Data race

Multiple accesses with at least one writer

```
for (int i = 0; i < INCREMENTS; i++) {
 int cur_count = counter;
 int new_count = cur_count + 1;
 counter = new_count;
}
```

Thread 1:	Thread 2:
`cur_count <- 11`	
	`cur_count <- 11`
`new_count <- 12`	
`counter <- 12`	
	`new_count <- 12`
	`counter <- 12`

time ↓

Assumes that in a single time-step we can only do either load, increment, or store

Atomics

Atomics: uninterruptible arithmetic instructions
(implemented by CPU)

```
for (int i = 0; i < INCREMENTS; i++) {
 int cur_count = counter;
 int new_count = cur_count + 1;
 counter = new_count;
}
```

time

Thread 1:	Thread 2:
`cur_count <- 11; new_counter <- 12; counter <- 12`	
	`cur_count <- 12; new_counter <- 13; counter <- 13`

Atomics vs. Mutexes

Why not use atomics always?

```
for (int i = 0; i < INCREMENTS; i++) {
 atomic_add(&counter, 1)
}
```

```
struct MultiThreadVec {
    Mutex mutex;
    std::vector<int> vec;
}

void push(MultiThreadVec v, int n) {
    v.mutex.lock();
    v.vec.push_back(n);
    v.mutex.unlock();
}
```

Resizing a vector is complicated! Can't use an atomic, as the vector potentially needs to be resized via copying

Misc. PLQ Answers

Would it be possible for the compiler to detect when Rc vs. Arc are required (e.g., give atomicity guarantees but optimize in the single-threaded case)?"

- Sort of! In C/C++, if you use mutexes in your code but don't link to the threads library, the mutex functions do nothing. Rust could do something similar, but doesn't

"Does Rust also have an equivalent to forking a process?"

- No! Forking could potentially violate safety (why?) (https://internals.rust-lang.org/t/why-no-fork-in-std-process/13770)
- Also, `fork` doesn't exist on Windows and Rust attempts to be cross-platform

Lecture 9

Message Passing

Mutex poisoning

Mutex locking returns an Option. Why?

```
impl<T> Mutex<T> {
  pub fn lock(&self) -> Option<MutexGuard<'_, T>> {
    // omitted
  }
}
```

If a thread panics while holding a Mutex, the mutex is “poisoned” instead of automatically unlocked.

Locking a poisoned mutex returns None

https://doc.rust-lang.org/src/std/sync/mutex.rs.html#315-320

Mutex poisoning

```
fn pay_salaries(accounts: Mutex<Accounts>) {
 let mut accounts = accounts.lock().unwrap();
 let employees = accounts.employees();

 for account in employees {
   account += 1000;
 }
 accounts.corporate() -= 1000 * employees.len();
}
```

Mutex poisoning

```
fn pay_salaries(accounts: Mutex<Accounts>) {
 let mut accounts = accounts.lock().unwrap();
 let employees = accounts.employees();

 for account in employees {
   account += 1000;
 }
 accounts.corporate() -= 1000 * employees.len();
}
```

When a thread panics while holding a mutex, application-specific invariants may not be upheld.

Mutex poisoning

```
fn pay_salaries(accounts: Mutex<Accounts>) {
 let mut accounts = accounts.lock().unwrap();
 let employees = accounts.employees();

 for account in employees {
   account += 1000;
 }
 accounts.corporate() -= 1000 * employees.len();
}
```

When a thread panics while holding a mutex, application-specific invariants may not be upheld.

Why Send and Sync

C uses `pthread_create` to spawn a new thread with the exact same stack, but this is immediately unsafe! We have two mutable references to every value.

Instead, we want the new thread to have access to nothing, and then only allow sharing values that we know are multithreading safe.

Send: it's ok to *move* this value to another thread (the original thread can no longer access it)

Sync: it's ok to *share* this value between threads (both can access it)

More practice with Send + Sync

https://stackoverflow.com/questions/59428096/understanding-the-send-trait

Case Study: spawning threads

See `spawn` directory in lecture code

Collatz conjecture

Consider the following operation on an arbitrary positive integer:

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

Does this process reach 1 for every starting number?

A preview of next week

So far, we've seen some Rust guarantees that hold about **all programs** at **all moments** during execution

- References are never null
- References point to alive values
- A value has at most one mutable reference pointing at it

- Values won't be dropped multiple times
- Values can't be accessed after being moved

- Non-thread-safe values can't be sent between threads

These guarantees have nice results:

- Rust programs never segfault
- Rust programs never have data races
- Rust programs never exhibit undefined behavior

For no performance penalty!

