Lecture 12

Mutual Exclusion

Mutexes

- Mutexes can lock data so that we can access it safely from multiple threads without data risks.

```
use std::sync::{Mutex,Arc};
use std::thread::spawn;
fn main(){
        let m = Arc::new(Mutex::new(0));
        let mut handles = Vec::with_capacity(10);
        for i in 0..10 {
                let am = m.clone();
                let j = i;
                handles.push(spawn(move||{
                        let mut p = am.lock().unwrap();
                        *p +=j; //Magic! Ignore
                        println!("j={}, p={}",j,*p);
                }));}
        for h in handles{
                h.join().unwrap();
        }
        println!("Done");
 }
```

Mutex can be implemented as anything, here we implement it as integer.

pub fn new(t: T) -> Mutex<T>

Mutexes

```
Output:
j=7, p=7
j=8, p=15
j=9, p=24
j=6, p=30
j=5, p=35
j=4, p=39
j=3, p=42
j=2, p=44
j=1, p=45
j=0, p=45
Done
```

Mutexes

- ***Arc*** is an atomic reference count pointer that stores immutable objects, but you can have one reference to the same object safely.
- The mutex pretends to be immutable, but it uses unsafe code inside it to allow you to make the changes to its contents in a way that is safe on the outside.
- In most cases, you are going to pass that mutex between threads using an atomic reference count.

```
pub fn new(t: T) -> Mutex<T>
```

```
let m = Arc::new(Mutex::new(0));
```

or

```
let message = Arc::new(Mutex::new(String::from("hello")));
```

Mutexes

- Mutexes can lock data so that we can access it safely from multiple threads without data risks.

```
use std::sync::{Mutex,Arc};
use std::thread::spawn;
fn main(){
        let m = Arc::new(Mutex::new(0));
        let mut handles = Vec::with_capacity(10);
        for i in 0..10 {
                let am = m.clone();
                let j = i;
                handles.push(spawn(move||{
                        let mut p = am.lock().unwrap();
                        *p +=j; //Magic! Ignore
                        println!("j={}, p={}",j,*p);
                }));}
        for h in handles{
                h.join().unwrap();
        }
        println!("Done");
 }
```

Mutex can be implemented as anything, here we implement it as integer.

pub fn new(t: T) -> Mutex<T>

Mutex Poisoning

- Mutexes implement a strategy called "poisoning" where a mutex is considered poisoned whenever a thread panics while holding the mutex.
- Once a mutex is poisoned, all other threads are unable to access the data by default as it is likely tainted (Wrong).
- For a mutex, this means that the lock method returns a Result which indicates whether a mutex has been poisoned or not.

Low-Level Primitives

Low-Level Primitives

- Atomics are low-level synchronization primitives that allow us to have causality by restricting the order of operations.
- These primitives need to be processor-level since in addition to restricting the compiler we want to restrict the processor from the cache-level reordering.
- Atomics give two guarantees:

1. we can perform read/write operations on them without fear of broken reads or writes;
2. atomic operations provide guarantees about the order in which they execute, relative to each other, even across the threads.
3. atomics even enforce the order on non-atomic operations.

Low-Level Primitives

- Each operation on an atomic variable is required to have an ordering type:

1. Ordering::Relaxed
2. Ordering::Acquire and Ordering::Release (or their joint alternative Ordering::AcqRel)
3. Ordering::SeqCst — short for sequential consistency

- You will almost always use **SeqCst** which applies the strongest constraints and is the easiest to reason about.
- **Relaxed** applies the weakest constraints and is extremely non-intuitive, so unless you are developing low-level high-performance code you should stay away from it.

Low-Level Primitives

- **Acquire/Release** is a middle ground, in terms of cognitive complexity, but you will still almost never prefer it over SeqCst.
- However, understanding Acquire/Release is very helpful for the understanding of high-level synchronization primitives.

