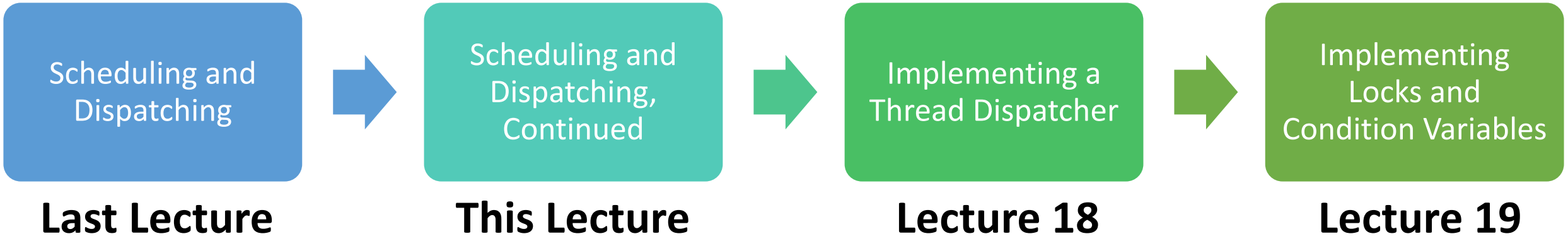
CS111, Lecture 17
Scheduling and Dispatching, Continued

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

- Understand the general mechanisms for switching between threads
- Explore the tradeoffs in deciding which threads get to run and for how long

Plan For Today

- **Recap:** Process Control Blocks and Threads
- **Demo:** Context Switch
- **Recap:** Thread States
- Scheduling Threads

```
cp -r /afs/ir/class/cs111/lecture-code/lect17 .
```

Plan For Today

- **Recap: Process Control Blocks and Threads**
- **Demo:** Context Switch
- **Recap:** Thread States
- Scheduling Threads

```
cp -r /afs/ir/class/cs111/lecture-code/lect17 .
```

Process and Thread State

Key question: How does OS track info about threads and processes?

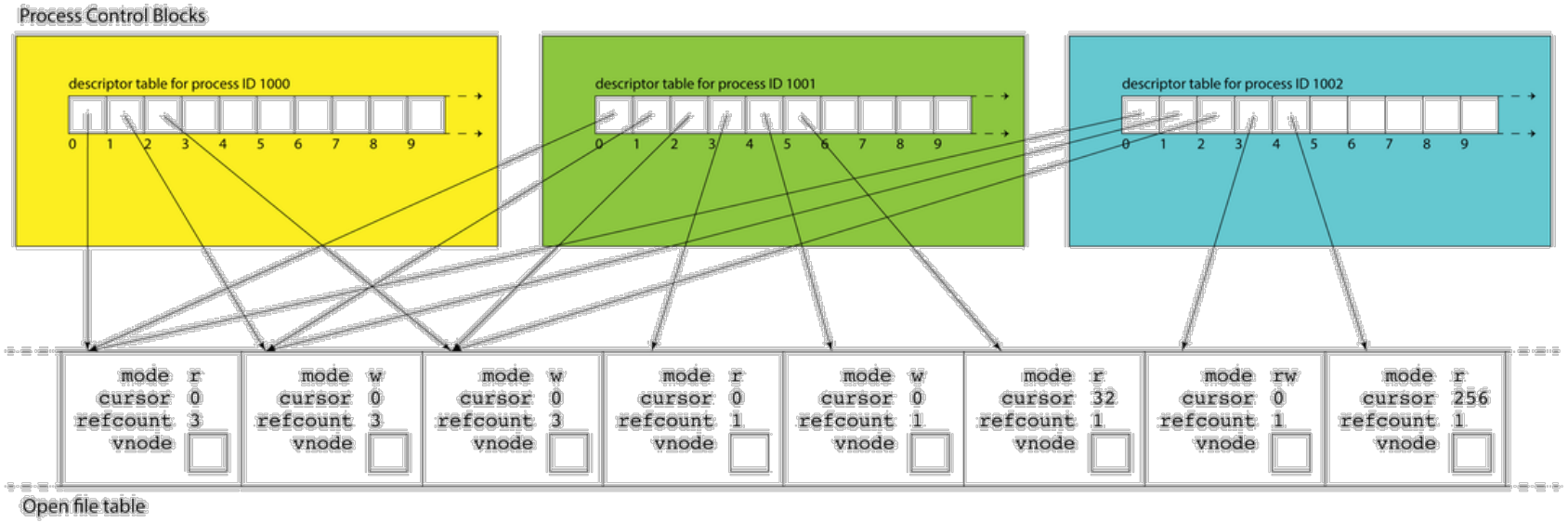
The OS maintains a **process control block** for each process - a set of relevant information about its execution. Lives as long as the process does.

- Information about memory used by this process
- File descriptor table
- Other misc. accounting and info
- State specific to each thread

File Descriptor Table

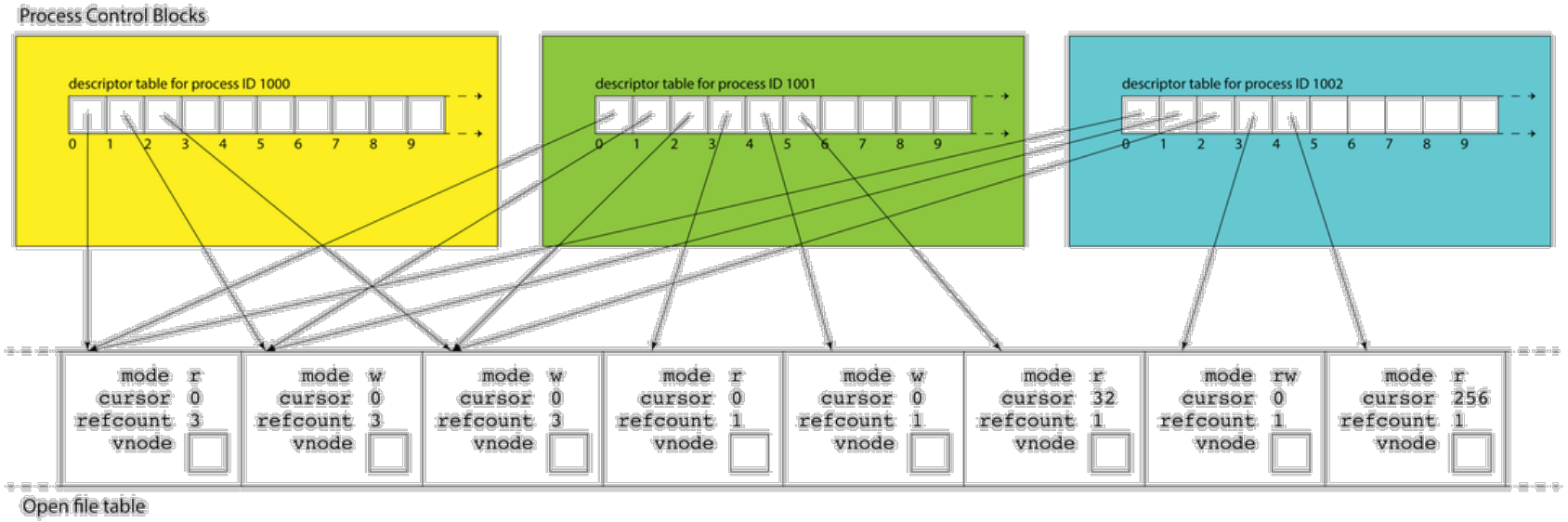
The file descriptor table is an array of info about open files/resources for this process. **Key idea: a** file descriptor is just **an index into the file descriptor table!**

- An entry in the file descriptor table is really a *pointer* to an entry in another table, the **open file table**.
- The **open file table** is one array of information about open files/resources across all processes.



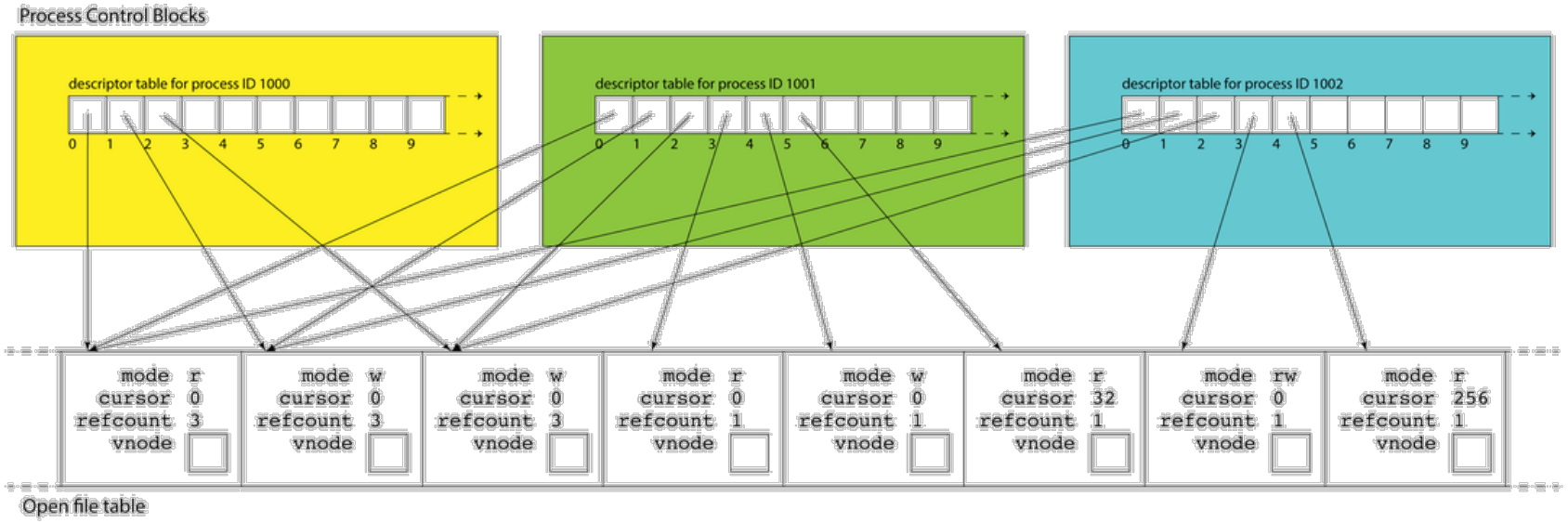
Open File Table

- When we call **open** in our program; new open file table entry created, new file descriptor index points to it.
- When we call **pipe** in our program; 2 new open file table entries created, 2 new file descriptor indexes point to them.
- When we call **fork** in our program; new PCB, with copy of parent's FD table; so all file descriptor indexes point to the same place!



