Async Rust Is A Bad Language

Sep 8, 2023

But to get at whatever the hell I mean by that, we need to talk about why `async` Rust exists in the first place. Let's talk about:

Modern Concurrency: They're Green, They're Mean, & They Ate My Machine

Suppose we want our code to go fast. We have two big problems to solve:

1. We want to use the whole computer. Code runs on CPUs, and in 2023, even my phone has eight of the damn things. If I want to use more than 12% of the machine, I need several cores.

2. We want to keep working while we wait for slow things to complete instead of just twiddling our thumbs. Sending a message over the Internet, or even opening a file[1]

takes eternities in computer time—we could literally do *millions* of other things meanwhile.

And so, we turn to our friends *parallelism and concurrency*. It's a favorite hobby of CS nerds to quibble over distinctions between the two, so to oversimplify:

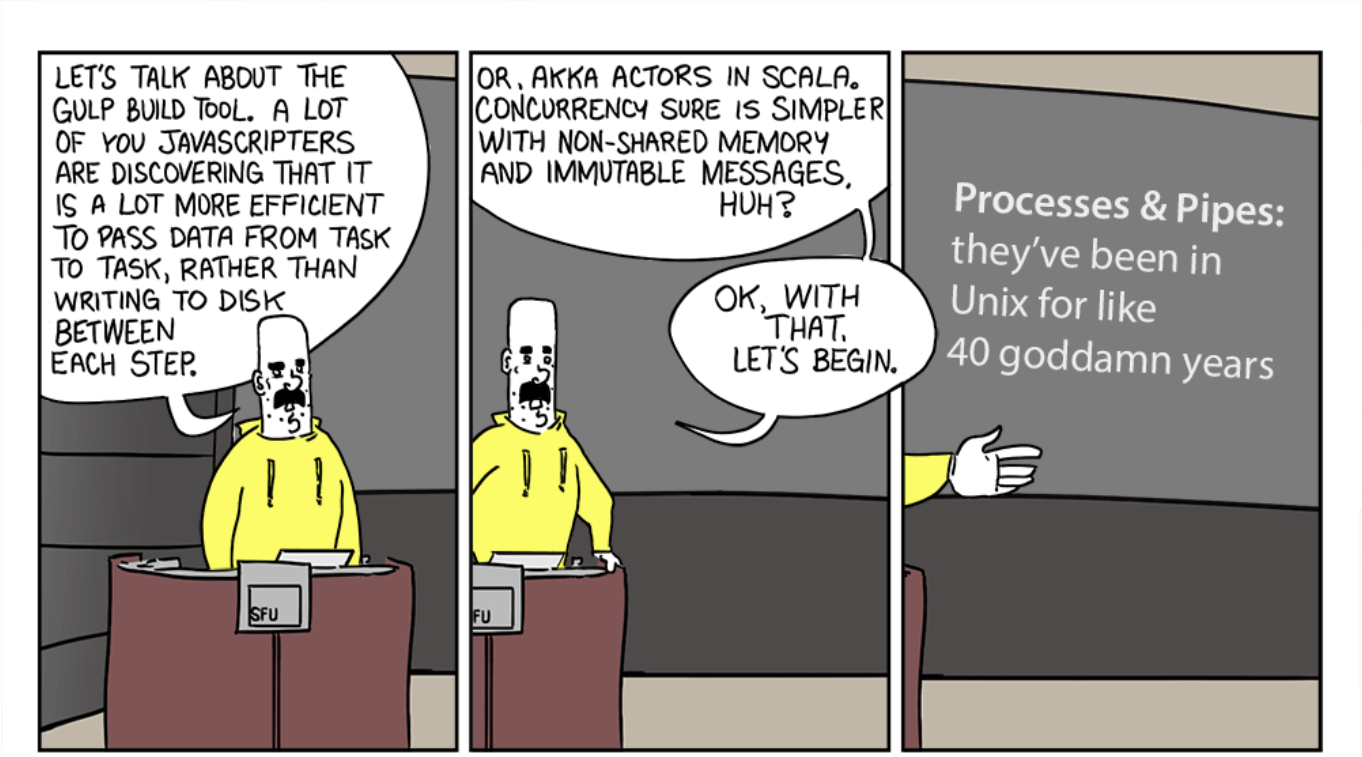
Parallelism is about running code *in parallel* on several CPUs.

