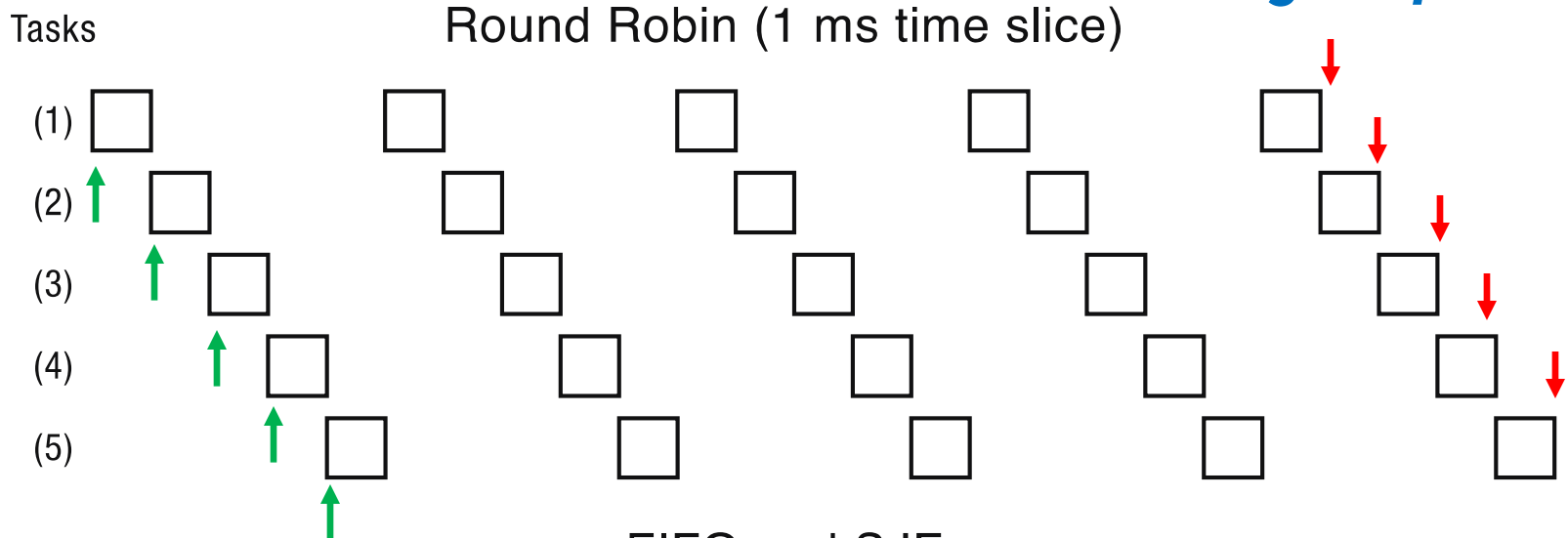
What is a good quantum size?

- Too long, and it morphs into FIFO
- Too short, and time is wasted on context switching
- Typical quantum: about 100X cost of context switch (~100ms vs.  $\ll 1$  ms)

