Problema de los filósofos

5 filósofos se sientan alrededor de una mesa. Cada uno necesita dos palitos chinos para comer. Solo puede agarrar los palitos que tiene a su izquierda y derecha.
Si un filósofo agarra un palito pero no puede agarrar el segundo, se queda esperando (y puede haber deadlock si todos hacen lo mismo).

Declaro 5 filósofos (5 palitos).

```
const N:usize = 5;
```

Vector de semáforos, uno por cada palito. Semaphore::new(1) indica que el palito está disponible inicialmente. Dentro de un Arc para que todos los hilos puedan usar el mismo vector.

```
fn main() {
    let chopsticks:Arc<Vec<Semaphore>> = Arc::new((0 .. N)
        .map(|_| Semaphore::new(1))
        .collect());
```

Threads de los filósofos, cada uno llama a la función philosopher(id, chopsticks_local).

```
    let philosophers:Vec<JoinHandle<()>> = (0 .. N)
        .map(|id| {
            let chopsticks_local = chopsticks.clone();
            thread::spawn(move || philosopher(id, chopsticks_local))
        })
        .collect();
```

Después de crear todos los hilos, se hace join() de cada uno. Espera a que todos terminen.

```
    for philosopher in philosophers {
        philosopher.join();
    }

}
```

Comportamiento de cada filósofo. Calcula el siguiente filósofo a la derecha con una lista circular (4 → 0). Se definen 2 palitos:

- first_chopstick: primero que intenta agarrar
- second_chopstick: segundo que intenta agarrar

```
fn philosopher(id: usize, chopsticks: Arc<Vec<Semaphore>>) {
    let next = (id + 1) % N;
    let first_chopstick;
    let second_chopstick;
```

¿Qué palito agarran primero? Todos agarran el palito izquierdo y luego el derecho.

Solución al deadlock comentada:

- El último filósofo debería agarrar al revés para romper el deadlock.

```
    // solucion al deadlock
    //if id == (N-1) {
    //    first_chopstick = &chopsticks[next];
    //    second_chopstick = &chopsticks[id];
    //} else {
       first_chopstick = &chopsticks[id];
       second_chopstick = &chopsticks[next];
    //}
```

Cada fiósofo espera un poco antes de empezar.

```
    // tratar de forzar tomar en el primer palito en el orden de id
    thread::sleep(Duration::from_millis(100 * id as u64));
```

Loop infinito: pensar, agarrar, comer.

```
    loop {
        println!("filosofo {} pensando", id);
```

Intenta agarrar su primer palito (access() hace acquire() sobre el semáforo para bloquearlo si está ocupado).

```
        println!("filosofo {} esperando palito izquierdo", id);
```

```
        {
            let first_access = first_chopstick.access();
```

Luego de agarrar el primer palito, espera 1 segundo.

```
            // pausa despues del primer palito para forzar el deadlock
            thread::sleep(Duration::from_millis(1000));
```

Luego intenta agarrar su segundo palito, si está ocupado, se queda bloqueado esperando.

```
            println!("filosofo {} esperando palito derecho", id);
            {
                let second_access = second_chopstick.access();
```

Cuando tiene los 2 palitos, come.

```
                println!("filosofo {} comiendo", id);

thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
            }
        }
    }
}
```

Problema de los fumadores

Consideremos que para fumar un cigarrillo se necesitan 3 ingredientes: tabaco, papel y fósforos. Hay 3 fumadores alrededor de una mesa. Cada uno tiene una cantidad infinita de uno solo de los ingredientes y necesita los otros dos para fumar. Cada fumador posee un ingrediente distinto. Existe además un agente que pone aleatoriamente dos ingredientes en la mesa. El fumador que los necesita los tomará para hacer su cigarrillo y fumará un rato. Cuando el fumador termina, el agente pone otros dos ingredientes.

Son 3 fumadores.

```
const N:usize = 3;
```

Enum que representa a los ingredientes con un número para cada uno.

```rust
#[derive(Clone, Copy, Debug, FromPrimitive)]
enum Ingredients {
    Tobacco = 0,
    Paper,
    Fire
}

const ALL_INGREDIENTS: [Ingredients; N] = [Ingredients::Tobacco,
Ingredients::Paper, Ingredients::Fire];
```

Inicializar semáforos:

- agent_sem: para controlar cuándo el agente puede poner nuevos ingredientes. Inicia en 1 para comenzar inmediatamente.
- ingredient_sems: vector de 3 semáforos, uno por ingrediente. Cada uno empieza en 0, no hay ingredientes en la mesa al inicio.

```rust
fn main() {

    let agent_sem = Arc::new(Semaphore::new(1));
    let ingredient_sems: Arc<Vec<Semaphore>> = Arc::new((0..N)
                                        .map(|_| Semaphore::new(0))
                                        .collect());
```

Clones para el agente.

```rust
    let agent_sem_a = agent_sem.clone();
    let ingredients_sem_a = ingredient_sems.clone();
```

Thread del agente.

```rust
    let agent = thread::spawn(move || loop {
```

El agente espera su semáforo y actúa sólo cuando su semáforo está disponible.

```rust
        println!("[Agente] Esperando sem");
        agent_sem_a.acquire();
```

Se copian todos los ingredientes, se mezclan aleatoriamente y se eligen los primeros 2.

```
        let mut ings = ALL_INGREDIENTS.to_vec();
        ings.shuffle(&mut thread_rng());
        let selected_ings = &ings[0..N-1];
```

Por cada ingrediente seleccionado:

- Se libera el semáforo correspondiente (release()).

