



Técnicas de Programación Concurrente

Ejercicios de Parcial

Modelos de Concurrencia

- Estado Mutable Compartido
- Paralelismo con Fork Join (caso especial Vectorización)
- Programación asincrónica
- Pasaje de mensajes (Canales)
- Actores

1º Cuatrimestre 2024

1. Revisando el diseño aplicado en algunos proyectos, se encontró el uso de las siguientes herramientas para resolver problemas de concurrencia. Para cada uno de los problemas enuncie ventajas o desventajas de utilizar la solución propuesta y menciona cual utilizaría usted.

- Renderizado de videos 3D en alta resolución, utilizando programación asincrónica.

La programación asincrónica nos es útil cuando tenemos que procesar muchas tareas de cómputo liviano (como las operaciones de I/O).

En este caso, tenemos una tarea de cómputo pesado, por lo que la programación asincrónica no es la mejor opción.

Para esta tarea convendría usar **vectorización, que es un caso particular del Fork Join**. Utilizando esta técnica se puede paralelizar lo más posible al cómputo y aprovechar el hardware al máximo.

- Aplicación que arma una nube de palabras a partir de la API de Twitter, utilizando barriers y mutex.

¿Tiene sentido utilizar el modelo de concurrencia de estado mutable compartido para este caso? No.

¿Qué hacen los barriers?

Permiten sincronizar varios threads para que se esperen entre sí hasta que todos lleguen a cierto punto. Esto es ineficiente para APIs ya que cada request puede demorarse tiempos variables.

¿Qué hacen los mutex?

Evitan que múltiples threads accedan al mismo tiempo a la sección crítica (recurso compartido).

Siguiendo este modelo de programación estamos más expuestos a tener condiciones de carrera, deadlocks que interferirían en el correcto funcionamiento de nuestra aplicación. Con estas herramientas garantizo el acceso seguro a los recursos a costa del rendimiento.

Lo correcto sería usar **programación asincrónica**, utilizando tareas asíncronas para cada request, que son operaciones de I/O. Estas tareas son muy livianas y evitan bloqueos innecesarios.

- Una aplicación para realizar una votación en vivo para un concurso de televisión, optimizada con Vectorización.

La vectorización permite que el procesador haga la misma operación sobre muchos datos al mismo tiempo, acelerando las tareas computacionales intensivas. ¿Me sirve para este caso? No.

La vectorización no resuelve problemas de concurrencia, sino que acelera los cálculos numéricos.

Como estamos trabajando con recursos compartidos que forman parte de la sección crítica del sistema de votación, debemos usar el modelo de **estado**

mutable compartido, de esta forma podemos asegurar la sincronización del conteo de los votos en tiempo real.

2. Programación asincrónica. Elija verdadero o falso y explique brevemente por qué:

- El encargado de hacer poll es el thread principal del programa. ❌

Esto es importante: el que ejecuta Poll es el Executor del runtime asincrónico y puede estar en cualquier hilo. El Executor es una abstracción que, por ejemplo, me ofrece Tokio en Rust para poder manejar las tareas asincrónicas.

```
async fn tarea() → i64 {  
    19  
}  
  
let future = tarea(); //No ejecuta, solo crea el Future  
tokio::spawn(future); //Acá está el executor
```

- El método poll es llamado únicamente cuando la función puede progresar. ❌

El Poll se llama cada vez que el executor lo considera necesario, por ejemplo cuando hay una notificación de que un recurso podría estar disponible.

La llamada a Poll no garantiza que la función pueda progresar, ya que puede devolver Pending, indicando que aún no está listo.

- El modelo piñata es colaborativo. ✅

Modelo colaborativo

Las tareas deciden cuándo ceder el control y no son interrumpidas de forma forzada.

El sistema cambia de tarea cuando una tarea dice: "Estoy esperando,

Modelo preventivo

El sistema puede interrumpir cualquier tarea en cualquier momento.

Es más robusto, pero requiere más sincronización.

podés pasar a otra”.

El modelo piñata o Poll es colaborativo porque las tareas usan `await` para suspenderse voluntariamente, permitiendo que el Executor haga Poll a otras tareas listas para ejecutarse.

