CS111, Lecture 16
Scheduling and Dispatching

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 7 up through Section 7.2

😷 masks recommended

Announcements

- Congratulations on finishing the midterm!
- Assign4 YEAH session tonight 7:30-8:30PM in Gates 403
- No section this week

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

CS111 Topic 3: Multithreading

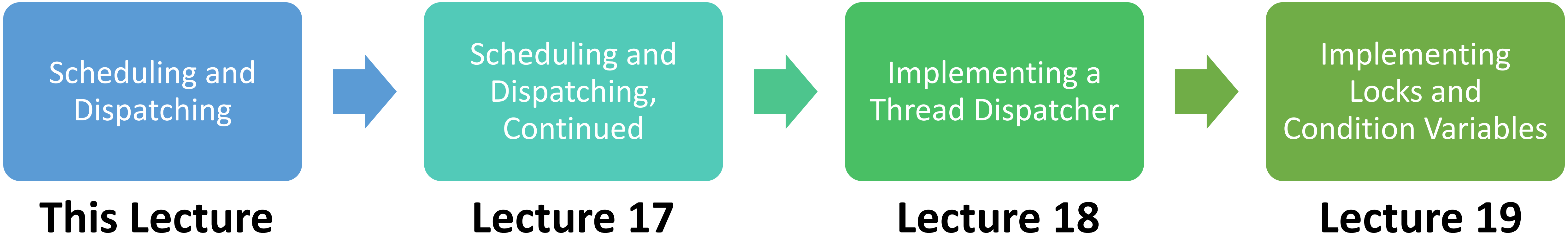
Multithreading - *How can we have concurrency within a single process? **How does the operating system support this?***

Why is answering this question important?

- Allows us to see how threads are represented and the fairness challenges for who gets to run next / for how long (next time)
- Shows us what the mechanism looks like for switching between running threads (today and next time)
- Allows us to understand how locks and condition variables are implemented (next week)

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

CS111 Topic 3: Multithreading, Part 2



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

Learning Goals

- Learn about how the operating system keeps track of threads and processes
- Understand the general mechanisms for switching between threads and when switches occur

Plan For Today

- **Overview:** Scheduling and Dispatching
- Process and Thread State
- Running a Thread
- Switching Between Threads
- Tracking All Threads

Plan For Today

- **Overview: Scheduling and Dispatching**
- Process and Thread State
- Running a Thread
- Switching Between Threads
- Tracking All Threads

Scheduling And Dispatching

- So far, we have learned about how user programs can create new processes and spawn threads in those processes
- But how does the operating system manage all of this internally? When we spawn a new thread or create a new process, what happens?

Key questions we will answer:

- How does the operating system track info for threads and processes?
- How does the operating system run a thread and switch between threads?
- How does the operating system decide which thread to run next?

Plan For Today

- **Overview:** Scheduling and Dispatching
- **Process and Thread State**
- Running a Thread
- Switching Between Threads
- Tracking All Threads

Process and Thread State

Key question #1: How does the operating system track info about threads and processes?

The OS maintains a (private) **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
- Info about threads in this process
- Other misc. accounting and info

Process and Thread State

Key question #1: How does the operating system track info about threads and processes?

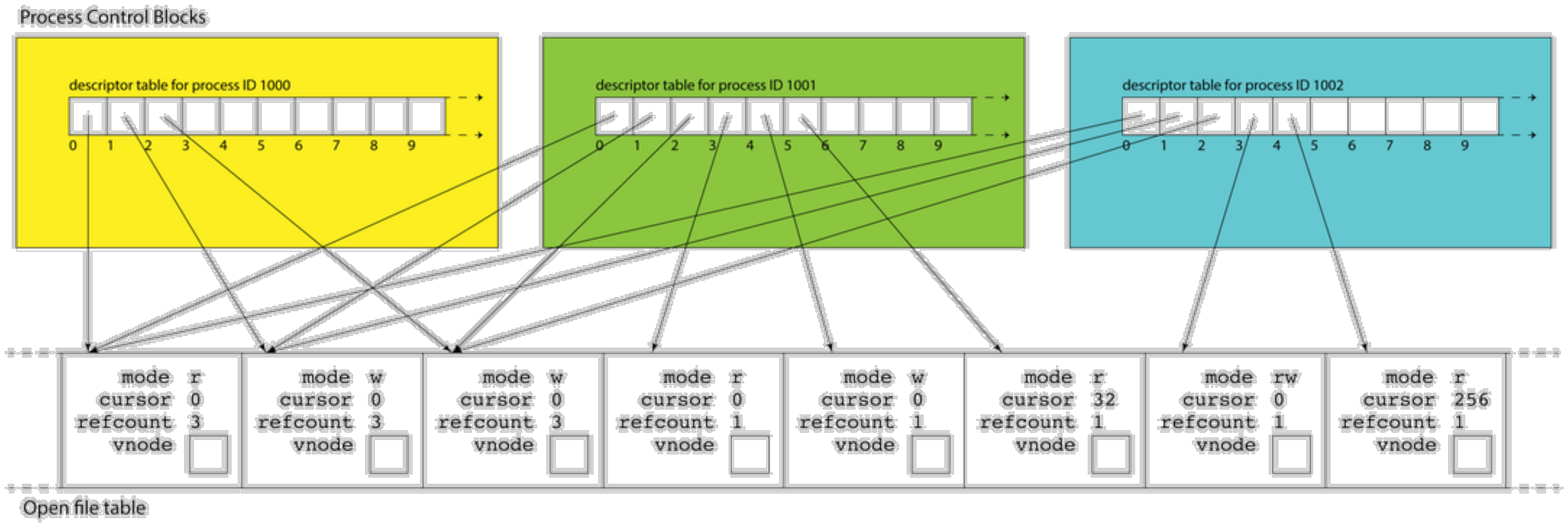
The OS maintains a (private) **process control block** for each process - a set of relevant information about its execution. Lives as long as the process does.

