.UBAfiuba
FACULTAD DE INGENIERÍA

Guía de Ejercicios 3: Concurrencia en Rust

75.42 / 95.08 Taller de Programación I - 1er C 2021
Ing. Pablo Deymonnaz

Ejercicio 1 - Cuentas bancarias

El fragmento de código 1 hace uso de *unsafe* para poder mutar una variable global. Esto introduce condiciones de carrera sobre los datos (*data races*) que provocan que el programa falle de manera imprevista al correrlo repetidas veces. Es decir, el problema se presenta en alguno de los posibles escenarios de ejecución.

Corregir el programa haciendo uso de las abstracciones que provee Rust para el manejo de la concurrencia de manera que no se produzcan errores.

Ejercicio 2 - ThreadPool

Un *threadpool* mantiene varios hilos de ejecución (*threads*) en espera de que el programa supervisor asigne tareas para su ejecución simultánea. Al mantener un grupo de threads, el modelo aumenta el rendimiento y evita la latencia en la ejecución debido a la frecuente creación y destrucción de threads para tareas de corta duración.

En este ejercicio se debe armar un *threadpool* sencillo haciendo uso de las herramientas para computación concurrente que nos provee la biblioteca estándar de Rust.

Para distribuir las tareas a realizar entre los threads del pool se puede utilizar una cola concurrente.

Consideraciones a tener en cuenta:

1. La estructura de datos utilizada para distribuir el trabajo.
2. ¿Que se hace cuando una tarea enviada al threadpool provoca que un thread muera? Esta situación no debería afectar a otros threads. Ademas tras la muerte de un thread, se debe crear otro de forma de que la cantidad total de threads en el pool no cambie.
3. Cuando la threadpool es terminada o sale de scope todos los threads deberian finalizar.

El fragmento de código 2 muestra un ejemplo de uso:

Ejercicio 3 - Contar palabras concurrente

Escribir un programa, basado en el ejercicio 2 de la guía 1, para contar las frecuencias de palabras únicas leídas desde varios archivos de entrada.

La lectura y procesamiento de los archivos debe ser realizada de forma concurrente. Una vez finalizado el procesamiento de los mismos, imprimirlos con sus frecuencias, ordenados primero por las más frecuentes.

Realizar las siguientes implementaciones y comparar los tiempos de ejecución:

- Un mapa de resultados parciales por thread (por archivo), unir las sumas parciales al hacer *join()*, utilizando el valor de retorno de los hilos.
- Un mapa de resultados parciales por thread, enviar las sumas parciales de los threads utilizando channels.

- Un mapa de resultados globales accedidos por thread.

Requerimientos no funcionales

Los siguientes son los requerimientos no funcionales para la resolución de los ejercicios:

- El proyecto deberá ser desarrollado en lenguaje Rust, usando las herramientas de la biblioteca estándar.
- No se permite utilizar *crates* externos. El único crate autorizado a ser utilizado es *rand*[1] en caso de que se quiera generar valores aleatorios.
- El código fuente debe compilarse en la versión stable del compilador y no se permite utilizar bloques unsafe.
- El código deberá funcionar en ambiente Unix / Linux.
- El programa deberá ejecutarse en la línea de comandos.
- La compilación no debe arrojar *warnings* del compilador, ni del linter *clippy*.
- Las funciones y los tipos de datos (*struct*) deben estar documentadas siguiendo el estándar de *cargo doc*.
- El código debe formatearse utilizando *cargo fmt*.
- Las funciones no deben tener una extensión mayor a 30 líneas. Si se requiriera una extensión mayor, se deberá particionarla en varias funciones.
- Cada tipo de dato implementado debe ser colocado en una unidad de compilación (archivo fuente) independiente.

[1] https://crates.io/crates/rand

```rust
use std::thread;

struct Account(i32);

impl Account {
        fn deposit(&mut self, amount: i32) {
                println!("op: deposit {}, available funds: {:?}", amount, self.0);
                self.0 += amount;
        }

        fn withdraw(&mut self, amount: i32) {
                println!("op: withdraw {}, available funds: {}", amount, self.0);
                if self.0 >= amount {
                        self.0 -= amount;
                } else {
                        panic!("Error: Insufficient funds.")
                }
        }

        fn balance(&self) -> i32 {
                self.0
        }
}

static mut ACCOUNT: Account = Account(0);

fn main() {
        let customer1_handle = thread::spawn(move || unsafe {
                ACCOUNT.deposit(40);
        });

        let customer2_handle = thread::spawn(move || unsafe {
                ACCOUNT.withdraw(30);
        });

        let customer3_handle = thread::spawn(move || unsafe {
                ACCOUNT.deposit(60);
        });

        let customer4_handle = thread::spawn(move || unsafe {
                ACCOUNT.withdraw(70);
        });

        let handles = vec![
        customer1_handle,
        customer2_handle,
        customer3_handle,
        customer4_handle,
        ];

        for handle in handles {
                handle.join().unwrap();
        }

        let savings = unsafe { ACCOUNT.balance() };

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

Listing 1: Fragmento de código 1

```
fn main() {
        let pool = ThreadPool::new(4);
        for i in 0..4 {
                pool.spawn(move || {
                        std::thread::sleep(std::time::Duration::from_millis(250 * i));
                        println!("This is Task {}", i);
                });
        }
        std::thread::sleep(std::time::Duration::from_secs(2));
}
```

Listing 2: Fragmento de código 2