

CS111, Lecture 17

Dispatching

Optional reading:

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



masks recommended

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

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?*

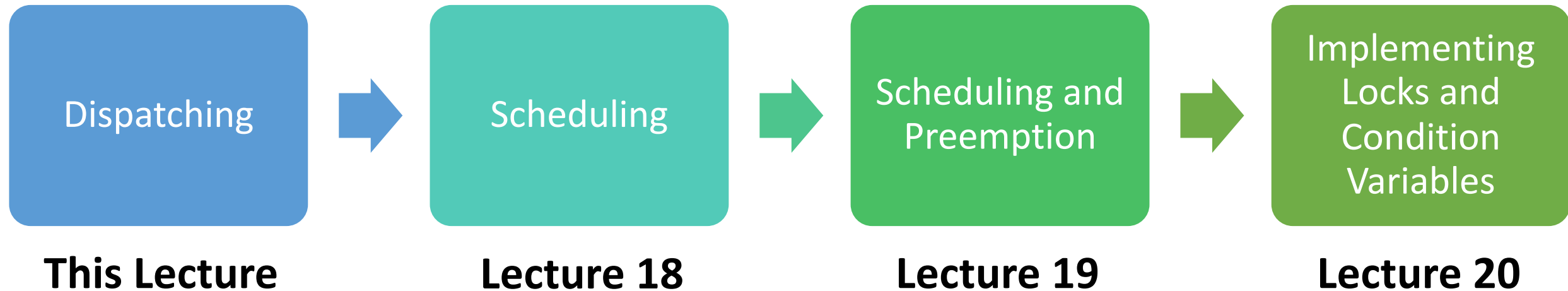
Why is answering this question important?

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

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



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: Dispatching and Scheduling
- Process and Thread State
- Running a Thread
- Switching to Another Thread

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

Plan For Today

- **Overview: Dispatching and Scheduling**
- Process and Thread State
- Running a Thread
- Switching to Another Thread

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

Scheduling And Dispatching

We have learned how user programs can create new processes and spawn threads. But how does the operating system manage this internally? What happens when we spawn a new thread or create a new process?

Key questions we will answer:

- How does the operating system track info for threads and processes? (today)
- How does the operating system run a thread and how does it switch between threads (“dispatching”)? (today)
- **Scheduling:** How does the operating system decide which thread to run next? (next time)

Plan For Today

- Overview: Dispatching and Scheduling
- **Process and Thread State**
- Running a Thread
- Switching to Another Thread

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

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

Process and Thread State

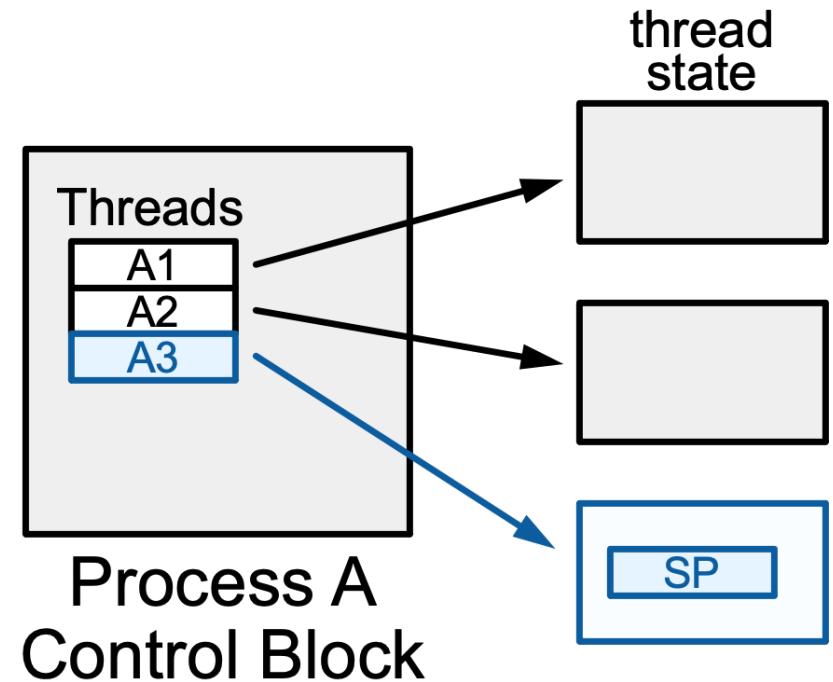
Key question #1: How does the operating system track info about threads and processes?

