Library Management System

Using Multi-Phased Concurrency and Shell Commands

Grant Versluis
Section 01
NETID: gverslui@students.kennesaw.edu

February 28, 2025

1 Introduction

This project demonstrates a multi-phase approach to handling threading operations in Rust, along with inter-process communication (IPC) using shell commands. Specifically, the goals are:

- Create various threading scenarios (Phases 1 to 4) to explore concurrency, synchronization, and potential deadlock conditions.
- Use `ls | grep 'csv'` to identify CSV files in the current directory, parse them, and generate a collection of `Book` structures.
- Show how Rust's ownership model and the borrow checker influence design decisions.

By structuring the project into phases, each focusing on a different aspect of concurrency, we illustrate both common pitfalls (e.g., borrow checker conflicts, deadlocks) and their resolutions (e.g., using mutexes and consistent lock ordering).

2 Implementation Details

This section discusses how the project is organized, including threading details and the IPC mechanism used to discover `.csv` files for creating books in a library.

2.1 Project Structure and Entry Point (`main`)

The overall flow starts in `main`, where we:

1. Call `new_book_shipment()` to parse CSV files using a shell pipeline.
2. Create initial user data.
3. Execute Phases 1 through 4 sequentially.

Listing 1: main.rs - Entry Point and Phase Invocations

```
mod book;
mod book_shipment_log;
mod book_action_logger;

use book::{Book, User, fill_book_data, fill_user_data};
use book_shipment_log::*;
use book_action_logger::*;

use rand::{Rng};
use std::io::{stdin};
use std::thread;
use std::thread::*;
use std::sync::{Arc, Mutex};
use std::time::{Duration, Instant};

fn main() {
    // Create some dummy data.
    let library = match new_book_shipment() {
        Ok(library) => library,
        Err(e) => {
            eprintln!("{e}");
            return
        },
    };

    let users = vec![
        User { username: "Alice".into(), loaned_books: Vec::new() },
        User { username: "Bob".into(), loaned_books: Vec::new() },
        User { username: "Charlie".into(), loaned_books: Vec::new() },
    ];

    // Run each phase.
```

```
    phase_1(&mut library.clone(), &mut users.clone());
    phase_2(library.clone(), users.clone());
    phase_3(library.clone(), users.clone());
    phase_4(library.clone(), users.clone());
}
```

2.2 IPC: Discovering `.csv` Files with a Shell Command

To create the initial `library`, we use a shell command pipeline:

```
ls | grep 'csv'
```

This command finds CSV files in the current directory, which are then opened, read, and converted into `Book` objects.

Listing 2: book_shipment_log.rs - IPC for CSV Discovery

```
use std::process::{Command, Stdio};
use std::str;
use std::fs::File;
use std::io::{BufReader, BufRead};
use std::error::Error;
use crate::book::Book;
use rand::Rng;

pub fn new_book_shipment() -> Result<Vec<Book>, Box<dyn Error + Send + Sync>> {
    let mut rng = rand::rng();

    let output = Command::new("sh")
        .arg("-c")
        .arg("ls | grep '\\.csv'")
        .stdout(Stdio::piped())
        .output()
        .expect("Failed to execute ls");

    if !output.status.success() {
        return Err(Box::<dyn Error + Send + Sync>::from("Error running search
            command..."));
    }

    let stdout = String::from_utf8(output.stdout).expect("Failed to parse output")
        ;

    let file_names: Vec<&str> = stdout.lines().collect();
    let mut library: Vec<Book> = Vec::new();

    for file in file_names {
        println!("Processing file: {}", file);
```

```
        let f = File::open(file).expect("Failed to open file");
        let reader = BufReader::new(f);

        for (i, line) in reader.lines().enumerate() {
            let line = line.expect("Failed to read line");
            if i == 0 { continue } // Skip headers
            let parts: Vec<&str> = line.split(',')
                .map(|s| s.trim())
                .collect();

            if parts.len() != 3 {
                eprintln!("Invalid CSV Format in file {}: {}", file, line);
                continue;
            }

            let mut num = rng.random_range(0..2);
            let borrow = match num {
                0 => true,
                1 => false,
                _ => false,
            };

            let book = Book{
                title: parts[0].into(),
                author: parts[1].into(),
                isbn: parts[2].into(),
                borrowed: borrow,
            };

            library.push(book);
        }
    };

    Ok(library)
}
```

2.3 Threading Phases

The concurrency portion is demonstrated in four phases, each exploring different threading and synchronization strategies.

2.3.1 Phase 1: Basic Thread Operations (No Mutex Protection)

- Spawns 100 threads for borrowing books and 100 threads for returning books.
- Clones the library and users to avoid conflicts with the borrow checker.

