Clase del día - 21/05/2021

En la clase de hoy vamos a comenzar con el tema de replicación.

Los datos son el principal activo de las empresas e instituciones.

Si bien es cierto que los sistemas informáticos son de gran importancia para automatizar los procesos en las empresas, los datos representan los objetos de negocio que mayor persistencia en el tiempo tienen, ya que ellos corresponden a lo que se sabe de los clientes, los productos, los insumos, los activos, los pasivos, las ventas, los procesos, los empleados, y muchos objetos de negocio más.

La replicación de los datos es una estrategia utilizada para mantener copias consistentes de los datos en diferentes locaciones, con el objetivo de tener la capacidad de recuperar la operación en caso de desastre.

**¿Por qué replicar los datos?**

Los datos se replican para satisfacer dos requerimientos no funcionales de los sistemas distribuidos: la confiabilidad y el rendimiento.

Replicar los datos en diferentes sistemas de archivos, aumenta la **confiabilidad**del sistema, ya que si una copia de los datos falla es posible seguir trabajando con otra copia de los datos.

Por ejemplo, en los sistemas manejadores de bases de datos se acostumbra configurar copias "espejo" de las tablas, de tal manera que cuándo se inserta, modifica o borra datos en una tabla, se realicen las mismas operaciones sobre la tabla "espejo", si la lectura de la tabla principal falla, entonces automáticamente se realiza la lectura sobre la copia "espejo", sin mayor intervención del sistema que accede a la base de datos.

La replicación de datos también mejora el **rendimiento**de un sistema distribuido que requiere escalar en tamaño y geografía. Es una buena práctica replicar los catálogos de los sistemas (por ejemplo, los catálogos de clientes, de productos, de cuentas, etc.) ya que se trata de datos que se modifican poco, por tanto en un sistema distribuido resulta más rápido el acceso a estos datos si los tiene cerca.

No obstante las ventajas que tiene la replicación de los datos, la principal dificultad es mantener la consistencia entre las copias.

La **consistencia**de los datos significa que todas las copias deben tener los mismo datos.

Mantener la consistencia entre copias puede impactar el rendimiento del sistema, debido a que la actualización de las copias representa un costo en tiempo y recursos (CPU, red, almacenamiento, etc). Entonces será necesario evaluar el costo de mantener consistentes las copias y el beneficio del aumento del rendimiento que trae la replicación de los datos.

**Modelos de consistencia**

Un **modelo de consistencia** es un acuerdo entre los procesos que acceden un almacén de datos y el almacén de datos.

El acuerdo establece las reglas que deben obedecer los procesos cuando acceden el almacén de datos de manera que los procesos puedan tener una imagen consistente de los datos.

El **almacén de datos** puede ser una base de datos distribuida, un sistema de archivos distribuido o una combinación de ambos.

El principio en que se basa un modelo de consistencia es que si un proceso realiza una operación de lectura sobre elemento de datos, se espera leer el resultado de la última escritura sobre el mismo elemento de datos, independientemente de qué proceso realizó la escritura.

**Consistencia secuencial**

El modelo de consistencia secuencial fue propuesto por Lamport (1979), y dice que un almacén de datos es secuencialmente consistente si:

"*El resultado de cualquier ejecución es el mismo que si las operaciones (de lectura y escritura) de todos los procesos efectuados sobre el almacén de datos se ejecutaran en algún orden secuencial y las operaciones de cada proceso individual aparecieran en esa secuencia en el orden especificado por su programa*".

Esto significa que los procesos que ejecutan en paralelo (en un sistema distribuido) deberán "ver" la misma secuencia de operaciones de lectura y escritura al almacén de datos, y tales operaciones deberán aparecer en el mismo orden en que se ejecutan en cada proceso individual.

La consistencia secuencial corresponde a un orden parcial de las operaciones de lectura y escritura, el cual puede ser implementado mediante la relación happen-before.

Por otra parte si se requiere que las operaciones de lectura y escritura se realicen en un orden completo, sería necesario implementar un reloj global.

**Consistencia de entrada**

El modelo de consistencia secuencial es un modelo de granularidad fina propuesto originalmente para acceder localidades de memoria compartida en sistemas multiprocesadores, sin embargo resulta muy costoso para los sistemas distribuidos dónde los datos tienen granularidad gruesa, es decir, se accede a registros, tablas, archivos, etc.

Berchad ([The Midway Distributed Shared Memory System](https://homes.cs.washington.edu/~bershad/Papers/00289730.pdf), 1993) propuso un modelo de consistencia basado en la relación entre objetos de sincronización (locks) que protegen secciones criticas y los datos compartidos dentro de las secciones críticas.

