ECE 570/670

David Irwin

Lecture 11

Administrative Details

- Should be working on Assignment 1
  - Assignment 1 disk scheduler (1d)
    - How to use concurrency primitives
    - Due today
  - Assignment 1 thread library (1t)
    - How concurrency primitives actually work
  - **Midterm next Wednesday**
    - **Review next class**

- Will post example questions on calendar
  - Will go over answers in class
- Posted videos on Moodle for review (see Echo360 link)

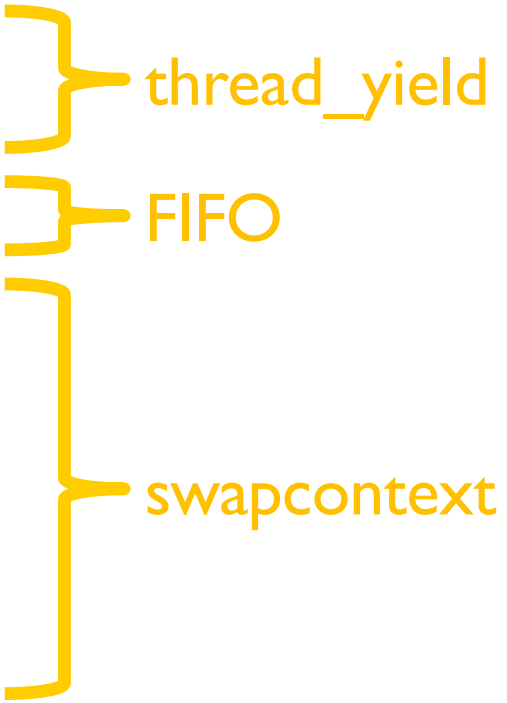
Administrative Details

- ECE670 project proposal due tomorrow
  - Upload to Moodle
  - A few groups have still not met with me

Switching threads

- What needs to happen to switch threads?

1. **Thread returns control to OS**
   - For example, via the "yield" call
2. **OS chooses next thread to run**
3. **OS saves state of current thread**
   - To its thread control block
4. **OS loads context of next thread**
   - From its thread control block
5. **Run the next thread**

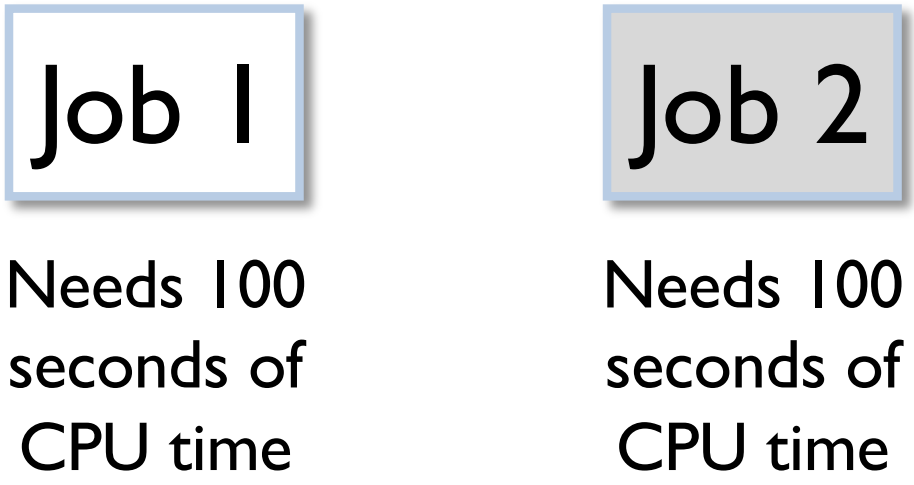


Scheduling goals

1. **Minimize average response time**
   - Elapsed time to do a job (what users care about)
   - Try to maximize idle time
     - Incoming jobs can then finish as fast as possible
2. **Maximize throughput**
   - Jobs per second (what admins care about)
   - Try to keep parts as busy as possible
   - Minimize wasted overhead (e.g. context switches)

Scheduling goals

3. **Fairness**
   - Share CPU among threads equitably
   - Key question: what does "fair" mean?



What does "fair" mean?

- How can we schedule these jobs?
  - Job 1 then Job 2
    - 1's response time = 100, 2's response time = 200
    - Average response time = 150
  - Alternate between 1 and 2
    - 1's response time = 2's response time = 200
    - Average response time = 200

Job 1

Job 2

Needs 100 seconds of CPU time

Needs 100 seconds of CPU time

Fairness

- First time thinking of OS as government
- Fairness can come at a cost
  - (in terms of average response time)
- Finding a balance can be hard
  - What if you have 1 big job, 1 small job?
  - Response time proportional to size?
  - Trade-offs come into play

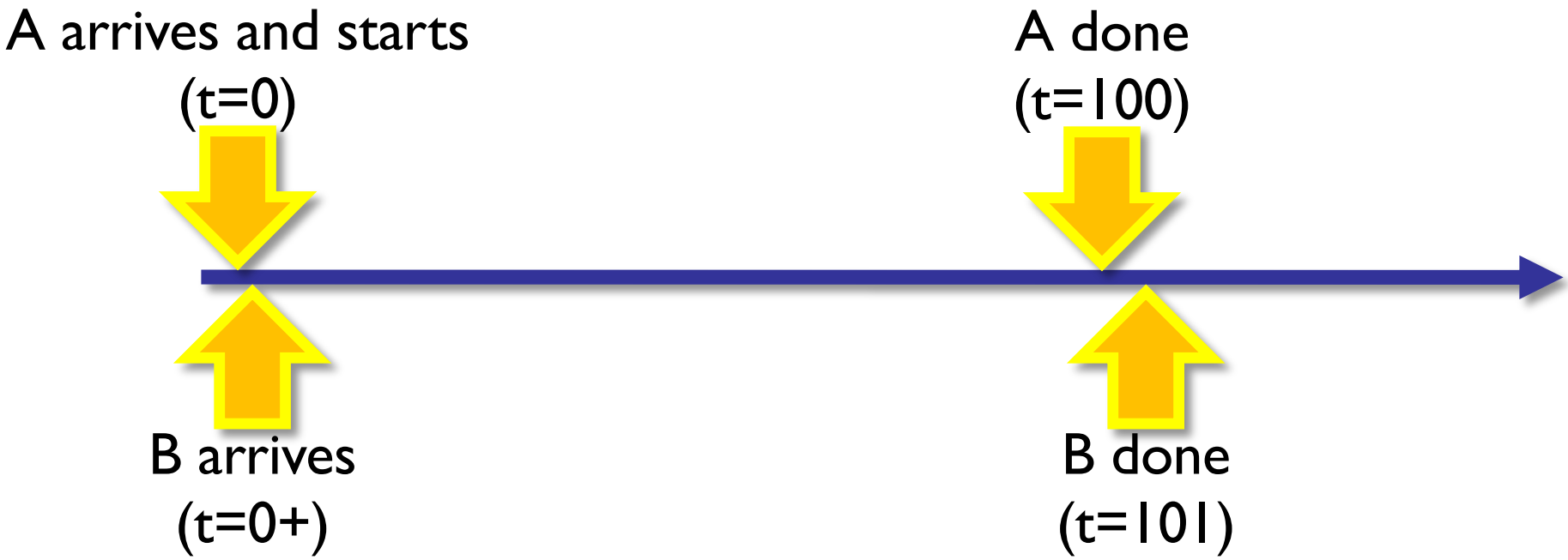
FCFS (first-come-first-served)

