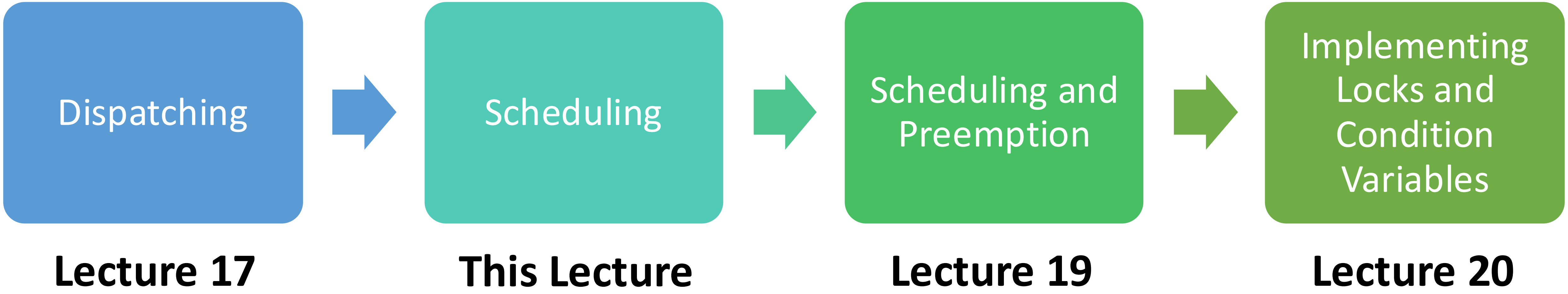
CS111, Lecture 18

Scheduling

CS111 Topic 3: Multithreading, Part 2

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



Dispatching

Lecture 17

Scheduling

This Lecture

Scheduling and Preemption

Lecture 19

Implementing Locks and Condition Variables

Lecture 20

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

Learning Goals

- Explore the tradeoffs in deciding which threads get to run and for how long
- Learn about 4 different scheduling algorithms and their tradeoffs

Plan For Today

- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

Plan For Today

- **Recap: Dispatching**
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

Running and Switching Between Threads

- When we want to run a new thread, we “freeze frame” (save register values) the current running thread, save that, and load in the “freeze frame” of the thread we want to run.
- Mechanisms (Traps, Interrupts) to ensure OS eventually regains control.
- Thread state stored in Process Control Block (PCB) – saved %rsp
- *Context switch* is changing to run another thread. It’s a function that, as part of its execution, returns to a *different function in a different thread* than it was called from.
- The *context switch* function pushes registers onto the current thread’s stack, saves its %rsp value to the thread state space, changes to the new thread’s %rsp value, and pops register values from its stack.

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 register 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
```

we start executing on one stack…

and end executing on another!

Context Switching

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

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

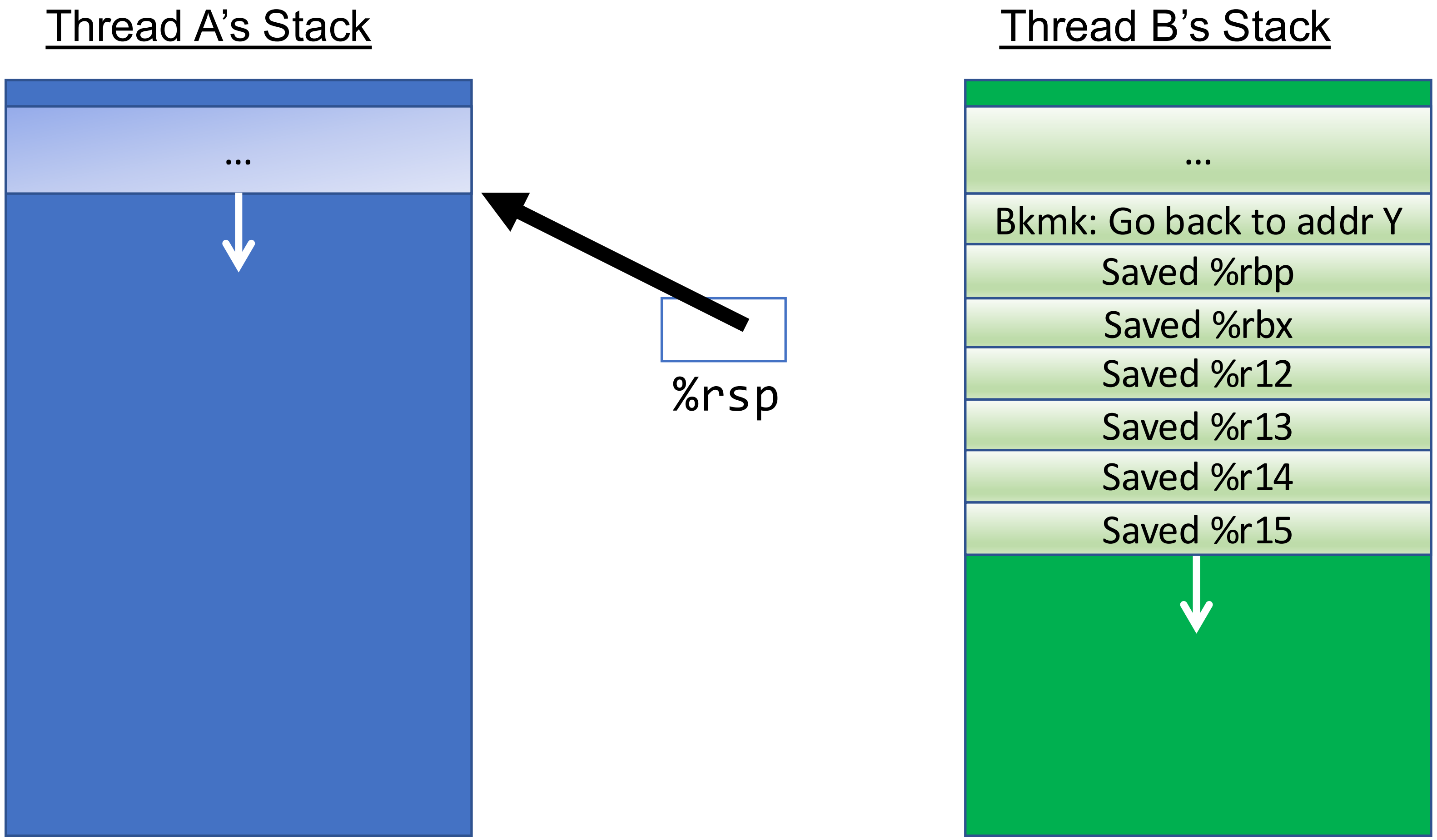
How do we switch what code is running?

It turns out information about what code to run is *also* stored in each thread's stack space, automatically! So by switching stacks, we switch code too.

Key Idea: whenever we call a function, before running that function we store info on the stack about where we should resume in the calling function when we are done. This is called the return address ("bookmark"). This includes when we call context switch.

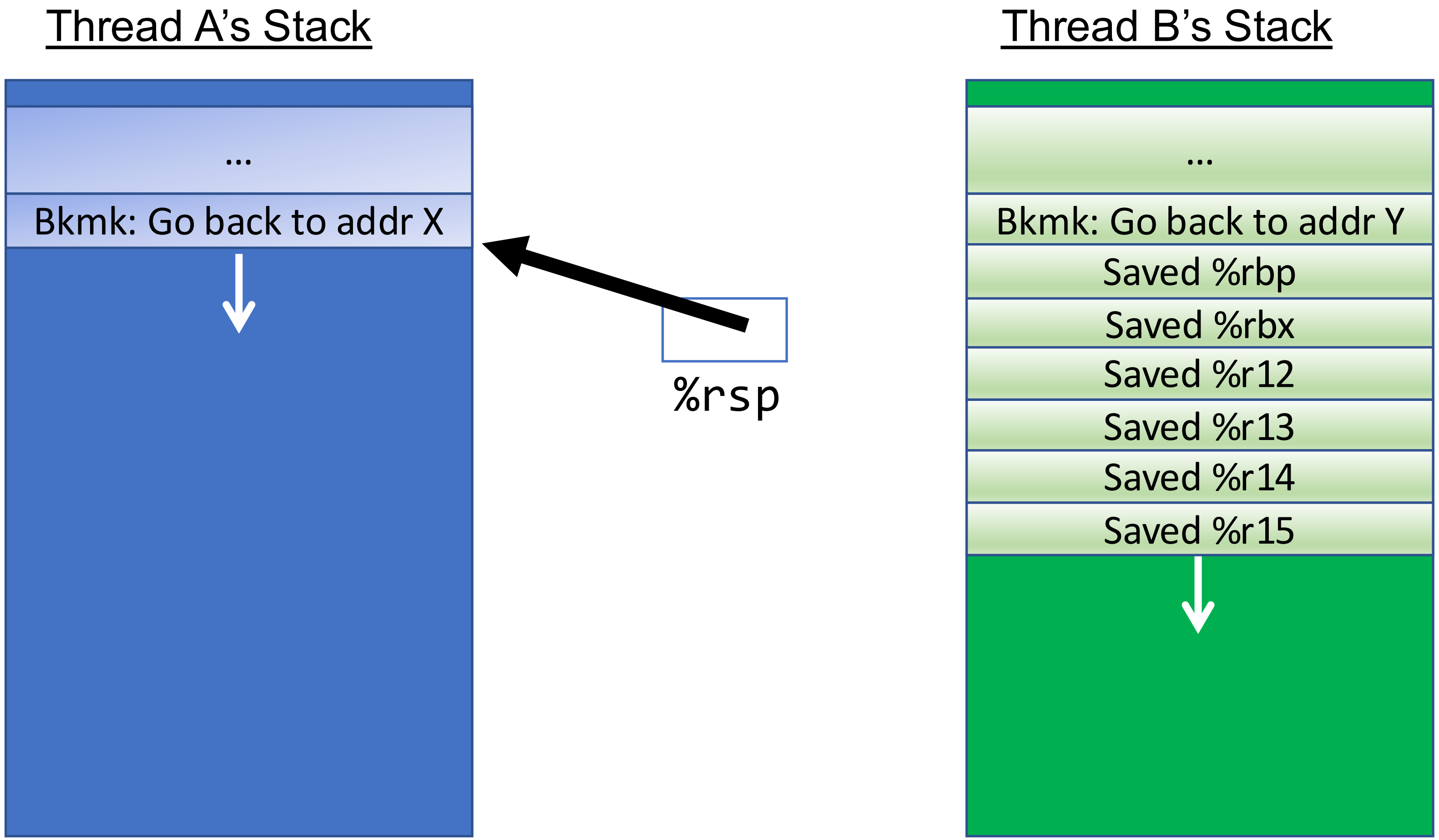
- The **callq** instruction (for calling a function) stores the return address
- The **ret** instruction pops the return address off the stack and resumes executing that code (pops value off and into the %rip register)

Context Switching



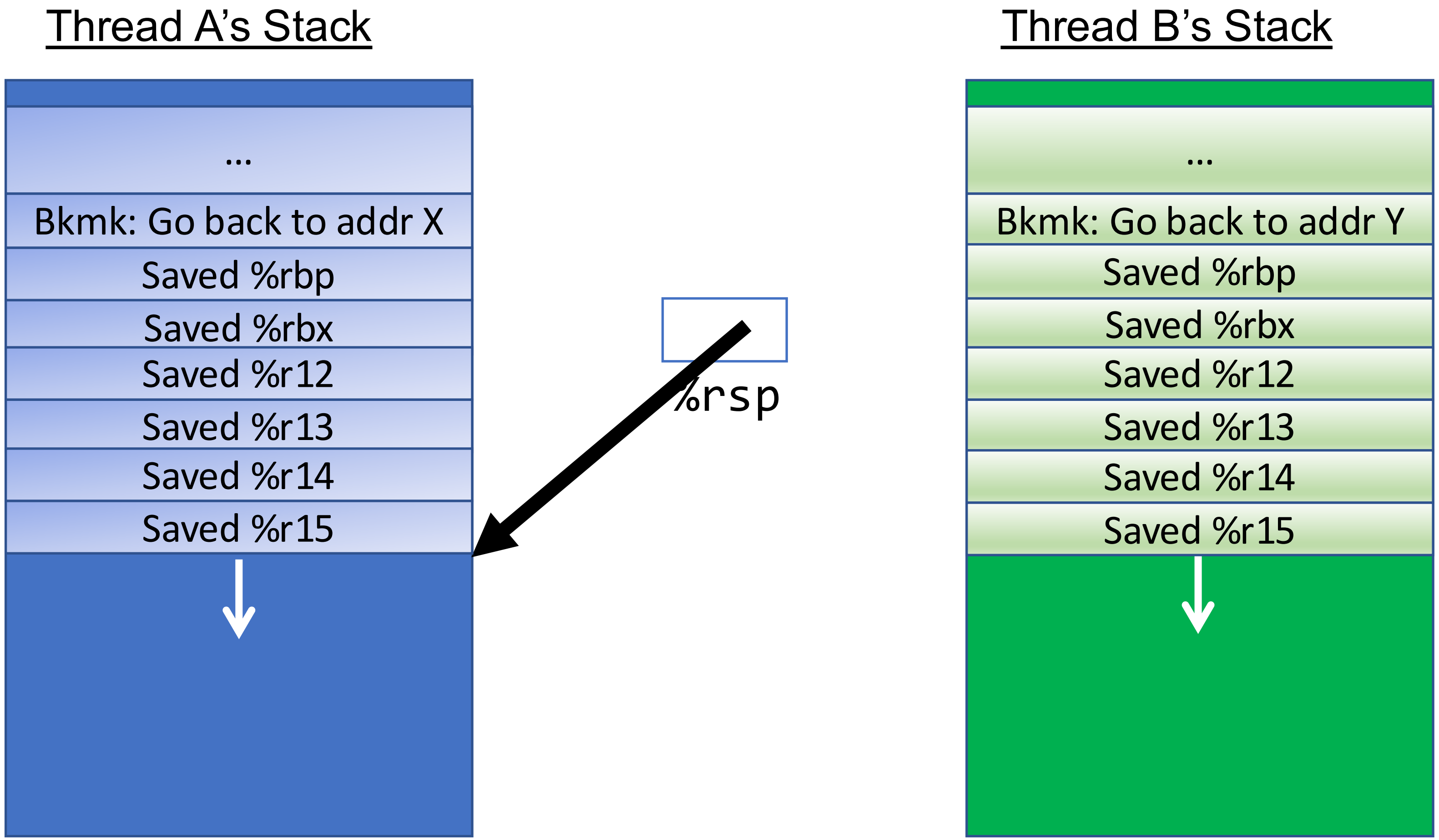
```
callq context_switch
...
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching



