Threads, Processes, and Dispatching, Continued

Mendel Rosenblum

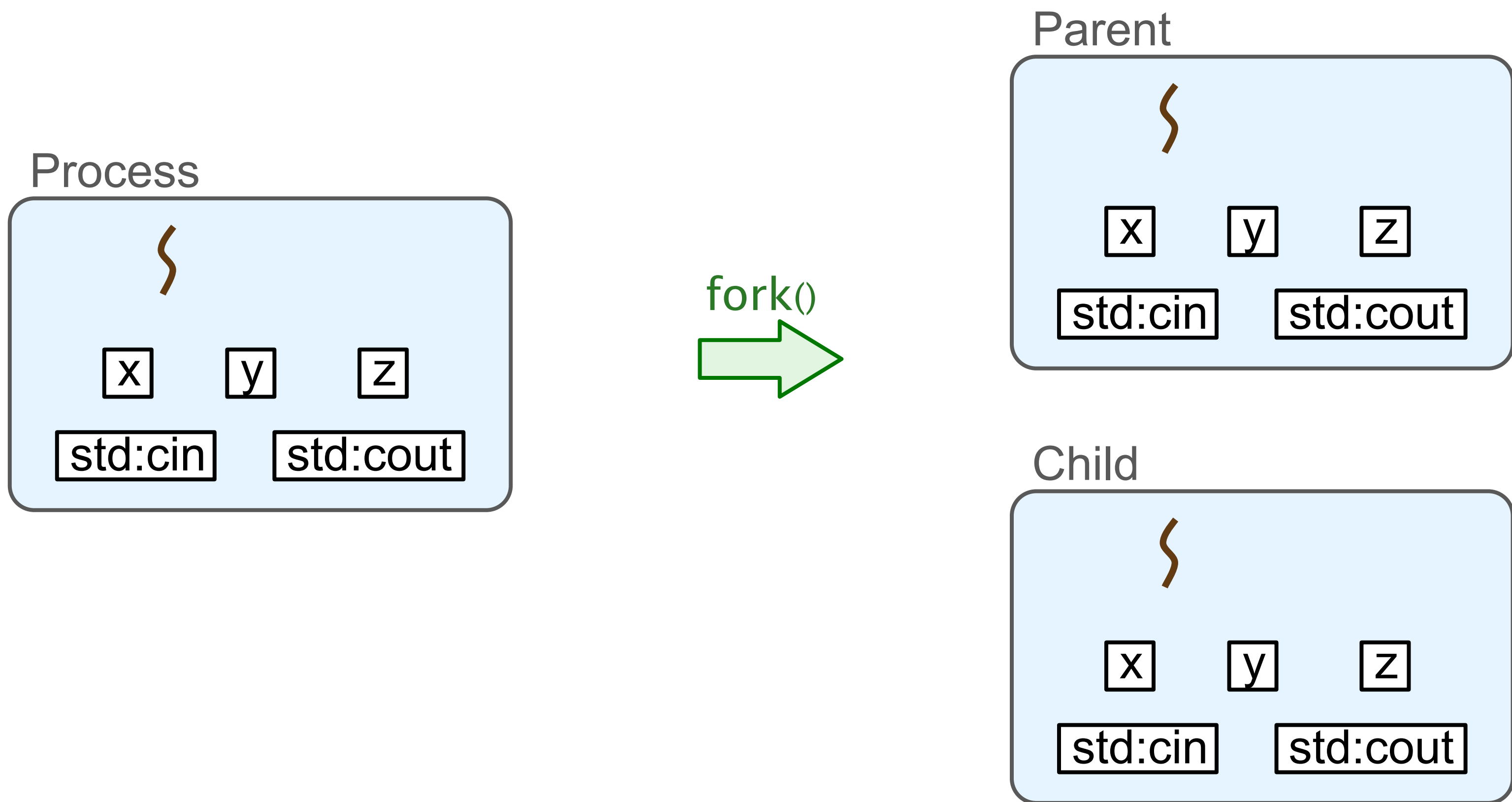
Execution abstractions review

- **Thread** - Smallest unit of execution
- **Process** - One or more threads and their execution state