A preview of next week

So far, we've seen some Rust guarantees that hold about **all programs** at **all moments** during execution

Can't prove these properties to the compiler?
Use dynamic checks

- Rc
- RefCell
- Arc
- Mutex

but there's a performance cost

A preview of next week

So far, we've seen some Rust guarantees that hold about **all programs** at **all moments** during execution

Sometimes the program is valid, but

- we can't prove it to the compiler
- we don't want a performance penalty

For example, implementing `Vec`

What to do?

Unsafe Rust

Temporarily disable some of Rust's safety checks

- e.g. allows using raw pointers

Use `unsafe` to build safe abstractions on top of unsafe code.

- `Vec` and `String` have unsafe code inside, but all public functions are safe to call.

raw pointer, not a reference!

```
let address = 0x012345;
let ptr = address as *const i32;
unsafe {
   println!("Value at address: {}", *ptr);
}
```

Unsafe example: building safe abstractions

Get mutable access to separate halves of vec

- Impossible to do in safe Rust; compiler can't verify halves don't overlap

```
pub const fn split_at_mut(v: Vec<T>, mid: usize)
 -> Option<(&mut [T], &mut [T])>
{
 if mid <= v.len() {
   let len = v.len();
   let ptr = v.ptr();

   unsafe {
     (
       Slice::from_raw(ptr, mid),
       Slice::from_raw(ptr.add(mid), len - mid),
     )
   }
 } else {
   None
 }
}
```

Unsafe example: making Mutex Send/Sync

```
pub struct Mutex<T> {
    inner: sys::Mutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

Internal types of Mutex are not necessarily safe to Send and Sync

```
unsafe impl<T: Send> Send for Mutex<T> {}
unsafe impl<T: Send> Sync for Mutex<T> {}
```

Since we are confident the mutex locking logic makes it safe to Send and Sync a `Mutex<T>` regardless of what T is, we can declare the trait implementations.

But! It's unsafe to impl these traits: if we impl Sync for a type that isn't safe to share, then Rust's guarantees no longer hold

Compare to `Copy`: always safe to impl even though poor judgement will cause bad performance

Unsafe Rust

If we can turn off safety checks, how is this better than C/C++?

- If a segfault occurs, only have to look at unsafe blocks instead of whole program
- Unsafe code in standard library and popular packages is audited to ensure correctness

Parallelism (again)

But with channels this time

Mutexes are hard, what else can we do?

Do not communicate by sharing memory; instead, share memory by communicating.

- Effective Go

Mutexes are hard, what else can we do?

Do not communicate by sharing memory; instead, share memory by communicating.

- Effective Go

View 1:

Programs are a set of threads running in parallel that operate on one shared heap

View 2:

Programs are a set of threads running in parallel operating on disjoint heaps and sharing data via inter-thread *channels*

Higher-level concurrency: channels

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // (transmit, receive)
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(10).unwrap();
    });
    println!("Got: {}", rx.recv().unwrap());
}
```

Two ends:

- Multiple producers
- Single consumer

Higher-level concurrency: channels

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // (transmit, receive)
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(10).unwrap();
    });
    println!("Got: {}", rx.recv().unwrap());
}
```

`send`ing a value is instantaneous

`recv`ing a value waits until a value is in the channel.

Higher-level concurrency: channels

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // (transmit, receive)
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(10).unwrap();
    });
    println!("Got: {}", rx.recv().unwrap());
}
```

`send` and `recv` return an `Option`

When can sending or receiving go wrong?

But what *is* a channel?

```rust
use std::collections::VecDeque;
use std::sync::Mutex;


pub struct Channel<T> {
    data: Mutex<VecDeque<T>>
}
```

```rust
impl<T> Channel<T> {
  pub fn new() -> Channel<T> { ... }


  pub fn send(&self, value: T) {
    self.data.lock().unwrap().push_back(value);
  }

  pub fn recv(&self) -> Option<T> {
    self.data.lock().unwrap().data.pop_front()
  }
}
```

But what *is* a channel?

```rust
use std::collections::VecDeque;
use std::sync::Mutex;


pub struct Channel<T> {
    data: Mutex<VecDeque<T>>
}
```

```rust
impl<T> Channel<T> {
  pub fn new() -> Channel<T> { ... }


  pub fn send(&self, value: T) {
    self.data.lock().unwrap().push_back(value);
  }

  pub fn recv(&self) -> Option<T> {
    self.data.lock().unwrap().data.pop_front()
  }
}
```