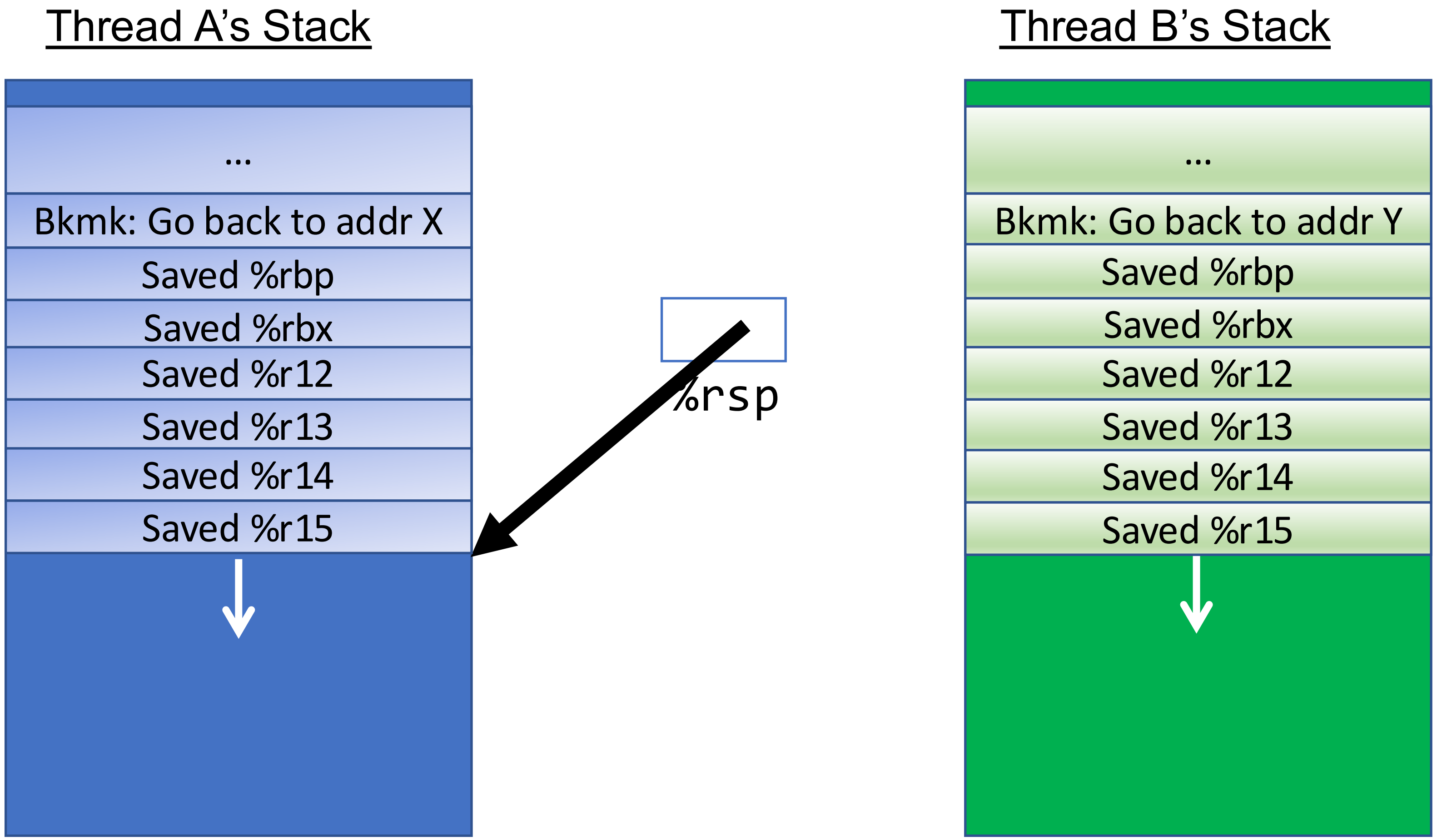
```
callq context_switch
...
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching



```
callq context_switch
...
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



