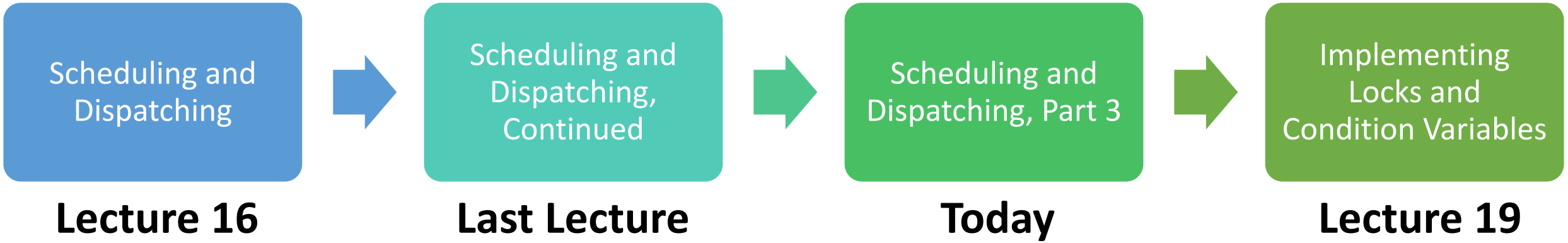
CS111, Lecture 18
Scheduling and Preemption

Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

CS111 Topic 3: Multithreading, Part 2



assign5: implement your own version of **thread**, **mutex** and **condition_variable**!

Learning Goals

- Explore the tradeoffs in deciding which threads get to run and for how long
- Learn about the assign5 infrastructure and how to implement a dispatcher with *preemption*

Plan For Today

- **Recap:** Context Switching
- Scheduling Threads
- Preemption and Interrupts

Plan For Today

- **Recap: Context Switching**
- Scheduling Threads
- Preemption and Interrupts

Context Switching

A *context switch* means changing the thread currently running to another thread. We must save the current thread state and load in the new thread state.

1. Push all registers besides stack onto current thread's stack
2. Save the current stack register (rsp) into the thread's state space
3. Load the other thread's saved stack register from its state space into rsp
4. Pop registers off the other thread's stack

Super funky: we are calling a function from one thread's stack and execution and returning from it in **another** thread's stack and execution!

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

1. Push all registers besides stack onto current thread's stack

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

2. Save the current stack register (rsp) into the thread's state space

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

3. Load the other thread's saved stack register from its state space into rsp

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

4. Pop registers off the other thread's stack

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

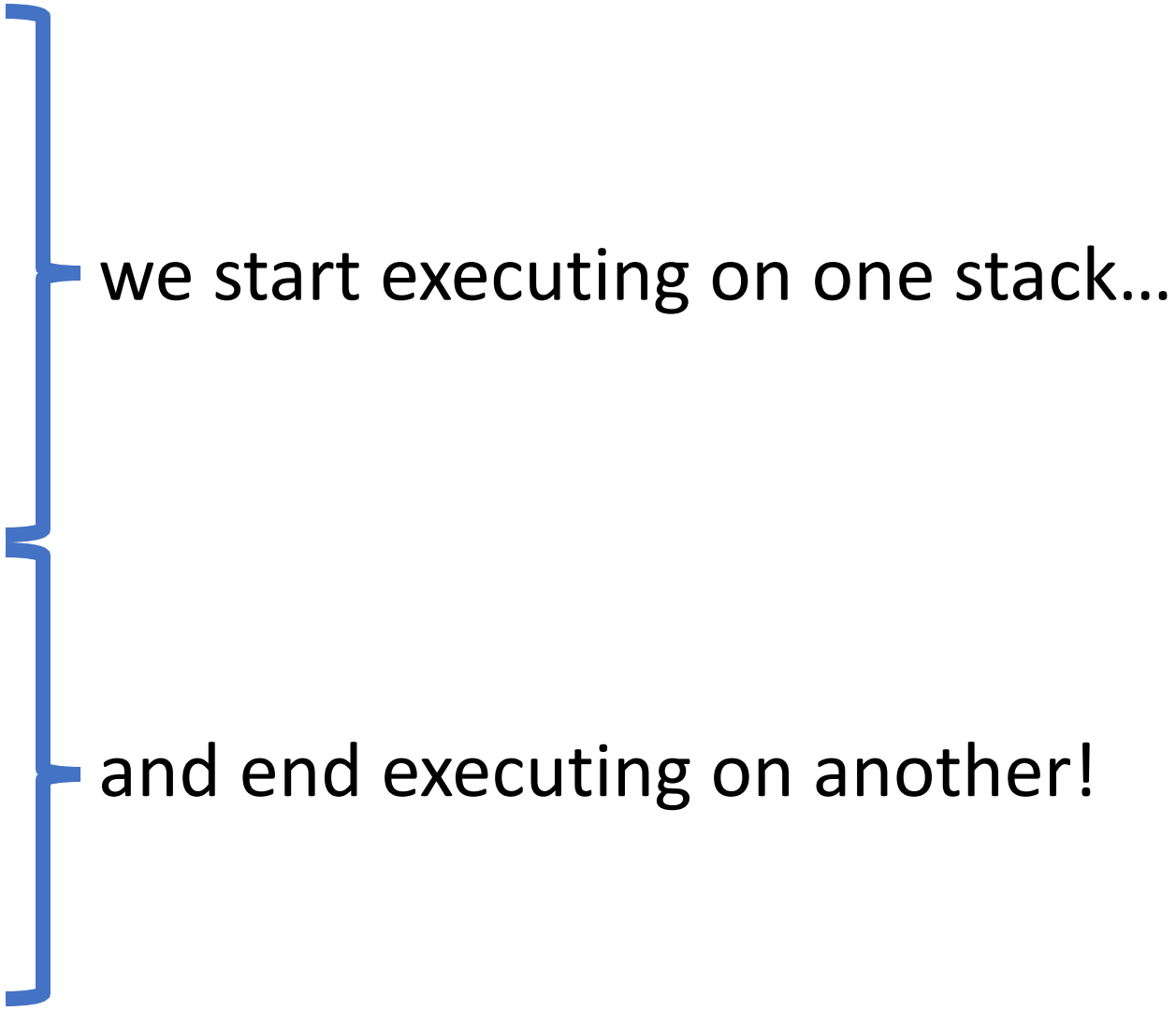
Now we return back to the function in the **new thread** that called **context_switch** previously! (recall: **ret** pops the address off the stack for the instruction we should resume at in the caller)