Concurrency is about breaking a problem into separate, independent parts.

These are not the same thing —single-core machines have been running code concurrently for half a century now—but they are related. So much online *well akshually*-ing ignores how we often break programs into concurrent pieces *so that* those pieces can run in parallel, and interleave in ways that keep our cores crunching! (If we didn't care about performance, why would we bother?)

How do I concurrency?

One of the simplest ways to build a concurrent system is to split code into multiple processes. After all, the operating system is a lean, mean, concurrency machine, conspiring with your hardware to make each process think it has the whole box to itself. And the OS's scheduler gives us parallelism for free, running *time slices* of any process that's ready on an available CPU core. Once upon a time this was *the* way, and we still employ it today whenever we pipe shell commands together.



All hail CubeDrone

But this approach has its limitations. Inter-process communication is not cheap, since most implementations copy data to OS memory and back.[2]

Mutex-Based Concurrency Considered Harmful, or, *Hoare Was Right*

> Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms.
> – Ned Batchelder

We can avoid these overheads using *threads*—processes that share the same memory. Common wisdom teaches us to connect them with mysterious beasts, like *mutexes*, *condition variables*, and *semaphores*. This is a dangerous game! Simple mistakes will plague you with *race conditions* and *deadlocks* and other terrible diseases that fill your code with bugs, but only on Tuesdays when it's raining and the temperature is is a multiple of three. And god help you if you want to learn how this stuff actually works on modern hardware.[3]

There is Another Way. In his 1978 paper, *Communicating Sequential Processes*, Tony Hoare suggested connecting threads with queues, or *channels*, which they can use to

send each other messages. This has many advantages:

- Threads enjoy process-like isolation from the rest of the program, since they don't share memory. (Bonus points for memory-safe languages that make it hard to accidentally scramble another thread!)

- Each thread has a very obvious set of inputs (the channels it receives from) and outputs (the channels it sends to). This is easy to reason about, and easy to debug! Instrument the channels for powerful visibility into your system, measuring each thread's throughput.

- Channels *are the synchronization*. If a channel is empty, the receiver waits until it's not. If a channel is full, the sender waits. Threads never sleep while they have work to do, and gracefully pause if they outpace the rest of the system.

After decades of mutex madness, many modern languages heed Hoare's advice and provide channels in their standard library. In Rust, we call them `std::sync::mpsc::sync_channel`.

Most software can stop here, building concurrent systems with threads and channels.[4] Combine them with tools to parallelize CPU-intensive loops (like Rust's Rayon or Haskell's `par`), and you've got a powerful cocktail.

But…

Ludicrous Speed, go!

Some problems demand a *lot* of concurrency. The canonical example, described by Dan Kegel as the *C10K problem* back in 1999, is a web server connected to tens of thousands of concurrent users. At this scale, threads won't cut it—while they're *pretty* cheap,[5] fire up a thread per connection and your computer will grind to a halt.

To solve this, some languages provide a concurrency model where:

1. Tasks are created and managed in *userspace*, i.e., without the operating system's help.

2. A *runtime* schedules these tasks onto a pool of OS threads, usually sized so that each CPU core gets a thread, to maximize parallelism.

This scheme goes by many names—*green threads, lightweight threads, lightweight processes, fibers, coroutines*, and more—complete with pedantic nerds endlessly debating the subtle differences between them.

Rust comes at this problem with an "async/await" model, seen previously in places like C# and Node.js.[6] Here, functions marked `async` don't block, but immediately return a *future* or *promise* that can be awaited to produce the result.

```
fn foo() -> i32 { /* returns an int when called */ }

async fn bar() -> i32 { /* returns a future we can .await to get an
```

`pain.await`

On one hand, futures in Rust are exceedingly small and fast, thanks to their *cooperatively scheduled, stackless* design. But unlike other languages with userspace concurrency, Rust tries to offer this abstraction while *also* promising the programmer total low-level control.

