CS 4414/5416: Systems Programming

Threads and Message Passing

David Bindel

2026-10-24

Logistics

Upcoming

- P4 (netsim) revision
  - No check-in – work on lab tomorrow instead
  - Shortened assignment due Monday, 11/2
- Lab 9 is Friday: working on P4

Final Exam

- Final is Dec 16 at 9 AM
- Alternate time request is linked here (and is posted on Ed)

Recap

Where We Are

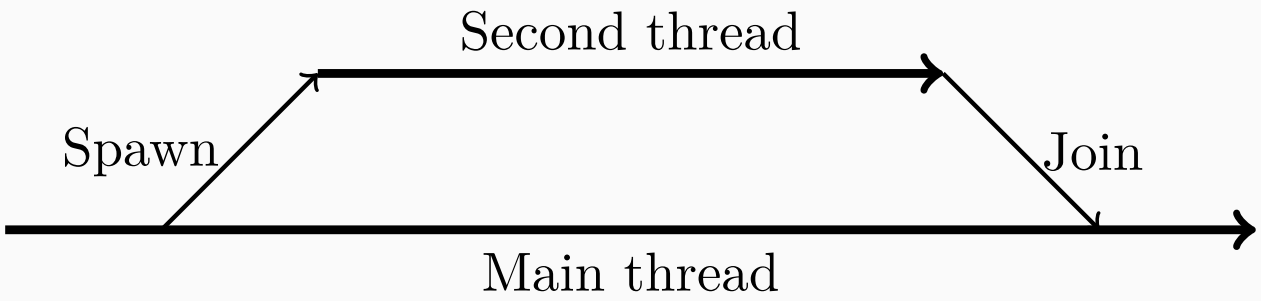
This module: Concurrent programming with threads

- Message-based concurrent programming
- Shared memory concurrent programming
- Synchronization and deadlock
- Concurrent network programming

This Week

- Tue: Concurrency, parallelism, and intro to Rust threads
- Today:
  - Borrowing and threads
  - Patterns with channels
  - Bounded channels, non-`std` crates
  - A network simulator project

Fork/Join



- `thread::spawn` takes `FnOnce() -> T`, returns `JoinHandle<T>`
- `handle.join()` returns `thread::Result<T>`
- Can also drop the handle without join
  - Result is a *detached thread*
  - Detached threads terminate on `main()` exit

Error Handling

```
type thread::Result<T> = Result<T, Box<dyn Any + Send + 'static>>
```

- Unrecoverable: `Err` variant on `JoinHandle`
  - `Any` is a trait to emulate dynamic typing
  - `Send` means safe to move between threads
  - Needs to have `'static` lifetime
- Recoverable: make closure return `T = Result<S,E>`
  - Get `Ok(Err(err))` for `err: E`

Channels

```
// mpsc channel signature
fn channel<T>() -> (Sender<T>, Receiver<T>);

// methods in Sender<T> / Receiver<T>
fn send(&self, x: T) -> Result<(), SendErr<T>>;
fn recv(&self) -> Result<T, RecvErr>;
```

Poll Everywhere!

Which of these is true?

(a) `send` takes ownership of the message argument

(b) Returns from `thread::spawn` need not satisfy `'static`

(c) `panic!()` in a thread will terminate the program

Safety and Threads

Marker Traits

- Rust has *marker traits* in `std::marker`
- Used as info about types vs for methods
- Some automatically supported (e.g. `Sized`)
- Others you derive (e.g. `Copy`)
- Two automatic traits for thread: `Send` and `Sync`

Ownership Transfer

Thread communication involves ownership transfer

- Sending across a channel does this
- So do arguments in / return values out
- Allocations in one thread to safely `drop` in another

Send

A `Send` type is safe to transfer across threads.