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```



Context Switching

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

We enter via a call from a function in the current thread

We exit to a call from a function in the new thread!

Plan For Today

- Recap: Context Switching
- **Scheduling Threads**
- Preemption and Interrupts

First-come-first-serve

Key Question: How does the operating system decide which thread to run next? (e.g. many **ready** threads). Assume just 1 core.

One idea - "first-come-first-serve": keep all ready threads in a *ready queue*. Add threads to the back. Run the first thread on the queue until it exits or blocks (no timer).

Problem: thread could run away with core and run forever!

Round Robin

Problem: thread could run away with core and run forever!

Solution: define a *time slice*, the max run time without a context switch (e.g. 10ms).

Idea: round robin scheduling – run thread for one time slice, then put at back of ready queue. (you'll use this on assign5)

Question: what's a good time slice?

Thought: we want to run many threads in the amount of time for human response time, so e.g. keystroke seems instantaneous. **So why not make the time slice microscopically small?**

Round Robin

Idea: round robin scheduling – run thread for one time slice, then put at back of ready queue. (you'll use this on assign5)

Question: what's a good time slice? Why not make it microscopically small?

If too small, context switch costs are very high, waste cores

Why not make it very large?

If too large, slow response, threads can monopolize cores

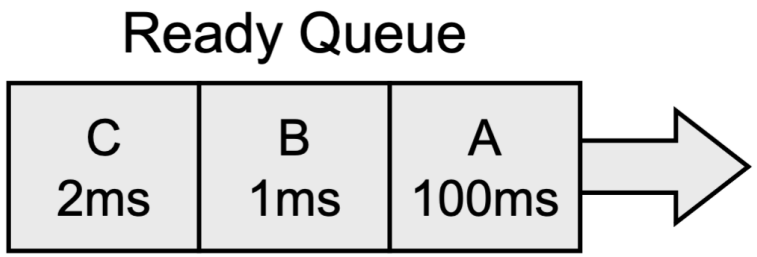
Try to balance: usually in 5-10ms range, Linux is 4ms