Acquire/Release

- Acquire/Release are hardware-level operations because they generate special instructions for the hardware.
- One can use Acquire/Release in the following way:

```
let x = AtomicUsize::new(0);
let mut result = x.load(Ordering::Acquire);
result += 1;
x.store(result, Ordering::Release); // The value is now 1.
```

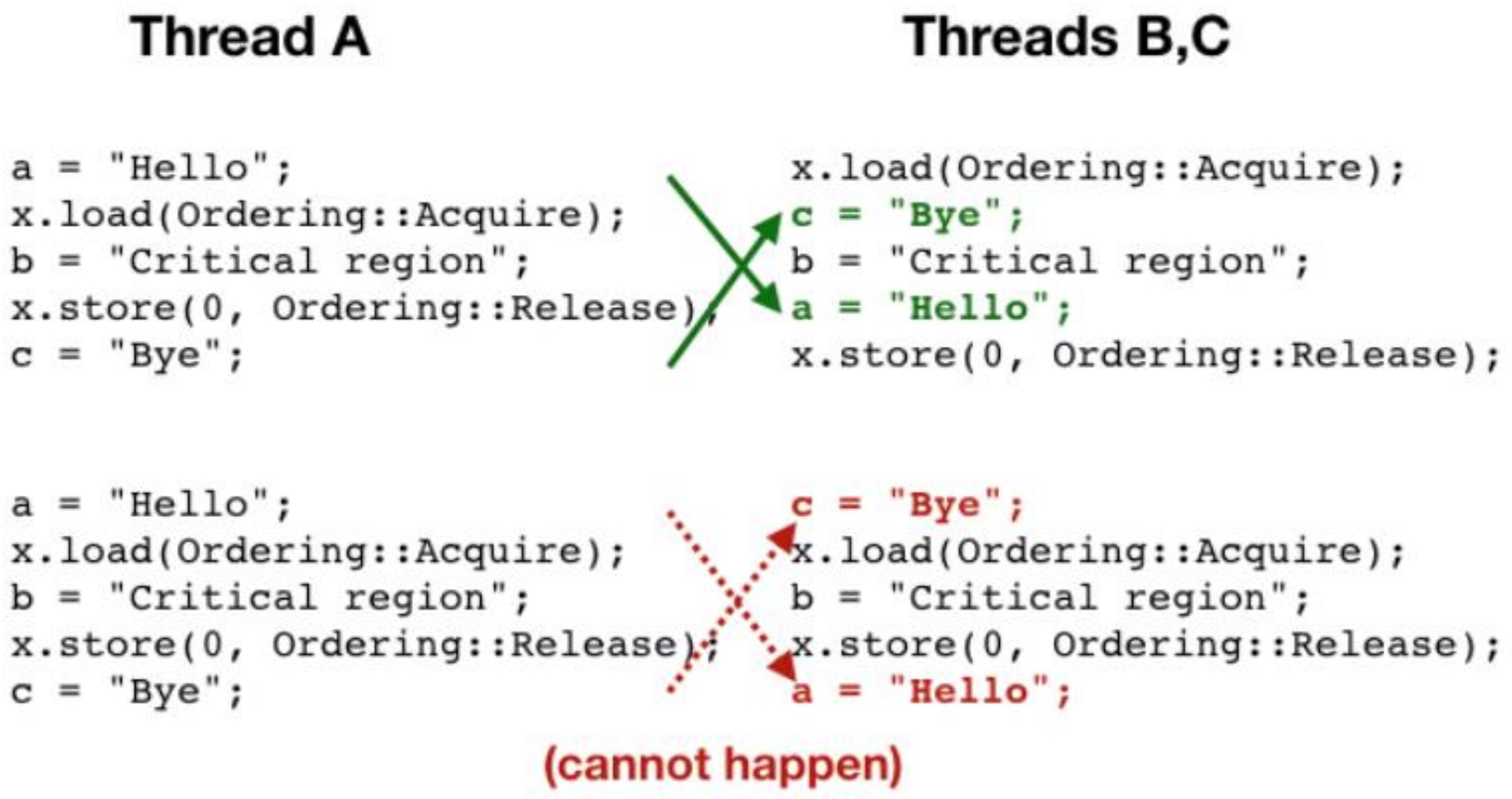
```
pub fn load(&self, order: Ordering) -> usize

Loads a value from the atomic integer.

pub fn store(&self, val: usize, order: Ordering)

Stores a value into the atomic integer.

load (and store) takes an Ordering argument which describes
the memory ordering of this operation.
```

Acquire/Release

- Acquire can only be used with load operations.
- Release can only be used with store operations.
- Acquire and Release have the following rules:

1. **Acquire** — all memory accesses that happen **after** it in the code stay after it as visible by all threads.
2. **Release** — all memory accesses that happen **before** it in the code stay before it as visible by all threads.