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
- 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)
- 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: Dispatching and Scheduling
- Process and Thread State
- **Running a Thread**
- Switching to Another Thread

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

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: Dispatching and Scheduling
- Process and Thread State
- Running a Thread
- **Switching to Another Thread**

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

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 (registers) and load in the new thread state.
- Context switches are funky – like running a function that, as part of its execution, returns to a *completely different function in a completely different thread!!*

Demo: context-switch.cc

Context Switch

```
Thread main_thread;
Thread other_thread;

void other_func() {
    cout << "Howdy! I am another thread." << endl;
    context_switch(other_thread, main_thread);
    cout << "We will never reach this line :(" << endl;
}

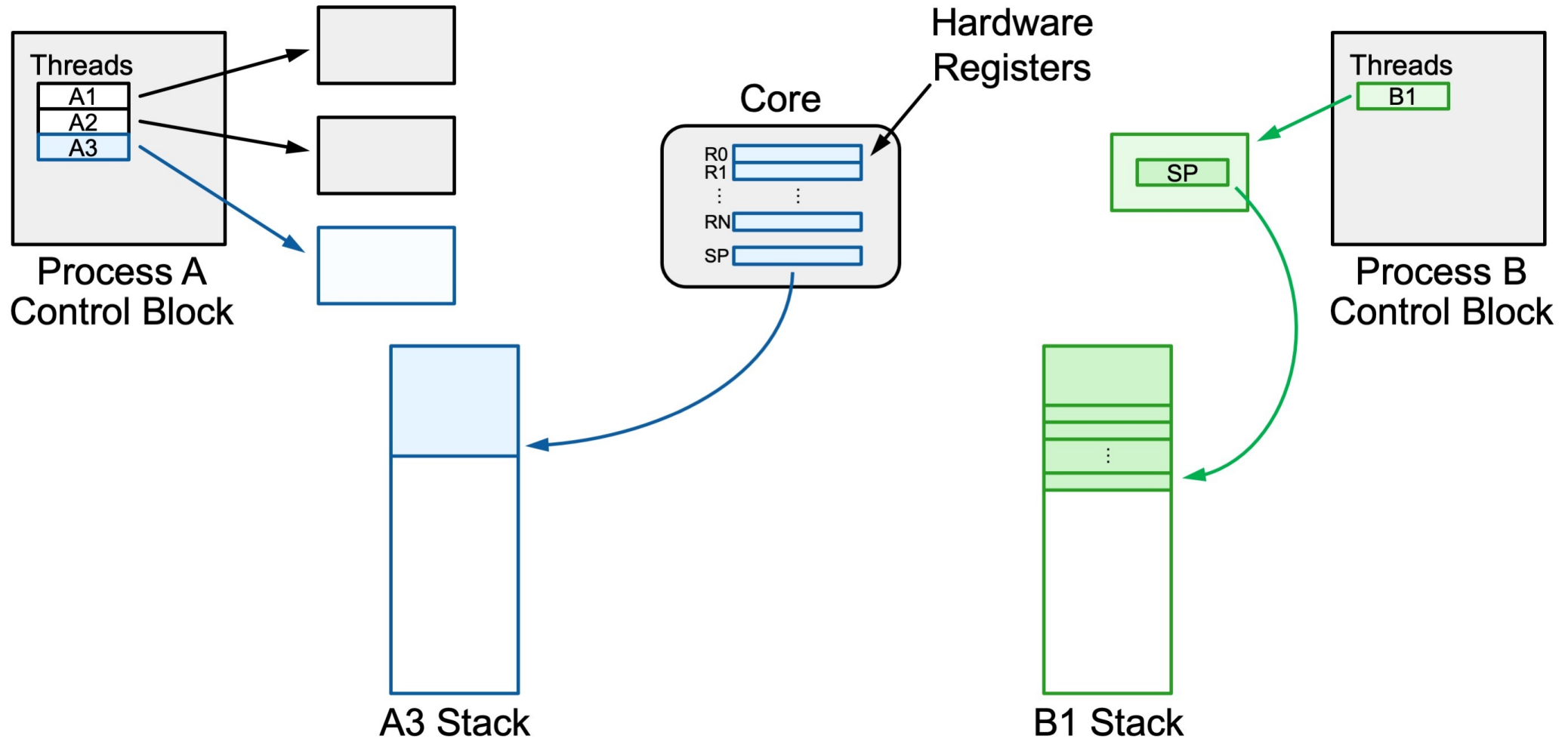
int main(int argc, char *argv[]) {
    // Initialize other_thread to run other_func
    other_thread = create_thread(other_func);

    cout << "Hello, world! I am the main thread" << endl;
    context_switch(main_thread, other_thread);
    cout << "Cool, I'm back in main()!" << endl;
    return 0;
}
```

- *context_switch* is called from one function, but returns to another
- The next time we switch back to the original thread, it resumes where it left off.