# Round Robin vs. FIFO

Jobs of same length that start at same time

*Avg. turnaround?  
Avg. response?*

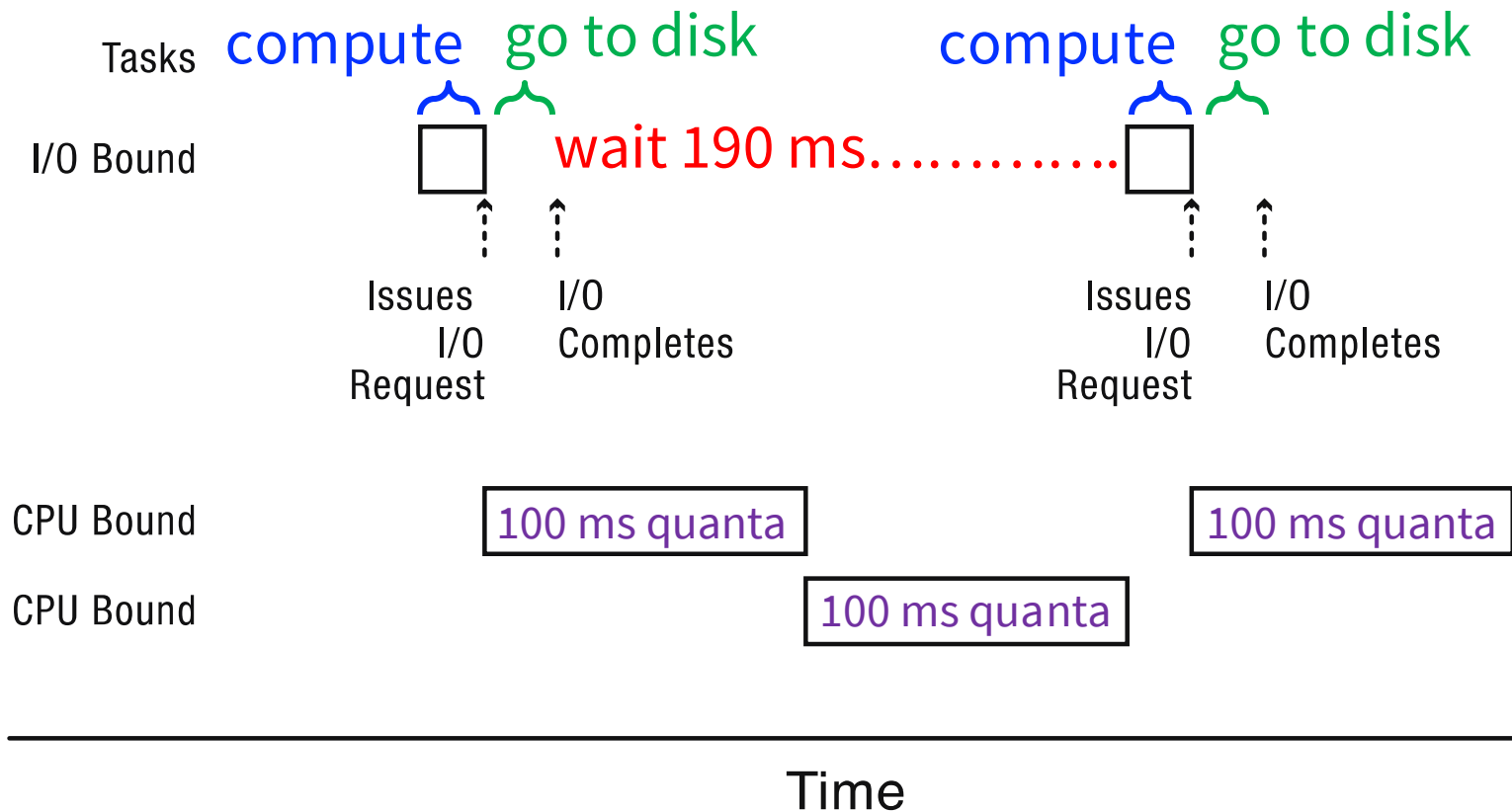


# More Problems with Round Robin

Mixture of one I/O Bound processes + two CPU Bound Processes

I/O bound: compute, go to disk, repeat

→ RR (with long quanta) doesn't seem so fair after all....