This doesn't match the interface we want

Ok, but really

```rust
use std::collections::VecDeque;
use std::sync::{Arc, Condvar, Mutex};

pub struct Channel<T> {
 data: Mutex<VecDeque<T>>,
 cv: Condvar,
}
```

Condition variable: allows a thread to sleep until a condition is met

Example: sleep until queue is non-empty

```rust
impl<T> Channel<T> {
  pub fn new() -> Channel<T> { ... }

  pub fn send(&self, value: T) {
    let mut data = self.data.lock().unwrap();
    data.push_back(value);
    self.cv.notify_one();
  }

  pub fn recv(&self) -> T {
    let mut data = self.data.lock().unwrap();
    while data.is_empty() {
      data = self.cv.wait(data).unwrap();
    }
    data.pop_front().unwrap()
  }
}
```

Wake one thread that is sleeping

Sleep til condition is met

Going further

If the queue is long enough, two threads should be able to send and receive without waiting for the mutex.

One mutex for each thread item?

In general: implementing channels is a challenging concurrency problem. See crossbeam for a good implementation.

Quiz

Suppose you have a multi-threaded web server with each thread processing requests, and they need to occasionally log events to a global, combined log.

How would you implement with channels?

With shared-state (mutex)?

What are the pros and cons?

```
fn worker_thread(args: ???) {
    loop {
        // do some work
        let event = generate_event();
        // log event somehow?
        ...
    }
}
```

Quiz

Shared state

```
static logs: Mutex<Vec<String>> =
  Mutex::new(Vec::new());

fn worker_thread() {
   loop {
       // do some work
       let event = generate_event();
       logs.lock().push(event);
   }
}
```

Quiz

Shared state

```
static logs: Mutex<Vec<String>> =
  Mutex::new(Vec::new());

fn worker_thread() {
    loop {
        // do some work
        let event = generate_event();
        logs.lock().push(event);
    }
}
```

Channels


```
fn worker_thread(logger: Sender<String>) {
    loop {
        // do some work
        let event = generate_event();
        logger.send(event);
    }
}
fn logger_thread(workers: Vec<Receiver<String>>) {
    let logs = Vec::new();
    loop {
        let event = recv_from_any_worker(workers);
        logs.push(event);
    }
}
```

Quiz

Shared-state

Pro:

- ??

Con:

- ??

Channels

Pro:

- ??

Con:

- ??

Quiz

Shared-state

Pro:

- Logger thread can’t become overwhelmed

Con:

- Worker threads waste time waiting for lock to be released

Channels

Pro:

- Worker threads can send the log instantly and get back to work

Con:

- Logger thread can get overwhelmed

Takeaways

Channels are a nice abstraction, but generally have higher overheads than using a mutex.

If your problem involves communication, use someone else's channel implementation instead of making your own!

If the performance isn't high enough, think about how you can use a mutex instead.

Another channel example

```
fn main() {
  let (tx, rx) = mpsc::channel();
  let rx = Arc::new(Mutex::new(rx));
  for i in 1..100 {
    tx.send(i).unwrap();
  }

  for _ in 0..10 {
    let rx = Arc::clone(&rx);
    std::thread::spawn(move || loop {
      let n: u64 = rx.lock().unwrap().recv().unwrap();
      let result = collatz(n);
      println!("Collatz({n}) = {result}");
    });
  }
}
```

How to turn a single-consumer channel into a multi-consumer channel?

“Thread Pool”: handful of threads collectively completing list of tasks

Concurrency or parallelism?

Website

10/15	Parallel	Shared Memory [code]	16
10/22	Parallel	Message Passing	16

Book

16. Fearless Concurrency

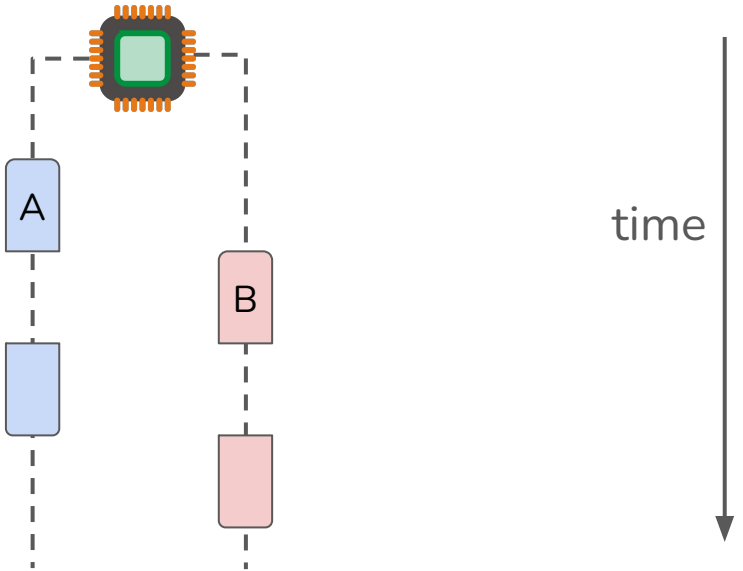
- **16.1.** Using Threads to Run Code Simultaneously
- **16.2.** Using Message Passing to Transfer Data Between Threads
- **16.3.** Shared-State Concurrency
- **16.4.** Extensible Concurrency with the Sync and Send Traits