- FIFO ordering between jobs
- No pre-emption (run until done)
  - Run thread until it blocks or yields
  - No timer interrupts

- Essentially what you're doing in Project 1t

FCFS example

- Job A (100 seconds), Job B (1 second)



- Average response time = 100.5 seconds

FCFS

- Pros
  - Simplicity
- Cons?
  - Short jobs stuck behind long ones
  - Non-interactive
  - (for long CPU job, user can't type input)

Round-robin

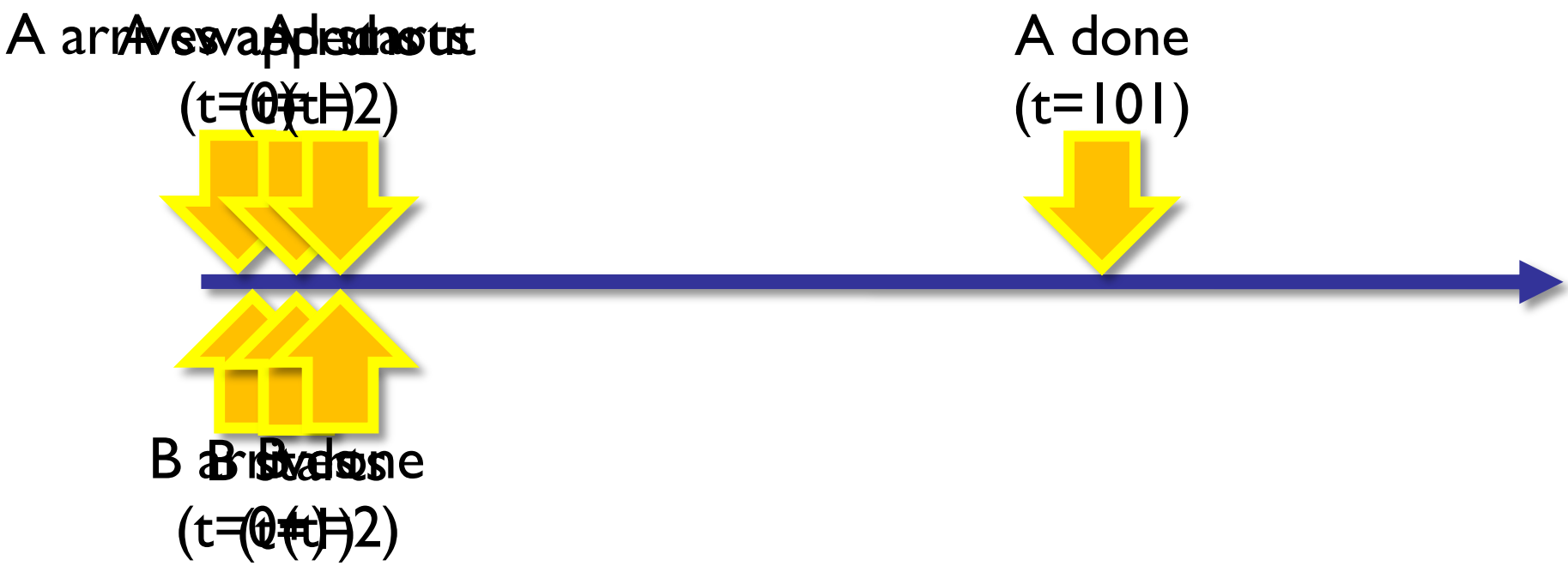
- Goal
  - Improve average response time for short jobs
- Solution to FCFS's problem
  - Add pre-emption!
  - Pre-empt CPU from long-running jobs
  - (timer interrupts)
- This is what most OSes do

Round-robin

- In what way is round robin fairer than FCFS?
  - Makes response times ~ job length
- In what way is FCFS fairer than round robin?
  - First to start will have better response time
- Like two children with a toy
  - Should they take turns?
    - "She's been playing with it for a long time."
  - Should first get toy until she's bored?
    - "I had it first."

Round-robin example

- Job A (100 seconds), Job B (1 second)