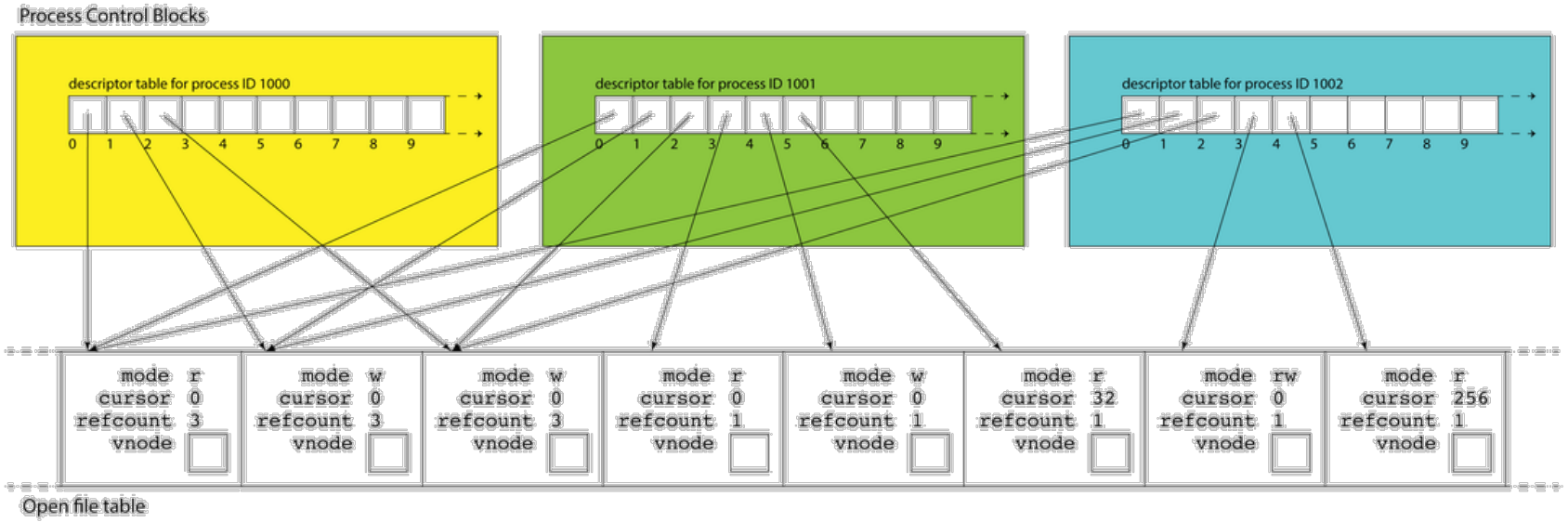
Open File Table

- When we call **dup2** in our program, one file descriptor index is copied to another index (so both point to same open file table entry)



Open File Table

- Each open file table entry keeps a *reference count*, a count of the number of file descriptor table entries pointing to it.
- This ref count increases whenever a new file descriptor index points to it.
- When we call **close** in our program, file descriptor index no longer points to open file table entry, open file table entry's ref count decremented.
- When open file table entry's ref count == 0, it's deleted



Parent makes a pipe, spawns a child: what is read end's ref count?

1

2

3

4

Parent makes a pipe, spawns a child: what is read end's ref count?

1

2

3

4

Parent makes a pipe, spawns a child: what is read end's ref count?

1

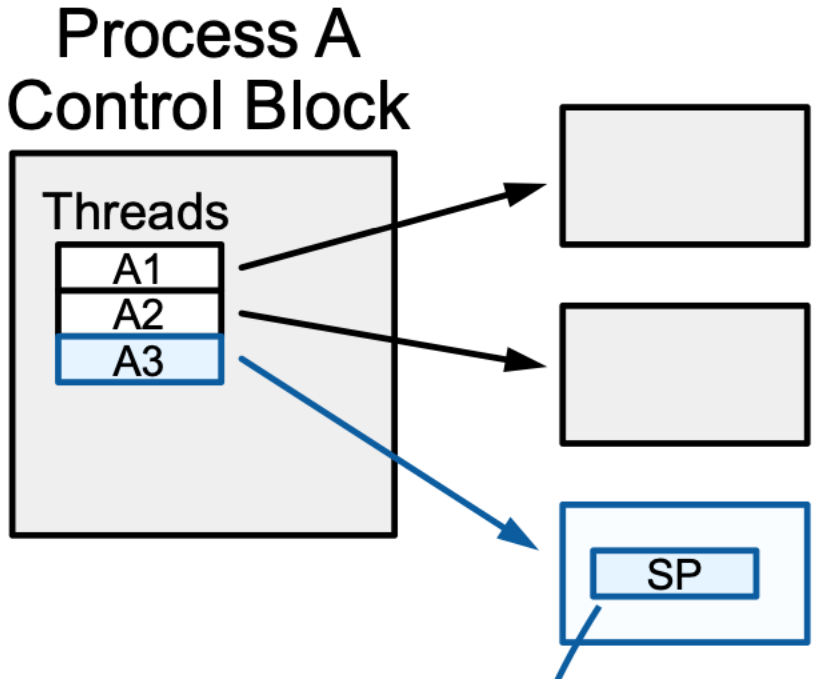
2

3

4

Thread State

- All main info in the PCB (e.g. memory info for the entire process) is relevant to all threads
- Each thread also has some of its own private info (e.g. stack location)
- When the OS wants to run a thread, it loads its state and starts or resumes it
- How does the OS regain control?



Regaining Control

There are several ways control can switch back to the OS:

1. "Traps" (events that require OS attention):
    1. System calls (like **read** or **waitpid**)
    2. Errors (illegal instruction, address violation, etc.)
    3. Page fault (accessing memory that must be loaded in) – more later…
2. "Interrupts" (events occurring outside current thread):
    1. Character typed at keyboard
    2. Completion of disk operation
    3. Timer – to make sure OS eventually regains control

At this point, OS could then decide to run a different thread.

Switching Between Threads

When the OS regains control, how does it switch to run another thread?

- **Key idea:** we must load the thread's state onto a processor core and run it
- State = **registers**
  - Registers store data being manipulated by the core
  - %rsp stores current top of stack
  - Stack remembers what function to continue executing
- If we can load this thread's %rsp + other saved registers, then it can resume right where it left off
- *Context switch* – changing the thread currently running to another thread. We must save the current thread state and load in the new thread state.

Switching Between Threads

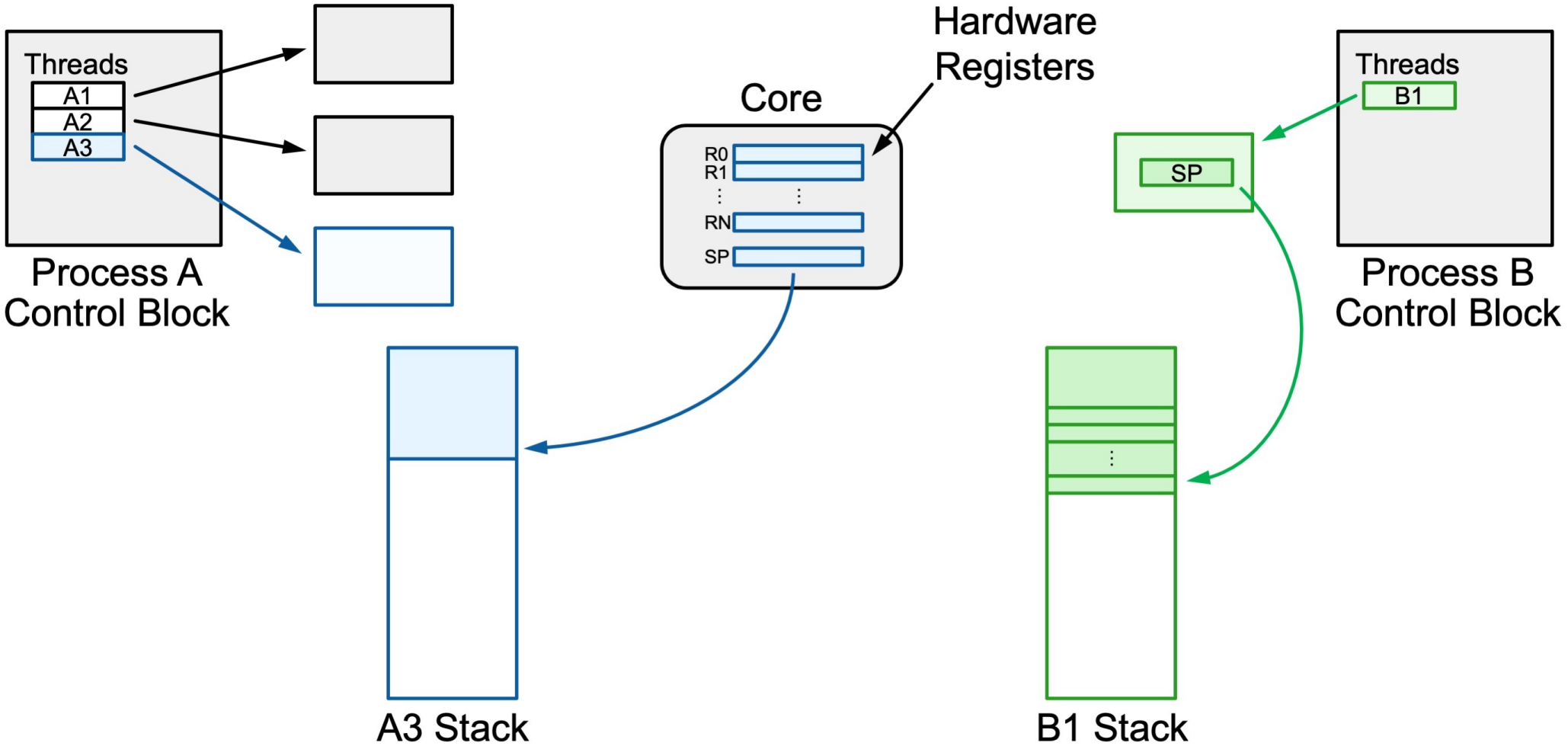
When the OS regains control, how does it switch to run another thread?

