PROGRAMACION CONCURRENTE

Introducción

Programa: conjunto de datos, asignaciones e instrucciones de control de flujo que compilan a instrucciones de máquina, las cuales se ejecutan secuencialmente en un procesador y acceden a datos almacenados en memoria principal o memorias secundarias.

Programa concurrente: conjunto finito de programas secuenciales que pueden ejecutarse en paralelo.

Proceso: cada uno de los programas secuenciales que conforman el programa concurrente. Están compuestos por un conjunto finito de instrucciones atómicas.

Ejecución del programa concurrente: resulta al ejecutar una secuencia de instrucciones atómicas que se obtiene de intercalar arbitrariamente las instrucciones atómicas de los procesos que lo componen.

Sistema paralelo: sistema compuesto por varios programas que se ejecutan simultáneamente en procesadores distintos.

Multitasking: ejecución de múltiples procesos concurrentemente en un cierto periodo de tiempo. El scheduler, parte del kernel del sistema operativo, se encarga de coordinar el acceso a los procesadores.

Multithreading: construcción provista por algunos lenguajes de programación que permite la ejecución concurrente de threads dentro del mismo programa.

Sincronización: coordinación temporal entre distintos procesos.

Comunicación[1]**:** datos que necesitan compartir los procesos para cumplir la función del programa.

Estado compartido

La serialización del acceso al estado compartido permite que los procesos se ejecuten simultáneamente, pero restringe la ejecución de ciertos conjuntos de procedimientos para que solo uno se ejecute a la vez. Si un procedimiento está en ejecución, cualquier otro proceso que intente ejecutar un procedimiento en el mismo conjunto deberá esperar hasta que finalice la primera ejecución. Esto se utiliza para controlar el acceso a las variables compartidas y se pueden marcar regiones de código que no pueden superponerse en la ejecución al mismo tiempo.

Modelos de Concurrencia

1. Estado Mutable Compartido

El estado es la situación de los valores del programa (las variables). Al ser compartido, múltiples procesos del programa concurrente van a acceder a ese estado.

Podemos marcar regiones de código que no pueden superponerse en la ejecución al mismo tiempo. Si se está ejecutando algún procedimiento en el conjunto, entonces cualquier otro proceso que intente ejecutar cualquier procedimiento en el conjunto será obligado a esperar hasta que la primera ejecución haya terminado

[1] Estas son las principales complejidades adicionales de los sistemas distribuidos

Usos ideales

- Programas donde varios procesos necesitan compartir y modificar datos
- Sistemas que deben mantener coherencia en datos críticos como:
  - ✓ Bases de datos en memoria
  - ✓ Servidores multicliente donde varias conexiones escriben en la misma estructura
  - ✓ Juegos en red donde se debe actualizar el mundo concurrentemente

Ejemplos:

- Un servidor web que gestiona el inventario de productos en una tienda: varios clientes pueden hacer compras al mismo tiempo y hay que proteger la actualización del stock.
- Un sistema bancario donde varios procesos puedne modificar el saldo de una misma cuenta
- Videojuego multijugador donde varios jugadores interactúan con un mismo mapa/inventario

Ventajas: permite compartir datos en memoria directamente

Desventajas: hay que proteger el acceso para evitar condiciones de carrera con locks o semáforos. Puede ralentizar el servicio si un proceso realiza muchos accesos al recurso compartido y los otros deben esperar para poder acceder este.

2. Paralelismo Fork-Join

Este modelo se asemeja a la programación en sistema paralelo. Se puede aplicar solo a cierto tipo de problemas, donde las tareas se puedan dividir en subtareas. Esas subtareas se pueden ejecutar en simultáneo, entonces ya no hay un estado mutable compartido. Se hace un fork y después un join para unir los resultados de esas tareas; el join espera a que las subtareas terminen de ejecutarse.

Usos Ideales

En computación científica, procesamiento de datos masivos o algoritmos recursivos de ordenamiento, búsqueda en arboles (por ejemplo búsqueda de caminos en grafos grandes), procesamiento de imágenes.

Ejemplos:

- Entrenamiento de redes neuronales; calculo de grandes productos de matrices
- Renderizado de imágenes: puedo dividir una imagen en regiones y procesarlas en paralelo
- Merge sort en paralelo: cada mitad del arreglo se ordena en paralelo
- Simulación física: cada objeto simulado puede calcular su próximo estado en paralelo

Ventajas: Máxima eficiencia en tareas **computacionalmente intensivas**

Desventajas: Es necesario que las tareas sean independientes. No puedo aplicar este modelo a todos los problemas.

3. Canales/Mensajes

Los canales son vías de comunicación entre procesos en un programa concurrente, que permiten transmitir información de un proceso a otro. Esta información son los mensajes. Existen distintas abstracciones para comunicar los distintos tipos de mensajes, de esta forma un proceso manda un mensaje al otro, entonces ya no hay estado compartido. No tengo mas el problema de serializar el acceso a la variable global compartida.

Usos Ideales

En programas donde los procesos no deben compartir estado directamente. Sistemas distribuidos, microservicios, simulación de redes, programas que manejan eventos

Ejemplos:

- Sistema de microservicios: cada servicio se comunica mediante mensajes (por ejemplo MQTT o RabbitMQ)
- Pipeline de procesamiento de datos: un modulo extrae datos, otro los transforma, otro los guarda
- Una aplicación de monitoreo de sensores: cada sensor envia datos como mensajes a un proceso central que los analiza

Ventajas: sin estado compartido se reducen los errores de concurrencia

Desventajas: Complejidad en coordinar mensajes entre procesos

4. Programación Asincrónica

Intenta tener en uso lo más posible el procesador y evitar las esperas o bloqueos cuando se hacen operaciones de entrada/ salida, pues estas operaciones son lentas y el procesador se quedaría esperando a recibir esos datos.

Acá tengo muchas tareas en ejecución que trabajan colaborativamente. Cuando llega una tarea, no se bloquea el sistema; permite que todas las tareas se siguan ejecutando pues ninguna tarea ocupa completamente al procesador.

Usos Ideales

Muy útil para casos de **procesamiento de muchas tareas de cómputo liviano y alta latencia como operaciones de I/O**

Aplicaciones donde hay **muchas operaciones de entrada/salida** (I/O) que pueden bloquear al sistema. Útil para servidores web que atienden muchas peticiones concurrentes, aplicaciones que consumen APIs externas o leen archivos de la red, App que no deben “colgarse” mientras esperan datos

Ejemplos:

- App web que hace múltiples llamadas a APIs externas y no puede quedar bloqueada
  Es útil porque hacer los requests a la API puede llevar mucho tiempo si la red tiene una alta latencia.
- Servidor Node.js que atiende miles de peticiones HTTP al mismo tiempo sin crear un hilo por conexión
- Una app de escritorio que sigue responsiva mientras descarga archivos en segundo plano

Ventajas: Evita bloqueos por I/O, muy eficiente para aplicaciones reactivas

Desventajas: Puede volverse difícil de leer si se abusa de callbacks o promesas mal estructuradas

5. Actores

Evolución al modelo de canales/mensajes, donde hay una entidad principal (el actor) con vida propia que tiene su estado interno. Ese estado lo puede mutar solamente ese mismo actor. El actor va a tener un buzón de entrada donde va a estar acumulando mensajes que mandan otros actores. Entonces los actores son los que se comunican mediante mensajes.

El actor es una entidad reactiva que va a responder a la recepción de los mensajes, con procesamientos con un handler a ese mensaje. Como resultado de ese procesamiento, puede mutar el estado interno del actor; pero esta mutación no va a generar cambios en los demás actores ni tampoco generará problema de concurrencia. Los mensajes de cada actor se leerán de manera secuencial. Puedo tener muchos actores en un mismo programa.

La diferencia con los canales/mensajes es que no hay una terminación formal de la entidad que cambia el estado, solo tengo mensajes para reemplazar el uso del estado compartido.

Usos Ideales

En programas donde los procesos no deben compartir estado directamente. Sistemas distribuidos, microservicios, simulación de redes, programas que manejan eventos

Ejemplos:

- Un juego en línea donde cada jugador o NPC es un actor que responde a eventos (recibir daño, moverse, atacar)
- Sistema de chat donde cada sala/usuario es un actor que maneja sus mensajes y estado
- Simuladores: un modelo de trafico donde cada coche es un actor que decide como avanzar

Ventajas: Aislamiento completo del estado, lo cual es muy escalable

Desventajas: Posibles cuellos de botella en actores muy activos (si recibe muchos mensajes y tarda en procesarlos puede crecer mucho su buzon).

No apto para tareas con fuerte dependencia de datos entre actores: Si los actores deben consultar el estado de otros o coordinarse fuertemente, el modelo se vuelve ineficiente o complejo.

Modelo Fork-Join

Estilo de paralelización donde el cómputo (task) es partido en sub-cómputos menores (subtasks). Los resultados de estos se unen (join) para construir la solución al cómputo inicial.

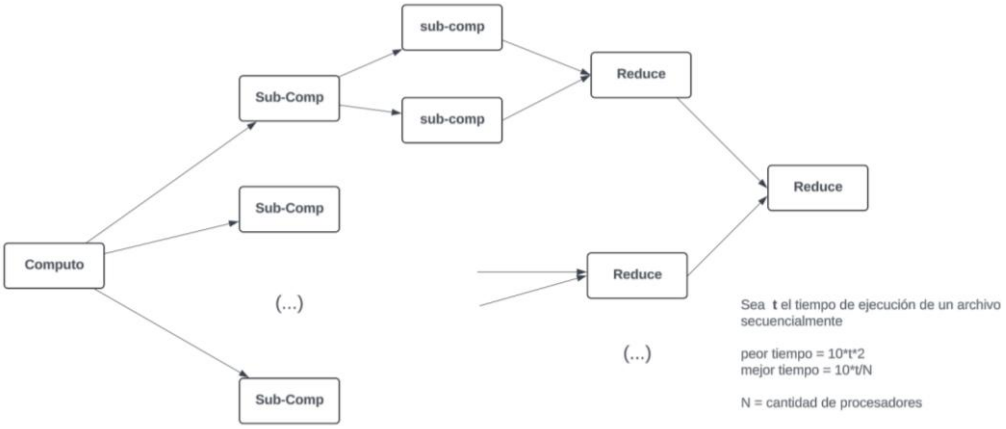
Características que tiene que tener el problema para poder aplicar este modelo

- Que pueda partir el problema en subtareas. Un thread por sub-tarea (independientes)
- Que pueda unir los resultados de cada subtarea para obtener el resultado total.
- Que se puede desarrollar recursivamente
- Cada sub-tarea debe ser **independiente** (aislada): Los datos que necesitan la tarea para hacer su computo los tiene todos la tarea; no necesita nuevos datos ni de otras tareas ni nadie.

Ventajas de este modelo

No requiere sincronización y es **determinístico.** No tiene race-conditions porque las tareas no compiten por acceder a recursos, si reciben todos los datos necesarios antes de iniciar. Las tareas, y por lo tanto los threads están aislados y el programa produce el mismo resultado independientemente de las diferencias de velocidad de los threads.

Performance ideal: **t (secuencial) / N(threads)**. Puede variar si es que tenemos una tarea grande y otra pequeña + el procesamiento de combinación.



Work-stealing

Es un algoritmo usado para hacer scheduling de tareas entre threads. Los threads libres/inactivos (aquellos que terminaron sus sub-tareas) intentar robarle tareas a otros threads ocupados para realizar balaceo de carga.

- Cada thread tiene su propia cola de dos extremos (deque) donde almacena las tareas listas por ejecutar.
- Cuando un thread termina la ejecución de una tarea, coloca las subtareas creadas al final de la cola.
- Cuando necesita una tarea para ejecutar, toma la siguiente tarea a ser ejecutada del final de la cola deque.
- Si la cola está vacía, y el thread no tiene más trabajo, tratar de robar tareas del inicio de una cola de otro thread (random).

¿Por qué le saca a otros threads del principio de sus deques y no del final? Porque el thread dueño está sacando tareas del final, entonces, de esta manera **no hay conflicto:** uno saca por delante y otro por atrás.

Los worker threads se comunican solamente cuando lo necesitan → menor necesidad de sincronización.

¿Por qué una cola 'deque' para cada thread y no una única cola compartida?

Si se utilizara una única cola de tareas compartida entre todos los threads, habría una mayor competición por el acceso a la cola. Cada vez que un thread quisiera agregar o tomar una tarea de la cola, tendría que competir con otros threads por el acceso a la cola. Esto podría aumentar el tiempo necesario para acceder a la cola y reducir el rendimiento del sistema.

Además, el uso de una única cola de tareas compartida podría aumentar la necesidad de sincronización entre los threads. Cada vez que un thread quisiera acceder a la cola, tendría que adquirir un bloqueo para evitar que otros threads accedan a la cola al mismo tiempo. Esto podría aumentar la complejidad del código y reducir el rendimiento del sistema.

Implementaciones: RAYON

Rayon permite realizar:

a. 2 tareas en paralelo:

```
let (v1, v2) = rayon::join(fn1, fn2);
```

b. N tareas en paralelo:

```
vec.par_iter().for_each(|value| {
    do_something(value);
});
```

- **.par_iter()** crea un iterador paralelo pero al ser de la biblioteca de rayon, este maneja lo threads y distribuye el trabajo.
- **.reduce()** y **.reduce_with()** se usan para combinar los resultados.

Rayon crea una tarea por elemento del vector que internamente ya implementa el work stealing.

```
use rayon::prelude::*

let s = ['a', 'b', 'c', 'd', 'e']
   .par_iter()
   .map(|c| ...)
   .reduce(|| String::new(),
      |mut a: String, b: String|
        {a.push_str(b);
         a});
```

MapReduce es un modelo de programación para procesar grandes volúmenes de datos. Divide el trabajo en dos partes: mapear datos y luego reducirlos, y lo hace en paralelo usando muchas computadoras.

Programación Asincrónica

La programación asincrónica es un modelo de programación en paralelo que **nos permite diferir la ejecución de una función a la espera de que se complete una operación** (ya sea I/O u procesamiento de una tarea en sí). De esta forma, se puede lanzar una tarea de forma asincrónica y seguir con el resto de la ejecución.

Más adelante, al momento de necesitarse el resultado de dicha tarea, podrá esperarse en caso de ser necesario (o podría haber ya terminado sin necesidad de esperar). Esto permite una mayor eficiencia en la ejecución del código y una mejor utilización de los recursos del sistema.

En lugar de detener todo hasta que termine esa tarea (por ejemplo leer un archivo, consultar una API, acceder a una BDD), **la función se "pone en pausa" y el programa sigue haciendo otras cosas**. Más adelante, cuando realmente se necesita el resultado, se puede **verificar si ya está listo** y, si no, esperar solo en ese momento.

Esto hace que el programa **aproveche mejor el tiempo y los recursos** del sistema, ya que no desperdicia el procesador esperando cosas lentas.

Características

- Pocos threads lanzados (puede llegar a haber 1 solo thread)
- Evita el overhead en memoria de crear threads
- Tareas asincrónicas son mucho más livianas que los threds; más rápidas de crear y es más eficiente pasarle el control a ellas (requieren mucho menos context switching)

Ejemplo de un servidor asincrónico

```
use std::{net, thread};
let listener = net::TcpListener::bind(address)?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await{
      let socket = socket_result?;
      let groups = chat_group_table.clone();
      task::spawn(async {
         log_error(serve(socket, groups).await);
      });
   }
}
```

El await lo que hace es “esperar a un futuro”, y eso lo tengo que hacer **cada vez que tengo una operación bloqueante en el caso sincrónico.** Otro caso bloqueante es cuando hago el iterador de las conexiones entrantes en el while. El hilo de ejecución se podría bloquear a la espera de un nuevo thread.

En vez de hacer un thread spawn, se hace un **task spawn.** La función que atiende al cliente (serve) ahora es asincrónica así que también uso await.

Futures

Representa una operación que todavía no terminó, pero que eventualmente va a producir un valor. Es una operación sobre la que se puede testear si se completó o no.

```
trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}
```

- `type Output`: el tipo de valor que va a devolver el Future cuando esté listo
- `poll(…) -> Poll<Output>:` método que verifica si el Future ya está completo o no.

```
enum Poll<T> {
    Ready(T),   // Ya se completó, T es el resultado.
    Pending,    // Todavía no está listo.
}
```

Modelo piñata de la programación asincrónica: lo único que se puede hacer con un future es golpearlo con poll hasta que caiga el valor.

Es por esto que el método **poll nunca bloquea**; nunca espera, simplemente responde si ya está listo o no Cada vez que es polleado, avanza todo lo que puede.

- Si la operación se completó, retorna: `Poll::Ready(output)` (output es el resultado final de la operación que buscabamos).
- Si no se completó, retorna `Pending` y el sistema se encarga de volver a llamar a `poll` más tarde.

Funciones Async y Expresiones Await

Ejemplo Asincrónico

```
fn read_to_string(&mut self, buf: &mut String)
  -> impl Future<Output = Result<usize>>;
```

Cada vez que es polleado, avanza todo lo que puede. Esto significa que el `Future` intenta avanzar su ejecución hasta que se encuentra con algo que tiene que esperar (como un system call que todavía no está listo). Si no puede avanzar más en ese momento, devuelve `Poll::Pending`.

El `Future` almacena lo necesario para realizar el pedido hecho por la invocación. Es una máquina de estados. Guarda en memoria en que paso del proceso esta, los datos que ya consiguió y las cosas que le faltan para seguir avanzando. Gracias a esto puede "retomar" desde donde se quedó cuando lo vuelvan a llamar con `poll()`.

