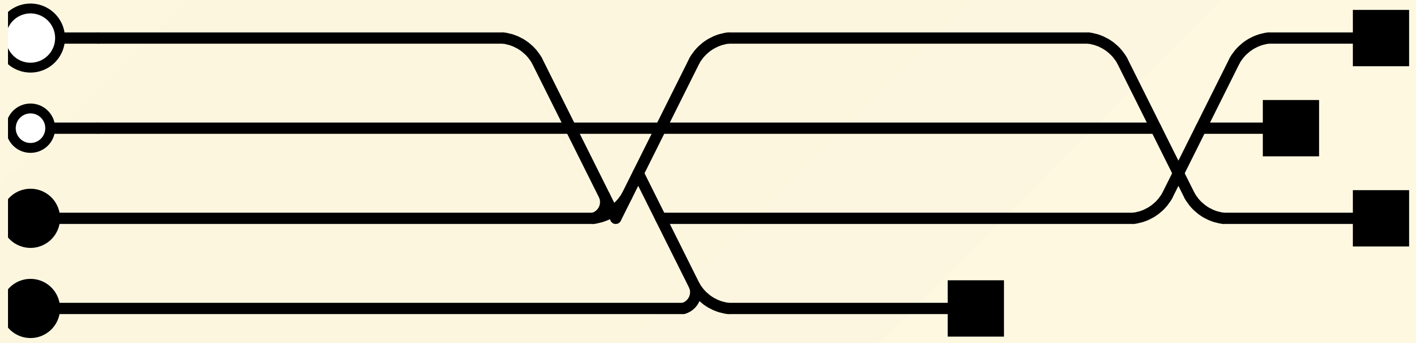
Concurrency and parallelism in Rust

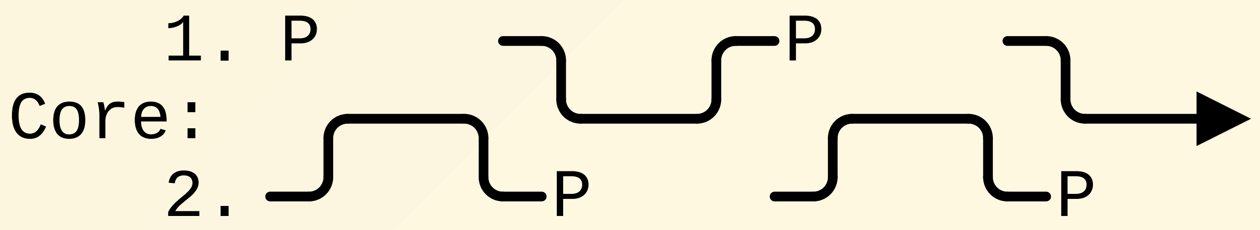


Why concurrency?

- Sometimes things need to happen at the same time
- Computer inputs and outputs cannot always wait
- Sometimes you wait for external resources to finish, e.g. I/O
- Performance

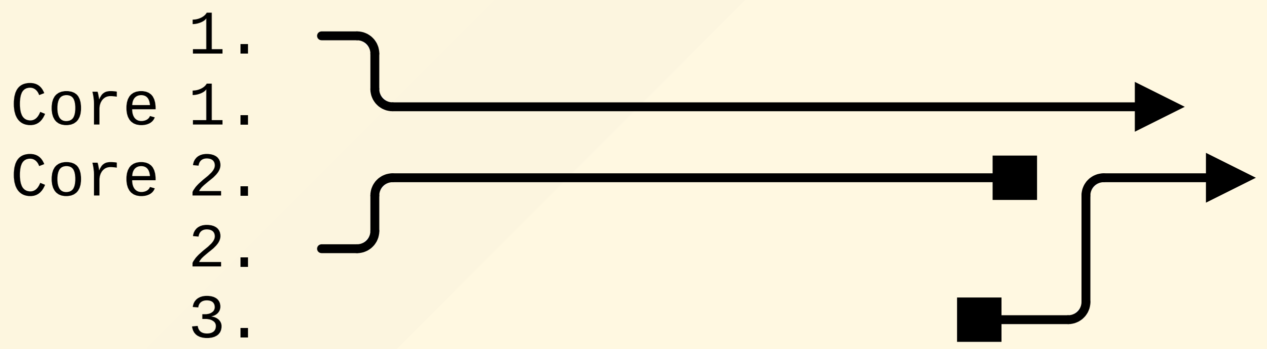
CPUs and concurrency

- CPUs come in many flavours:
  - Single-core CPUs permits a single active thread of execution, i.e. *one thing at a time*
  - Permits concurrency through chunking execution into smaller parts and running chunks of different threads
- Running two threads and switching often feels like parallelism, but in reality only *one* threads run at any point



Multi-cores and parallelism

- Having more than one CPU-core allows executing more than one thread of instructions at any time
- *True* parallelism
- Different naming between OS's (naming is hard), but generally called *processes*, which can spawn *threads*, which themselves can be run on different CPU cores, dependant on OS implementation



Parallelism outside the CPU

- Computers rarely work alone
- Multi-CPU setups
- E.g. two CPUs with single-cores each, or many CPUs with many cores each
- Supercomputers, networks, distributed systems, clusters, multiple servers, etc.