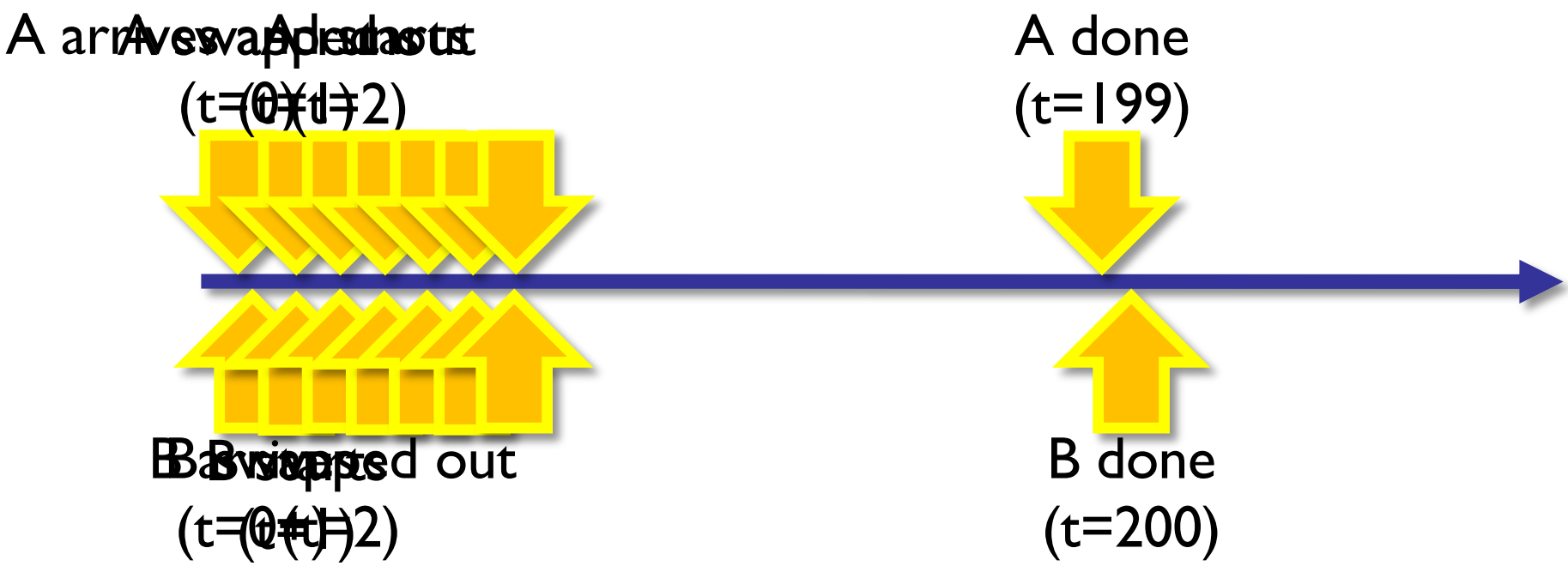


- Average response time = 51.5 seconds

Does round robin always provide lower response times than FCFS?

Round-robin example 2

- Job A (100 seconds), Job B (100 second)

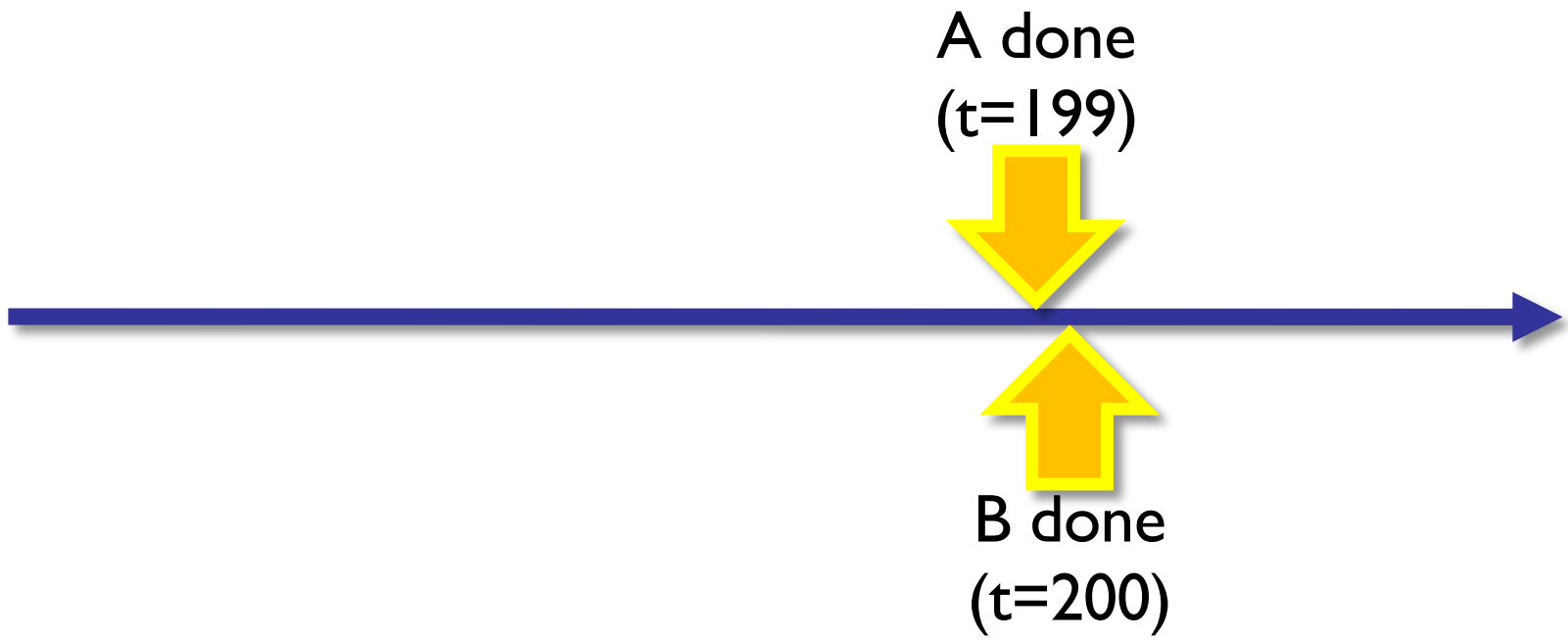


- Average response time = 199.5 seconds

Any hidden costs that we aren't counting? Context switches

Round-robin example 2

- Job A (100 seconds), Job B (100 second)

