CS111, Lecture 18
Dispatching and Scheduling

😷 masks recommended
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

This Lecture
Scheduling and Dispatching

Lecture 18
Scheduling and Preemption

Lecture 19
Preemption and Implementing Locks

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

• Understand the general mechanisms for switching between threads and when switches occur
• Explore the tradeoffs in deciding which threads get to run and for how long
Plan For Today

• Recap and continuing: Context Switching
• Thread States
• Scheduling Threads
Plan For Today

• Recap and continuing: Context Switching
• Thread States
• Scheduling 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 (registers) 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*!!
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;
}
Context switch: how do we switch from thread A3 to thread B1?
**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.
Step 3: load B1’s saved stack register from its thread state space.
Step 4: pop B1’s other registers from its stack space.
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 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**

1. Push all registers besides stack onto current thread’s stack

```assembly
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
```
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

4. Pop registers off the other thread’s stack
Context Switching

```assembly
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)
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

we start executing on one stack...

and end executing on another!
Context Switching

We enter via a call from a function in the current thread!

We exit to a call from a function in the new thread!
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.
Plan For Today

- Recap and continuing: Context Switching
- Thread States
- Scheduling 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

- Running
- Ready
- Blocked
When a thread is created, it starts out **ready**.
When the OS lets a thread run on a core, the thread goes to **running**.
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.
Maybe a thread is running and reaches a point where it can’t run anymore (e.g. waiting for file contents from disk). The thread will go to **blocked**.
Maybe a thread is running and reaches a point where it can’t run anymore (e.g., waiting for file contents from disk). The thread will go to **blocked**.
If the event the thread is waiting for happens, and a core is immediately available for it, it switches back to running.
If the event the thread is waiting for happens, but the thread can’t run yet, it switches to **ready**.
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.
Key question: if we have many ready threads, how do we decide who to run next, and for how long?
Plan For Today

• Recap and continuing: Context Switching
• Thread States
• Scheduling Threads
**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 (no timer).

**Problem:** thread could run away with core and run forever!
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?

Thought: we want to run many threads in the amount of time for human response time, so e.g. keystroke seems instantaneous. So why not make the time slice microscopically small?
**Round Robin**

**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? Why not make it microscopically small?

*If too small,* context switch costs are very high, waste cores

Why not make it very large?

*If too large,* slow response, threads can monopolize cores

Try to balance: usually in 5-10ms range, Linux is 4ms
How do we decide whether a scheduling algorithm is good?

- Minimize response time (time to useful result)
  - e.g. keystroke -> key appearing, or “make” -> program compiled
  - Assume useful result is when the thread blocks or completes

- Use resources efficiently
  - keep cores + disks busy
  - low overhead (minimize context switches)

- Fairness (e.g. with many users, or even many jobs for one user)
Recap

• Recap and continuing: Context Switching
• Thread States
• Scheduling Threads

Lecture 18 takeaway: Context switching switches from one thread’s stack to the next, saving and restoring registers. For scheduling, we want to minimize response time, use resources efficiently, and be fair.

Next time: preemption and implementing mutexes