# CS 420 Advanced Programming Languages
Fall Semester, 2022

# Doc 9 Rust Concurrency
Sep 22, 2022

# Rust Concurrent Features

Fork-Join Parallelism

Channels

Shared mutable state

```rust
use std::iter::zip;
use std::time::Duration;
use std::thread;
fn looper(label: &str,  n: i32) {
    for k in 0..n {
        println!("{}: {}", label, k);
        thread::sleep(Duration::from_millis(1));
    }
}

fn main() {
    let mut thread_handles = vec![];
    let a = zip([5, 5, 5], ["a", "b", "c"]);
    for (amount, label) in a {
        thread_handles.push(thread::spawn(move || looper(label, amount)));
        };
    for handle in thread_handles {
        handle.join().unwrap();
    }
}
```

Output

```
c: 0
a: 0
b: 0
c: 1
b: 1
a: 1
a: 2
b: 2
c: 2
a: 3
b: 3
c: 3
a: 4
c: 4
b: 4
```

# Sharing Data Among Threads

```
struct SampleLargeStruct {
    sample_large_struct: [u8; 1024 * 1024]
}

fn looper(label: &str,  n: i32,data: Arc<SampleLargeStruct>) {
    for k in 0..n {
        println!("{}: {}", label, k);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
let data = Arc::new(SampleLargeStruct { sample_large_struct: [0; 1024 * 1024] });
let mut handles = vec![];
for i in 0..10 {
    let data = data.clone();                    // Just clone Arc
    let handle = thread::spawn(move || {
        looper(&format!("thread {}", i), 10, data);
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
```

# Rayon - Library to Simplify Parallelism

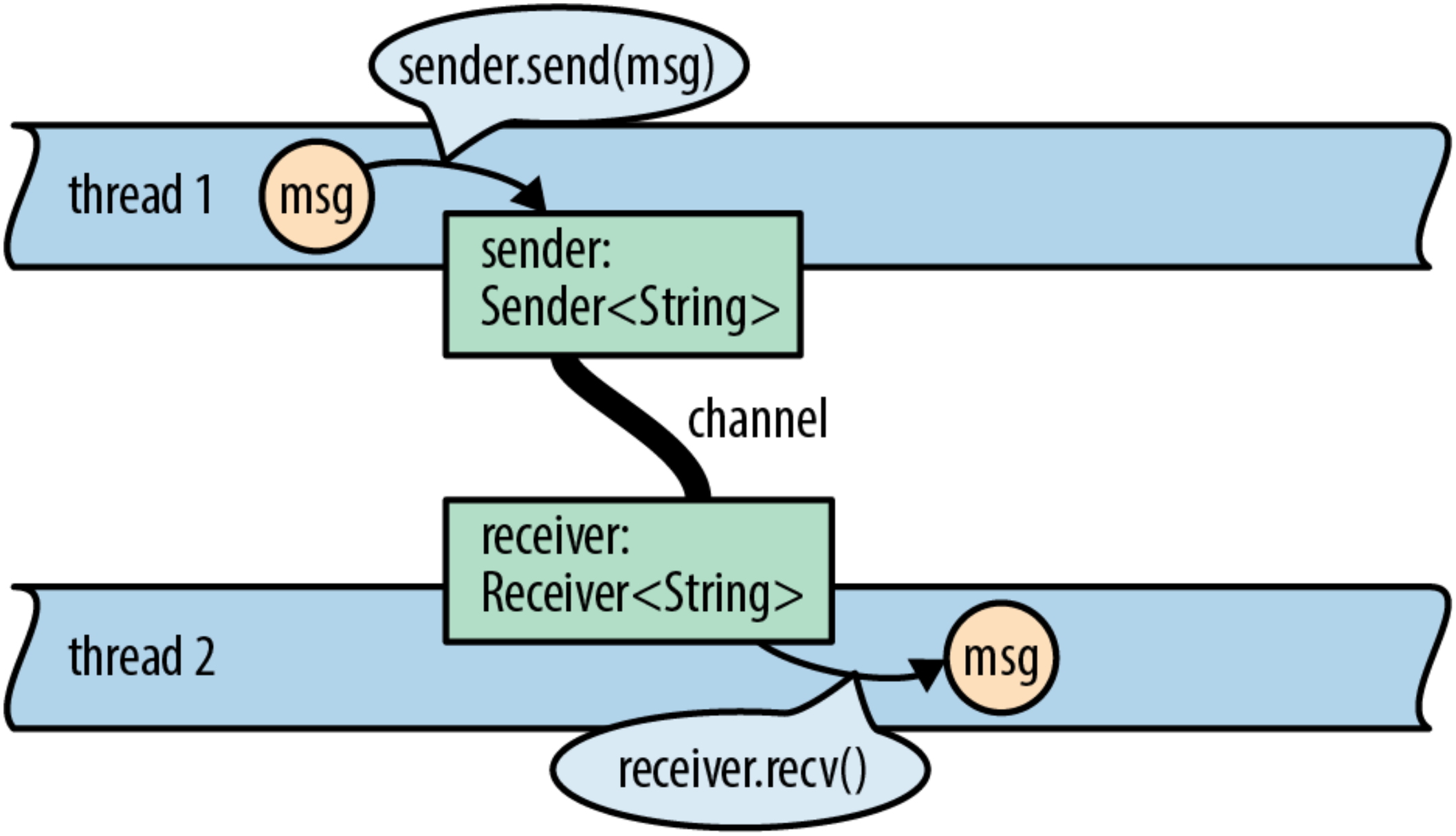
```
use rayon::prelude::*;

fn main() {
    let mut arr = [0, 7, 9, 11];
    arr.par_iter_mut().for_each(|p| *p -= 1);
    println!("{:?}", arr);

    let mut vec = vec![2, 4, 6, 8];

    assert!(!vec.par_iter().any(|n| (*n % 2) != 0));
    assert!(vec.par_iter().all(|n| (*n % 2) == 0));
    assert!(!vec.par_iter().any(|n| *n > 8 ));
    assert!(vec.par_iter().all(|n| *n <= 8 ));
}
```

