CS 241: Systems Programming
Lecture 25. Threads

Spring 2026

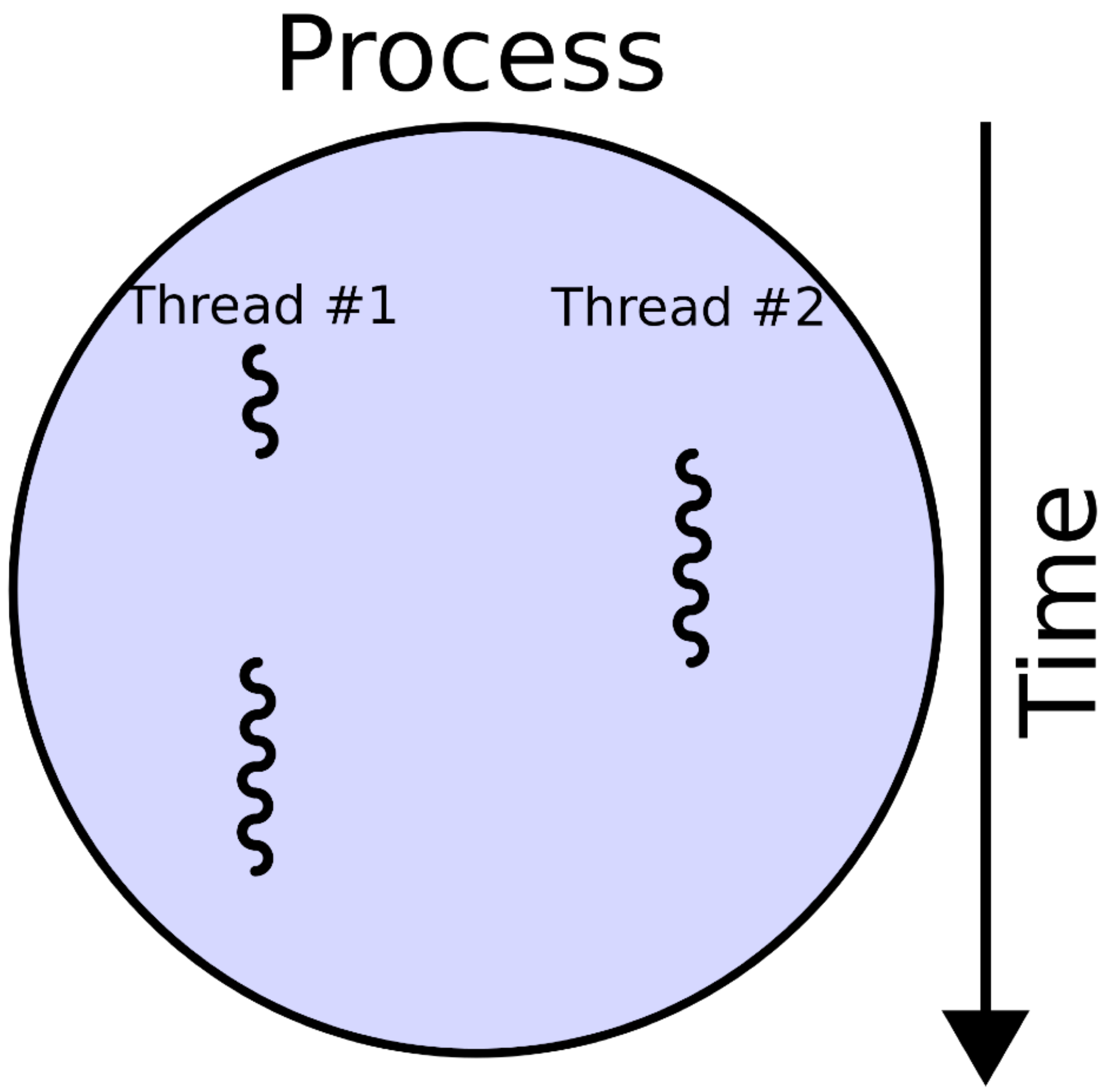
Prof. Stephen Checkoway

Threads

A process may be composed of multiple **threads of execution**

Each thread runs concurrently with but independent of other threads in the process

