TP N.º 1 Programación Concurrente

Abud Matías, Barella Lautaro, Mohr Agustín

2024

- Ingeniería en Sistemas de Información.
- Comisión "A".
- Docente: Meinero Silvina Haydee.
- UTN FRSF.

1. Implementación

Para poder modelar el escenario planteado, lo primero que hicimos fue definir las clases ”ListaEnlazada” y ”Nodo”, las mismas cuentan con los siguientes atributos y constructores:

```
public class ListaEnlazada{
private Nodo head;

public ListaEnlazada() {
    head = new Nodo (Integer.MIN_VALUE,null);
    head.next = new Nodo (Integer.MAX_VALUE,null);
    }
}

public class Nodo {
    private Lock lock = new ReentrantLock();
    public Object item;
    public int key;
    public Nodo next;
    public Nodo (int key, Object item) {
        this.key=key;
        this.item=item;
    }
}
```

Cada lista enlazada tendrá dos nodos que funcionaran a modo de "banderas"para identificar el principio el final de la estructura de datos. Por otro lado, cada nodo tendrá como atributo el objeto que se almacena en el mismo, su key (hashcode) y el nodo siguiente.

1.1. Granularidad Fina

Para trabajar la solución de granularidad fina, desarrollamos los siguientes métodos:

```
public void addFina(Object o, Integer id) {
    Nodo pred = null, curr = null;
    int key= o.hashCode();
    try {
        System.out.println("Hilo "+id+"A intentando agregar nodo " + o);
        pred=head;
        pred.lock();
        curr=pred.next;
        curr.lock();
        while(curr.key<key) {
            pred.unlock();
            pred=curr;
            curr=curr.next;
            curr.lock();
        }
        if(key== curr.key) {
            System.out.println("Ya existe el nodo "+o);
        }
        else{
            Nodo n= new Nodo(key,o);
            pred.next=n;
            n.next=curr;
            System.out.println("Nodo "+o+" agregado");
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        pred.unlock();
        curr.unlock();
    }
}

public void removeFina(Object o, Integer id) {
    Nodo pred = null, curr = null;
```

```
37      int key= o.hashCode();
38      try {
39          System.out.println("Hilo "+id+"B intentando eliminar nodo " + o);
40          pred=head;
41          pred.lock();
42          curr=pred.next;
43          curr.lock();
44          while(curr.key<key) {
45              pred.unlock();
46              pred=curr;
47              curr=curr.next;
48              curr.lock();
49          }
50          if(key== curr.key) {
51              pred.next=curr.next;
52              System.out.println("Nodo "+o+" eliminado");
53          }
54          else{
55              System.out.println("No existe el nodo "+o);
56          }
57      }
58      catch(Exception e){
59          e.printStackTrace();
60      }
61      finally{
62          pred.unlock();
63          curr.unlock();
64      }
65  }
```

La estrategia de estos métodos consiste en primero obtener el lock sobre los nodos actual y anterior, para luego buscar el nodo que se necesita eliminar, o el lugar donde se debe agregar el nuevo nodo, una vez encontrado, se realiza la acción deseada, y luego se liberan los locks.

1.2. Sincronización Optimista

Para este enfoque se realizaron los siguientes metodos:

```
1   public void addOptimista(Object o, Integer id) {
2       int key= o.hashCode();
3       while(true) {
4           Nodo pred = head, curr = head.next;
5           while(curr.key<key) {
6               pred=curr;
7               curr=curr.next;
8           }
9       try {
10          System.out.println("Hilo "+id+"A intentando agregar nodo " + o);
11          pred.lock();
12          curr.lock();
13          if(validate(pred,curr)) {
14              if(key== curr.key) {
15                  System.out.println("Ya existe el nodo "+o);
16                  break;
17              }
18              else{
19                  Nodo n= new Nodo(key,o);
20                  pred.next=n;
21                  n.next=curr;
22                  System.out.println("Nodo "+o+" agregado");
23                  break;
24              }
25          }
26      }
27      catch(Exception e){
28          e.printStackTrace();
29      }
30      finally{
31          pred.unlock();
32          curr.unlock();
```

```
    }
    }
}
public void removeOptimista(Object o, Integer id) {

    int key= o.hashCode();
    while(true) {
        Nodo pred = head, curr = head.next;
        while(curr.key<key) {
            pred=curr;
            curr=curr.next;
        }
    try {
        System.out.println("Hilo "+id+"B intentando eliminar nodo " + o);
        pred.lock();
        curr.lock();
        if(validate(pred,curr)) {
            if(key== curr.key) {
                pred.next=curr.next;
                System.out.println("Nodo "+o+" eliminado");
                break;
            }
            else{
                System.out.println("No existe el nodo "+o);
                break;
            }
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        pred.unlock();
        curr.unlock();
    }
    }
}
private boolean validate(Nodo pred, Nodo curr) {
    Nodo aux= head;
    while(aux.key<=pred.key) {
        if(aux==pred)
            return pred.next == curr;
        aux=aux.next;
    }
    return false;
}
```

