Parallel programming

(without functional programming)

You can run parallel code with std::thread::spawn

```rust
use std::{thread, time::Duration};

const SLEEP_TIME: Duration = Duration::from_millis(1);

pub fn main() {
    let v_sent = vec![1, 2, 3, 4, 5];
    let v_main_thread = vec![6, 7, 8, 9];

    thread::spawn(move || {
        for element in v_sent {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });

    for element in v_main_thread {
        println!("main thread prints {element}");
        thread::sleep(SLEEP_TIME);
    }
}
```

You can wait for the thread to finish with join()

```rust
use std::{thread, time::Duration};

const SLEEP_TIME: Duration = Duration::from_millis(1);

pub fn main() {
    let v_sent = vec![1, 2, 3, 4, 5];
    let v_main_thread = vec![6, 7, 8, 9];

    let handle = thread::spawn(move || {
        for element in v_sent {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });

    for element in v_main_thread {
        println!("main thread prints {element}");
        thread::sleep(SLEEP_TIME);
    }

    handle.join().unwrap();
}
```

If a thread only borrows…

- And I can join manually
  - (which means the thread will stop borrowing my data)
- Surely the borrowing will end… ?

The compiler will reject this (cool, because there is a bug!)

```
use std::{thread, time::Duration};

const SLEEP_TIME: Duration = Duration::from_millis(1);

pub fn values_on_stack() {
    let v_sent = vec![1, 2, 3, 4, 5];

    let handle = thread::spawn(|| {
        for element in &v_sent {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });
}
```

The compiler will reject this (cool, because there is a bug!)

```
use std::{thread, time::Duration};

const SLEEP_TIME: Duration = Duration::from_millis(1);

pub fn values_on_stack() {
    let v_sent = vec![1, 2, 3, 4, 5];

    let handle = thread::spawn(|| {
        for element in &v_sent {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });
}
```

values_on_stack finishes before the thread, so the reference is invalid

```
error[E0373]: closure may outlive the current function, but it borrows `v_sent`, which is owned by the current function
 --> <source>:8:32
  |
8 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v_sent`
9 |         for element in &v_sent {
  |                        ------ `v_sent` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> <source>:8:18
  |
8 |       let handle = thread::spawn(|| {
  |  __________________^
9 | |         for element in &v_sent {
10 | |             println!("thread prints {element}");
11 | |             thread::sleep(SLEEP_TIME);
12 | |         }
13 | |     });
  | |_____^
help: to force the closure to take ownership of `v_sent` (and any other referenced variables), use the `move` keyword
  |
8 |     let handle = thread::spawn(move || {
  |                                ++++
For more information about this error, try `rustc --explain E0373`.
```

rustc --explain E0373

```
fn foo() {
        let x = 0u32;
        let y = 1u32;

        let thr = std::thread::spawn(|| {
        x + y
        });
}
```

Since our new thread runs in parallel, the stack frame containing x and y may well have disappeared by the time we try to use them. Even if we call thr.join() within foo (which blocks until thr has completed, ensuring the stack frame won't disappear), we will not succeed: the compiler cannot prove that this behavior is safe, and so won't let us do it.

The solution to this problem is usually to switch to using a move closure. This approach moves (or copies, where possible) data into the closure, rather than taking references to it. For example:

```
fn foo() -> Box<dyn Fn(u32) -> u32> {
        let x = 0u32;
        Box::new(move |y| x + y)
}
```

Solution 1: use scoped thread

```
use std::{thread, time::Duration};

const SLEEP_TIME: Duration = Duration::from_millis(1);

pub fn main() {
    let v_sent = vec![1, 2, 3, 4, 5];

    thread::scope(|s| {
        s.spawn(|| {
            for element in &v_sent {
                println!("thread prints {element}");
                thread::sleep(SLEEP_TIME);
            }
        });
    });

    println!("{:?}", v_sent);
}
```

Guaranteed to join!

Solution 2: clone

```
pub fn main() {
    let v_sent = vec![1, 2, 3, 4, 5];
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        for element in clone {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });

    println!("{:?}", v_sent);

    handle.join().unwrap();
}
```

Solution 3a: Arc

```
pub fn main() {
    let v_sent = std::sync::Arc::new(vec![1, 2, 3, 4, 5]);
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        for element in clone.iter() {
            println!("thread prints {element}");
            thread::sleep(SLEEP_TIME);
        }
    });

    println!("{:?}", v_sent);

    handle.join().unwrap();
}
```

Solution 3b: for mutability, Arc + Mutex

```
pub fn main() {
    let v_sent = std::sync::Arc::new(std::sync::Mutex::new(vec![1, 2, 3, 4, 5]));
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        let unique_access_vec = clone.lock().expect("cannot lock the mutex");
        for element in unique_access_vec.iter() {
            println!("thread prints {}", *element);
            thread::sleep(SLEEP_TIME);
        }
    });

    let unique_access_vec = v_sent.lock().expect("cannot lock the mutex");
    for element in unique_access_vec.iter() {
        println!("main thread prints {}", *element);
        thread::sleep(SLEEP_TIME);
    }

    handle.join().unwrap();
}
```