Scheduling Algorithms

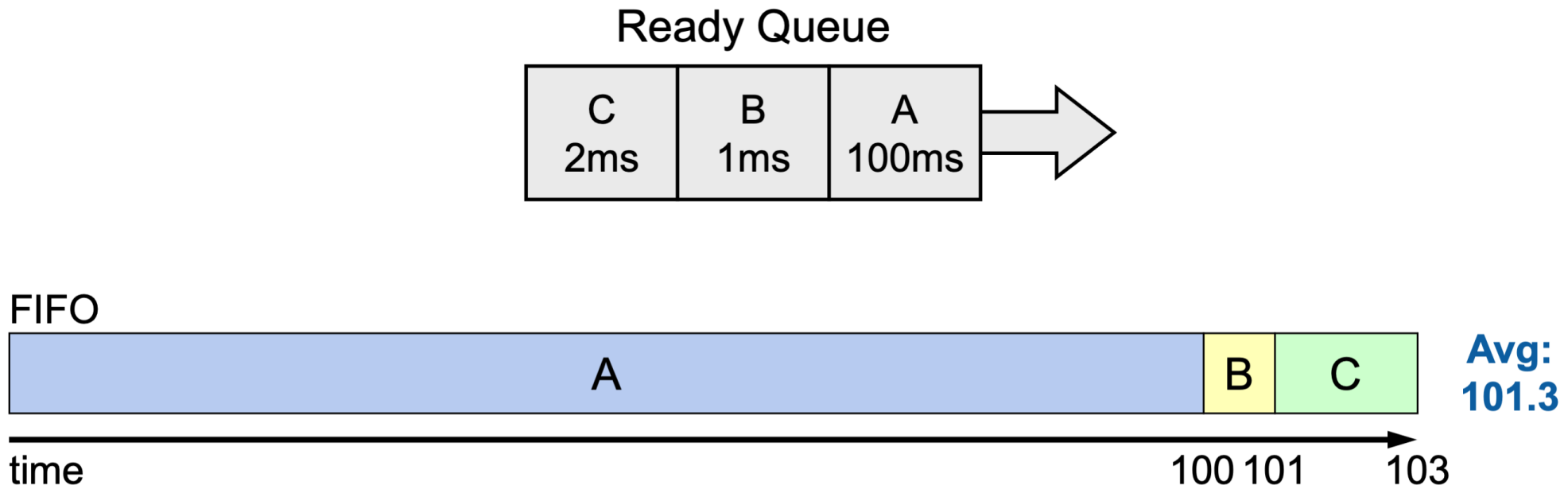
How do we decide whether a scheduling algorithm is good?

- Minimize response time (time to useful result)
  - e.g. keystroke -> key appearing, or “make” -> program compiled
  - Assume useful result is when the thread blocks or completes
- Use resources efficiently
  - keep cores + disks busy
  - low overhead (minimize context switches)
- Fairness (e.g. with many users, or even many jobs for one user)

