Concurrentes - Resumen

▼ 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: 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.

▼ Corrección / Sección Crítica / Locks

Propiedades de Corrección

- *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.
- *Liveness*: debe volverse verdadera eventualmente. 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.

Propiedades tipo Safety

- *Exclusión mutua*: dos procesos no deben intercalar instrucciones de la sección crítica.
- *Ausencia de deadlock*: un sistema que aún no finalizó debe poder continuar realizando su tarea, es decir, avanzar productivamente. De esta forma, si dos procesos están tratando de entrar a la sección crítica, eventualmente alguno de ellos debe tener éxito.

Propiedades tipo Liveness

- *Ausencia de starvation*: todo proceso que esté listo para utilizar un recurso debe recibir dicho recurso eventualmente.
- *Fairness (equidad o justicia)*: Se refiere a la probabilidad de que diferentes hilos puedan avanzar en lo que están haciendo. En general, un escenario débilmente fair garantiza que una instrucción que está continuamente habilitada eventualmente aparecerá en el escenario, mientras que un escenario 100% fair significa que todos los hilos deben avanzar en su trabajo en porciones casi iguales.

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. Un grafo de asignación de recursos (RAG) se puede utilizar para prevenir deadlocks al analizar el estado actual del sistema y predecir la ocurrencia de un deadlock. 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.

- *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.
- *Espera y retención:* Esta condición establece que un proceso puede mantener uno o más recursos mientras espera la asignación de otros recursos.
- *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.

- *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.

▼ **Sincronización / Semáforos / Barreras**

Semáforos

Mecanismos de sincronismo, implementado como una construcción de programación concurrente de más alto nivel.

- Tipo de dato compuesto por dos campos: un entero no negativo llamado V y un set de procesos llamado L.
- Se inicializa con un valor $k \geq 0$ y con el conjunto vacío $\varnothing$.
- Se definen dos operaciones atómicas sobre un semáforo S:
  - wait(S) también llamada p(S)
  - signal(S) también llamada v(S)

Un semáforo es un contador que representa la cantidad de recursos disponibles:

- Si contador > 0 ⇒ recurso disponible
- Si contador ≤ 0 ⇒ recurso no disponible

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

Barreras

Permiten sincronizar varios threads en puntos determinados de un cálculo o algoritmo.

- Creación de la barrera: `fn new(n: usize) -> Barrier`
- Bloquear al thread hasta que todos se encuentren en el punto: `fn wait(&self) -> BarrierWaitResult`

▼ **Monitores**

Una **condition variable** C:

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

Monitores

Herramientas de sincronización que permite a los hilos tener exclusión mutua y la posibilidad de esperar (block) por que una condición se vuelva falsa.

Semáforo	Monitor
`wait` puede o no bloquear	`waitC` siempre bloquea
`signal` desbloquea un proceso arbritrario	`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

Variables volatile

Los hilos guardan los valores de las variables compartidas en sus caches.

La palabra clave volatile indica al compilador que el valor de la variable no debe cachearse y debe leerse siempre de la memoria principal. De este modo, los hilos verán siempre el valor más actualizado de la variable.
La declaración de una variable como volatile no realiza ningún lockeo en dicha variable.

▼ **Modelo Fork Join**

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.

Partir el cómputo se realiza en general de forma recursiva: los sub-cómputos son independientes → el cómputo se puede realizar en paralelo.

Las sub-tareas se pueden crear en cualquier momento de la ejecución de la tarea.

Las tareas no deben bloquearse, excepto para esperar el final de las subtareas.

Modelo de concurrencia sin condiciones de carrera.

Los programas fork-join son determinísticos, los threads están aislados. El programa produce el mismo resultado independientemente de las diferencias de velocidad de los threads.

Work stealing

Algoritmo usado para hacer scheduling de tareas entre threads. Work stealing (Robo de trabajo): worker threads inactivos roban trabajo a threads ocupados, para realizar balanceo de carga. Los worker threads se comunican solamente cuando lo necesitan → menor necesidad de sincronización.

- 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 nal de la cola.
- Luego, toma la siguiente tarea para ser ejecutada del final de la cola.
- 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).

> Si se utilizara una única cola de tareas compartida entre todos los threads, habría una mayor contención 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.