- Information about memory used by this process
- ***File descriptor table***
- Info about threads in this process
- Other misc. accounting and info

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!**

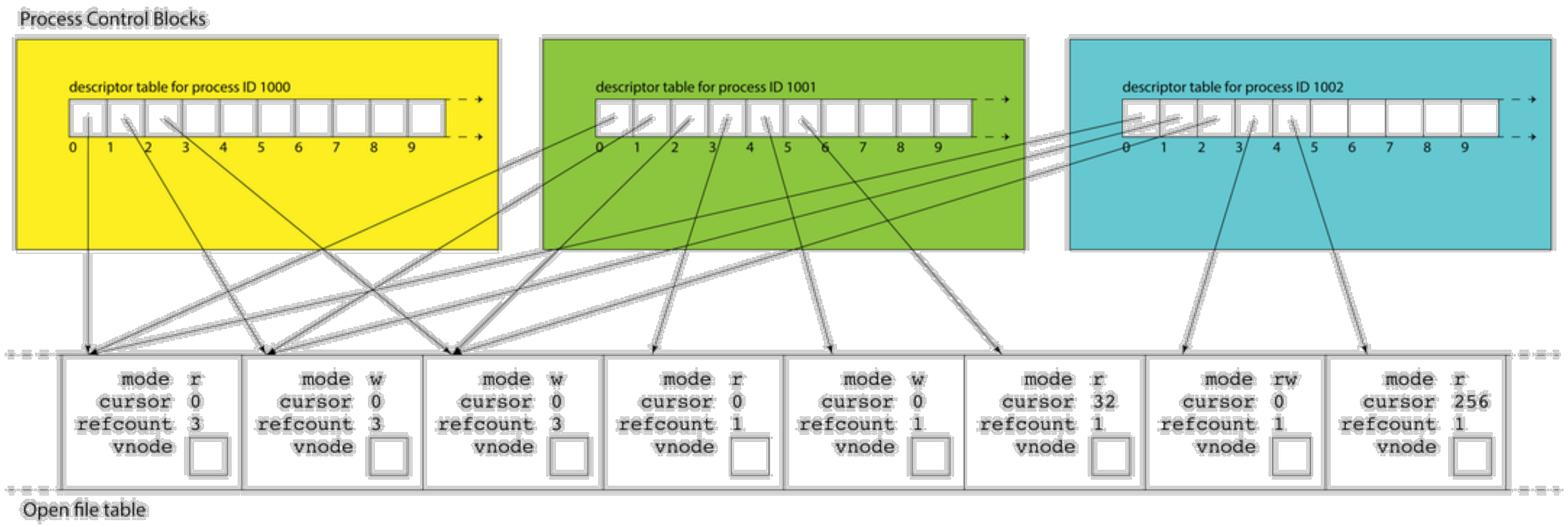
- An entry in the file descriptor table is really a *pointer* to an entry in another table, the **open file table**.
- The **open file table** is one array of information about open files/resources across all processes.



File Descriptor Table

An entry in the file descriptor table is really a *pointer* to an entry in another table, the **open file table**.

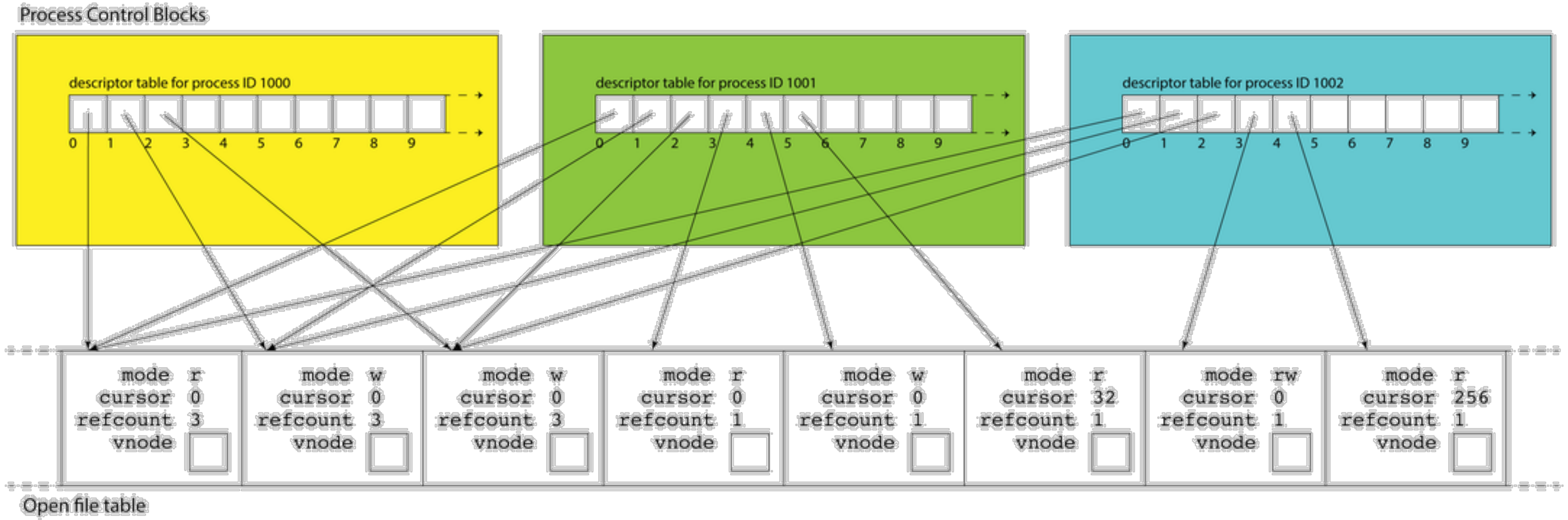
- An open file table entry stores changing info like "cursor" (how far into file are we?)
- Multiple file descriptor entries (even across processes!) can point to the same open file table entry. This is how parents and children share file descriptors.