There's a fundamental tension between the two, and the poor `async` Rust programmer is perpetually caught in the middle, torn between the language's design goals and the massively-concurrent world they're trying to build. Rust attempts to statically verify the lifetime of every object and reference in your program, all at compile time. Futures promise the opposite: that we can break code *and the data it references* into thousands of little pieces, runnable at any time, on any thread, based on conditions we can only know once we've started! A future that reads data from a client should only run when that client's socket has data to read, and no lifetime annotation will tells us when that might be.

Send help

Assuring the compiler that everything will be okay runs into the same challenges you see when working with raw threads. Data must either be marked `Send` and moved, or passed through references with `'static` lifetimes. Both are easier said than done. Moving (at least without cloning) is often a non-starter, since it's common in `async` code to spawn many tasks that share common state. And references are a pain too—there's no `thread::scope` equivalent to help us bound futures' lifetimes to anything short of "forever".

```
fn foo(&big, &chungus)
```

is out,

```
async fn foo(&BIG_GLOBAL_STATIC_REF_OR_SIMILAR_HORROR, sendable_cl
```

is in.

And unlike launching raw threads, where you might have to deal with these annoyances in a handful of functions, this happens *constantly* due to `async`'s viral nature. Since any function that calls an `async` function must itself be `async`,[7] you need to solve this problem everywhere, all the time.

Just Arc my shit up

A seasoned Rust developer will respond by saying that Rust gives us simple tools for dynamic lifetimes spanning multiple threads. We call them "atomic reference counts", or `Arc`. While it's true that they solve the immediate problem—borrows check and our code compiles—they're far from a silver bullet. Used pervasively, `Arc` gives you the world's worst garbage collector. Like a GC, the lifetime of objects and the resources they represent (memory, files, sockets) is unknowable. But you take this loss without the wins you'd get from an actual GC!

Don't buy the "GC is slow" FUD—the claim is a misunderstanding of latency vs. throughput at best and a bizarre psyop at worst. A modern, moving garbage collector gets you more allocation throughput, less fragmentation, and means you don't have to play Mickey Mouse games with weak pointers to avoid cycle leaks. You can even trick systems programmers into leveraging GC in one of the world's most important software projects by calling it "deferred destruction". More on that another day.

Other random nonsense

- Because Rust coroutines are stackless, the compiler turns each one into a state machine that advances to the next `.await`.[8] But this makes any recursive `async` function a recursively-defined type! A user just trying to call a function from itself is met with inscrutable errors until they manually box it or use a crate that does the same.

- There's an important distinction between a *future*—which does nothing until awaited—and a `task`, which spawns work in the runtime's thread pool… returning a future that marks its completion.

- There's nothing keeping you from calling blocking code inside a future, and there's nothing keeping that call from blocking the runtime thread it's on. You know, the entire thing we're trying to avoid with all this `async` business.

Running away

Mixed together, this all gives `async` Rust a much different flavor than "normal" Rust. One with many more gotchas, that is harder to understand and teach, and that pushes users to either:

- Develop a deep understanding of how these abstractions actually work,[9] writing complicated code to handle them, or

- Sprinkle `Arc`, `Pin`, `'static`, and other sacred runes throughout their code and hope for the best.

Rust proponents (I'd consider myself one!) might call these criticisms overblown. But I've seen whole teams of experienced developers, trying to use Rust for some new project, mired in this minutia. To whatever challenges teaching Rust has, `async` adds a whole new set.

The degree to which these problems *just aren't a thing* in other languages can't be overstated either. In Haskell or Go, "async code" is just normal code. You might say this isn't a fair comparison—after all, those languages hide the difference between blocking and non-blocking code behind fat runtimes, and lifetimes are handwaved with garbage collection. But that's exactly the point! These are pure wins when we're doing this sort of programming.

Maybe Rust isn't a good tool for massively concurrent, userspace software. We can save it for the 99% of our projects that don't have to be.