`Future` se usa para la arquitectura **async**. En esta, se llama a `poll` cuando realmente vale la pena (cuando retorna `Ready` o puede avanzar)

Ejemplo de un request a un server de forma async

```
use async_std::io::prelude::*;
use async_std::net;
use async_std::net::Shutdown;

async fn cheapo_request(host: &str, port: u16, path: &str) ->
std::io::Result<String> {
    let mut socket = net::TcpStream::connect((host, port)).await?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n",
path, host);
    socket.write_all(request.as_bytes()).await?;

    socket.shutdown(Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

Al invocar una función **async**, **no se ejecuta su cuerpo de inmediato**. En lugar de eso, retorna inmediatamente un **Future** (sin empezar la ejecución de la función), que es una estructura que contiene:

- Los argumentos pasados
- El estado local de la función
- El punto en el que debe continuar su ejecución cuando sea reanudada

Es decir, ese future es como un plan de trabajo con toda la información necesaria para hacer esa tarea más adelante.

Este **Future** se puede ejecutar mediante el método **poll()**, que:

1. Comienza a ejecutar el cuerpo de la función
2. Se detiene en el primer .**await** (avancé hasta que llego a algo que tarda, como conectarme a un servidor por ejemplo)
3. Si la operación **await** aún no se completó, devuelve **Poll::Pending**

Poll es llamado periódicamente por el executor. Si bien el executor procura no hacer llamadas de más, es posible que haga poll y la función siga bloqueada.

La expresión .await **registra al future en un executor**

- Toma control del Future interno que representa la operación esperada
- Ejecuta poll() sobre él
- Si la operación está lista, devuelve el resultado (Poll::Ready(valor)) y continúa la ejecución

- Si no, devuelve Poll::Pending y se suspende la ejecución de la función

¿Quién hace el poll() y cuando?

El await le dice al runtime/executor[2]: "hasta acá puedo llegar por ahora; avisame cuando pueda seguir". El **sistema operativo o alguna biblioteca** le avisa al runtime cuando algo está listo (por ejemplo, una conexión de red o lectura de archivo), y entonces el runtime retoma la ejecución del Future desde ese punto. El executuro guarda el future y llama a poll cuando haya progreso

Cuando el evento externo (como una conexión de red o lectura de archivo) se completa, el sistema "despierta" al Future", lo vuelve a poner en la cola de tareas del runtime y el runtime reintenta poll() y la función continúa desde el último await.

Este proceso permite que la ejecución sea no bloqueante y muy eficiente en términos de recursos, ya que **el runtime solo reanuda tareas cuando hay trabajo para hacer**.

La siguiente invocación a poll() sobre la función cheapo_request retoma la ejecución desde el punto donde se quedó el Future anterior (por ejemplo, después del .await al connect).

El future almacena automáticamente el punto exacto donde debe continuar su ejecución, y el estado local (incluyendo variables y datos necesarios). Las expresiones await pausan la ejecución y luego permiten que la función continue desde ese mismo lugar cuando se reanuda.

*Ejemplo paso a paso de la función **async fn** cheap_request()*

- Se llama cheap_request() desde main por ejemplo. NO se ejecuta aun, te da un Future
- El runtime llama **poll()** al future
  - ✓ Empieza la ejecución de la función
  - ✓ Llega el .await de connect((host, port)) y si la conexión todavía no está lista: Se pausa y devuelve Poll:pending
- Cuando se establece la conexión, el runtime vuelve a hacer poll(), y ahí la operación devuelve Poll::Ready(socket) o sea: el socket TCP ya conectado.
- Se avanza hasta el próximo await, y se repite el mismo proceso con el resto de la función hasta que termina

Operación	Espera (await)	Resultado al estar lista
`TcpStream::connect(...)`	✅	`Poll::Ready(socket)`
`socket.write_all(...)`	✅	`Poll::Ready(())` (todo escrito)
`socket.read_to_string(&mut response)`	✅	`Poll::Ready(())` (todo leído)
`Ok(response)` (el return de la función)	❌	Valor final del Future (tipo `String`)

Block-on

es una función sincrónica que produce el valor final de la función asincrónica. Es un adaptador del mundo asincrónico al sincrónico; Traduce el mundo sync con el async. **No** debe usarse en una función async (bloquea a todo el thread, incluyendo a otras tareas async y rompe el modelo de programación colaborativa).

*¿Qué hace **block_on** en Rust?*

[2] Ejemplos de runtimes en Rust: Tokyo y async-std. Son los sistemas/motores que gestionan las tareas asincrónicas.

- block_on ejecuta un Future hasta que esté completamente listo, es decir, bloquea el hilo actual y espera el resultado final.
- Se usa para "entrar" al mundo asincrónico desde código sincrónico.
- Es útil, por ejemplo, en el main() de un programa que no es async pero necesita correr una función async.
- Toma un Future (como el que devuelve una función async) y empieza a hacerle poll repetidamente. block_on conoce cuánto hacer sleep hasta hacer poll de nuevo.
- Si el Future devuelve Pending, block_on espera hasta que valga la pena intentar de nuevo (por ejemplo, cuando hay datos disponibles en un socket).
- Cuando el Future retorna Ready(valor), block_on devuelve ese valor

PIN

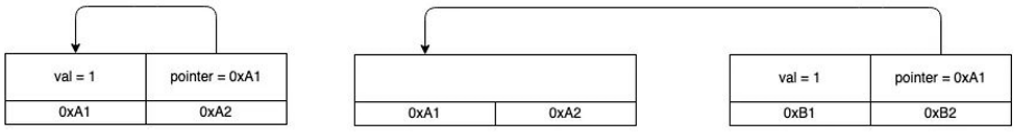
Cuando se hace poll, recibe el self dentro de un *PIN*:

```
fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Self::Output>;
```

¿Por qué?

- Los tipos autogenerados de async que implementan Future guardan referencias a sí mismos, por una cuestión de cómo tienen que volver, y demás.
- Ahora… si ellos fueran movidos (por ejemplo por estar en el stack) sus referencias internas no se actualizarían

Entonces, Las self-references se encierran en un tipo Pin (ej Pin<Box<T>>). Lo hay que hacer es declarar que estos tipos futuros NO se pueden guardar en el stack. Esto se hace con el trait Pin.



Pin evita lo que está en la imagen de arriba. Estos tipos deben mantener una referencia a ellos mismos, debemos evitar que se muevan para que no apunten a otro lado.

¿Cómo?

- Todos los tipos por defecto implementan el autotrait Unpin (como Send, por ejemplo)
  - ✓ Salvo que específicamente se marquen como !Unpin
- Las self-references se encierran en un tipo Pin (ej Pin<Box<T>>)
- Si T es !Unpin, Pin "evita" que se mueva, haciendo imposible llamar métodos que requieran &mut T como por ejemplo mem::swap. Entonces, no puedo guardar los Futures en el stack. Siempre van a estar en el Heap.

Uso de Async vs. Uso de Threads

Característica	*Async/await*	*Threads tradicionales*
Uso de memoria	Mucho menor: ~1 KB por tarea	Mayor: típicamente ~20–100 KB por thread
Tiempo de creación	Muy rápido (~300 ns)	Más lento (~15 µs o más)
Cambio de contexto	Muy eficiente (no involucra al sistema operativo)	Costoso (cambio de stack, registros, etc.)
Escalabilidad	Excelente (puede manejar miles/millones de tareas)	Limitada (decenas/miles de threads)

Bloqueo de recursos	Evita bloqueos si se usa correctamente	Bloqueo común (uso de mutex, semáforos, etc.)
Consumo CPU cuando está inactivo	Muy bajo (espera sin bloquear)	Puede desperdiciar CPU si hay threads bloqueados

Corrección de Concurrencia y Locks

Corrección

En programas secuenciales (sin concurrencia), si algo sale mal, podés depurar (debuggear) porque el programa siempre se comporta igual para una misma entrada. En cambio, en programas concurrentes, el resultado puede variar dependiendo del orden en que se ejecuten las tareas.

La corrección de concurrencia asegura que el resultado de la ejecución concurrente sea el mismo que si las operaciones se hubieran ejecutado en algun orden secuencial válido, sin errores.

Propiedades de Corrección:

1. **Safety:** debe ser verdadera siempre. Indica que durante la ejecución del programa no deben ocurrir cosas malas, como por ejemplo, que no se ejecute una operación imposible o que no entre un proceso en una sección crítica cuando ya hay otro proceso dentro.

Ejemplos:

- *Exclusión mutua*: dos procesos no deben ejecutar instrucciones de la sección crítica al mismo tiempo (por ejemplo, modificar una variable global compartida).
- *Ausencia de deadlock*: los procesos no deben quedarse todos esperando entre sí indefinidamente sin avanzar. Un sistema que aún no finalizó debe poder continuar realizando su tarea, es decir, avanzar productivamente

2. **Liveness**: Garantiza que algo bueno ***eventualmente*** ocurra. Se enfoca en que el sistema no quede colgado ni congelado. Indica que durante la ejecución del programa deben ocurrir cosas buenas, es decir, que el programa avance en sus tareas y cumpla con ciertas condiciones.

Ejemplos:

- *Ausencia de starvation* (inanición): si un proceso está listo para usar un recurso, en algún momento debe poder acceder a él.
- *Fairness (Equidad):* si una instrucción está *siempre habilitada* (lista para ejecutarse), entonces en algún momento debe ejecutarse. Garantiza que ningún proceso será permanentemente excluido o postergado indefinidamente.

Sección Crítica: Definición del Problema

Cada proceso concurrente suele ejecutarse en un loop infinito cuyo código puede dividirse en parte crítica y parte no-crítica.

La sección crítica debe progresar (osea una vez que entra, debe finalizar eventualmente), mientras que la sección NO crítica no requiere progreso ni exclusividad(el proceso puede terminar o entrar en un loop infinito sin afectar la concurrencia)

Especificaciones de corrección de concurrencia en la sección crítica:

- **Excusion mutua**: No deben intercalarse instrucciones de la sección crítica. Nunca deben ejecutarse al mismo tiempo dos secciones críticas de procesos distintos

- **Ausencia de Deadlock**: Si dos procesos están tratando de entrar a la sección crítica, eventualmente alguno de ellos debe tener éxito. El sistema no debe quedar bloqueado esperando
- **Ausencia de Starvation:** Si un proceso trata de entrar a su sección crítica, eventualmente debe tener éxito. Nadie debe quedarse esperando para siempre

Locks

Son mecanismos de sincronización que aseguran exclusión mutua: permiten que solo un proceso acceda a una sección crítica a la vez. Para la implementación se necesita soporte tanto del hardware como del sistema operativo.

Se implementan mediante variables de tipo lock, que contienen el estado del mismo. Se utilizan mediante los métodos lock() y unlock().

- *Método lock():* el proceso se bloquea hasta poder obtener el lock.
- *Método unlock():* el proceso libera el lock que tomó previamente con lock().

Características

- Los threads deben tomar el lock (o esperar a que se libere) y al terminar liberarlo para que otro pueda tomarlo
- Si no se libera el lock, se entra en un ***deadlock***
- Lock lectura: pueden haber múltiples accediendo a la sección crítica mientras no hayan de escritura
- Lock escritura: solo puede haber 1 solo accediendo a la sección crítica.

Locks en UNIX

- Shared Locks
- Exclusive locks

Locks en RUST

Traits Send y Sync: El trait marker **Send** indica que el ownership del tipo que lo implementa puede ser transferido entre threads.

- Casi todos los tipos de Rust son Send, pero hay excepciones, como **Rc<T>**.
- Los tipos compuestos formados por tipos Send son automáticamente Send.
- Casi todos los tipos primitivos son Send, excepto los *raw pointers*.

El trait marker **Sync** indica que es seguro para un tipo que implementa Sync ser referenciado desde múltiples threads.

- Un tipo T es Sync si &T (una referencia a T) es Send.
- Los tipos primitivos son Sync, y los tipos compuestos formados por tipos Sync son automáticamente Sync.

Rust provee locks compartidos (de lectura) y locks exclusivos (de escritura) en el módulo `std::sync::RwLock`. No se provee una política específica, sino que depende del sistema operativo.

Se requiere que **T sea Send** para ser compartido entre threads **y Sync para permitir acceso concurrente entre lectores**.

```
use std::sync::RwLock;

let lock = RwLock::new(5);
```

Obtener un Lock de Lectura

```
fn read(&self) -> LockResult<RwLockReadGuard<T>>
```

Bloquea al thread hasta que pueda obtener el lock con acceso compartido. Puede haber otros threads con el lock compartido.

Ejemplo de lock de lectura

```
use std::sync::RwLock;

fn main() {
        let lock = RwLock::new(1);

        let n = lock.read().unwrap();

        println!("El valor encontrado es: {}", *n);

        assert!(lock.try_write().is_err());
}
```

Obtener un Lock de Escritura

```
fn write(&self) -> LockResult<RwLockWriteGuard<T>>
```

Bloquea al thread hasta que pueda obtener el lock con acceso exclusivo. Retorna una protección que libera el lock con RAII. Una vez obtenido el lock, se puede acceder al valor protegido.

Locks Envenenados

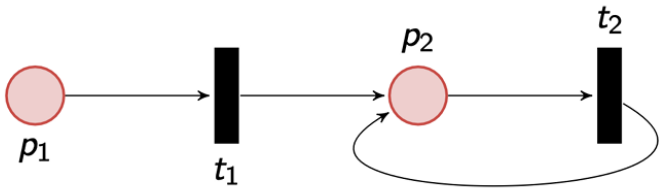
Un Lock queda en estado *envenenado* cuando un thread lo toma de forma exclusiva (write lock) y mientras tiene tomado el lock, ejecuta **panic!**

Las llamadas posteriores a read() y write() sobre el mismo lock, devolverán error.

Prevención de Deadlocks

Un deadlock es una situación en la que dos o más procesos compiten por recursos limitados y tienen la capacidad de adquirir y retener un recurso, evitando que otros lo usen. En un sistema distribuido, los deadlocks pueden ser más complicados de detectar y resolver debido a la naturaleza descentralizada del sistema.

Los procesos quedan bloqueados para siempre esperando recursos que están siendo retenidos por otros procesos que también están esperando.



Para evitar un deadlock, el sistema operativo puede evitar cualquiera de las cuatro condiciones necesarias para que se produzca: exclusión mutua, espera y retención, no apropiación y espera circular.

Si estas condiciones se cumplen simultáneamente, puede ocurrir un deadlock:

1. ***Exclusión mutua**:* Esta condición establece que solo un proceso puede tener acceso a un recurso en un momento dado. Si otro proceso desea acceder al mismo recurso, debe esperar hasta que el proceso que lo tiene lo libere.

2. ***Espera y retención**:* Esta condición establece que un proceso puede mantener uno o más recursos mientras espera la asignación de otros recursos.
3. ***No apropiación**:* Esta condición establece que un recurso asignado a un proceso no puede ser tomado por la fuerza por otro proceso. El recurso solo puede ser liberado voluntariamente por el proceso que lo tiene.
4. ***Espera circular**:* Esta condición establece que debe haber un conjunto de procesos en el que cada proceso está esperando un recurso que está siendo retenido por el siguiente proceso en el conjunto. Hay una cadena de procesos donde cada uno espera algo que otro está usando, formando un círculo de espera.

El problema del banquero

Un viejo banquero, invierte su plata 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.

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

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;

        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
    }
}
```

El problema de hacerlo así es que todas las semanas dividimos, repatirmos el saldo y lanzamos un thread por cada inversor (todas las semanas). Estoy levantando y matando threads todo el tiempo, cuando siempre hacen lo mismo.

Además, capaz no es positivo que cada inversor se spawnee todas las semanas. Acá son stateless; tal vez deberían estar "vivos" de verdad. Por ejemplo, si esto fuera un juego, estaría bueno que mantuvieran una entidad que mantenga vivos a los jugadores.

Vamos a transformar esto del mundo ***stateless*** del fork-join al mundo ***statefull*** del estado compartido.

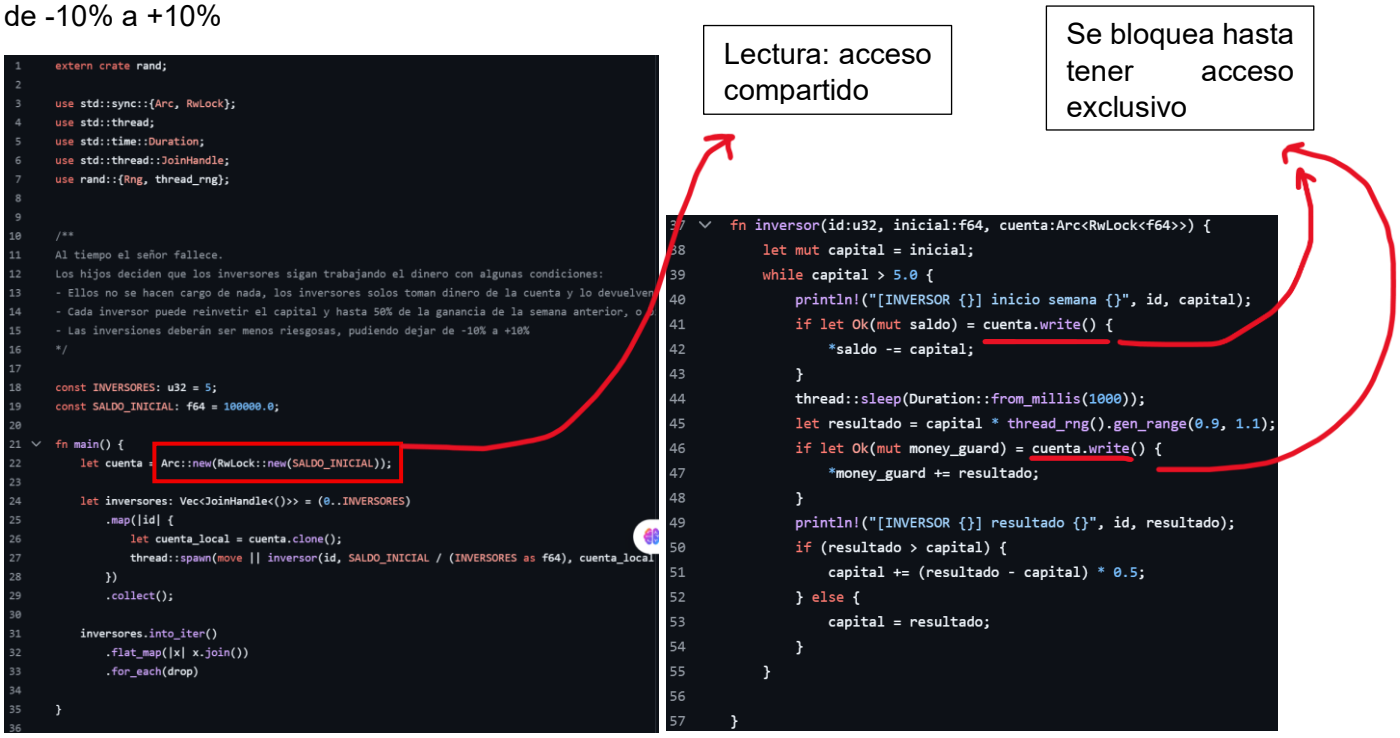
Para cambiar esta implementación de fork-join para usar estado mutable compartido, sin caer en un **busy wait**[3]:

- Si no tenemos una herramienta de sincronización, este problema, con esta mutable compartido, NO se puede resolver. Puedo inventarlo con busy wait (le mando un while hasta que tenga saldo), pero eso esta mal.

Por eso, hacemos una variante: 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

[3] Cuando un proceso espera constantemente viendo si se cumplió una condición o no, antes de continuar con su ejecución. En vez de dormir o bloquearse esperando, el hilo continúa ejecutándose e un bucle repetitivo, verificando constantemente si se cumple la condición para continuar con la siguiente. El BW, se utiliza para situaciones donde el tiempo de espera es corto y se necesita una rta rápida para continuar, pero puede ser ineficiente si el tiempo de espera es largo, ya que desperdicia recursos del CPU sin hacer avances reales en el procesamiento.

puede reinvertir el capital y hasta 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%




```rust
extern crate rand;

use std::sync::{Arc, RwLock};
use std::thread;
use std::time::Duration;
use std::thread::JoinHandle;
use rand::{Rng, thread_rng};


/**
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
- Cada inversor puede reinvetir el capital y hasta 50% de la ganancia de la semana anterior, o
- Las inversiones deberán ser menos riesgosas, pudiendo dejar de -10% a +10%
*/

const INVERSORES: u32 = 5;
const SALDO_INICIAL: f64 = 100000.0;

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

    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)

}
```

```rust
fn inversor(id:u32, inicial:f64, cuenta:Arc<RwLock<f64>>) {
    let mut capital = inicial;
    while capital > 5.0 {
        println!("[INVERSOR {}] inicio semana {}", id, capital);
        if let Ok(mut saldo) = cuenta.write() {
            *saldo -= capital;
        }
        thread::sleep(Duration::from_millis(1000));
        let resultado = capital * thread_rng().gen_range(0.9, 1.1);
        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;
        }
    }

}
```

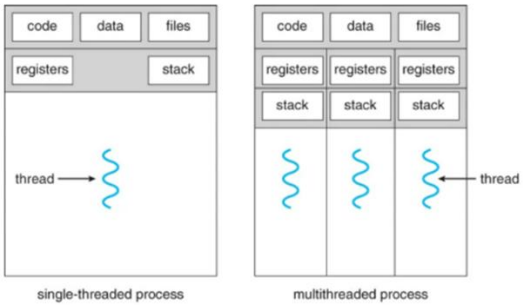
Esta versión SI se puede implementar: acá ya no necesito sincronizar cada semana. Vos le podés poner un mutex a la cuenta de acuerdo al invesor, lo bloqueas, le metes la plata y después en un futuro cuando tenes ganas bloqueas y te fijas cuanta palta hay. Así, los inversores pueden trabajar cuando quieran sin tener que esperar a que arranque la semana para llenar la cuenta.