Linux process create:
Single thread

```
int child_pid_or_zero = fork();
if (child_pid_or_zero == 0) {
    execvp("ls", argv);
    // Not reached
} else {
    waitpid(child_pid_or_zero, &status, options);
};
```

fork - create new process with new thread



Thread Creation

- Processes typically start with a single thread
- A system call used to start an additional thread
  - Linux: `clone`;
  - MacOS: `thread_create_running`
  - Windows: `NtCreateThreadEx`
- A process with more than one thread is called **multithreaded**

Thread creation system calls

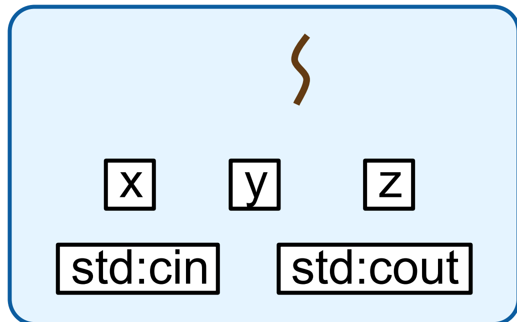
- Required information:
  - Starting program counter for thread (e.g. routine to call)
  - Starting thread stack region in memory and initial stack pointer
  - Usually some way of passing parameters to initial routine
- Examples:
  - Linux: `clone(typeof(int (void *)) *fn, void *stack, int flags, …)`
  - MacOS: `thread_create_running(parent_task, flavor, new_state, …)`
  - Windows: `NtCreateThreadEx (..., startRoutine, Argument, StackSize, …)`
- Programming language libraries wrap the thread start system call

C++ Thread Example

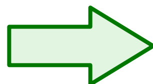
```
#include <thread>
...
std::thread t(func);
...
t.join(); // wait
```

```
void func()
{
    /* This code will run concurrently with
       the code to the left. */
    ...
}
```

Process

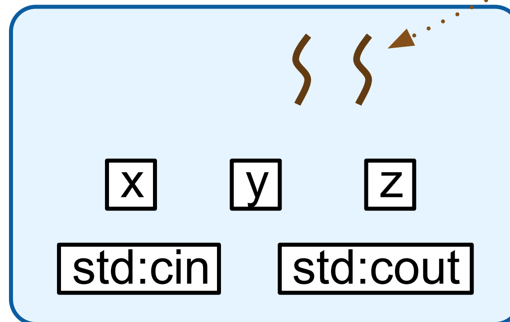


new std::thread(...)



Process

New thread shares state with existing threads



Thread Creation: Other Programming Languages

- C language

```
pthread_create(&thread, NULL, func, &arg);
```

- go language

```
go func(arg);
```

- Python language

```
t = threading.Thread(target=func, args=(arg,))
t.start()
```

Thread **Dispatching** - Running threads

Thread runs on a **core** - What are cores?

Messy terminology evolution:

- Early machines a **Central Processing Unit** (CPU)
  - Execute streams of instructions - (i.e. a thread)
  - Shorten to **processor**
  - Machines could have more than one CPU - called a **multiprocessor**
- Early microprocessors were chips implementing a single CPU
  - Multiprocessor machines constructed with multiple CPU chips
- Put multiple processors on a single chip: called **multicore**
  - Each **core** is capable of running a thread
  - Machines can have multiple **sockets** for multicore chips
- Simultaneous multithreading (hyper-threading) single core look like multiple

Core count evolution

- Until about 20 years ago, most computers were **uniprocessors**:
  - A single core capable of running one thread at a time.
- Today, most computers have many **cores** each capable of running a thread
  - Some **cores** support simultaneous multithreading (Intel's hyper-threading)
- Today:
  - Phones ~8 cores
  - Laptop ~12–16 cores
  - Desktop ~16–24 cores
  - Servers ~128–256 cores (256-512 hardware threads)

How does an OS execute threads?