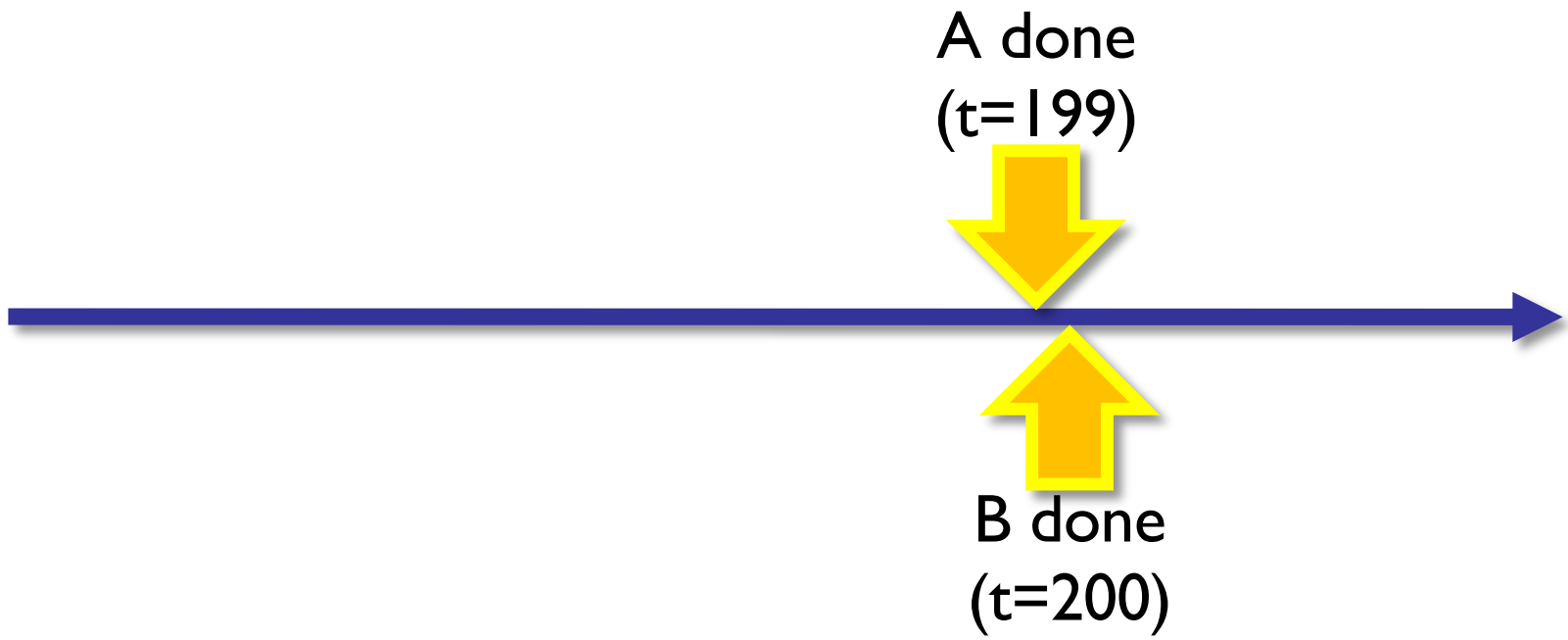


- Average response time = 199.5 seconds

What would FCFS' s avg response time be? 150 seconds

Round-robin example 2

- Job A (100 seconds), Job B (100 second)



- Average response time = 199.5 seconds

Which one is fairer? It depends …

Round-robin

- Pros
  - Good for interactive computing
  - Better than FCFS for mix of job lengths
- Cons
  - Less responsive for uniform job lengths
  - Hidden cost: context switch overhead
- How should we choose the time-slice?
  - Typically a compromise, e.g. 100 ms
  - Most threads give up CPU voluntarily much faster

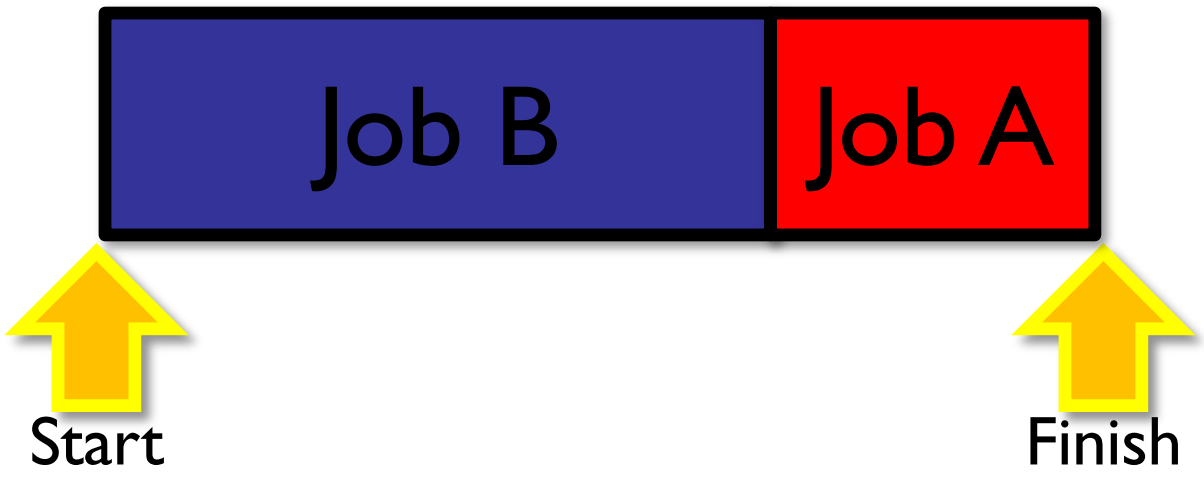
STCF and STCF-P

- Idea
  - Get the shortest jobs out of the way first
  - Improves short job times significantly
  - Little impact on big ones

1. **Shortest-Time-to-Completion-First**
   - Run whatever has the least work left before it finishes
2. **Shortest-Time-to-Completion-First (Pre-emp)**
   - If new job arrives with less work than current job has left
   - Pre-empt and run the new job

STCF is optimal

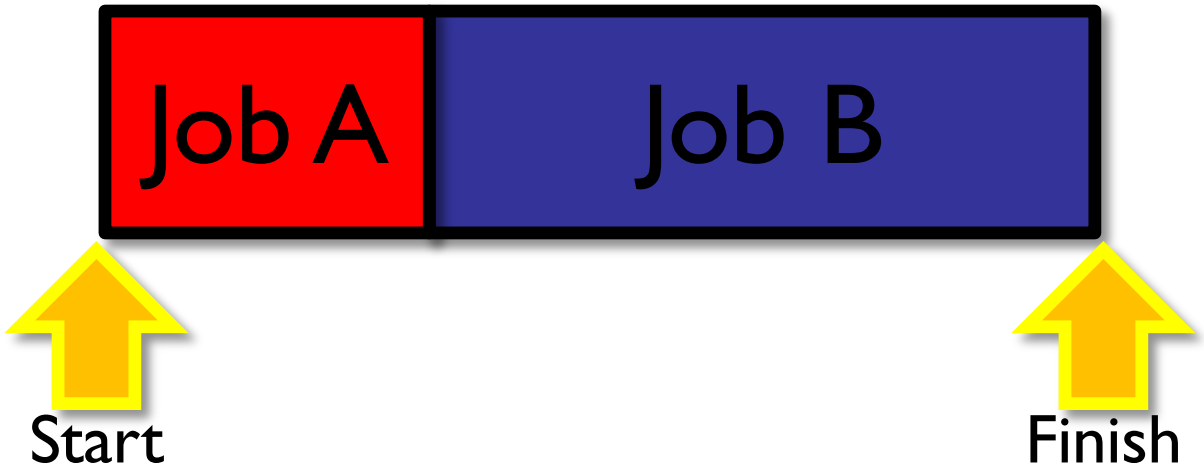
- (among non-pre-emptive policies)



What happened to total time to complete A and B?