Ejemplo:

```
use std::fs;
use std::thread;

fn count_lines_in_file(file: &str) -> u32 {
    let contents = fs::read_to_string(file).unwrap();
    contents.lines().count() as u32
}

fn count_lines_in_directory(dir: &str) -> u32 {
    let mut total_lines = 0;
    let mut handles = vec![];

    for entry in fs::read_dir(dir).unwrap() {
        let entry = entry.unwrap();
        let path = entry.path();
        if path.is_dir() {
            handles.push(thread::spawn(move || count_lines_in_directory(path.to_str().unwrap())));
        } else {
            total_lines += count_lines_in_file(path.to_str().unwrap());
        }
    }

    for handle in handles {
        total_lines += handle.join().unwrap();
    }

    total_lines
}
```

```
fn main() {
    let dir = "/path/to/dir";
    let total_lines = count_lines_in_directory(dir);
    println!("Total lines: {}", total_lines);
}
```

▼ Programación Asincrónica

La programación asincrónica es un medio 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.

Tareas asincrónicas de Rust

- Se puede usar Tareas asincrónicas de Rust para intercalar tareas en un único thread o en un pool de threads.
- Son mucho más livianas que los threads y, además, más rápidas de crear, más eficiente de pasar el control a ellas.

La mayor parte del tiempo, el thread principal está bloqueado a la espera de system calls. Para evitar esto, un thread debe poder tomar otras tareas mientras espera que la system call se complete.

Futures

Representa una operación sobre la que se puede testear si se completó.

Modelo piñata de la programación asincrónica: lo único que se puede hacer con un future es golpearlo con poll hasta que caiga el valor. (El método poll nunca bloquea). 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).
- Si no se completó, retorna Pending.

Funciones Async y Expresiones Await

- Invocar una función async retorna inmediatamente, antes de que comience a ejecutarse el cuerpo de la función.
- Se obtiene un Future del valor, que contiene todo lo necesario para que la función pueda ejecutarse (argumentos, espacio para variables locales, etc).
- La expresión await toma ownership del future y hace el poll.

- El Future alacena el punto donde debe retomarse en el siguiente poll y el estado local.

`block_on`

- `block_on` es una función sincrónica que produce el valor final de la función asincrónica.
- Es un adaptador del mundo asincrónico al sincrónico.
- No debe usarse en una función async (bloquea a todo el thread).
- `block_on` conoce cuánto hacer sleep hasta hacer poll de nuevo.

```
use std::net::SocketAddr;
use tokio::net::TcpListener;
use tokio::prelude::*;

async fn handle_client(mut stream: TcpStream) {
    let mut buffer = [0; 1024];

    loop {
        let n = match stream.read(&mut buffer).await {
            Ok(n) if n == 0 => return,
            Ok(n) => n,
            Err(e) => {
                eprintln!("failed to read from socket; err = {:?}", e);
                return;
            }
        };

        if let Err(e) = stream.write_all(&buffer[0..n]).await {
            eprintln!("failed to write to socket; err = {:?}", e);
            return;
        }
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr: SocketAddr = "127.0.0.1:8080".parse()?;
    let mut listener = TcpListener::bind(&addr).await?;

    loop {
        let (stream, _) = listener.accept().await?;
        tokio::spawn(async move {
            handle_client(stream).await;
        });
    }
}
```

▼ Pasaje de Mensajes / Canales

▼ Actores

Actor

El actor es la primitiva 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.

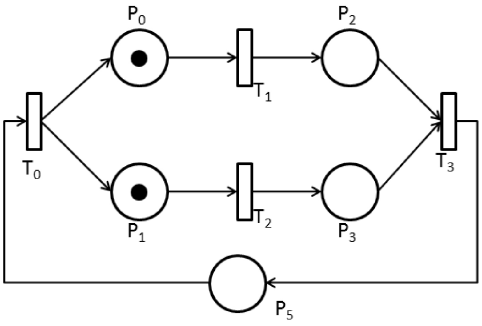
- Compuesto por:
  - Dirección: adonde enviarle mensajes.
  - Casilla de correo (mailbox): un FIFO de los últimos mensajes recibidos.
- Son aislados de otros actores: no comparten memoria.
- El estado privado solo puede cambiarse a partir de procesar mensajes.
- Pueden manejar un mensaje por vez.
- Los actores se comunican entre ellos solamente usando mensajes.
- Los mensajes son procesados por los actores de forma asincrónica.
- Los mensajes son estructuras simples inmutables.