Threads are a bit like cooperating processes inside a process



https://en.wikipedia.org/wiki/Thread_(computing)#/media/File:Multithreaded_process.svg

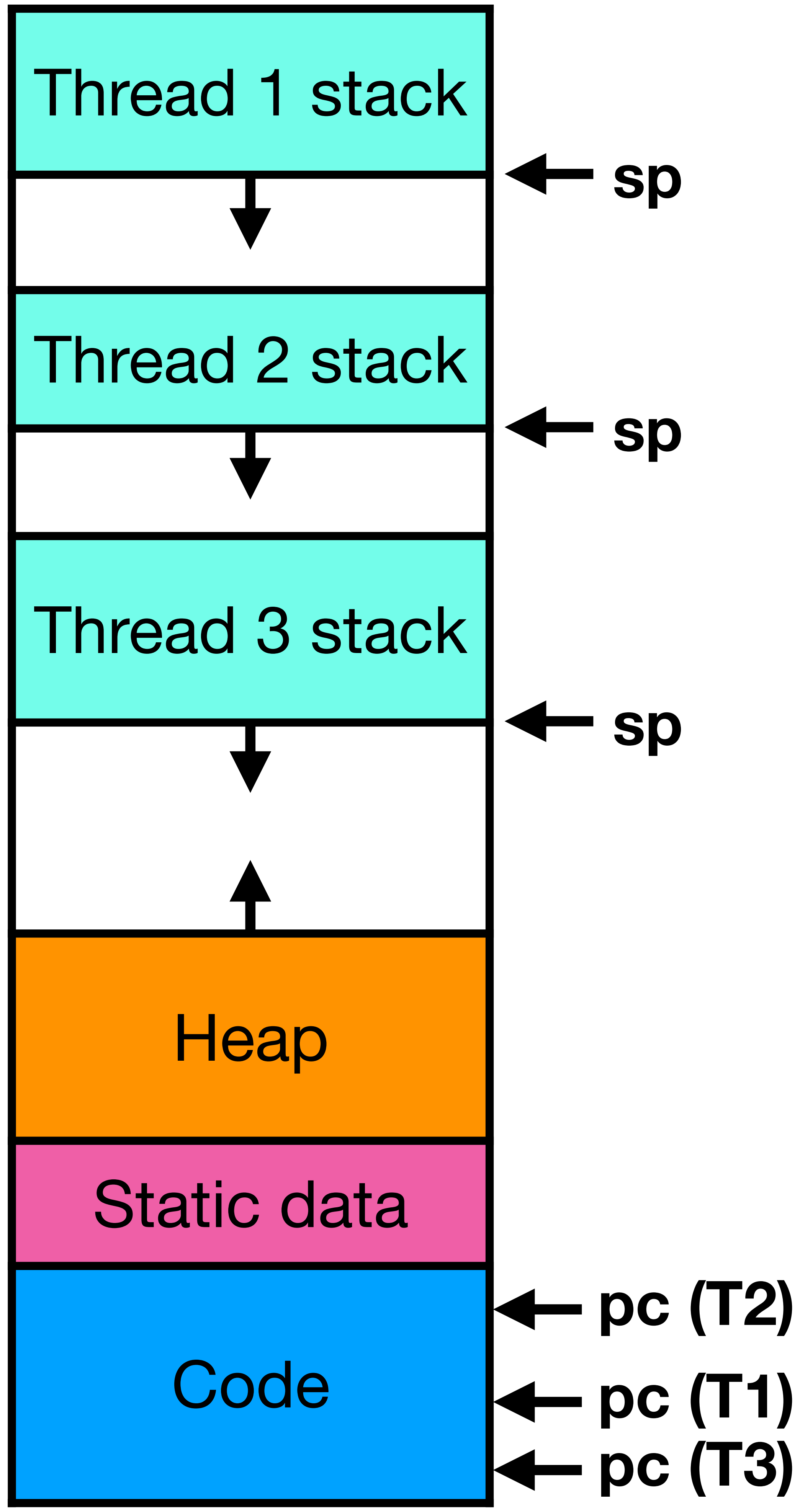
Relationship between threads

Each thread in a process shares all of the process's

- memory (data and code)
- open files
- permissions (e.g., to access the file system)
- user ID, group ID, process ID