---

1. …a file which could also be on the other side of the Internet! Thanks, NFS! ↩

2. We could cut down on IPC overhead by sharing memory between the processes, but this gives away one of the main advantages of multiple processes: that the OS isolates them from each other. ↩

3. Mara Bos recently put out a fantastic book that despite targeting Rust specifically, does a wonderful job of explaining the fundamentals of low-level concurrency in any language. If you don't have time for a whole book, I've done my best to sum it up in a few pages. ↩

4. Of course I'm simplifying here. Not every program can be expressed as a DAG, and you'll still find good occasions for other primitives—say, atomic flags to indi-

cates changes in global state. Still, Hoare's model is a great default, and I've always found it helpful to think about how data flows through my system. ↩

5. Each thread has a 4kB control block in Linux, and switching between threads requires a trip to the operating system scheduler. This *context switch* to the OS's memory is much more expensive than a normal function call. ↩

6. Uniquely, Rust doesn't provide a runtime for its futures in the language, delegating instead to libraries like Tokio. This is great for users—Rust's build tooling (`cargo`) and ecosystem gives developers the freedom to choose alternatives that better suit unique environments they find themselves in. But it's a detail that's largely immaterial to our discussion; one can imagine a world where Tokio is built into the language and all the same rules apply. ↩

7. You can break the chain by commanding the entire runtime to block on the completion of a future, but you probably shouldn't do this pervasively since it isn't composable. If a function blocks on a future, and that future calls a function that blocks on a future, congrats! The runtime panics! ↩

8. Learn more in Without Boats's *Futures and Segmented Stacks* or the C++ paper P1364: *Fibers under the magnifying glass*. ↩

9. Amos Wenger AKA fasterthanlime's *Pin and Suffering* is a fantastic and snarky intro. ↩

Tima Kinsart
a month ago

A year ago I wrote a similar blog post about async Rust called "Rust Is Hard, Or: The Misery of Mainstream Programming". I pretty much agree with you on the subject matter: Rust makes it hard to write async code without going through the worst language parts. I advice anyone just to use `Arc` anytime you are not able to resolve a lifetime error within a minute: it will just make your life easier.

3 0 Reply • Share ›

Jon Forrest
a month ago

"threads - processes that share the same memory."

It's misleading to say that a thread is a process. This will confuse readers.

"Threads enjoy process-like isolation from the rest of the program,
since they don’t share memory."

This contradicts what you said above.

There's no memory protection among the threads in
a process. That's why the mysterious beasts you mention, such as
mutexes, condition variables, and semaphores are necessary.

2 0 Reply • Share ›

Sarfaraz Nawaz → Jon Forrest
14 days ago

>>> "Threads enjoy process-like isolation from the rest of the program,
since they don’t share memory."

> This contradicts what you said above.

The author said "process-like isolation" in the context of message passing. That is, if you use message passing (instead of sharing memory) for communication between threads, then in this usage of threads, they're more like process!

0 0 Reply • Share ›

Adrian May
a month ago

Perhaps the problem is that you're not making up your mind which parallelism strategy to use. First you said mutexes etc are risky so you prefer channels, but then you want to share data between threads. The channelly way would have been to coordinate that data in a thread that takes edit requests through a channel, but since Rust offers defanged mutexes on a plate you decided to coordinate your data the mutexy way after all. Rust actually gave you a bullet-proof solution to all the mixed design you asked of it but you find your own solution complex. Is this really Rust's fault? It's also wrong to say that Haskell would magically do the kind of things you want seamlessly - you'd actually do transactional memory instead, which is a totally different ball game. Both languages annoy me by completely ignoring read-copy-update as seen in the Linux kernel.

1 1 Reply • Share ›

Bernard
16 days ago

Is it possible to add GC to Rust? Can we have a set of pointers that live in a GC memory pool? Then get all the effects of GC only where we need it, without forcing it on the rest of the software code?

0 0 Reply • Share ›

Paul Burlumi
a month ago

Also see https://journal.stuffwithst...

0 0 Reply • Share ›

Marijan Smetko → Paul Burlumi
a month ago

Yes, that blog was on the back of my mind this entire time

1 0 Reply • Share ›

pascal martin
a month ago

The beauty of using processes and shared memory is that variables are private by default, while the multithread model shares by default, but does not protect. With shared memory, one chooses what is shared and can plan how to protect it. There is much less surprises, especially if you hide these shared variables through a protective access class layer.

Contact:
matt <at> bitbashing.io

mrkline

CompareAndSwap

Yet another programming blog. Thoughts on software and related misadventures.