Thread A's Stack

…

Bkmk: Go back to addr X

Saved %rbp

Saved %rbx

Saved %r12

Saved %r13

Saved %r14

Saved %r15

%rsp

Thread B's Stack

…

Bkmk: Go back to addr Y

Saved %rbp

Saved %rbx

Saved %r12

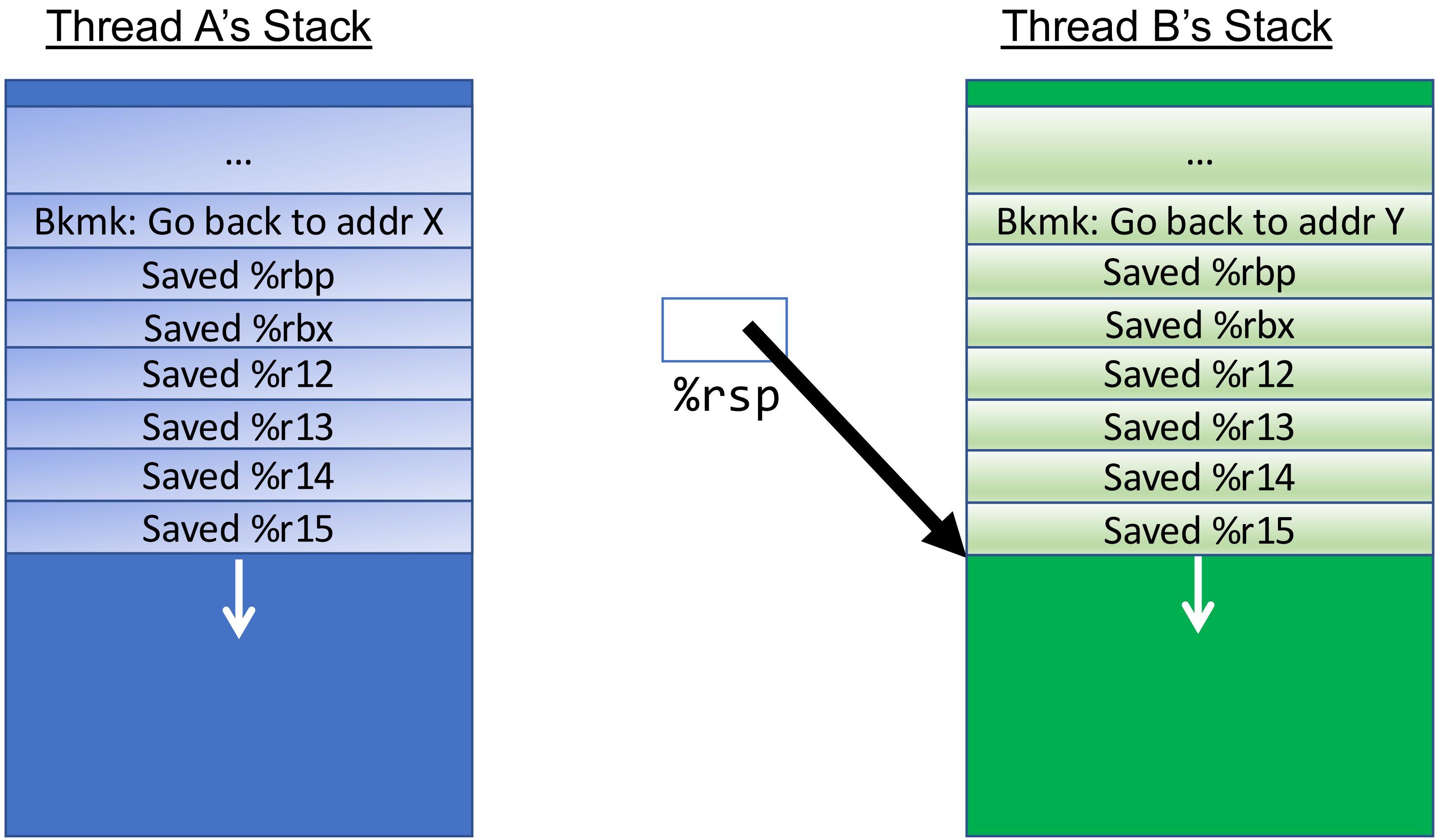
Saved %r13

Saved %r14

Saved %r15

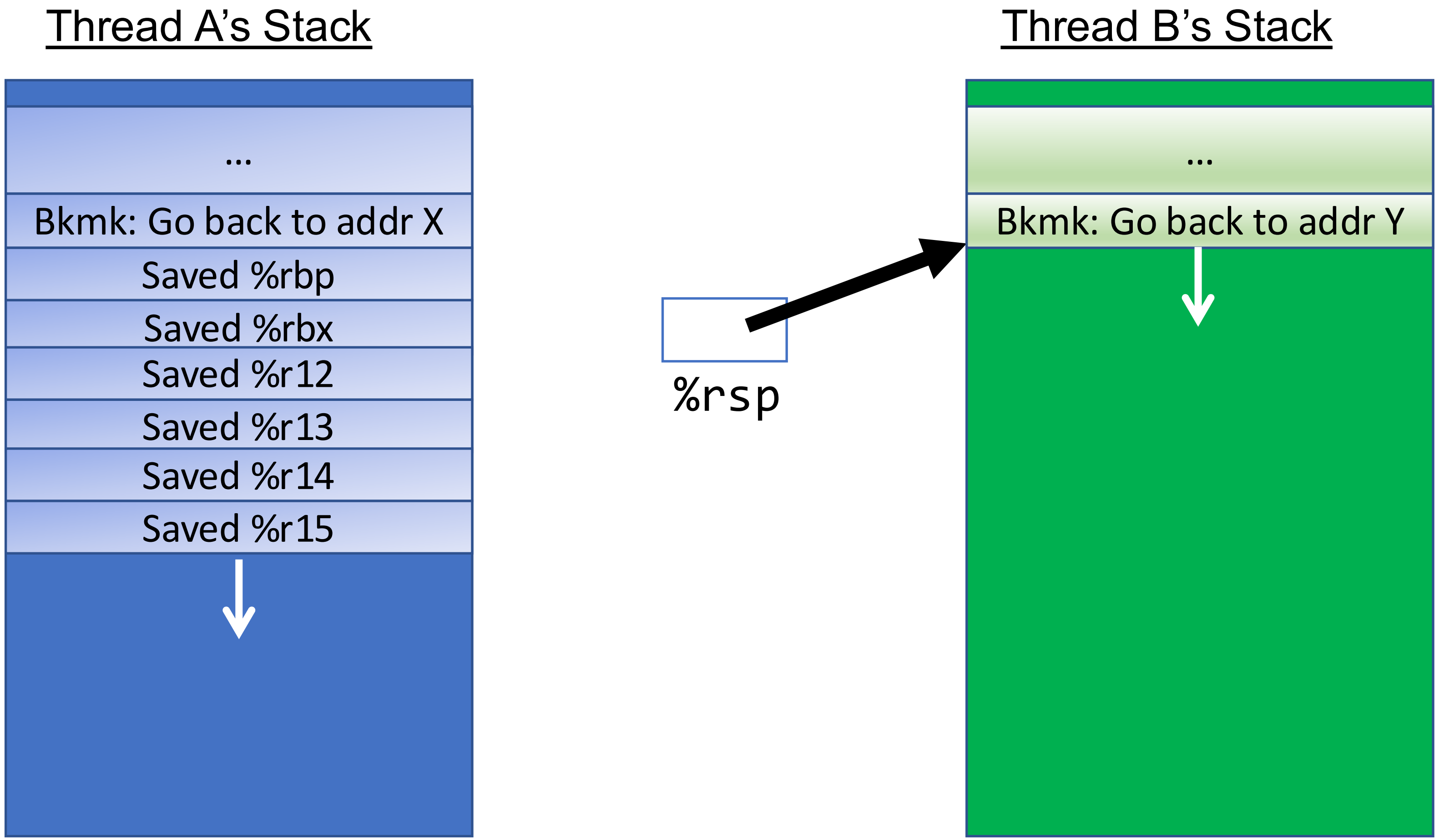
```
callq context_switch
...
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching



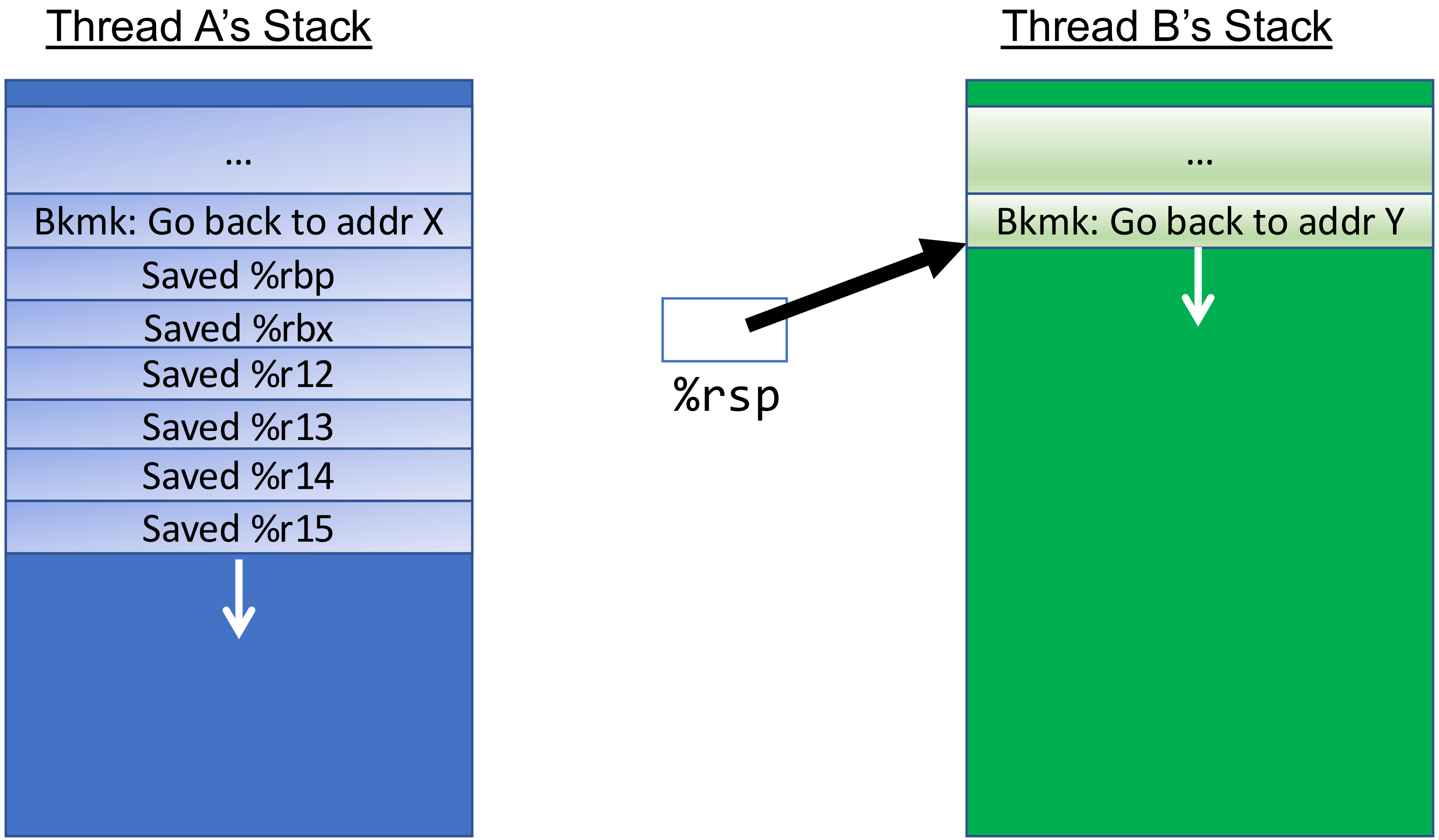
```
callq context_switch
...
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching



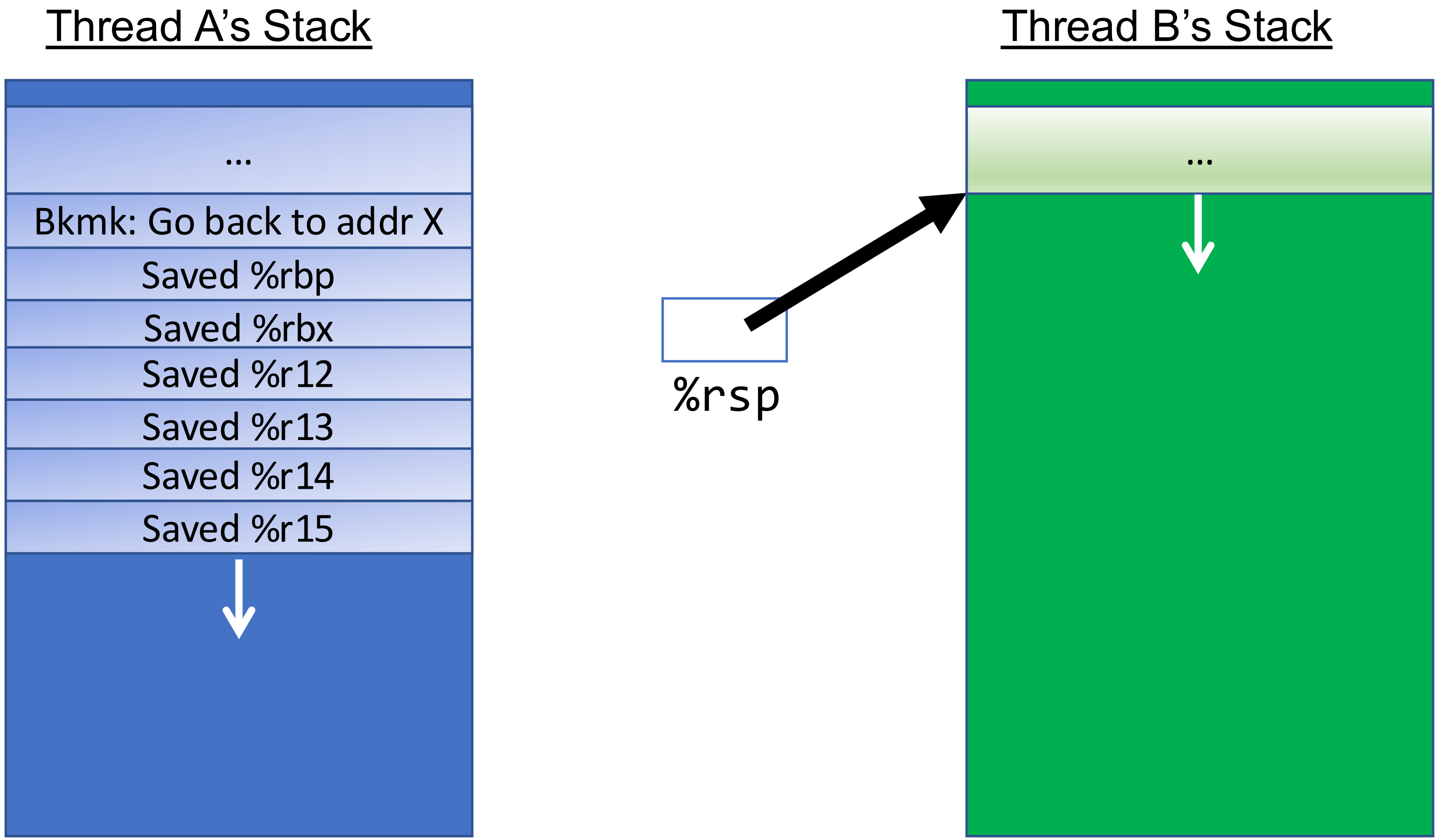
```
callq context_switch
...
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
movq %rsp,0x2000(%rdi)
movq 0x2000(%rsi),%rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
ret
```

