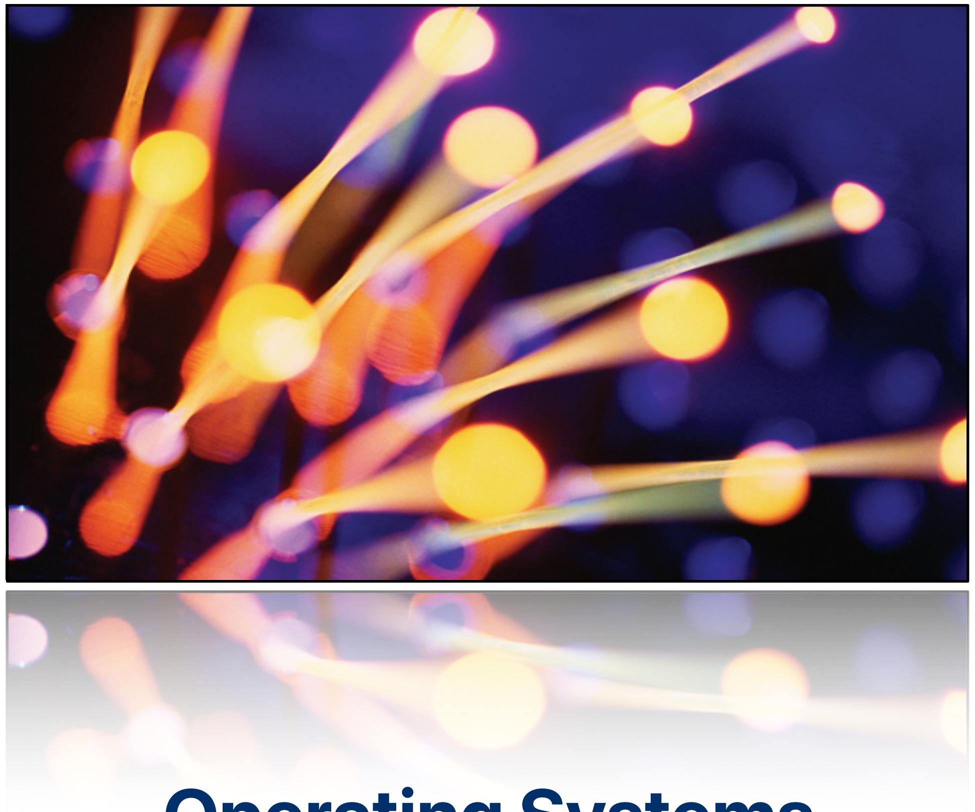
# Proportional-Share Scheduling



**Operating Systems**

Baochun Li

University of Toronto

# Revisiting our design objectives

Turnaround time: total time needed to complete a job

Response Time: the time from when the job arrives to the first time it is scheduled

Fairness: give each job its fair share — a certain percentage of CPU time

# Focusing on Fairness

**How can we design a scheduler to share the CPU in a proportional manner?**

**What are the key mechanisms?**

- Lottery scheduling
- Stride scheduling

# Lottery Scheduling

# Design principle: Randomization

## Use tickets to represent the CPU share

## Hold a lottery every time slice

## If job A holds 75% of the tickets, B holds 25% —

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63 | 85 | 70 | 39 | 76 | 17 | 29 | 41 | 36 | 39 | 10 | 99 | 68 | 83 | 63 | 62 | 43 | 0 | 49 | 49 |
| A | | A | A | | A | A | A | A | A | A | | A | | A | A | A | A | A | A |
| | B | | | B | | | | | | | B | | B | | | | | | |

# It’s a great idea to randomize!

## Simple, lightweight, and fast!

Just need to generate random numbers

## Requires little state to be tracked

A deterministic scheduling algorithm may need to how much CPU each thread has received so far

Lottery scheduling only needs to know the total number of tickets

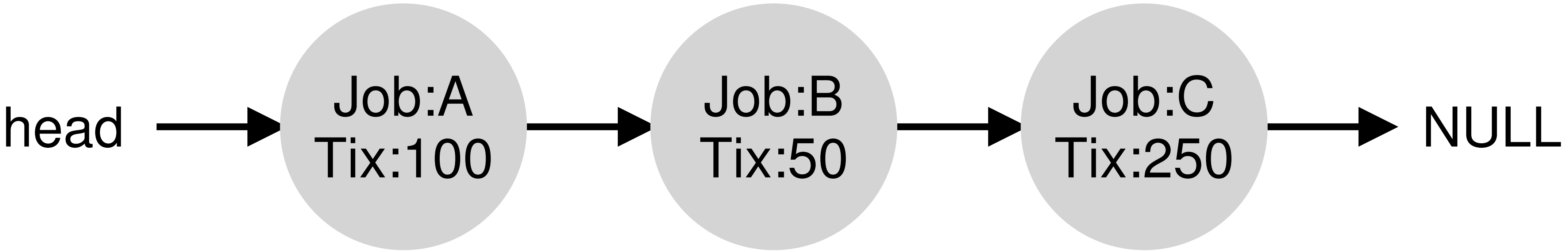
# Implementing lottery scheduling

Randomly generate a value "winner"