Example



Thread A

Threads B,C

```
a = "Hello";                              x.load(Ordering::Acquire);
x.load(Ordering::Acquire);                c = "Bye";
b = "Critical region";                    b = "Critical region";
x.store(0, Ordering::Release);            a = "Hello";
c = "Bye";                                x.store(0, Ordering::Release);
```

```
a = "Hello";                              c = "Bye";
x.load(Ordering::Acquire);                x.load(Ordering::Acquire);
b = "Critical region";                    b = "Critical region";
x.store(0, Ordering::Release);            x.store(0, Ordering::Release);
c = "Bye";                                a = "Hello";
```

(cannot happen)

Sequential consistency

- More formally, SeqCst follows the rule:

  All atomic operations that happen before/after SeqCst operation stay before/after it on all threads. Ordinary non-atomic reads and writes may move down across an atomic read, or up across an atomic write.
- In Rust, **SeqCst** involves emitting a memory barrier that prevents undesirable reordering.
- Unfortunately, SeqCst is more expensive than pure Acquire/Release,
- However, it is still negligible in the global picture.
- Therefore it is highly recommended to use SeqCst whenever possible.

Very Low-Level Primitives

What are threads?

1. hardware threads, a.k.a hyperthreading;
2. OS threads;
3. green threads

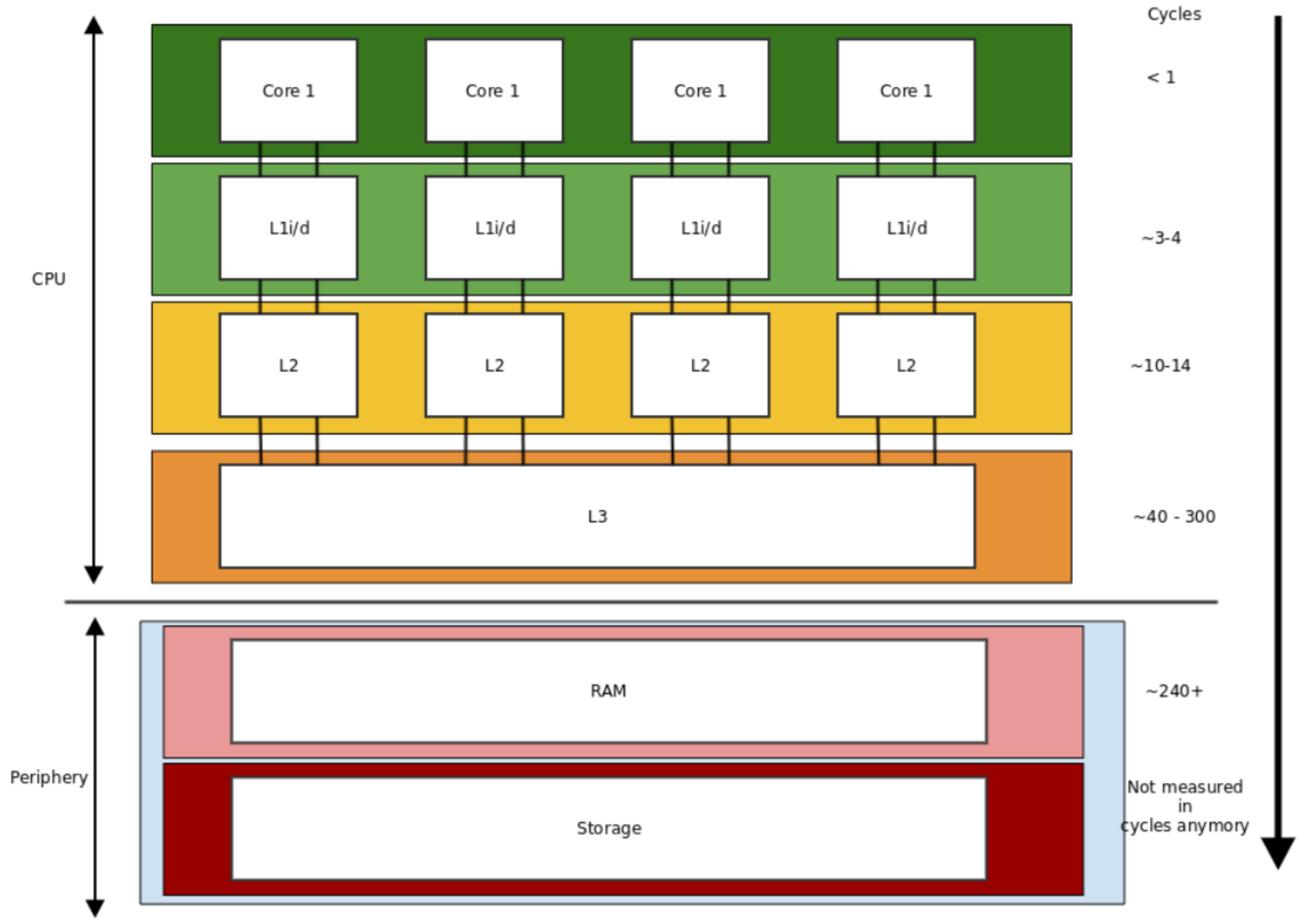
Hyperthreading & OS Threads

- Hyperthreading is when the processor virtually splits each physical core into two virtual cores.
- OS threads are created and managed internally by the OS.
- Most OS make the number of threads practically unlimited,
- Starting them is expensive since it requires allocation of a stack.

Green Threads

- Green threads are implemented by the user software, and they run on top of OS threads.
- The advantages of the green threads are:
  - they work even in the environments that do not have OS thread support.
  - they are much faster to spin-up than regular threads.

CPU - MESI


Cycles
Core 1
Core 1
Core 1
Core 1
< 1
L1i/d
L1i/d
L1i/d
L1i/d
~3-4
CPU
L2
L2
L2
L2
~10-14
L3
~40 - 300
RAM
~240+
Periphery
Storage
Not measured in cycles anymory