Listing 3: phase_1 function - Basic Thread Operations

```
fn phase_1(lib: &mut Vec<Book>, memb: &mut Vec<User>) {
    println!("--- Phase 1: Basic Thread Operations ---");

    // Borrowing phase
    for _ in 0..100 {
        let book_index = random_range(0, lib.len());
        let user_index = random_range(0, memb.len());

        unsafe {
            scope(|s| {
                s.spawn(|| {
                    if !lib[book_index].borrowed {
                        memb[user_index].add_book(&mut lib[book_index]);
                        println!(
                            "Phase 1: User '{}' borrowed book '{}'",
                            memb[user_index].username,
                            lib[book_index].title
                        );
                    }
                })
                .join()
                .expect("Thread panicked");
            });
        }
    }

    // Returning phase
    for _ in 0..100 {
        let user_index = random_range(0, memb.len());
        thread::scope(|s| {
            s.spawn(|| {
                if !memb[user_index].loaned_books.is_empty() {
                    memb[user_index].remove_return_book();
                    println!(
                        "Phase 1: User '{}' returned a book",
                        memb[user_index].username
                    );
                }
            })
            .join()
            .expect("Thread panicked");
        });
    }

    println!("--- End Phase 1 ---\n");
}
```

2.3.2 Phase 2: Resource Protection with Mutexes

- Wraps the `library` and `users` vectors in `Arc<Mutex<...>>`.
- Spawns 100 threads, each performing borrowing and returning under mutex protection.

Listing 4: phase_2 function - Using Arc¡Mutex¿ for Concurrency

```
fn phase_2(lib: Vec<Book>, memb: Vec<User>) {
    println!("--- Phase 2: Resource Protection with Mutex ---");

    let lib = Arc::new(Mutex::new(lib));
    let memb = Arc::new(Mutex::new(memb));

    thread::scope(|s| {
        let mut handles = Vec::new();

        for _ in 0..100 {
            let lib_clone = Arc::clone(&lib);
            let memb_clone = Arc::clone(&memb);

            handles.push(s.spawn(move || {
                // Borrowing operation
                {
                    let mut lib_guard = lib_clone.lock().unwrap();
                    let mut memb_guard = memb_clone.lock().unwrap();
                    let book_index = random_range(0, lib_guard.len());
                    let user_index = random_range(0, memb_guard.len());

                    if !lib_guard[book_index].borrowed {
                        memb_guard[user_index].add_book(&mut lib_guard[book_index]);
                        println!(
                            "Phase 2: (Mutex) User '{}' borrowed book '{}'",
                            memb_guard[user_index].username,
                            lib_guard[book_index].title
                        );
                    }
                }

                // Returning operation
                {
                    let mut memb_guard = memb_clone.lock().unwrap();
                    let user_index = random_range(0, memb_guard.len());
                    if !memb_guard[user_index].loaned_books.is_empty() {
                        memb_guard[user_index].remove_return_book();
                        println!(
                            "Phase 2: (Mutex) User '{}' returned a book",
                            memb_guard[user_index].username
                        );
```

```
                }
            }
        }));
    }

    // Join all threads
    for handle in handles {
        let _ = handle.join();
    }
});

println!("--- End Phase 2 ---\n");
}
```

2.3.3 Phase 3: Deadlock Creation

- Demonstrates how deadlocks can arise if threads lock resources in inconsistent orders.
- Uses a custom `try_lock_with_timeout()` function to show deadlock or timeout.

Listing 5: phase_3 function - Intentional Deadlock Creation

```
fn phase_3(lib: Vec<Book>, memb: Vec<User>) {
    println!("--- Phase 3: Deadlock Creation with Timeout ---");

    let lib = Arc::new(Mutex::new(lib));
    let memb = Arc::new(Mutex::new(memb));

    let timeout = std::time::Duration::from_secs(2);

    scope(|s| {
        // Thread A: Lock library first, then members
        let lib_a = Arc::clone(&lib);
        let memb_a = Arc::clone(&memb);
        let handle_a = s.spawn(move || {
            let _lib_guard = lib_a.lock().unwrap();
            println!("Phase 3: Thread A locked library");
            thread::sleep(std::time::Duration::from_millis(50));
            match try_lock_with_timeout(&memb_a, timeout) {
                Some(_memb_guard) => {
                    println!("Phase 3: Thread A locked members");
                }
                None => {
                    println!("Phase 3: Thread A timed out waiting for members lock
                        ");
                }
            }
```

```
        });

        // Thread B: Lock members first, then library
        let lib_b = Arc::clone(&lib);
        let memb_b = Arc::clone(&memb);
        let handle_b = s.spawn(move || {
            let _memb_guard = memb_b.lock().unwrap();
            println!("Phase 3: Thread B locked members");
            thread::sleep(std::time::Duration::from_millis(50));
            match try_lock_with_timeout(&lib_b, timeout) {
                Some(_lib_guard) => {
                    println!("Phase 3: Thread B locked library");
                }
                None => {
                    println!("Phase 3: Thread B timed out waiting for library lock
                        ");
                }
            }
        });

        let _ = handle_a.join();
        let _ = handle_b.join();
    });

    println!("--- End Phase 3 ---\n");
}
```

2.3.4 Phase 4: Deadlock Resolution

- Both threads lock the resources in the same order (library first, then members), allowing us to avoid circular waiting.

