Concurrency in Rust

Jonathan Aldrich

17-363

Concurrency is tricky!

- Race condition: result of program depends on timing of thread execution
- Deadlocks: two threads are stuck waiting for each other
- Memory leaks: hard to consistently clean up memory shared between threads

- Rust’s type system prevents many of these errors

Shared Memory Concurrency

```
let v = 123;                    // data to be shared (could be of any type)
let m = Mutex::new(v);          // v is protected by m
let mw = Arc::new(m);           // m is tracked by mw
for _ in 0..2 {                 // execute twice (for unused values 0 and 1)
  let mwc = Arc::clone(&mw);
  thread::spawn(move || {
    let mut p = mwc.lock().unwrap();            // start critical section
    *p += 1;
  });
}
thread::sleep(Duration::from_millis(100));      // 1/10th second pause
let p = mw.lock().unwrap();                     // start critical section
println!("{}", *p);                             // probably print 125
```

Notes

Must use reference counting because can’t guarantee the original variable will outlive references to it.

Must use atomic reference counting (Arc) because it is shared between threads. Rc would not work as it doesn’t implement the Send trait.

Unwrapping lock can fail if lock was held by a thread that was killed.

Lock is released when p goes out of scope.

Exercise: What would go wrong here?

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });
    handle.join().unwrap();
}
```

What would go wrong here?

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });
    handle.join().unwrap();
}
```

Answer

The compiler can't tell how long the thread will run. It might run past the end of main()! Since v will be dropped at the end of main, we can't borrow it in the closure.

Making the closure into a move closure will fix the problem.

Message Passing Concurrency

```
let (tx, rx) = mpsc::channel();            // new channel
        // tx is the sending end; rx is the receiving end
for t in 0..2 {                // execute twice (for t = 0 and t = 1)
  let txc = tx.clone();
  thread::spawn(move || {
    txc.send(123 + t).unwrap();
  });
}
let v1 = rx.recv().unwrap();
let v2 = rx.recv().unwrap();
println!("{} {}", v1, v2);          // prints 123 124 or 124 123
```

Notes

Sending end can be cloned, receiving end can't.

Unwrapping on send() will fail if the receiver has been dropped.

Likewise, unwrapping on recv() will fail if the sender has been dropped.

Sending a value transfers ownership. Values implementing the Copy trait are implicitly copied. All values sent must have the same type.

Can use try_recv() to get a value only if one is available, without blocking. Returns error if not available.

An Async Page Scraper

```
async fn page_title(url: &str) -> (&str, Option<String>) {
    let text = trpl::get(url).await.text().await;
    let title = Html::parse(&text)
        .select_first("title")
        .map(|title| title.inner_html());
    (url, title)
}
```

Notes

An async function returns a *promise*; the caller can await it to get a result.

Asyncs execute lazily; nothing happens until the returned promise is await-ed.

Execution proceeds until the first await. It will suspend (and some other async function can be run) until a value is available, then it will continue.

Calling the Page Scraper

```
fn main() {
    let args: Vec<String> = std::env::args().collect();
    trpl::run(async {
        let title_fut_1 = page_title(&args[1]);
        let title_fut_2 = page_title(&args[2]);
        let (url, maybe_title) = match trpl::race(title_fut_1, title_fut_2).await {
            Either::Left(left) => left,
            Either::Right(right) => right,
        };
        println!("{url} returned first");
    })
}
```

Notes

You can't just call an async function from main, you have to use a runtime that executes async blocks. Here we use the Tokio runtime (packaged up in trpl for the rust book)

We call 2 async functions and use race to await both. They will start executing and the result of the one that finishes first will be returned.

Waiting for results from 2 tasks

```rust
let fut1 = async {
    for i in 1..10 {
        println!("hi number {i} from the first task!");
        trpl::sleep(Duration::from_millis(500)).await;
    }
};
let fut2 = async {
    for i in 1..5 {
        println!("hi number {i} from the second task!");
        trpl::sleep(Duration::from_millis(500)).await;
    }
};
trpl::join(fut1, fut2).await;
```

Async and channels

```
let (tx, mut rx) = trpl::channel();
let tx_fut = async move {
    let vals = vec![ String::from("hi"), String::from("from"),
                     String::from("the"), String::from("future"), ];
    for val in vals {
        tx.send(val).unwrap();
        trpl::sleep(Duration::from_millis(500)).await;
    }
};
let rx_fut = async {
    while let Some(value) = rx.recv().await {
        eprintln!("received '{value}'");
    }
};
trpl::join(tx_fut, rx_fut).await;
```

Notes

Unlike with the channels we saw before, we must await when receiving a message here.

Interleaving long-running operations

```rust
let a = async {
    println!("'a' started.");
    slow("a", 30);
    trpl::yield_now().await;
    slow("a", 10);
    trpl::yield_now().await;
    slow("a", 20);
    trpl::yield_now().await;
    println!("'a' finished.");
};
```

```rust
let b = async {
    println!("'b' started.");
    slow("b", 75);
    trpl::yield_now().await;
    slow("b", 10);
    trpl::yield_now().await;
    slow("b", 15);
    trpl::yield_now().await;
    slow("b", 35);
    trpl::yield_now().await;
    println!("'b' finished.");
};
```

Async vs. Threads

- Generally Threads are a cleaner abstraction
  - No yield; no special “async” functions; no explicit await
  - Avoids some other pain I didn’t show you, e.g. pinning references to allow joining dynamically-determined numbers of tasks
  - Use threads unless you are sure you need async
- But Async can be higher performance
  - Rust’s threads are OS threads. They’re expensive to create and there’s a limited number of them.
  - Using async allows fast creation and execution of many more tasks