Problema del lector-escritor

Un estado se comparte entre varios procesos.

- Algunos procesos necesitan actualizar dicho estado, mientras que otros solo necesitan leerlo.
- Mientras que un proceso está leyendo el estado, otros pueden leerlo, pero ninguno modificarlo.
- Mientras que un proceso está modificando el estado, ningún otro puede leerlo ni modificarlo.

Process 1	Process 2	Allowed / Not Allowed
Writing	Writing	Not Allowed
Reading	Writing	Not Allowed
Writing	Reading	Not Allowed
Reading	Reading	Allowed

Versión simple

Desventaja:

- Starvation de escritores: si los lectores siguen llegando todos el tiempo, nunca dejan espacio a los escritores. No hay control de turno.

ReadWrite guarda el estado compartido:
- readers: cuántos lectores están leyendo actualmente.
- writing: si hay un escritor escribiendo ahora.

```
#[derive(Debug)]
struct ReadWrite {
    readers: i32,
    writing: bool
}
```

Guarda el dato real al que los lectores y escritores acceden.

```
struct DataHolder {
    data: UnsafeCell<i32>
}
unsafe impl Sync for DataHolder {}
```

5 lectores y 2 escritores.

```
fn main() {
    const READERS: i32 = 5;
    const WRITERS: i32 = 2;
```

Inicializar estado compartido:

- pair que contiene el Mutex que protege el estado ReadWrite y la Condvar.
- data: guarda el dato entero inicializado en 42.

```
    let pair = Arc::new((Mutex::new(ReadWrite { readers: 0, writing:
false }), Condvar::new()));
    let data = Arc::new(DataHolder { data: UnsafeCell::new(42) } );
```

Threads de los lectores, cada lector tiene un clone del Arc para compartir pair y data.

```
    let readers: Vec<JoinHandle<()>> = (0..READERS)
        .map(|me| {
            let pair_reader = pair.clone();
            let data_reader = data.clone();

            thread::spawn(move || loop {
```

Se obienen los valores del Mutex y la variable de condición.

```
                let (lock, cvar) = &*pair_reader;
```

Espera mientras haya un escritor escribiendo (state_writing == true). Cuando puede entrar, incrementa readers.

```
                // Sacar esto para llegar a starvation del writer
                //
```

```
thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                {
                    let mut _guard =
cvar.wait_while(lock.lock().unwrap(), |state| {
                        println!("[Lector {}] Chequeando {:?}", me,
state);
                        state.writing
                    }).unwrap();
                    _guard.readers += 1;
                }
```

Lee el dato y simula el tiempo de lectura.

```
                unsafe {
                    println!("[Lector {:?}] Leyendo {}", me,
data_reader.data.get().read());
                }

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                println!("[Lector {:?}] Terminé", me);
```

Terminar de leer.

```
                lock.lock().unwrap().readers -= 1;
                cvar.notify_all();
            })
        })
        .collect();
```

Threads de los escritores, clona las referencias compartidas.

```
    let writers: Vec<JoinHandle<()>> = (0..WRITERS)
        .map(|me| {
            let pair_writer = pair.clone();
            let data_writer = data.clone();

            thread::spawn(move || loop {
```

Se obienen los valores del Mutex y la variable de condición.

```
                let (lock, cvar) = &*pair_writer;
```

Espera para escribir mientras hay otro escritor escribiendo (state_writing == true) o hay lectores activos (state.readers > 0). Cuando entra marca writing = true.

```
                let mut _guard = cvar.wait_while(lock.lock().unwrap(),
|state| {
                        println!("[Escritor {}] Chequeando {:?}", me,
state);
                        state.writing || state.readers > 0
                    }).unwrap();
                    _guard.writing = true;
                }
```

Escribir datos.

```
                unsafe {
                    println!("[Escritor {:?}] Escribiendo", me);
                    data_writer.data.get().write(me);
                }

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                println!("[Escritor {:?}] Terminé", me);
```

Marca que ya no está escribiendo, notifica a los demás threads.

```
                lock.lock().unwrap().writing = false;
                cvar.notify_all();
            })
        })
        .collect();
```

Deja que todos terminen.

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

}
```

Versión con prioridad a escritores para evitar starvation

Desventaja: starvation en lectores.

Ahora nuevo campo en ReadWrite: writer que cuenta cuántos escritores están esperando para entrar.

```
#[derive(Debug)]
struct ReadWrite {
    readers: i32,
    writing: bool,
    writers: i32,
}
```

```
struct DataHolder {
    data: UnsafeCell<i32>
}
unsafe impl Sync for DataHolder {}


fn main() {
    const READERS: i32 = 5;
    const WRITERS: i32 = 2;
```

```
    let pair = Arc::new((Mutex::new(ReadWrite { readers: 0, writing:
false, writers: 0 }), Condvar::new()));
    let data = Arc::new(DataHolder { data: UnsafeCell::new(42) } );
```

Threads de los lectores.