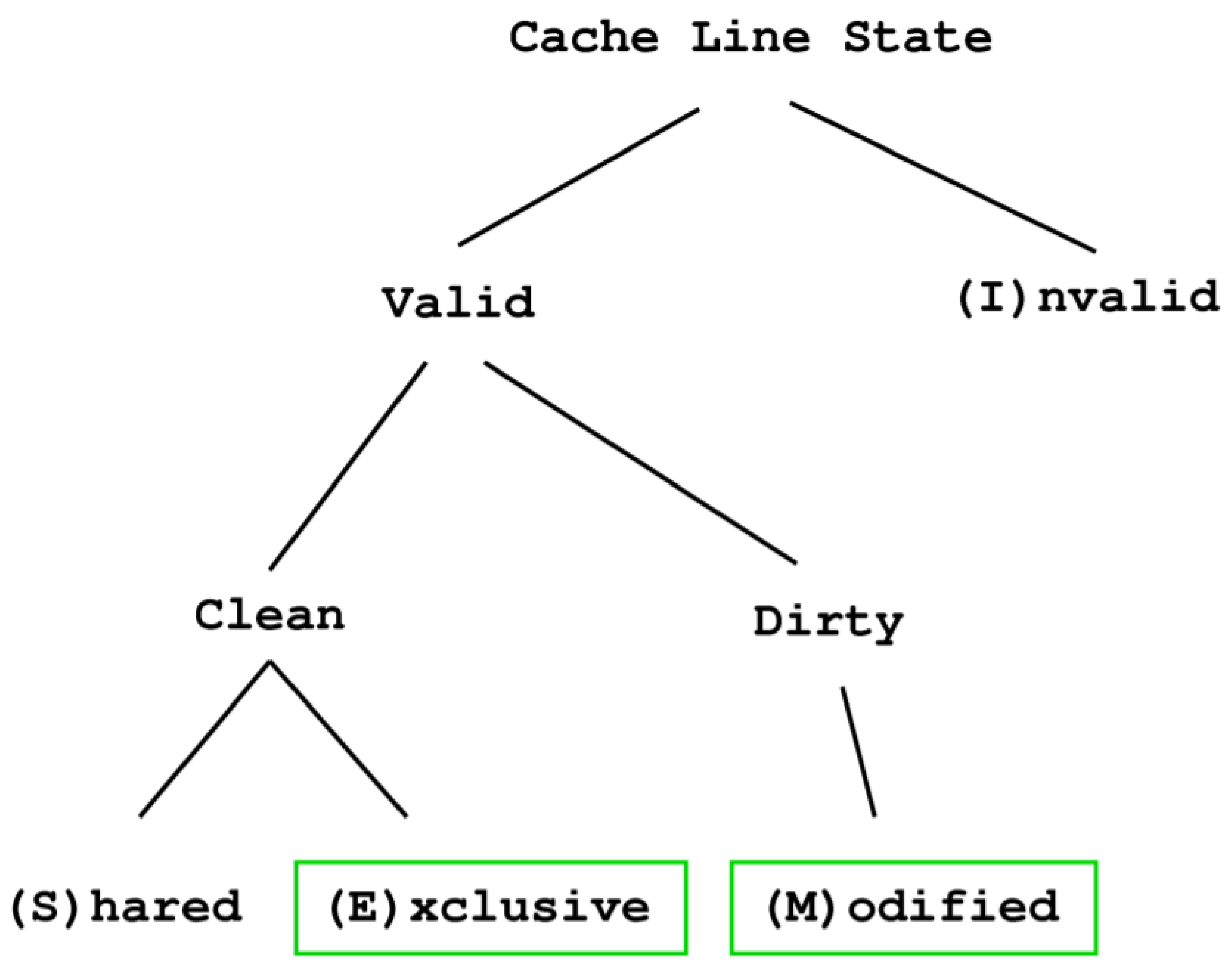
CPU - MESI

- The usual task of an CPU is as follows
  - fetch information
  - process information
  - store information
- This is what every CPU does, it's in the name, central processing unit.
- But how do the cores work? and how do the registers and cache work? and why is this even relevant?
- If there is only 1 core in the CPU it's not. As soon as we have multiple cores that should work on the same problems the cooperation problem starts.

CPU - MESI

- How do we coordinate the flow of data between different cores? We know we want it to be fast! So we try to fill the lower caches with the information needed if possible.