- Automatically derived in most cases
- Basic types, `String`, `TCPStream`, `Vec` are `Send`
- Also: `Receiver<usize>` and `Cell<usize>`
- `!Send`: `Rc<String>`, raw pointers

Sync

A type `T` is `Sync` if `&T` is `Send`

- More restrictive than `Sync`!
- Basic types, `String`, `TCPStream`, `Vec` are `Send`
- `!Sync`: `Receiver<usize>` and `Cell<usize>`
- Also `!Sync`: `Rc<String>`, raw pointers

Lifetimes

- Hard to reason about lifetimes across threads!
- Standard default: Transfer ownership vs references
- Transferred references should be `'static`
- `'static` means “contains no non-static references”
    - Owned types generally satisfy `'static` bound

Deliberate leaks

How do we get static data?

- String constants are static...
- Otherwise: `leak` on containers (`Box`, `String`, `Vec`)
- Returns `'static` mutable ref or slice to contents
- Use cautiously!

Scoped threads

```
use std::thread;

fn main() {
    let s = String::from("Hello");
    thread::scope(|scope| {
        scope.spawn(|| {
            println!("This thread knows about {s}");
        });
    });
}
```

- Sometimes we want non-`'static` data!
- When `thread::scope` completes, all threads guaranteed to be joined

Ask Me a Question!

Channel Patterns

The Ack Pattern

Channels are *unidirectional* – how to get replies?

- `Ack` pattern: `send` data containing a `Sender`
- Recipient `send`s data on the receiver
- One-shot messaging gets special fast path in Rust
  - Switch methods when sending second message
  - Also switch methods when sender is `clone`d

The Ack Pattern

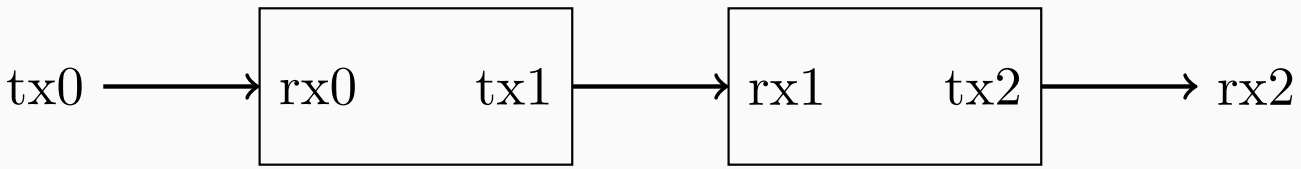
```
use std::sync::mpsc::{channel, Receiver, RecvError, Sender};
use std::thread;

struct Command(String, Sender<String>);

fn echo_client(id: usize, tx: Sender<Command>) {
    let (response_tx, response_rx) = channel();
    tx.send(Command(format!("Ping {id}"), response_tx)).unwrap();
    println!("{id} received: {}", response_rx.recv().unwrap());
}

fn echo_server(server_rx: Receiver<Command>) -> Result<(), RecvError> {
    loop {
        let Command(s, tx) = server_rx.recv()?;
        println!("Server received: {s}");
        tx.send(s).unwrap();
    }
}
```

The Pipeline Pattern



The Pipeline Pattern

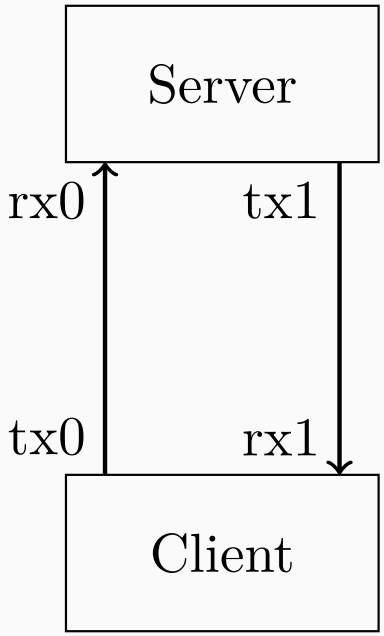
```
use std::thread;
use std::sync::mpsc::{channel, Sender, SendError, Receiver};

fn stage<T>(channel_in: Receiver<T>,
            channel_out: Sender<T>, f:
            impl Fn(T) -> T) -> Result<(), SendError<T>> {
    while let Ok(data) = channel_in.recv() {
        channel_out.send(f(data))?;
    }
    Ok(())
}
```