Each thread in a process has its own

- function call stack with a stack pointer (sp)
- program counter (pc) indicating the next instruction to execute



Threads are useful in two key situations:

1. There is a significant quantity of data processing that needs to occur and the data can be processed independently (or with limited interaction)

2. The process is performing independent tasks at the same time

Think of some examples of each

A. Select A

Scheduling threads

There are multiple ways to implement threads

Most common: Each application thread is independently scheduled by the operating system's scheduler

Less common (green threads): The application manages threads itself using its own scheduler

There are pros and cons of each approach but Rust's native threads use the common approach

Creating a new thread

To create a new thread, use `std::thread::spawn()` and pass it a closure

```rust
use std::thread;

fn main() {
    let t = thread::spawn(|| {
        println!("Hello from the spawned thread!");
    });
    println!("Hello from the main thread!");
    t.join().unwrap();
}
```

Code in closure runs in the new thread

Output will appear in some, undefined order

The spawn function

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

The f parameter is the closure to call

- It will only be called a single time, so it's an `FnOnce`
- The closure returns some type `T` when the thread exits
- The `spawn()` function returns a `JoinHandle<T>` which can be used to wait for the thread to exit
- If there is no return value, then `T` is `()`

Joining a thread

The process exits when the main thread exits, regardless of what other threads are doing

If you want to wait for a thread to complete, call the .join() method on the JoinHandle returned by spawn()

```
    let t = thread::spawn(|| { … });
    // …
    t.join().unwrap();
```

`.join()` returns a Result which you'll want to unwrap

- The Err case is when the thread panics and there's usually not a good way to recover from that, so calling unwrap is fine

Thread run order

Absent explicit synchronization, threads run concurrently

Threads may be **preempted** (interrupted and another thread starts running) at any time

The order of output produced by threads is not defined (with respect to each other)

Example of unpredictable output

```
for thread_num in 0..10 {
    let t = thread::spawn(move || {
        for _ in 0..5 {
            println!("Hello from thread {thread_num}");
        }
    });
}
Partial output:
Hello from thread 5
Hello from thread 6
Hello from thread 3
Hello from thread 4
Hello from thread 4
```

Data races

A data race is when one thread tries to access a value in memory while another thread is writing to the same location

This can corrupt the data being written

- This is part of what makes programming with threads so difficult

Rust’s type system prevents data races in safe Rust code!

Data race example: unsafe Rust

```rust
fn main() {
    static mut COUNTER: i32 = 0;
    let mut threads = Vec::new();
    // Spawn 10 threads
    for _ in 0..10 {
        let t = thread::spawn(move || {
            for _ in 0..1000 {
                unsafe { COUNTER += 1 };
            }
        });
        threads.push(t);
    }
    // Wait for all 10 threads to complete
    for t in threads {
        t.join().unwrap();
    }
    println!("COUNTER's value, {}, should be 10000", unsafe { COUNTER });
}
```

unsafe to read/write
mutable global variables

Data race example: unsafe Rust

```rust
fn main() {
    static mut COUNTER: i32 = 0;
    let mut threads = Vec::new();
    // Spawn 10 threads
    for _ in 0..10 {
        let t = thread::spawn(move || {
            for _ in 0..1000 {
                unsafe { COUNTER += 1 };
            }
        });
        threads.push(t);
    }
    // Wait for all 10 threads to complete
    for t in threads {
        t.join().unwrap();
    }
    println!("COUNTER's value, {}, should be 10000", unsafe { COUNTER });
}
```

unsafe to read/write
mutable global variables

```
COUNTER's value, 8653, should be 10000
```

Value changes each time

In the previous example, the COUNTER value was not 10000 at the end.