Context Switching



```
callq context_switch
...
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



```
callq context_switch
...
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
```

Creating New Threads

Problem: when a thread runs for the first time, it won’t have a “freeze frame”. How does context-switching to a new thread work?

- *Key idea:* when created, we give a thread a fake “saved state” that appears as though it was frozen right before executing its first function.
- In other words; we put fake saved registers and a return address that, when ret runs, will take us "back" to the specified function it should run.

Plan For Today

- **Recap:** Dispatching
- **Scheduling and Thread States**
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

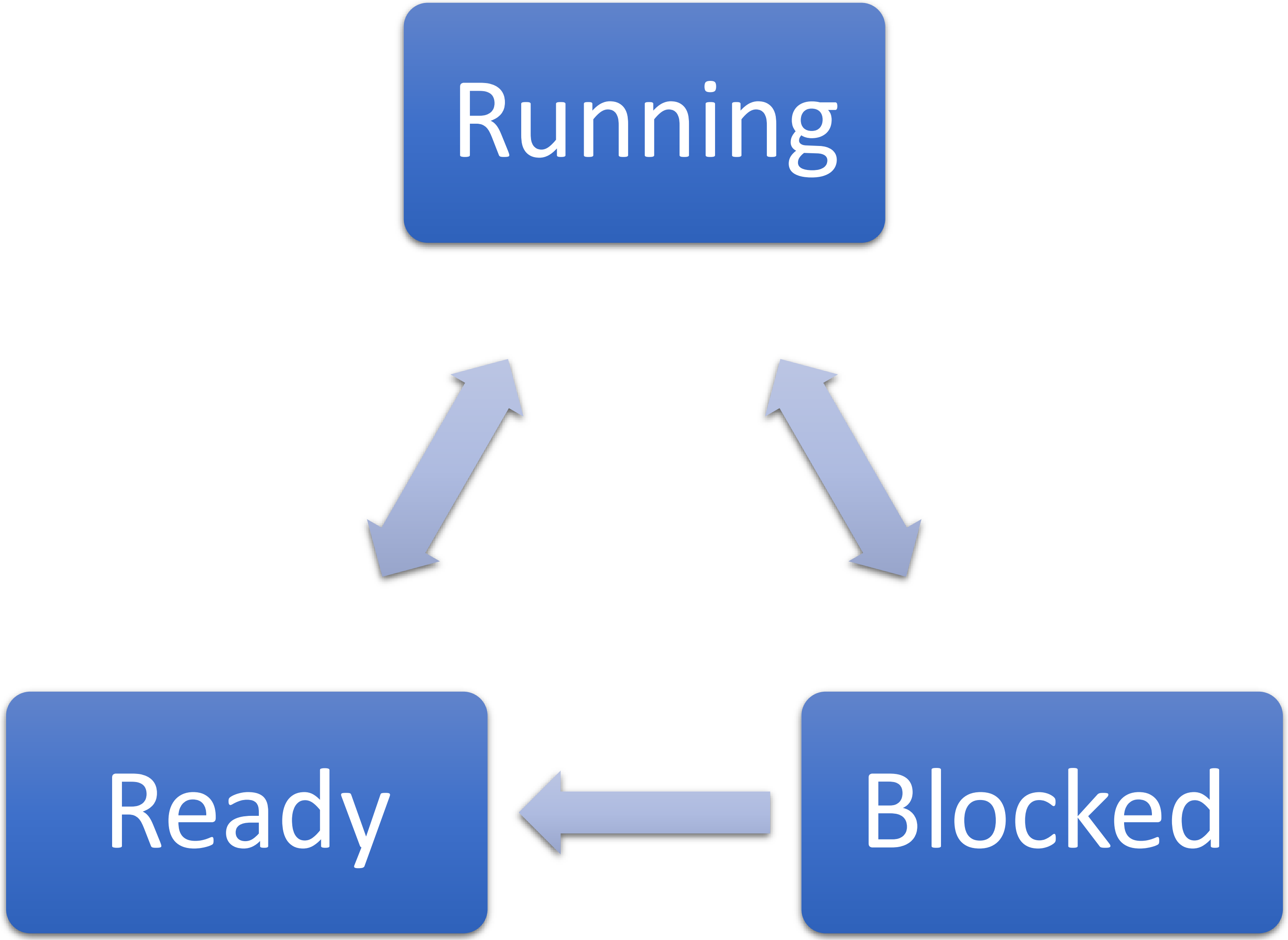
Tracking All Threads

How does the OS track/remember all user threads on the system?

Key idea: at any given time, a thread is in one of three states:

1. **Running**
2. **Blocked –** waiting for an event (disk I/O, network connection, etc.)
3. **Ready** – able to run, but waiting for CPU time

Thread States



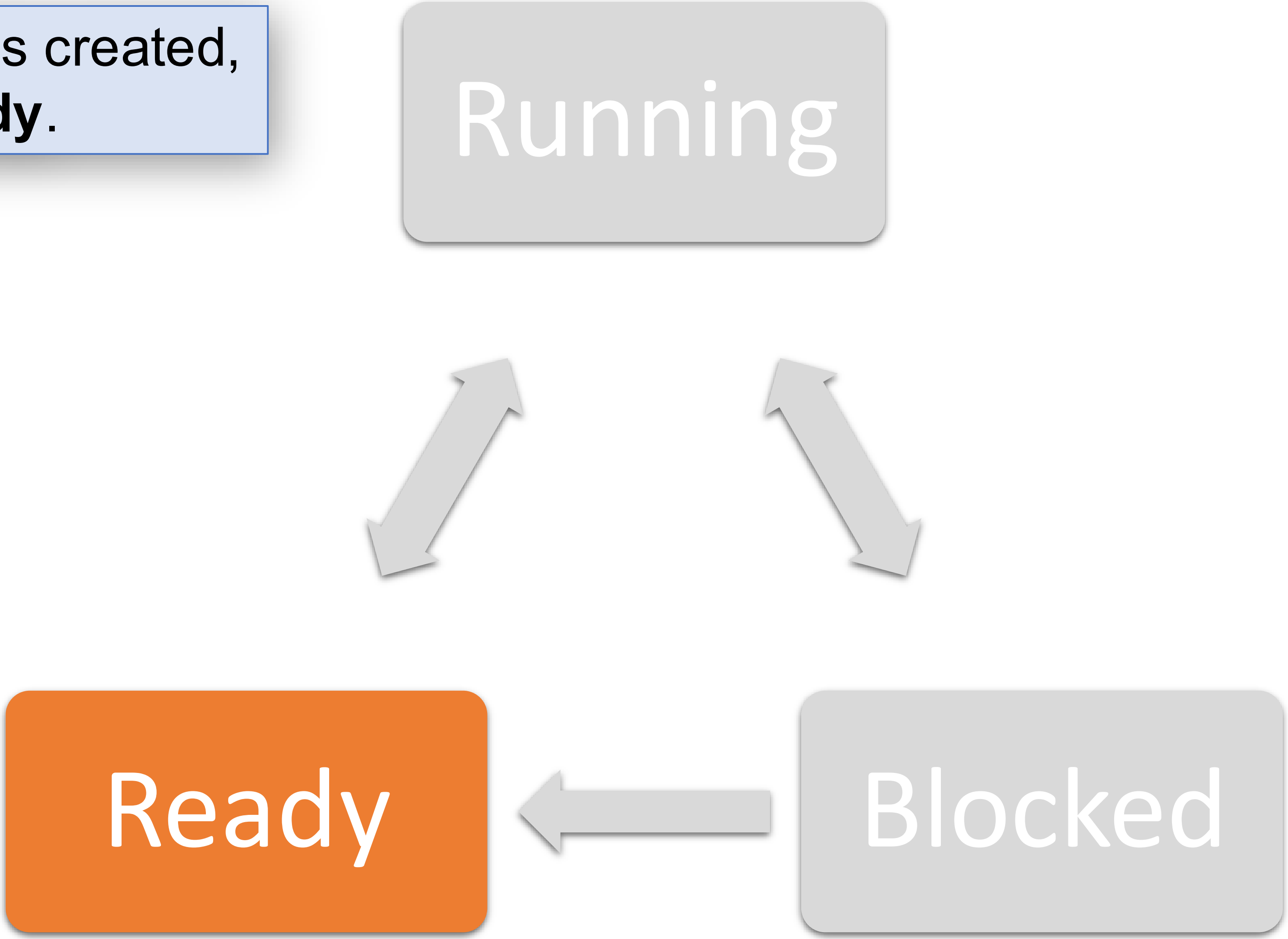
Running

Ready

Blocked

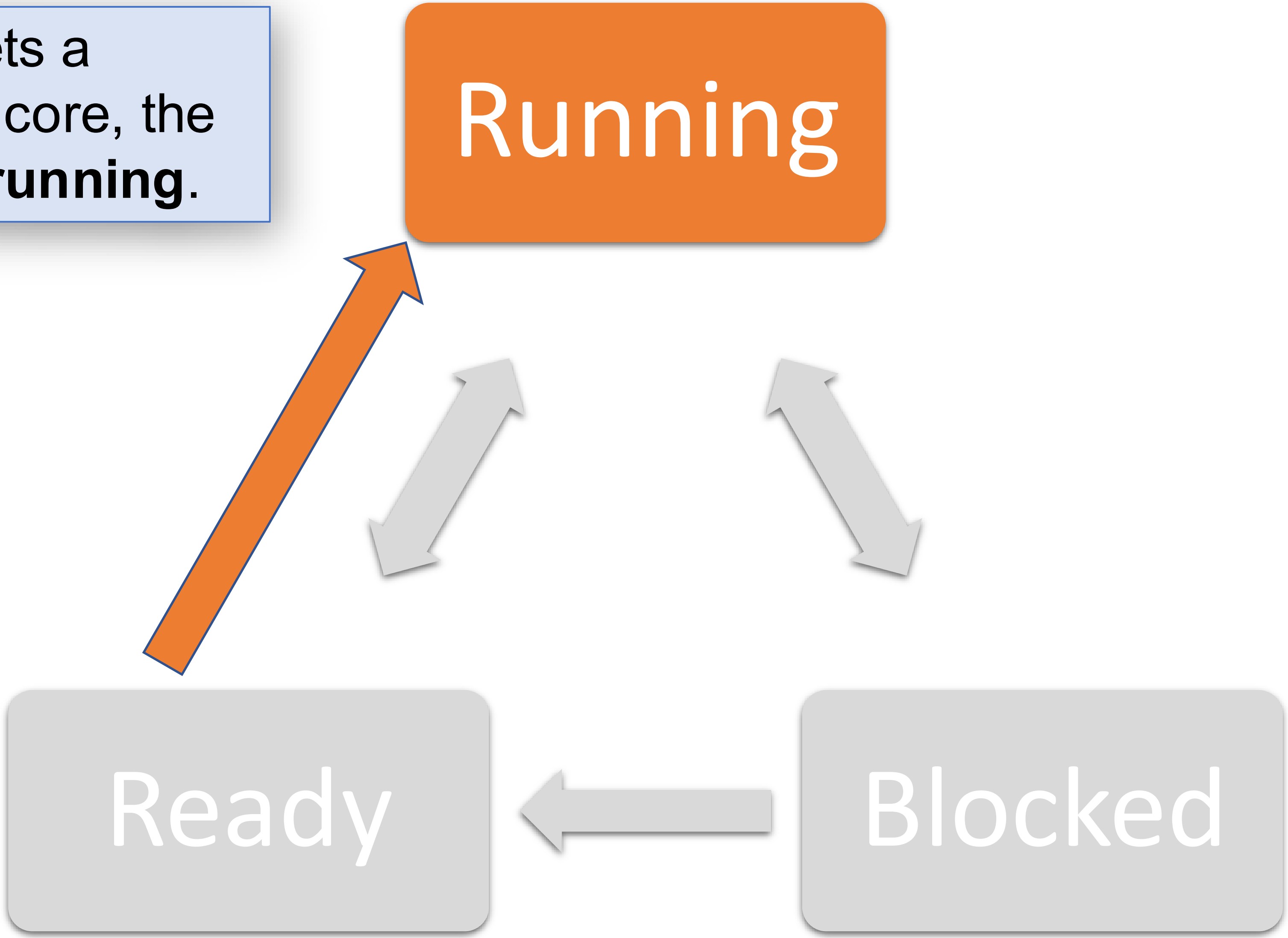
Thread States

When a thread is created, it starts out **ready**.



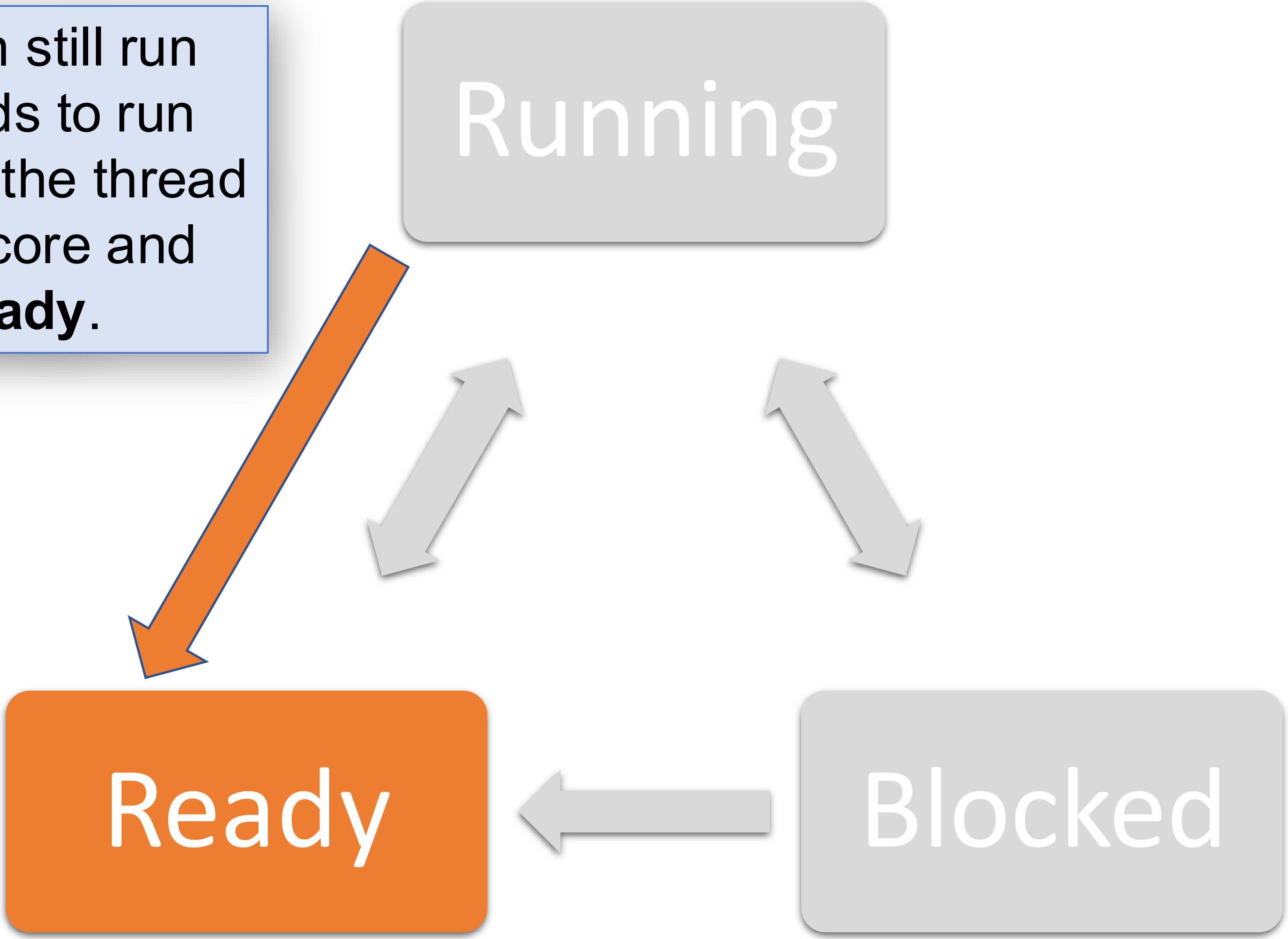
Thread States