STCF is optimal

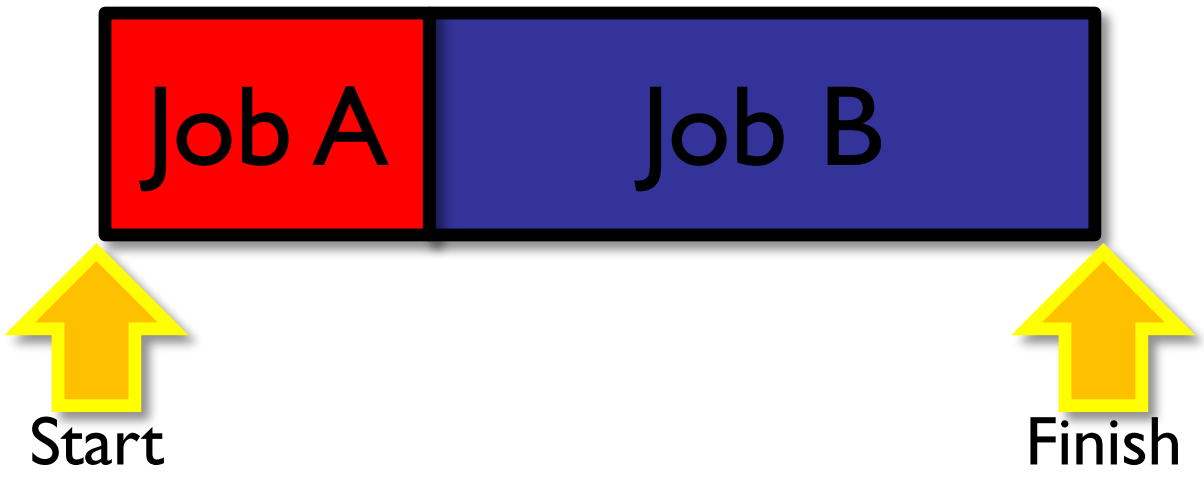
- (among non-pre-emptive policies)



What happened to the time to complete A?
What happened to the time to complete B?

STCF is optimal

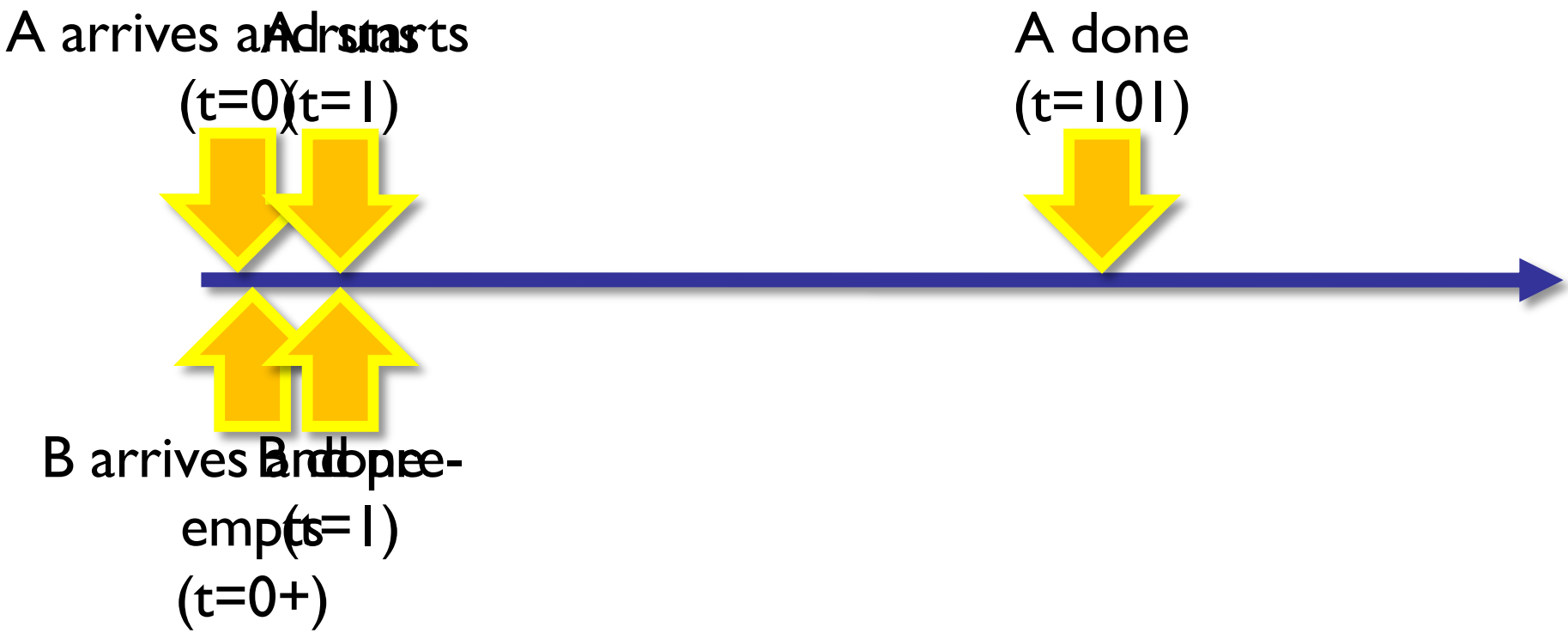
- (among non-pre-emptive policies)



What happened to the average completion time?

STCF-P is also optimal

- (among pre-emptive policies)
- Job A (100 seconds), Job B (1 second)



- Average response time = 51 seconds

STCF-P

- Pros
  - Optimal average response time
- Cons?
  - Can be unfair
  - What happens to long jobs if short jobs keep coming?
  - Legend from the olden days …
    - IBM 7094 was turned off in 1973
    - Found a "long-running" job from 1967 that hadn't been scheduled
  - Requires knowledge of the future

Knowledge of the future

- You will see this a lot.
- Examples?
  - Cache replacement (next reference)
- How do you know how much time jobs take?
  - Ask the user (what if they lie?)
  - Use the past as a predictor of the future

Grocery store scheduling

- How do grocery stores schedule?
- Kind of like FCFS
- Express lanes
  - Make it kind of like STCF
  - Allow short jobs to get through quickly
- STCF-P would probably be considered unfair

I/O

- What if a program does I/O too?

```
while (1) {
  do 1ms of CPU
  do 10ms of I/O
}
```

- To scheduler, is this a long job or a short job?
  - Short
  - Thread scheduler only cares about CPU time