What, specifically, went wrong?

A. The program printed the COUNTER value before all threads completed (why?)

B. The unsafe keyword indicates that the behavior isn’t defined

C. Multiple threads reading/writing the same location caused some increments to be lost (how?)

D. Rust prevented multiple threads from writing to the same location at the same time by preventing all but one thread from writing to the location at a time (which one?)

Communicating between threads

Using a mutable global variable was a bad idea

Some good options:

- Atomics — like the AtomicBool we saw with signal handlers
- Mutexes — Mutex = MUTual EXclusion, a way to limit access to a piece of data to only one thread at a time
- RwLock — Like a Mutex but supports multiple simultaneous readers
- Multi-producer, single-consumer (MPSC) queue — A one-way channel for multiple threads (the producers) to send data to a single thread (the consumer)

MPSC in Rust

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    // Create a simple streaming channel
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(10).unwrap();
    });

    let num = rx.recv().unwrap();
    println!("{num}");
}
```

tx is the transmitter (sender)
rx is the receiver

Data sent via tx.send() is received via rx.recv()

MPSC with multiple producers

We can clone the tx end of the channel for each thread and use it

Partially contrived example: Finding a bunch of prime numbers

- Prime numbers are *incredibly* useful in cryptography where you typically need pretty large primes
- For this example, let's just generate some small primes for simplicity

Approach

- Clone the tx for each thread
- Inside each thread in a loop, generate a random number and if it is prime, send it through the channel
- In the main thread, receive the primes and add them to a vector

Generating the primes

```
let (tx, rx) = mpsc::channel();
for _ in 0..8 {
    let tx = tx.clone();
    thread::spawn(move || {
        let mut rng = rand::thread_rng();
        loop {
            let num: u32 = rng.gen();

            if is_prime(num) {
                if tx.send(num).is_err() {
                    return;
                };
            }
        }
    });
}
```

I implemented the naïve “test division by every odd number up to the square root of num”

Receiving the primes in the main thread

```
// Get 20000 primes
let mut primes = Vec::new();
for _ in 0..20000 {
    primes.push(rx.recv().unwrap());
}
```

With 1 thread, it takes my laptop about 9 or 10 seconds to generate 20000 random (smallish) prime numbers

With 8 threads, it takes my laptop about 1.5 seconds

Scoped threads

What we want:

- To share local variable with multiple threads

Problem:

- Variable may go out of scope when the function that spawns the threads returns

```
fn compute_results() {
    let dataset = get_dataset();

    let t1 = thread::spawn(|| {
        for data in &dataset {
            // Compute some result
        }
        // Save result
    });
    let t2 = thread::spawn(|| {
        for data in &dataset {
            // Compute some other result
        }
        // Save result
    });

    t1.join();
    t2.join();
}
```

error[E0373]: closure may outlive the current function, but it borrows `dataset`, which is owned by the current function

The previous example gave an error that the closure may outlive the current function but the closure borrows dataset which is owned by the function.

The function ended by joining the threads. Is the error message correct?

```
fn compute_results() {
    let dataset = get_dataset();

    let t1 = thread::spawn(|| { … });
    let t2 = thread::spawn(|| { … });

    t1.join();
    t2.join();
}
```

A. Yes [Why?]

B. No [Why?]

Scoped threads!

Scoped threads let us declare a new scope for the threads such that the function spawning the threads cannot exit the scope until the threads have been joined

```
thread::scoped(|s| {
    s.spawn(|| { … } );
});
```

Inside the closure, s can be used to spawn threads that will be joined before the end of the thread::scoped() call

```
fn compute_results() {
    let dataset = get_dataset();

    thread::scope(|s| {
        let t1 = s.spawn(|| {
            for data in &dataset {
                // Compute some result
            }
            // Save result
        });
        let t2 = s.spawn(|| {
            for data in &dataset {
                // Compute some other result
            }
            // Save result
        });
    }); // threads are joined here
}
```