- Mechanism: **Dispatching** - OS dispatches threads on cores

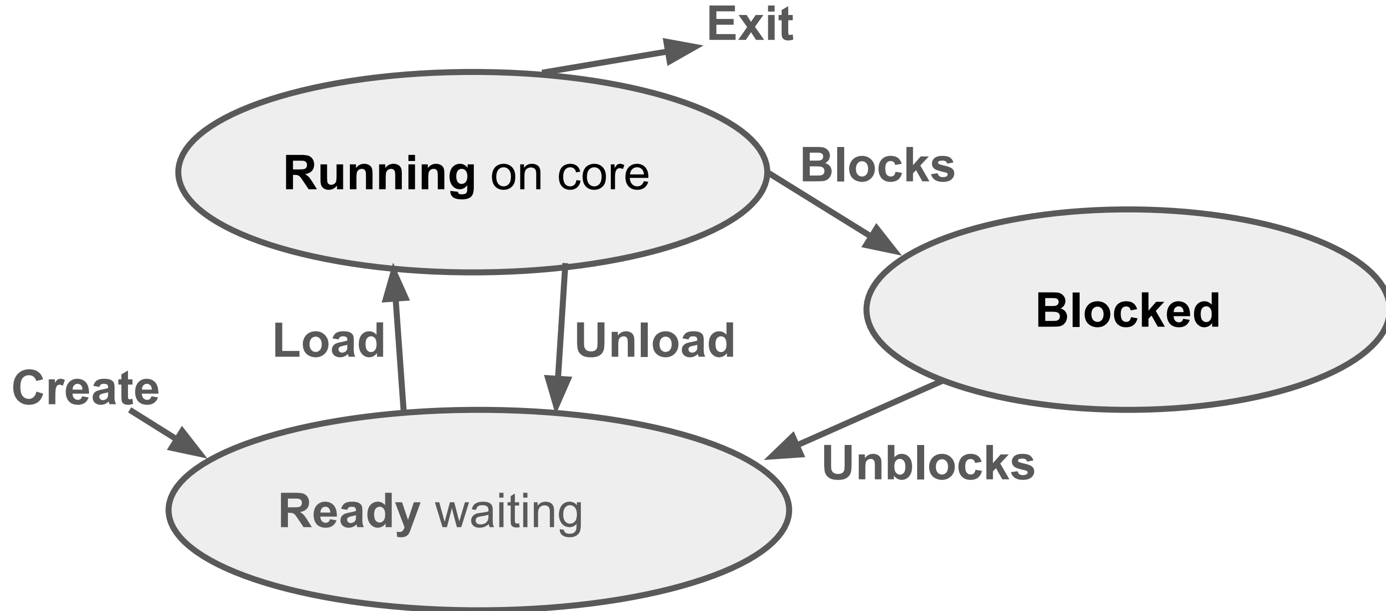
Dispatching threads on cores

- Typically have many more threads than cores
  - Fortunately most do not need to execute (waiting for something)
- OS job: execute a collection of threads on a collection of cores
  - Make sure all threads eventually get a chance to run
    - Fair scheduling
  - Make sure threads don't damage each other's state
    - Protection - can't damage other threads and the OS
    - Isolation

Process Control Block

- OS data structure for tracking processes: **Process Control Block** (PCB)
  - Saved execution state for each thread (saved registers, etc.)
  - Scheduling information
  - Information about memory used by this process
  - Information about open files
  - Accounting and other miscellaneous information

OS: Thread States



Dispatcher (one core now)

