## Project 2: Thread dispatcher

- **Threads are the most common abstraction for concurrency**
- **Can implement threads inside the OS ("kernel threads")**
  - Thread create, exit, yield, etc., could be system calls
- **Or can implement threads on top of OS ("Green threads")**
  - This is what you will do for this project
  - Simplifying matters, you will assume only one CPU core
- **OS kernels themselves typically contain threads**
  - Each process managed by a corresponding thread in the kernel
  - Implementation of intra-kernel threads much like Green threads
  - So project will also explain part of how kernels work
- **Goal of project: Implement simple preemptive FIFO scheduler**



## Background: C++ lambda expressions

*[captures] (args) {body}* (inferred return type)

*[captures] (args) -> return-type {body}*

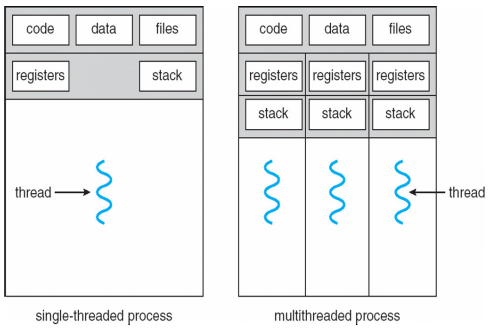
- **Lambda expressions create an unnamed function-like object**
  - Example: `[](int a, int b) -> int { return a+b; }`
  - Or implicit return type: `[](int a, int b) { return a+b; }`
- **Can also *capture* enclosing variables and refer to them**

```
int x = 7, y = 1;
// Capture x by reference, capture y by const copy
auto fn = [&x,y]() { x += y; };
y = 1000;
fn(); // x is now 8, because y was copied into lambda
```

- **Can cast lambda to `std::function<ret(args...)>` type**
  - `std::function<int(int,int)> adder = [](int a,int b){ return a+b; };`
- **Demo:** `lambda.cc`



## Threads vs. processes

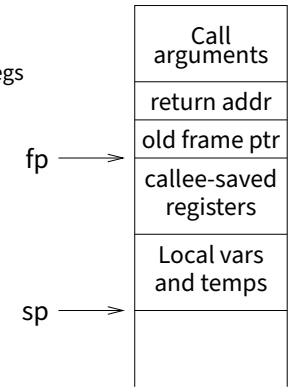



- **A thread is a schedulable execution context**
  - Program counter, stack, registers, …
- **A process contains an address space, file descriptors, plus one or more threads executing instructions**
- **Demo:** `thread-v-process.cc`



## Background: Calling conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs
  - But must restore *callee-saved* ones to original value upon return
- ***sp* register always base of stack**
  - Frame pointer (*fp*) is old *sp* (unless optimized out)
- **Local variables stored in registers and on stack**
- **Function arguments go in caller-saved regs and on stack**






## Procedure calls

**Procedure call**

save active caller registers
place arguments in registers/stack
call `foo` (pushes pc)
→ save needed callee registers
…do stuff…
restore callee saved registers
jump back to calling function
→ restore stack+caller regs.

- **Caller must save some state across function call**
  - Return address, caller-saved registers
- **Other state does not need to be saved**
  - Callee-saved regs, global variables, stack pointer
- **Demo:** `procedure.cc`



## Threads vs. procedures

- **Threads may resume out of order**
  - Cannot use LIFO stack to save state
- **General solution: one stack per thread**
  - Switching threads amounts to switching stacks

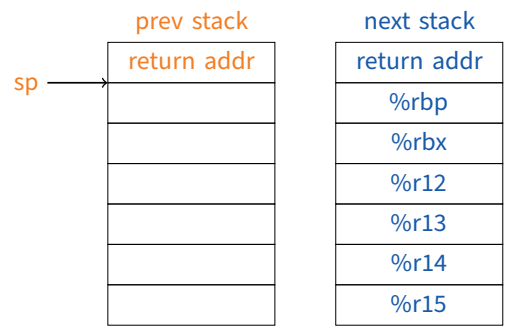
# Stack switching

- **Each thread has its own call stack pointed to by sp**
- **How to implement stack switch?**
  - Push all registers to save onto stack
  - Save the stack pointer where you can find it again
  - Set the stack pointer to saved value from another stack
  - Pop all saved registers from stack (except now from next stack)
  - Return (except to return address from next stack)
- **How to create a new stack?**
  - Allocate memory for it (nothing special about memory required)
  - Synthesize a call stack that returns to startup function
  - Make sure startup destroys thread if main function returns
- **Project gives you these functions:**

```
using sp_t = std::uintptr_t *;
sp_t stack_init(void *stack, size_t bytes, void(*start)());
void stack_switch(sp_t *prev_sp, const sp_t *next_sp);
```