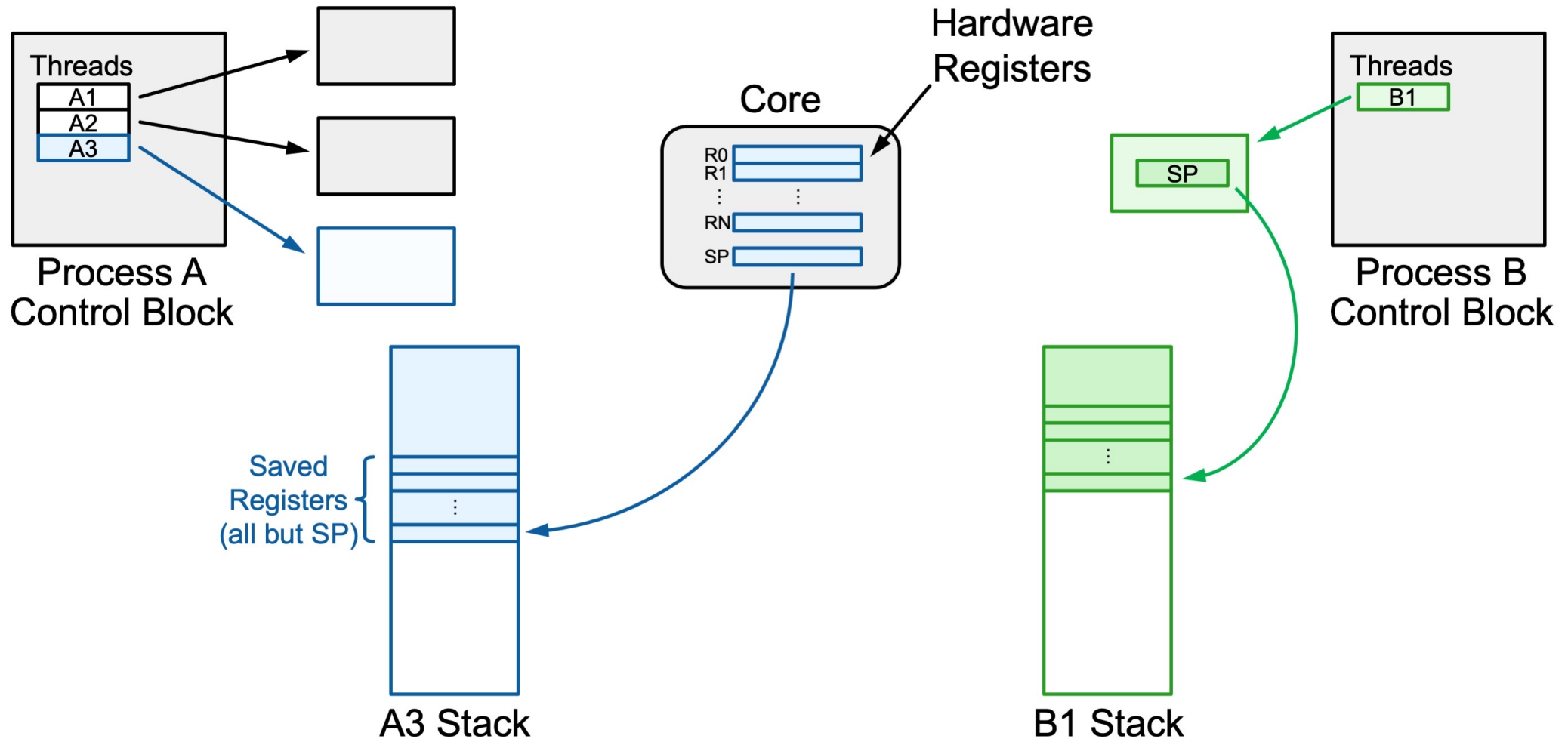
Context Switching

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



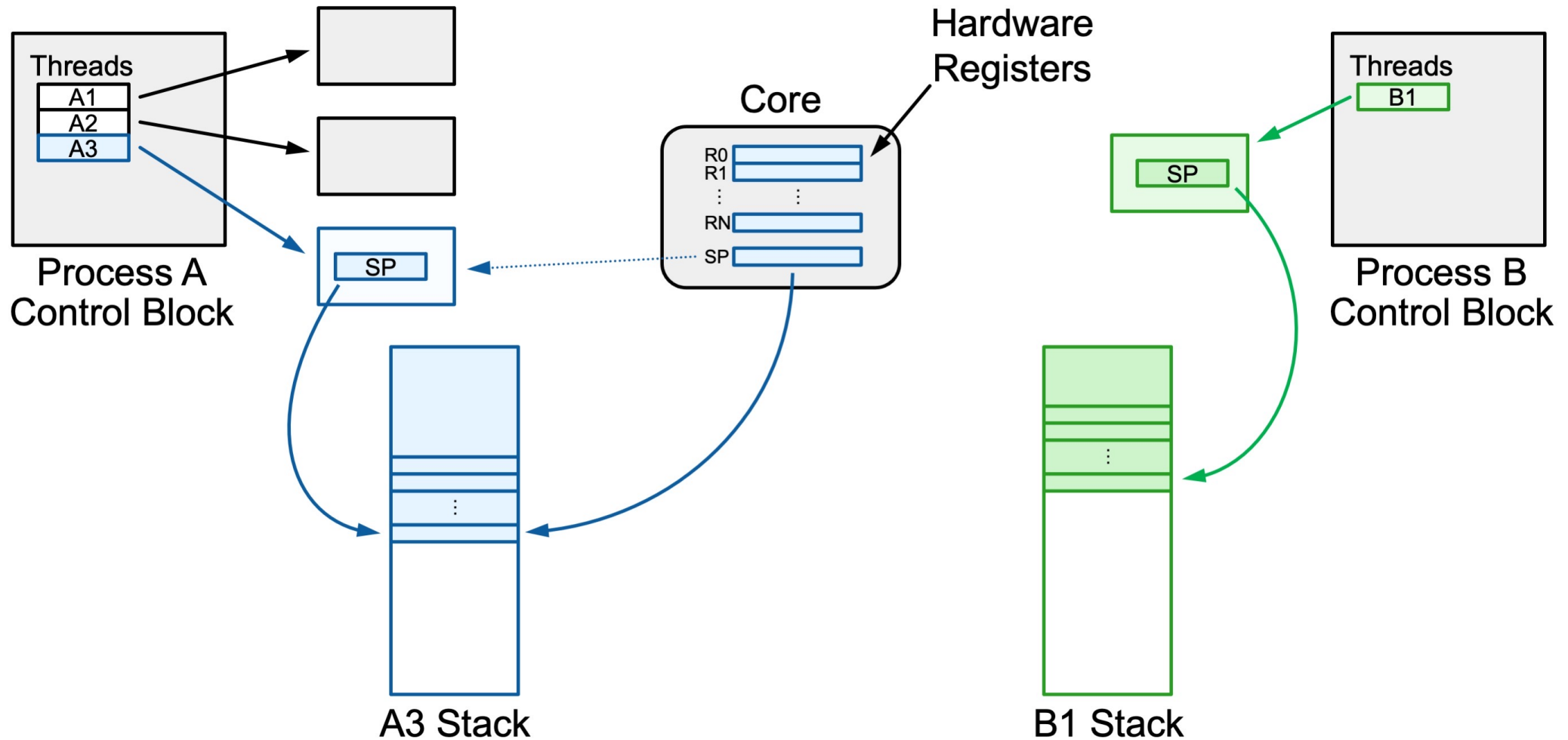
Context Switching

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



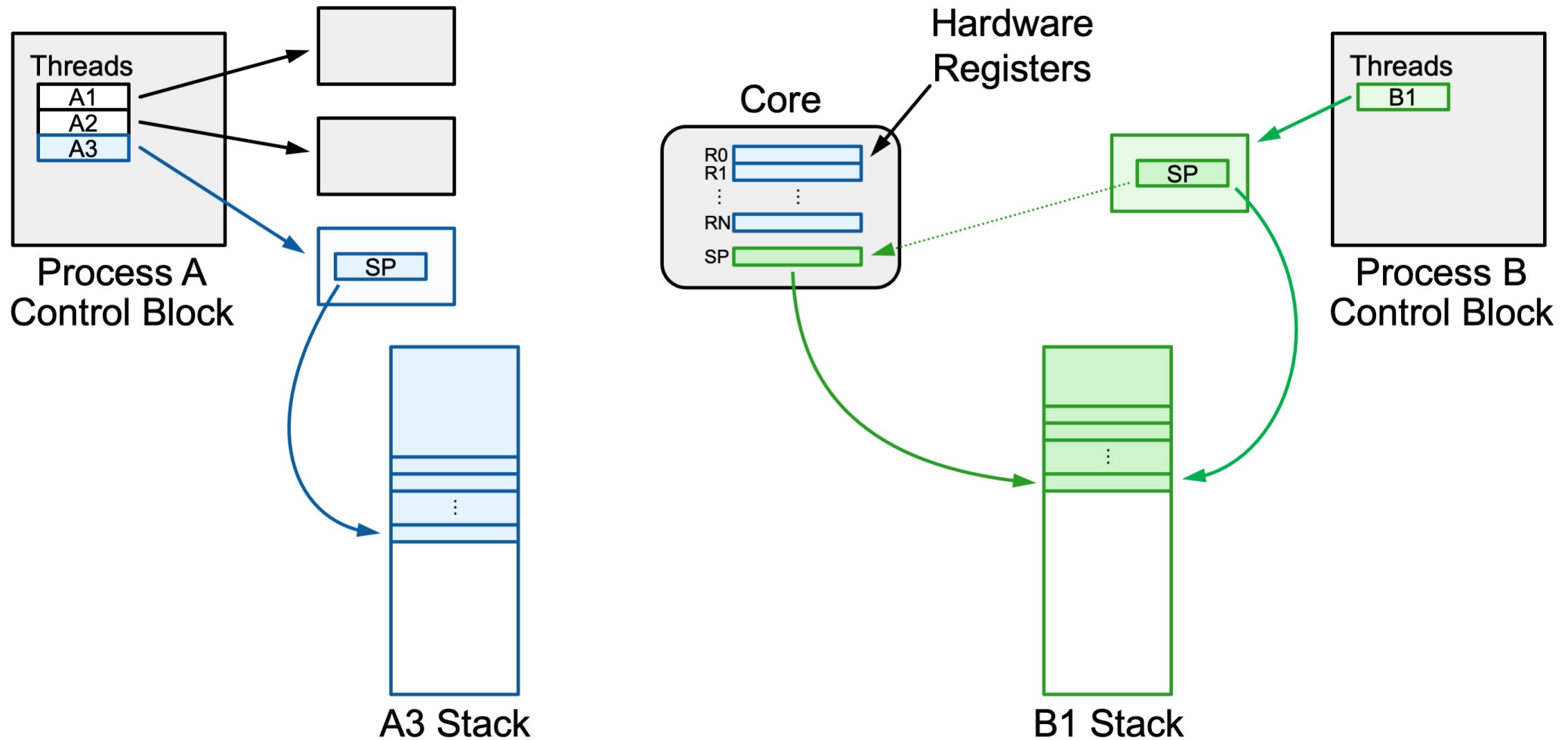
Context Switching

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



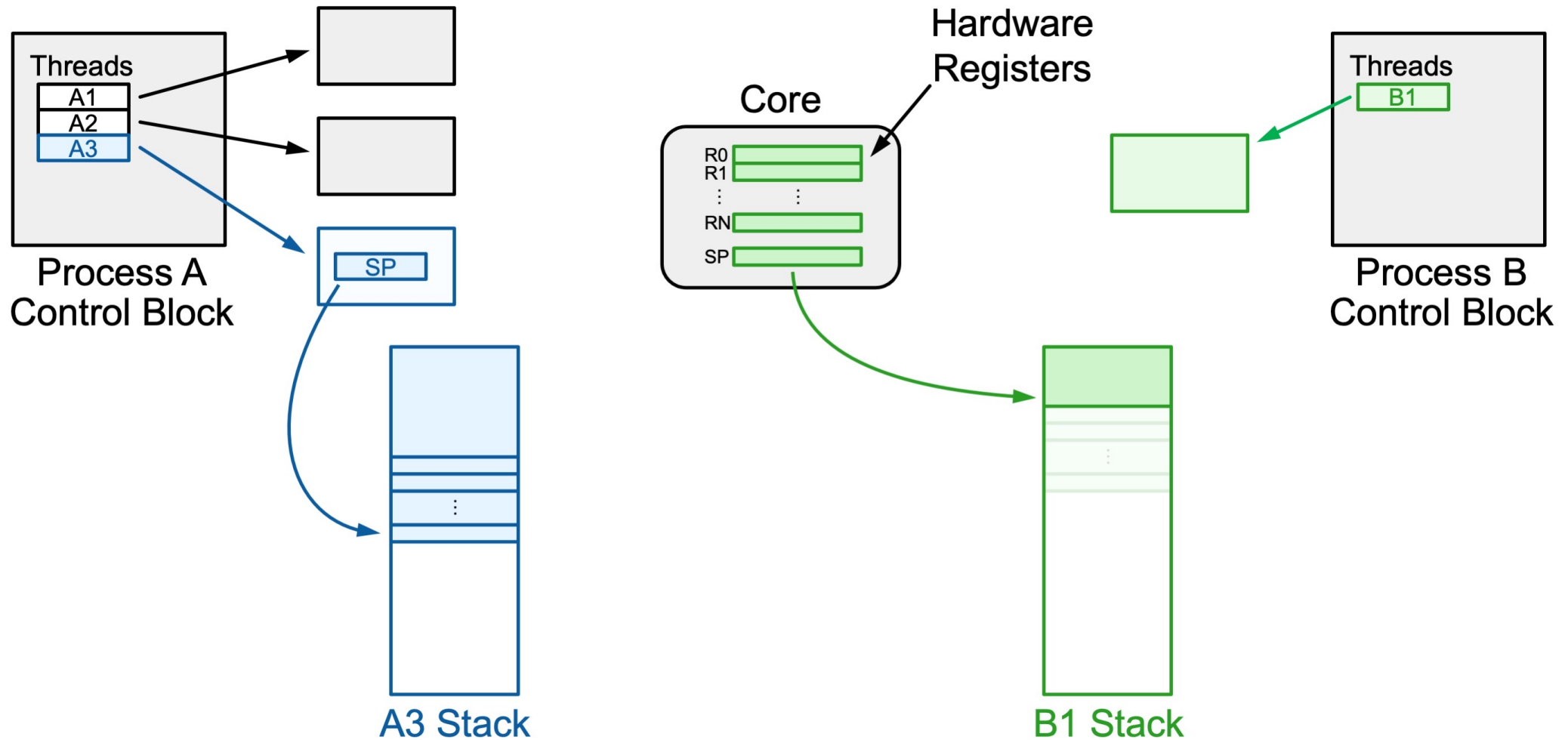
Context Switching

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.



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 Switch

```
Thread main_thread;