File Descriptor Table

An entry in the file descriptor table is really a *pointer* to an entry in another table, the **open file table**.

- This also clarifies what **dup2** does; it copies a pointer to a new file descriptor table index
- All of these data structures are private to the operating system. They are layered on top of the filesystem data itself.



Process and Thread State

Key question #1: How does the operating system track info about threads and processes?

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

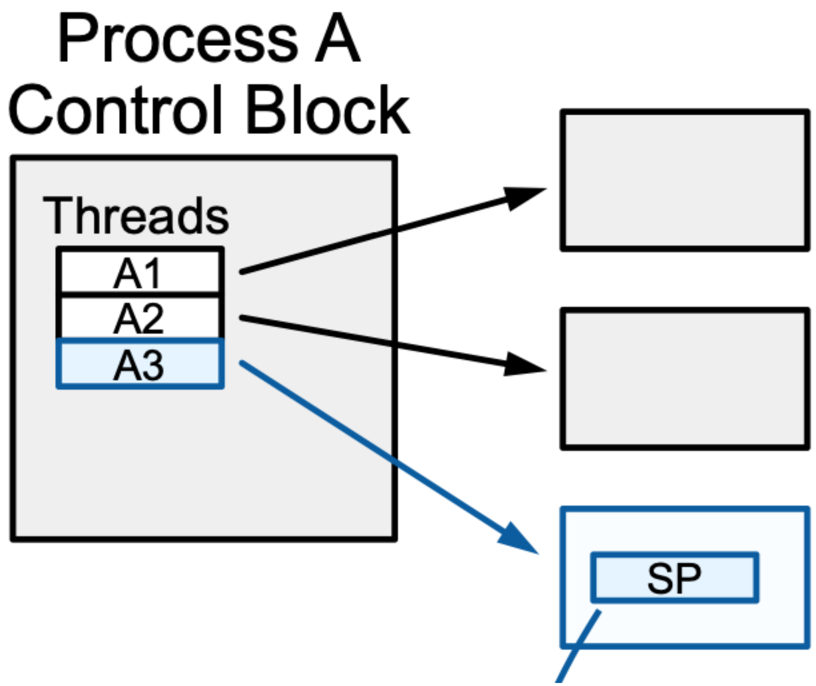
- Information about memory used by this process
- File descriptor table
- ***Info about threads in this process***
- Other misc. accounting and info

Thread State

- Every process has 1 main thread and can spawn additional threads.
- Threads are the "unit of execution" – processes aren't executed, threads are

Threads share info in PCB plus also have their own private state in the PCB, e.g. thread's stack info

- Recall: there is a register called %rsp that points to the top of the stack ("stack pointer"). Non-running threads must save their %rsp somewhere for later.



Aside: x86-64 Assembly Refresher

- A **register** is a 64-bit space inside a processor core.
- Each core has its own set of registers.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold parameters and return values for functions.
- Some registers have special responsibilities – e.g. **%rsp** always stores the address of the current top of the stack.
- When a thread is being kicked off, it must remember its **%rsp** value so it knows where its stack is the next time it runs. (we’ll see how it remembers other register values later)

Plan For Today

- **Overview:** Scheduling and Dispatching
- Process and Thread State
- **Running a Thread**
- Switching Between Threads
- Tracking All Threads

Running a Thread

Key Question #2: How does the operating system run a thread and switch between threads?

- A processor has 1 or more “cores” - Each core contains a complete CPU capable of executing a thread
- Typically have more threads than cores, but most may not need to run at any given point in time (why? They are waiting for something)
- When the OS wants to run a thread, it loads its state (e.g. %rsp and other registers) into a core, and starts or resumes it
- **Problem:** once we run a thread, the OS is not running anymore! (e.g. 1 core) How does it 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.

Plan For Today

- **Overview:** Scheduling and Dispatching
- Process and Thread State
- Running a Thread
- **Switching Between Threads**
- Tracking All Threads