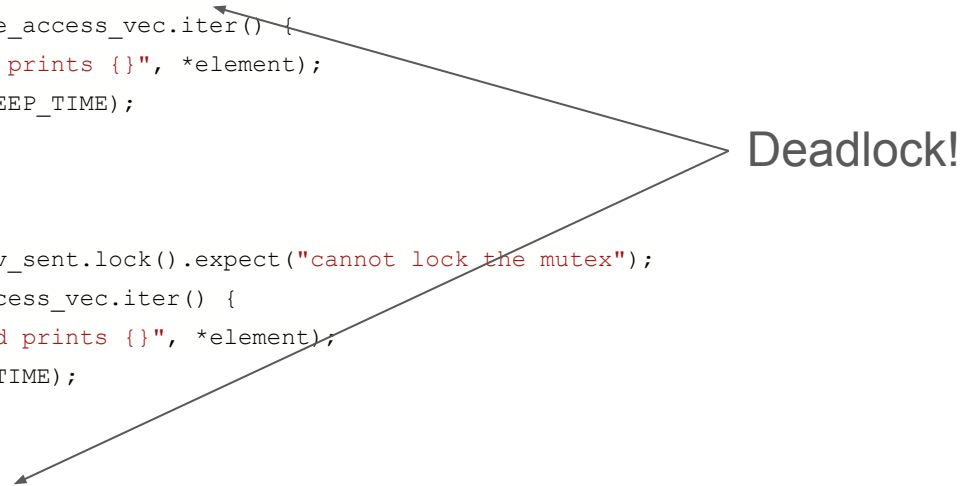
Solution 3b: for mutability, Arc + Mutex

```
pub fn main() {
    let v_sent = std::sync::Arc::new(std::sync::Mutex::new(vec![1, 2, 3, 4, 5]));
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        let unique_access_vec = clone.lock().expect("cannot lock the mutex");
        for element in unique_access_vec.iter() {
            println!("thread prints {}", *element);
            thread::sleep(SLEEP_TIME);
        }
    });

    let unique_access_vec = v_sent.lock().expect("cannot lock the mutex");
    for element in unique_access_vec.iter() {
        println!("main thread prints {}", *element);
        thread::sleep(SLEEP_TIME);
    }

    handle.join().unwrap();
}
```

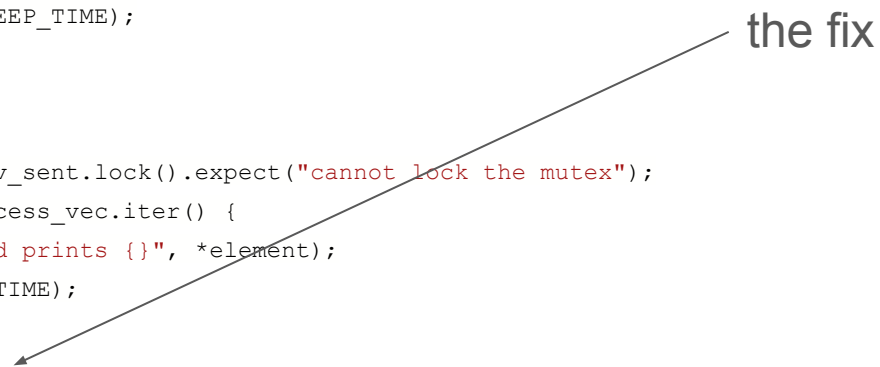


Solution 3b: for mutability, Arc + Mutex

```
pub fn main() {
    let v_sent = std::sync::Arc::new(std::sync::Mutex::new(vec![1, 2, 3, 4, 5]));
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        let unique_access_vec = clone.lock().expect("cannot lock the mutex");
        for element in unique_access_vec.iter() {
            println!("thread prints {}", *element);
            thread::sleep(SLEEP_TIME);
        }
    });

    let unique_access_vec = v_sent.lock().expect("cannot lock the mutex");
    for element in unique_access_vec.iter() {
        println!("main thread prints {}", *element);
        thread::sleep(SLEEP_TIME);
    }
    drop(unique_access_vec);
    handle.join().unwrap();
}
```



Solution 3b: for mutability, Arc + Mutex

```
pub fn main() {
    let v_sent = std::sync::Arc::new(std::sync::RwLock::new(vec![1, 2, 3, 4, 5]));
    let clone = v_sent.clone();

    let handle = thread::spawn(move || {
        let unique_access_vec = clone.read().expect("cannot lock the mutex");
        for element in unique_access_vec.iter() {
            println!("thread prints {}", *element);
            thread::sleep(SLEEP_TIME);
        }
    });

    let unique_access_vec = v_sent.read().expect("cannot lock the mutex");
    for element in unique_access_vec.iter() {
        println!("main thread prints {}", *element);
        thread::sleep(SLEEP_TIME);
    }
    drop(unique_access_vec);
    handle.join().unwrap();
}
```

a reader does not block other reader