La estrategia para los mismos es, primero buscar el nodo correspondiente, y luego obtener el lock sobre el mismo y el anterior, para después validar que sigan siendo adyacentes (es decir, que ambos nodos siguen existiendo y no se agrego uno nuevo entre los mismos) para posteriormente realizar la acción correspondiente y finalmente liberar los locks.

1.3. Sin Locks

La estrategia para este enfoque es utilizar la misma busqueda que en los anteriores, pero no utilizar en ningun momento locks:

```
public void addSinLocks (Object o, Integer id) {
    Nodo pred=head, curr=head.next;
    int key=o.hashCode();
    System.out.println("Hilo "+id+"A intentando agregar nodo " + o);
    while(key>curr.key) {
        pred=curr;
        curr=curr.next;
    }
    if(key==curr.key) {
        System.out.print("Ya existe el nodo "+o);
    }
    else {
```

```
        Nodo n = new Nodo(key, o);
        pred.next=n;
        n.next=curr;
        System.out.println("Nodo "+o+" agregado");
    }
}
public void removeSinLocks(Object o, Integer id) {
    System.out.println("Hilo "+id+"B intentando eliminar nodo " + o);
    Nodo pred=head, curr=head.next;
    int key=o.hashCode();
    while(key>curr.key) {
        pred=curr;
        curr=curr.next;
    }
    if(key==curr.key) {
        pred.next=curr.next;
        System.out.print("Nodo "+o+" eliminado");
    }
    else {
        System.out.println("No existe el nodo "+o);
    }
}
```

2. Escenario

2.1. Hilos

Realizamos dos clases de hilos, HiloA, encargado de agregar nodos e HiloB, encargado de eliminarlos. Ambos tienen los mismos atributos y constructores, la diferencia esta en los métodos que utilizan durante sus ciclos de vida:

```
public class HiloA implements Runnable {
  private Integer id;
  private ListaEnlazada l;
  private Integer caso;

  public HiloA(Integer id, ListaEnlazada l, Integer caso) {
    this.id=id;
    this.l=l;
    this.caso=caso;
  }
```

Contienen un ID para identificar que hilo realiza cada operación, la lista enlazada con la que trabajaran, y un numero entero llamado çaso", el cual servirá para identificar que estrategia de sincronización se utilizara:

- caso=1 → Granularidad fina.
- caso=2 → Sincronización optimista.
- caso=3 → Sin locks.

