CS 4414/5416: Systems Programming

Intro to Asynchronous Programming

David Bindel

2026-11-10

Logistics

Upcoming

- P5 (Tetris) posted! **Groups of 3-4** preferred
  - Yes, preferred vs required
  - Check-in is Nov 11
  - Final is Nov 18
- Lab 12 is Friday: practice with async/await

Final Exam Plans

- Targeting practice out before Thanksgiving
- Early exam is Dec 7, 6-8 PM
    - If you filled the form, you're set!
    - If not, and you want this, email me ASAP
- Regular final is Dec 16, 9-11 AM

Where We Are

This module: Asynchronous programming and company

- Asynchronous programming essentials
- Behind the scenes and nonblocking I/O

Poll Everywhere!

Which of these might be most appropriate for a web browser to interact with a game server?

(a) Long polling
(b) Server-sent events
(c) Web sockets

Lay of the Land

Recall

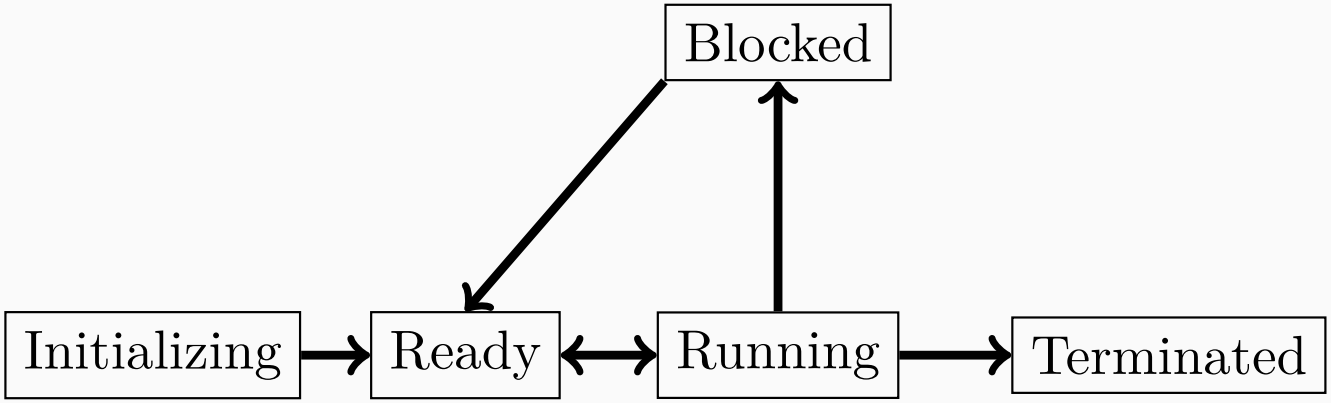
Concurrency vs parallelism:

- Concurrency: *logically* overlapping
- Parallelism: *physically* overlapping
- Parallelism implies concurrency, but not vice-versa

Why Concurrency?

- Because sometimes it is “natural”
- For parallelism
- To do useful work during I/O
- Can mix all of the above!

Process Picture



Pre-Emption

To share a core pre-emptively:

- A process gets a time slice
- When timer interrupt ends slice
  - Save process info, move to “ready”
  - Schedule another process that is “ready”
- What about “waiting” state?

Waiting

Three ways to exit running state

- Termination
- Pre-emption at end of time slice
- Yield CPU to wait for an *event*
    - Could be I/O or a timeout

Concurrency Approaches

Spectrum based on style and workload:

- How much compute? Throughput?
- Many small tasks or few big ones?
- Scheduling with pre-emption or cooperative?
- Organize around time order or by events?

Concurrency Approaches

Spectrum based on style and workload:

- One thread per core
- Several threads/core
- *Many* interacting actors

Concurrency Approaches

One thread per core

- Great for compute-intensive workloads
  - This is all about parallelism
  - Cache stays warm, little scheduling overhead
- Have to manually manage load balance?
- Twiddle thumbs during I/O?

Concurrency Approaches

Several threads/core

- Better for I/O, high throughput situations
- Context switch still not free

Concurrency Approaches

Run-time scheduled tasks or handlers

- Dominated by cooperative sharing
  - Does not rule out parallelism!
- Good for waiting on many I/O streams

Asynchronous Approaches

- Explicit `select` loops (event loops)
  - Loop to ask the OS what file handles have data
  - When data is ready, process it and go back to loop
  - Modern takes: `poll`, `epoll`, `kqueue`
- Event-based models
  - Register *event handlers* for incoming data, clicks, etc
  - Still feels like “programming by state machine”
- Task-based / coroutine-based models (our focus)

Terminology

With some ambiguity in all cases:

- Green threads
- Coroutines
- Tasks
- Generators
- Futures

Green Threads

Term from Java (designed by Green Team at Sun)

- Cooperative thread abstraction, managed by run-time
- Some run-time calls (e.g. I/O) yield control to scheduler
- Lighter weight than full threads (but no pre-emption)

Coroutines

A coroutine is a subroutine that can be suspended

- Can call, suspend, and re-enter an invocation
- Local data persists across calls

Many variations across languages!

Generator

A generator is a common *type* of coroutine

- Use `yield` to return multiple outputs
- Common for iterators in some languages (Python)

Rust uses `generator`s as an internal

Tasks

- A task is a coroutine + execution state
- Often like to dynamically create many tasks!
- This is useful for parallelism or concurrency

Future (aka Promise)

- Also known as a delayed or deferred result
- Idea: A handle for a result
  - Split *start* of execution from *result*
  - E.g. think of a `JoinHandle` in Rust

A Rust `Future` is between this and a coroutine

Tricky Business

- Even languages with “same” model differ a lot!
- See “A Design Space Exploration of Async/Await”
    - Also a nice blog post

Ask Me a Question!