The **dispatcher** is OS code that runs on each core that switches between threads and does context switches

- You're going to write parts of one on assign5!
- Context switches are funky – like running a function that, as part of its execution, switches to a *completely different function in a completely different thread!!*

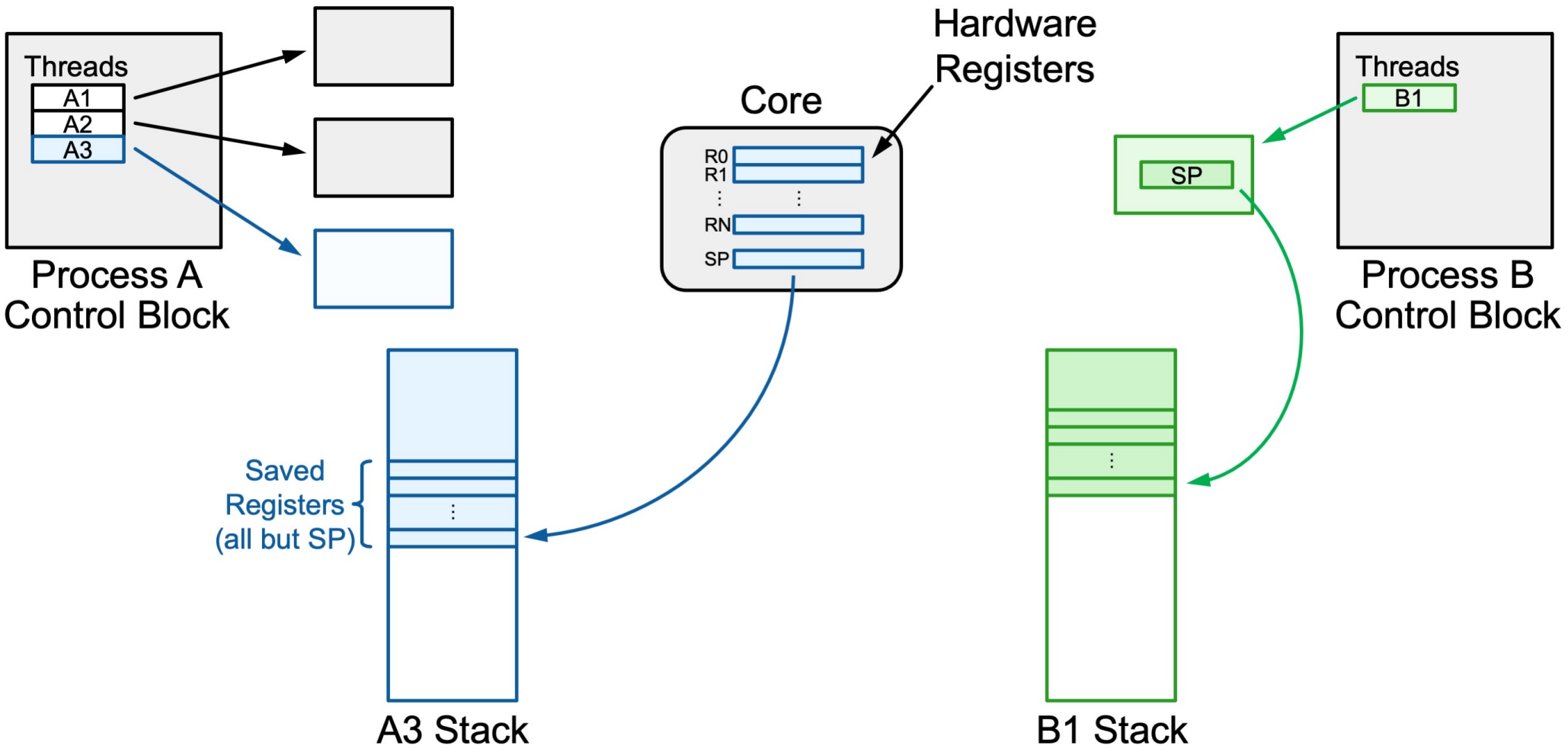
Context Switching

*Context switch: h*ow do we switch from thread A3 to thread B1?