2.2. Escenario Principal

El escenario principal, con la configuración utilizada para realizar el primer experimento se modelo de la siguiente forma:

```
public class Escenario {
    private static Integer N=6;
    public static void main(String[] args) {
    //GRANULARIDAD FINA//
    //VARIABLES PARA CONTROLAR EL TIEMPO DE EJECUCION
    long inicio;
    long fin;
    inicio= System.nanoTime();
    ListaEnlazada l= new ListaEnlazada(); //LISTA ENLAZADA PARA G.F.
```

```
    //CREACION Y COMIENZO DE EJECUCION DE HILOS TIPO A
    for(int i=0; i<N/2+1;i++) {
        new Thread(new HiloA(i,l,1)).start();
    }
     //CREACION Y COMIENZO DE EJECUCION DE HILOS TIPO B
    for(int i=0; i<N/2-1;i++) {
        new Thread(new HiloB(i,l,1,N)).start();
    }
    //PAUSA PARA QUE TODOS LOS HILOS TERMINEN DE EJECUTAR
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    //VERIFICACION DE NODOS EN LA LISTA, LOS VALORES NULL SON LAS BANDERAS
    l.imprimir();
    fin=System.nanoTime();
    long total=fin-inicio;
    System.out.println("Sincronizacion de granularidad fina: "+ total +" ns");

    ///SINCRONIZACION OPTIMISTA///

    inicio= System.nanoTime();
    ListaEnlazada l2= new ListaEnlazada();
    for(int i=0; i<=N/2+1;i++) {
        new Thread(new HiloA(i,l2,2)).start();
    }
    for(int i=0; i<=N/2-1;i++) {
        new Thread(new HiloB(i,l2,2,N)).start();
    }
    //PAUSA PARA QUE TODOS LOS HILOS TERMINEN DE EJECUTAR
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    //VERIFICACION DE NODOS EN LA LISTA, LOS VALORES NULL SON LAS BANDERAS
    l2.imprimir();
    fin=System.nanoTime();
    total=fin-inicio;
    System.out.println("Sincronizacion optimista: "+ total +" ns");

    ///SIN LOCKS///

    inicio= System.nanoTime();
    ListaEnlazada l3= new ListaEnlazada();
    for(int i=0; i<=N/2+1;i++) {
        new Thread(new HiloA(i,l3,3)).start();
    }
    for(int i=0; i<=N/2-1;i++) {
        new Thread(new HiloB(i,l3,3,N)).start();
    }
    //PAUSA PARA QUE TODOS LOS HILOS TERMINEN DE EJECUTAR
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    //VERIFICACION DE NODOS EN LA LISTA, LOS VALORES NULL SON LAS BANDERAS
    l3.imprimir();
    fin=System.nanoTime();
    total=fin-inicio;
    System.out.println("Sin locks: "+ total +" ns");


    }
}
```

Donde N es la constante que controla la cantidad de hilos a crear, primero se crea una lista enlazada que se utilizara para trabajar con granularidad fina, luego se crean los diferentes hilos, se hace una pausa para asegurarse de que todos los hilos terminen su ejecución, y se mide el tiempo. Luego se repite el proceso para los otros dos enfoques, con nuevas listas e hilos.

3. Experimentos y resultados

A continuación se realizaran diferentes experimentos, variando parámetros para analizar las estrategias propuestas. Cabe destacar que debido a que el tiempo de ejecución puede variar de una ejecución a otra, se opto por realizar 5 ejecuciones y tomar el tiempo promedio de las mismas. Además, hay que tener en cuenta que en todos los escenarios se toma una pausa de 1000 milisegundos para asegurar que todos los hilos finalicen correctamente.

3.1. Primer experimento

El primer experimento se realizo con un $N = 6$, los primeros 4 hilos crearan los nodos que contienen el numero de su ID, el mismo más 1, más 2 y más 3, es decir, el hilo 0 creara el nodo 0, 1, 2 y 3; el 1 intentara crear el nodo 1, 2, 3 y 4, etc. De esta forma solo se deberían poder crear los nodos 0, 1, 2, 3, 4, 5 y 6. Los siguientes 2 hilos serán tipo B, e intentaran eliminar los nodos con su ID, y con su ID más 1, más 2 y más 3. Cabe destacar que los resultados pueden variar según que hilo consigue los locks primero. A continuación los resultados obtenidos luego de varias ejecuciones:

- Granularidad Fina: Se -tomaron los siguientes tiempos: 1006431700 ns, 1008835900 ns, 1017236200 ns, 1021342000 ns, 1022184700 ns. Tiempo promedio: 1015206100 ns.
- Sincronización Optimista: los resultados varian entre ejecuciones. Se tomaron los siguientes tiempos: 1003934900 ns, 1001031900 ns, 1002431400 ns, 1011808800 ns, 1006841000 ns. Tiempo promedio: 1005209600 ns.
- Sin Locks: Al no tener locks, el resultado puede variar mucho de una EJECUCION a otra. Se tomaron los siguientes tiempos: 1009575400 ns, 1015792500 ns, 1014968100 ns, 1009088900 ns, 1010798300 ns. Tiempo promedio: 1012044640 ns