What's the difference between concurrency and parallelism? Is there one?

Concurrency or parallelism?

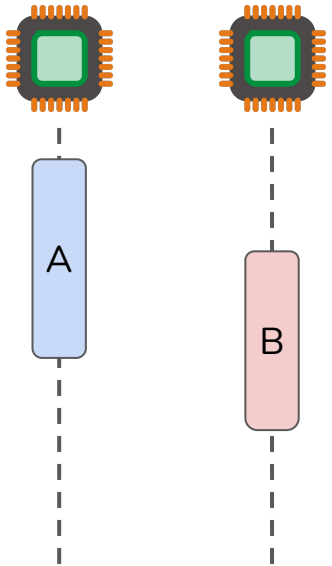
Concurrency

- View 1: tasks are interruptible
- View 2: multiple tasks can make progress



Parallelism

- Subset of concurrency
- Multiple tasks are executed *at the same time*

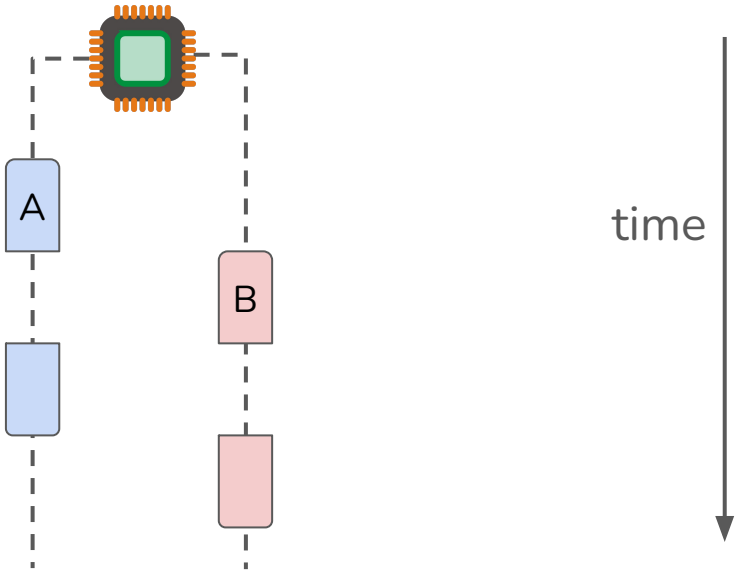


Concurrency or parallelism?

Also usually exhibits interleaving, since a single CPU thread runs many OS threads