Final example

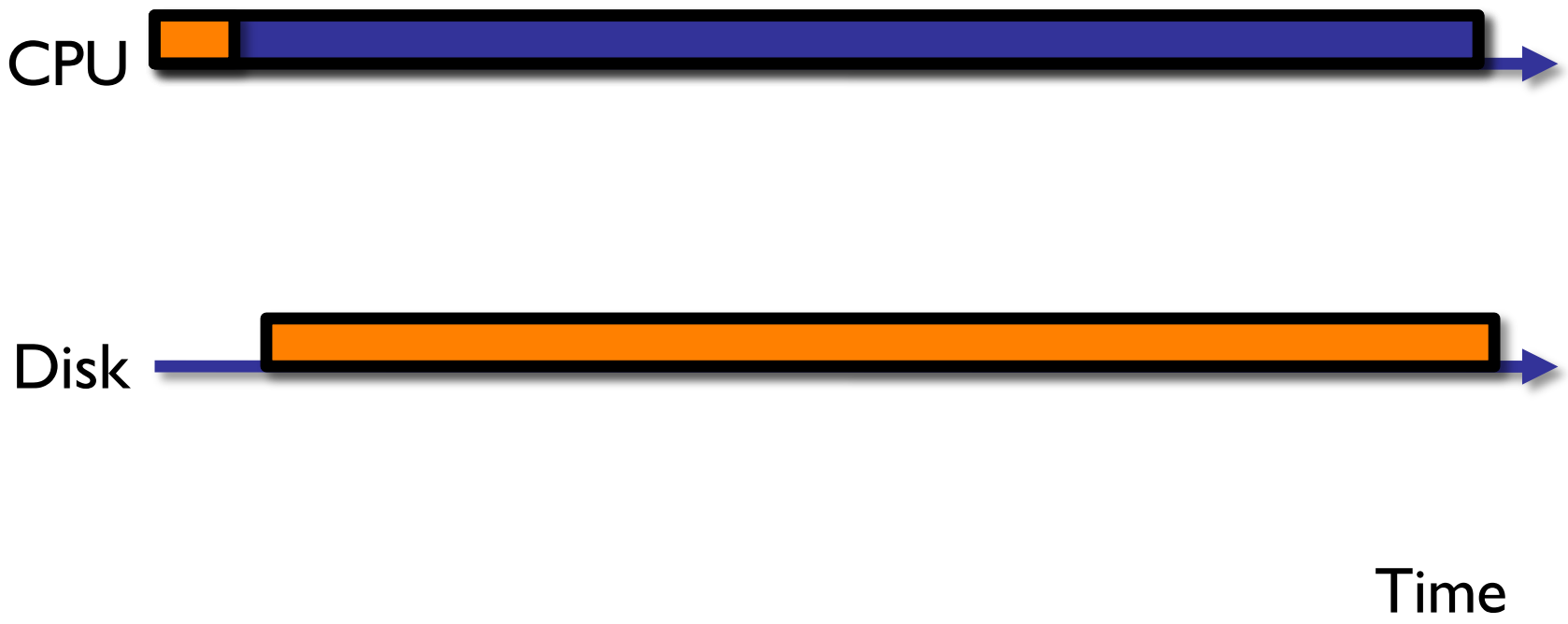
- Job A (CPU-bound)
  - No blocking for I/O, runs for 1000 seconds
- Job B (CPU-bound)
  - No blocking for I/O, runs for 1000 seconds
- Job C (I/O-bound)

```
while (1) {
  do 1ms of CPU
  do 10ms of I/O
}
```

Each job on its own

A: 100% CPU, 0% Disk B: 100% CPU, 0% Disk

C: 1/11 CPU, 10/11 Disk



Mixing jobs (FCFS)

A: 100% CPU, 0% Disk B: 100% CPU, 0% Disk

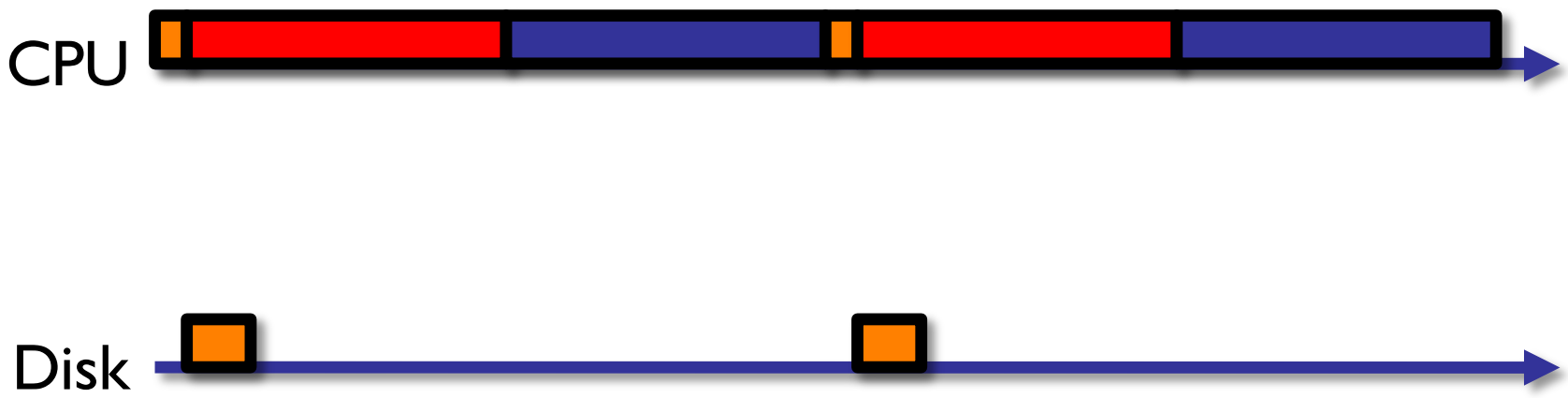
C: 1/11 CPU, 10/11 Disk

How well would FCFS work? Not well.