- Dispatcher: per core

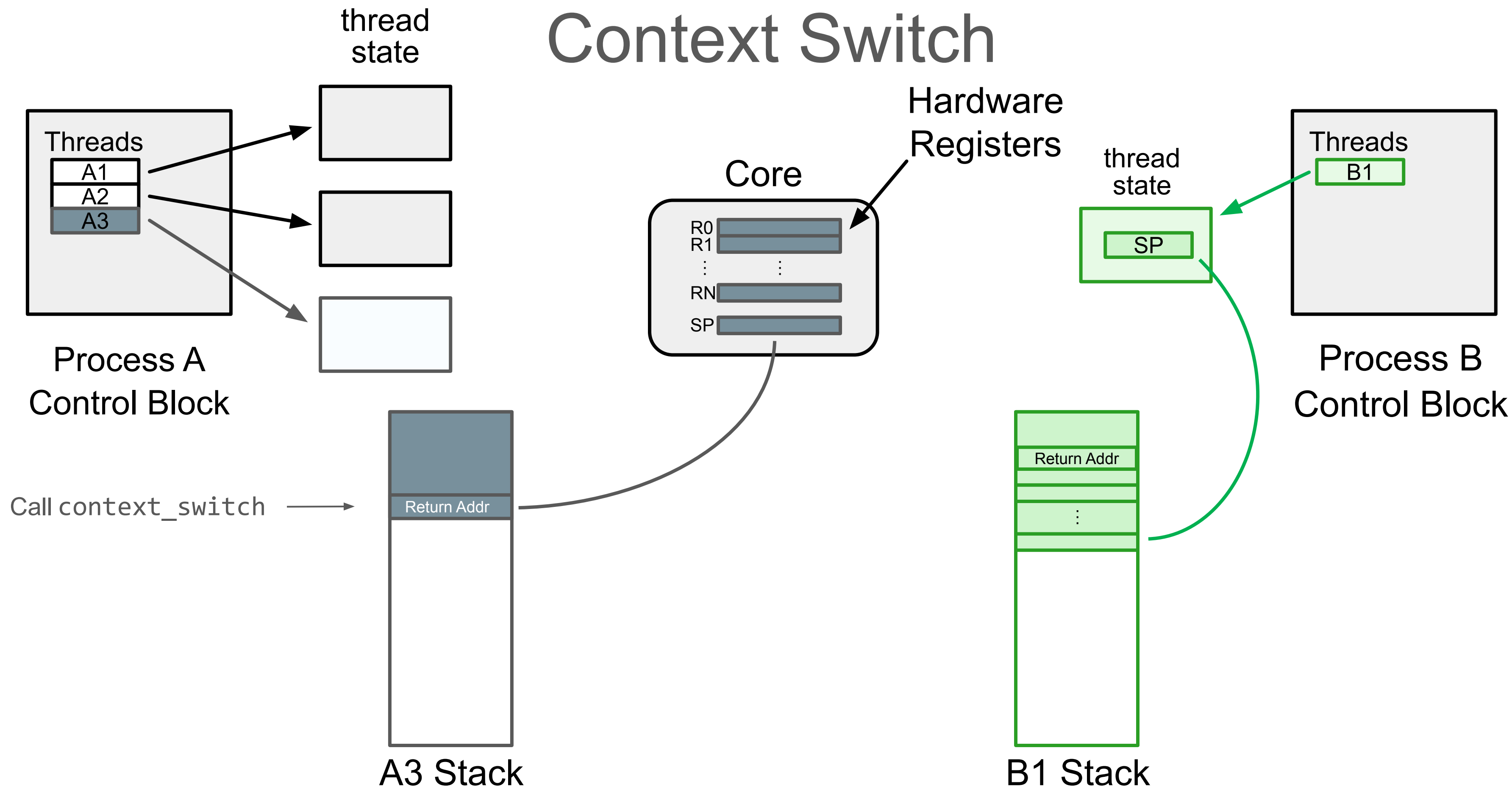
  Assume we have loaded a thread:

  - `while true:`
    - Let a thread run for a while
    - Save its execution state
    - Load state of another thread
  - `end while`