Async Rust

Recall Last Thursday

- Did some examples with axum
- “Ignore `async / await`, we’ll come back to it next week!”
- OK, now it’s next week!

Hello world!

```rust
use tokio::time::{sleep, Duration};

async fn hello(id: usize, time: u64) {
    println!("{id}: Hello, ... (and pause for {time} s)");
    sleep(Duration::from_secs(time)).await;
    println!("{id}: world!");
}

#[tokio::main]
async fn main() {
    // All guaranteed to run on the same thread
    tokio::join!(hello(0,2), hello(1,1), hello(2,1));
}
```

Moving Pieces

- Tokio provides an async run-time (no Rust default)
  - Also: async version of std library, lots of extras
- `async` functions and blocks return a `Future`
- `await` awaits completion of an operation
  - Only from an `async` function or block
  - Does not block thread

Async Functions

`async fn hello(...)` is syntactic sugar for

```
fn hello(id: usize, time: u64) -> impl Future<Output=()> {
    async move {
        println!("{id}: Hello, ... (and pause for {time} s)");
        sleep(Duration::from_secs(time)).await;
        println!("{id}: world!");
    }
}
```

where an `async` block compiles to a `Future` instance

The Future Trait

```
use std::pin::Pin;      // Make sure Future is not moved in memory
use std::task::Context; // For scheduling to poll again

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>,
            cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

`cx` used to access `Waker` (more on this Thursday)

Cold Start

```rust
use tokio::time::{sleep, Duration};

async fn hello(id: usize, time: u64) {
    println!("{id}: Hello, ... (and pause for {time} s)");
    sleep(Duration::from_secs(time)).await;
    println!("{id}: world!");
}

#[tokio::main]
async fn main() {
    let h = hello(0, 1);
    println!("Starting");
    h.await;
}
```

What is the output?

Cold Start

```
#[tokio::main]
async fn main() {
    let h = hello(0, 1);
    println!("Starting");
    h.await;
}
```

- A `Future` doesn't execute itself!
- `h.await` pauses current `async` function until result of `h` is ready

Futures and Streams

```
trait Stream {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

- Only get *one* output with `await`
- Can establish a channel for richer communication
- Or use `Stream` (cf `async_stream` crate)

A Peek Behind the Curtain

- A `Future` or `Stream` compiles to a state machine
- States are:
  - `Start`: Initial state
  - `Done`: Completed state
  - A state for each `await` or `yield`
- Poll operation checks for transitions
- Code executes on the transitions

(A little like the state machine for a process)

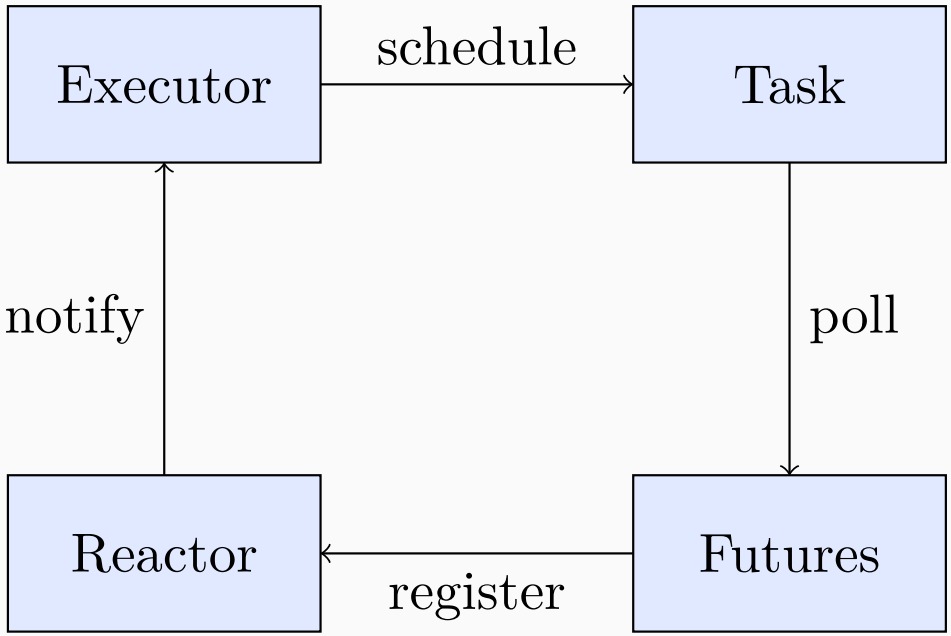
Executor and Tasks

- Examples: `tokio`, `smol`
- Manage/schedule/execute collection of tasks
  - A *task* is the concurrent unit
  - Contains a top-level future
  - May also have one or more nested tasks
- Executor advances scheduled tasks, which `poll` futures
- The `#[tokio::main]` lauches `async main` in executor

Reactors (Event Loops)

- Only want to be awakened when something happens!
- Reactors manage this (say when to `poll`)
- React to OS event (e.g. data on a socket) or other
  - Build on `epoll` / `kqueue` / etc (maybe via `mio`)
  - Mark as runnable any tasks that are attending

The Full Picture



Adapted from

https://rust-training.ferrous-systems.com/latest/slides/async-intro#/17

Ask Me a Question!

Poll Everywhere!

Which of these best describes a Rust task?

(a) A type of thread

(b) An execution state for a future (or futures)

(c) Another name for an event loop

Wrap-Up

Up Next

- P5 (Tetris) posted! **Groups of 3-4** preferred
  - Check-in is Nov 11
  - Final is Nov 18
- Lab 11 is Friday: multithreaded web service
- Next week: Start asynchronous programming