When the OS lets a thread run on a core, the thread goes to **running**.



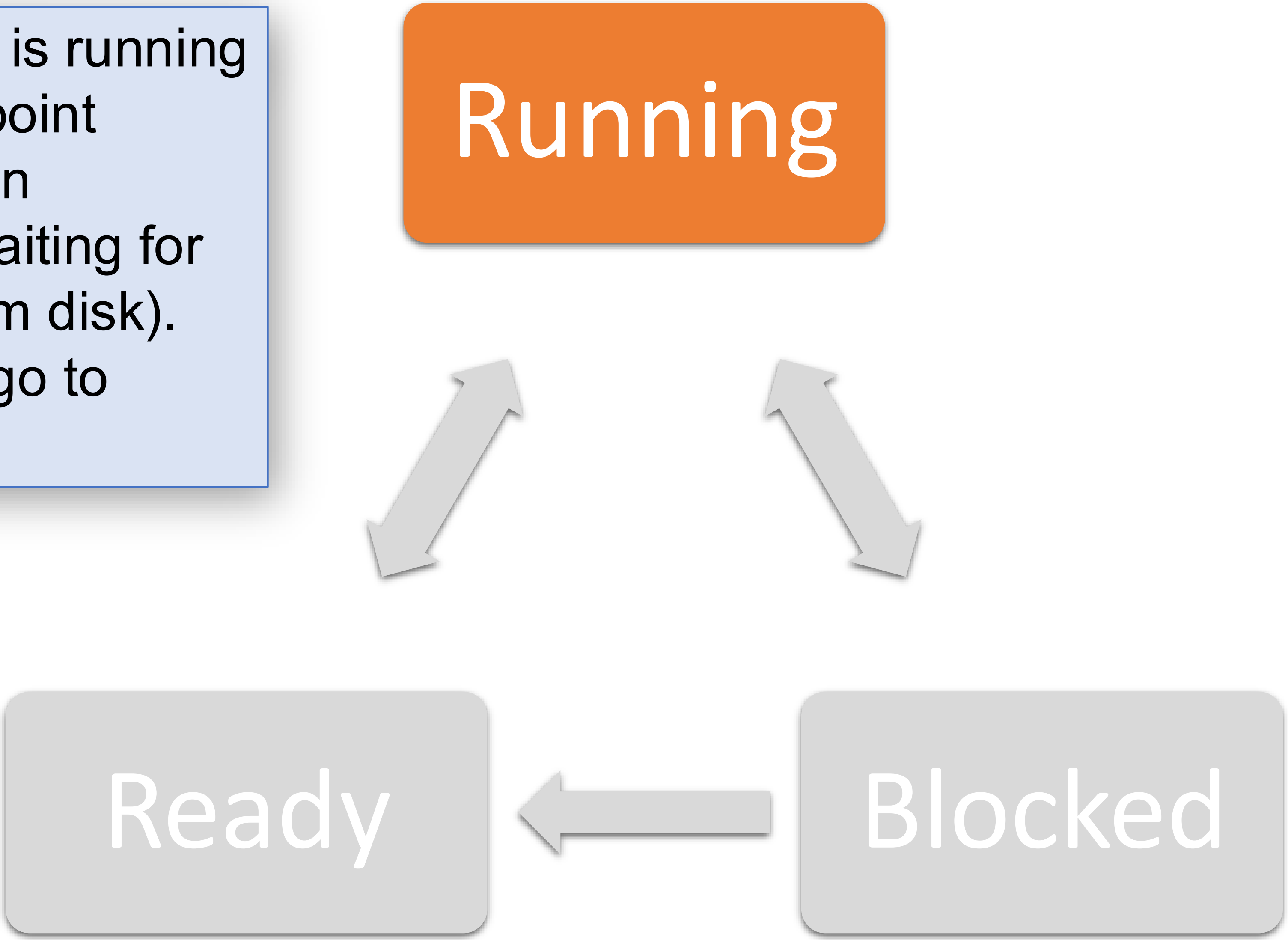
Thread States

If the thread can still run but the OS needs to run another thread, the thread is taken off the core and goes back to **ready**.



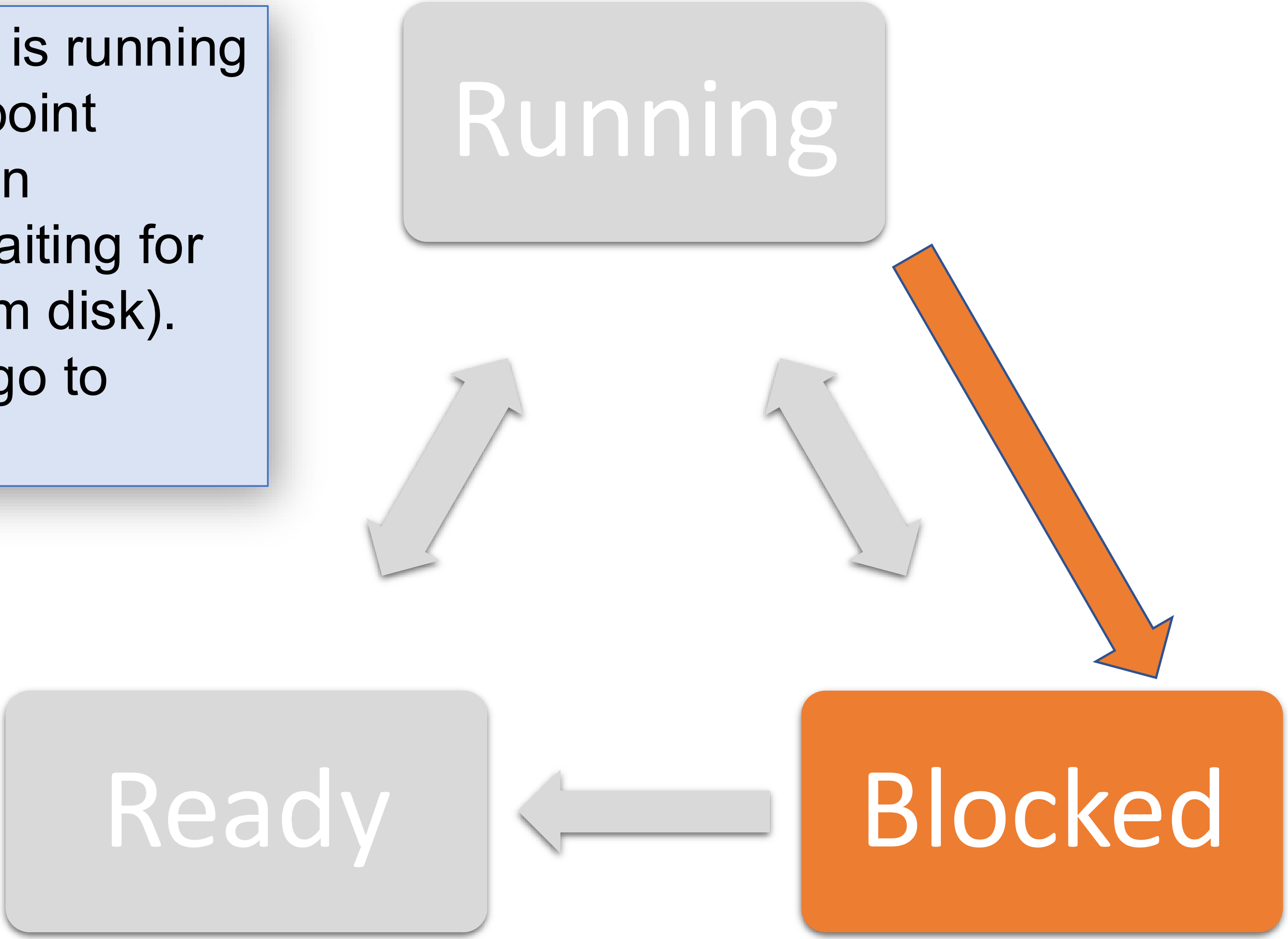
Thread States

Maybe a thread is running and reaches a point where it can't run anymore (eg. waiting for file contents from disk). The thread will go to **blocked**.



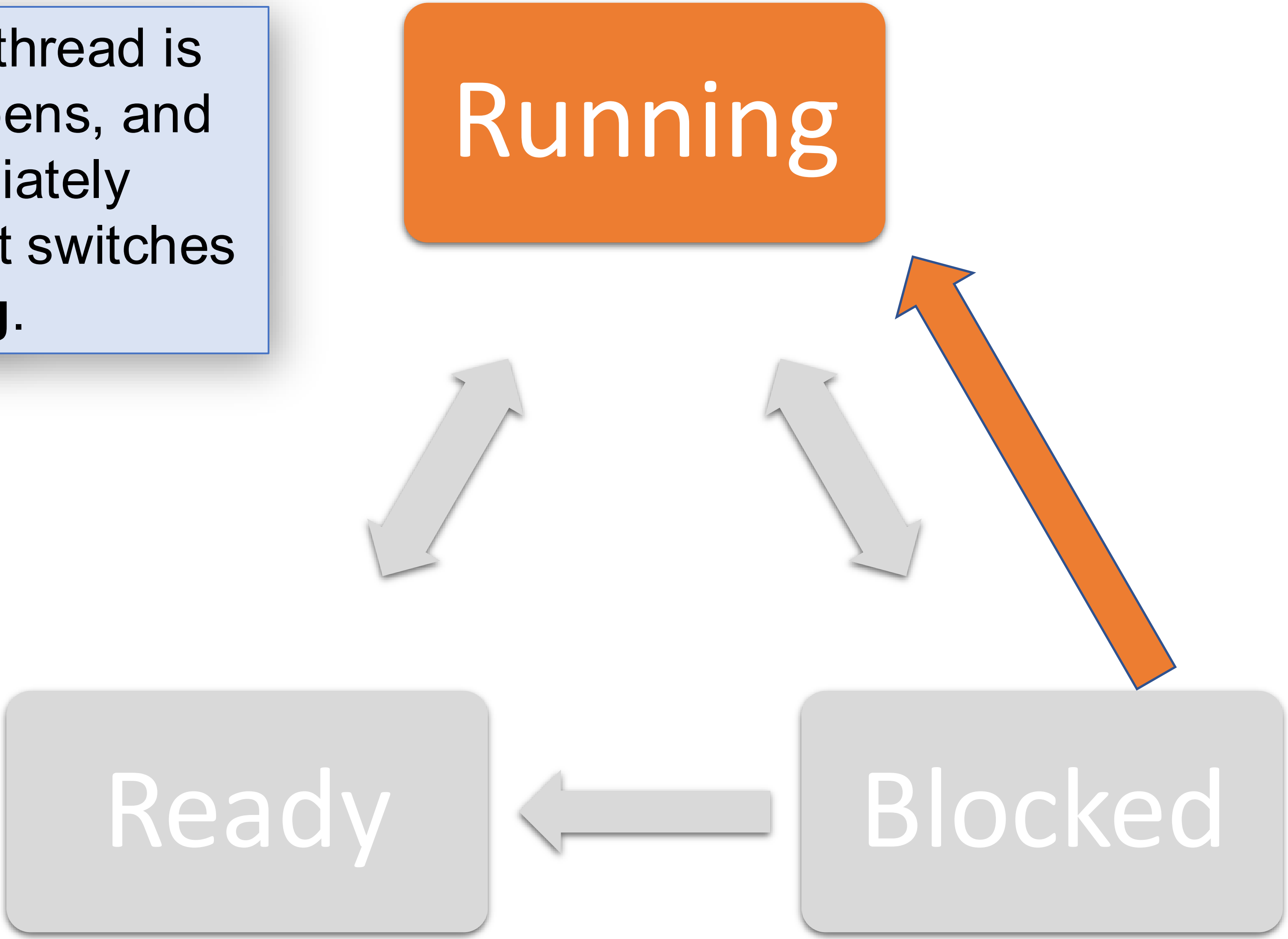
Thread States

Maybe a thread is running and reaches a point where it can't run anymore (eg. waiting for file contents from disk). The thread will go to **blocked**.



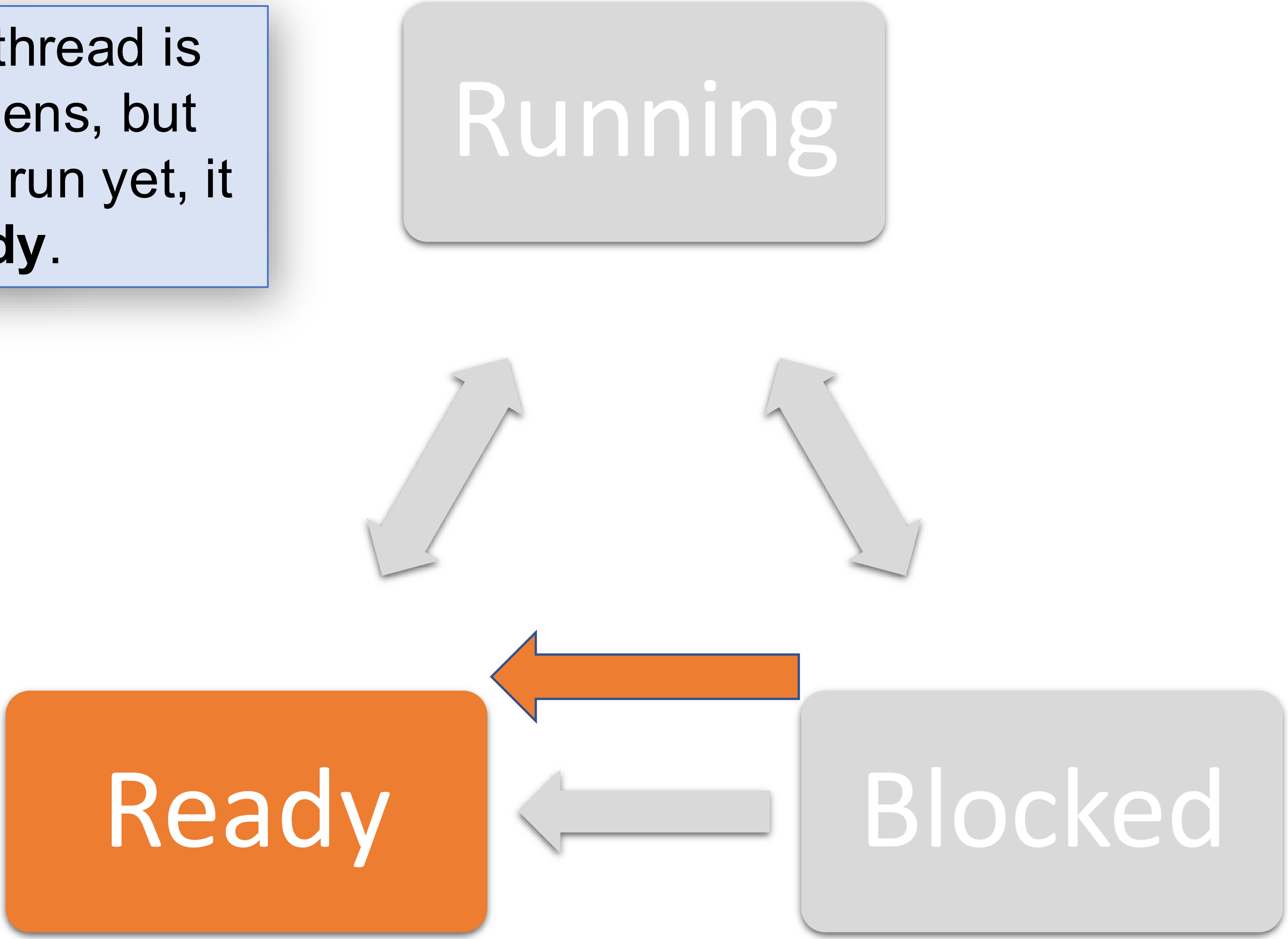
Thread States

If the event the thread is waiting for happens, and a core is immediately available for it, it switches back to **running**.



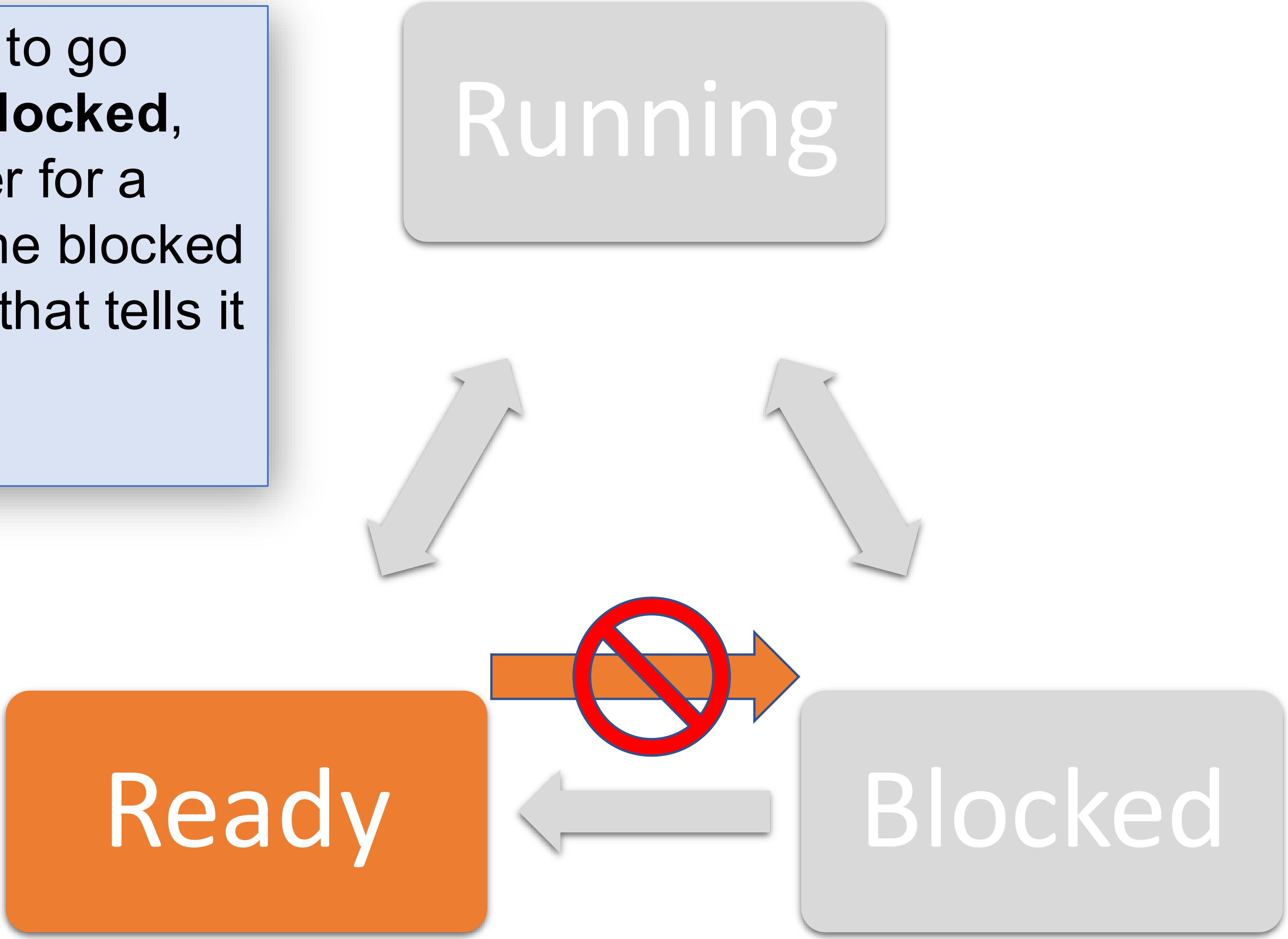
Thread States

If the event the thread is waiting for happens, but the thread can't run yet, it switches to **ready**.



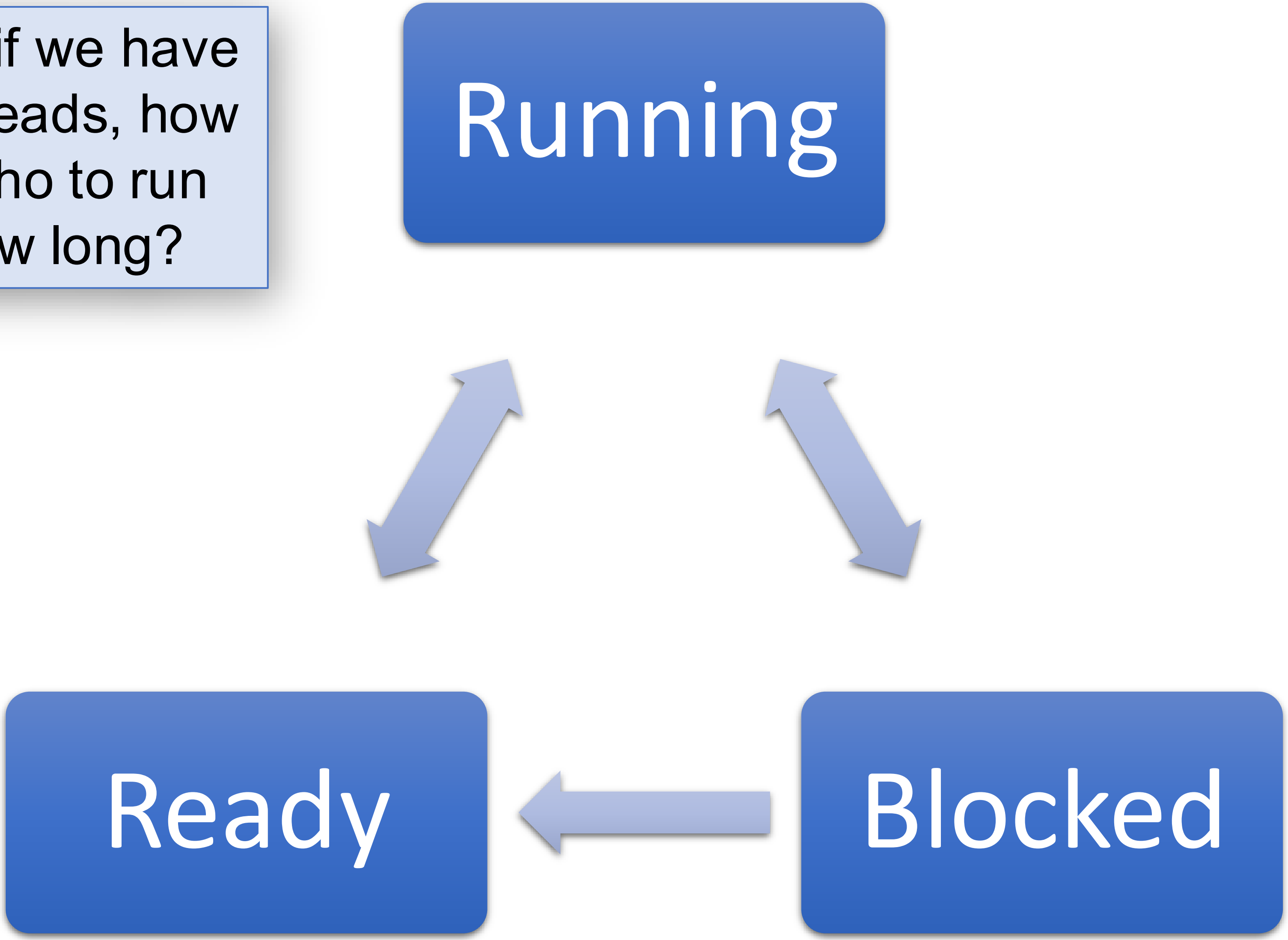
Thread States

It’s not possible to go from **ready** to **blocked**, because in order for a thread to become blocked it must do work that tells it it must wait for something.



Thread States

Key question: if we have many **ready** threads, how do we decide who to run next, and for how long?



Plan For Today

- **Recap:** Dispatching