# RR Roundup



- + No starvation
- + Can reduce response time



- Context switch overhead
- Mix of I/O and CPU bound



- bad avg. turnaround time for equal length jobs

# Shortest Remaining Time First (SRTF)

- SJF + Preemption
- At end of each quantum, scheduler selects the job with the least remaining time to run next
  - Often this means the same job can run until completion, **avoiding context switch overhead**
  - But new short jobs still see an **improved response time**



# SRTF Roundup



- + Good for response time and turnaround time of I/O-bound processes
- + Low context switch overhead



- Needs estimate of execution time of each job

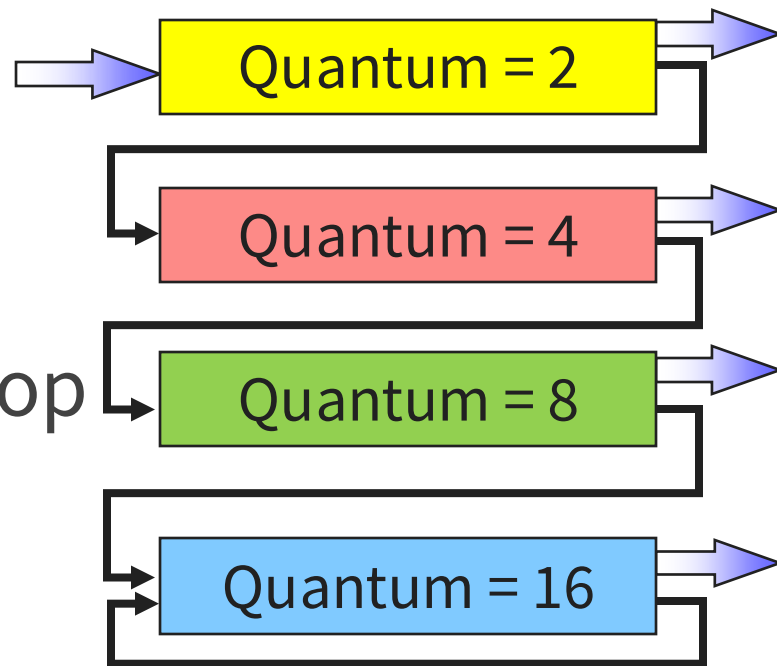


- Suffers from starvation

# Multi-Level Feedback Queue (MLFQ)

- Multiple levels of RR queue
- Jobs start at the top
  - Use quantum? **move down**
  - Don't? **Stay where you are**
- Periodically all jobs back to top
- *Approximates SRTF*

Highest priority



Lowest priority

*Used by MacOSX,  
Windows, some  
versions of Linux, ...*

Need parameters for:

- Number of queues
- Quantum length per queue
- Time to move jobs back up

# “Completely Fair Scheduler” (CFS)

Define “Spent Execution Time” (SET) to be the amount of time that a process (not job) has been executing.

Let  $\Delta$  be some time constant (typically, 20-50ms or so).

1. Scheduler selects process with lowest SET
2. Let  $N$  be the number of processes on the run queue
3. Process runs for up to  $\Delta/N$  time (there is a minimum value)
4. Update SET of the process
5. If it used up the quantum, reinsert into the run queue
6. Repeat

If a process is new or it sleeps and wakes up, then its new SET is the maximum of its old SET and the minimum of the SETs of the processes on the run queue

*Used by most  
versions of Linux, ...*

# Gaming the Scheduler

Processes can cheat by

- splitting app into multiple processes
- periodically terminating and restarting
- yielding CPU just before quantum expires
- ...

# Multi-core Scheduling

## Desirables:

- **Balance load**
  - each job should get approximately the same amount of CPU, no matter what core it runs on
- **Scheduling affinity**
  - avoid moving processes between cores
    - avoid wasting cache content (L1, TLB, etc.)
- **Avoid access contention** on run queue
  - locking of run queue data structure
    - avoid for scalability

# Multi-core Scheduling Options

	Single Shared Queue	One Queue Per Core
Balance Load	✓	✗
Scheduling Affinity	✗	✓
Avoid Contention	✗	✓

# Multi-core Scheduling Options

	Single Shared Queue	One Queue Per Core
Balance Load	✓	✓
Scheduling Affinity	✗	✓
Avoid Contention	✗	✓

## Work stealing:

- Periodically balance the load between the cores
- Creates some loss of cache efficiency
- Creates some, but not much contention