“Las tareas cooperan para que el sistema sea eficiente”

- La operación asincrónica inicia cuando se llama a un método declarado con `async`. ❌

Cuando se llama una función `async` no se ejecuta de forma inmediata. Retorna inmediatamente un `Future` que encapsula la operación pendiente. La ejecución comienza cuando el Executor hace Poll sobre el `Future`.

3. Para cada uno de los siguientes fragmentos de código indique si es o no es un busy wait. Justifique en cada caso (Nota: `mineral` y `batteries_produced` son locks).

Recordemos: ¿Qué es un busy wait? Es una espera activa, pregunto activamente si un recurso está disponible para poder continuar.


1º caso: ❌ No es busy wait.

```
for _ in 0..MINERS {
  let lithium = Arc::clone(&mineral);
  thread::spawn(move || loop {
    let mined = rand::thread_rng().gen();
    let random_result: f64 = rand::thread_rng().gen();
    *lithium.write().expect("failed to mine") += mined;
    thread::sleep(Duration::from_millis((5000 as f64 * random_result) as u64))
  })
}
```

¿Qué hace el código?

- Tenemos threads para los `MINERS` que simulan una operación sobre un recurso compartido `lithium`.

- Se hace un `write()` que espera a que el recurso esté libre para poder sumar el valor del resultado.
- Luego el thread hace un `sleep`, liberando el CPU. No ocupa el procesador mientras espera.

2º caso:  Sí es busy wait

```
for _ in 0..MINERS {
    let lithium = Arc::clone(&mineral);
    let batteries_produced = Arc::clone(&resources);
    thread::spawn(move || loop {
        let mut lithium = lithium.write().expect("failed");
        if lithium >= 100 {
            lithium -= 100;
            batteries_produced.write().expect("failed to produce") += 1
        }
        thread::sleep(Duration::from_millis(500));
    })
}
```

¿Qué hace el código?

- Nuevamente thread para cada MINER.
- Dos recursos compartidos: `lithium` y `batteries_produced`.
- Se hace un `write()` esperando a que `lithium` esté libre para: preguntar por su valor y hacer una operación sobre el mismo. El thread queda bloqueado hasta que el recurso esté disponible y mientras tanto no hace nada. Eso es un busy wait.
- El `sleep` que se encuentra después es de 500 mS, tiempo bastante corto. Si `lithium` no cambia en mucho tiempo, estoy intentando al pepe cada 500 mS.

4. Dada la siguiente estructura, nombre si conoce una estructura de sincronización con el mismo comportamiento. Indique posibles errores en la implementación.

```
pub struct SynchronizationStruct {
    mutex: Mutex<i32>,
```

```

    cond_var: Condvar,
}

impl SynchronizationStruct {

    pub fn new(size: u16) → SynchronizationStruct {
        SynchronizationStruct {
            mutex: Mutex::new(size),
            cond_var: Condvar::new(),
        }
    }

    pub fn function_1(&self) {
        let mut amount = self.mutex.lock().unwrap();
        if *amount <= 0 {
            amount = self.cond_var.wait(amount).unwrap();
        }
        *amount -= 1;
    }

    pub fn function_2(&self) {
        let mut amount = self.mutex.lock().unwrap();
        *amount += 1;
        self.cond_var.notify_all();
    }
}

```

La estructura implementada es un semáforo construido por monitores. El cual tiene 2 estados internos:

- V: entero no negativo: contador de recursos disponibles. Inicialmente comienza con un valor mayor a cero.
- L: conjunto de procesos esperando. Inicia vacío.

wait() "obtener acceso"

Le resta 1 al contador

Si $V > 0 \rightarrow V -= 1$

Si $V \leq 0 \rightarrow$ El proceso se bloquea y se agrega a L

signal() "liberar"

Si L está vacío $\rightarrow V += 1$

Sino, se remueve un proceso de L y se lo pone en estado Ready (no se define cuál)

Entonces, ¿qué problema tiene el código? Veamos las funciones y las implementaciones de las mismas.