Switching Between Threads

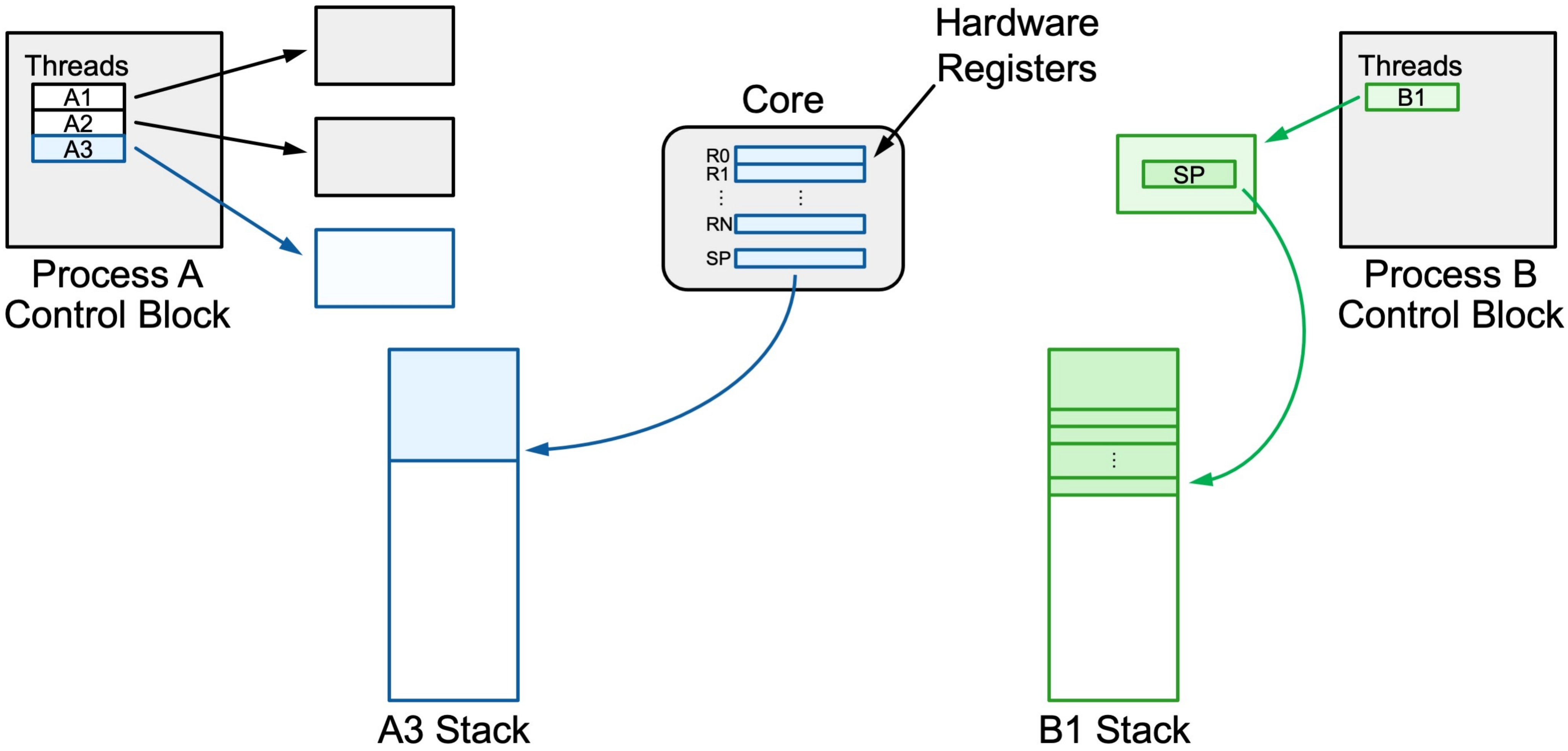
When the OS regains control, how does it switch to run another thread?

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

- Not a thread – code that is invoked to perform the dispatching function
- Lets a thread run, then switches to another thread, etc.
- *Context switch* – changing the thread currently running to another thread. We must save the current thread state and load in the new thread state.
- 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!!*