# Channels - One-Way message send



# Sending Message

```
let documents = vec!["index.html", "about.html", "contact.html"];
let (sender , receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    let mut response : Result<&str,&str>;
    for file in documents {
        if sender.send(file).is_err() {
            response = Err("Error sending data");
        }
    }
    response = Ok("Success");
});


for text in receiver {
    println!("{}", text);
}
```

receiver read block until message arrives

# Multiple Senders, Only one Receiver

```
let documents = vec!["index.html", "about.html", "contact.html"];
 let (sender , receiver) = mpsc::channel();
 for file in documents {
     let sender = sender.clone();
     let handle = thread::spawn(move || {
         let mut response : Result<&str,&str>;
         let text = file;
         if sender.send(text).is_err() {
             response = Err("Error sending data");
         }
         response = Ok("Data sent");
         response;
     });
};

for text in receiver {
       println!("{}", text);
}
```

# Deadlock

```
let documents = vec!["index.html", "about.html", "contact.html"];
let (sender , receiver) = mpsc::channel();

let handle = thread::spawn(move || {
   let text = if let Ok(text) = receiver.recv() { text} else { "Something happened" };

    let mut response : Result<&str,&str>;
    for file in documents {
        if sender.send(file).is_err() {
            response = Err("Error sending data");
        }
    }
    response = Ok("Success");
});
```

# Sharing mutable data between threads

Way of sharing results

But what happens when multiple threads try to access the same data simultaneously

- Multiple reads OK
- Multiple writes - Bad
- Read & Write - Bad

# Atomic vs Non-Atomic Writes

Atomic Writes

- Done with one machine instruction
- Writing one word
- Two threads can not write at the same time on one data element

Non-Atomic Writes

- Take multiple steps to complete

| Name | Amount | Source | Destination | Date |
|---|---|---|---|---|
| | | | | |

# Locks

With shared mutable data

- Require thread to obtain a lock before doing write or read
- After done thread releases the lock

Semaphores

- Lock separate from data or critical statements

Monitors

- Hide data in structure/object
- Methods provide the lock

# Semaphores

Way to provide a lock

Semaphore

Data structure with 2 operations

Specify how many threads can obtain lock at the same time

wait

If another thread has a lock the calling thread blocks until the lock is released