- But what if Processor 1 does the first part and processor two does the second part? if we look at the diagram we see that the L1 and L2 caches are not shared.
- How do can we efficiently communicate between our CPUs to ease some of the problems any distributed system has.

MESI to the rescue.



MESI to the rescue.

- And If you think about Rust again and how ownership works, this get's really close on an abstract idea of computation
- you need to own it to write (**exclusive**)
- you can share for read (**shared**)
- the **modified** and **invalid** are prohibited in the model since you have to ARC it.

```
let m = Arc::new(Mutex::new(0));
```

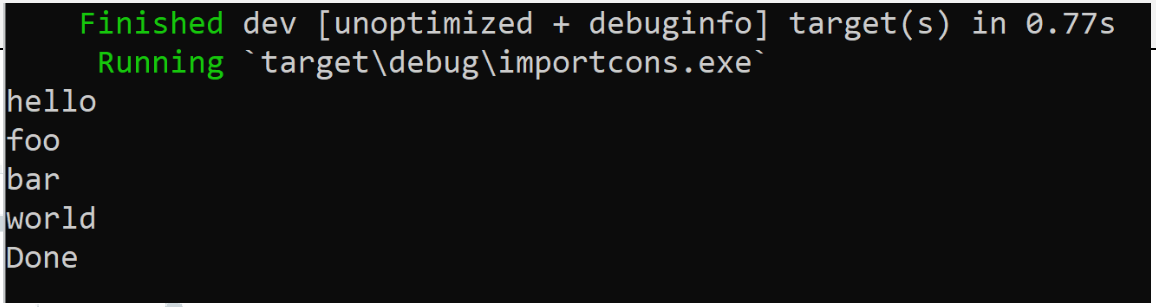
```
let message = Arc::new(Mutex::new(String::from("hello")));
```