```
    let readers: Vec<JoinHandle<()>> = (0..READERS)
        .map(|me| {
            let pair_reader = pair.clone();
            let data_reader = data.clone();

            thread::spawn(move || loop {
```

Acceso al estado compartido.

```
                let (lock, cvar) = &*pair_reader;
```

El lector espera mientras hay un escritor escribiendo (writing == true) o hay escritores esperando (writers > 0)
Esto significa que si hay escritores esperando, los lectores nuevos no pueden entrar. Cuando puede entrar, incrementa readers.

```
  {
                    let mut _guard =
cvar.wait_while(lock.lock().unwrap(), |state| {
                        println!("[Lector {}] Chequeando {:?}", me,
state);
                        state.writing || state.writers > 0
                    }).unwrap();
                    _guard.readers += 1;
                }

                unsafe {
                    println!("[Lector {:?}] Leyendo {}", me,
data_reader.data.get().read());
                }

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                println!("[Lector {:?}] Terminé", me);
```

Decrementa el contador de lectores y notifica a otros hilos.

```
                lock.lock().unwrap().readers -= 1;
                cvar.notify_all();
            })
        })
        .collect();
```

Threads de escritores, acceso al estado compartido.

```
   let writers: Vec<JoinHandle<()>> = (0..WRITERS)
        .map(|me| {
            let pair_writer = pair.clone();
            let data_writer = data.clone();

            thread::spawn(move || loop {
                let (lock, cvar) = &*pair_writer;
```

Duerme un tiempo aleatorio, incrementa writers para indicar que hay un escritor esperando.

```
                // Sacar esto para llegar a starvation del reader

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                lock.lock().unwrap().writers += 1;
```

Espera para escribir mientras hay otro escritor escribiendo o hay lectores leyendo. Cuando entra writing = true.

```
                {
                    let mut _guard =
cvar.wait_while(lock.lock().unwrap(), |state| {
                        println!("[Escritor {}] Chequeando {:?}", me,
state);
                        state.writing || state.readers > 0
                    }).unwrap();
                    _guard.writing = true;
                }

                unsafe {
                    println!("[Escritor {:?}] Escribiendo", me);
                    data_writer.data.get().write(me);
                }

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
                println!("[Escritor {:?}] Terminé", me);
```

Sale y notifica.

```
                let mut state = lock.lock().unwrap();
                state.writing = false;
                state.writers -= 1;
                cvar.notify_all();
            })
        })
        .collect();
```

Espera a que terminen todos los threads.

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

```
}
```

Versión 3: elección justa de escritores y lectores.

Se agrega a ReadWrite:

- queue: ID del proceso que ahora tiene derecho a entrar (puede ser lector o escritor).
- next: ID que se asignará al próximo proceso que quiere entrar.

Se crea una cola de turno, garantizando un orden de llegada.

```
#[derive(Debug)]
struct ReadWrite {
    readers: i32,
    writing: bool,
    queue: u32,
    next: u32,
}
```

Dentro de cada lector (thread):

- Toma un ID igual a next.
- Incrementa next para el próximo.

```
let (lock, cvar) = &*pair_reader;
let mut _guard = lock.lock().unwrap();
let id = _guard.next;
_guard.next += 1;
```

El lector espera si hay un escritor escribiendo (state_writing == true) o **no es su turno (id != state.queue).**
Cuando puede entrar incrementa readers y avanza la queue para el siguiente proceso.

```
let mut _guard = cvar.wait_while(_guard, |state| {
    println!("[Lector {}] Chequeando {:?}", id, state);
    state.writing || id != state.queue
}).unwrap();
_guard.readers += 1;
_guard.queue += 1;
```

Dentro de cada escritor (thread):

- Toma su ID igual a next.
- Incrementa next.

```
let (lock, cvar) = &*pair_writer;
let mut _guard = lock.lock().unwrap();
let id = _guard.next;
_guard.next += 1;
```

Espera para escribir si hay un escritor escribiendo, hay un lector leyendo o no es su turno.
Cuando termina de escribir, marca que terminó y avanza la queue para el siguiente proceso.

```
let mut _guard = cvar.wait_while(_guard, |state| {
    println!("[Escritor {}] Chequeando {:?}", id, state);
    state.writing || state.readers > 0 || id != state.queue
}).unwrap();
_guard.writing = true;
let mut state = lock.lock().unwrap();
state.writing = false;
state.queue += 1;
```