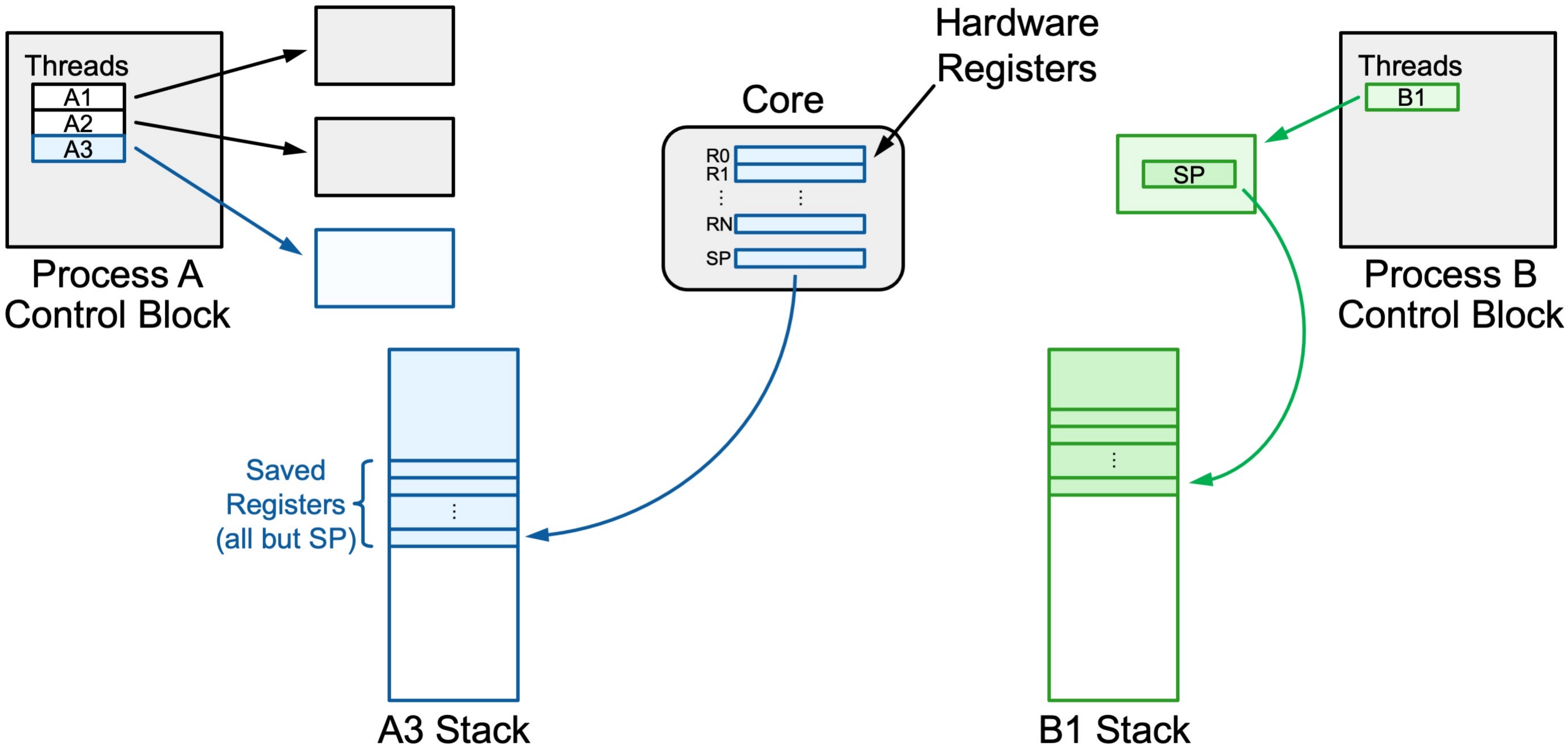
Context Switching

Context switch: how do we switch from thread A3 to thread B1?