High-level Packages

Thread Pools and Workers

- With a worker pool, you can send multiple jobs onto this stack, and they will be implemented when a thread is available to run them.

```
use threadpool::ThreadPool;

fn main(){
  let pool = ThreadPool::new(2);
 pool.execute(|| println!("hello"));
 pool.execute(|| println!("world"));
 pool.execute(|| println!("foo"));
 pool.execute(|| println!("bar"));
 pool.join(); //Block the current thread (main) until all jobs
        in the pool have been executed.

 println!("Done ");
}
```



```
    Finished dev [unoptimized + debuginfo] target(s) in 0.77s
     Running `target\debug\importcons.exe`
hello
foo
bar
world
Done
```

Rayon for Parallel Problems

- Rayon is a crate that provides useful methods for handling concurrency.
- Let's assume we have a big list, and then, we iterate through it; and perform some computationally expensive work.

```
use std::thread;
use std::time::Duration;

fn main(){
        let mut v= Vec::with_capacity(500);
        for i in 0..500{ v.push(i);    }
        let v2:Vec<i32> = (&v).into_iter()

        .map(|&x|{thread::sleep(Duration::from_millis(100));
                println!("{}", x);
                x*x
        }).collect();
        println!("{:?}", &v2[490..500] );}
```

Rayon for Parallel Problems

- If we run it, you'll see that it's slowly working through each one, waiting that extra 100 milliseconds to do that work.

```
//Output:
0
1
2
3
4
5
6
…
…
…

499
[240100, 241081, 242064, 243049, 244036, 245025, 246016,
247009, 248004, 249001]
```

Rayon for Parallel Problems

- So, the question is: how can we make that parallel so that we can use multiple processes to make it quicker?
- We will just change *into_iter()* to *par_iter()*

```
use std::thread;
use std::time::Duration;
use rayon::prelude::*;
fn main(){
        let mut v= Vec::with_capacity(500);
        for i in 0..500{ v.push(i);   }
        let v2:Vec<i32> = (&v).par_iter()

        .map(|&x|{thread::sleep(Duration::from_millis(100));
                println!("{}", x);
                x*x
        }).collect();
        println!("{:?}", &v2[490..500] );}
```

Rayon for Parallel Problems

- When we run the program, it will not produce every number in order. It is still correct.
- The program is a lot quicker than the other time.

```
//Output:
0
62
187
250
125
7
…
…
…
08
310
311
309
[240100, 241081, 242064, 243049, 244036, 245025, 246016,
247009, 248004, 249001]
```

Rayon for Parallel Problems

- Everything in Rayon works on the method called *join*.

```
pub fn square_split(v:&mut[i32]){
        if v.len()<4{
          for i in v{
                thread::sleep(Duration::from_millis(10));
                       println!("{}",*i );
                       *i = (*i) * (*i);
          }
          return
        }
        let (mut a, mut b)= v.split_at_mut(v.len()/2);
        square_split(&mut a); //recursion
        square_split(&mut b); //recursion
}
```

Rayon for Parallel Problems

- Then, we can call the method square_split()

```
fn main(){
        let mut v= Vec::with_capacity(500);
        for i in 0..500{
                v.push(i);
        }
        square_split(&mut v);
        println!("{:?}", &v[490..500] );
}
//Output:
0
1
2
3
…
498
499
[240100, 241081, 242064, 243049, 244036, 245025, 246016,
247009, 248004, 249001]
```

Rayon for Parallel Problems

- We can make this go much faster if we use join.
- We'll call a closure which will do square_split on a.
- and then another closure which will do square_split on b.
- And what join will do: it will only run them in parallel.

```
pub fn square_split(v:&mut[i32]){
        if v.len()<4{
          for i in v{
             thread::sleep(Duration::from_millis(10));
             println!("{}",*i );
             *i = (*i) * (*i);
           }
           return
        }
        let (mut a, mut b)= v.split_at_mut(v.len()/2);
        join(||square_split(&mut a), ||square_split(&mut b));
}
```