When one read releases lock on thread block on wait is given the lock

release

Called to release the lock

# Kotlin Example

```
val s = Semaphore(1)
val bankTransaction = Transaction()
val t1 = Thread(Runnable
{
    s.wait()
    bankTransaction.amount = -1,000
    bankTransaction.date = 9/20/22
    s.release()
})

val t2 = Thread(Runnable
{
    s.wait()
    bankTransaction.date = 9/18/22
    bankTransaction.amount = 2,000
    s.release()
})
```

Critical section

Code that can only be run by one thread at a time

# Not Safe

```
val s = Semaphore(1)
val bankTransaction = Transaction()
val t1 = Thread(Runnable
{
    s.wait()
    bankTransaction.amount = -1,000
    bankTransaction.date = 9/20/22
    s.release()
})

val t2 = Thread(Runnable
{

    bankTransaction.date = 9/18/22
    bankTransaction.amount = 2,000

})
```

Critical section

Code that can only be run by one thread at a time

# Rust - Mutex

A mutual exclusion primitive useful for protecting shared data

Data is placed inside the mutex

To access the data call lock on the mutex

- If another thread has the lock then lock blocks the calling thread

lock()
- Request a lock, blocks

try_lock()
- Request a lock, non-blocking

```
use std::sync::Mutex;

#[derive(Debug)]
struct Transaction<'a> {
    from: Box<&'a str>,
    to: Box<&'a str>,
    amount: u32,
}

fn main() {
    let transaction = Mutex::new(Transaction {
        from: Box::from("Alice"),
        to: Box::from("Bob"),
        amount: 100,
    });
    let mut transaction_changer :MutexGuard<Transaction> = transaction.lock().unwrap();
    *transaction_changer.from = "Charlie";
     println!("Transaction: {:?}", transaction_changer);
}
```

# Releasing the Lock

The lock is released

When the MutexGuard goes out of scope, or

call drop()

```
fn main() {
    let transaction = Mutex::new(Transaction {
        from: Box::from("Alice"),
        to: Box::from("Bob"),
        amount: 100,
    });
    {
        let mut transaction_changer :MutexGuard<Transaction> = transaction.lock().unwrap();
        *transaction_changer.from = "Charlie";
        println!("Transaction: {:?}", transaction_changer);
    }
    //Lock released here
    println!("Transaction: {:?}", transaction);
}
```

# Releasing the Lock

The lock is released

When the MutexGuard goes out of scope, or

call drop()

```
fn main() {
    let transaction = Mutex::new(Transaction {
        from: Box::from("Alice"),
        to: Box::from("Bob"),
        amount: 100,
    });

    let mut transaction_changer :MutexGuard<Transaction> = transaction.lock().unwrap();
    *transaction_changer.from = "Charlie";
    println!("Transaction: {:?}", transaction_changer);

    drop(transaction_changer);
}
```

# Example with Threads

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc::channel;
const N: usize = 10;

fn main() {
    let data = Arc::new(Mutex::new(0));

    let (tx, rx) = channel();
    for _ in 0..N {
        let (data, tx) = (Arc::clone(&data), tx.clone());
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;
            if *data == N {
                tx.send(()).unwrap();
            }
        });
    }
    rx.recv().unwrap();
}
```

# Poisoning the Mutex

If a thread panics while holding a mutex lock

- The mutex is poisoned
- Default - other threads can not access the data in the mutex

# Recovering from Poisoned Mutex

```
use std::sync::{Arc, Mutex};
use std::thread;

let lock = Arc::new(Mutex::new(0_u32));
let lock2 = Arc::clone(&lock);

let _ = thread::spawn(move || -> () {
    let _guard = lock2.lock().unwrap();
    panic!();
}).join();

let mut guard = match lock.lock() {
    Ok(guard) => guard,
    Err(poisoned) => poisoned.into_inner(),
};

*guard += 1;
```

# Java Synchronized Methods

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

While a thread calls a synchronized method on an object other threads block on calling a synchronized method on the same object

# Java Synchronized Statements

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```