- Scheduling and Thread States
- **Approach #1: First-Come First-Serve**
- Approach #2: Round Robin
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

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!

Plan For Today

- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- **Approach #2: Round Robin**
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

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

Plan For Today

- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- **What makes a scheduling algorithm "good"?**
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

Scheduling Algorithms

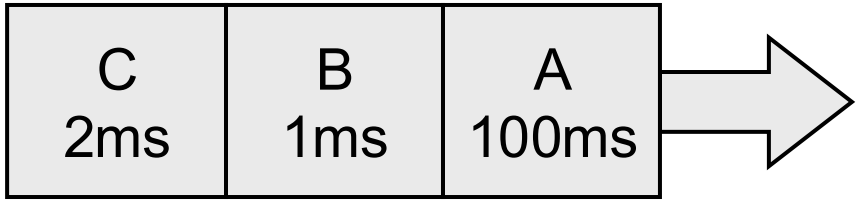
How do we decide whether a scheduling algorithm is good?

- 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

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

Ready Queue



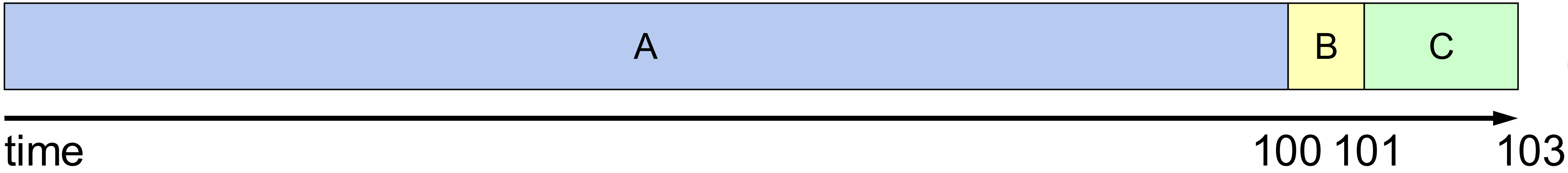
C 2ms

B 1ms

A 100ms

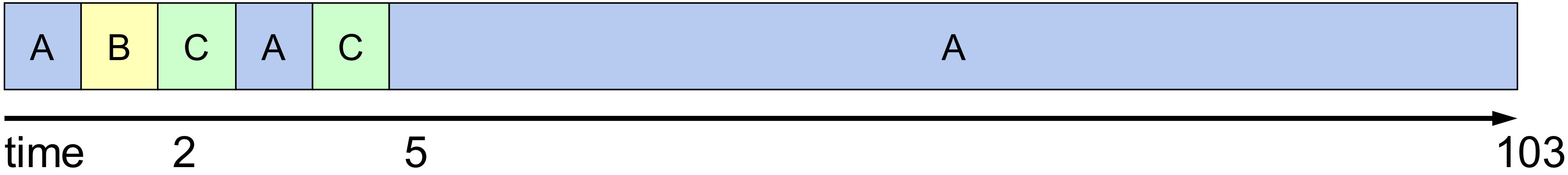
Completion timestamp

FIFO



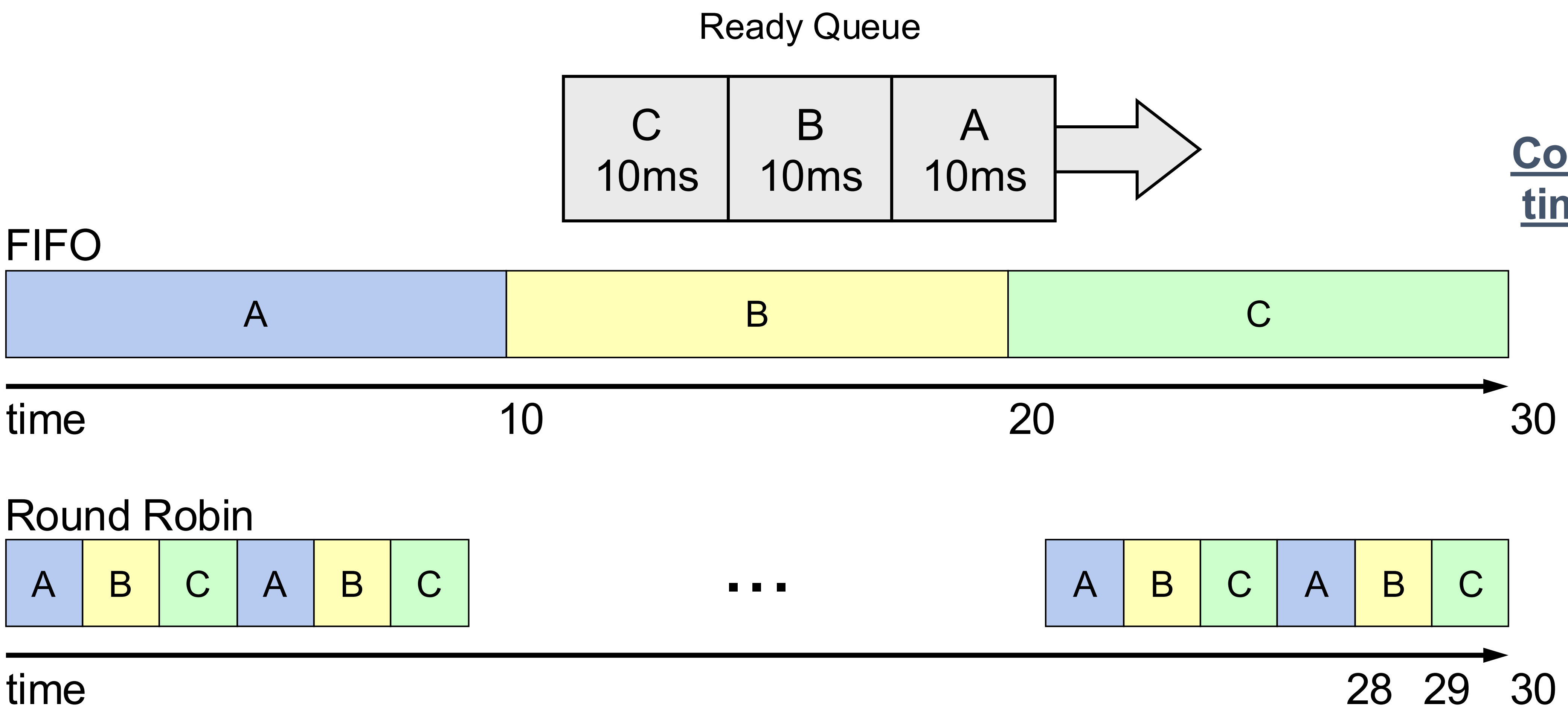
Avg: 101.3

Round Robin



Avg: 36.7

Comparing FCFS/RR: Scenario 2



Ready Queue

C 10ms | B 10ms | A 10ms

FIFO

A | B | C

time 10 20 30

Round Robin

A | B | C | A | B | C … A | B | C | A | B | C

time 28 29 30

Completion timestamp

Avg: 20

Avg: 29

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

Plan For Today

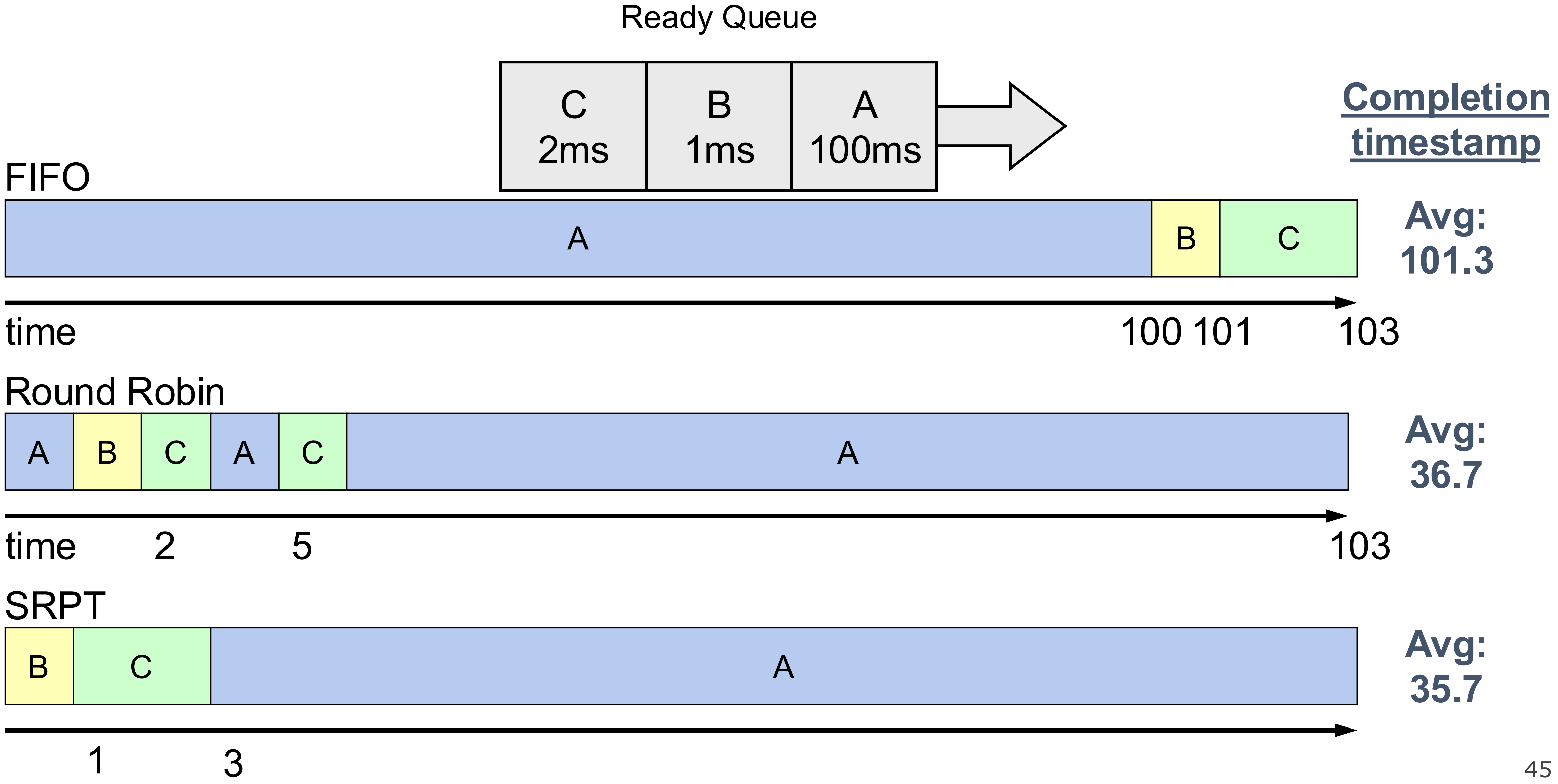
- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm "good"?
- **Approach #3: Shortest Remaining Processing Time**
- Approach #4: Priority-Based Scheduling

Shortest Remaining Processing Time

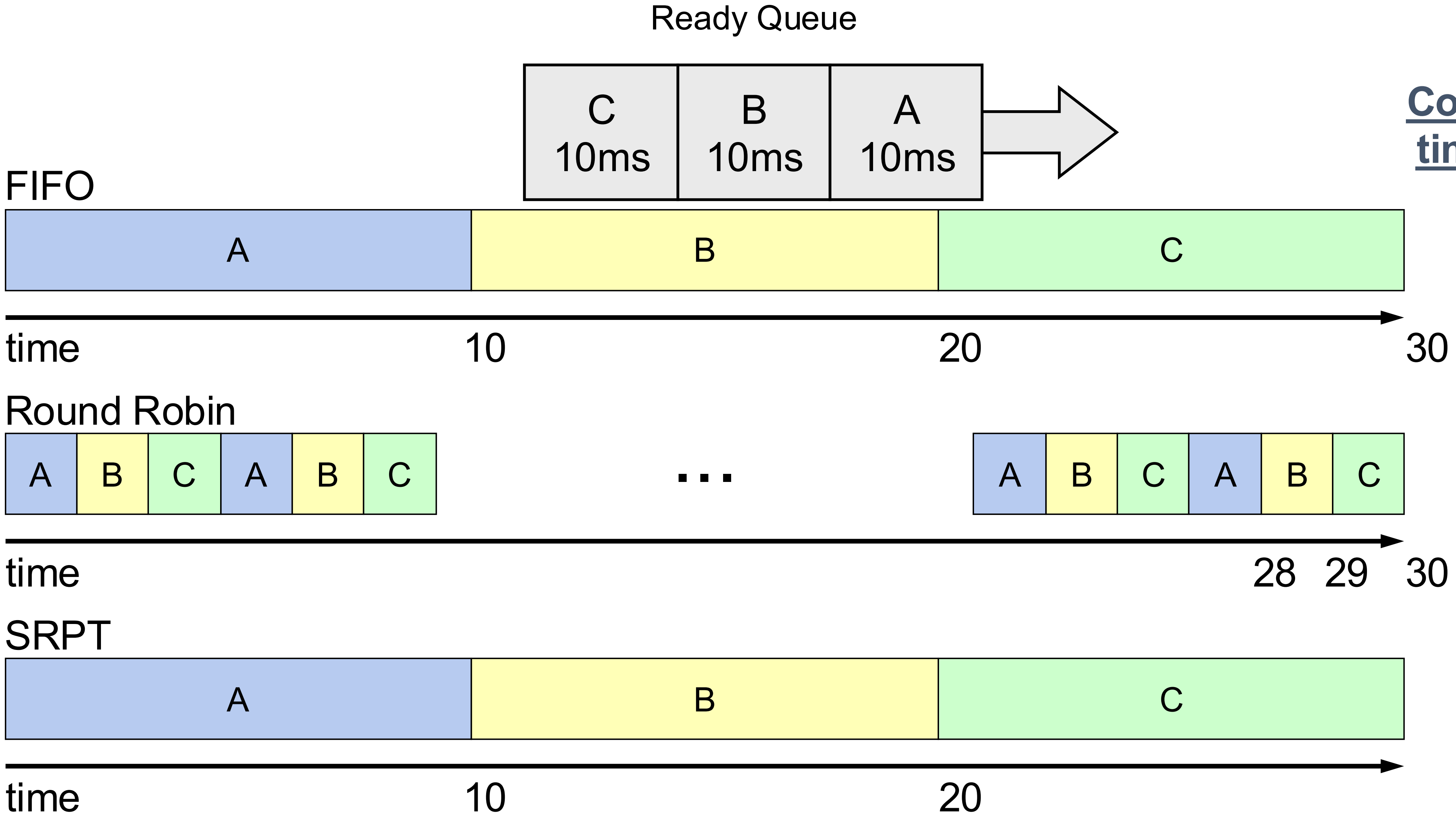
What would it look like if we optimized for completion time? (time to finish, or time to block).

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



Ready Queue

C 10ms | B 10ms | A 10ms

FIFO

A | B | C

time 10 20 30

Round Robin

A B C A B C … A B C A B C

time 28 29 30

SRPT

A | B | C

time 10 20

Completion timestamp

Avg: 20

Avg: 29

Avg: 20

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 on PollEv:
pollev.com/cs111

What are some problems/challenges with the SRPT approach?

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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?

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

SRPT

Another advantage of SRPT: improves overall resource utilization

- 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**.
    - “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
    - E.g. with I/O-bound thread wanting 1ms before reading from disk vs. CPU-bound thread wanting 100ms, we will run I/O bound thread first.

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

Plan For Today

- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm “good”?
- Approach #3: Shortest Remaining Processing Time
- **Approach #4: Priority-Based Scheduling**

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!

Alternate approach: 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"

Scheduling

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

We discussed 4 main designs:

1. **First-come-first-serve (FIFO / FCFS):** keep threads in ready queue, add threads to the back, run thread from front until completion or blocking.
2. **Round Robin:** run thread for one time slice, then add to back of queue if wants more time
3. **Shortest Remaining Processing Time (SRPT):** pick the thread that will complete or block the soonest and run it to completion.
4. **Priority-Based Scheduling:** threads have priorities, and we have one ready queue per priority. Threads adjust priorities based on time slice usage, or based on recent CPU usage (4.4 BSD Unix)

Recap

- **Recap:** Dispatching
- Scheduling and Thread States
- Approach #1: First-Come First-Serve
- Approach #2: Round Robin
- What makes a scheduling algorithm "good"?
- Approach #3: Shortest Remaining Processing Time
- Approach #4: Priority-Based Scheduling

Lecture 18 takeaway: For scheduling, we want to minimize response time, use resources efficiently, and be fair. SRPT is the best to minimize average response time, but we can only approximate it due to needing to predict the future.

Next time: preemption