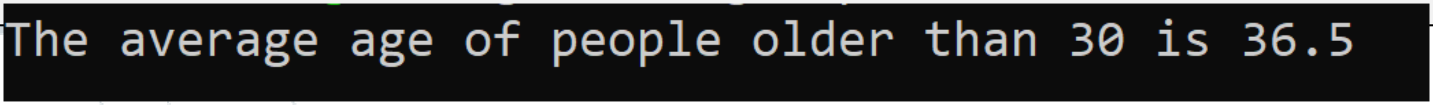
Rayon for Parallel Problems

- Rayon makes the same guarantees: *if you try and do something parallel that won't work parallel, the compiler will still catch that.*

```
125
375
281
343
….
49
57
53
104
61
[240100, 241081, 242064, 243049, 244036, 245025, 246016,
247009, 248004, 249001]
```

Rayon for Map/Reduce

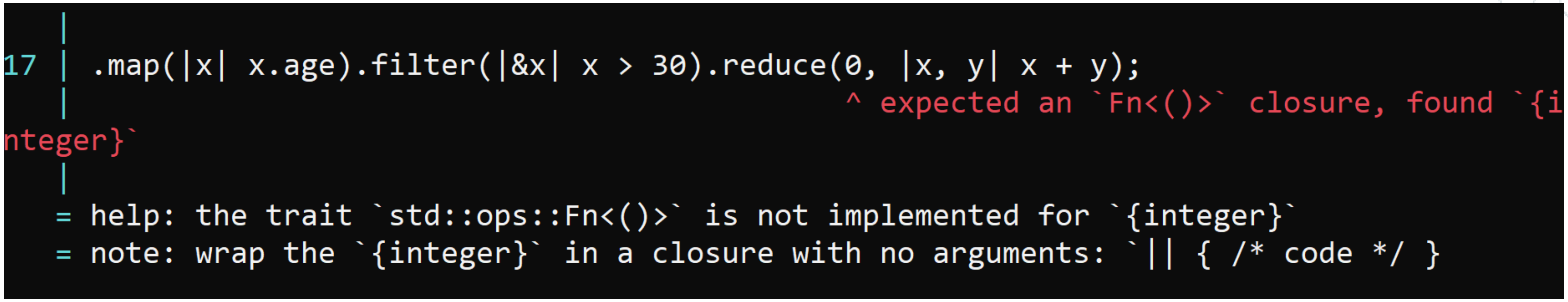
```
use rayon::prelude::*;
struct Person {  age: u32, }
fn main() {
    let v: Vec<Person> = vec![
        Person { age: 23 },
        Person { age: 19 },
        Person { age: 42 },
        Person { age: 17 },
        Person { age: 17 },
        Person { age: 31 },
        Person { age: 30 },
    ];
    let num_over_30 = v.par_iter().filter(|&x| x.age > 
30).count() as f32;
    let sum_over_30 = v.par_iter()
.map(|x| x.age).filter(|&x| x > 30).reduce(|| 0, |x, y| x + y);
    let avg_over_30 = sum_over_30 as f32 / num_over_30;
    println!("The average age of people older than 30 is {}", 
avg_over_30);
}
```



```
The average age of people older than 30 is 36.5
```

Rayon for Map/Reduce

```
let sum_over_30 = v.par_iter()
.map(|x| x.age).filter(|&x| x > 30).reduce(0, |x, y| x + y);
```



```
   |
17 |  .map(|x| x.age).filter(|&x| x > 30).reduce(0, |x, y| x + y);
   |                                              ^ expected an `Fn<()>` closure, found `{i
nteger}`
   |
   = help: the trait `std::ops::Fn<()>` is not implemented for `{integer}`
   = note: wrap the `{integer}` in a closure with no arguments: `|| { /* code */ }
```

- Rayon documentation stops at this point!
- || 0 - is pushing an identity operation to Rayon.
- So, reduce, but not fold, is perhaps reality.