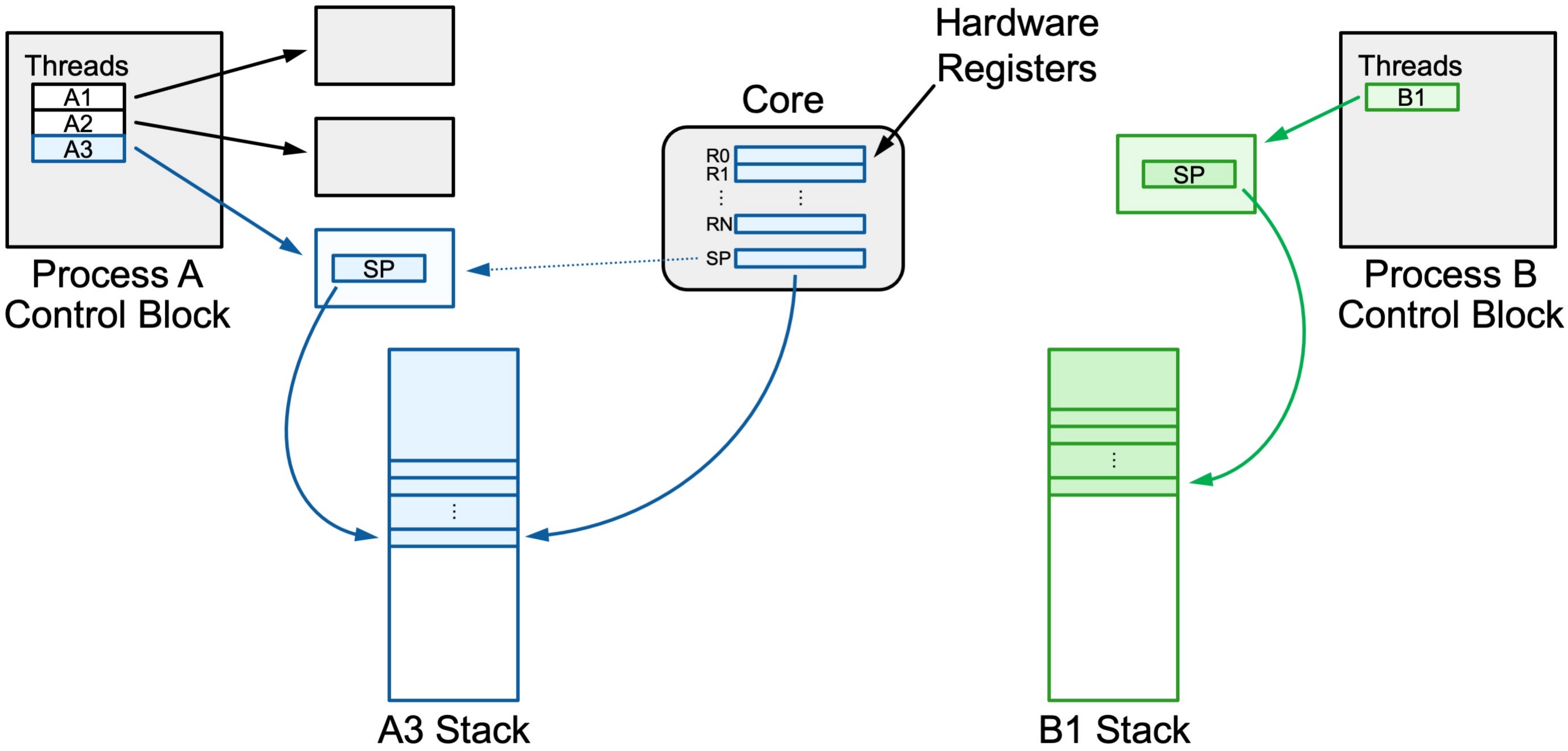
Context Switching

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



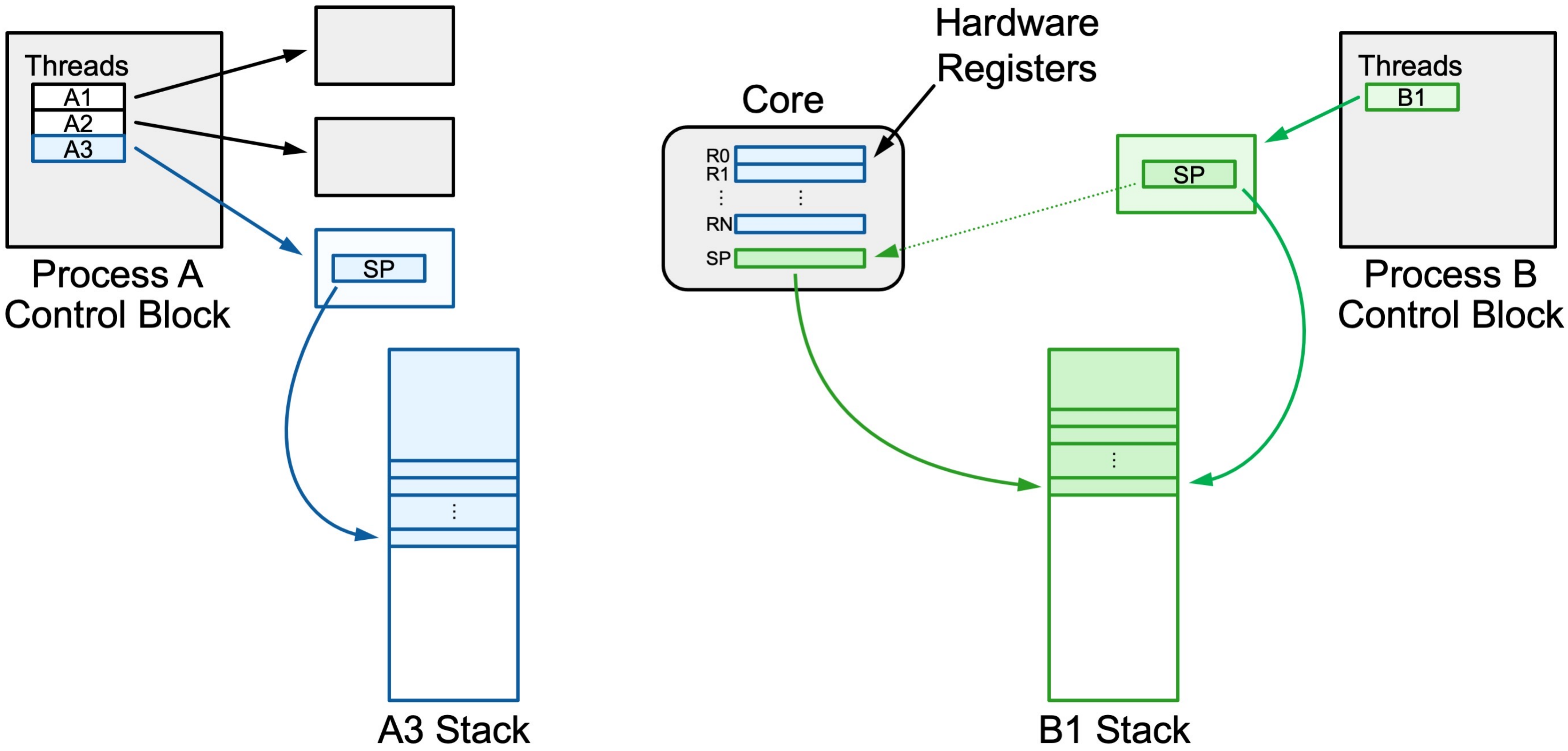
Context Switching

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



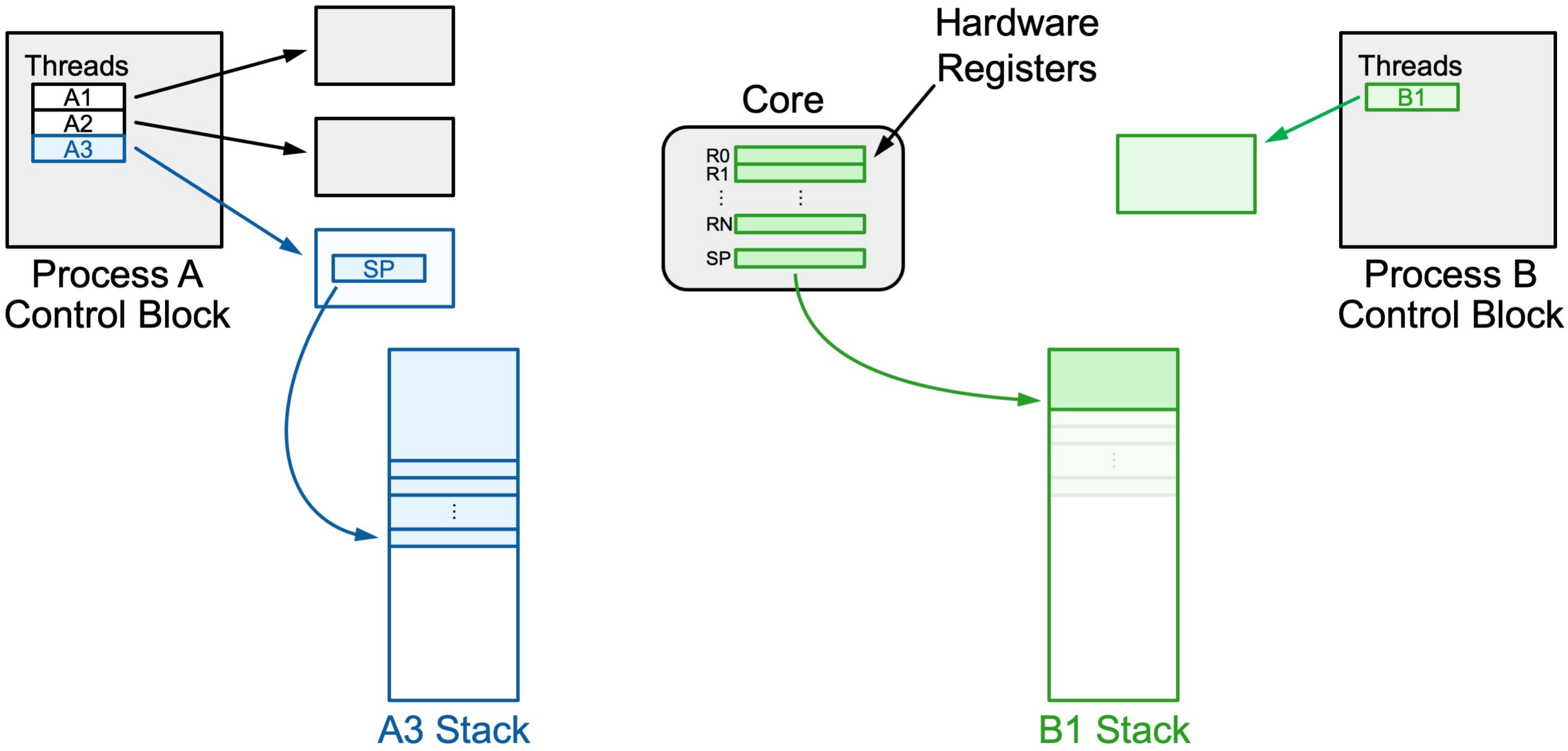
Context Switching

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.*



Plan For Today

- **Overview:** Scheduling and Dispatching