Tenemos una cuenta por cada inversor, con un ArcMutex. Cada inversor tiene acceso a su cuenta. Tendría que bloquear cada vez que ingresa plata. Despues, cuando quiera, YO puedo bloquear las cuentas de los inversores y ver cuanta plata hay, sin frenar el trabajo de los inversores.

A cada inversor, le doy una copia del RC (que va a estar apuntando al lock) (hago el clone xq sino estoy moviendo la cuenta_local al hilo, y lo “pierdo”, así que hago una copia local al RC (reference counter)

Procesos vs. Threads

Característica	Proceso	Hilo (Thread)
Definición	Programa en ejecución	Subunidad de ejecución de un proceso
Memoria	Tiene su propio espacio de memoria	Comparte la memoria del proceso
Independencia	Totalmente independiente	Dependiente del proceso y de otros hilos
Costo de creación	Alto (más recursos del sistema)	Bajo (ligero)
Comunicación entre ellos	Difícil (IPC: pipes, sockets, etc.)	Fácil (comparten variables, memoria)
Fallos	Si un proceso muere, no afecta a otros	Si un hilo falla, puede afectar a todos los hilos del proceso



Los threads comparten recursos del proceso, entre ellos, el espacio de memoria. Cada thread tiene su propia información de estado en su **propio stack** (donde almacenan variables locales, dirección de retorno, datos temporales, etc) cada thread tiene un contexto de ejecución independiente.

Lo que comparten los threads en un procesos on el heap y los segmentos de memoria global. Compartir el heap requiere de **sincronización explicita** porque:

- Dos threads pueden leer al mismo tiempo sin problema, pero si uno escribe mientras el otro lee tendremos condiciones de carrera

Los threads también necesitan de **context switching** porque tienen stacks y contextos de ejecución independientes.

Programacion Asincrónica vs Programacion Multihilo

Rust (y otros lenguajes modernos) diferencian **paralelismo (usando procesos o threads)** de **concurrencia asincrónica (futures, async/await)**:

Programación asincrónica

- Usa 1 solo thread (por defecto) que espera múltiples tareas (futures).
- No crea nuevos procesos ni threads para cada tarea.
- Usa un executor (como Tokio o async-std) que usa poll y await para ejecutar tareas cuando están listas.
- Ejemplo típico: manejar 1000 conexiones sin abrir 1000 threads (ideal para servidores).

Programación multithread (paralelismo)

- Usa múltiples threads del sistema.
- Cada thread puede ejecutar una tarea diferente al mismo tiempo en distintos núcleos del procesador.
- Ideal para tareas intensivas en CPU.

Programación multiproceso

- Usa múltiples procesos separados.
- Se usa cuando se necesita aislamiento fuerte o por restricciones del sistema operativo.
- Ejemplo: cada pestaña del navegador como un proceso aislado.

Característica	Procesos	Threads	Async/Futures (en 1 thread)
Aislamiento	Alto	Bajo	Bajo
Compartición de memoria	No	Sí	Sí (en el mismo thread)
Costo de creación	Alto	Medio	Muy bajo
Ideal para	Aislar tareas	CPU-bound	I/O-bound, alta concurrencia
Preemptivo	Sí	Sí	No (cooperativo)

Modelo Preemtive vs Modelo Colaborativo

Modelo Preemptive (con concurrencia por interrupción)

- El sistema (SO o runtime) decide cuándo una tarea debe ser interrumpida.
- Una tarea puede ser pausada en cualquier momento, incluso si no ha terminado voluntariamente.
- Esto permite que el sistema reparta equitativamente el uso de CPU entre muchas tareas.

Ejemplo: En sistemas operativos modernos, como Linux o Windows, los threads son preemptivos. Si un hilo consume demasiado tiempo, el sistema lo interrumpe y da paso a otro.

Modelo Cooperativo (colaborativo o cooperativo)

- Las tareas ceden voluntariamente el control al sistema.
- El sistema no puede interrumpir una tarea si esta no lo permite.
- Esto requiere que las tareas estén bien comportadas: deben ceder el control regularmente (por ejemplo, usando await o yield).

Ejemplo: En la programación asincrónica en Rust con async/.await, el modelo es cooperativo: un Future solo avanza cuando se hace await o se libera el control.

Sincronización: Semáforos, Barreras y Monitores

La sincronización es coordinar la ejecución de múltiples threads para que accedan correctamente a recursos compartidos, evitando errores como bloqueos, inconsistencias o condiciones de carrera.

Semáforos

Son mecanismos de sincronización que permiten controlar el acceso concurrente a recursos compartidos. Son una construcción de programación concurrente de alto nivel.

Es un tipo de dato compuesto por dos **campos**:

- V = **un contador** representado por un entero no negativo. Se inicializa con un valor $K \geq 0$
- L = **un set de procesos**. Se inicializa este conjunto vacio $\emptyset$

Un semáforo es un contador que representa la cantidad de recursos disponibles. Si contador > 0 ⇒ recurso disponible, y si contador ≤ 0 ⇒ recurso no disponible

Si el valor de contador es cero o uno, se llaman semáforos binarios y se comportan igual que los locks de escritura.

Se definen dos **operaciones atómicas** sobre un semáforo S:

- **wait(S)** también llamada p(S).
  se usa para intentar ocupar un recurso
  Si hay recursos disponibles (V > 0), ejecuta el proceso y se acualiza S.V = S.V - 1. En caso contrario, el proceso se bloquea y se agrega a la lista de procesos en espera **L**
- **signal(S)** también llamada v(S).
  se usa para liberar o devolver el recurso.
  Si existen procesos bloqueados en L, lo despierta (no se especifica cual), lo pone en estado *ready* y el proceso continúa su ejecución (continua justo después de donde quedó, luego del wait) en su sección crítica.
  Si no hay procesos esperando entonces libera recursos, actualiza S.V = S.V +1

Lo que se limita es la cantidad de threads que pueden acceder a una sección crítica al mismo tiempo.

Operación *wait(S)*

```
if S.V > 0
  S.V := S.V - 1
else
  S.L add p
  p.state := blocked
```

Operación *signal(S)*

```
if S.L is empty
  S.V := S.V + 1
else
  sea q un elemento arbitrario del conjunto S.L
  S.L remove q
  q.state := ready
```

Tanto wait como signal son operaciones atómicas. Se serializa la ejecución de dos operaciones atómicas. Entonces, si dos procesos quieren ejecutar, uno va a tener que esperar; eso lo asegura el sistema operativo. Acá no hay condición de carrera; uno va a quedar bloqueado.

Invariantes de semáforos[4]

Estas son características que se deben cumplir ***siempre.*** Sirven para probar que nunca se va a llegar a un estado inválido y verifican que el algoritmo esté bien diseñado.

- S.V ≥ 0 → nunca puede haber un número negativo de recursos
- S.V = k + #signal(S) − #wait(S) → Esta es una manera de verificar que el número actual de recursos (S.V) es igual a la cantidad con la que empezó (k) + las señales realizadas – los bloqueos realizados.

Semáforos en Rust

Usamos el crate **std-semaphore:**

- Inicializacion

  ```
  let sem = Semaphore::new(5); // Crea un semáforo con valor inicial 5
  ```

  El 5 simboliza que se pueden acceder hasta 3 recursos al mismo tiempo

- Obtener acceso **(wait)**

  ```
  sem.acquire(); // Disminuye el valor del semáforo o bloquea si es 0
  ```

  Bloquea el hilo si no hay lugar; espera hasta que alguien libere.

- Liberar el semáforo **(signal)**

  ```
  sem.release(); // Incrementa el valor del semáforo
  ```

  Suma uno al contador, liberando el recurso

- Obtener acceso con el patrón RAII **(wait)**

  ```
  let guard = sem.access(); //Lock automático que se libera al salir del scope
  ```

  Cuando guard sale del scope, automáticamente se libera el recurso, esto se basa en el partron RAII (resource Acquisition Is Initialization). El recurso se adquiere al crear la variable y se libera al destruirla

Ejemplo básico

[4] Invariantes: condiciones que deben cumplirse/mantenerse en entornos concurrentes. Son importantes para garantizar la consistencia e integridad de los datos compartidos y para prevenir comportamientos inesperados

```rust
fn main() {
    let sem: Arc<Semaphore> = Arc::new(data: Semaphore::new(count: 2)); // Máximo 2 hilos simultáneos

    for i: i32 in 0..5 {
        let sem_clone: Arc<Semaphore> = Arc::clone(self: &sem);
        thread::spawn(move || {
            let _access: SemaphoreGuard<'_> = sem_clone.access(); // wait automático
            println!("Hilo {} entrando", i);
            std::thread::sleep(dur: std::time::Duration::from_secs(2));
            println!("Hilo {} saliendo", i);
            // release automático al finalizar el scope
        });
    }

    std::thread::sleep(dur: std::time::Duration::from_secs(5));
}
```

```
     Running `target\debug\semaphores.exe`
Hilo 2 entrando
Hilo 4 entrando
Hilo 4 saliendo
Hilo 2 saliendo
Hilo 0 entrando
Hilo 1 entrando
Hilo 0 saliendo
Hilo 1 saliendo
Hilo 3 entrando
```

Barriers

Permiten sincronizar varios threads en puntos determinados de un cálculo o algoritmo. Se usan para que se esperen entre sí hasta que todos alcancen cierto punto. Las barreras en Rust son **reutilizables** automáticamente tras cada sincronización.

- Creación de la barrera: `fn new(n: usize) -> Barrier`

```
let barrier = Barrier::new(n); // n: cantidad de threads a sincronizar
```

- Bloquear al thread hasta que todos se encuentren en el punto: `fn wait(&self) ->BarrierWaitResult`

```
let wait_result = barrier.wait(); // El thread se bloquea hasta que lleguen todos
```

Bloquea los procesos en un punto de sincronismo y se desbloquea cuando el semáforo llega a 0 (el thread líder lo desbloquea)

- Saber si un thread es el líder: el `método BarrierWaitResult::is_leader()` devuelve true en el thread líder:

```
if wait_result.is_leader() {
    // Este thread es el "líder" del grupo
}
```

Cuando se hace un llamado a **.wait(),** esta devuelve un valor tipo BarrierWaitResult, que cuenta con el método is_leader(). Esta función devuelve true solo para un hilo: el **último** que llegó a la barrera. Este hilo es conocido como el **"líder"**. Los demás hilos que llamaron a .wait() recibirán false.

Las Barriers son una herramienta de sincronización en programación concurrente que permiten coordinar múltiples hilos para que todos lleguen a un punto determinado antes de seguir adelante. Sirven para asegurar que todos los hilos hayan terminado una tarea antes de que empiece la siguiente fase.

Son útiles en problemas de, por ejemplo, simulaciones por pasos, algoritmos paralelos por fases (como multiplicación de matrices por bloques), motores de videojuegos donde las tareas (renderizado, física, AI) se sincronizan por frame.

Problemas Clásicos: Productor-Consumidor

Se definen dos familias de procesos: Productores y Consumidores.

Requisitos que deben cumplir:

1. no se puede consumir lo que no hay
2. todos los ítems producidos son eventualmente consumidos
3. al espacio de almacenamiento se accede de a uno
4. se debe respetar el orden de almacenamiento y retiro de los ítems. Es de tipo FIFO. Los ítems se van a consumir en el orden que se produjeron.

Esto se puede aplicar a muchos casos. Por ejemplo, en un sitio de ventas online. El consumidor es el vendedor que va consumiendo las órdenes de compra de sus usuarios, por ejemplo.

Podemos sincronizar un problema productor-consumidor con ***semáforos,*** iniciados en 0 tal que cuando producimos notificamos a quien espera, y así logramos sincronización.

Al utilizar un buffer de comunicación se presentan los siguientes problemas de sincronización:

1. No se puede consumir si el buffer está vacío: Esto se desprende de una de las premisas del programa: no se puede consumir lo que no hay. **Sincronizamos al consumidor con que el buffer no este vacío**
2. No se puede producir si el buffer está lleno: de esta forma, frenamos al productor para que no siga produciendo, porque no puede colocar más elementos en el buffer. **Sincronizamos al productor con que el buffer no esté lleno**

Buffer Infinito

Solo se presenta el problema 1. En este modelo no tenemos el problema 2 porque nunca se llena el buffer. Es teórico porque ninguna computadora tiene memoria infinita.

buffer := emptyQueue	*sem notEmpty (0, ∅)*
Productor	**Consumidor**

`dataType d`
`loop forever`
`p1: append(d, buffer)`
`p2: signal(notEmpty)`
	`dataType d`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: d <- take(buffer)`

Se resuelve de la siguiente manera: inicializo el buffer y el semáforo *sem.* Inicialmente vale 0 porque no tengo ningún elemento producido cuando arranco. Tanto el prod como el sum están en un loop infinito. El loop siempre va a poder colocar el producto d en el buffer, y le va a permitir al consumidor consumirlo. El consumidor hace un loop infinito con el wait en el semáforo, que bloquea al consumidor hasta que haya algo para leer. Ahí, toma el producto.

Buffer Acotado

Se presentan ambos problemas

buffer := emptyQueue sem notEmpty (0, ∅) sem notFull (N, ∅)	
Productor	**Consumidor**

`dataType d`
`loop forever`
`p1: producir`
`p2: wait(notFull)`
`p3: append(d, buffer)`
`p4: signal(notEmpty)`
	`dataType d`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: d <- take(buffer)`
	`q3: signal(notFull)`
	`q4: consume(d)`

- wait() lo hace el consumidor: espera a que haya algo disponible.
- signal() lo hace el productor: avisa que agregó un ítem.

No es necesario que productor y consumidor sean el mismo hilo.

En un caso real, el recurso es ***cuanto espacio disponible tengo para colocar elementos.*** En el caso del semáforo, es cuantos elementos disponibles tengo para consumir

El sem notFull se inicializa en N, porque inicialmente, cuando el programa arranca, tengo todo el buffer disponible para leer.

Ahora el productor va a hacer el wait sobre el semáforo que no está lleno, porque tiene que esperar a que le permitan colocar en el buffer. Una vez que se desbloquea, sabemos que hay lugar y hace

el append. Luego, el consumidor es homologo: hace el wait para esperar que haya producto. Hace el signal de notFull para indicare al productor que tiene espacio donde consumir.

Diferencias entre Locks y Semáforos

Lock	Semaforo Binario (Mutex)
Solo el thread que tiene el lock puede liberarlo	Cualquier thread libera recursos y despierta a un thread bloqueado
Los threads utilizan los métodos lock y unlock para tomar y liberar un recurso, y lo deben hacer ellos unicamente	El wait y el signal lo pueden realizar dos procesos distintos, son una herramienta mas flexible
El thread interesado está preguntando si se liberó el lock	Se notifica al thread interesado que se liberó el recurso despertandolo

Monitors

Son **mecanismos de sincronización** que combinan exclusión mutua con condiciones de espera/notificación (wait/notify)

Son como semáforos busy-wait que usan cond-var (conditional variables) que notifican recursos. Es una estructura que combina exclusión mutua y sincronización de condiciones en uno

Los monitores son estructuras de sincronización que:

- Permiten exclusión mutua entre hilos: si se llaman múltiples operaciones de un monitor, la implementación asegura que se corren en exclusión mutua
- Permiten que un hilo espere (block) hasta que una condición sea verdadera.
- Tienen un mecanismo para señalizar a otros hilos cuando su condición se cumple

Todo este manejo se hace de manera ***interna,*** para evitar errores comunes (como olvidarse de hacer signal() en un semáforo)

Un monitor tiene:

- Un nombre identificador.
- Variables internas, accesibles solo dentro del monitor. (protegen el estado compartido)
- Procedimientos que acceden a las variables internas (métodos del monitor).
- Una interfaz pública para los procesos externos.
- Variables de condición para sincronización interna: Permiten que un hilo se duerma esperando a que se cumpla algo; **se usa junto a un mutex**
- Una rutina de inicialización para las variables internas.

```
Let lock = mutex.lock().unwrap()

While !condition{

      condvar.wait(lock)

}
```

```
#otr proceso llama a:

condvar.notify_all()     o
notify_one()
```

La condición se chequea dentro de un bucle while [5]porque puede ocurrir sino un ***spurious wake-up:*** el thread se despierta sin que nadie lo haya notificado, o puede ocurrir que la condición todavía no se haya cumplido cuando despierta (otro thread la cambió)

Los procesos que interactúan con un monitor pueden estar en los siguientes estados:

- Esperando entrar al monitor (si otro hilo lo está usando).

[5] Me genera un busy-wait

- Ejecutando dentro del monitor.
- Bloqueado en una variable de condición (FIFO).
- Recién liberado de una condición **waitC**.
- Recién completó una operación **signalC**.

Proveen una estructura que concentra la responsabilidad en módulos. Los procesos de los monitores son ***atómicos,*** porque corren en exclusión mutua. Los monitores encapsulan la sección crítica, y actúan como semáforos bloqueantes ya que no tienen S.L

Si un semáforo olvido hacer signal(S), puedo entrar en un deadlock. Esto, con monitores no me ocurre porque el objeto mismo se encarga de esto; No me encargo de sincronizar.

Diferencias entre Monitores y Semáforos

Semáforo	Monitor
wait puede o no bloquear	**waitC** siempre bloquea
signal siempre tiene efecto	**signalC** no tiene efecto si la cola está vacía
signal desbloquea un proceso arbitrario	**signalC** desbloquea el proceso del tope de la cola
un proceso desbloqueado con signal, puede continuar la ejecución inmediatamente	un proceso desbloqueado con signalC debe esperar que el proceso señalizador deje el monitor

El proceso señalizador es el que manda el .notify()

Conditional Variables

Son mecanismos de sincronización que permiten que un hilo espere activamente hasta que otra condición se cumpla, típicamente gracias a la señalización de otro hilo. Son expresiones booleanas que se usan para probar expresiones. Para los valores de la Cond Var:

- ***False:*** bloquear
- ***True:*** correr

Una condition variable C:

- No guarda ningún valor
- Tiene asociado un FIFO
- Consta de tres operaciones atómicas:
  - ✓ waitC(cond)
  - ✓ signalC(cond)
  - ✓ empty(cond)

Aportan sincronismo al monitor. Su nombre representa la condición que se quiere:

- waitC(not True): Wait for not true to be true
- signalC(not True): signal that not zero is true

Las Condvars se usan junto con un lock (bloqueo) porque la idea es **esperar por un cambio en el estado de algún dato compartido**, y ese dato tiene que estar protegido para evitar condiciones de carrera. Permiten hacer cosas como:

- ✓ Un hilo espera a que una cola tenga datos
- ✓ Otro hilo pone datos y despierta al primero

Son una manera más estructurada y potente que solo usar **synchronized** y **wait()/notify()** .Usan un lock explícito (**ReentrantLock**)

```
lock.lock();
try {
    while (!condicion) {
        condition.await();  // el hilo se duerme y suelta el lock
    }
    // condición cumplida, hace algo...
} finally {
    lock.unlock();  // libera el lock al final
}
```

```
lock.lock();
try {
    condicion = true;
    condition.signal();  // despierta a un hilo esperando
} finally {
    lock.unlock();
}
```

Operaciones de Monitores

- waitC(cond)

```
cond.append (p)
p.state := blocked
monitor.releaseLock ()
```

- signalC(cond)

```
if ( cond <> empty )
begin
q := cond.remove ()
q.state := ready
end
```

- empty(cond)

```
return cond = empty
```

Ejemplo

- ✓ Un **consumidor** quiere leer de un buffer. Pero si el buffer está vacío, **tiene que esperar** a que haya algo.
- ✓ Un **productor** llena el buffer, y cuando lo hace, **avisa** que hay datos disponibles.