La `function_1` es el `wait()`

```
pub fn function_1(&self) {
    let mut amount = self.mutex.lock().unwrap();
    if *amount <= 0 {
        amount = self.cond_var.wait(amount).unwrap();
    }
    *amount -= 1;
}
```

Acá hay un gran problema, el `if *amount <= 0`. Usando esa lógica, nuestra función `wait()` funcionaría de la siguiente manera:

- Espera una vez si $\text{amount} \leq 0$. Pero el método `Condvar::wait()` en Rust puede retornar incluso si nadie hizo `notify()` (spurious wakeup).
- Si se despierta cuando `amount` sigue siendo 0, el código sigue y hace `*amount -= 1`, lo cual rompe la invariante del semáforo (que debe mantenerse en $\text{amount} \geq 0$).

¿Cómo se arregla? cambiando el `if` por un `while`.

```
pub fn function_1(&self) {
    let mut amount = self.mutex.lock().unwrap();
    while *amount <= 0 {
        amount = self.cond_var.wait(amount).unwrap();
    }
    *amount -= 1;
}
```

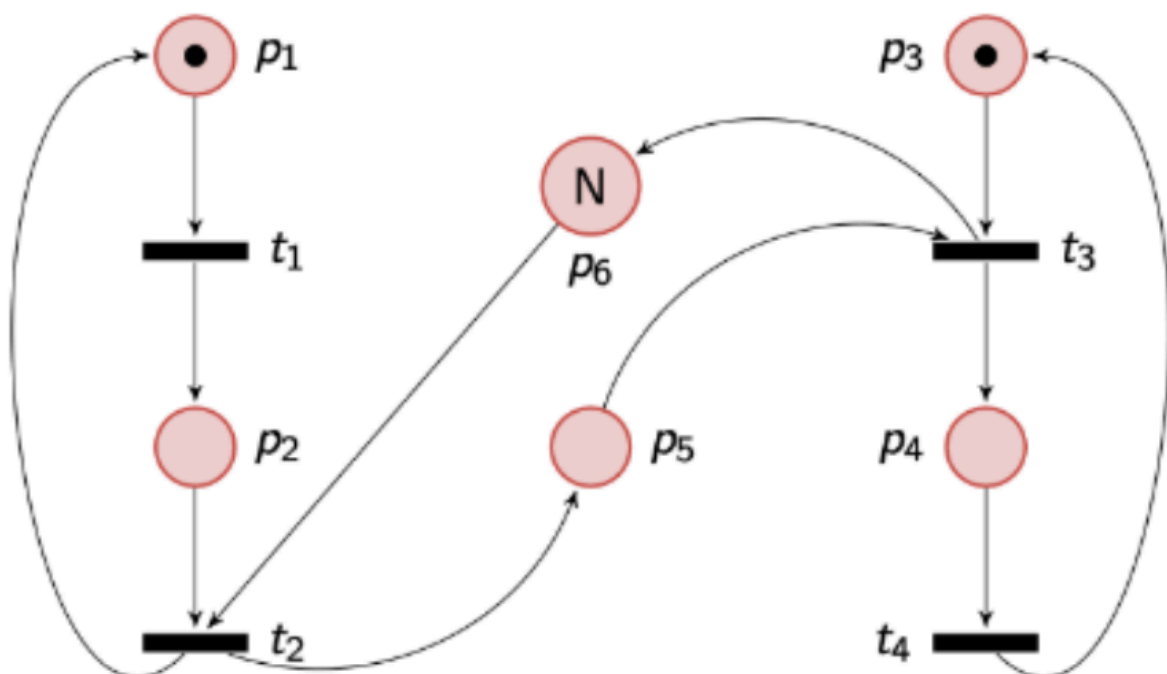
De esta forma, se pregunta cada vez que el thread despierta si el `amount` ≥ 0 o no.

La `function_2` es `signal()`

```
pub fn function_2(&self) {  
    let mut amount = self.mutex.lock().unwrap();  
    *amount += 1;  
    self.cond_var.notify_all();  
}  
}
```