Comparing FCFS/RR: Scenario 1

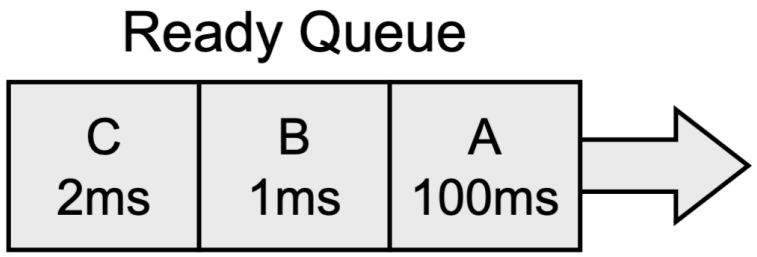


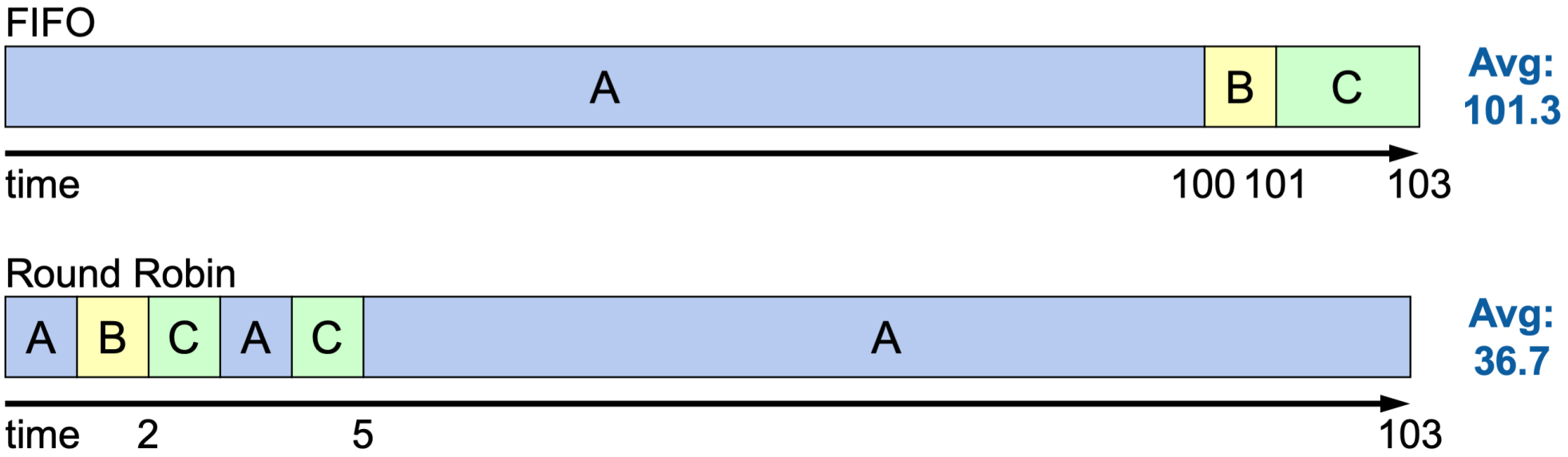
Comparing FCFS/RR: Scenario 1



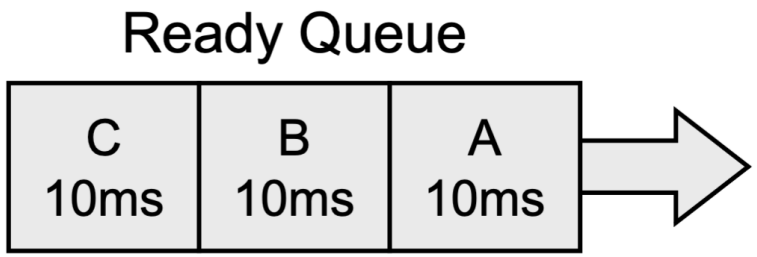
Comparing FCFS/RR: Scenario 1

Is RR ***always*** better than FCFS?