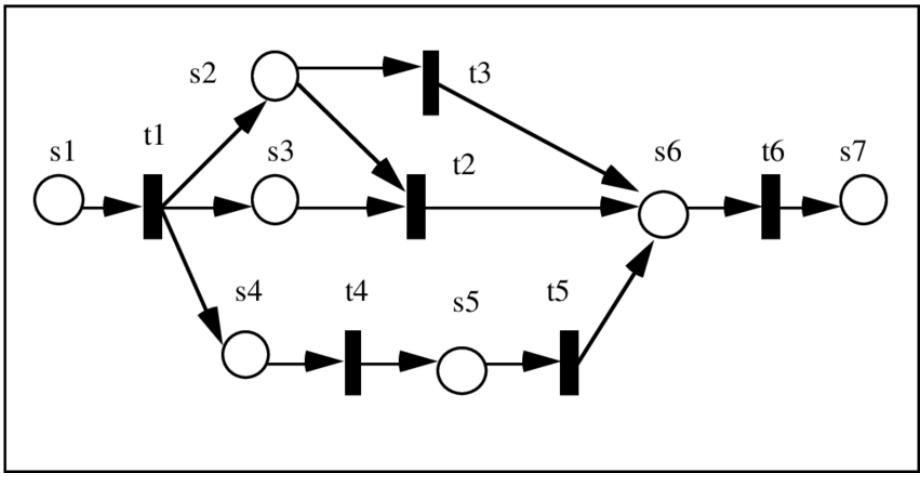
```
use std::time::Duration;
use actix::prelude::*;

struct Producer {
    interval: Duration,
    consumer: Addr<Consumer>,
}

impl Actor for Producer {
    type Context = Context<Self>;

    fn started(&mut self, ctx: &mut Self::Context) {
        ctx.run_interval(self.interval, |act, _ctx| {
            let num = rand::random::<u32>();
            act.consumer.do_send(AddNumber(num));
        });
    }
}

struct AddNumber(u32);

impl Message for AddNumber {
    type Result = ();
}

struct GetSum;

impl Message for GetSum {
    type Result = u32;
}

struct Consumer {
    sum: u32,
}

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

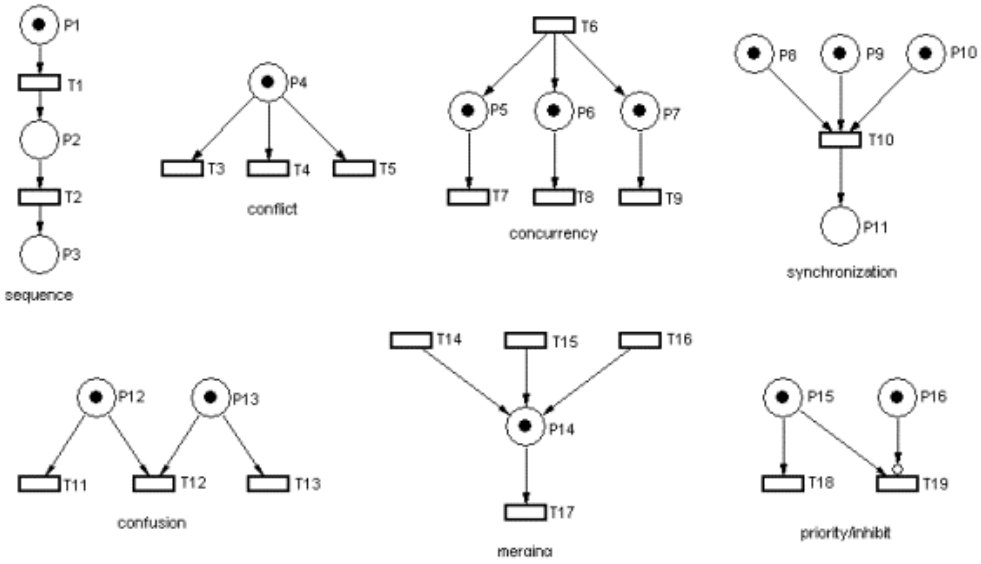
```
impl Handler<AddNumber> for Consumer {
    type Result = ();

    fn handle(&mut self, msg: AddNumber, _ctx: &mut Self::Context) -> Self::Result {
        self.sum += msg.0;
    }
}

impl Handler<GetSum> for Consumer {
    type Result = u32;

    fn handle(&mut self, _msg: GetSum, _ctx: &mut Self::Context) -> Self::Result {
        self.sum
    }
}

fn main() {
    let system = System::new();

    let consumer = Consumer { sum: 0 }.start();
    let _producer = Producer {
        interval: Duration::from_secs(1),
        consumer,
    }
    .start();