- Process and Thread State
- Running a Thread
- Switching Between Threads
- **Tracking All Threads**

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:

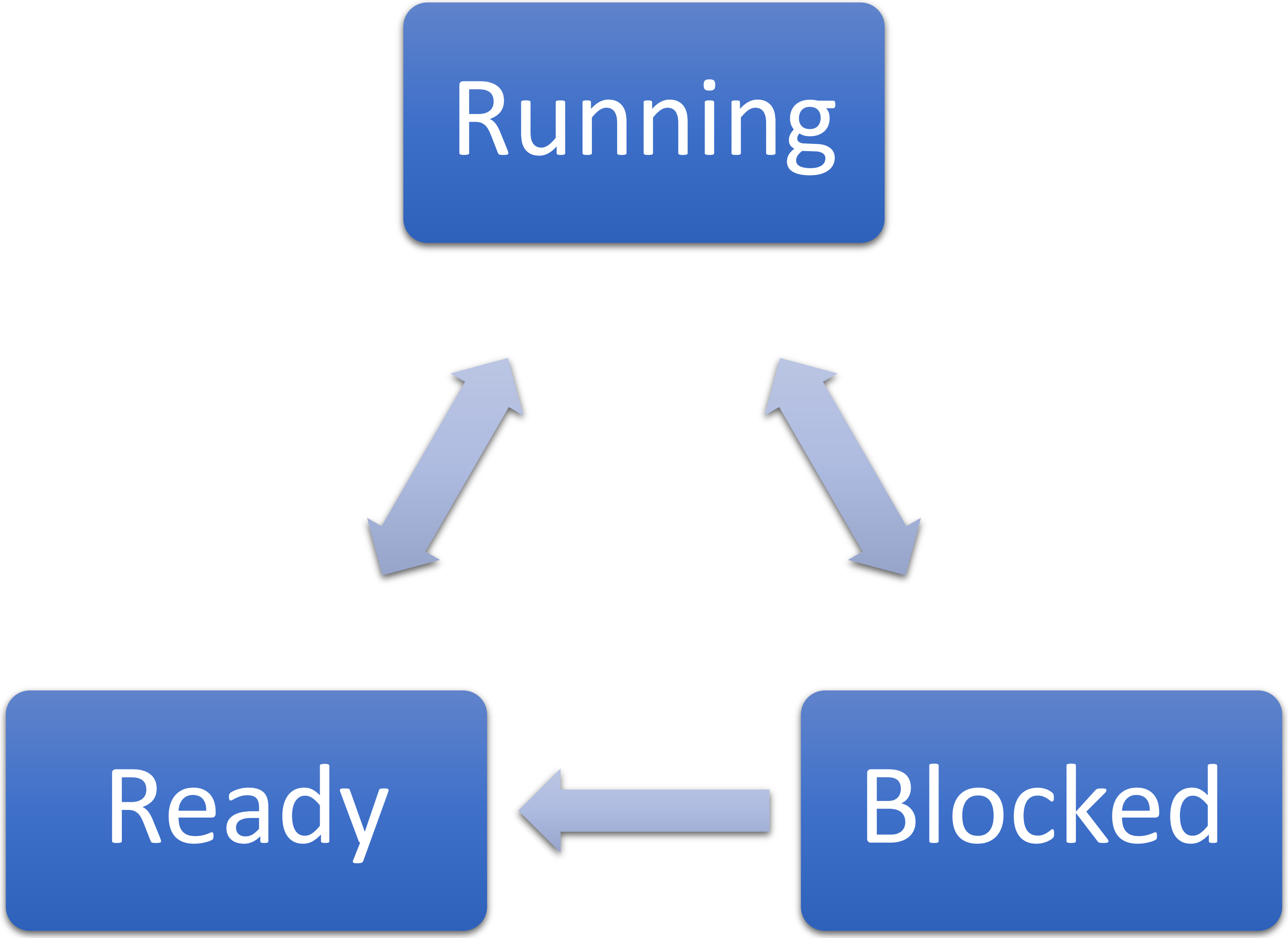
1. **Running**
2. **Blocked –** waiting for an event (disk I/O, network connection, etc.)
3. **Ready** – able to run, but waiting for CPU time

Thread States

Threads can either be **running**, **blocked** or **ready**.