Thread other_thread;

void other_func() {
    cout << "Howdy! I am another thread." << endl;
    context_switch(other_thread, main_thread);
    cout << "We will never reach this line :(" << endl;
}

int main(int argc, char *argv[]) {
    // Initialize other_thread to run other_func
    other_thread = create_thread(other_func);

    cout << "Hello, world! I am the main thread" << endl;
    context_switch(main_thread, other_thread);
    cout << "Cool, I'm back in main()!" << endl;
    return 0;
}
```

- *context_switch* is called from one function, but returns to another
- The next time we switch back to the original thread, it resumes where it left off.

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
```



we start executing on one stack...




and end executing on another!

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!



Context Switch

```
Thread main_thread;
Thread other_thread;

void other_func() {
    cout << "Howdy! I am another thread." << endl;
    context_switch(other_thread, main_thread);
    cout << "We will never reach this line :(" << endl;
}

int main(int argc, char *argv[]) {
    // Initialize other_thread to run other_func
    other_thread = create_thread(other_func);

    cout << "Hello, world! I am the main thread" << endl;
    context_switch(main_thread, other_thread);
    cout << "Cool, I'm back in main()!" << endl;
    return 0;
}
```

- *context_switch* is called from one function, but returns to another
- The next time we switch back to the original thread, it resumes where it left off.

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.

Context Switch Practice