The Pipeline Pattern

```
fn main() {
    let (channel_in, channel_out1) = channel();
    let (channel_in2, channel_out2) = channel();
    let (channel_in3, channel_out) = channel();
    thread::spawn(move || {
        stage(channel_out1, channel_in2, |x: usize| 2*x)
    });
    thread::spawn(move || {
        stage(channel_out2, channel_in3, |x: usize| x+1)
    });
    channel_in.send(5).expect("Channel died prematurely");
    channel_in.send(10).expect("Channel died prematurely");
    assert_eq!(channel_out.recv().unwrap(), 11);
    assert_eq!(channel_out.recv().unwrap(), 21);
}
```

The Client-Server Pattern



The Client-Server Pattern

```
use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread;

fn echo_server(rx: Receiver<String>, tx: Sender<String>) {
    while let Ok(s) = rx.recv() {
        let reply = format!("Echo: {s}");
        let Ok(_) = tx.send(reply) else {
            return;
        };
    }
}
```

The Client-Server Pattern

```
fn main() {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();
    thread::spawn(move || echo_server(rx1, tx2));
    tx1.send(String::from("Hello")).unwrap();
    println!("Received: {}", rx2.recv().unwrap());
    tx1.send(String::from("World")).unwrap();
    println!("Received: {}", rx2.recv().unwrap());
}
```

Worker Queue Pattern

- Work queue at a server (e.g. work queue)
- Worker clients interact by
    - Sending work requests
    - Sending work results
- Alternative: Queue is client, workers are servers

Ask Me a Question!

Beyond

Bounded channels

- `channel` produces an *asynchronous* channel
  - Will allocate space for pending messages
  - `send` never blocks on calling thread
- `sync_channel(b)` produces a *synchronous* channel
  - Buffers up to `b` messages
  - `send` blocks until there is buffer space

Crossbeam and Rayon

- Crossbeam
  - **deque** and MPMC queues
  - Epoch-based garbage collector
  - And more!
- Rayon
  - A *data parallelism* library
  - Supports fork/join, scopes, thread pools
  - Also parallel iterators, parallel sort, parallel collections

Poll Everywhere!

What is the default channel type from `std::mpsc::channel`?

(a) A bounded, synchronous channel with a fixed buffer
(b) An unbounded, asynchronous channel that does not block on `send`
(c) A multi-producer multi-consumer channel
(d) A single-producer multi-consumer channel

A Network Simulator

High-Level Picture

Want to understand performance of networked applications

- Maybe with hardware we don't have
- Maybe at a scale we don't have
- Maybe in presence of rare transient errors

Hard to do in practice. Simulate!

Moving pieces

- Actors
  - *Manager* thread keeps global state
  - *Process* threads simulate node apps
- Timeline comprising
  - Log of past events
  - Priority queue of future events
- Collection of messages
- Virtual network for time of comm events

The Story So Far

https://github.coecis.cornell.edu/cs4414-f26/p4-netsim

- This version got too ambitious!
- Working on scaling it down before release

Ask Me a Question!

Wrap-Up

Summary

- Rust safety model enables “fearless concurrency”
  - Relatively – still plenty of room to screw up!
- Several common channel-based patterns (mirroring IPC)
- There is a wider ecosystem beyond!
- A realistic problem brings interesting challenges

Up Next

- Next week: Concurrency with shared state
- Two weeks: Deadlock and live-lock; multithreaded network programming

Outro

- On Ed: What is the muddiest point from this week?
- Remember to fill the alternate final time form if relevant