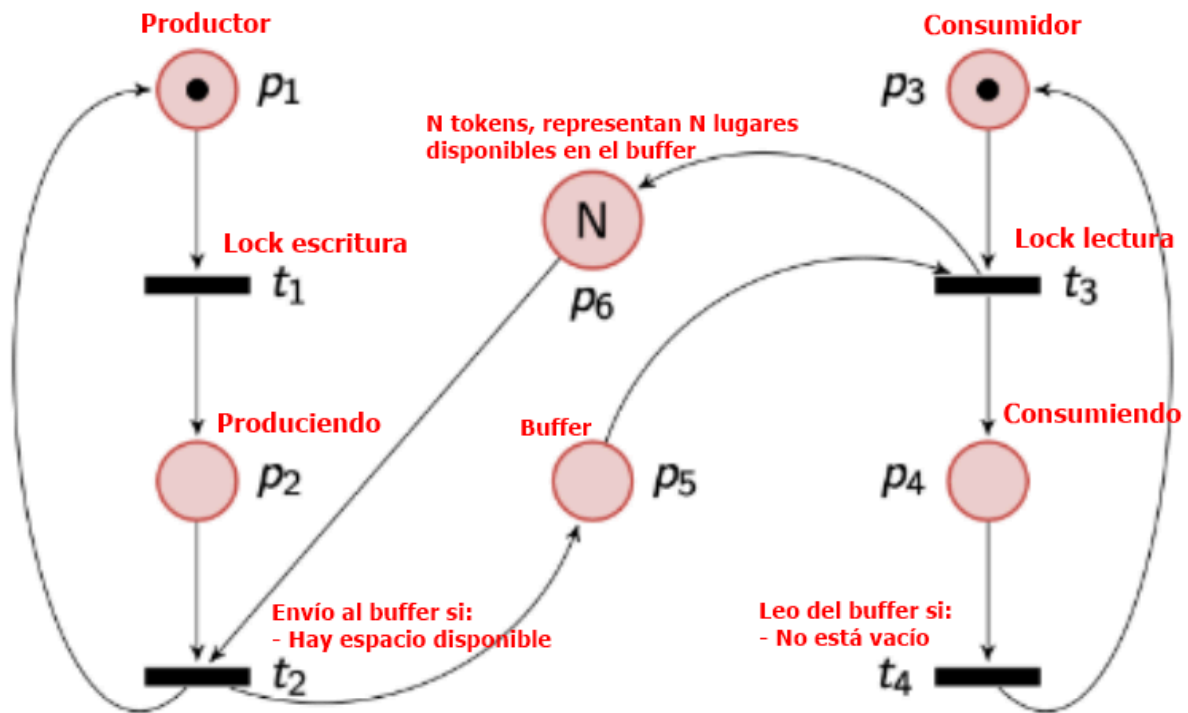
Acá solamente un pequeño detalle: el método `notify_all()`. No tiene sentido despertar a todos los threads, ya que uno solo será el que tome el control del recurso, puede usarse `notify_one()`.

5. Dados la siguiente red de Petri y fragmento de código, indique el nombre del problema que modelan. Indique si la implementación es correcta o describa cómo mejorarla.



Problema: Productor-Consumidor con buffer finito.

¿Cómo funciona la Red de Petri de este problema?



Veamos el código que nos proponen:

```
fn main() {
    let sem = Arc::new(Semaphore::new(0));
    let buffer = Arc::new(Mutex::new(Vec::with_capacity(N)));
    let sem_cloned = Arc::clone(&sem);
    let buf_cloned = Arc::clone(&buffer);
    let t1 = thread::spawn(move || {
        loop {
            // heavy computation
            let random_result: f64 = rand::thread_rng().gen();
            thread::sleep(Duration::from_millis((500 as f64 *
                random_result) as u64));
            buf_cloned.lock().expect("").push(random_result);
            sem_cloned.release()
        }
    });
    let sem_cloned = Arc::clone(&sem);
```

```

let buf_cloned = Arc::clone(&buffer);
let t2 = thread::spawn(move || {
    loop {
        sem_cloned.acquire();
        println!("{}", buf_cloned.lock().expect("").pop());
    }
});
t1.join().unwrap();
t2.join().unwrap();
}

```

Este código serviría para el problema Productor-Consumidor con buffer infinito. Ya que nunca se pregunta por el espacio de tamaño N del buffer. Para poder convertir el problema a buffer finito, debemos agregar otro semáforo más que indique si el buffer está lleno o no.

Productor

```

notFull.acquire()
producir()
lock buffer
push(producto)
notEmpty.release()

```

Consumidor

```

notEmpty.acquire()
lock buffer
consumir()
notFull.release()

```