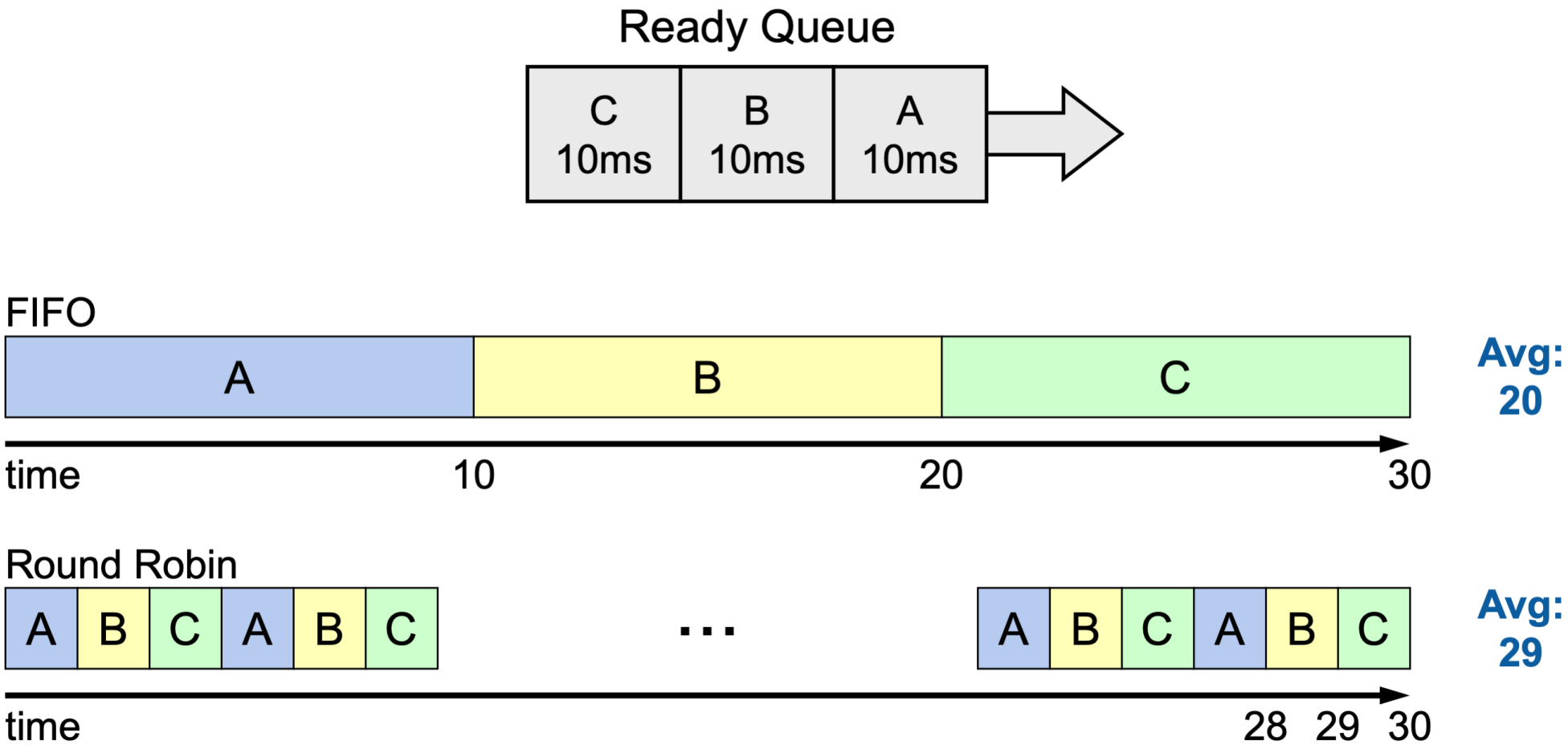




Comparing FCFS/RR: Scenario 2



Comparing FCFS/RR: Scenario 2



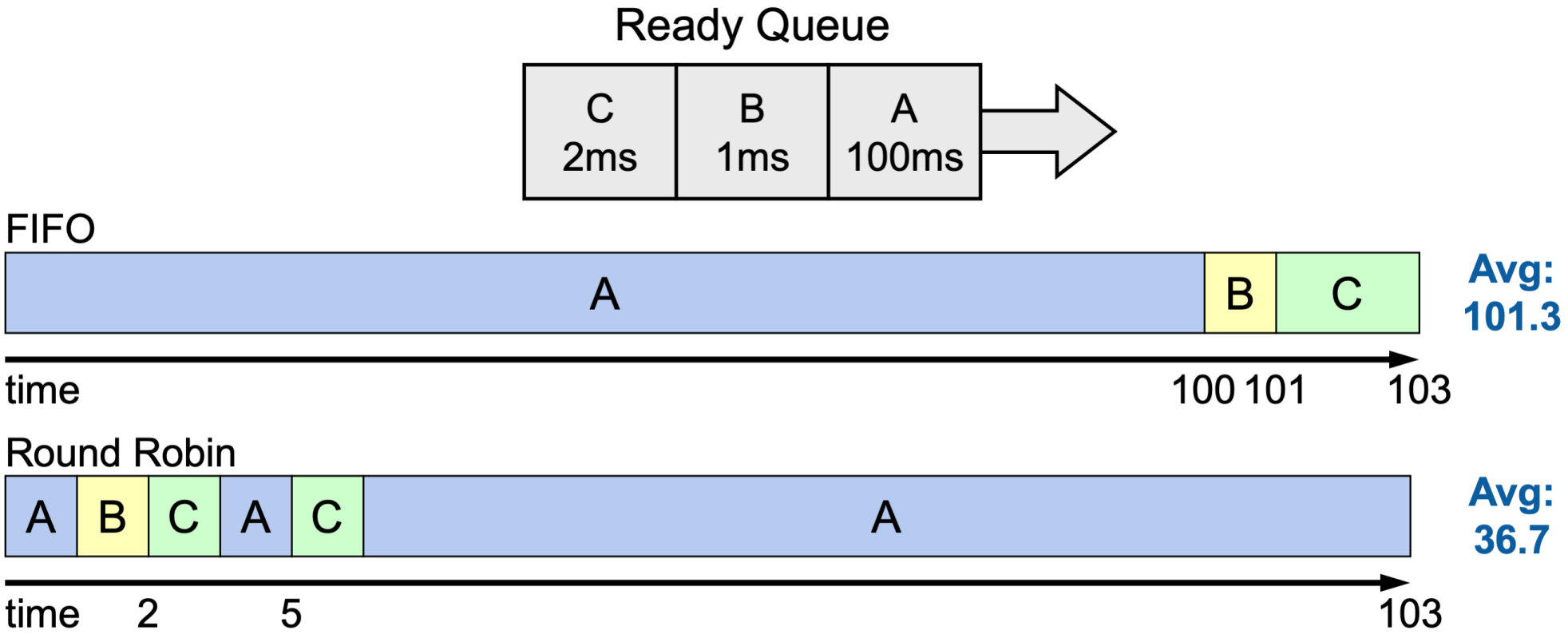
What's the optimal approach if we want to minimize average response time?