Listing 6: phase_4 function - Enforcing Consistent Lock Ordering

```
fn phase_4(lib: Vec<Book>, memb: Vec<User>) {
    println!("--- Phase 4: Deadlock Resolution ---");

    let lib = Arc::new(Mutex::new(lib));
    let memb = Arc::new(Mutex::new(memb));

    thread::scope(|s| {
        // Thread A
        let lib_a = Arc::clone(&lib);
        let memb_a = Arc::clone(&memb);
        let handle_a = s.spawn(move || {
            let lib_guard = lib_a.lock().unwrap();
            println!("Phase 4: Thread A locked library");
```

```
            thread::sleep(std::time::Duration::from_millis(50));
            let _memb_guard = memb_a.lock().unwrap();
            println!("Phase 4: Thread A locked members");
        });

        // Thread B
        let lib_b = Arc::clone(&lib);
        let memb_b = Arc::clone(&memb);
        let handle_b = s.spawn(move || {
            let lib_guard = lib_b.lock().unwrap();
            println!("Phase 4: Thread B locked library");
            thread::sleep(std::time::Duration::from_millis(50));
            let _memb_guard = memb_b.lock().unwrap();
            println!("Phase 4: Thread B locked members");
        });

        let _ = handle_a.join();
        let _ = handle_b.join();
    });

    println!("--- End Phase 4 ---\n");
}
```

2.4 Helper Functions

The code below includes utility functions for random range generation and attempting to lock a `Mutex` with a timeout. These are helpful for demonstrating deadlock or contention scenarios:

Listing 7: Helper Functions for Random Range and Timed Lock

```
fn random_range(min: usize, max: usize) -> usize {
    let mut rng = rand::thread_rng();
    rng.gen_range(min..max)
}

/// Try to lock a mutex, but return None if the lock isn't acquired within '
    timeout'.
fn try_lock_with_timeout<T>(
    mutex: &Mutex<T>,
    timeout: Duration
) -> Option<std::sync::MutexGuard<T>> {
    let start = Instant::now();
    loop {
        match mutex.try_lock() {
            Ok(guard) => return Some(guard),
            Err(_) => {
                if start.elapsed() >= timeout {
```

```
                    return None;
                }
                std::thread::sleep(Duration::from_millis(10));
            }
        }
    }
}
```

3 Environment Setup and Tool Usage

All code was developed using **Ubuntu** as the base operating system with the **RustRover** IDE. Rust (stable toolchain) was installed via `rustup`. A few items to note:

- Setting up **IPC** required making sure `bash` or `sh` was available for executing commands like `ls | grep`.
- Rust's **borrow checker** initially posed challenges for passing references to threads, leading to frequent use of `Arc` and `Mutex`.
- **Cargo.toml** was used to manage dependencies such as `rand` and `any` external libraries.

4 Challenges and Solutions

4.1 Borrow Checker and Data Cloning

A major challenge involved handling Rust's borrow checker when multiple threads need to access or modify the same data. Solutions included:

- Cloning vectors (`.clone()`) to avoid partial moves or invalid references.
- Wrapping shared data in `Arc<Mutex<...>` to manage concurrency safely.

4.2 Deadlock Scenarios

Intentionally creating a deadlock in **Phase 3** highlighted how inconsistent locking order across multiple threads can stall the program. The resolution in **Phase 4** was to adopt a strict lock order policy (always lock library first, then members).

4.3 IPC and Environment Issues

Some minor environment-related issues arose when the shell commands failed if `grep` was not present or if the project directory had no `.csv` files. Adding error handling and ensuring the environment supports these commands solved the problem.

5 Results and Outcomes

- **Phase 1** successfully spawned 200 total threads (100 borrowing, 100 returning) without conflicting with the borrow checker by cloning.
- **Phase 2** confirmed that using `Arc<Mutex>` allows shared, mutable data to be safely accessed by multiple threads, avoiding data races.
- **Phase 3** demonstrated deadlock creation; the locking order was intentionally mismatched, resulting in one or both threads timing out.
- **Phase 4** successfully avoided deadlocks by standardizing lock order, proving that consistent ordering is an effective solution.
- **IPC** using `ls | grep 'csv'` worked as expected, generating a collection of `Book` structs from any CSV files found.

Performance-wise, each phase ran as expected during stress tests with up to 200 threads. While overhead from locking was minimal, future enhancements might include exploring lock-free or more fine-grained locking designs.

6 Reflection and Learning

Through this project, I gained:

- A deeper understanding of Rust's concurrency model and how to respect the ownership/borrowing rules in multi-threaded contexts.
- Experience handling deadlocks by controlling the lock acquisition order (**Phase 3 vs. Phase 4**).
- Practical insight into basic inter-process communication using shell commands, ensuring the Rust code can interface with system utilities.
- Appreciation for how `Arc<Mutex>` allows safe sharing but can also introduce performance bottlenecks if misapplied.

This experience significantly broadened my knowledge of both operating systems concepts (threading, synchronization, IPC) and Rust's unique approach to memory safety.

7 References

- **Rust Documentation.**
  `https://doc.rust-lang.org/book/`
- **IEEE Citation Format.**
  `https://www.ieee.org/documents/ieeecitationref.pdf`
- Official **Cargo** and **Crates.io** references.
  `https://doc.rust-lang.org/cargo/`