    system.run().unwrap();
}
```

▼ 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.
Una red de Petri está formada por lugares, transiciones, arcos dirigidos y marcas o fichas que ocupan posiciones dentro de los lugares. Las reglas son: Los arcos conectan un lugar a una transición así como una transición a un lugar. No puede haber arcos entre lugares ni entre transiciones. Los lugares contienen un número finito o infinito contable de marcas. Las transiciones se disparan, es decir consumen marcas de una posición de inicio y producen marcas en una posición de llegada. Una transición está habilitada si tiene marcas en todas sus posiciones de entrada.
La presencia de marcas se interpreta habitualmente como presencia de recursos. El franqueo de una transición (la acción a ejecutar) se realiza cuando se cumplen unas determinadas precondiciones, indicadas por las marcas en las fichas (hay una cantidad suficiente de recursos), y la transición (ejecución de la acción) genera unas postcondiciones que modifican las marcas de otras fichas (se liberan los recursos) y así se permite el franqueo de transiciones posteriores.







▼ Concurrencia Distribuida: Algoritmos de Exclusión Mutua

Exclusión Mutua Distribuida

Algoritmo Centralizado

1. Un proceso es elegido coordinador
2. Cuando un proceso quiere entrar a la SC, envía un mensaje al coordinador

3. Si no hay ningún proceso en la SC, el coordinador envía OK; si hay, el coordinador no envía respuesta hasta que se libere la SC

Algoritmo Distribuido

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:

- Si no está en la CS y no quiere entrar, envía OK
- Si está en la CS, no responde y encola el mensaje. Cuando sale de la SC, envía OK
- Si quiere entrar en la CS, compara el timestamp y gana el menor

Algoritmo Token Ring

- Se conforma un anillo mediante conexiones punto a punto
- 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, continua circulando el token
- El proceso no puede entrar a otra SC con el mismo token

Modelo Cliente-Servidor

Sockets

- Permiten la comunicación entre dos procesos diferentes
  - En la misma máquina
  - 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

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.

Tipos de Sockets:

- Stream sockets: usan el protocolo TCP: entrega garantizada del flujo de bytes
- Datagram sockets: usan el protocolo UDP: la entrega no está garantizada; servicio sin conexión.
- Raw sockets: permiten a las aplicaciones enviar paquetes IP.
- Sequenced packet sockets: similares a stream sockets, pero preservan los delimitadores de registro. Utilizan el protocolo SPP (Sequenced Packet Protocol).

▼ **Concurrencia Distribuida (parte 2): Algoritmos de Elección**

- 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.

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
4. El nuevo coordinador se anuncia con un mensaje COORDINATOR

Siempre gana el proceso con mayor número.

Algoritmo Ring

1. Los procesos están ordenados lógicamente; cada uno conoce a su sucesor
2. Cuando un proceso nota que el coordinador falló, arma un mensaje ELECTION que contiene su número de proceso y lo envía al sucesor

3. El proceso que recibe el mensaje, agrega su número de proceso a la lista dentro del mensaje y lo envía al sucesor
4. Cuando el proceso original recibe el mensaje, lo cambia a COORDINATOR y lo envía. El nuevo coordinador es el proceso de mayor número de la lista. La lista se mantiene para informar el nuevo anillo
5. Cuando este mensaje finaliza la circulación, se elimina del anillo

▼ **Concurrencia Distribuida (parte 3): Transacciones**

Modelo

- 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
- Storage estable:
  - Se implementa con discos
  - La probabilidad de perder los datos es extremadamente pequeña

Primitivas

- BEGIN TRANSACTION: marca el inicio de la transacción
- END TRANSACTION: finalizar la transacción y tratar de hacer commit
- 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

- Atómicas: la transacción no puede ser dividida
- Consistentes: la transacción cumple con todos los invariantes del sistema
- Aisladas o serializadas: las transacciones concurrentes no interfieren con ellas mismas
- Durables: una vez que se commitean los cambios, son permanentes (Excepción: transacciones anidadas no son durables)

En inglés, ACID

Implementación

Private Workspace:

- Al iniciar una transacción, el proceso recibe una copia de todos los archivos a los cuales tiene acceso
- Hasta que hace commit, el proceso trabaja con la copia
- Al hacer commit, se persisten los cambios
- Desventaja: extremadamente costoso salvo por optimizaciones

Writeahead Log:

- 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)

Commit en dos fases:

- El coordinador es aquel proceso que ejecuta la transacción
- Fase 1
  - El coordinador escribe prepare en su log y envía el mensaje prepare al resto de los procesos
  - Los procesos que reciben el mensaje, escriben ready en el log y envían ready al coordinador
- Fase 2
  - El coordinador hace los cambios y envía el mensaje commit al resto de los procesos
  - Los procesos que reciben el mensaje, escriben commit en el log y envían finished al coordinador