Mixing jobs (RR with 100ms)

A: 100% CPU, 0% Disk B: 100% CPU, 0% Disk

C: 1/11 CPU, 10/11 Disk

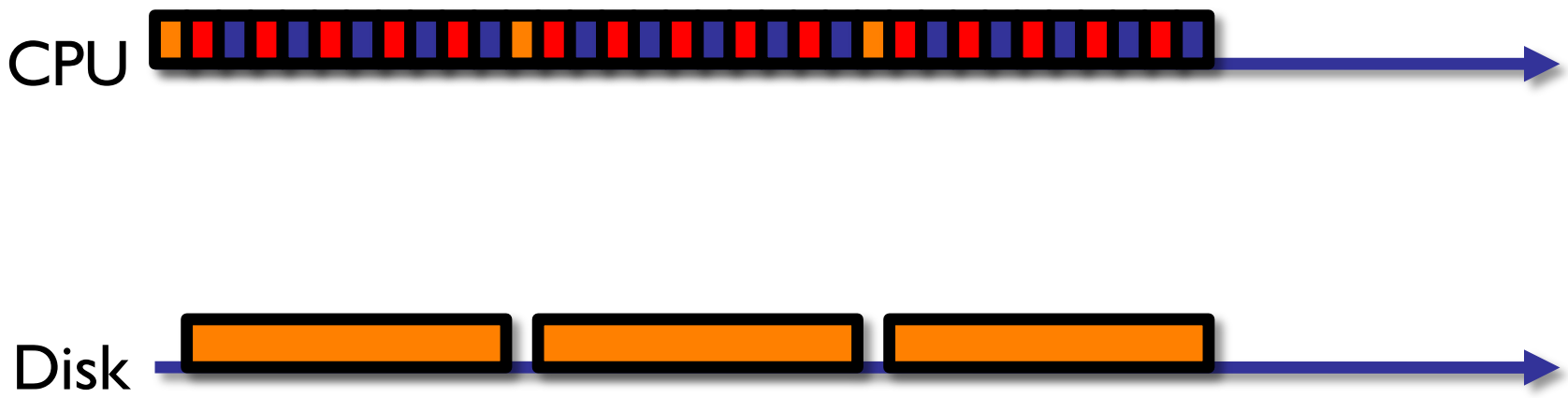


How well would RR with 100 ms slice work?

Mixing jobs (RR with 1ms)

A: 100% CPU, 0% Disk B: 100% CPU, 0% Disk

C: 1/11 CPU, 10/11 Disk



How well would RR with 1 ms slice work?

Good Disk utilization (~90%)

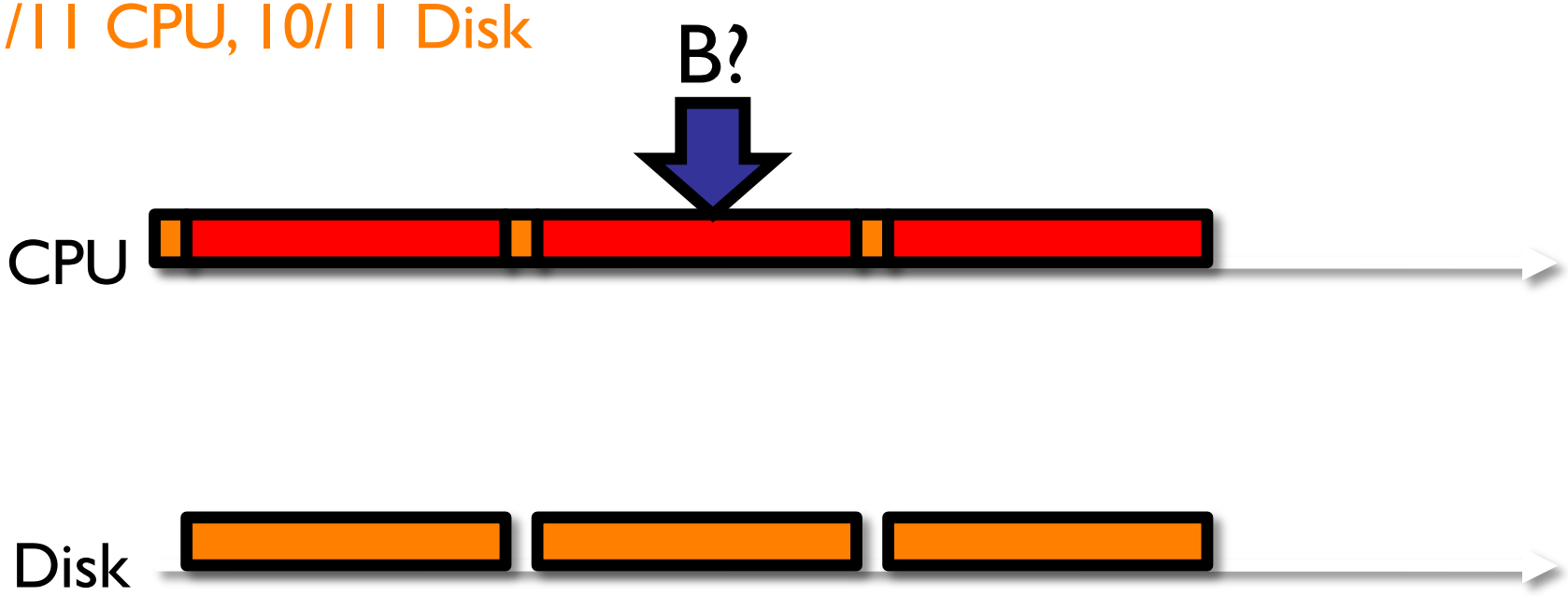
Good principle: start things that can be parallelized early

A lot of context switches though …

Mixing jobs (STCF-P)

A: 100% CPU, 0% Disk B: 100% CPU, 0% Disk

C: 1/11 CPU, 10/11 Disk



How well would STCF-P work? (run C as soon as its I/O is done)

Good Disk utilization (~90%)

Many fewer context switches

Why not run B here? When will B run? When A finishes

Real-time scheduling

- So far, we've focused on average-case
- Alternative scheduling goal
  - Finish everything before its deadline
  - Calls for worst-case analysis
- How do we meet all of our deadlines?
  - **Earliest-deadline-first** (optimal)
  - Used by students to complete all homework assignments
  - Used by professors to juggle teaching and research…and life
- Note: sometimes tasks get dropped (for both of us)

Earliest-deadline-first (EDF)

- EDF
  - Run the job with the earliest deadline
  - If a new job comes in with earlier deadline
    - Pre-empt the current job
    - Start the next one
- This is optimal
  - (assuming it is possible to meet all deadlines)