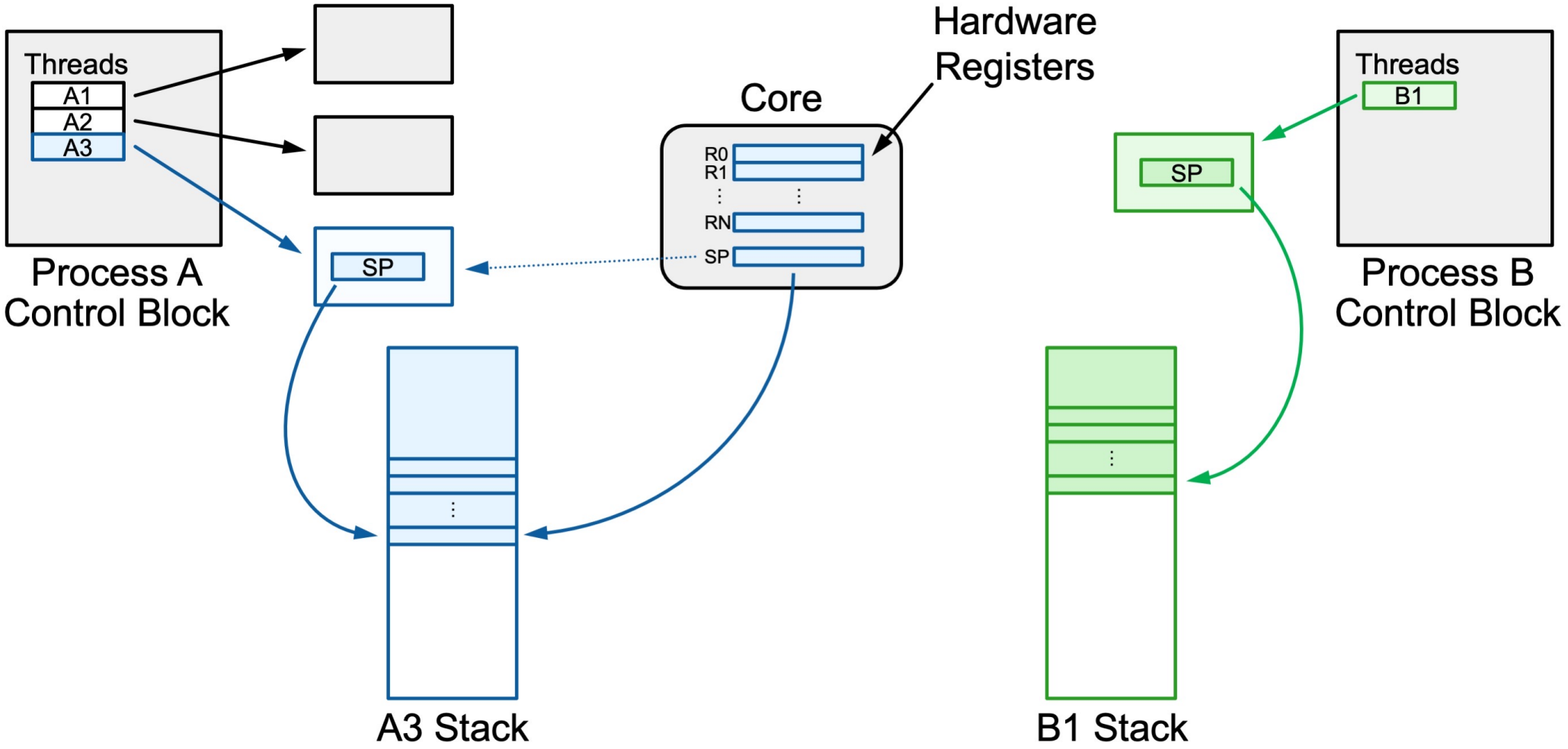
Context Switching

Step 1: *push all registers besides stack register onto the thread's stack.*



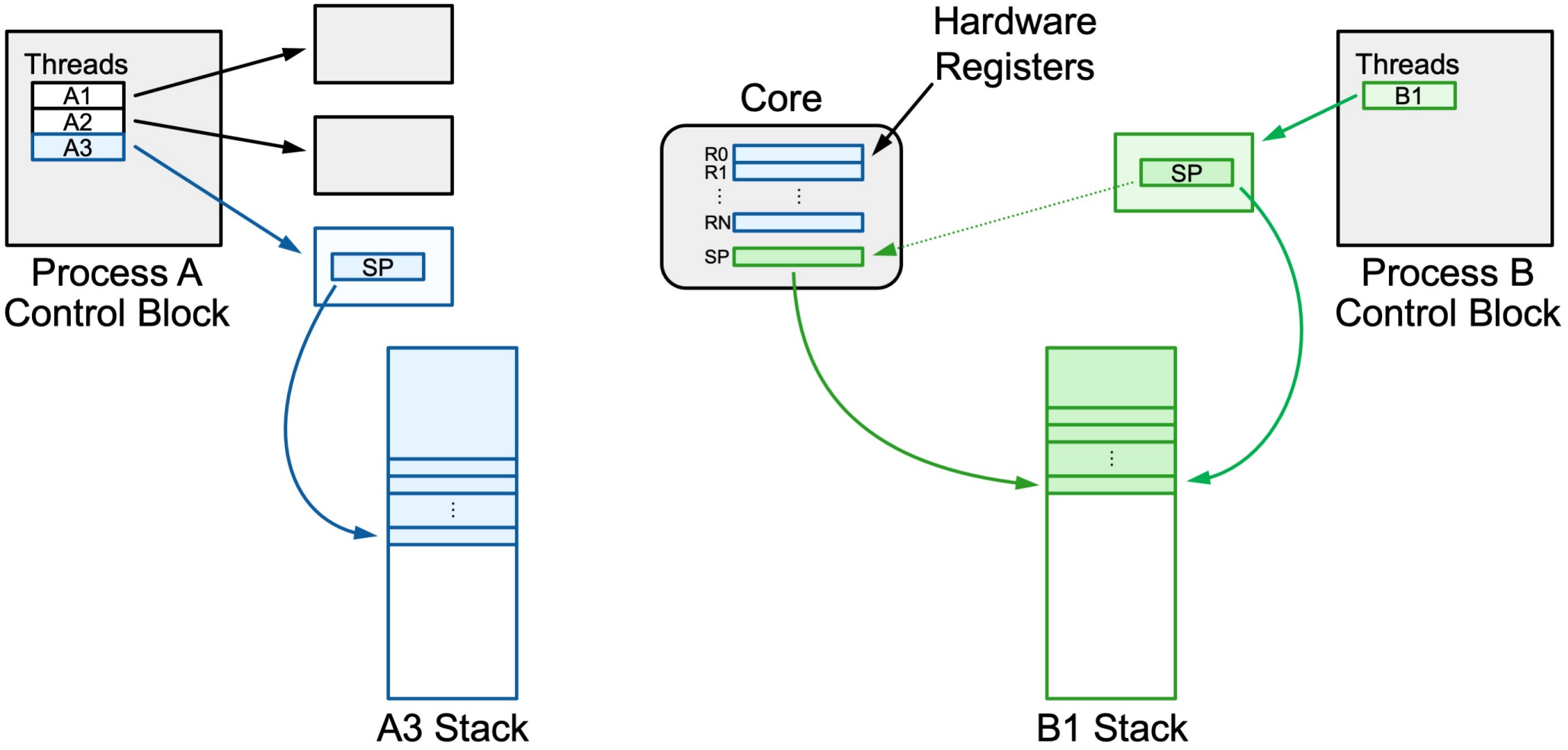
Context Switching

Step 2: *save the stack register into the thread's state space.*



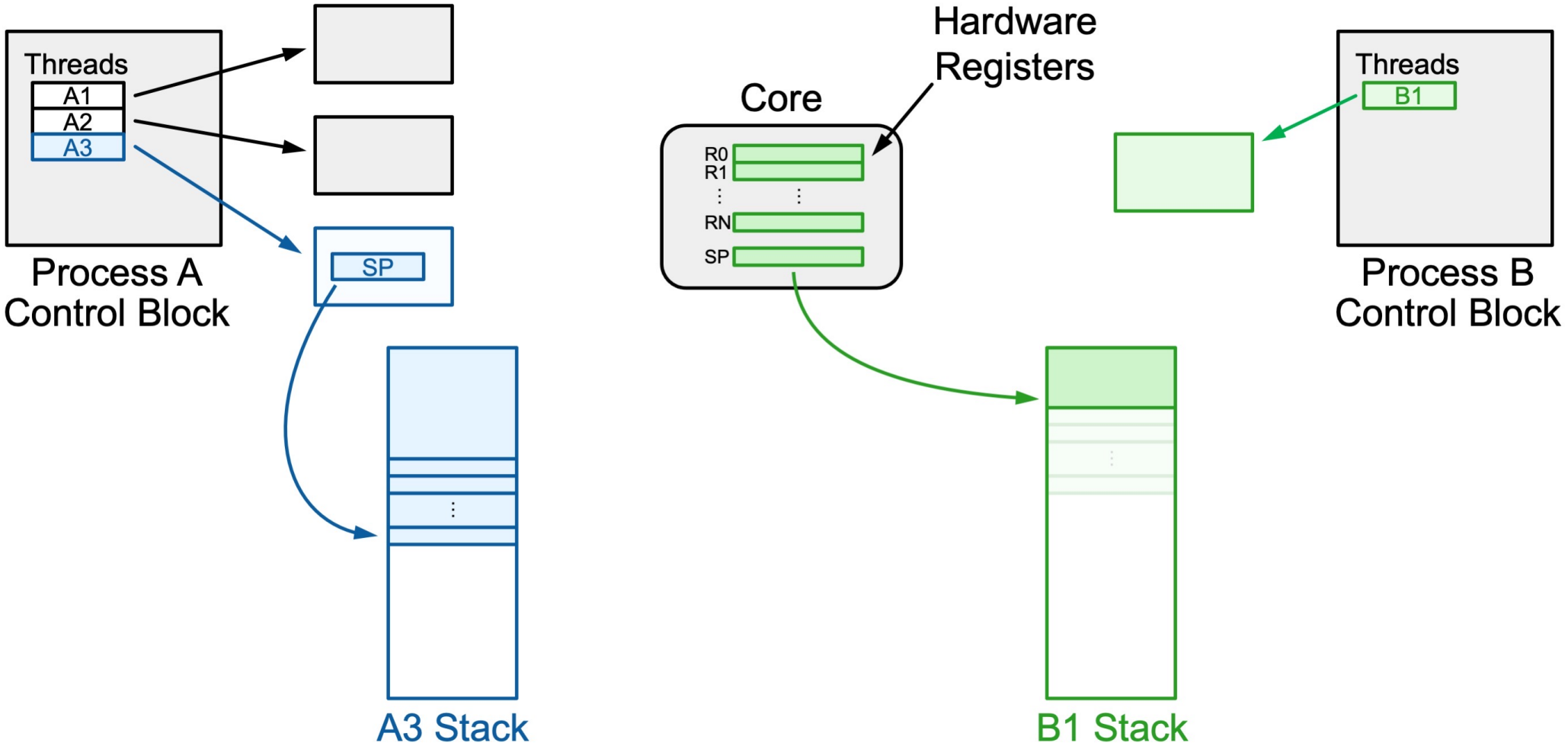
Context Switching

Step 3: *load B1's saved stack register from its thread state space.*



Context Switching

Step 4: *pop B1's other registers from its stack space.*



Demo: context-switch.cc

Plan For Today

- **Recap:** Process Control Blocks and Threads
- **Demo:** Context Switch
- **Recap: Thread States**
- Scheduling Threads