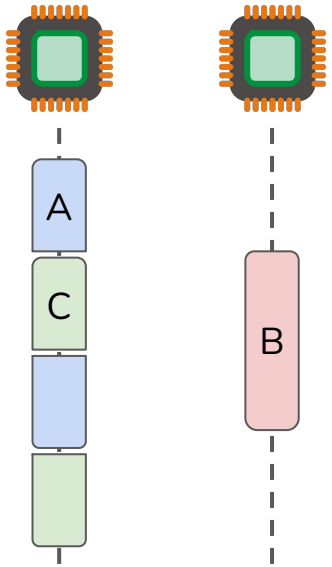
Concurrency

- View 1: tasks are interruptible
- View 2: multiple tasks can make progress



Parallelism

- Subset of concurrency
- Multiple tasks are executed *at the same time*

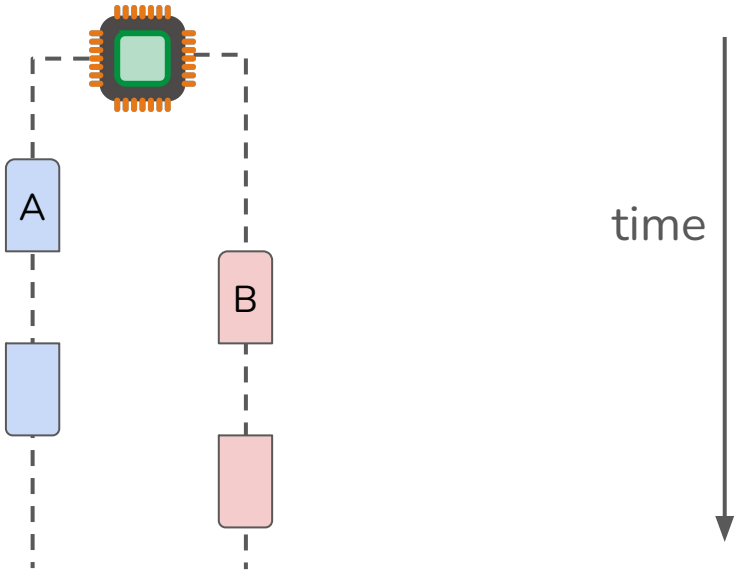


Concurrency or parallelism?

Harder than single-threaded
Faster than single-threaded

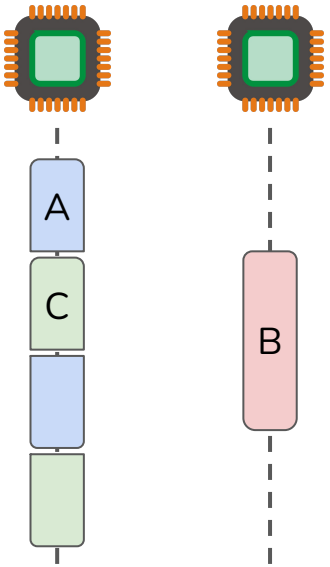
Concurrency

- View 1: tasks are interruptible
- View 2: multiple tasks can make progress



Parallelism

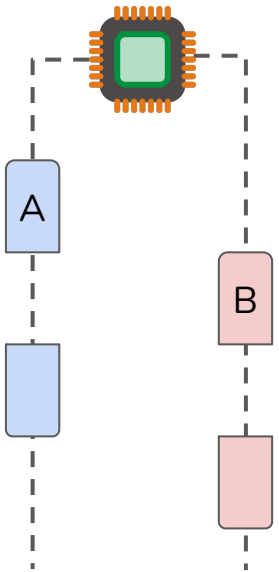
- Subset of concurrency
- Multiple tasks are executed *at the same time*



Concurrency or parallelism?

Why care about concurrency?

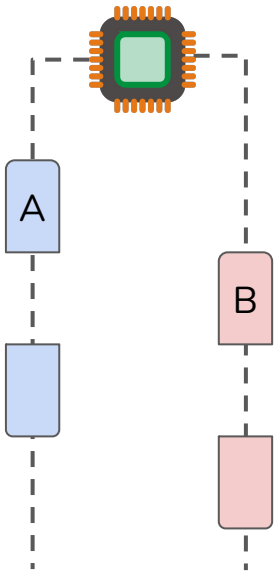
- Is it faster?
- Is it hard?



Concurrency or parallelism?

Why care about concurrency?

- Is it faster?
- Is it hard?



Is it faster? (than sequential)

Yes! Some operations require waiting on someone else. Do something else while you wait.

- Send request to a server -> wait on network
- Read from a file -> wait on OS

Is it hard?

Not as hard as parallelism. No data races, but still need to worry about tasks getting interrupted

- What if your task gets interrupted after popping a `Vec` element but before updating the length?

Concurrency example

Sequential

```
fn main() {
    let servers = vec![...];
    for server in servers {
        let request = make_request(server);
        request.wait_for_response();
    }
}
```

Concurrent (not parallel)

```
fn main() {
    let servers = vec![...];
    let mut requests = vec![];
    for server in servers {
        requests.push(make_request(server));
    }
    for request in requests {
        request.wait_for_response();
    }
}
```

Concurrency or parallelism?

	Non-interleaving	**Interleaving**
Non-simultaneous	Fully sequential	Single-threaded concurrency
Simultaneous		Multi-threading

concurrent

concurrent & parallel

Concurrency or parallelism?

First part of course

previous slide & async Rust

	Non-interleaving	**Interleaving**
Non-simultaneous	Fully sequential	Single-threaded concurrency
Simultaneous		Multi-threading

concurrent

concurrent & parallel

Last two lectures

Concurrency or parallelism?

First part of course

previous slide & async Rust

	Non-interleaving	**Interleaving**
Non-simultaneous	Fully sequential	Single-threaded concurrency
Simultaneous	Multi-threading but simpler	Multi-threading

concurrent

concurrent & parallel

Last two lectures

Want to learn more?

Concurrency is Not Parallelism

https://go.dev/blog/waza-talk