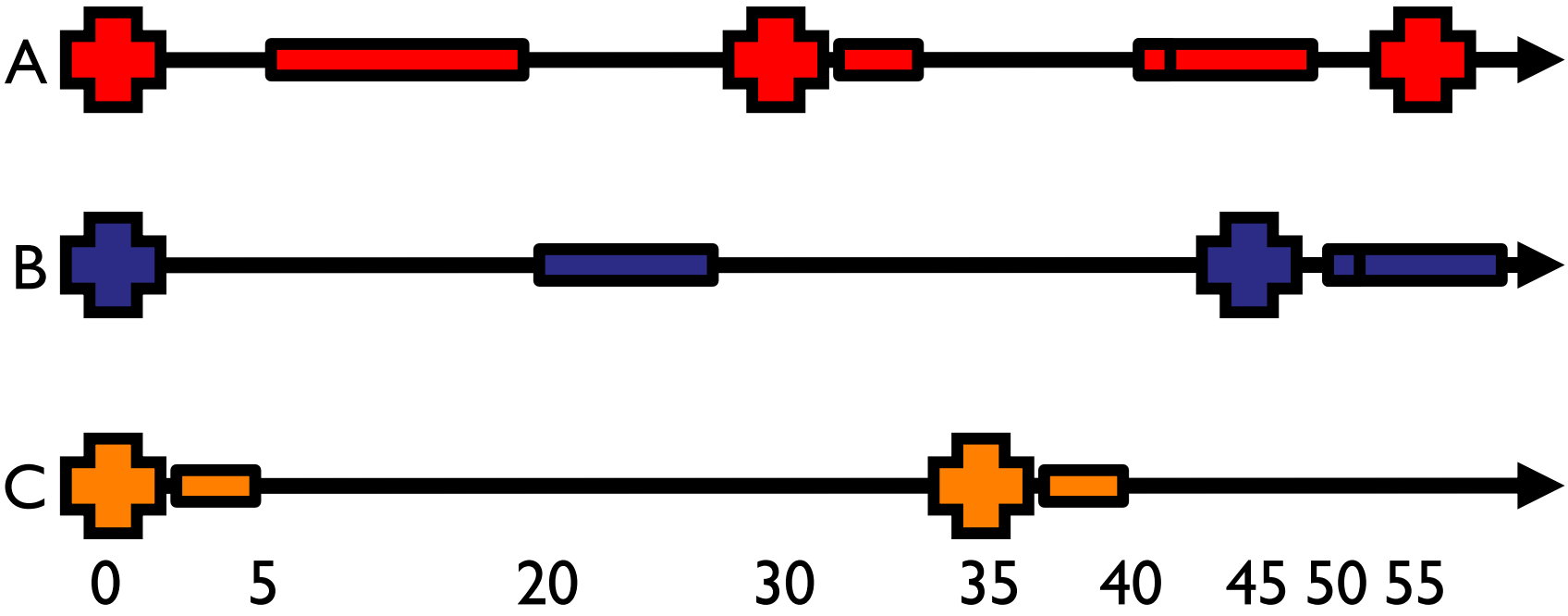
EDF example

- Job A
  - Takes 15 seconds
  - Due 20 seconds after entry
- Job B
  - Takes 10 seconds
  - Due 30 seconds after entry
- Job C
  - Takes 5 seconds
  - Due 10 seconds after entry

EDF example

A: takes 15, due in 20 B: takes 10, due in 30

C: takes 5, due in 10



Proportional-share Schedulers

- A general class of scheduling algorithms
- Process P_i given a CPU weight $w_i > 0$

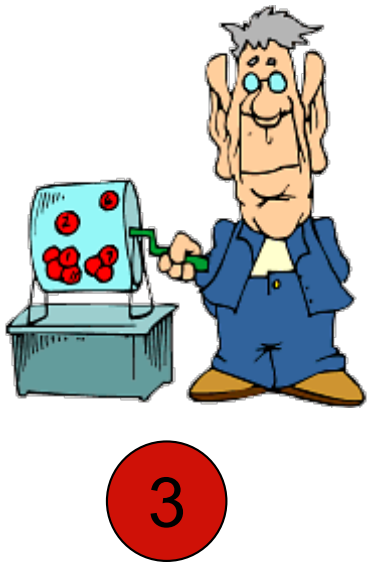
- The scheduler ensures the following
  - *forall* i, j, $|T_i(t_1, t_2)/T_j(t_1,t_2) - w_i/w_j| \leq e$
  - Given P_i and P_j were not blocked during $[t_1,t_2]$
- Who chooses the weights and how?
  - Application modeling problem: non-trivial
  - Approaches: analytical, empirical

Lottery Scheduling

- Perhaps the simplest proportional-share scheduler
- Create lottery tickets equal to the sum of the weights of all processes

- Draw a lottery ticket and schedule the process that owns that ticket
  - What if the process is not ready?
  - Draw tickets only for ready processes

Lottery Scheduling



3

P1=6

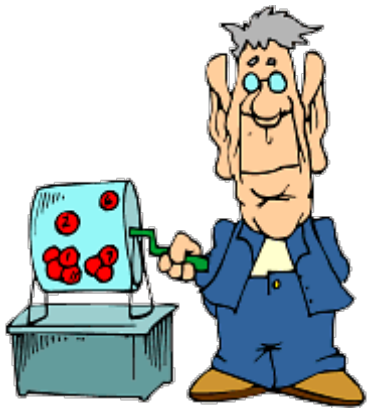
1	4
2	5
3	6

P2=9

7	10	13
8	11	14
9	12	15

Schedule P1

Lottery Scheduling



9

P1=6

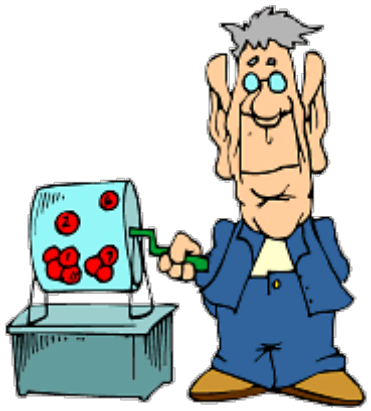
1	4
2	5
3	6

P2=9

7	10	13
8	11	14
9	12	15

Schedule P2

Lottery Scheduling



11

P1=6 *P2=9*

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

Schedule P2

- As t approaches infinite, processes will get their share (unless they were blocked a lot)
  - P1 = 6/15 = 40%, and P2 = 9/15 = 60%
- Problem: only a probabilistic guarantee
- What does the scheduler have to do
  - When a new process arrives?
  - When a process terminates?

Lottery Scheduling

- **Simple Implementation of proportional-share**
  - Used by many VMs to fairly share CPU of a single computer
  - Implementation is efficient and requires no history
  - Currency abstraction is hierarchical
    - Can delegate lottery tickets to threads or other processes
  - What happens if I take lottery tickets away after you "spend" them?

- **Proportional-share is important in other areas**
  - Example: bandwidth allocation, memory
  - Conflicts between principals (e.g., VMs) may reduce efficiency
    - As with a mechanical disk