Problema:

Un viejo banquero retirado se mudó a vivir al campo lejos de toda la tecnología.

Para vivir, invierte la plata que hizo durante sus años de trabajo mediante los amigos que tiene en
diversos fondos de inversión.

Al inicio de cada semana les envía por correo el dinero para invertir a sus amigos; luego espera
hasta el final de la semana a que le envíen a su buzón el resultado de esas inversiones.

Modelar la situación planteada en Rust, considerando que:
- A todos los amigos les envía el mismo monto
- Que el dinero resultante lo vuelve a invertir la próxima semana
- Que las inversiones pueden dar una ganancia entre -50% y 50% de lo invertido.

Solución 1:

Dependencias

```
extern crate rand;
use std::time::Duration;
use std::thread;
use std::thread::JoinHandle;
use rand::{Rng, thread_rng};
```

Declaro constantes de la cantidad de inversores y el saldo inicial.

```
const INVERSORES: i32 = 5;
const SALDO_INICIAL: f64 = 100000.0;
```

Tengo un loop donde cada semana, el banquero divide el saldo por la cantidad de inversiones.

```
fn main() {
```

```
    let mut saldo = SALDO_INICIAL;
    let mut semana = 1;

    loop {
        println!("[BANQUERO] semana {}, tengo saldo {}", semana, saldo);
        let mut inversores = vec![];
        let saldo_individual = saldo / (INVERSORES as f64);
        saldo = 0.0;
```

Por cada uno de los inversores, genero un thread donde le aplico la función inversor.

- Primer for:
  Crea y lanza los threads (inversores). Los threads arrancan a trabajar en paralelo.
- Segundo for:
  Espera (join) a que cada thread termine. Suma el dinero devuelto a la cuenta del banquero.

```
        for id in 0..INVERSORES {
            inversores.push(thread::spawn(move || inversor(id,
saldo_individual)))
        }

        // Para pensar: Por qué dos fors?
        for inversor in inversores {
            match inversor.join() {
                Ok(plata) => saldo += plata,
                Err(str) => panic!(str)
            }
        }

        semana += 1
    }
}
```

Esta función:

- Imprime cuánto recibió el inversor.
- Genera un número aleatorio r entre 0.0 y 1.0.
- Duerme al thread entre 0 y 1000ms para simular demora.
- Calcula el resultado de la inversión: prestamo * (0.5 + r) → entre 50% y 150% del capital inicial.
- Imprime cuánto devuelve.
- Retorna el resultado al hilo principal (main).

```rust
fn inversor(id: i32, prestamo: f64) -> f64 {
    println!("[INVERSOR {}] tengo prestamo {}", id, prestamo)
    let random_result: f64 = rand::thread_rng().gen();
    thread::sleep(Duration::from_millis((random_result * 1000.0) as
u64));

    let result = prestamo * (random_result + 0.5);

    println!("[INVESOR {}] devuelvo {}", id, result);
    result
}
```

Solución 2: al estilo funcional

Usa `map(...).collect()` para crear los threads.

Usa `into_iter().flat_map(...).sum()` para esperar los threads y sumar los resultados en una sola expresión.

```rust
     let inversores: Vec<JoinHandle<f64>> = (0..INVERSORES)
            .map(|id| thread::spawn(move || inversor(id, saldo /
(INVERSORES as f64))))
            .collect();

        saldo = inversores.into_iter()
            .flat_map(|x| x.join())
            .sum();

        semana += 1
    }
}
```

Problema 2:

Al tiempo el señor fallece.
Los hijos deciden que los inversores sigan trabajando el dinero con algunas
condiciones:
- Ellos no se hacen cargo de nada, los inversores solos toman dinero de la cuenta y lo devuelven al final de la semana

- Cada inversor puede reinvertir el capital y hasta el 50% de la ganancia propia de la semana anterior, o bien todo el capital en caso de haber perdido.
- Las inversiones deberán ser menos riesgosas, pudiendo dejar de -10% a +10%.

Solución:

`cuenta` es el saldo total disponible para todos.
Es un `RwLock` envuelto en un `Arc` para que pueda ser compartido de forma segura entre threads.

```
fn main() {
    let cuenta = Arc::new(RwLock::new(SALDO_INICIAL));
```

Se crean los threads inversores. Cada inversor comienza con un saldo inicial y cada thread ejecuta la función inversor. Se espera a que los inversores terminen con el join()

```
    let inversores: Vec<JoinHandle<()>> = (0..INVERSORES)
        .map(|id| {
            let cuenta_local = cuenta.clone();
            thread::spawn(move || inversor(id, SALDO_INICIAL /
(INVERSORES as f64), cuenta_local))
        })
        .collect();

    inversores.into_iter()
        .flat_map(|x| x.join())
        .for_each(drop)

}
```

```
fn inversor(id:u32, inicial:f64, cuenta:Arc<RwLock<f64>>) {
    let mut capital = inicial;
    while capital > 5.0 {
        println!("[INVERSOR {}] inicio semana {}", id, capital);
```

El inversor toma su capital de la cuenta y como escribe sobre el recurso compartido usa `write()` para obtener acceso exclusivo.

```
        if let Ok(mut saldo) = cuenta.write() {
            *saldo -= capital;
        }
```

Se simula el tiempo de inversión

```
        thread::sleep(Duration::from_millis(1000));
```

Se calcula la ganancia o pérdida

```
        let resultado = capital * thread_rng().gen_range(0.9, 1.1);
```

Se devuelve el resultado a la cuenta común, de vuelta `write()` porque se modifica el valor

```
        if let Ok(mut money_guard) = cuenta.write() {
            *money_guard += resultado;
        }
        println!("[INVERSOR {}] resultado {}", id, resultado);
        if (resultado > capital) {
            capital += (resultado - capital) * 0.5;
        } else {
            capital = resultado;
        }
    }

}
```