La Condvar es la forma de decir: "Voy a esperar acá hasta que me avises que puedo seguir".

Problema del Banquero: ***resuelto con Barriers***

Necesito dos barreras porque necesito saber dos cosas

1. saber cuando ya están todos listos para iniciar la semana (empezar a repartir la plata e invertir): una vez que llegan todos, puedo dividir la plata entre los N amigos
2. saber cuando ya esta calculado/cuanto se debe llevar cada uno. Una vez que todos leyeron cuanta plata se lleva cada uno, pueden escribir/calcular cuando van a hacer esa semana. No puede ninguno poner/llevarse mas plata hasta saber cuanto tiene cada uno.

```
3   use std::sync::{Arc, RwLock, Barrier};
4   use std::thread;
5   use std::time::Duration;
6   use rand::Rng;
7
8   const FRIENDS: u32 = 10;
```

```
10  /**
11  Al tiempo el señor banquero fallece.
12  Los hijos deciden que los inversores sigan trabajando el
13  dinero pero ellos no se hacen cargo de nada. Los inversores solos deberán tomar el
14  dinero de la cuenta al inicio de la semana y devolverlo al final.
15  */
```

```rust
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));

    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)));
    }

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

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

    while *lock.read().unwrap() > 1.0 {
        // Espero a que todos inicien la semana
        barrier.wait();

        let prestamo = *lock.read().unwrap() / FRIENDS as f64;
        println!("inversor {} inicio semana {} plata {}", id, semana, prestamo);
        // Espero a que todos hayan leido el saldo disponible
        barrier2.wait();

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

        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);

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

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

Problema del Barbero: ***resuelto con semaforos***

Podemos resolverlo con el uso de ***tres semáforos independientes:***

1. semáforo de cliente esperando a ser atendido
   - el barbero “espera al recurso”
   - el cliente “libera” al recurso = hay un cliente disponible a ser atendido
2. semáforo del barbero para avisar que está listo para atender o que esta ocupado
   - el cliente espera al barbero
   -el barbero “Libera” el recurso cuando esta listo para atender a un cliente
3. semáforo del barbero para avisarle al cliente que termino de cortarle el pelo
   - el cliente espera mientras que el barbero le corta
   -el barbero libera el recurso cuando termino el corte

Acá estamos usando un conjunto de semáforos para modelar una cola de espera y un punto de encuentro.

```rust

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

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

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

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

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

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

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

```rust
    let customers: Vec<JoinHandle<()>> = (0..(N+1))
        .map(|_| {
            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 {
                thread::sleep(Duration::from_secs(thread_rng().gen_range(2, 10)));

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

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

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

                println!("[Cliente {}] Me siento en la silla del barbero", me);

                println!("[Cliente {}] Esperando a que me termine de cortar", me);
                haircut_done_customer.acquire();

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

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

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

Problema de los Filósofos: ***resuelto con Semáforos***

dos filósofos contiguos no pueden comer al mismo tiempo; se tienen que esperar. Si todos quieren comer al mismo tiempo, se quedan todos esperando para siempre a otro palito. Esto conlleva un par de asunciones:

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

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

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

}
```

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

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

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

    loop {
        println!("filosofo {} pensando", id);
        //thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
        println!("filosofo {} esperando palito izquierdo", id);
        {
            let first_access = first_chopstick.access();
            // pausa despues del primer palito para forzar el deadlock
            thread::sleep(Duration::from_millis(1000));
            println!("filosofo {} esperando palito derecho", id);
            {
                let second_access = second_chopstick.access();
                println!("filosofo {} comiendo", id);
                thread::sleep(Duration::from_millis(thread_rng().gen_range(500, 1500)));
            }
        }
    }
}
```

- que un filosofo puede reservar un palito individualmente (osea que lo agarra y ya esta, sin pedirle permiso a nadie)

La primera solución de este problema, se llama la solución del mozo; no hay palitos en la mesa, sino que viene un mozo que me da los dos palitos que necesito. Los uso, se los devuelvo y después se los dará a otro. Teniendo 5 palitos, hasta dos filósofos podrían comer al mismo tiempo. Osea tengo una entidad a la que le pido los palitos y me los da de a pares. Me está obligando a acceder atómicamente al recurso.

Ahora, que pasa si los palitos son ***semáforos binarios.*** En el caso de modelarlo de esta manera, existe la posibilidad de llegar al deadlock.

- Están todos pensando
- Paso1: uno agarra un palito
- Paso2: agarra un 2do palito
- Si el de al lado quiere agarrarlo, va a tener que esperar a que el otro lo suelte

Si justo pasa que todos se organizan de forma tal de agarrar el primer palito, estamos en un deadlock. Cuando quieran agarrar el 2do van a quedarse todos esperando.

¿Cómo lo resolvemos cuando tenemos 1 semáforo por palito?

La solución es que uno de los filosofs sea distinto al resto: Todos toman primero el izquierdo, 2do el derecho, excepto un filosofo que lo hace al revés: esto rompe el ciclo de espera circular.

Problema de los Fumadores: ***Resuelto con Monitores***

El principal problema acá: imaginemos que el agente puso tabaco y fósforos. Si cada elemento de la mesa fuera un semáforo, cuando el agente los pone en la mesa, se les manda la señal que están habilitados. Acá, dependiendo del orden de ejecución, o de despertada, podemos llegar a un deadlock.

Con el ejemplo de tabaco y fósforos, el que puede fumar es el que tiene papel. Si este va y agarra el tabaco, cuando va a agarrar los fósforos, se los primereo el fumador de tabaco.

La solución es poner una especie de “adaptador”/”agente en el medio.

Tenemos:

- 1 agente
- 3 fumadores
- 1 monitor “mesa” que protege el acceso a la mesa y contiene condvar para los fumadores

El agente elihe 2 ingredientes, llama a la mesa con (ingrediente1, ingrediente2), y ella despierta al fumador que tiene el ingrediente3 (osea el que falta). El fumador llama mesa.tomar(faltante), fuma y luego libera la mesa.

```rust
fn main() {

    const N:usize = 3;

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

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

    let pair_agent = pair.clone();

    let agent = thread::spawn(move || loop {
        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();