Shortest Remaining Processing Time

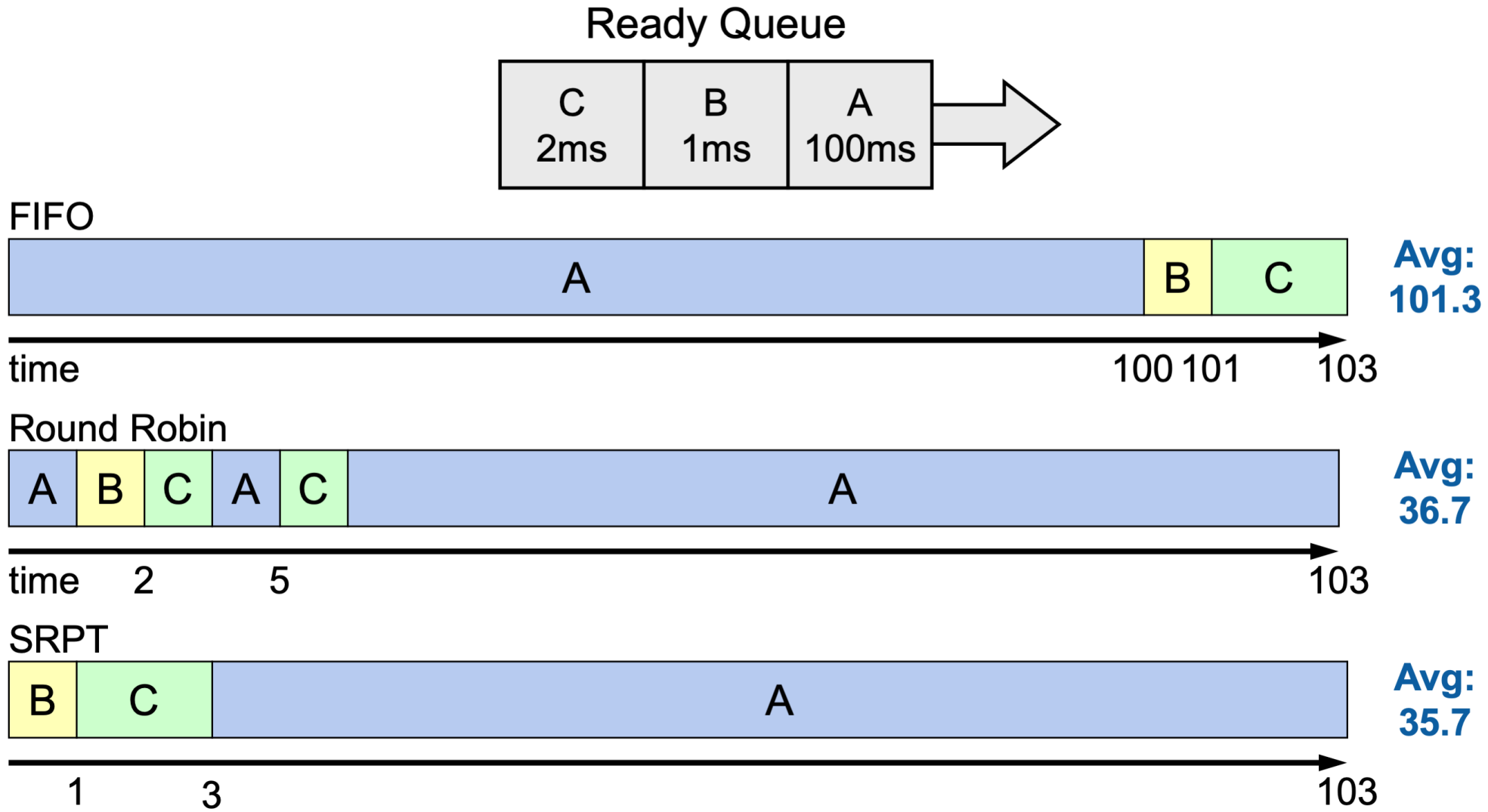
What would it look like if we optimized for completion time? (time to finish, or time to block).