De esta forma, el ingrediente está disponible en la mesa.

```
        for ing in selected_ings {
            println!("[Agente] Pongo {:?}", ing);
            ingredients_sem_a[*ing as usize].release();
        }
    });
```

Crear los threads de los fumadores.

```
    let smokers:Vec<JoinHandle<()>> =  (0..N)
        .map(|i|  {
            let agent_sem_smoker = agent_sem.clone();
            let ingredient_sems_smoker = ingredient_sems.clone();
            thread::spawn(move || loop {
```

Cada fumador convierte su número a un Ingredients, para saber cuál ingrediente propio tiene.

```
                let me = Ingredients::from_usize(i).unwrap();
```

Recorre todos los ingredientes e ignora el que ya tiene.

```
                for ing_id in 0..N {
                    if ing_id != i {
```

Por cada ingrediente que necesita espera (acquire()) hasta que ese ingrediente está en la mesa.

```
                        let ing =
Ingredients::from_usize(ing_id).unwrap();
                        println!("[Fumador {:?}] Esperando {:?}", me,
ing);
                        ingredient_sems_smoker[ing_id].acquire();
                        println!("[Fumador {:?}] Obtuve {:?}", me, ing);
                    }
                }
```

Cuando consigue los dos ingredientes que necesita, fuma y libera el semáforo del agente (release() en agent_sem) avisando que ya puede poner más ingredientes.

```
                println!("[Fumador {:?}] Fumando", me);
                 thread::sleep(Duration::from_secs(2));
                 agent_sem_smoker.release();
                 println!("[Fumador {:?}] Terminé", me);
            })
        })
        .collect();
```

Espera a que todos los threads terminen.

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

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

Smoker with Tobacco	Smoker with Paper	Smoker with Match

sem_wait(match_sem);
SUCCESS
sem_wait(paper_sem);
BLOCKED
	sem_wait(tobacco_sem);
	SUCCESS
	sem_wait(match_sem);
	BLOCKED
		sem_wait(paper_sem);
		BLOCKED
		sem_wait(tobacco_sem);
		BLOCKED

Versión 2: con condvar

`pair` contiene un Mutex que protege un array donde cada posición representa si hay tabaco, papel o fósforos en la mesa. Además, tiene una **condvar (variable de condición)** para que los threads esperen a

que cambie el estado. Todo envuelto en un Arc para compartir entre threads.

```
   let pair = Arc::new((Mutex::new([false, false, false]),
Condvar::new()));
```

Crear el thread del agente.

```
    let pair_agent = pair.clone();

    let agent = thread::spawn(move || loop {
```

El agente se bloquea si hay ingredientes en la mesa y despierta cuando la mesa está completamente vacía. En el wait_while mientras la condición full_table == true se cumple, el agente espera. Cuando todos son false, el agente puede seguir.

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

        println!("[Agente] Esperando a que fumen");
        let mut state = cvar.wait_while(lock.lock().unwrap(), |ings| {
            let full_table = (*ings).iter().any(|i| *i);
            println!("[Agente] Esperando a que fumen {:?} - {}", ings,
full_table);
            full_table
        }).unwrap();
```

El agente pone 2 ingredientes de forma aleatoria.

```
        let mut ings = vec!(Ingredients::Tobacco, Ingredients::Paper,
Ingredients::Fire);
        ings.shuffle(&mut thread_rng());
        let selected_ings = &ings[0..N-1];
```

Por cada ingrediente seleccionado, lo pone como true en el array.

```
        for ing in selected_ings {
            println!("[Agente] Pongo {:?}", ing);
            state[*ing as usize] = true;
        }
```

El agente despierta a todos los fumadores para que chequeen si pueden fumar.

```
        cvar.notify_all();
    });
```

Threads de cada fumador.

```
    let smokers:Vec<JoinHandle<()>> =  (0..N)
        .map(|fumador_id|  {
            let pair_smoker = pair.clone();
            let me = Ingredients::from_usize(fumador_id).unwrap();

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

Cada fumador se bloquea mientras no sea su turno. Es su turno cuando: todos los ingredientes excepto el suyo estén disponibles, si falta alguno que necesita sigue esperando.

```
                let mut _guard = cvar.wait_while(lock.lock().unwrap(),
|ings| {
                    let my_turn = (0..N).all(|j| j == fumador_id ||
ings[j]);
                    println!("[Fumador {:?}] Chequeando {:?} - {}", me,
ings, my_turn);
                    !my_turn
                }).unwrap();
```

Fumador fuma.

```
                println!("[Fumador {:?}] Fumando", me);
                thread::sleep(Duration::from_secs(2));
```

Limpia todos los ingredientes de la mesa y notifica a todos los otros threads (también el del agente) que puede actuar de nuevo.

```
                for ing in (*_guard).iter_mut() {
                    *ing = false;
                }
                println!("[Fumador {:?}] Terminé", me);
```

```
                cvar.notify_all();
            })
        })
        .collect();
```