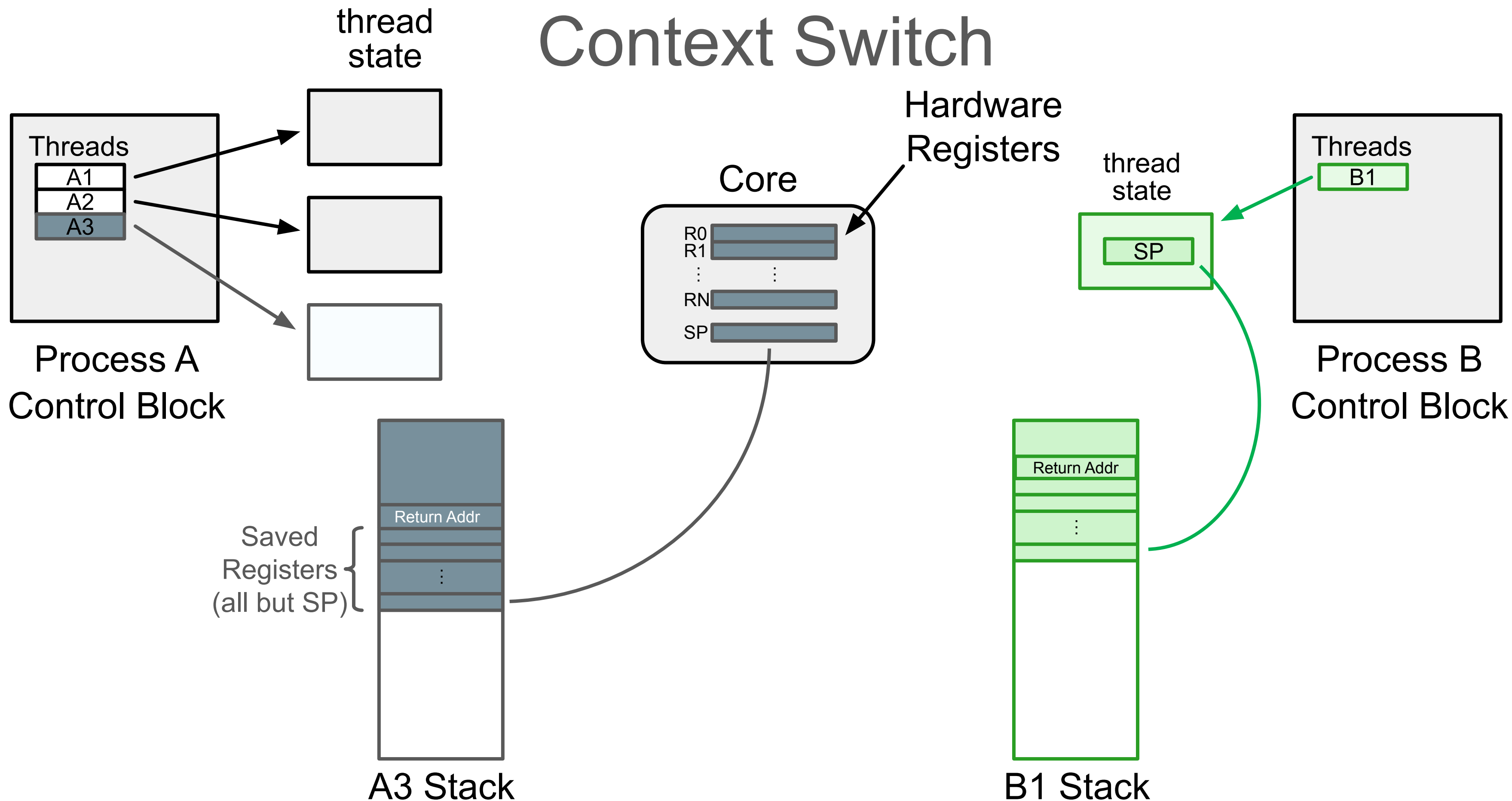
Context switch

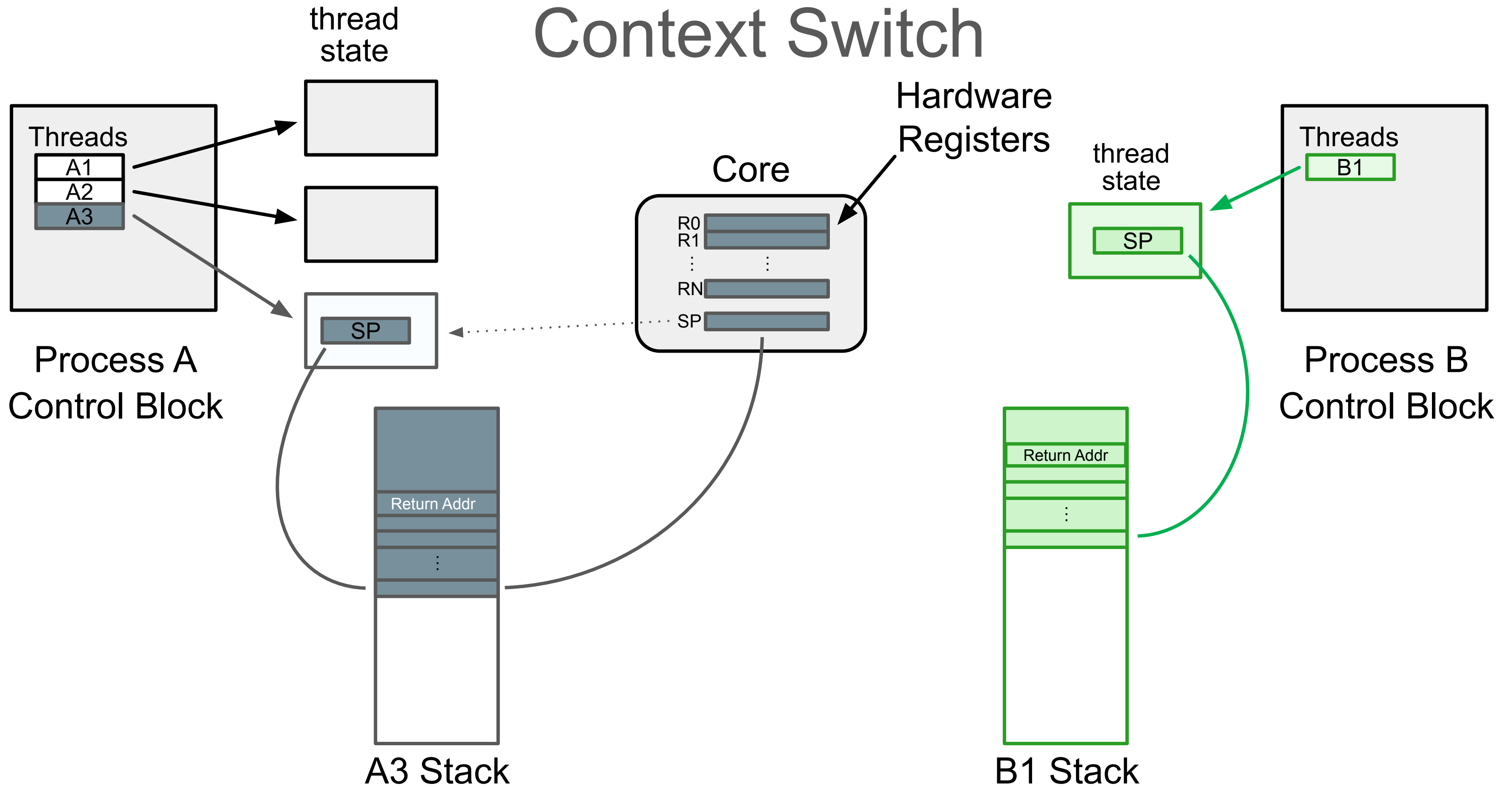
Announcements here

Context Switch



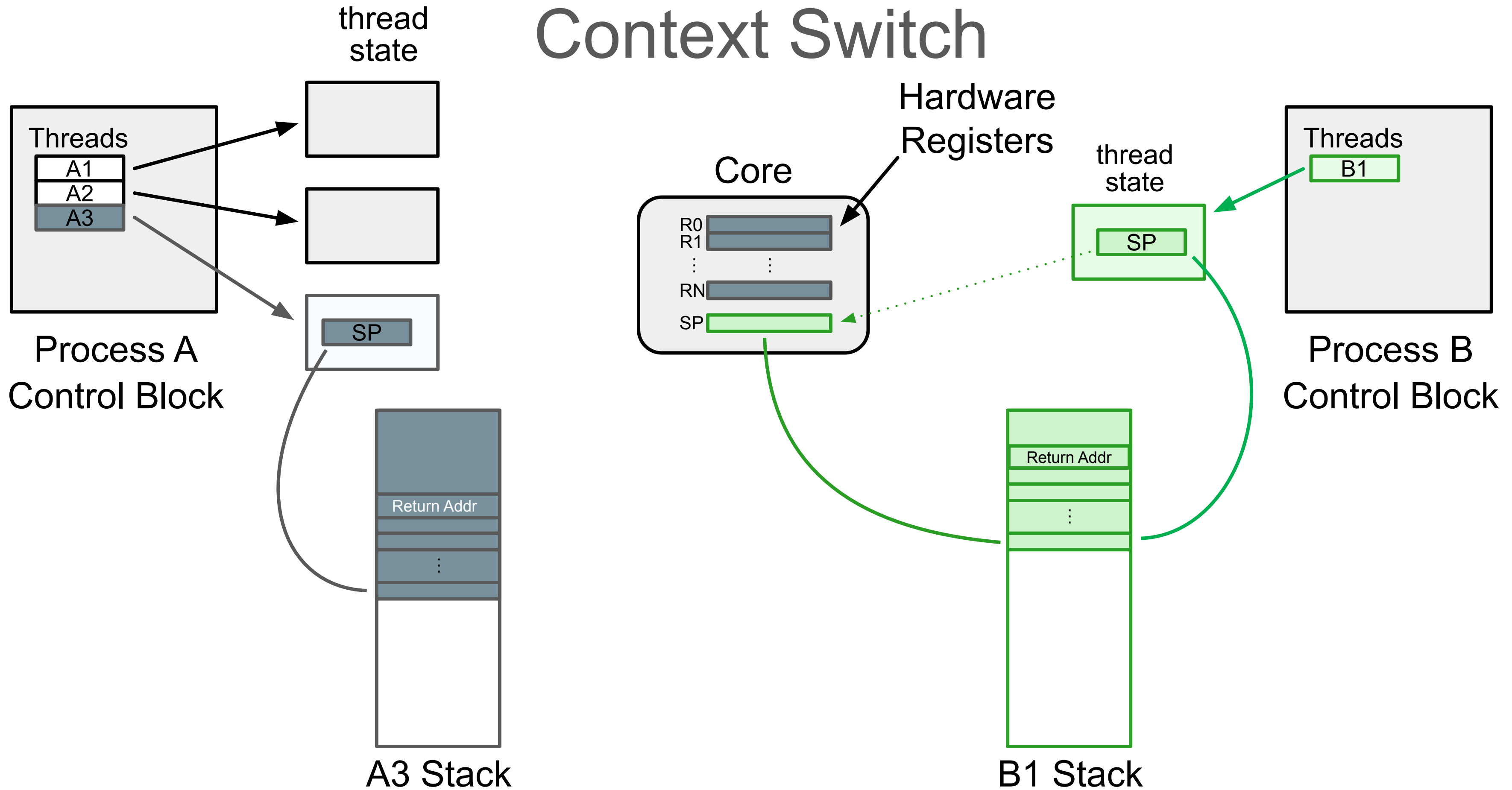
Context Switch



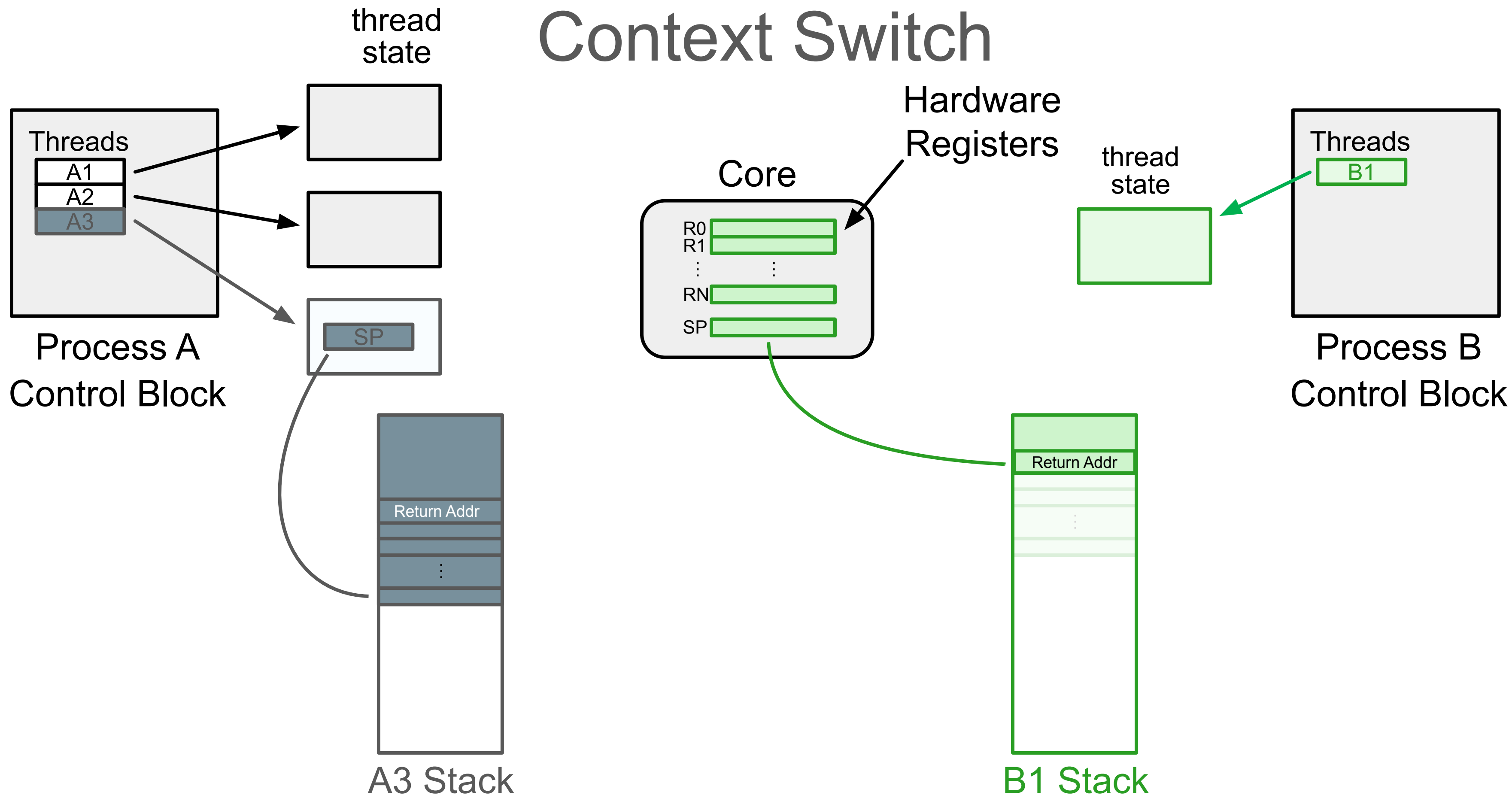


thread
state
Context Switch
Hardware
Registers
Threads
A1
A2
A3
Core
R0
R1
RN
SP
thread
state
SP
Threads
B1
SP
Process A
Control Block
Process B
Control Block