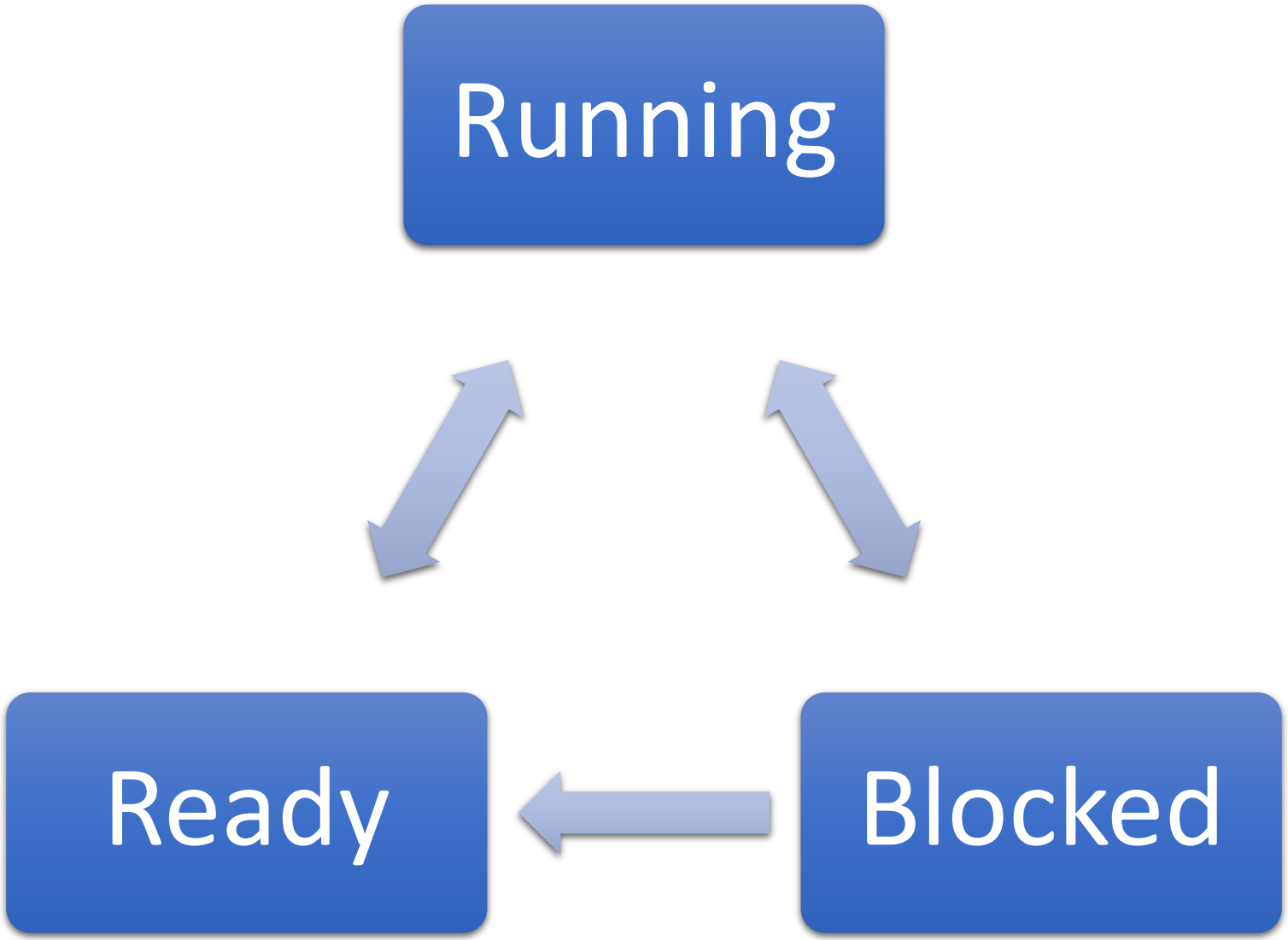
```
cp -r /afs/ir/class/cs111/lecture-code/lect17 .
```

Tracking All Threads

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

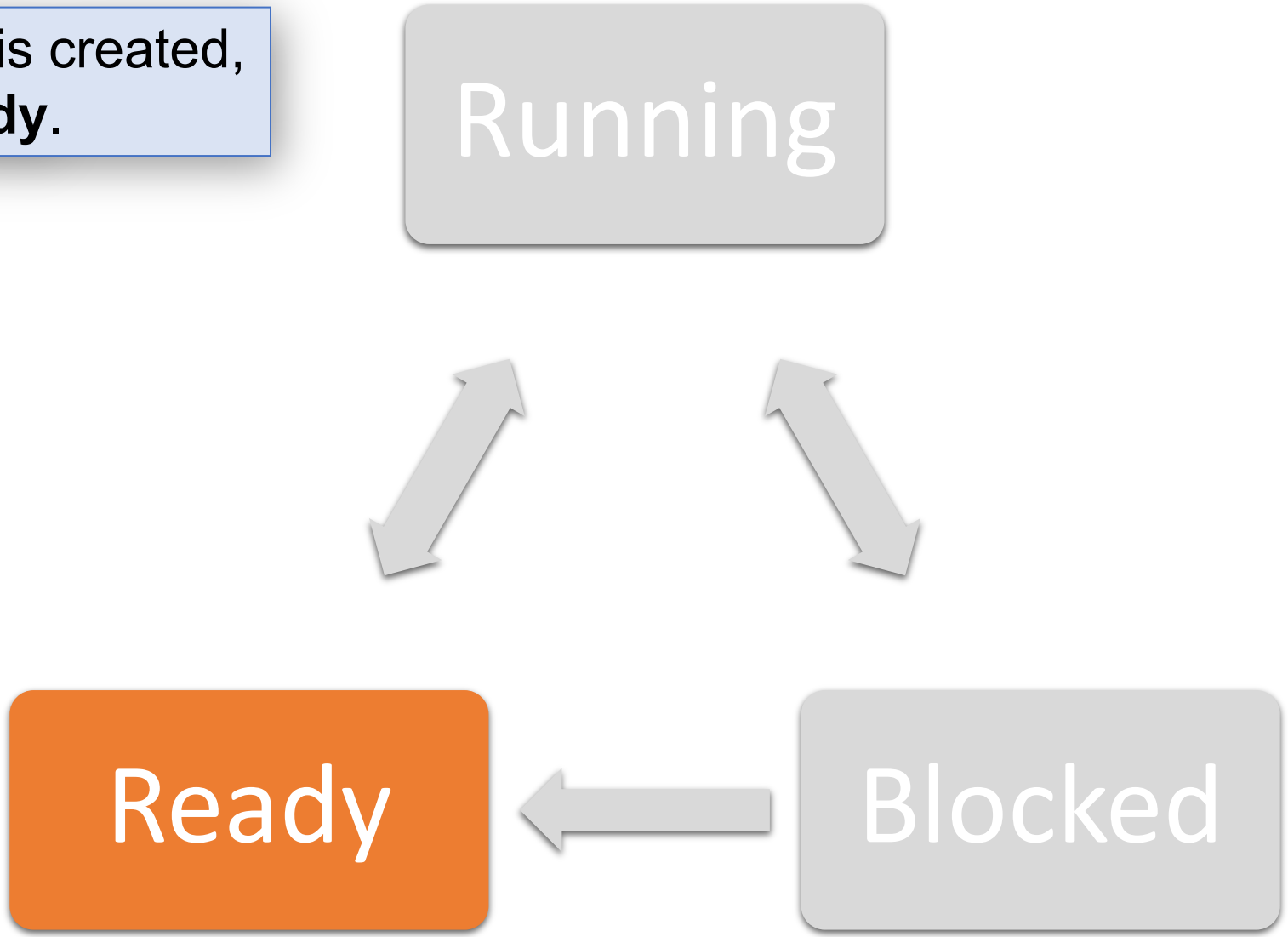
1. **Running**
2. **Blocked –** waiting for an event (disk I/O, network connection, etc.)
3. **Ready** – able to run, but waiting for CPU time