```
Thread main_thread;
Thread other_thread;

void other_func() {
    context_switch(other_thread, main_thread);
    cout << "D" << endl;
    context_switch(other_thread, main_thread);
    cout << "A" << endl;
}

int main(int argc, char *argv[]) {
    other_thread = create_thread(other_func);
    cout << "B" << endl;
    context_switch(main_thread, other_thread);
    cout << "C" << endl;
    context_switch(main_thread, other_thread);
    return 0;
}
```

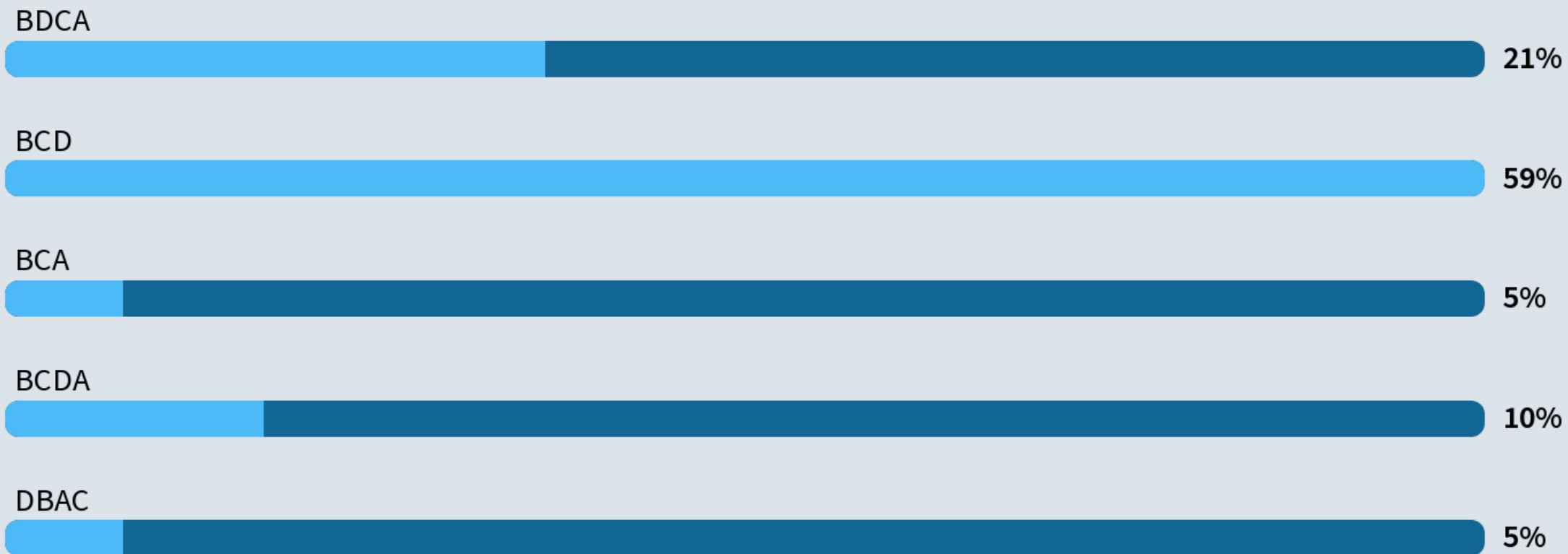
What would be outputted by this program? *Key points:*

- *context_switch* is called from one function, but returns to another
- The next time we switch back to the original thread, it resumes where it left off.

Respond on PollEv: pollev.com/cs111fall23
or text CS111FALL23 to 22333 once to join.



What would be outputted by this program?



Context Switch Practice

```
Thread main_thread;
Thread other_thread;

void other_func() {
    context_switch(other_thread, main_thread);
    cout << "D" << endl;
    context_switch(other_thread, main_thread);
    cout << "A" << endl;
}

int main(int argc, char *argv[]) {
    other_thread = create_thread(other_func);
    cout << "B" << endl;
    context_switch(main_thread, other_thread);
    cout << "C" << endl;
    context_switch(main_thread, other_thread);
    return 0;
}
```

Answer: *BCD*

What would be outputted by this program? *Key points:*

- *context_switch* is called from one function, but returns to another
- The next time we switch back to the original thread, it resumes where it left off.

Recap

- Overview: Dispatching and Scheduling
- Process and Thread State
- Running a Thread
- Switching to Another Thread

Lecture 17 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: how do we decide which thread to run?