Dangers of concurrency

Data hazards

- Non-*atomic* actions on shared data are unsafe to perform in a concurrent context
- Read-after-write (RAW): **A** tries to read from store before **B** finishes writing to it
- Write-after-read (WAR): **A** reads from store, update data, **B** writes to store, **A** writes, then data from **B** is overwritten
- Write-after-write (WAW): **A** has updated data, writes it to store, but **B** wrote before **A** got the chance

Runtime solutions

- Locking mechanisms: mutexes, semaphores, barriers, condvars, critical sections, yield-locking, spin-locking, etc.
- Pros and cons
  - 👍 Simple, tons of easy-to-access resources
  - 👎 Can lead to deadlocks, livelocks, convoying, resource starvation, indeterminate results, priority inversion, instability, etc.
  - 👎 Runtime overhead, possibly a lot more than a non-negligible amount

Compile-time solutions

- Non-blocking algorithms: Failure or suspension of any thread cannot cause failure or suspension of another thread
  - Usually implemented through atomic CPU instructions such as compare-and-swap and clever algorithm design
  - Can be wait-free, lock-free or obstruction-free, ordered from greatest to weakest guarantees
  - Pros and cons:
    - 👍 No runtime-overhead of lock synchronization
    - 👍 Fewer hazardous pitfalls
    - 👎 Very difficult to implement and may not always be possible

Other solutions

- Formal verification techniques
  - Process calculus, e.g. ⍰-calculus, CSP (communicating sequential processes)
  - TLA+, language for designing, modelling and verifying distributed and concurrent systems
  - Pros and cons:
  - 👍 Formally verifies the absence of bugs such as race conditions
  - 👎 Requires advanced and niche expertise
  - 👎 Often (e.g. TLA+) requires you to model your program, verify it, and keep the model in sync with the implementation

Other compile-time solutions

- Ownership and lifetimes
    - Rust's ownership and lifetime system guarantees safe-to-execute code in concurrent systems
    - Pros and cons:
    - 👍 Successful compilation guarantees data race-freedom
    - 👍 Simpler to program with, alerted of bugs early
    - 👎 Doesn't guarantee race condition freedom (which only formal verification does)

How does Rust do it?

- `T` means you own a variable of type `T`
- `&'a T` means you borrow an immutable reference to `T` with lifetime `'a`
- `&'a mut T` means you borrow an exclusive reference to `T` with lifetime `'a`
- `T: Send` means `T` can be safely sent across thread boundaries. Auto-implemented if available
- `T: Sync` means `T` can be safely synchronized/shared between threads

What are `Send` and `Sync` types?

Examples:

- `bool: Send + Sync`
- `AtomicBool: Send + Sync`
- `&T: !Send + !Sync`
- `Mutex<T>: Send + Sync where T : Send`

Compiler figures it out based on recursively checking contained types for `Send` and `Sync`-bounds or non-bounds.

Threads in Rust

- Spawning a thread:

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

- Can be read as:
  - Closure `f` returns a type `T` and `f` must be sendable across a thread boundary (e.g. cannot reference local unsendable data), must live for `'static` since it's undecidable when thread dies.
  - `T` must be sendable to another thread (since thread returns it)

What about non-static data?

Rust 1.63.0 to the rescue:

```
fn scope<'env, F, T>(f: F) -> T
where
    F: for<'scope> FnOnce(&'scope Scope<'scope, 'env>) -> T,
```

with `Scope`-definition:

```
fn spawn<F, T>(&'scope self, f: F) -> ScopedJoinHandle<'scope, T>
where
    F: FnOnce() -> T + Send + 'scope,
    T: Send + 'scope,
```

Scoped threads usage

```rust
let mut a = vec![1, 2, 3]; let mut x = 0;
std::thread::scope(|s| {
    s.spawn(|| {
        println!("hello from the first scoped thread");
        // We can borrow `a` here.
        dbg!(&a);
    });
    s.spawn(|| {
        println!("hello from the second scoped thread");
        // We can even mutably borrow `x` here, because no other threads are using it.
        x += a[0] + a[2];
    });
    println!("hello from the main thread");
});
// After the scope, we can modify and access our variables again:
a.push(4);
assert_eq!(x, a.len());
```

Rust doesn't always save you!

You can still mess up, e.g. by deadlocking, etc.
However, Rust markets itself with *fearless concurrency*:

https://doc.rust-lang.org/book/ch16-00-concurrency.html

"Simple" atomics

- If you just need to perform simple integer (or other primitives) synchronization across threads, don't use `Mutex<i32>`, but atomic primitives:
  - `AtomicBool`
  - `AtomicI8`, `AtomicI16`, `AtomicI32`, `AtomicI64`, `AtomicIsize`
  - `AtomicPtr`
  - `AtomicU8`, `AtomicU16`, `AtomicU32`, `AtomicU64`, `AtomicUsize`
- Lock-free implementations, as they use CPU-specific features

Using atomic primitives

Useful for e.g. creating a global counter across threads:

```
let atomic_counter = AtomicUsize::new(42);
assert_eq!(atomic_counter.load(Ordering::Relaxed), 42);
atomic_counter.store(10, Ordering::Relaxed);
assert_eq!(atomic_counter.load(Ordering::Relaxed), 10);
```

Good resource on concurrency, atomics and explanation of *memory ordering*: https://marabos.nl/atomics/atomics.html