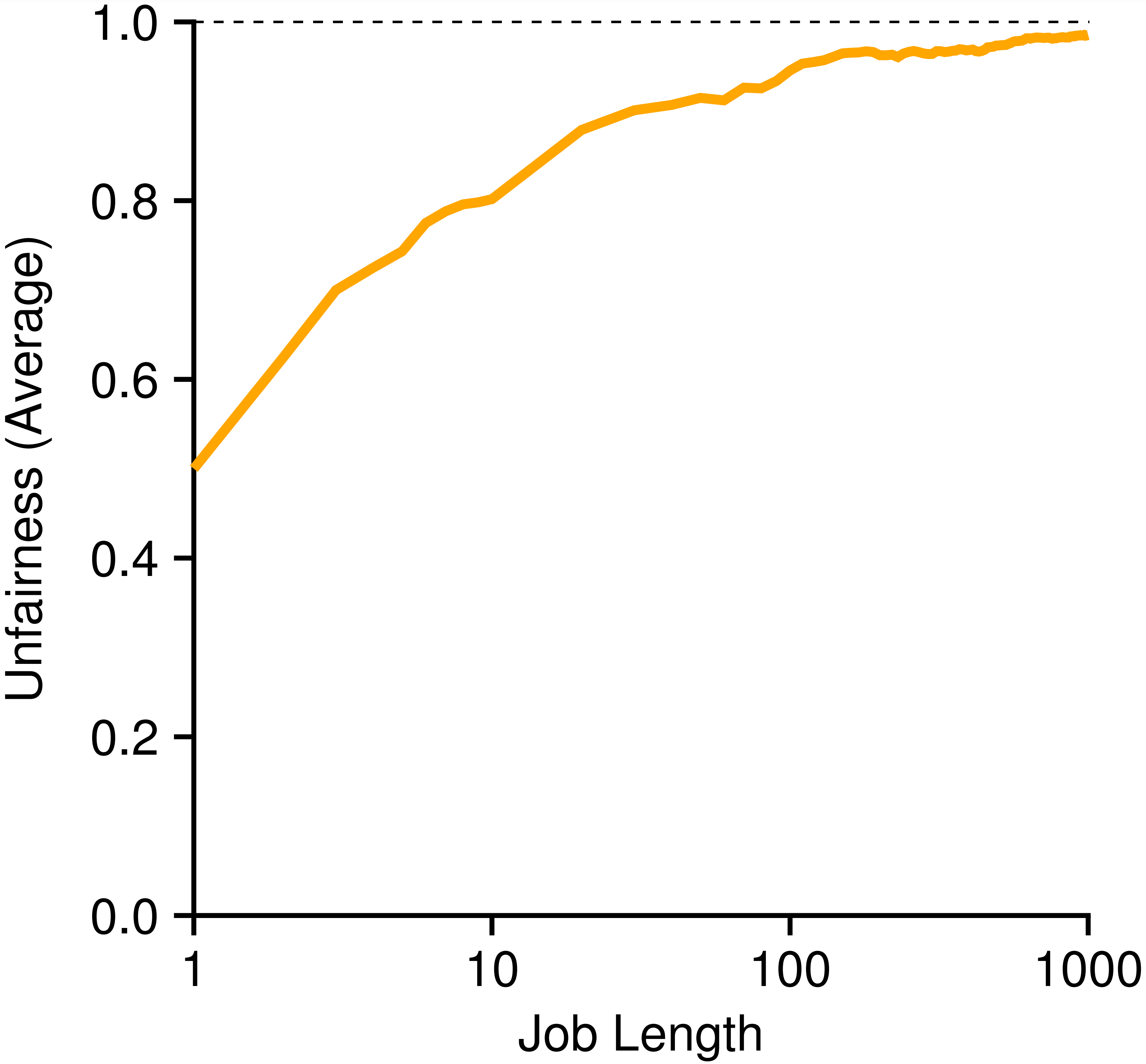
Walks the list of jobs

Adds each ticket value to "counter" until its value exceeds "winner"

- To make it more efficient, sort the list in decreasing order



# Unfairness



# Stride Scheduling

**Each job has a stride, which is inversely proportional to the number of tickets it has**

With jobs A, B, and C, with 100, 50, and 250 tickets, their stride values can be 100, 200, and 40

**Every time a job runs, increment a counter, called its pass value, by its stride**

**Select the job that has the lowest pass value to run**

```
current = remove_min(queue);
schedule(current);
current->pass += current->stride;
insert(queue, current);
```

# Example of Stride Scheduling

| Pass(A)<br>(stride=100) | Pass(B)<br>(stride=200) | Pass(C)<br>(stride=40) | Who Runs? |
|---|---|---|---|
| 0 | 0 | 0 | A |
| 100 | 0 | 0 | B |
| 100 | 200 | 0 | C |
| 100 | 200 | 40 | C |
| 100 | 200 | 80 | C |
| 100 | 200 | 120 | A |
| 200 | 200 | 120 | C |
| 200 | 200 | 160 | C |
| 200 | 200 | 200 | … |

# Why use lottery scheduling at all?

Stride scheduling is more precise, as lottery scheduling achieves the proportions probabilistically

But lottery scheduling has no global state!

Imagine a new job enters in the middle of our example

- What should its pass value be? 0?

# Completely Fair Scheduler: Linux 2.6.23

**wait_runtime is maintained for each thread, in nanoseconds: the amount of time the thread should now run on the CPU for it to become completely fair**

- when the thread finishes running, its runtime is deducted from wait_runtime
- wait_runtime accumulates when a thread sleeps

**fair_clock is maintained as the CPU time a thread would have fairly received**

**(fair_clock - wait_runtime) is used to sort threads in a tree**

- O(log n) insertion, O(1) to retrieve thread with (roughly) the lowest value to be scheduled

# What we've covered so far

## Three Easy Pieces, Chapter 9 (Scheduling: Proportional Share)