El modelo de consistencia de entrada utiliza objetos de sincronización exclusiva y no-exclusiva (compartida) para garantizar el orden en que se ejecutan las operaciones de lectura y escritura sobre un mismo elemento de datos.

Una sección crítica comienza con una operación de adquisición del lock y termina con la liberación de lock. A estas operaciones se les llama generalmente "lock" y "unlock".

Las reglas que se debe cumplir en este modelo de consistencia son:

1. Cuando un proceso ejecuta la operación "lock" ésta debe esperar a que se realicen todas las operaciones de escritura de los datos compartidos por el proceso.

2. Un proceso no puede adquirir un lock si algún otro proceso lo adquirió ya sea en forma exclusiva o compartida.

3. Si un proceso adquirió un lock en forma exclusiva, ningún otro proceso puede adquirir el lock en forma compartida.

Estas reglas garantizan que las lecturas de datos compartidos (que se realizan dentro de una sección crítica) obtendrán los datos actualizados por la última escritura, la cual también se debió realizar dentro de una sección crítica.

Como puede observarse, no importa el orden en que se realizan las lecturas y escrituras a los elementos de datos, lo que importa es el orden en que se realizan las operaciones "lock" y "unlock".

Para lograr la consistencia de los elementos de datos compartidos, los objetos de sincronización (locks) deberán implementarse en forma global, tal como se explicó en el tema de "Sincronización y coordinación".

Veamos un ejemplo del uso de locks para sincronizar un elemento de datos compartido por dos threads que ejecutan en paralelo en una computadora con dos o más núcleos.

Supongamos que tenemos  dos threads **t1** y **t2**, y cada thread ejecuta un ciclo dónde se incrementa la variable global **n**.

**class A extends Thread  
{  
  static long n;  
  public void run()  
  {  
    for (int i = 0; i < 100000; i++)  
        n++;  
  }  
  public static void main(String[] args) throws Exception  
  {  
    A t1 = new A();  
    A t2 = new A();  
    t1.start();  
    t2.start();  
    t1.join();  
    t2.join();  
    System.out.println(n);  
  }  
}**

Al ejecutar varias veces el programa anterior podemos ver que el valor final de **n** no es el mismo ¿por qué?

La instrucción **n++** copia la variable global **n** a un registro del procesador, el valor en el registro se incrementa y se escribe a la variable **n**.

Debido a que los threads **t1** y **t2** ejecutan en paralelo en una computadora con dos o más núcleos, el thread **t1** leerá y escribirá la variable **n** al mismo tiempo que el thread **t2** realiza las mismas operaciones.

Dado que la lectura que realiza **t1** no está ordenada con respecto a la escritura que hace **t2** y que la lectura que realiza **t2** no está ordenada con respecto a la escritura que hace **t1**, es posible que no se escriba algún incremento en la variable **n**, lo que produce un valor final menor a 200000.

Para resolver el problema se requiere que el programador identifique las secciones críticas y agregue las operaciones "lock" y "unlock" necesarias.

En este caso, la sección crítica es la instrucción **n++**, que es dónde se lee y escribe la variable **n** compartida por los dos threads.

Por lo tanto, es necesario ejecutar "lock" antes de **n++** y ejecutar "unlock" después.

En java se utiliza la instrucción **synchronized**(*objeto*){ *bloque-de-instrucciones* } para definir un bloque de instrucciones como sección crítica controlada por el lock que contiene el *objeto* (recordemos que en Java todos los objetos incluyen un lock).

Entonces el código del programa queda de la siguiente manera:

**class A extends Thread  
{  
  static long n;  
  static Object obj = new Object();  
  public void run()  
  {  
    for (int i = 0; i < 100000; i++)  
      synchronized(obj)  
      {  
        n++;  
      }  
  }  
  public static void main(String[] args) throws Exception  
  {  
    A t1 = new A();  
    A t2 = new A();  
    t1.start();  
    t2.start();  
    t1.join();  
    t2.join();  
    System.out.println(n);  
  }  
}**

Al ejecutar varias veces el programa anterior podemos ver que el valor final de la variable **n** siempre es 200000.

Actividades individuales a realizar

1. Compilar y ejecutar los programas que vimos.  
2. ¿Qué pasa si el número de iteraciones del ciclo es 100 o 1000000 en el primer programa?

3. ¿Qué pasa si el primer programa se ejecuta en una computadora con un solo procesador? ¿Es necesario sincronizar los threads?

4. Si el segundo programa crea más threads que incrementan la variable n ¿El valor de n desplegado al final sigue siendo correcto?