Idea - SRPT: pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

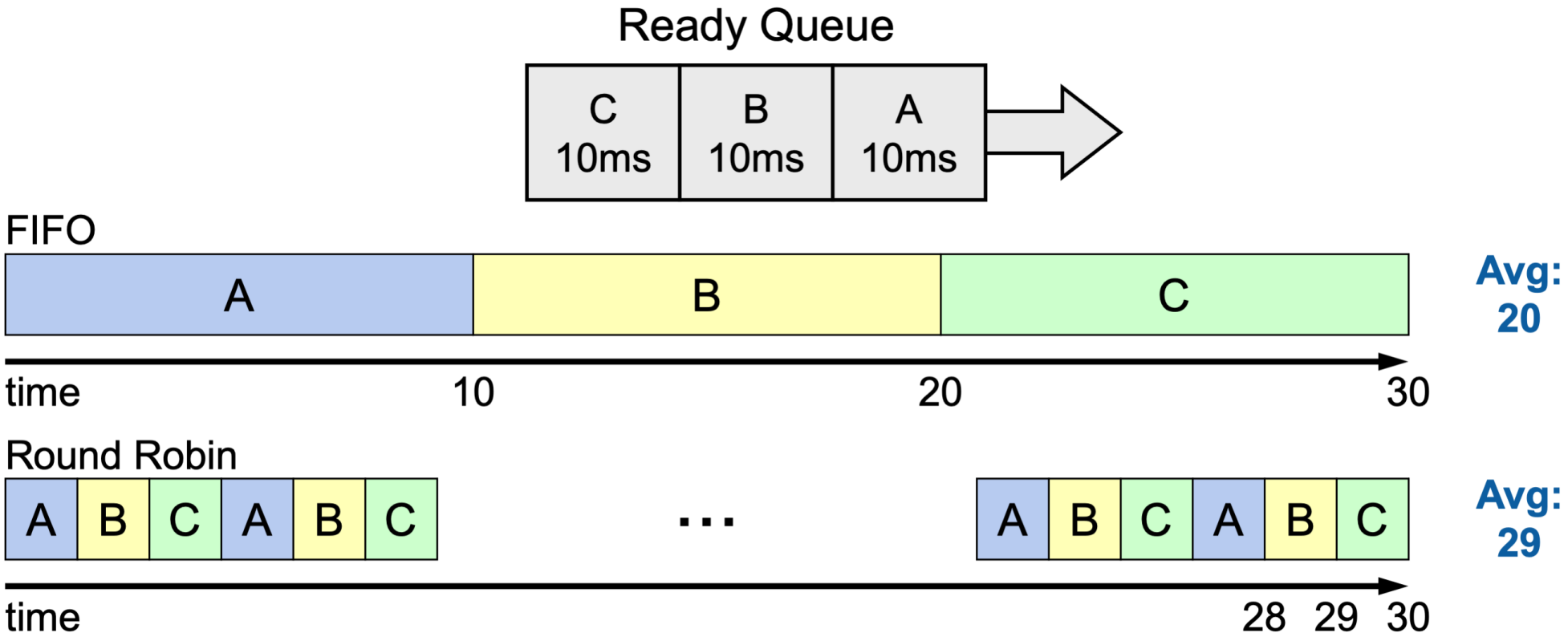
Evaluating SRPT



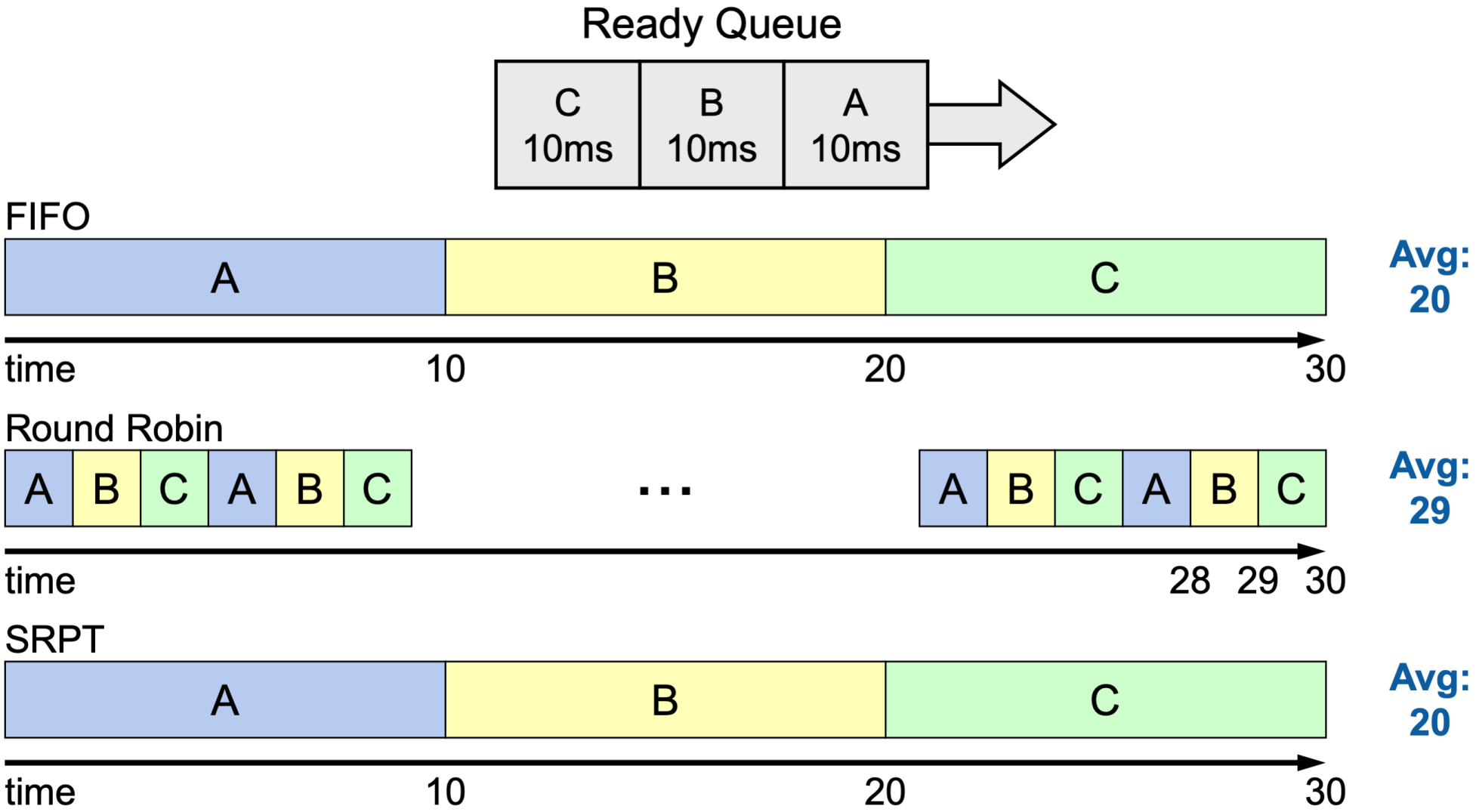
Evaluating SRPT



Evaluating SRPT



Evaluating SRPT



Shortest Remaining Processing Time

SRPT: pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

What are some problems/challenges with the SRPT approach?

Respond with your thoughts on PollEv:
pollev.com/cs111 or text CS111 to 22333 once to join.

What are some problems/challenges with the SRPT approach?

Shortest Remaining Processing Time

SRPT: pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

Problem #1: how do we know which one will finish most quickly? (we must be able to predict the future…)

Problem #2: if we have many short-running threads and one long-running one, the long one will not get to run (“starvation”)

SRPT

Another advantage of SRPT: improves overall resource utilization

- "I/O-Bound" - the time to complete them is dictated by how long it takes for some external mechanism to complete its work (disk, network)
- "CPU-Bound - the time to complete them is dictated by how long it takes us to do the CPU computation
- If a thread is **I/O-Bound** – e.g. constantly reading from disk (frequently waits for disk), it will get priority vs. thread that needs lots of CPU time – **CPU Bound**.

Gives preference to those who need the least.

Problem: how can we get close to SRPT but without having to predict the future or neglect certain threads?

Priority-Based Scheduling

Goal: we want to get close to SRPT, but without having to predict the future, and without neglecting certain threads.

Key Idea: can use past performance to predict future performance.

- Behavior tends to be consistent
- If a thread runs for a long time without blocking, it's likely to continue running

Priority-Based Scheduling

Goal: we want to get close to SRPT, but without having to predict the future, and without neglecting certain threads.