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

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

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

```rust
    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;

                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();

                println!("[Fumador {:?}] Fumando", me);
                thread::sleep(Duration::from_secs(2));
                for ing in (*_guard).iter_mut() {
                    *ing = false;
                }
                println!("[Fumador {:?}] Terminé", me);
                cvar.notify_all();
            })
        })
        .collect();

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

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

Para cada fumador, la mesa va a tener además de un estado, una **condition variable.** Cada fumador va a esperar a esa condavar para que cuando se pongan cosas en la mesa, estén todas menos la de ese fumador (línea 60).

Si no están todas, vuelvo a dormir esperando a que estén todas, pero se duerme sin el lock tomado. Se despierta, verifico mi condición, tomo el lock y libero.

Se gastan los ingredientes y avisa que ya está.

Del lado del agente, espera que fumen (osea que la mesa este vacia), pone elementos aleatorios nuevos y los despierta a todos.

Redes de Petri

Una **Red de Petri** es una representación matemática o gráfica de un sistema a eventos discretos en el cual se puede describir la topología de un sistema distribuido, paralelo o concurrente.

Modela el estado en el que se encuentran los threads y las transiciones entre estados.

Red ordinaria

Grafo dirigido bipartito que cumple estar formado con T, P, A:

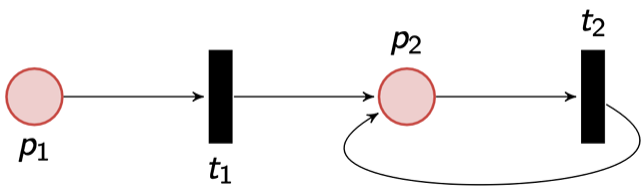
PN = (T, P, A)

Es un grafo que está formado por dos categorías de elementos, donde se vinculan elementos de la categoría1 con los de la categoria2 (o vice-versa) únicamente. Las dos categorías de elementos, son los ***tipos de nodos,*** que en este caso son T y P.

- T = t_1, t_2, ⋯, t_n: conjunto de transiciones (eventos que ocasionan los cambios de estado)
- P = p_1, p_2, ⋯, p_n: conjunto de lugares (estados del sistema)
- A ⊆ (T × P) ∪ (P × T): conjunto de arcos

El conjunto de arcos A esta comprendido en (T x P) el conjunto de todos los ejes de T a P, único con el conjunto (P X T), que son todos los ejes de P a T

Ejemplo:



pi son los estados del sistema

ti son los eventos que ocasionan los cambios de estado

Vamos a representar siempre los ***lugares*** **(places)** con círculos y a las ***transiciones*** con las barras. Los arcos me conectan **lugares con transiciones.** *NO* se puede usar un arco para unir un lugar con otro lugar, o una transición con otra transición.

Función de Marca

$$\mathbf{M: P \rightarrow N \cup 0}$$

- El domino de la función es el conjunto P (el dominio de todos los places)
- La imagen es el conjunto de todos los números naturales, a partir de 0

Esto quiere decir que la función de marca le va a asignar un nro natural a cada uno de los lugares.

Esa asociación de un valor a un place se conoce como **token**. Cuando el token está en el lugar ***p1***,entonces M(p1)=1 y M(p2)=0.Por lo tanto, M_0=(1,0). Es ir asignándole valor a cada una de las posiciones del vector de places.

Funciones de Entrada y Salida

Sea t ∈ PN =(T,P,A) una transición *t* vamos a definir las funciones:

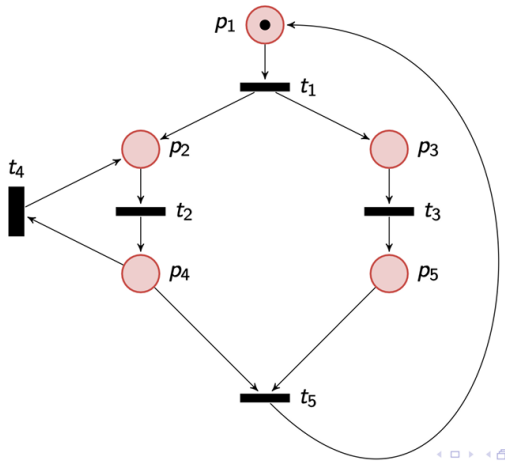
$I(t) = p/p \in P/(p,t) \in A \subset P$ es la entrada o *input* de la transición t

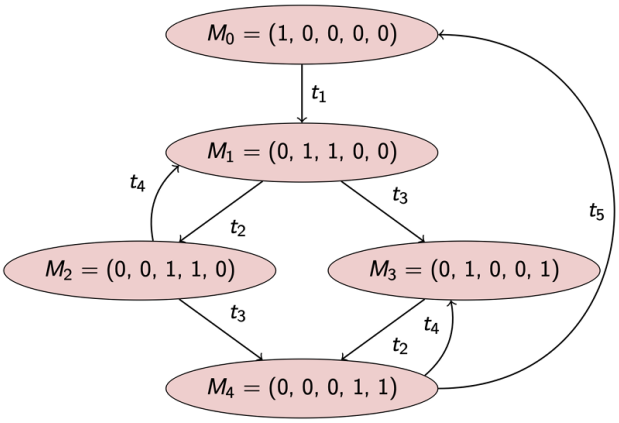
Osea, todos los posibles lugares desde los que se puede hacer una transición hacia ese place p

$O(t) = p/p \in P/(t,p) \in A \subset P$ es la salida o output de la transición t

Osea todos los posibles lugares hacia los que se puede hacer una transición desde ese place p

Ejemplo 1 y su grafo de alcance:





Tenemos los lugares p1, p2, p3, p4, p4, las posibles transiciones t, y los arcos que conectan a lugares con transiciones

Inicialmente tenemos un token en p1, así que vamos a tener un 1 en el p1 y 0 en el resto de los lugares.

Como hay un token en p1 (a ficha), se puede disparar la transición t1. Desde la transición t1, se coloca un token en p2 y p3, ya que se puede disparar la salida a p2 o a p3 (a alguna de esas).

Una vez que se hace eso, se coloca 0 en p1 (ver el M1 del grafo). Una vez que llego acá, tengo dos tokens independientes (están en lugares distintos), por lo que se podrían disparar transiciones de forma independiente.

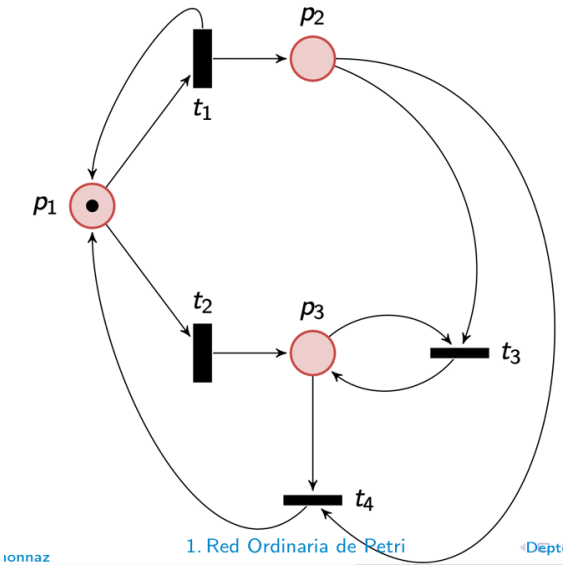
Se podría disparar una transición de p2, y que pase a tener un token en p4. Luego, estando en p4 podria dispararse t4 y volver el token a p2, o podría disparse t5.

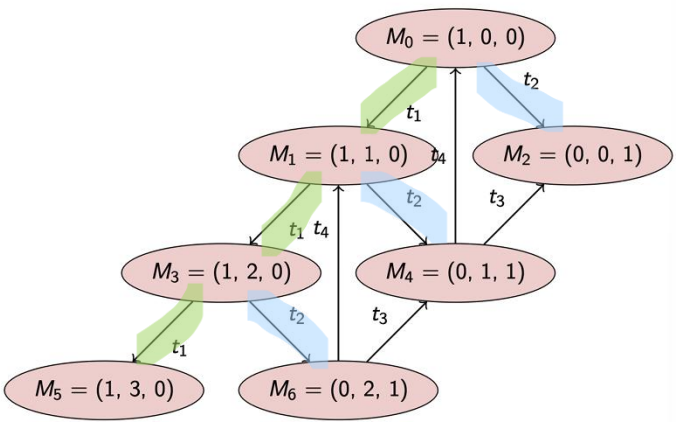
Para que se dispare t5, tiene que tener un token en p5 porque t5 tiene dos entradas.

Análogo a lo que pasa con p2 pasa lo mismo en p3. Así es como se hacen los cambios de estados.

***Grafo de alcance*:** se usa para representar a la función Marca en cada nodo para ese estado. El arco es la transición que hace que se produzca ese cambio de estado.

Ejemplo 2 y su grafo de alcance:





Acá modelizamos otro sistema con 3 places y 4 transiciones entre los places.

- Inicialmente tenemos un token en p1 (ver M0). Con un token en p1, pueden dispararse as transiciones t1 y t2
- Si se dispara t1 va a haber un token en p1 y p2 (ver los arcos)
- Si se dispara t2 va a haber un token en p3
- Teniendo un token en p3 se puede disparar t4, ***siempre y cuando haya un token en p2, porque t4 tiene dos arcos que van hacia ella.*** Si se dispara t4, hace que se vuelva a poner un token en p1.
- Teniendo un token en p3 tambien se puede disparar t3, ***siempre y cuando haya un token en p2, porque t3 tiene dos arcos que van hacia ella,*** que hace que se vuelva a guardar el token en p3

Viendo el grafo de alcance, vemos una particularidad en este sistema: empezamos con la función de marca M0 con (1, 0, 0), que es el estado inicial del sistema

- Si se dispara la transición t1, la función de marca pasa a ser M1=(1, 1, 0), porque se pone un token en p2 y uno devuelta en p1 por el arco de arriba
- Si se dispara t2 (cuyo requisito es que haya un token en t1) tenemos M2(0, 0, 1).
  Pero el token en t1 lo sigo teniendo en en M1=(1, 1, 0), porque se recreo a sí mismo, entonces puede volver a dispararse t1, por lo que voy a tener un M3=(1, 2, 0), porque vuelve a poner un token en p2 y vuelve a recrear el token de p1, Lo mismo se puede volver a reproducir en un M5. Además, si en M3 voy a volver a estar con un token en p1, podría volver a dispararse t2, caso en el cual voy a tener 2 token en p2 y 1 en p3. Así hasta el infinito.
- Si se dispara t3, estamos consumiendo 2 token y generando 1 solo. Consumo el token de p2 y el que viene de p2, y regenera el de p3 con esa flecha que vuelve

En el caso M2(0,0,1), si solo tengo un token en p3, no se puede disparar ninguna transición, porque para disparar t3, necesito un token en p2, y para disparar t4, necesito un token en p2.

Esta red de Petri nos permite identificar un **deadlock** en nuestro sistema, porque en el esta M2, es un estado en el que el sistema no termino, y no hay transición que nos deje salir de ahí.

El hecho de encontrar un deadlock, es que puede generarse en algunos casos y no en otro.si se dispara la rama de la izquierda (M0, M1, M3, M5), no pasa nada. Ahora… se lega a disparar la otra rama (M6, M4, M2 o la rama M0, M2) y llegamos a un deadlock.

Algunas Interpretaciones

Lugares de entrada	Transiciones	Lugares de salida
Precondiciones	Eventos	Postcondiciones
Datos de entrada	Cómputos	Datos de salida
Señales de entrada	Procesamiento de Señales	Señales de salida
Bufferes de entrada	Procesador	Bufferes de salida

Red General

Grafo dirigido bipartito que cumple:

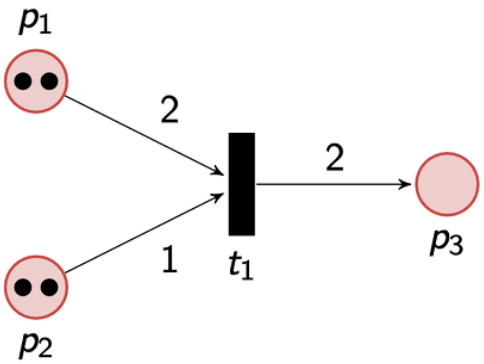
PN = (T, P, A, W, M0)

- T = t_1, t_2, ⋯, t_n: conjunto de transiciones (eventos que ocasionan los cambios de estado)
- P = p_1, p_2, ⋯, p_n: conjunto de lugares (estados del sistema)
- $A \subseteq (T \times P) \cup (P \times T)$: conjunto de arcos
- $W : A \rightarrow \mathbb{N} \in$ función de peso
- $M0 : P \rightarrow N \cup \{0\} \in$ función de marca inicial

La función de peso es una función que para cada arco le asignamos un numero natural

La función Mo (de marca inicial) es lo que nos indica con cuantos tokens va a iniciar nuestro programa.

Ejemplo



El peso se usa para generalziar el disparo de transiciones. Se asigna de manera tal de que me va a indicar **cuantos tokens se tienen que consumir para poder realizar esa transición.**

- Para disparar t1 se van a tener que consumir dos tokens de p1 y 1 token de p2
- Como output, va a generar 2 tokens para p3

Reglas generales de Disparo de Transiciones

La transición t está habilitada si y sólo si:

$M(p) \geq W(p,t) : \forall\ p \in I(t)$

Osea, la cantidad de tokens de p es mayor o igual al peso de todos los p que tienen como destino a esa transición, para todos los p pertenecientes al input de esa transicion

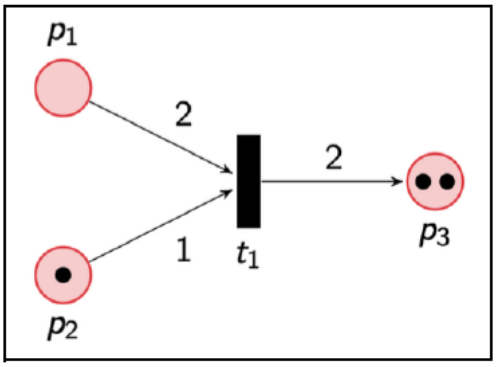
Cuando t se dispara:

- $\forall\ p \in I(t) : M(p) \leftarrow M(p) - W(p,t)$

A todos los P pertenecientes al input de la transición se les va a asignar el valor de marca correspondiente al input que tenían – el peso de ese arco

- $\forall\ p' \in O(t) : M(p') \leftarrow M(p') + W(p',t)$

Para el caso de la salida, la función de marca para todos los casos de salida es el valor de marca que tenía + el peso de ese arco.



Siguiendo el ejemplo anterior.

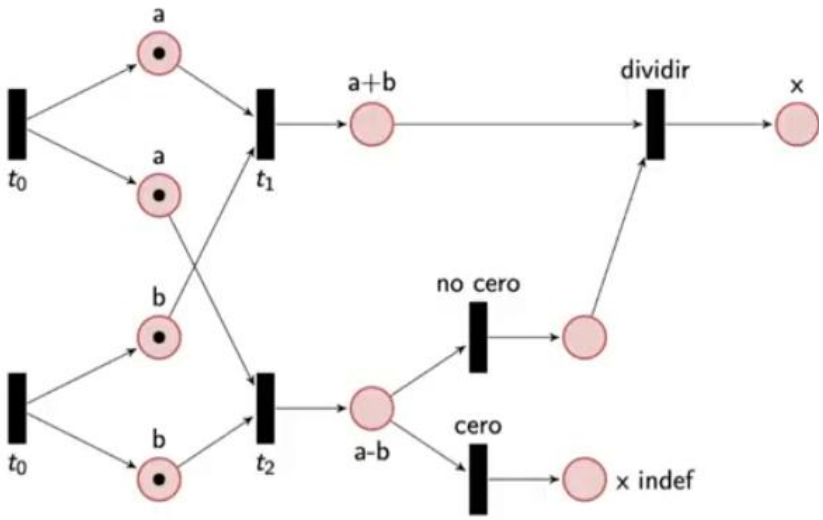
- En p1 tenía M(p1)=2 tokens. Le tengo que asignar $M(p1) - W(p1,t)$ 2-2 = 0

- En p2 tenía M(p2)=2 tokens. Le tengo que asignar M(p2) − W(p2,t) 2-1 = 1
- Inicialmente en p3 tenia M(p3) =0 token. Le tengo que asignar M(p3) + W(p3,t) 0 + 2 = 2

Ejemplos para el uso de Redes de Petri

Con esas definiciones, vamos a poder modelar un montón de sistemas.

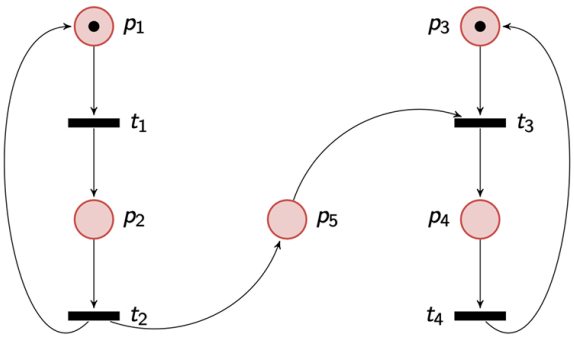
1. *Ejemplo: Modelando un cómputo aritmético*



- T0 lo hacemos para modelar que no se necesita ningún token para que se disparen esas transiciones
- A partir de t0, se crean dos a y dos b, como si se inicializaran
- T1 necesita que haya un valor en a y un valor en b (osea que estén setteados = que tengan un token). A partir de ahí, puede hacer la suma.
- Lo mismo aplica para t2 para que pueda hacer la resta de a-b
- En esta instancia voy a estar teniendo un token en a+b y otro token en a-b
- Ahora modelamos **transiciones con condición:** no solo necesito un token, sino que se tiene que cumplir una condicon a ver si esa transición se hace o no.
    - ✓ Si a-b es cero, se dispara esa transición, pasa a haber un token en el lugar x indef
    - ✓ Si a-b no es cero, pasa a haber un token en el siguiente lugar. Si queda un token en a+b, entonces se puede disparar la operación dividir como: a+b/a-b, y llegamos al lugar x

Vemos que la división se puede representar como que en un caso sí se puede hacer y en otro no. Además, este programa es finito. Llegamos a un estado final.

2. *Ejemplo: Productor Consumidor con Buffer Infinito*



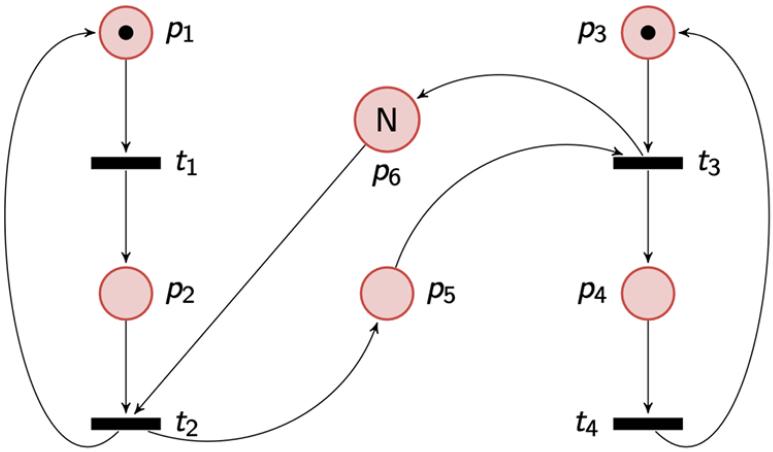
Esto representa que productor y consumidor se están ejecutando en un loop infinito. En este estado, con un token en p1 y p3, no hay ningún elemento en el buffer (ya que el buffer está representado por p5 acá).

t1 es producir, t2 es colocar en el buffer, t3 es leer del buffer para el consumidor, y t4 es elemento leído para el consumidor, y volver a consumir.

El consumidor no puede consumir hasta que no haya algo en el buffer (token en p5) porque sino no se puede disparar t3.

El productor sincroniza al consumidor. El productor puede producir infinitamente para guardar un token en p5, y el consumidor puede consumir de eso.

3. *Ejemplo: Productor Consumidor con Buffer Acotado*



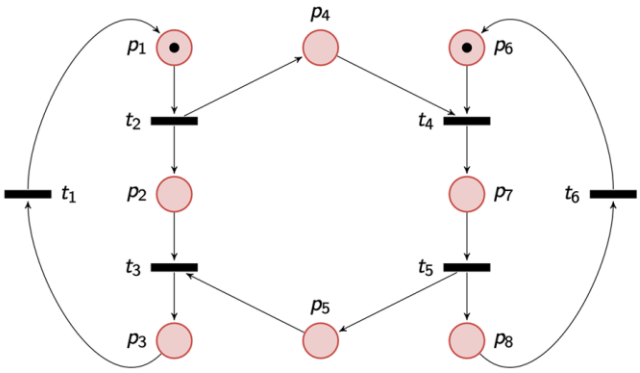
El buffer acotado quiere decir que el productor no puede producir si esta lleno el buffer. Eso es una restricción adicional sobre el caso teórico anterior. Tenemos que limitar al producto. Entonces, a la red anterior le agregamos la restricción de p6.

P6 es el place que va a estar modelando cuantos lugares libres tenemos en el buffer. Inicialmente, es un buffer con N lugares disponibles. Inicializamos la red de Petri con N token en el p6.

Ahora, para que se produzca la transición de volver al estado inicial, y colocar elemento, para poder colocar un elemento en el buffer, tiene que haber lugar libre. Para t2, tenemos que si o si consumir uno de los tokens de p6 para liberar espacio.

Luego, en p5, cuando consume, puede disparar t3 que le agrega de nuevo un elemento a p6.

4. *Ejemplo: Cliente Servidor*



Tenemos dos hilos separados. Uno es el cliente, otro el servidor, que se ejecutan independientemente, de forma tal que el servidor (ala de la derecha) queda a la espera de que el cliente le envie un mensaje, y en base a ese mensaje le envía una respuesta.

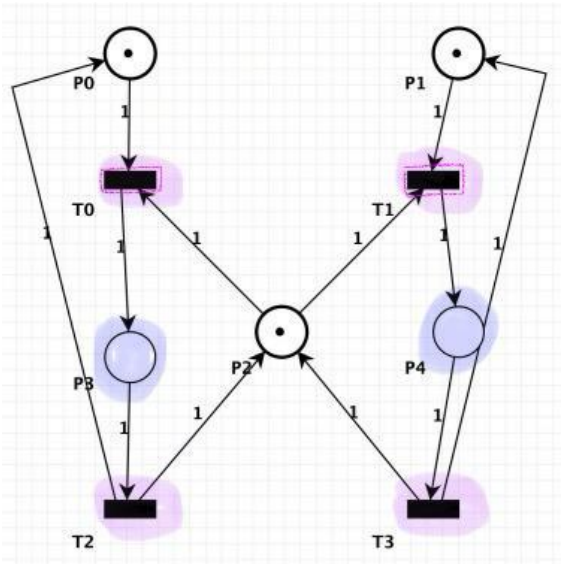
T4 es la transición de recibir mensaje. Para eso, el cliente ejecuta t2 (enviar mensaje), coloca el mensaje en la red (p4). P2 es estoy esperando recibir respuesta.

P7 es mensaje recibido y procesando mensaje. Una vez procesado se dispara t5 (envió mensaje por la red), colocándolo en p5 y vuelve a colocarlo en p8 que es disponible para recibir otro mensaje, que a través de t6 lo vuelve a colocar en t6.

Luego de p5 se puede disparar t3, que lo pasa a estado p3 (respuesta recibida) y permite pasar por t1 para volver al estado inicial de preparado para realizar pedido.

Ejemplos clase práctica

Mutex

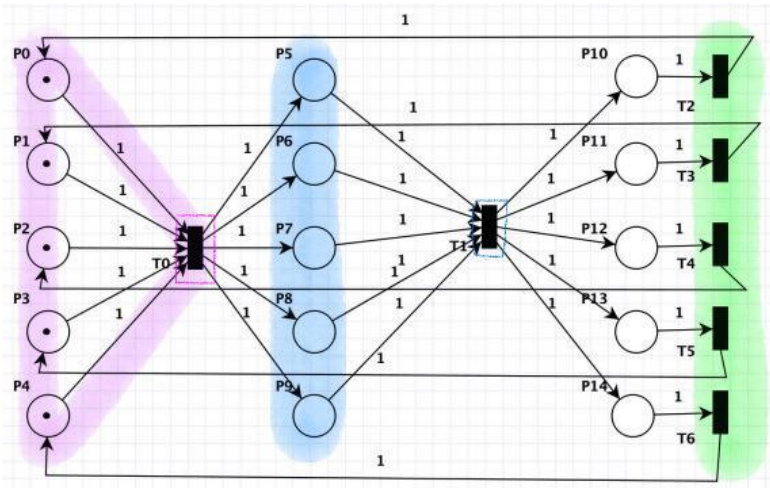


Exclusión mutua básica con 2 procesos → la red no puede llegar a un estado en donde haya tokens tanto en p3 como en p4 al mismo tiempo, entonces la exclusión se da entre p3 y p4. Estos dos estarían modelando los hilos de ejecución que se están excluyendo entre ellos. Entonces podemos pensar que nuestro código estaría encerrado en las 4 transiciones .

Ambos procesos p0 y p1 llegan a T0 y T1 al momento de representar la exclusión mutua, entonces puedo arrancar por cualquier proceso. Arranco por T0, que llega a T2 y se queda cuanto tiempo quiera. La red marca en ese momento que P3 está dentro de la región crítica y que al poder avanzar a T2, no se puede ejecutar T1. Vuelvo al proceso P0 y arranco de vuelta: Arranco de P1 a T1 a T3.

Banquero

Tengo N inversores que toman la plata al principio de la semana y la devuelven al final de ésta.

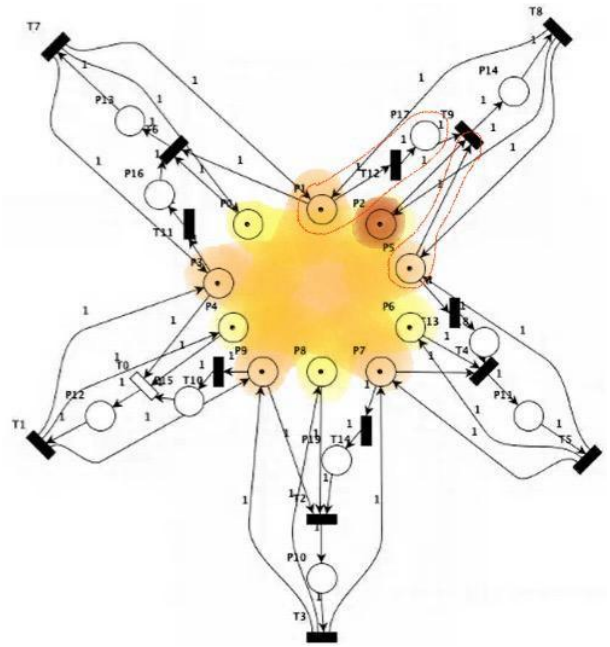


Habíamos resuelto el problema con ***barriers.***

Acá vemos que tenemos N lugares y una transición que recibe todos los lugares. Se puede disparar sólo si todos los places están listos (si todos los inversores empezaran la nueva semana). De esa misma transición no hace falta que salgan todos juntos, puede salir 1 solo si así quiere xq son independientes.

Tenemos nuevos places con cada uno de los inversores → acá es el momento en donde leen el salto. Tienen que estar todos listos, haber terminado de leerlos, hay una nueva barrera para la inversión. A partir de acá son todos independientes, en algún momento terminarán de hacer su inversión y quedan listos para empezar su nuevo ciclo.

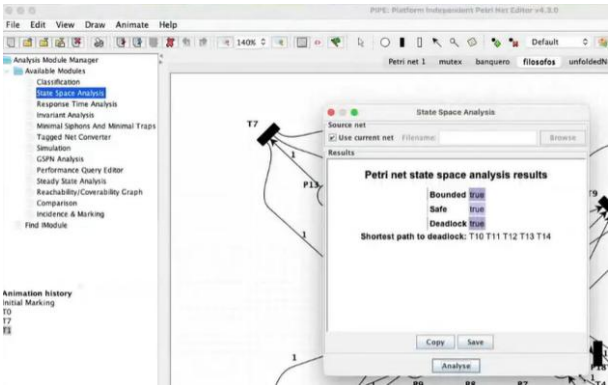
Filósofos



Idea inicial: Cada palito es un token. Cada filósofo es un place. Hay que modelar el deseo de comer, no todos quieren de una. Cuando quiere comer y tiene los 2 palitos, está listo para entrar comiendo.

Red: Están modelados en plan todos los filósofos sentados alrededor de la mesa con sus palitos. Simplifico con P2: en P2 quiere comer. de T12 a P17 es que toma el palito izquierdo, con el derecho no hace transición, avanza directamente. En P14 sería que está comiendo.

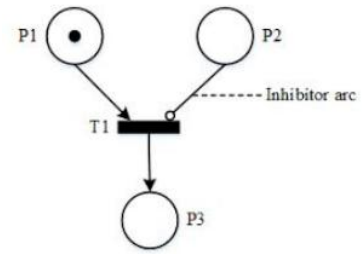
Está resuelto de esta forma para poder mostrar el deadlock. Entonces, en la aplicación PIPE (link en la diapositiva) puedo ver el camino hacia el deadlock - shortest path to deadlock: T10, T11, T12, T13, T14.

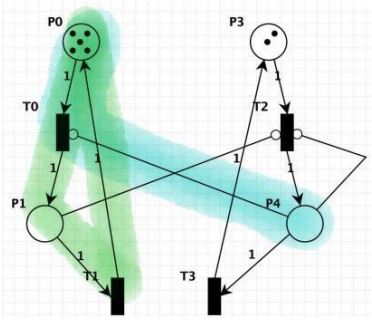


Lector - escritor

necesitamos extender los conceptos para soportar el problema

Arco inhibidor: La transición se habilita solamente si no hay ningún token.





- P0 conjunto de Lectores
- P3 conjunto de escritores

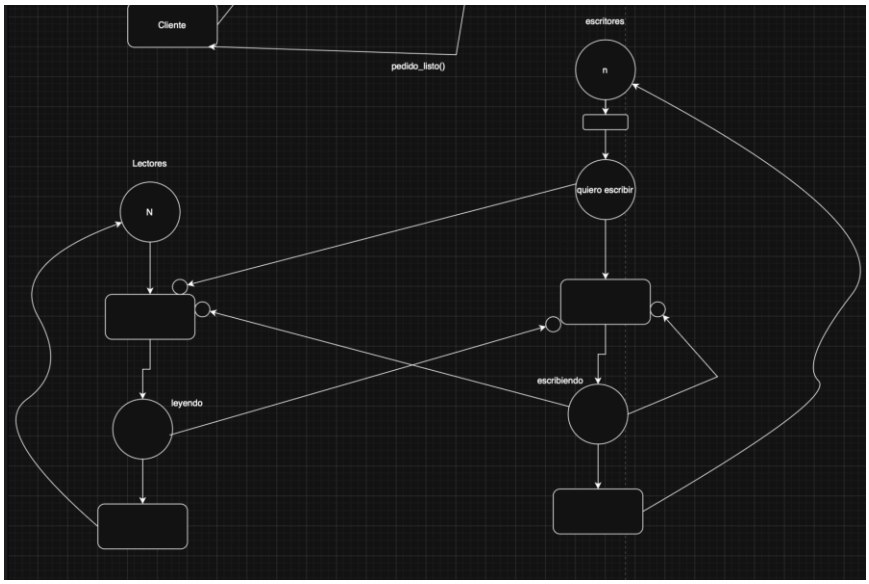
Uso de arco inhibidor

Cuando quiero leer, no tiene que haber nadie escribiendo. Si no hay nadie escribiendo, puede entrar gente a leer.

Cuando entro a leer, puede entrar más gente a leer pero ya nadie más puede entrar a escribir.

El arco inhibidor siempre va desde un estado hacia una transición.

Lector – escritor con preferencia de escritura



Channels y Mensajes

Herramienta para comunicar threads o tareas que conviven dentro de un mismo proceso.

¿Cómo son los distintos modelos de comunicación? Podemos clasificarlo por:

Tipo de comunicación:

Sincrónica:

- ✓ El emisor espera a que el receptor reciba el mensaje (bloqueante).
- ✓ Ej: send() y recv() sincronizados.

Asincrónica (con buffer):

- ✓ El emisor no espera; el mensaje se almacena en un buffer intermedio.

Direccionamiento ¿Cómo se decide a quién se envía el mensaje?

- Simétrico: ambos procesos conocen al otro explícitamente.
- Asimétrico: solo el emisor o receptor conoce al otro.
- Sin direccionamiento: los mensajes se matchean por contenido (como en Linda o tuplespaces).

Flujo de Datos

- Unidireccional: un solo sentido (emisor → receptor).
- Bidireccional: ida y vuelta (como un socket full-duplex).

Channels

Conectan procesos (o hilos). Tienen un nombre y son tipados (e.g., canal de int).

Pueden ser: Sincrónicos (no tienen buffer) o Asincrónicos (tienen buffer interno). Además, son usualmente unidireccionales.

Ejemplo productor-consumidor

channel of Integer ch

Productor	Consumidor

task body Producer is
  I : Integer
begin
  loop
  Produce(I);
  ch <= I;
  end loop
end Producer
	task body Producer is
	  I : Integer
	begin
	  loop
	  ch => I ;
	  Consume(I);
	  end loop
	end Consumer

ch <= I → envío, ch => I → recepción

En el productor tenemos un tipo de dato entero. Iniciamos el loop infinito en el que produce el elemento ***I***, obtiene algo en el ***I***, y envía el producto por el canal. Así infinitamente. No hay que pedir el valor, ni bloquear nada.

En el consumidor, tenemos el tipo de dato ***I*** entero, y un loop infinito en el que extrae el dato del canal. El ch⇒I está sincronizando, xq va a estar extrayendo el elemento del canal p y poniendolo en la variable; pero si no hay elemento, va a estar bloqueando el proceso a la espera de que haya un nuevo elemento. Además, la variable I es local al consumidor, entonces ejecutamos Consume y no tenemos ningún bloqueo ni nada xq es propia de ese hilo.

Selective input

Permite escuchar múltiples canales y reaccionar ante el primero que reciba algo:

```
either
  ch1 = > var1
or
  ch2 = > var2
or
  ch3 = > var3
```

Ejemplo Problema de los Filosofos con Channels

channel of Boolean forks[5]

Filósofo i	Fork i

task body Philosopher is
  dummy : Boolean
begin
  loop
  Think;
  forks[i] => dummy;
  forks[i+1] => dummy;
Eat;
forks[i] <= true;
forks[i+1] <= true;
  end loop;
end Philosopher
	task body Fork is
	  dummy : Boolean
	begin
	  loop
	  forks[i] <= true;
	  forks[i] => dummy;
	  end loop
	end Fork

Se define un canal de booleanos de 5 elementos (tenedores), suponiendo canales bidireccionales. ⇒ puedo usar los canales para mandar algo que no sea representativo, estoy enviando un dato pero no importa cual específicamente. Entrego algo dummy. los canales permiten el sincronismo.

en Filósofo tenemos el loop infinito. Primero piensa, después extrae el canal i por el dummy, también en i+1 ⇒ espero por los 2 tenedores, saco lo que hay y cuando puedo obtengo los 2. cuando tiene los 2 tenedores puede comer– luego los devuelve enviando algo por el canal (true). esto se repite infinitamente.

En fork, envía el true y recibe el elemento, con esto permite modelar los tenedores, enviando y extrayendo del canal.

Remote Procedure Calls

Permiten al cliente ejecutar funciones en un servidor localizado en otro procesador.

- Se requiere la implementación de stubs en ambos extremos
- Los stubs conforman interfaces remotas utilizadas para compilar cliente y servidor
- Localización de servicios
- Envio de parametros al servidor (parameter marshalling)

Un stub es un pedacito de código intermediario que actúa como un puente entre el cliente y el servidor en una llamada a procedimiento remoto (RPC). Cuando un proceso quiere llamar una función que no está en su máquina sino en otra, no puede simplemente hacer miFuncion(). Hace de representante local de esa función remota. Los stubs conforman interfaces remotas utilizadas para compilar cliente y servidor

Channels en Rust

Rust ofrece canales como mecanismo de comunicación segura entre hilos (threads), disponibles en el módulo **std::sync::mpsc.**

- Un canal tiene dos extremos: un emisor (**tx**) y un receptor (**rx**).
- Son unidireccionales, extremo de lectura va a un hilo y el de escritura a otro: Una parte del código invoca métodos sobre el transmisor, con los datos que se quiere enviar, otra parte chequea el extremo de recepción por la existencia de mensajes.
- Los canales son tipados (transmiten valores de un tipo específico).
- Son seguros para múltiples productores, pero un solo receptor: ***mpsc***. podemos clonar el extremo de transmisión donde están los productores y que distintos threads escriban en esos extremos clonados. No podemos clonar el extremo de recepción.
- **Transfieren ownership**: el dato enviado ya no puede usarse por el emisor después del envío. Evita problemas de acceso a la misma región de memoria en el mismo momento.
- Si querés múltiples productores, podés clonar el transmisor (el extremo del envío).

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
  let (tx, rx) = mpsc::channel();

  thread::spawn(move || {
    let val = String::from("Hola");
    tx.send(val).unwrap();
  });
  let received = rx.recv().unwrap();
  println!("Recibido: {}", received);
}
```

- cuando ejecuto el channel, le tengo que asignar un tipo. Aca no le asigno pues lo infiere de la primera vez que utilizo el extremo de transmisión → me impide utilizar ese canal para otro tipo de dato
- ***val*** alojada en el heap, se envía el string a través de tx.
- rx recibe el elemento a través del result. Se transfirió el ownership, ahora el receive es dueño de ese string

1. **mpsc::channel()** crea un canal unidireccional.
2. **tx.send(val)** envía un **String** a través del canal.
3. **rx.recv()** bloquea hasta recibir el mensaje.

Actores

Es un modelo basado en el pasaje de mensajes. El actor es la entidad principal del modelo. Son livianos, se pueden crear miles (en lugar de threads). Encapsulan comportamiento y estado. El actor supervisor puede crear otros actores hijo.

Compuesto por:

- Dirección: adonde enviarle mensajes. Es como su “nombre” o ID
- Casilla de correo (mailbox): un FIFO de los últimos mensajes recibidos.

Se encolan los mensajes que se reciben; entonces cada actor va a estar procesando de su mailbox cada mensaje de forma secuencial. De esta manera evitan las secciones críticas. Hay memoria local para cada actor que se va a mutar a partir del procesamiento secuencial de cada mensaje.

Son aislados de otros actores: no comparten memoria. Además, su estado privado solo puede cambiarse a partir de procesar mensajes. Pueden manejar un mensaje por vez.

Su estado interno es un **mutex implícito.** como solo pueden procesar un mensaje por vez, mientras el actor maneja ese mensaje no puede procesarse otro al mismo tiempo, y por ende no voy a poder tener dos cosas intentando de mutar el estado interno del actor al mismo tiempo.

Los actores se comunican entre ellos solamente usando *mensajes*, que son procesados por los actores de forma asincrónica. Los mensajes son estructuras simples inmutables. No hay condiciones de carrera entre los datos, no hay que poner locks entre los datos; una vez que el emisor tiene el mensaje que le va a enviar, el receptor no lo puede modificar, el mensaje es fijo, entonces no hay peligro de estar sobreescribiendo los datos.

Los actores pueden crear otros actores. Estos nuevos actores serán **actores hijos y el actual que los crea es un actor supervisor**. **No comparte memoria** con otros actores.

Actores en Rust (usando Actix, un framework de actores)

Usa tokio (runtime asincrónico) y futures (operaciones async y await).

Tiene un Arbiter. Un arbiter es básicamente un hilo del SO que: corre un event loop interno, aloja uno o más actores (son livianos) y administra tareas asincrónicas como futures, timers y el envío de mensajes entre actores. Posee un handler que se usa para enviar mensajes al actor.

Se ejecutan en un contexto de ejecución Context<A>. Cada actor tiene su propio Context<A> donde:

- Se gestiona su casilla de mensajes.
- Se ejecuta su lógica.
- Puede detenerse o reiniciarse.

Para crear un actor:

1. Crear una estructura (struct) para representar al actor.
2. Implementar el trait Actor para esa estructura.
3. Definir un tipo de mensaje (implementando el trait Message).
4. Implementar un handler para ese mensaje usando el trait Handler<M>.
5. Spawnear el actor con .start(), que lo ejecuta dentro de un Arbiter.

```
use actix::prelude::*;

// 1. Definimos el actor
struct MyActor;

// 2. Implementamos el trait Actor
impl Actor for MyActor {
    type Context = Context<Self>;
}

// 3. Definimos un mensaje
struct Ping;

// 4. Implementamos el trait Message para Ping
impl Message for Ping {
    type Result = &'static str;
}

// 5. Implementamos el handler para manejar Ping
impl Handler<Ping> for MyActor {
    type Result = &'static str;

    fn handle(&mut self, _msg: Ping, _ctx: &mut Context<Self>) ->
Self::Result {
        "Pong!"
    }
}
```

```
// 6. Enviamos un mensaje al actor
#[actix::main]
async fn main() {
    let addr = MyActor.start(); // Spawn del actor
    let res = addr.send(Ping).await.unwrap(); // Enviamos mensaje
    println!("Respuesta del actor: {}", res);
}
```

Implemento el trait actor y le paso el type Context.

Ahora tengo que implementar el handler para cada tipo de mensaje q quiero recibir → implemento Handler ping para mi actor y le indico que el Result (que es el tipo asociado del trait) es un usize.

Implemento el método handle donde va el comportamiento del actor cuando recibe el mensaje. Recibe referencia mutable al mismo xq se puede mutar el estado interno con el resultado del procesamiento, el ownership del mensaje de tipo Ping y el Context mutable sobre el mismo actor.

Devuelve un Result, que no es el de Rust sino que el de Self (el usize).

Ciclo de vida de un Actor

Estado	Descripcion
Started	El actor se acaba de inicializar. Podés sobreescribir `started(&mut self, ctx: &mut Context<Self>)` si querés ejecutar algo apenas arranca. A partir de acá, el contexto del actor está disponible
Running	Estado normal. Está vivo, recibiendo y procesando mensajes. Puede estar en este estado de forma indefinida.
Stopping	Puede entrar en este estado si se llama a `ctx.stop()`, si ningún otro actor lo referencia, o si su contexto quedó vacío.
Stopped	Desde el estado anterior no modifi có su situación. Fin del ciclo. No procesa más mensajes.

Dirección de un Actor

Los actores son referenciados únicamente por la dirección.

```
struct MyActor;
impl Actor for MyActor {
  type Context = Context<Self>;
}

let addr = MyActor.start();
```

Los structs pueden tener o no atributos. Tengo que implementar traits para que sea un actor de Actix.

- El Trait tiene el tipo de dato Context asociado al propio actor.
- Lo invoco con el start, devuelve la dirección del actor. Es la única forma con la que puedo referenciar actores.

Uso de mensajes Tipados para actores

Un actor se comunica con otro enviando mensajes. Todo mensaje que quieras enviar a un actor debe implementar el trait Message. Este trait define el tipo del resultado que devuelve el actor cuando procesa ese mensaje.

```
struct Ping;
impl Message for Ping {
    type Result = Result<bool, std::io::Error>;
}
```

Formas de enviar un mensaje

Cada una de estas formas se ejecuta sobre la dirección del actor

Método	Descripción
addr.do_send(msg)	Envío "fire-and-forget": no espera respuesta ni devuelve Result. Si el actor está caído o la casilla cerrada, el mensaje se descarta. Ignora errores en el envío del mensaje.
addr.try_send(msg)	Envío sincrónico inmediato. Falla si la casilla está llena o cerrada. Retorna Result<(), SendError>.
addr.send(msg)	Envío asincrónico y seguro. Devuelve un Future que podés await para obtener la respuesta del actor.

Do_send se usa cuando no me interesa saber la rta; no es que siempre debemos usar await.

Contexto

El tipo Context<A> (donde A es tu actor) representa el contexto interno de ejecución del actor. Es lo que:

- Mantiene su estado de ejecución.
- Gestiona su casilla de mensajes (mailbox). Los mensajes llegan a la casilla primero, luego el contexto de ejecución llama al handler específico.
- Permite ejecutar acciones asincrónicas o detener el actor.
- Es pasado como parámetro a cada handler.

Arbiter

Provee el contexto de ejecución asincrónica para los actores, funciones y futuros. Alojan el entorno donde se ejecuta el actor.

Realizan varias tareas: crear un nuevo Thread del SO, ejecutar un event loop, crear tareas asincrónicas en ese event loop.

En actix, los handlers de los actores se ejecutan como operaciones asincrónicas → esto se tiene que tener en cuenta, para que las operaciones no tengan alta demanda de procesamiento, xq si rompe algo bloquea el executor.

Comunicación Distribuida – Cliente Servidor

Desafío de pasar de Channels (comunicación centralizada) a Sockets (comunicación distribuida)

- Espacios de memoria separados
- Serialización y necesidad de delimitar un protocolo
- Conexiones y desconexiones
- Perdida y/o reordenamiento de paquetes
- Relojes no sincronizados
- Entidades unresponsive
- Identificación de que alguien se conectó
- Tiempo de viaje de un mensaje
- No se aseguro que los datos lleguen a destino

Modelo Cliente Servidor

Sockets

Permiten la comunicación entre dos procesos diferentes ya se en la misma máquina o en dos máquinas diferentes.

Se usan en aplicaciones que implementan el modelo cliente - servidor:

- Cliente: es activo porque inicia la interacción con el servidor
- Servidor: es pasivo porque espera recibir las peticiones de los clientes. Es pre-existente ya que debe estar funcionando antes de querer conectarme al sitio web de la facultad

Arquitectura Cliente Servidor

Hay diferentes formas de diagramar la arq C-S:

- Arquitectura a dos niveles: El cliente interactúa directamente con el servidor
- Arquitectura de tres niveles: ***middleware.*** Hay una capa de software ubicada entre el cliente y el servidor que provee principalmente seguridad y balanceo de carga. Pueden proveer tambien servicios sobre middlewares para comunicar sistemas operativos diferentes.

Tipos de servidor:

- Iterativo: Atiende las peticiones de a una a la vez
- Concurrente: Puede atender varias peticiones a la vez

En el modelo de Redes, hay diferentes tipos de servicios que pueden ofrecerse, y según eso, se pueden utilizar diferentes tipos de sockets:

Tipos de Servicio

- *Sin conexión*: Los datos se envían al receptor y no hay control de flujo ni de errores.
- *Sin conexión con ACK*: Por cada dato recibido, el receptor envía un acuse de recibo conocido como ACK.
- *Con conexión (Tres fases):* establecimiento de la conexión, intercambio de datos y cierre de la conexión. Hay control de flujo y control de errores. Van a estar reservando recursos para entablar y llevar a cabo esa comunicación.

Sockets en UNIX

- STREAM → protocolo TCP (entrega garantizada del flujo de bytes)
- DATAGRAM → protocolo UDP (entrega no garantizada; servicio sin conexión)
- RAW → permiten a las aplicaciones enviar paquetes IP
- SEQUENCED PACKET → similares a stream sockets, pero preservan los delimitadores de registro; utilizan el protocolo SPP (Sequenced Packet Protocol). Son más para el uso de redes privadas (nosotros no los vamos a usar)

Cómo se usan (ejemplo cliente)

1. Creación de un socket `int socket ( int family,int type,int protocol );` para reservar recursos. Retorna el file descriptor del socket en caso de éxito o -1 en caso de error
2. Inicia una conexión con el servidor `int connect ( int sockfd,struct sockaddr *serv_addr,int addrlen );`. Lanza la conexión TCP hacia el servidor (handshake). Esta acción es **bloqueante** para nuestro programa.
3. Luego contamos con funciones de:
    - Read(): lee bytes del socket
    - Write(): escribe bytes en el socket
    - Send() y recv(): para comunicación usando *Stream* sockets – es lo mismo que read write pero son una "distinción" de una extensión para sockets, porque read y write son de los file descriptors, pero no hacen nada diferente.
    - Funciones sendto() y recvfrom() para *datagram sockets*
    - Cierre de conexión con close() – liberación de la conexión

Cómo se usan (ejemplo servidor)

Este tiene que prepararse *previamente* hasta que llegue el cliente. Por lo tanto, su socket lo convierte en un **socket pasivo.**

1. Creación de un socket `int socket ( int family,int type,int protocol );` para reservar recursos. Retorna el file descriptor del socket en caso de éxito o -1 en caso de error
2. Se le asigna una dirección local al socket `int bind ( int sockfd,struct sockaddr *my_addr,int addrlen );`

   Es imprescindible asignarle una dirección al servidor, porque el cliente tiene que saber a donde conectarse. Ahora, no es necesario hacer esto para el cliente porque una vez que el cliente le escribe al servidor una solicitud, su dirección viaja en esa información, el servidor ya sabe entonces a que dirección contestarle, y luego el SO le asigna un puerto aleatorio (uno que este libre) para que el cliente reciba la información.

3. Se convierte el socket en pasivo: `int listen ( int sockfd,int backlog );`. Esto hace que el servidor se quede escuchando solicitudes de los clientes en esa dirección. Va a ir aceptando conexiones y va a ir almacenando en una cola las conexiones con clientes, que cada una va a estar en diferentes etapas del handshake hasta que es finalmente establecida.

4. Retorna la siguiente conexión completa de la cola de conexiones, Extrae de la cola de conexiones ya establecidas con clientes una conexión:
   `int accept ( int sockfd,struct sockaddr *cliaddr,socklen_t *addrlen );` Retorna el file descriptor del cliente en caso de éxito; se utiliza para comunicarse con el cliente o -1 en caso de error. Es una **operación bloqueante.** Si no hay ninguna operación nuestro programa queda a la espera de que se conecte algún cliente.

5. Luego contamos con funciones de:
   - Read(): lee bytes del socket
   - Write(): escribe bytes en el socket
   - Send() y recv(): para comunicación usando *Stream* sockets
   - Funciones sendto() y recvfrom() para *datagram sockets*
   - Cierre de conexión con close() – liberación de la conexión

El cierre de conexión en TCP es unilateral. Cualquiera de los dos, cliente-servidor puede enviar un mensaje. Cada parte puede decidir cerrar la conexión. Luego, el otro le manda el close propio, como para "clarificar" el cierre. Por eso se hace Close →, se recibe Close Ack ← se recibe Close ← y se envía un último Ack →

Sockets en Rust

Se usa la biblioteca std::net

Cómo se usan (ejemplo servidor TCP)

1. Asociar el socket a una dirección: `pub fn bind<A: ToSocketAddrs>(addr: A) -> Result<TcpListener>` y el listener retornado está listo para aceptar conexiones.

