Problema del banquero

Técnicas: Uso de barriers para sincronizar momentos críticos en el flujo de ejecución de múltiples hilos.

Han habido algunos problemas de organización, y los hijos del señor banquero, si bien desean que los inversores sigan trabajando de forma autónoma, deben tomar el dinero de la cuenta y devolverlo al final de la semana. Cada inversor debe tomar exactamente el mismo dinero.

```
const FRIENDS: u32 = 10;
```

“Al tiempo el señor banquero falleció. Los hijos deciden que los inversores sigan trabajando el dinero pero ellos no se hacen cargo de nada. Los inversores solo deberán tomar el dinero de la cuenta al inicio de la semana y devolverlo al final.”

```
fn main() {
    let money: f64 = 1000.0;
    let lock = Arc::new(RwLock::new(money));
    let barrier = Arc::new(Barrier::new(FRIENDS as usize));
    let barrier2 = Arc::new(Barrier::new(FRIENDS as usize));
```

`Arc` para compartir recursos entre threads.
`Rwlock` para la lectura de forma concurrente y escritura exclusiva del dinero.
`Barrier` para sincronizar varios hilos en un punto de encuentro (sincronización de los inversores).

Monto inicial de $1000.
Barrier 1: sincronización para el inicio de la semana.
Barrier 2: sincroniza el momento de leer el saldo (antes de tomar el dinero).

```
    let mut friends = vec![];

    for id in 0..FRIENDS {
        let lock_clone = lock.clone();
        let barrier_clone = barrier.clone();
```

```
        let barrier2_clone = barrier2.clone();
        friends.push(thread::spawn(move || inversor(id, lock_clone,
barrier_clone, barrier2_clone)));
    }
```

Se crea un vector de threads friends. Para cada inversor, se clonan los punteros Arc para compartir el lock y barriers. Se lanza un thread que ejecuta la función inversor().

```
    for friend in friends {
        friend.join().unwrap();
    }

}
```

Se bloquea al main() hasta que cada thread de los inversores termine.

```
fn inversor(id: u32, lock: Arc<RwLock<f64>>, barrier: Arc<Barrier>,
barrier2: Arc<Barrier>) {
    let mut semana = 0;
```

El inversor trabaja mientras haya dinero en la cuenta.

```
    while *lock.read().unwrap() > 1.0 {
```

Todos los inversores esperan juntos para empezar la semana al mismo tiempo.

```
// Espero a que todos inicien la semana
        barrier.wait();
```

Cada uno lee el saldo actual (esto es una lectura concurrente segura) y calcula su préstamo (total/#inversores).

Otros threads pueden leer también, NO escribir (solo lectura).

```
        let prestamo = *lock.read().unwrap() / FRIENDS as f64;
        println!("inversor {} inicio semana {} plata {}", id, semana,
prestamo);
```

Todos los inversores esperan a que todos hayan leído antes de tocar el dinero.

```
// Espero a que todos hayan leido el saldo disponible
        barrier2.wait();
```

Cada inversor descuenta su parte del saldo de forma segura usando write() del RwLock.

```
        // Tomo el dinero
        if let Ok(mut money_guard) = lock.write() {
            *money_guard -= prestamo;
        }
```

Simulación de la inversión y la ganancia.

```
        semana += 1;
        let mut rng = rand::thread_rng();
        let random_result: f64 = rng.gen();
        thread::sleep(Duration::from_millis((2000 as f64 *
random_result) as u64));
        let earn = prestamo * (random_result + 0.5);
        println!("inversor {} voy a devolver {}", id, earn);
```

Cada inversor devuelve su ganancia a la cuenta con una escritura exclusiva.

```
        if let Ok(mut money_guard) = lock.write() {
            *money_guard += earn;
        }

        println!("inversor {} devolví {}", id, earn);
    }
}
```

Problema del barbero

Técnicas: Semáforos para sincronizar cliente-barbero.

Una barbería tiene una sala de espera con sillas:

- Si la barbería está vacía, el barbero se pone a dormir.

- Si un cliente entra y el barbero está durmiendo, lo despierta.
- Si el barbero está atendiendo, se sienta en una de las sillas y espera su turno.
- El cliente espera a que le corten el pelo.

Defino 5 clientes en la barbería

```
fn main() {
    const N: usize = 5;
```

Creo los semáforos:
- *customer_waiting*: indica cuántos clientes están esperando.
- *barber_ready*: indica que el barbero está listo para cortar.
- *haircut_done*: indica que el corte de pelo terminó.

```
    let customer_waiting = Arc::new(Semaphore::new(0));
    let barber_ready = Arc::new(Semaphore::new(0));
    let haircut_done = Arc::new(Semaphore::new(0));
```

Asigno un único ID a cada cliente.

```
    let customer_id = Arc::new(AtomicI32::new(0));
```

Clones de los semáforos para que los use el barbero. Se crea el thread del barbero con un bucle infinito.

```
    let customer_waiting_barber = customer_waiting.clone();
    let barber_ready_barber = barber_ready.clone();
    let haircut_done_barber = haircut_done.clone();
    let barber = thread::spawn(move || loop {
```

El barbero espera con el acquire() a que un cliente lo despierte.

```
        println!("[Barbero] Esperando cliente");
        customer_waiting_barber.acquire();
```

Una vez que un cliente lo despierta, señala que está listo (release() en barber_ready) y corta el pelo simulando con un sleep de 2 segundos.

```
        barber_ready_barber.release();
        println!("[Barbero] Cortando pelo");

        thread::sleep(Duration::from_secs(2));
```

Señala que el corte terminó (release() en haircut_done)

```
        haircut_done_barber.release();
        println!("[Barbero] Terminé");
    });
```

Se crean N+1 hilos de los clientes y cada uno es un JoinHandle

```
    let customers: Vec<JoinHandle<()>> = (0..(N+1))
        .map(|_| {
```

Cada cliente tiene también un loop infinito y clones de los semáforos y del contador de IDs.

```
            let barber_ready_customer = barber_ready.clone();
            let customer_waiting_customer = customer_waiting.clone();
            let haircut_done_customer = haircut_done.clone();
            let customer_id_customer = customer_id.clone();
            thread::spawn(move || loop {
```

Simula la llegada de cada cliente en un momento aleatorio entre 2 y 10 segundos.

```
            thread::sleep(Duration::from_secs(thread_rng().gen_range(2,
10)));
```

Cada cliente con un ID único.

```
                let me = customer_id_customer.fetch_add(1,
Ordering::Relaxed);
```

Cliente llega a la barbería y avisa (release() en customer_waiting).

```
                println!("[Cliente {}] Entro a la barberia", me);
                customer_waiting_customer.release();
```

Cliente espera a que el barbero diga que está listo (acquire() en barber_ready)

```
                println!("[Cliente {}] Esperando barbero", me);
                barber_ready_customer.acquire();
```

Cliente se sienta y espera a que le terminen de cortar (acquire() en haircut_done)

```
                println!("[Cliente {}] Me siento en la silla del
barbero", me);
                println!("[Cliente {}] Esperando a que me termine de
cortar", me);
                haircut_done_customer.acquire();
```

Finalmente el cliente se va y el ciclo se repite.

```
                println!("[Cliente {}] Me terminaron de cortar", me);
            })
        })
        .collect();
```

Se hace un join() de todos los clientes

```
    let _:Vec<()> = customers.into_iter()
        .flat_map(|x| x.join())
        .collect();
```

También se hace un join() al thread del barbero.

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