Idea: let's make threads have priorities that adjust over time as they run. We'll have 1 ready queue for each priority, and always run highest-priority threads.

- Overall idea: threads that aren't using much CPU time stay in the higher-priority queues, threads that are migrate to lower-priority queues.
- After blocking, thread starts in highest priority queue
- If a thread reaches the end of its time slice without blocking it moves to the next lower queue.

Problem: could still neglect long-running threads!

Priority-Based Scheduling

Idea: let's make threads have priorities that adjust over time as they run. We'll have 1 ready queue for each priority, and always run highest-priority threads.

Problem: could still neglect long-running threads!

Let's keep track of *recent CPU usage per thread*. If a thread hasn't run in a long time, its priority goes up. And if it has run a lot recently, priority goes down. (4.4 BSD Unix used this, ideas carried forward)

- No more neglecting threads: a thread that hasn't run in a long time will get its priority increased
- If there are many equally-long threads that want to run, the priorities even out over time, at a kind of "equilibrium"

Plan For Today

- Recap: Context Switching
- Scheduling Threads
- **Preemption and Interrupts**

Preemption and Interrupts

On assign5, you'll implement a **dispatcher with scheduling** using the Round Robin approach.

- *Preemptive*: threads can be kicked off in favor of others (after time slice)

To implement this, we've provided a **timer** implementation that lets you run code every X microseconds.

- Fires a timer interrupt at specified interval

Plan For Today

- Recap: Context Switching
- Scheduling Threads
- Preemption and Interrupts

Next time: preemption and implementing mutexes

Lecture 18 takeaway: For scheduling, we want to minimize response time, use resources efficiently, and be fair. SRPT is optimal, and we want to get close to that. To implement preemption, we can use a timer and context switch if needed when it fires.