- When a thread is created, it starts **ready.**
- When it is run, it goes from **ready** -> **running**
- Maybe it reaches a point where it can't run anymore (e.g. reading a file from disk). It goes from **running -> blocked**
- When the event it's waiting for has happened, it goes from **blocked -> ready** or **blocked -> running** (if it can get a core immediately)
- Sometimes we might want to interrupt a running thread to run another thread. It goes from **running -> ready**

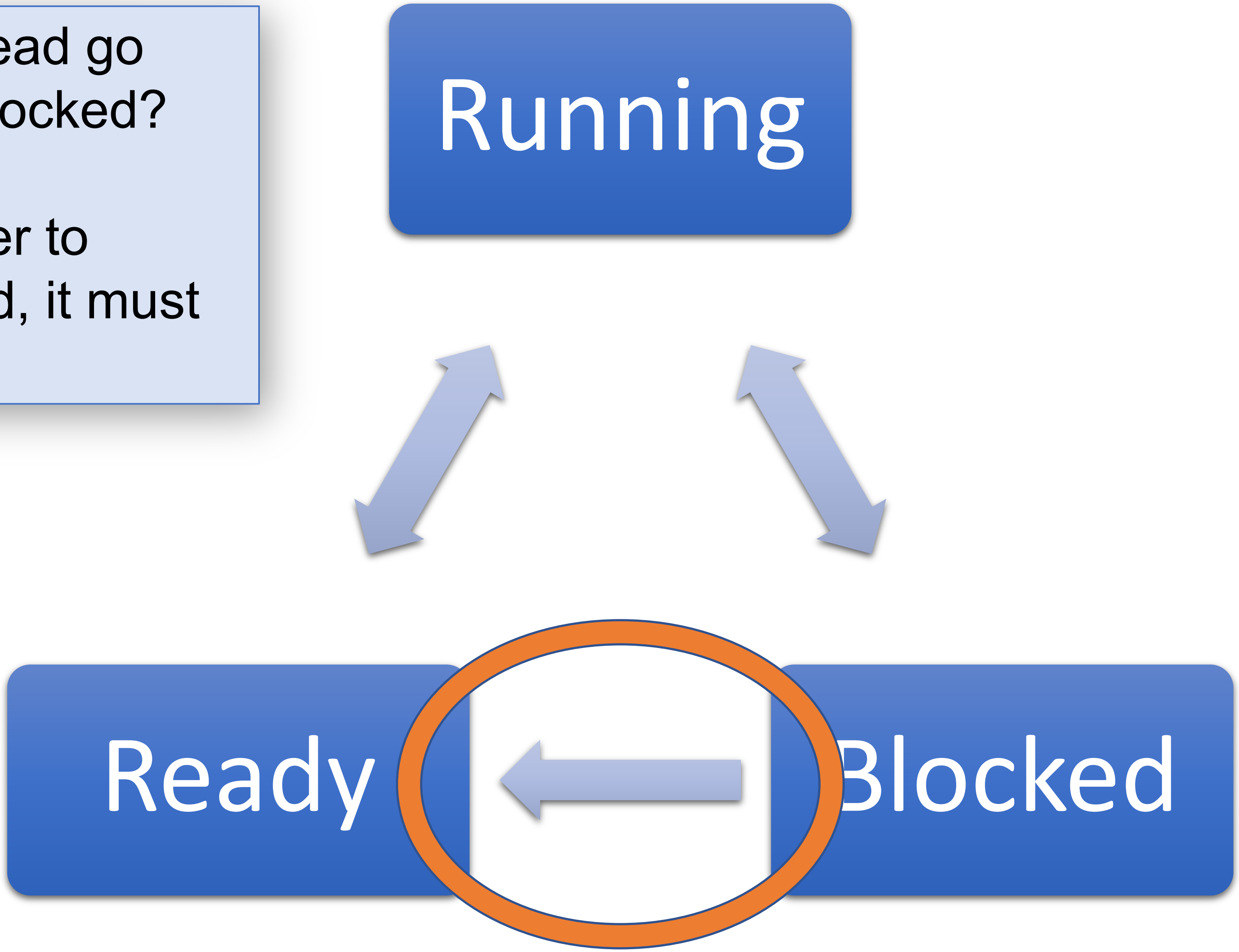
Thread States



Thread States

Why can't a thread go from ready to blocked?

Because in order to become blocked, it must run code.



Plan For Today

- **Overview:** Scheduling and Dispatching
- Process and Thread State
- Running a Thread
- Switching Between Threads
- Tracking All Threads

Lecture 16 takeaway: The OS keeps a process control block for each process and uses it to context switch between threads. To switch we must freeze frame the existing register values and load in new ones.

Next time: more about scheduling and dispatching