2.1 Sobre la estructura TcpListener se obtienen conexiones establecidas. El método incoming retorna un iterador que devuelve una secuencia de streams de tipo TcpStream

```
for stream in listener.incoming() {
let stream = stream.unwrap();
println!("Conexion establecida!");
}
```

cada stream representa una conexión abierta entre el cliente y el servidor. La iteración es sobre "intentos de conexiones". Puede retornar Err. Bloquea hasta que empieza a recibir resultados

2.2 Sino, se pueden obtener conexiones establecidas con accept

`pub fn accept(&self) -> Result<(TcpStream, SocketAddr)>` El método accept obtiene una conexión establecida de un listener. Bloquea y devuelve directamente el result, no el iterador.

3. Leer los datos del socket con ***read.*** `fn read(&mut self, buf: &mut [u8]) -> Result<usize>`
4. Escribe una respuesta con write.

```
stream.write(response.as_bytes()).unwrap();
stream.flush().unwrap();
```

El método flush realiza una espera, previniendo que el programa continúe sin haber escrito en la conexión todos los bytes

Acá en rust, el crear un socket y hacer el bind es un único paso.

Cómo se usan (ejemplo cliente TCP)

1. El cliente debe establecer la conexión con el servidor. Puede construir la dirección de destino a partir de:
   a. Una dirección IP:

```
let socket =
  SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)),
    8080);
```

```
let mut addrs_iter = "localhost:443".to_socket_addrs()
    .unwrap();
```

b. Un nombre con el método to_socket_addrs: `pub fn connect<A: ToSocketAddrs>(addr: A) -> Result<TcpStream>` . Este método abre una conexión al host remoto. Si se le envía un array de direcciones, intenta conectarse a cada una, hasta lograrlo

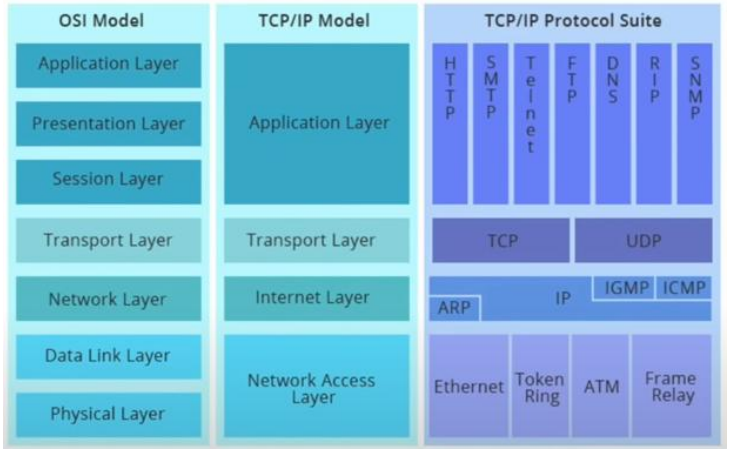
2. Se conecta al servidor:
3. Ejecuta los métodos read y write para enviar y recibir datos.

Cierre de conexión

El cierre de la conexión TCP puede ser realizado de forma individual. La conexión establecida con TcpStream se cierra cuando el valor ejecuta drop. Esto inicia el envío del mensaje close de TCP.

El método shutdown pude cerrar el extremo de escritura, de lectura o ambos.

Modelo OSI en Redes



El **modelo OSI (Open Systems Interconnection)** es un marco teórico creado por la **ISO (Organización Internacional de Normalización)** que divide el proceso de comunicación en redes en **7 capas**, desde el nivel físico hasta el nivel de aplicación. Cada capa tiene funciones específicas y se comunica con la capa superior e inferior.

Nº	Capa	Objetivo principal	Características
7	**Aplicación**	Proporcionar servicios directos al usuario final (como HTTP, FTP, SMTP).	Interfaces para software de usuario, acceso a servicios de red.
6	**Presentación**	Traducir datos entre el formato de la red y el formato de la aplicación.	Codificación, compresión, encriptación de datos.
5	**Sesión**	Controlar el diálogo entre aplicaciones, estableciendo, gestionando y terminando sesiones.	Control de conexión, sincronización.
4	**Transporte**	Asegurar la entrega completa, correcta y ordenada de los datos.	Protocolos como TCP (fiable) y UDP (rápido pero no fiable).
3	**Red**	Determinar la ruta y direccionamiento de los paquetes en una red.	IP, enrutamiento, subredes.
2	**Enlace de datos**	Proporcionar una conexión libre de errores entre dos nodos adyacentes.	MAC, control de errores, control de flujo.
1	**Física**	Transmitir bits crudos a través del medio físico.	Voltajes, cables, conectores, señales eléctricas u ópticas.

Concurrencia Distribuida I

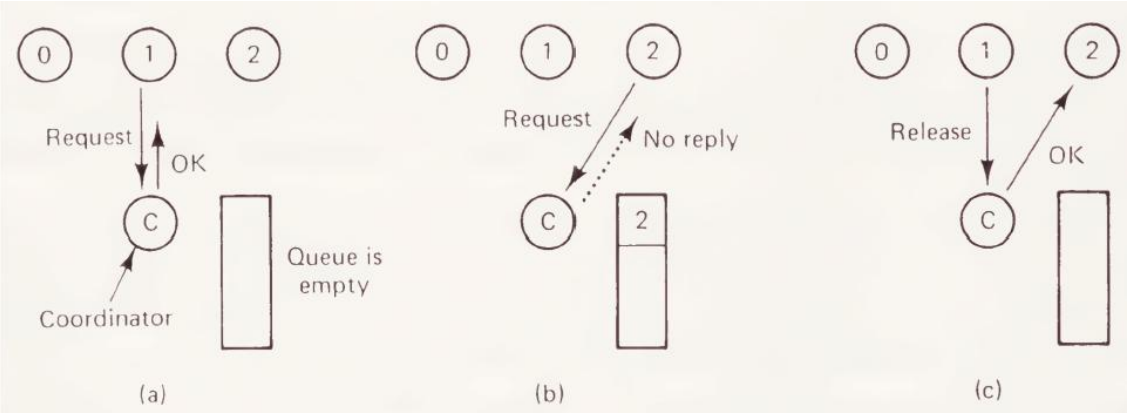
Vamos a llevar el **sincronismo y la comunicación** de datos a un sistema distribuido (diferentes computadoras que se intentan comunicar)

Exclusión Mutua Distribuída

Vamos a analizar tres algoritmos. Tenemos nuestro software en un conjunto de computadoras que funcionan coordinadamente (por ejemplo un cluster de servidores) y requieren que se coordine cierta are de nuestro programa que debe ejecutarse de manera exclusiva.

Algoritmo Centralizado

Un proceso es elegido coordinador (no importa cual; es el dueño de los bloqueos). Cuando un proceso quiere entrar a la SC, envía un mensaje al coordinador. Si no hay ningún proceso en la SC, el coordinador envía OK; si sí hay alguien, el coordinador no envía respuesta hasta que se libere la SC.



Obviamente debería haber algún tipo de chequeo de mensajes para verificar que si hay demoras en las respuestas no necesariamente es porque el recurso no está disponible, sino que capaz hay problemas en la red.

Antes de que se ejecute este algoritmo centralizado, se debe ejecutar una fase inicial de elección de coordinador.

Ventajas

- Garantiza la exclusión mutua: el coordinador sólo deja entrar a la sección crítica a un proceso a la vez
- Es justo: los pedidos son garantizados en el orden en el que llegan al coordinador y ningún proceso muere de starvation
- El esquema es simple de implementar
- La conexión solo requiere tres mensajes: request, grant, release
- Puede ser utilizado para alojamiento de recursos más generales

Desventajas

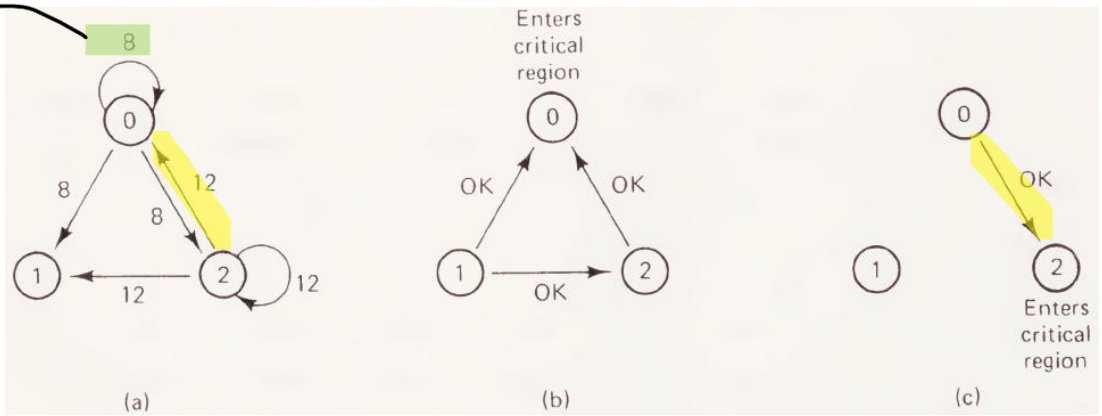
- El algoritmo requiere de mantener a un líder (y si se cae reestablecerlo o seleccionar otro), lo cual no necesariamente es tan trivial.
- El coordinador es un único punto de falla (single point of failure) → si se cae, todo el sistema lo hace
- Si los procesos se bloquean después de hacer un request, no pueden distinguir entre un coordinador caído de un permiso denegado ya que no se recibe ningún mensaje
- En un sistema grande, un sólo coordinador puede conllevar a un cuello de botella

Algoritmo Distribuido

No requiere de un líder ya seleccionado. Cuando un proceso quiere entrar en una sección crítica, construye un mensaje con el nombre de la sección crítica, el número de proceso y el timestamp. Al recibir el mensaje (el mensaje se broadcastea a todos los procesos a través de la red, y recibe las respuestas):

1. Si el receptor no está en la CS y no quiere entrar, envía OK
2. Si el receptor está en la CS, no responde y encola el mensaje. Cuando sale de la SC, envía OK
3. Si el receptor quiere entrar en la CS, compara el timestamp del mensaje que le llega con el mensaje que le envío a todos los demás y gana el menor (gana el proceso que decidió que necesita entrar antes a la sección crítica).
    a. Si el proceso emisor inicial tiene el timestamp más pequeño, el receptor envía un mensaje de OK.
    b. Si el proceso receptor tiene el timestamp más pequeño, encola el pedido y no envía nada al emisor.

timestamp



Luego de enviar el pedido de permiso para entrar a una sección crítica, el proceso espera a que todos los demás le den permiso. Cuando se reciben todos, entra a la sección crítica. Cuando sale de la sección crítica, envía OK a todos los procesos que encoló y los borra de la cola.

El que tiene menor timestamp es un proceso que, en general, es más vieja que otra que tiene mayor timestamp. Estadisticamente, esa operación puede haber ejecutado mas operaciones que una que tiene mayor timestamp, y no queremos que espere bloqueada, se prefiere que termine antes.

ACA "2" le envió un mensaje de que quería entrar en la Seccion Critica pero el timestamp de "0" es menor. Entró "0" y no le respondió a 2; encolo el mensaje. Una vez que sale le responde OK

Conviene desarrollarlo con UDP porque sino tendría que mantener vivas conexiones punto a punto entre cada par de procesos

Ventajas

- Garantiza la exclusión mutua sin deadlock ni starvation

Desventajas

- Se reemplaza el single point of failure por n points of failure → si algún proceso cae, falla en responder los pedidos y el silencio será interpretado incorrectamente como una denegación de permiso bloqueando todos los intentos subsecuentes de otros procesos para entrar a las secciones críticas.

  Se puede arreglar respondiendo todos los mensajes, tanto de autorización como de denegación de permiso. Siempre que un pedido o una respuesta estén perdidos, el emisor

sigue intentando establecer una conexión hasta que se lance un timeout y el emisor concluya que el destinatario está caído. Luego de que un pedido haya sido denegado, el emisor debe bloquearse esperando por un mensaje OK.

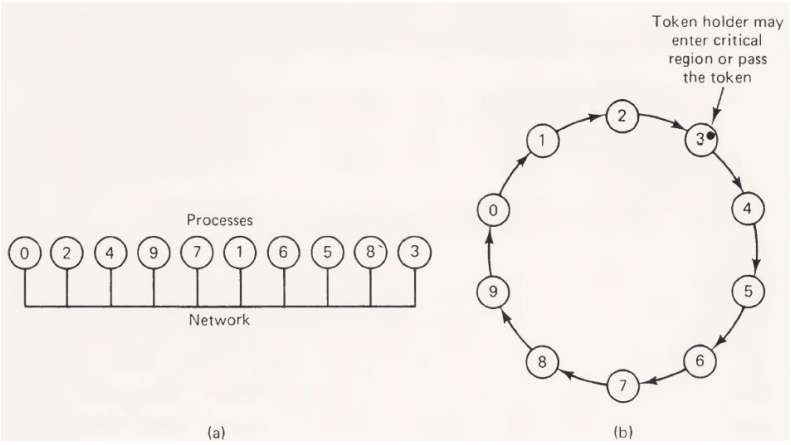
- Tiene que existir un grupo de comunicación primitivo, o cada proceso debe mantener una lista de miembros incluyendo los procesos entrantes, salientes y caídos. El método funciona mejor con grupos pequeños de procesos que nunca cambian sus grupos.
- Empeora el cuello de botella → todos los procesos están envueltos en todas las decisiones que incluyen entrar o no a una sección crítica.

Se pueden hacer algunas mejoras pero el algoritmo es bastante malo, aún contra el centralizado

Algoritmo Token Ring

Se conforma un anillo mediante conexiones punto a punto entre los procesos. Cada uno conoce y puede enviarle mensajes al siguiente, y recibe mensajes del anterior. Al inicializar, el proceso 0 recibe un token que va circulando por el anillo. ***Sólo el proceso que tiene el token puede entrar a la SC***.

Cuando el proceso sale de la SC, continúa circulando el token. **El proceso no puede entrar a otra SC con el mismo token.** Es decir, que tiene que esperar a recibirlo nuevamente. En general el token se define según la sección crítica.



Ventajas

- Sólo un proceso tiene el token en cada momento, de modo que sólo un proceso puede entrar a la región crítica.
- Como el token pasa a lo largo de todo el anillo, no puede haber starvation.

Deventajas

- Una vez que un proceso decide que quiere entrar a una región crítica, tiene que esperar a que le den el token.
- Este proceso significa que vamos a tener mensajes en la red constantemente (pues tiene que circular el token)
- Si el token se pierde tiene que ser regenerado, y detectar la pérdida es difícil: no se puede saber con exactitud el tiempo que le toma a un proceso tener el token. Un proceso caído es detectado cuando no recibimos el **ACK** del receptor

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

El algoritmo centralizado es el más simple y el más eficiente

Concurrencia Distribuida II

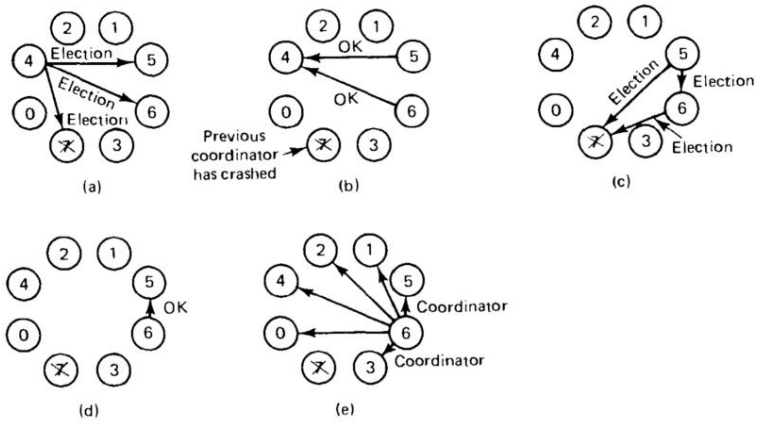
Varios algoritmos requieren de un coordinador con un rol especial (ej: algoritmos de exclusión mutua distribuida). En general, no es importante cuál es el proceso, sino que debe cubrirse el rol.

- Se asume: todos los procesos tienen un ID único, se ejecuta un proceso por máquina y conocen el número de los demás procesos.
- El objetivo: cuando la elección comienza, concluye con un elegido.
- Se asegura que cuando una elección empieza, termina con todos los procesos
- aceptando quién será el nuevo coordinador.

Algoritmo Bully

Cuando un proceso P nota que el coordinador no responde, inicia el proceso de elección:

1. P envía el mensaje ELECTION a todos los procesos que tengan número mayor
2. Si nadie responde, P gana la elección y es el nuevo coordinador
3. Si contesta algún proceso con número mayor, éste continúa con el proceso y P finaliza su trabajo.
4. El nuevo coordinador se anuncia con un mensaje COORDINATOR



El 4 detecto que el líder (7) se cayó, entonces 4 le va a mandar mensaje a todos los ids mas grandes que el (5, 6 y 7, aunque se haya caído, simplemente no le contesta). 5 y 6 le responden OK y siguen el mismo proceso. Como al 6 no le responde nadie, el 6 es el líder, así que les envía a todo el resto el mensaje coordinator.

Siempre gana el proceso con mayor número.

Un proceso puede recibir un mensaje de elección en cualquier momento de alguno de los procesos con números más pequeños que él. Cuando el mensaje llega, el receptor envía un mensaje de OK al emisor para indicar que está vivo y se va a encargar de la elección.

Eventualmente todos los procesos terminan la elección menos uno, que será el nuevo coordinador. El coordinador anuncia su victoria enviando a todos los demás un mensaje avisándoles que él es el nuevo coordinador

Si un proceso previamente caído vuelve a la red, comienza una elección. Si sucede que tiene el número más grande de procesos ejecutando, va a ganar la elección y tomar el trabajo del coordinador.

Este es un algoritmo **bastante costoso** y puede generar particiones (ej: si 5 no se puede comunicar ni con 6 ni 7, se autoproclama líder, aislado.)

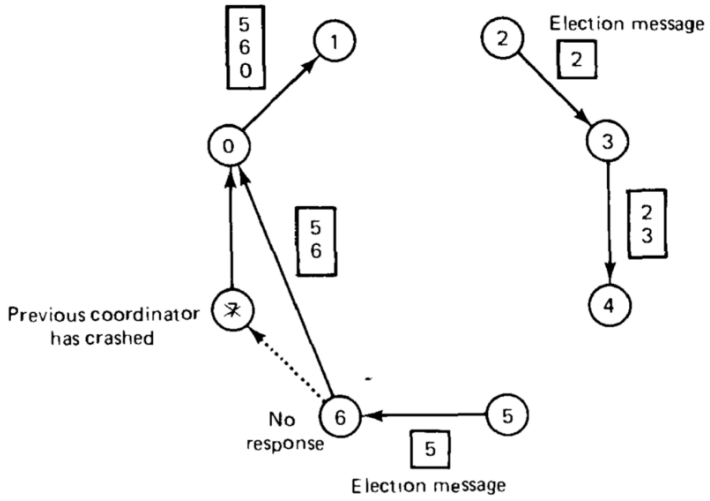
Algoritmo Ring

Los procesos están ordenados lógicamente; cada uno conoce a su sucesor

1. Cuando un proceso nota que el coordinador falló, arma un mensaje ELECTION que contiene su número de proceso y lo envía al sucesor. Si el sucesor esta caído, el emisor lo salta y envía el mensaje al siguiente miembro del anillo, o el siguiente a él hasta que algún proceso en ejecución sea localizado

! este paso 1 es necesario para tener una elección estable. Si yo directamente digo: se cayó el líder y me autoproclamo nuevo líder, puede haber otros procesos que concurrentemente detectaron lo mismo y también arrancan a circular que son el nuevo líder. **Esto NO sería una solución consistente.**

2. El proceso que recibe el mensaje agrega su número de proceso a la lista dentro del mensaje y lo envía al sucesor
3. Cuando el proceso original recibe el mensaje, ve que contiene su propio número, entonces envía un mensaje de tipo COORDINATOR. El nuevo coordinador es el proceso de mayor número de la lista. La lista de procesos también se envía para informar quienes están en el nuevo anillo.
4. Cuando este mensaje finaliza la circulación, se elimina del anillo y los procesos vuelven a trabajar.