Thread States



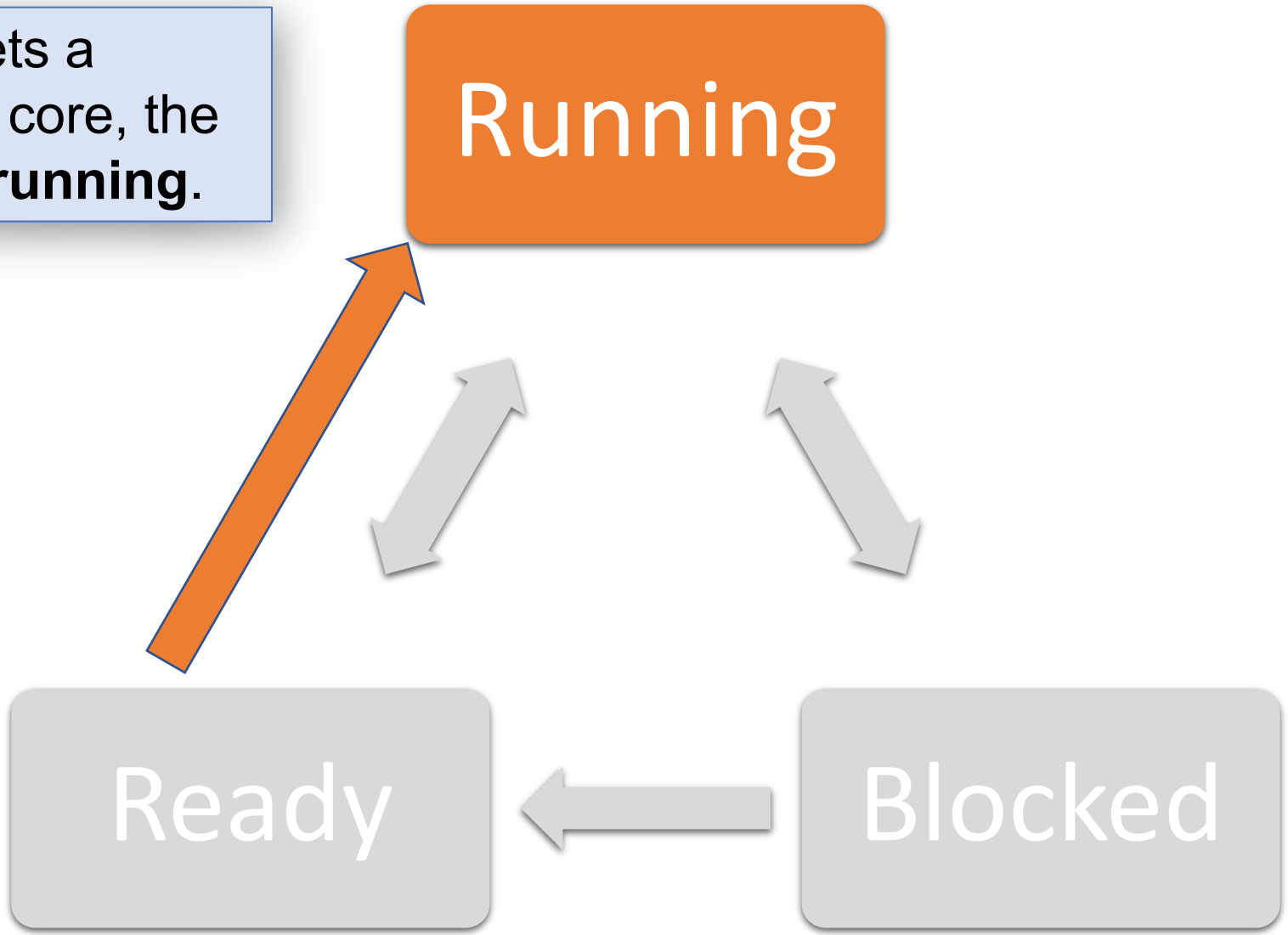
Thread States

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



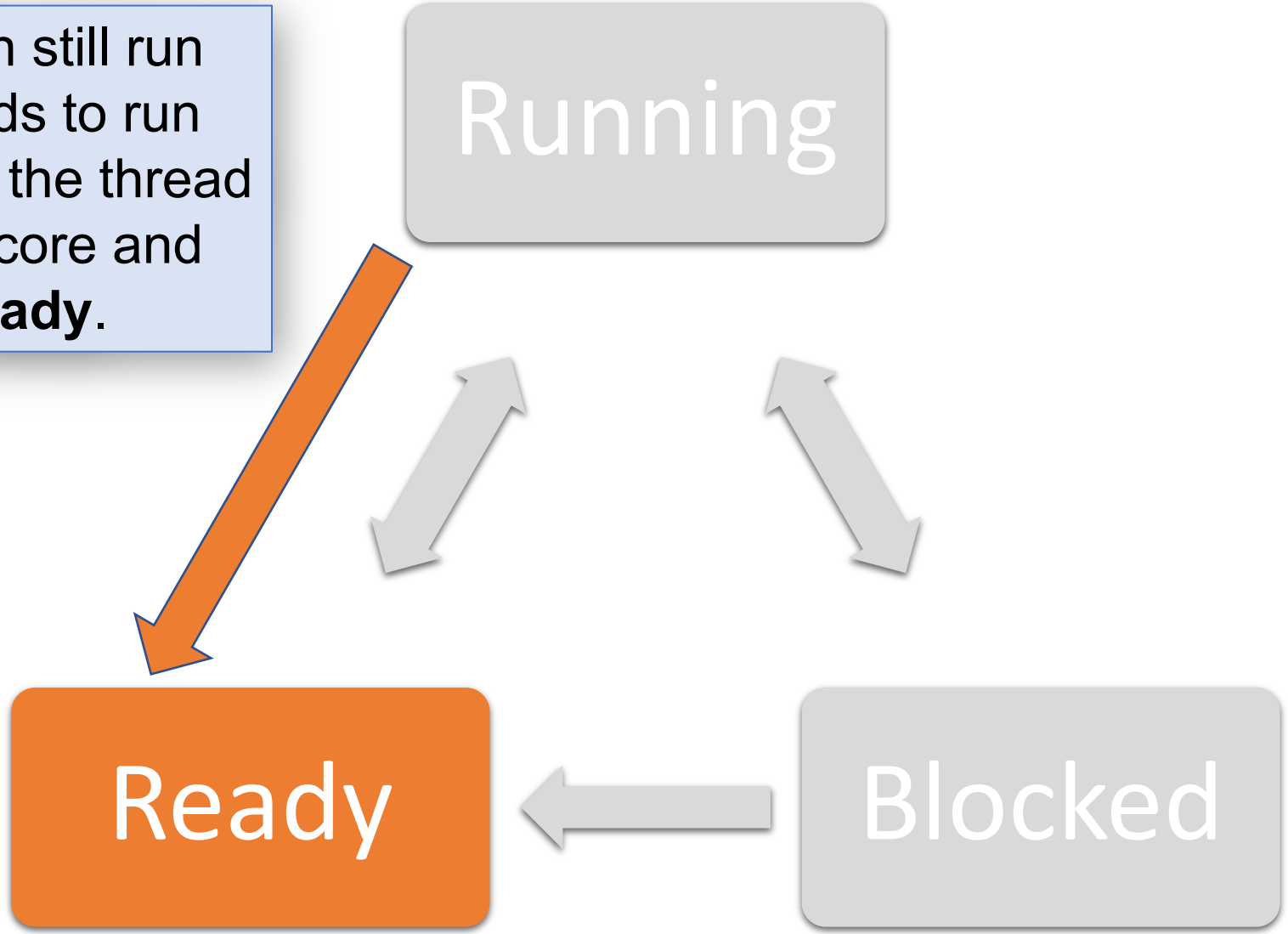
Thread States

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



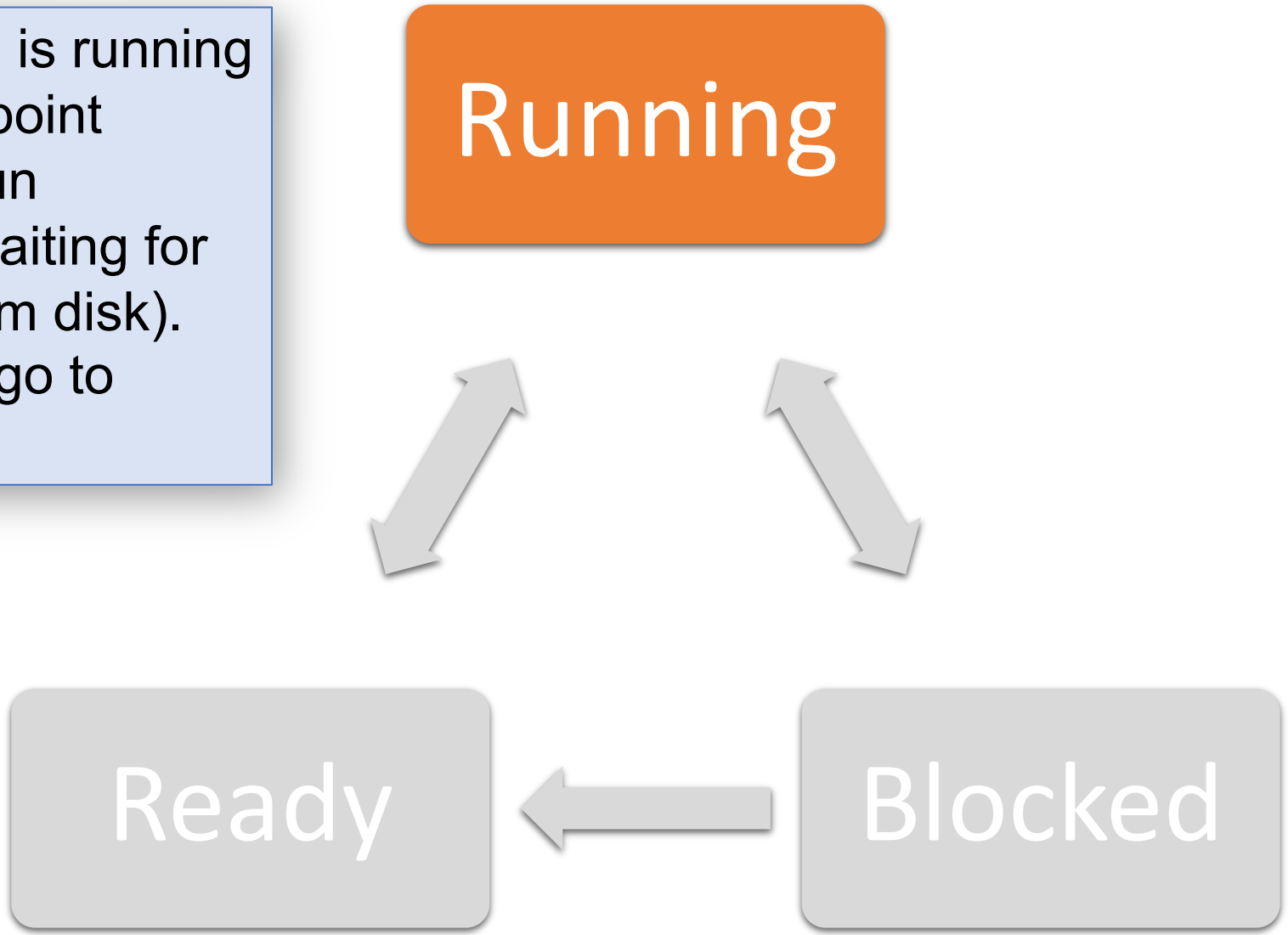
Thread States

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**.