Ventajas y desventajas de cada implementación:

- *Private Workspace:* La ventaja de esta implementación es que permite a cada proceso trabajar con su propia copia de los archivos, lo que reduce la necesidad de sincronización entre los procesos. Sin embargo, la desventaja es que puede ser extremadamente costoso en términos de recursos, ya que requiere hacer una copia de todos los archivos al iniciar una transacción.
- *Writeahead Log:* La ventaja de esta implementación es que permite modificar los archivos directamente, lo que puede mejorar el rendimiento del sistema. Además, el uso de un log para registrar los cambios aplicados permite deshacer fácilmente los cambios en caso de que sea necesario hacer un rollback. Sin embargo, la desventaja es que puede aumentar la complejidad del código y requerir más recursos para mantener el log.

- *Commit en dos fases:* La ventaja de esta implementación es que permite coordinar el commit entre múltiples procesos, lo que puede mejorar la consistencia y la confiabilidad del sistema. Sin embargo, la desventaja es que puede aumentar la complejidad del código y requerir más recursos para coordinar el commit entre los procesos.

> 💡 **¿Cuál sería la más conveniente en un sistema de datos persistente?**
> En un sistema de datos persistente, una opción conveniente podría ser utilizar el Commit en dos fases. Esta implementación permite coordinar el commit entre múltiples procesos, lo que puede mejorar la consistencia y la confiabilidad del sistema. Al utilizar un coordinador para gestionar el commit, se puede garantizar que todos los procesos involucrados en una transacción estén de acuerdo antes de realizar el commit. Esto puede reducir el riesgo de inconsistencias en los datos y mejorar la confiabilidad del sistema.

Control de concurrencia

Lockeo: two-phase locking

- Fase de expansión: se toman todos los locks a usar
- Fase de contracción: se liberan todos los locks (no se pueden tomar nuevos locks)
- Garantiza propiedad serializable para las transacciones
- Pueden ocurrir deadlocks
- Strict two-phase locking: la contracción ocurre después del commit

Concurrencia Optimista

- 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 y favorece el paralelismo.

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

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
- Al commitear se actualizan los timestamps del archivo

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

Algoritmo distribuido

- 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
- Si el mensaje llega al proceso original, tenemos un ciclo en el grafo

Prevención de deadlocks

Algoritmo wait-die

- Se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)
- Cuando un proceso está por bloquearse en un recurso (que tiene otro proceso), se comparan los timestamps
  - Si el timestamp es menor (proceso más viejo), espera
  - Si no, el proceso aborta la transacción

Algoritmo wound-wait

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

- Cuando un proceso está por bloquearse en un recurso (que tiene otro proceso), se comparan los timestamps
  - Si el timestamp es menor (proceso más viejo), se aborta la transacción del proceso que tiene el recurso, para que el más viejo pueda tomarlo
  - Si no, el proceso espera

▼ **Ambientes Distribuidos**

Entidades

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

Cada entidad cuenta con las siguientes capacidades:

- 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
- Activación del reloj
- Un impulso espontáneo

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

Acción

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

Regla

Es la 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 × evento → acción

Comportamiento

Es el conjunto B(x) 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: $B(E) = B(x) : \forall x \in E$

El comportamiento colectivo es homogéneo si todas las entidades que lo componen tienen el mismo comportamiento, o sea: $\forall x, y \in E, B(x) = B(y)$
Propiedad: Todo comportamiento colectivo se puede transformar 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) $\subseteq$ E: conjunto de entidades a las cuales x puede enviarles un mensaje directamente
  - N_IN (x) $\subseteq$ 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 entre sus vecinos N_IN

- Una entidad puede distinguir qué vecino le envía un mensaje
- 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
- Tiempo
  - Tiempo total de ejecución del protocolo
  - Tiempo ideal de ejecución: tiempo medido bajo ciertas condiciones, como delays de comunicación unitarios y relojes sincronizados

Tiempo y Eventos

Tipos de eventos

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

Los eventos desencadenan 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

- Estado interno de x en el instante t σ(x,t): contenido de los registros de x y el valor del reloj cx 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

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 = kVk), número de arcos del grafo (m = kEk), 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

---

Versión PDF (por si no funciona la opción de Notion para exportar)

Img - Concu.pdf

Resumen - Concu.pdf