Return Addr
Return Addr
A3 Stack
B1 Stack

Context Switch



Context Switch



Dispatcher Loop

while true:

1. Let a thread run for a while
2. Save its execution state
3. Load state of another thread

end while

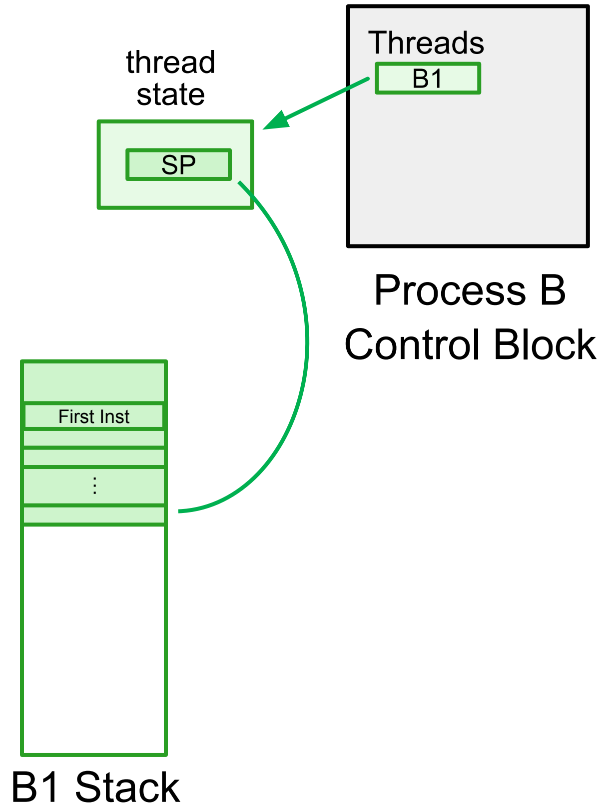
What causes the dispatcher to run?

- Ask the process to call back to the OS?
  - Called **cooperative multitasking** - misbehaving process kills machine
- Interrupts and traps jump into the OS on most architectures
  - Traps: System calls, Illegal instructions, seg faults, page faults, etc.
  - Interrupts: Keyboard key push, disk transfer finished, etc.
  - Thread will enter OS on from a trap or device interrupt
- Program a timer to interrupt if the process hasn't given back the core
  - Periodic timer is commonly used

Is the dispatcher a thread?

Thread Creation

- How does a thread get started?
- Create initial PCB thread state and stack to look like it had just context switch from the first instruction of the thread



Scheduling

- How does the dispatcher decide which thread to run next?
  - Simplest approach: ready queue (FIFO)
  - More complex: priorities or multiple queues
- Much more on this in Lecture 8