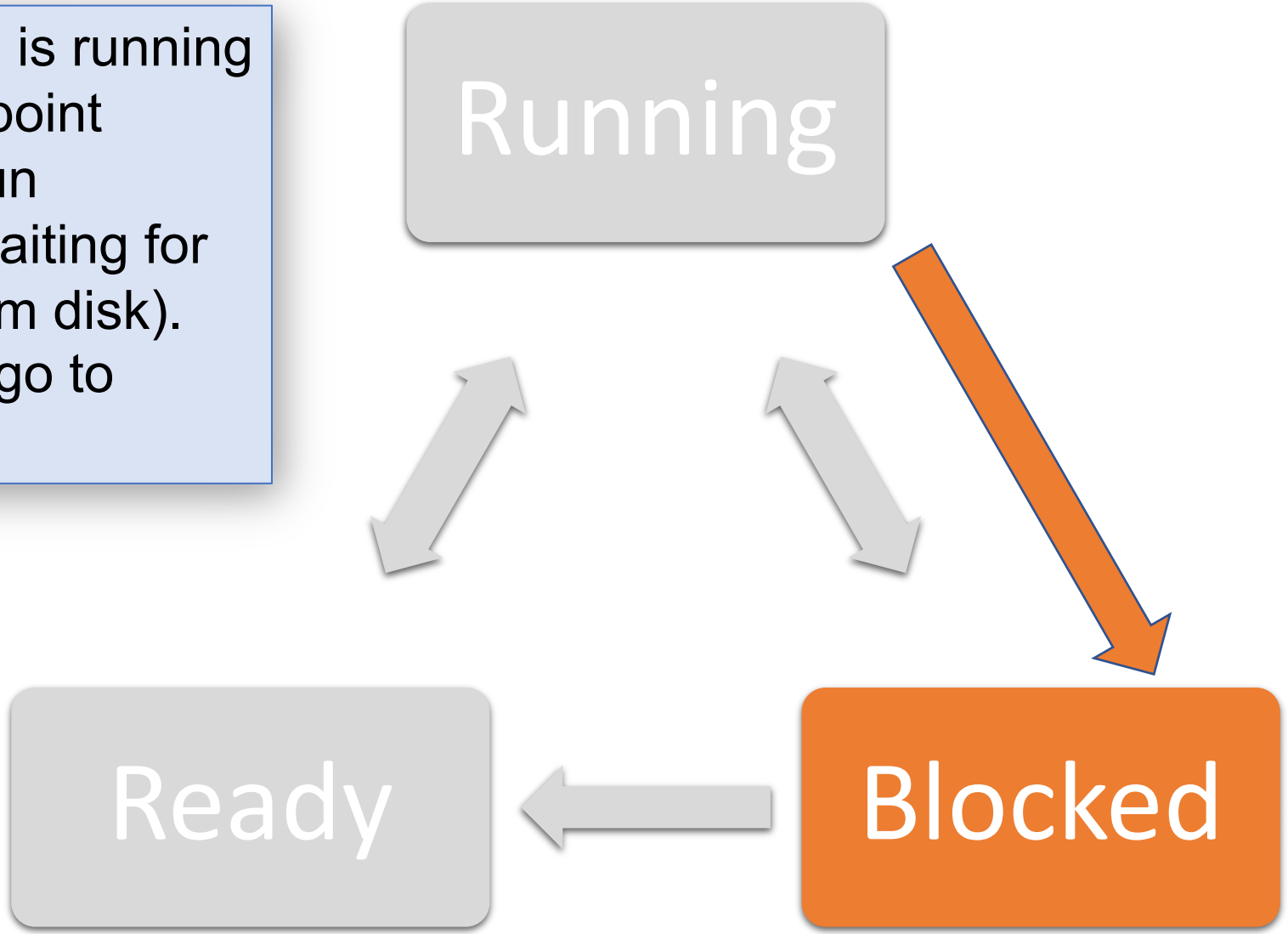
Thread States

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**.



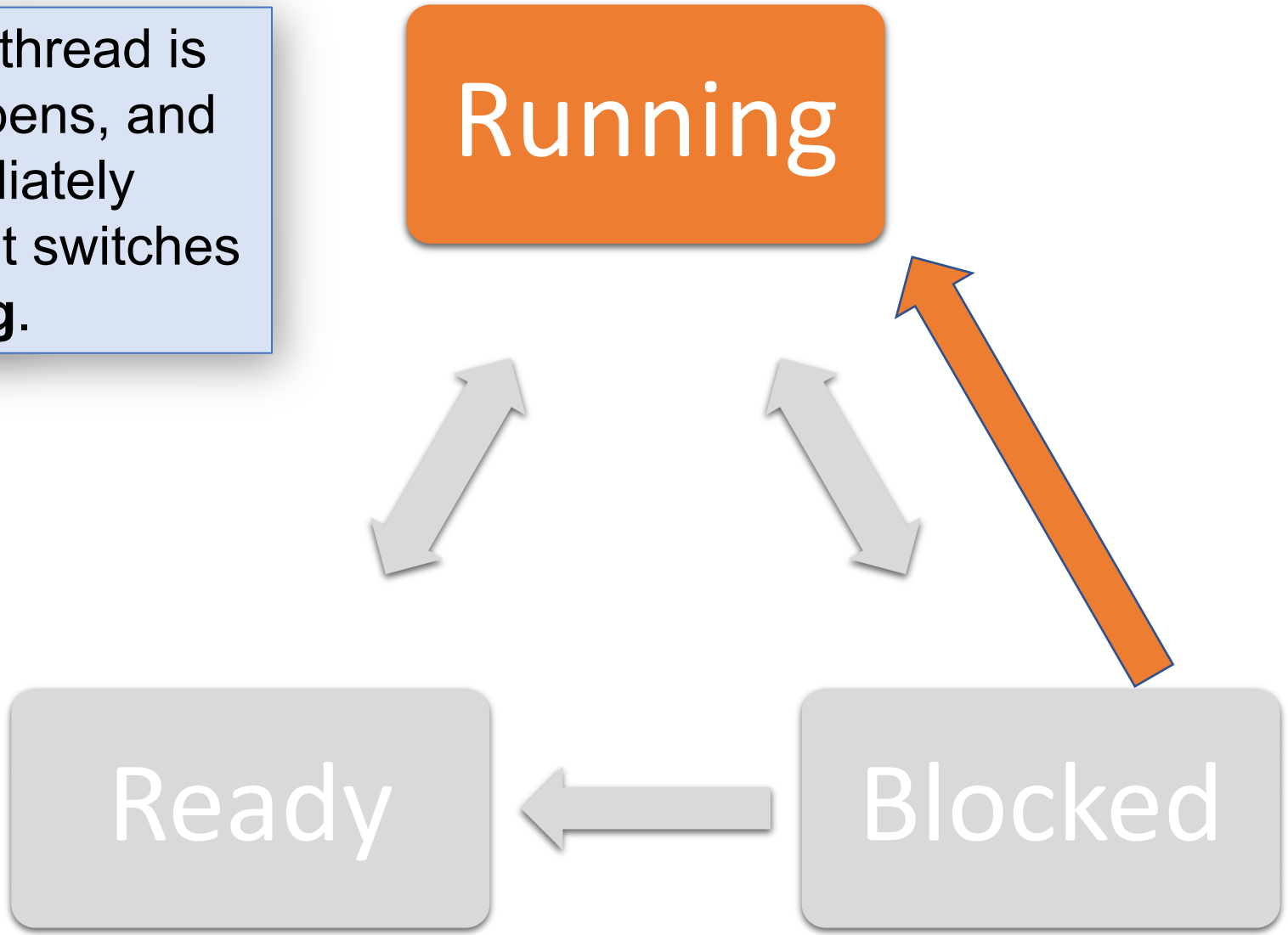
Thread States

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**.