3.2. Segundo experimento

Para el segundo experimento se variara la proporción de hilos que realiza cada operación, manteniendo constante la cantidad total de hilos. En este caso se utilizaran 2 hilos tipo A y cuatro tipo B, realizando cada hilo la misma cantidad de operaciones que en el experimento anterior. A continuación los resultados:

- Granularidad Fina: se llega al resultado correspondiente en todas las ejecuciones, y los tiempos fueron los siguientes: 1015134200 ns, 1015554600 ns, 1018021100 ns, 1020149300 ns, 1015866200 ns. Tiempo promedio: 1016945080 ns.
- Sincronización Optimista: en algunas ejecuciones se llega a resultados diferentes. Se obtuvieron los siguientes tiempos: 1001605400 ns, 1002819300 ns, 1004314000 ns, 1008600500 ns, 1013306600 ns. Tiempo promedio: 1006129160 ns.
- Sin Locks: se tomaron los siguientes tiempos: 1004138700 ns, 1003984000 ns, 1010540300 ns, 1015138600 ns, 1009312500 ns. Tiempo promedio: 1008622820 ns.

3.3. Tercer experimento

Para el tercer experimento se cambio la cantidad de hilos totales, dejando fija la proporción hilos que realizan cada operación y la cantidad de operaciones totales con respecto al primer experimento; es decir, se crean 12 hilos, pero cada hilo realiza solo dos operaciones. Se obtuvieron los siguientes resultados:

- Granularidad Fina: 1008261300 ns, 1008902500 ns, 1013823700 ns, 1013735800 ns, 1009252200 ns. Tiempo promedio: 1010795100 ns.
- Sincronización Optimista: 1014845200 ns, 1010161500 ns, 1007706000 ns, 1005774600 ns, 1005848300 ns. Tiempo promedio: 1008867120 ns
- Sin Locks: 1009392500 ns, 1012317600 ns, 1008919400 ns, 1003989800 ns, 1009907800 ns. Tiempo promedio: 1008905420 ns.

Cabe destacar que en todas las ejecuciones, el método de Granularidad Fina arrojo los mismos resultados: $[null, 6, 7, null]$.

3.4. Cuarto experimento

En este experimento, se varían la cantidad de hilos totales, dejando fija la cantidad de operaciones que realiza cada hilo, por consecuencia se varian la cantidad de operaciones totales.

- Granularidad Fina: 1005722700 ns, 1012599700 ns, 1015377400 ns, 1015172000 ns, 1009194000 ns.Tiempo promedio: 1011613160 ns.
- Sincronización Optimista: 1005333200 ns, 1005337000 ns, 1009676800 ns, 1008938800 ns, 1008813200 ns.Tiempo promedio: 1007619800 ns.
- Sin Locks: 1005476700 ns, 1006264600 ns, 1006828100 ns, 1007450800 ns, 1017026200 ns. Tiempo promedio: 1008609280 ns.

4. Análisis de resultados

4.1. Análisis de los experimentos

Los resultados obtenidos son:

- Granularidad Fina: Es el mas lento de los tres, entre un 0.20 % y 1 % menos que los demás.
- Sincronización Optimista: Es el mas rápido de los tres.
- Sin Locks: Pierde con sincronización optimista entre un 0.005 % y un 0.7 %.

4.2. Conclusiones

Luego de analizar los resultados descriptos anteriormente, elaboramos el siguiente cuadro comparativo entre los métodos implementados, teniendo en cuenta la concurrencia, DeadLocks y el costo computacional.

Comparación de métodos			
	Concurrencia	DeadLock	Costo
Granularidad Fina	Muy buena ya que divide la sección critica en porciones muy pequeñas por lo tanto pueden desarrollarse varios hilos a la vez	No tiene riesgo de deadlock	Resulta costoso por la cantidad de obtenciones y liberaciones de locks
Sincronización optimista	Puede lograr muy buena concurrencia en escenarios en los cuales los conflictos sean poco frecuentes	No necesariamente deadlock, pero podría ocurrir algo similar en caso que entre en un bucle de conflictos	El costo se puede llegar a ser elevado en casos que tenga que tenga que deshacer cambios, pero en general es muy bueno
Libre de locks	Asegura progreso aunque algún thread este demorado o bloqueado gracias al uso de las operaciones atómicas	No tiene riesgo de deadlock	El costo es mas a nivel de implementación ya que puede variar según el entorno que se aplique