---

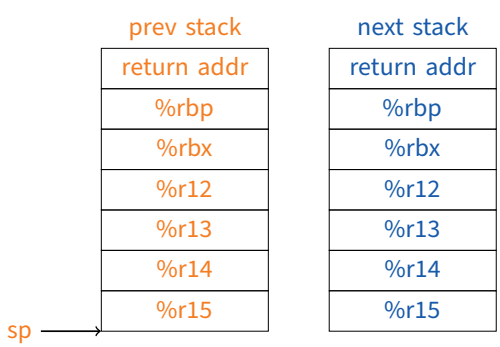
# `stack_switch` function



- **Demo:** `coroutine.cc`

---

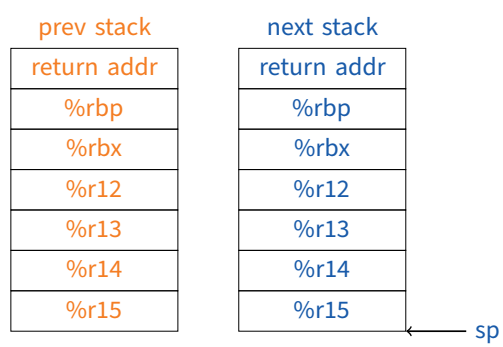
# `stack_switch` function



- **Demo:** `coroutine.cc`

---

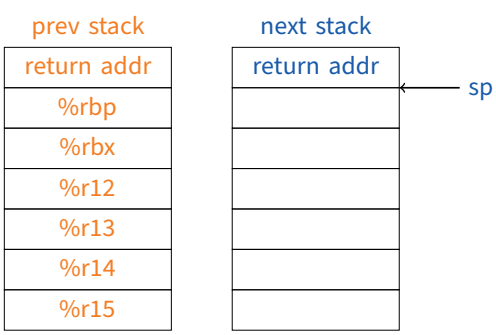
# `stack_switch` function



- **Demo:** `coroutine.cc`

---

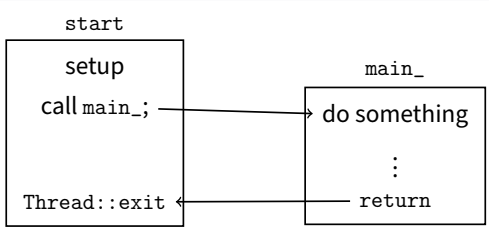
# `stack_switch` function



- **Demo:** `coroutine.cc`

---

# Thread initialization



- `stack_init` **takes a** `void(*)()` **(start function)**
  - Start function part of your library, same for all threads
  - Must set up environment and deal with thread returning
  - No lambda, so no captures; global current thread may be useful
- `Thread::create` **takes a** `std::function<void()>` **(thread main)**
  - Thread main supplied by user of thread library
  - Allows users to initialize threads with lambdas

## Cooperative threads

- `Thread` **class bundles all info about thread**
  - Mandatory: stack pointer
  - Possibly useful: memory for stack (for deleting), initial `std::function` to run, pointers for queues
- **Global variable with current thread is useful to have**
- **At lowest level, to switch threads switch stacks**
- **Demo:** `threadlet.cc`
- **To create, delete, block, unblock threads, will want queue**
  - Can use `std::queue<Thread*>` or keep global head/tail pointers
- **Don't delete the memory of the current thread's stack**
  - Okay to keep one garbage stack around you delete next time



## Preemption

```
void timer_init(uint64_t usec, std::function<void()> handler);
bool intr_enabled();
void intr_enable(bool on);
class IntrGuard;
```

- `timer_init` **causes timer handler to be called periodically**
- **Warning: beware of data races!**
  - Disable interrupts when touching data used by multiple threads
  - Global data includes `curthread` and scheduler queue
- `IntrGuard` **makes it easy to disable interrupts**
  - Merely creating an `IntrGuard` object disables interrupts
  - Destroying the `IntrGuard` restores interrupts to previous state
- **Demo:** `preempt.cc`