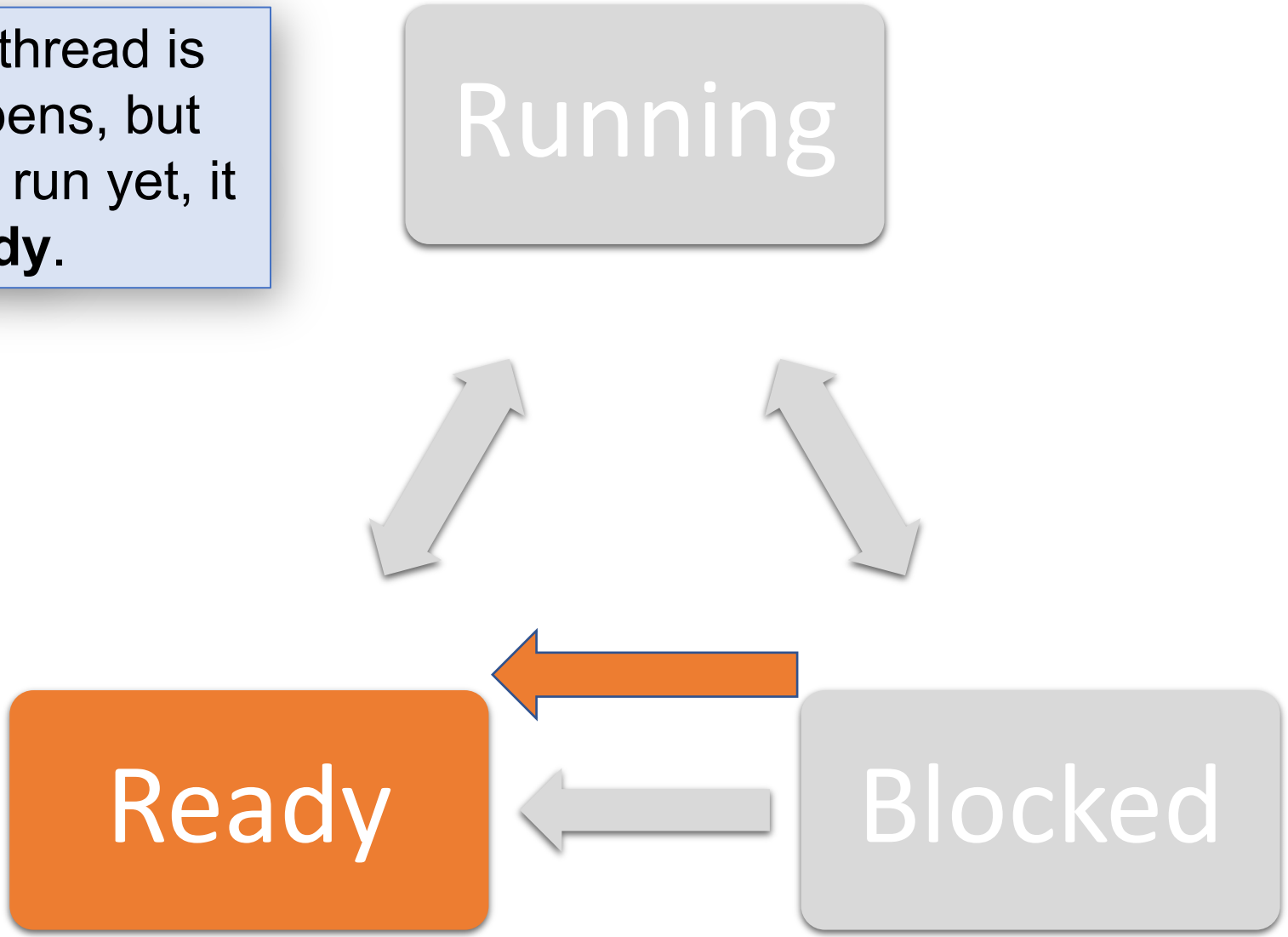
Thread States

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



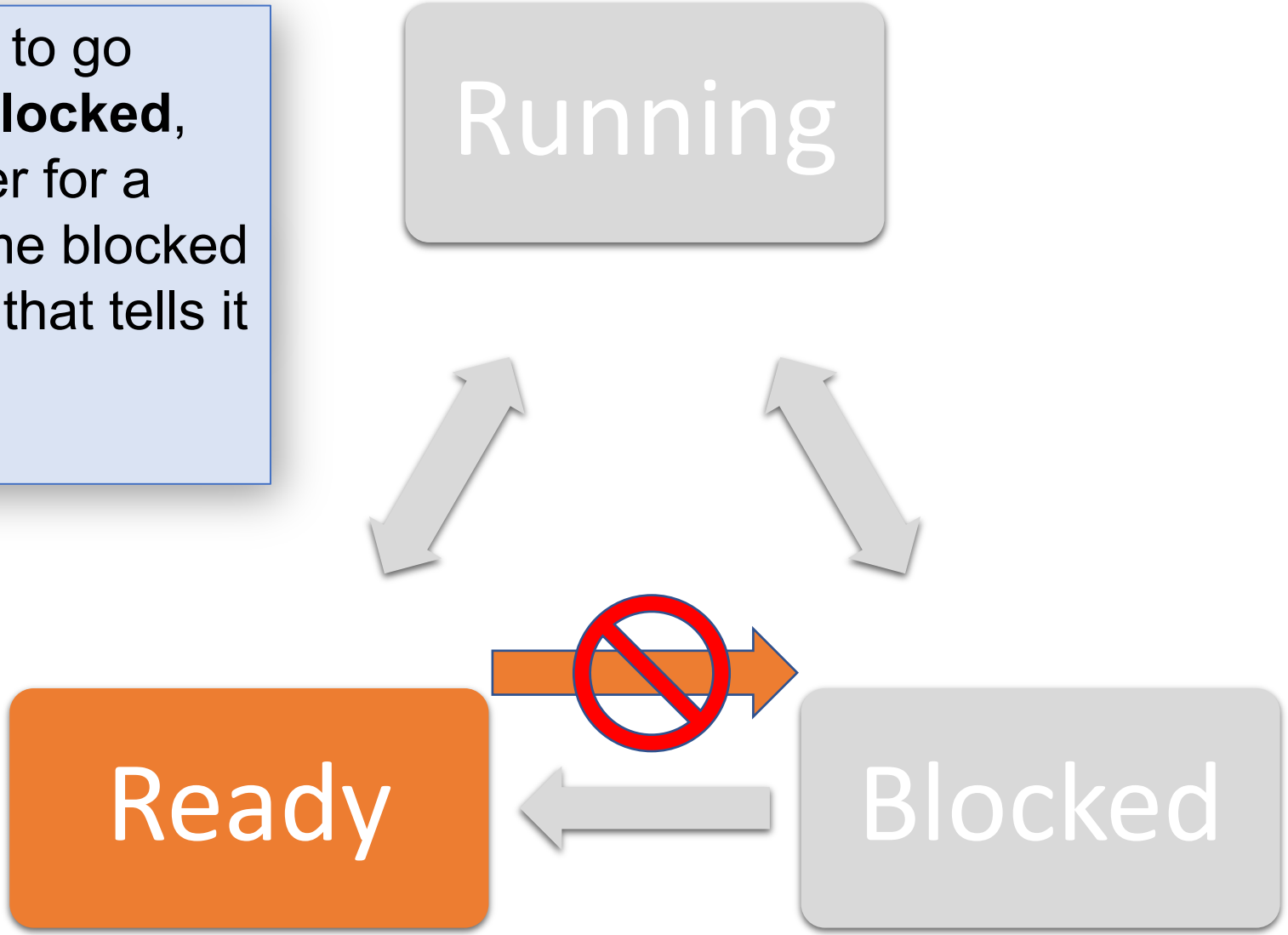
Thread States

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



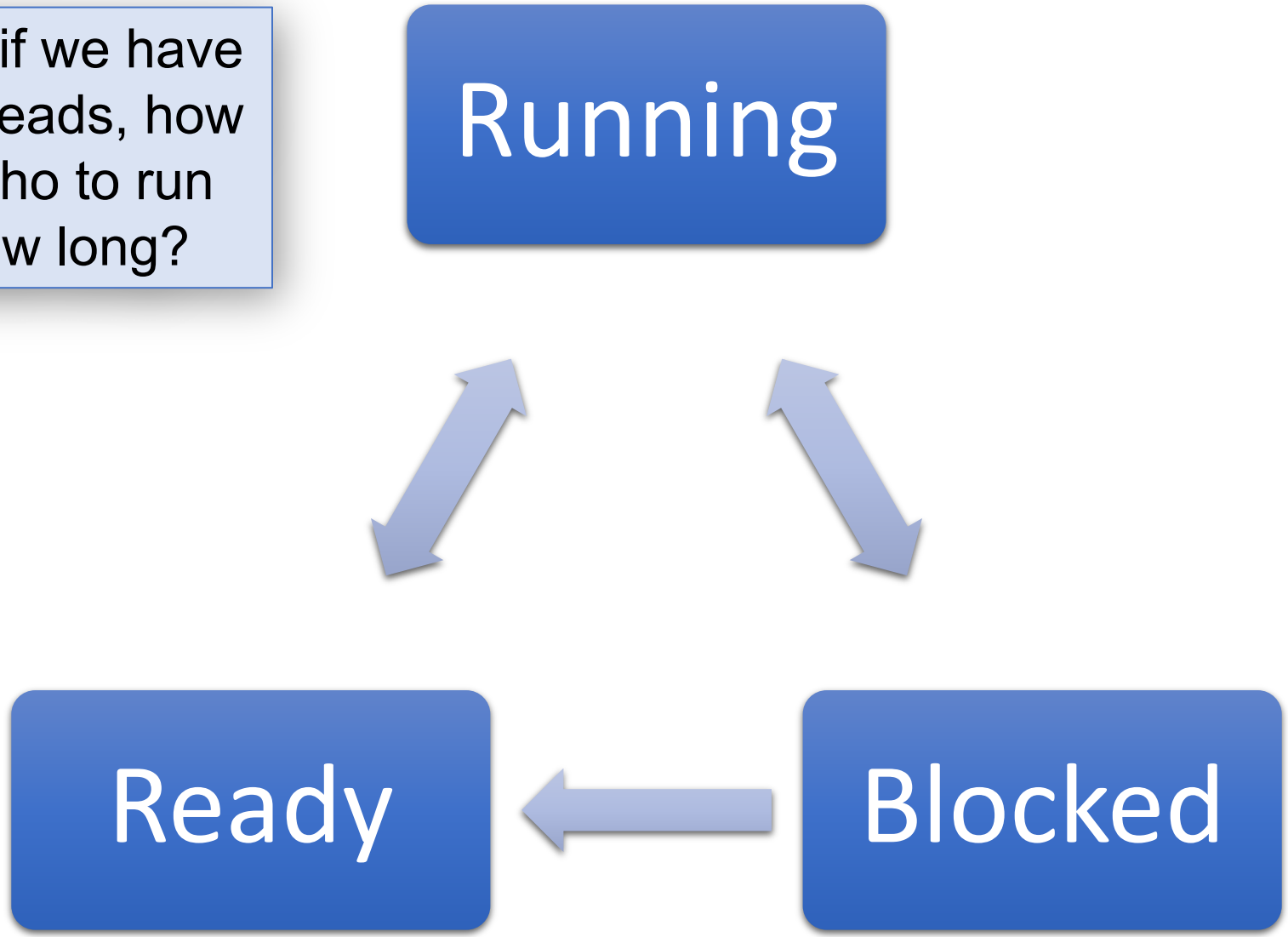
Thread States

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:** Process Control Blocks and Threads
- **Demo:** Context Switch
- **Recap:** Thread States
- **Scheduling Threads**

```
cp -r /afs/ir/class/cs111/lecture-code/lect17 .
```

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.

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?

Plan For Today

- **Recap:** Process Control Blocks and Threads
- **Demo:** Context Switch
- **Recap:** Thread States
- Scheduling Threads

Lecture 17 takeaway:
Context switching saves the current thread state and switches to a completely different stack frame! When we have many ready threads, how do we decide which gets to run, and for how long?

Next time: more discussions about scheduling, and implementing dispatchers.