Hay dos mensajes de election dando vuelta pero igual van a converger al mismo líder.

Transacciones

Precondiciones

para que se puedan ejecutar con éxito:

1. Existencia de recursos: existen datos que la transacción quiere leer/modificar
2. Bloqueo de recursos: adquisición de Locks sobre los recursos que la trx quiere modificar
3. Condiciones de estado: el estado del sistema garantiza la coherencia y consistencia antes de que se ejecute la trx
4. Validación de reglas de negocio: verificación que la trx cumple con las reglas establecidas

Requisitos

Requisitos de un sistema informático para poder trabajar con transacciones:

- El sistema está conformado por un conjunto de **procesos independientes**; cada uno puede fallar aleatoriamente.
- Los errores en la comunicación son manejados transparentemente por la capa de comunicación.
- Nuestros servidores están sustentados sobre **Storage estable**: una vez que los datos se graban en el medio de almacenamiento, los datos permanecen ahí.
  - Se implementa con discos
  - La probabilidad de perder los datos es extremadamente pequeña

Primitivas

Las transacciones son operaciones/funciones que implementan las siguientes primitivas:

BEGIN TRANSACTION: marca el inicio de la transacción

END TRANSACTION: finalizar la transacción y tratar de hacer commit (se "trata" de hacer commit porque puede ser que las condiciones hayan cambiado y que no se pueda efectivizar)

ABORT TRANSACTION: finalizar forzadamente la transacción y restaurar los valores anteriores

READ: leer datos de un archivo u otro objeto

WRITE: escribir datos a un archivo u otro objeto

Propiedades ACID

- Atómicas (Atomicity): la transacción no puede ser dividida. Confirmamos la finalización de una transacción *completa,* no por pedacitos.
- Consistentes (Consistency): la transacción cumple con todos los invariantes del sistema (= respeta las restricciones).
  Por ejemplo si estoy implementando un sistema bancario, una posible transacción del sistema es *realizar transferencia* entre cuentas. *Lleva las trx de un estado valido a otro.*

- Aisladas o serializadas (Isolation): las transacciones concurrentes no interfieren con ellas mismas.
  En un sistema concurrente, el resultante es como si la soperaciones se hubieran ejecutado una después de la otra de forma serializada.
- Durables (Durability): una vez que se commitean los cambios, son permanentes
  Esto se sustenta sobre la premisa que nuestro sistema almacena la información en discos.

Excepción a la propiedad de durabilidad: transacciones anidadas no son durables. Puede ocurrir que la transacción padre, por ejemplo, no pueda ejecutar el commit, y eso hace que se deshagna todos los cambios de esa transacción, y todos los cambios de la transacción hijo. Por lo tanto, los cambios de la transacción hijo no fueron durables, pero porque fue contenida dentro de otra más grande.

Algoritmos de implementación de Transacciones

Private Workspace

Al iniciar una transacción, el proceso recibe una copia de todos los archivos a los cuales tiene acceso esa transacción. Hasta que hace commit, el proceso trabaja con una copia local y al hacer commit, se persisten los cambios. Se va a revisar si no hay conflictos (otro proceso realizó modificaciones) y si no los hay, se persisten y almacenan los cambios.

Desventaja: extremadamente costoso salvo por optimizaciones

Ventaja: isolacion, y tiene menos overhead que el WAL, porque no requiere el mismo nivel de registro detallado.

Ejemplo: Google Docs

Writeahead Log

Es escribir un archivo log de antemano. Se implementa en muchas BDD relacionales, como PostgreSQL.

Los archivos se modifican in place, pero se mantiene una lista de los cambios aplicados (primero se escribe la lista y luego se modifica el archivo). Al commitear la transacción, se escribe un registro commit en el log

Si la transacción se aborta, se lee el log de atrás hacia adelante para deshacer los cambios (rollback)

Ventajas:

> Es robusto frente a, por ejemplo, problemas eléctricos. Si se vienen modificando archivos y ocurre un desperfecto, se lo vuelve a colocar en un estado válido llevándolo al inicio de la transacción que quedó cortada. (esto lo vimos en bdd y no me acuerdo un choto)
>
> Durabilidad: garantiza qu los datos en la BDD sean duraderos incluso en caso de fallas
>
> Recuperación: facilita la recuperación frente a fallas, porque se puede usar el log para recuperar el estado/hacer rollback

Desventajas: se debe escribir dos veces en el log antes de aplicarse los cambios.

Ejemplo: PostgreSQL

Commit de dos fases

El coordinador es aquel proceso que ejecuta la transacción. Escribe datos en su base de datos y se los comunica a los demás.

Fase 1: P**repare**

1. El **coordinador escribe** prepare en su log interno y envía el mensaje prepare al resto de los procesos
2. Los procesos que reciben el mensaje ***escriben ready*** en el log y envían ready al coordinador

Fase 2: **Commit o Abort**

1. El coordinador hace los cambios y envía el mensaje commit al resto de los procesos
2. Los procesos que reciben el mensaje escriben commit en el log y envían finished al coordinador

Ejemplo: Sistema de pagos

Algoritmos de Control de Concurrencia

Controlar la concurrencia está relacionado con la sincronización y el ordenamiento de la ejecución de las transacciones, así también como evitar los deadlocks.

Lockeo: Two-Phase locking

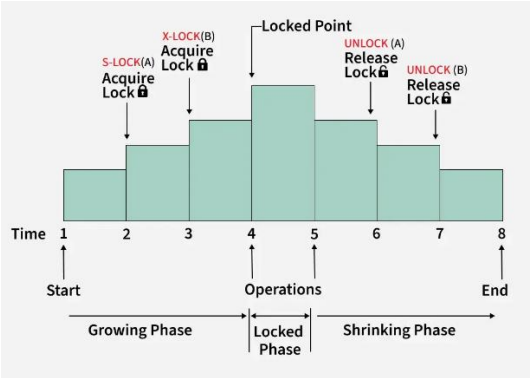
1. Fase de expansión: se toman todos los locks a usar
2. Fase de contracción: se liberan todos los locks (no se pueden tomar nuevos locks)

Ventajas: garantiza propiedad serializable para las transacciones

Desventajas:

- pueden ocurrir deadlocks: si tengo transacciones distintas que están pidiendo un conjunto de Locks ya tengo una gran chance de tener deadlocks

Strict two-phase locking: existe una versión donde la contracción ocurre después del commit de la transacción.



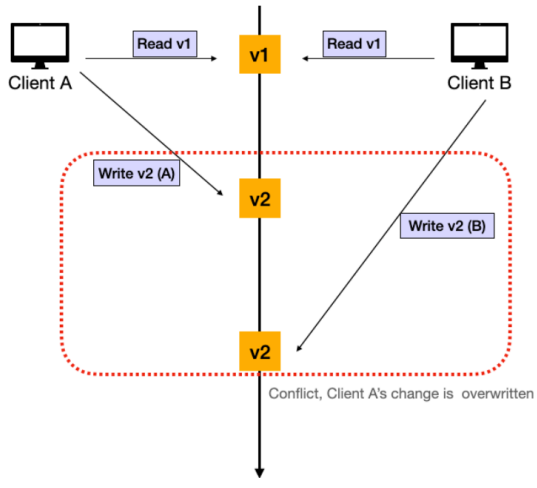
Concurrencia Optimista (GitHub)

Los cambios se hacen de manera optimista, sin lockeos, porque se asume que no va a haber conflictos.

- El proceso modifica los archivos sin ningún control, esperando que no haya conflictos
- Al commitear, se verifica si el resto de las transacciones modificó los mismos archivos. Si es así, se aborta la transacción

Ventajas: Libre de deadlocks, por lo que favorece el paralelismo.

Desventajas: Rehacer todo puede ser costoso en condiciones de alta carga.



En general es bueno cuando tenemos una proporción mucho más grande de consultas que de modificaciones, porque no vamos a tener muchos casos donde se intenten editar los mismos archivos.

Timestamps

Existen timestamps únicos globales para garantizar orden (ver algoritmo de relojes de Lamport). Cada archivo tiene dos timestamps: lectura y escritura y qué transacción hizo la última operación en cada caso. Cada transacción al iniciarse recibe un timestamp.

Se compara el timestamp de la transacción con los timestamps del archivo:

- Si es mayor, la transacción está en orden y se procede con la operación
- Si es menor, la transacción se aborta (osea cuando la transacción es más vieja que la última modificación del archivo)

Al commitear se actualizan los timestamps del archivo

Deadlocks

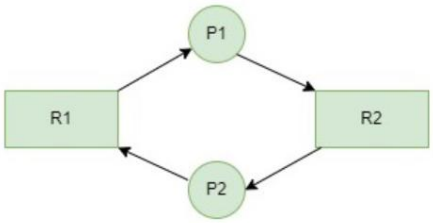
Algoritmos referentes a deadlocks, y cómo prevenir deadlocks en transacciones

Algoritmos de Detección de Deadlocks

Algoritmo Centralizado

El proceso coordinador mantiene el grafo de uso de recursos. Los procesos envían mensajes al coordinador cuando obtienen / liberan un recurso y el coordinador actualiza el grafo. **El deadlock ocurre cuando se presenta un ciclo el el grafo**. Voy a estar esperando por algo que nunca vot a recibir porque viene de otro que está esperando que yo lo libere.

- *Problema:* los mensajes pueden llegar llegar desordenados y generar falsos deadlocks.
- *Posible solución:* utilizar timestamps globales para ordenar los mensajes (algoritmo de Lamport)



R=resource, p=process

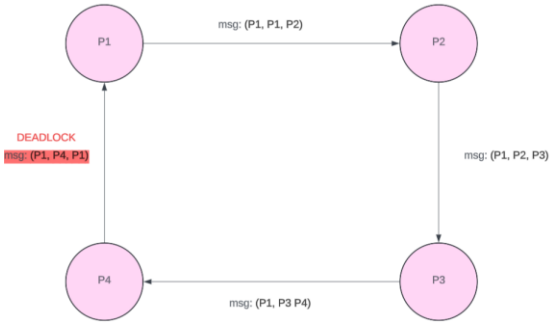
Algoritmo Distribuído

Cuando un proceso debe esperar por un recurso, envía un *probe message* al proceso que tiene el recurso. El mensaje contiene: id del proceso que se bloquea, id del proceso que envía el mensaje y id del proceso destinatario.

Al recibir el mensaje, el proceso actualiza el id del proceso que envía y el id del destinatario y lo envía a los procesos que tienen el recurso que necesita. Se va enviando entre ellos quién necesitaba el recurso originalmente y que recursos necesita.

Deadlock: Si el mensaje llega al proceso original, tenemos un ciclo en el grafo. Esto diría el destinatario original era él mismo y el resto están esperando un recurso que tiene él, y él está esperando otro recurso.

Ventajas: no tenemos un coordinador donde se centraliza el algoritmo, por lo que no contamos con un single point of failure.



Estos dos algoritmos me permiten detectar deadlocks, con ciclos en grafos.

Algoritmos de Prevención de Deadlocks

Algoritmo wait-die

Se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport).

Cuando un proceso (T1) está por bloquearse en un recurso (que tiene otro proceso (T2)), se comparan los timestamps.

Osea, T1 quiere hacer el lock, y T2 es quién tiene tomado el recurso actualmente.

- Si el timestamp es menor: T1 < T2 → T1 espera (el proceso más viejo espera): ***wait***
- Si no, si T1 > T2 → T1 aborta (el nuevo NO espera al viejo, **aborta la transacción**): ***die***

¿Cuál es el criterio de este algoritmo de hacer esperar a la transacción más vieja, y por qué no al revés?

Como la transacción más vieja estadísticamente realizó más operaciones que la transacción más nueva, es preferible minimizar los aborts de la transacción más nueva porque es menos costoso (hizo menos cosas)

La transacción más joven aborta.

Algoritmo wound-wait

Se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)

Cuando un proceso (T1) está por bloquearse en un recurso (que tiene otro proceso (T2)), se comparan los timestamps.

- Si el timestamp es menor: T1 < T2 → T2 aborta (el nuevo (T2) aborta para que el proceso más viejo pueda tomarlo): ***wound***
- Si no, si T1 > T2 → T1 espera (el nuevo proceso espera al viejo): ***wait***

En este, la transacción más joven es abortada por la otra.

Ambientes Distribuidos

Entidades

Es la unidad de cómputo de ambiente informático distribuido. Puede ser un proceso, un procesador, etc.

Capacidades

Cada entidad cuenta con las siguientes capacidades (es decir, que pueden hacer estas cosas):

- Acceso de lectura y escritura a una memoria local (no compartida con otras entidades)
    - Registro de estado: *status(x)*
    - Registro de valor de entrada: *value(x)*
- Procesamiento local
- Comunicación: preparación, transmisión y recepción de mensajes
- Setear y resetear un reloj local

Eventos Externos

La entidad solamente responde a eventos externos (**es reactiva**). Los posibles eventos externos son:

- Llegada de un mensaje de otra entidad
- Activación del reloj
- Un impulso espontaneo (exterior al sistema)

A excepción del impulso espontáneo, los eventos se generan dentro de los límites del sistema.

Reglas

Acciones

Secuencia finita e indivisible de operaciones. Es atómica porque se ejecuta sin interrupciones.

Reglas

Relación entre el evento que ocurre y el estado en el que se encuentra la entidad cuando ocurre dicho evento, de modo tal que $estado \times evento \rightarrow acción.$

Comportamiento

Es el conjunto B(x) (Behaviour) de todas las reglas que obedece una entidad x.

- Para cada posible evento y estado debe existir una única regla B(x).
- B(x) se llama también protocolo o algoritmo distribuido de x

Comportamiento colectivo del ambiente distribuido: **B(E) = B(x) : ∀x ∈ E**

Comportamiento Homogéneo

El comportamiento colectivo es homogéneo si todas las entidades que lo componen tienen el mismo comportamiento, o sea: ∀x, y ∈ E, B(x) = B(y)

Propiedad: Todo comportamiento colectivo se puede transformar en homogéneo.

Por ejemplo, si tenemos una entidad líder, que realiza diferentes operaciones que el resto, podemos restructurar la condición de “líder” como un estado interno, y luego evaluar si es líder o no y según eso hacer una acción distinta, pero en definitiva es: frente a X cualidad, hacer Y comportamiento, solo que la reacción es distinta, pero el sistema colectivo se convirtió en homogéneo.

Comunicación

Una entidad se comunica con otras entidades mediante mensajes (un mensaje es una secuencia finita de bits). Puede ocurrir que una entidad sólo pueda comunicarse con un subconjunto del resto de las entidades:

N_{OUT} (x) ⊆ E : conjunto de entidades a las cuales x puede enviarles un mensaje directamente

N_{IN} (x) ⊆ E: conjunto de entidades de las cuales x puede recibir un mensaje directamente

Axiomas

Delays de comunicación finitos: en ausencia de fallas los delays en la comunicación tienen una duración finita

Orientación local: Una entidad puede distinguir entre sus vecinos N_{OUT} y sus vecinos N_{IN}. Es decir, que Una entidad puede distinguir qué vecino le envía un mensaje y una entidad puede enviar un mensaje a un vecino específico.

Restricciones de confiabilidad

- Entrega garantizada: cualquier mensaje enviado será recibido con su contenido intacto
- Confiabilidad parcial: no ocurrirán fallas
- Confiabilidad total: no han ocurrido ni ocurrirán fallas

Restricciones temporales

- Delays de comunicación acotados: existe una constante Δ tal que en ausencia de fallas el delay de cualquier mensaje en el enlace es a lo sumo Δ
- Delays de comunicación unitarios: en ausencia de fallas, el delay de cualquier mensaje en un enlace es igual a una unidad de tiempo
- Relojes sincronizados: todos lo relojes locales se incrementan simultáneamente y el intervalo de incremento es constante

Costo y Complejidad

Son las medidas de comparación de los algoritmos distribuidos

Cantidad de actividades de comunicación

- Cantidad de transmisiones o costo de mensajes, M
- Carga de trabajo por entidad y carga de transmisión

Para comparar algoritmos distribuidos es importante comparar cuantos mensajes requiere para esa cantidad de entidades la ejecución del algoritmo (por ejemplo, cantidad de mensajes para elección de líder). El otro parámetro de comparación es el tiempo

Tiempo

- Tiempo total de ejecución del protocolo
- Tiempo ideal de ejecución: tiempo medido bajo ciertas condiciones, como delays de comunicación unitarios y relojes sincronizados

Con estos parámetros se puede estimar costo y complejidad de algoritmos distribuidos.

Tiempo y Eventos

Las entidades son reactivas, por lo que responden a eventos que reciben.

Tipos de eventos

- Impulso espontáneo
- Recepción de un mensaje
- Alarma del reloj activada

Los eventos desencadenan la ejecución de acciones en un tiempo futuro. Los distintos delays resultan en distintas ejecuciones del protocolo con posibles resultados diferentes.

Los eventos disparan acciones que pueden generar nuevos eventos. Si suceden, los nuevos eventos ocurrirán en un tiempo futuro: **Future(t).** Una ejecución se describe por la secuencia de eventos que ocurrieron.

Estados y Configuraciones

Las entidades tienen un estado interno y tienen acceso total a leerlo y escribirlo. Llamamos al estado interno con la letra “sigma”

- Estado interno de x en el instante t σ(x,t): contenido de los registros de x y el valor del reloj c_x en el instante t
- El estado interno de una entidad cambia con la ocurrencia de eventos

Sea una entidad x que recibe el mismo evento en dos ejecuciones distintas, y σ1 y σ2 los estados internos: Si σ1 = σ2 ⇒ el nuevo estado interno de x será el mismo en ambas ejecuciones.

Conocimiento

Es la información/datos disponibles en un entorno donde múltiples sistemas/nodos operan en conjunto.

Conocimiento local: contenido de la memoria local de x y la información que se deriva. En ausencia de fallas, el conocimiento no puede perderse.

Tipos de conocimiento

- Información métrica: información numérica sobre la red. Ej: número de nodos del grafo (n = ||V||), número de arcos del grafo (m = ||E||), diámetro del grafo y demás
- Propiedades topológicas: conocimiento sobre propiedades de la topología. Ej: el grafo es un anillo, el grafo es acíclico y demás
- Mapas topológicos: un mapa de la vecindad de la entidad hasta una distancia d. Ej: matriz de adyacencia del grafo