Trabajo Práctico Nº 3: Threads y sincronización

1. Primera parte: Ejercicios preliminares

1. Suponga un programa que crea *n* threads, donde cada thread posee una variable interna propia que se incrementa e imprime por pantalla en un bucle indefinidamente. En un momento dado, ¿es posible que un thread haya hecho muchas más impresiones por pantalla que otro? ¿En qué casos esto podría darse? Justifique.

2. ¿Cuáles de las siguientes clases Java A, B y C son reentrantes (thread safe)?. En caso de que no lo sean, ¿qué modificaciones podría introducir para que sí lo sean?.

```
class A {
 float square(float x) {
  return x*x;
 }
 float cube(float x) {
  return x*x*x;
 }
}

class B {
 int tmp;
 int[] swap(int[] a) {
  int len = a.length;
  if (a==null)
   return null;
  if (len<2)
   return a;
  else {
   tmp = a[0];
   a[0] = a[len];
   a[len] = tmp;
  }
  return a;
 }
}

class C {
 final double s = Math.PI/1024;
 int sin(int x) {
  double a = Math.sin(s*x);
  return a/s;
 }
 int cos(int x) {
  double a = Math.cos(s*x);
  return a/s;
 }
}
```

   Nota: Un programa es reentrante (o también thread-safe) si éste puede ser ejecutado concurrentemente por más de un hilo y a su vez esto no genera inconsistencias en sus datos.

2. Segunda parte: Ejercicios en máquina (Java)

Para completar la resolución de los ejercicios se requiere únicamente una computadora con algún JDK 7+ instalado e IDE de desarrollo, y descargar el código Java provisto en `http://www.exa.unicen.edu.ar/~cmateos/files/code.zip`. Al resolver los ejercicios, se deben evitar las clases de estructura de datos marcadas como *thread-safe* en la documentación de Java. Por ejemplo, `Vector`[1] y `Hashtable`[2]

[1] Java 7 Vector: *http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html*
[2] Java 7 Hashtable: *http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html*

son *thread-safe* por lo que se debe utilizar `ArrayList`[3] y `HashMap`[4]. Para la implementación de *mutex*, semáforos y demás estructuras de sincronización, si bien se recomienda la utilización de los monitores de Java, también pueden ser utilizadas las clases provistas en el paquete `java.utils.concurrent`.

Introducción a threads 1.0

Considere el siguiente programa:

Listado de código 1: Clase Main.java

```
1  public static void main(String[] args) {
2          int id = 0;
3          List<String> datos =
4                          Collections.synchronizedList(
5                                  new ArrayList<String>());
6          Recurso r1= new Recurso(id,
7                          Collections.synchronizedList(
8                                  new ArrayList<String>()));
9          Recurso r2= new Recurso(id, datos);
10         Recurso r3= new Recurso(id, datos);
11         Thread t1 = new Thread(new Tarea(r1,r2,1,2),"T1");
12         Thread t2 = new Thread(new Tarea(r2,r3,3,4),"T2");
13         t1.start();
14         t2.start();
15         try {
16                 t1.join();
17                 t2.join();
18         } catch (InterruptedException e) {
19                 // TODO Auto-generated catch block
20                 e.printStackTrace();
21         }
22 }
```

Listado de código 2: Clase Recurso.java

```
1  public class Recurso {
2          private int id;
3          private List<String> datos;
4
5          public Recurso(int id, List<String> datos) {
6                  this.id = id;
7                  this.datos = datos;
8          }
9
10         public void agregarDato(String dato){
11                 this.datos.add(dato);
12         }
13
14         public int getId() {
15                 return id;
16         }
17
18         public void setId(int id) {
```

[3]Java 7 ArrayList: *http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html*
[4]Java 7 HashMap: *http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html*

```
19                      this.id = id;
20              }
21
22              public List<String> getDatos() {
23                      return datos;
24              }
25
26              @Override
27              public String toString() {
28                      return "Recurso [id=" + id +
29                                      ", datos=" + datos + "]: " +
30                                      super.toString();
31              }
32      }
```

Listado de código 3: Clase Tarea.java

```
1   public class Tarea implements Runnable {
2
3           private Recurso r1;
4           private Recurso r2;
5           int id1;
6           int id2;
7
8           public Tarea(Recurso r1, Recurso r2, int id1, int id2) {
9                   super();
10                  this.r1 = r1;
11                  this.r2 = r2;
12                  this.id1 = id1;
13                  this.id2 = id2;
14          }
15
16          @Override
17          public void run() {
18                  String name = Thread.currentThread().getName();
19                  r1.setId(id1);
20                  r2.setId(id2);
21                  r1.agregarDato(name);
22                  r2.agregarDato(name);
23                  System.out.println(name+" r1:"+r1);
24                  System.out.println(name+" r2:"+r2);
25          }
26
27  }
```

En el siguiente estado de ejecución:

- Thread Main: Clase Main - línea 16
- Thread T1: Clase Tarea - línea 23
- Thread T2: Clase Tarea - línea 23

Conteste:

1. ¿Es posible? ¿Podría el thread Main ser el siguiente en ejecutar y pasar a la línea 17de la clase Main?

2. Dibuje esquemáticamente el estado de la memoria, identifique objetos, tipos primitivos, referencias y pilas de ejecución.

3. Discuta qué contendrán las variables r1, r2, y r3 de la clase Main después de ejecutar la línea 17. ¿Para qué variables es determinista el resultado (considere variables de instancia de r1, r2 y r3)?

2.1. *"Hello World!"* V2.0... ahora con *Threads*!

Escribir el código Java que escriba en pantalla n veces el mensaje: "Hola Threads!. Soy el thread [idThread] ejecutando por [n-esima] vez", donde [idThread] es el identificador de *thread* asignado por Java y [n-esima] es la cantidad de veces que se imprimió el texto por pantalla. Para obtener este identificador se puede utilizar el siguiente código:

```
int threadId = Thread.currentThread().getId();
```

Para implementar la lógica que ejecutará cada *thread* existen al menos dos alternativas. La primera, consiste en crear una nueva clase que extienda de la clase `Thread` y reimplementar el método `public void run()`. Para ejecutar el *thread* se debe crear una instancia de su clase e invocar el método `start()`. La segunda, consiste en crear una nueva clase que implemente la interfaz `Runnable` e implementar el método `public void run()`. En este caso, para ejecutar el nuevo *thread* se debe crear una instancia de la clase `Thread` y pasarle como parámetro al constructor una instancia de la clase que implementa `Runnable`. El código Java debe crear 10 *threads* y ejecutar cada uno de ellos 10.000 veces ($n = 10{,}000$). Describir brevemente el resultado de la ejecución.

2.2. Secuencial o *Multi-Thread*?... he ahí la cuestión

Existen aplicaciones que pueden aprovechar las características de los procesadores *multi-core* dado que implican realizar n tareas independientes. Un ejemplo de esto es la multiplicación de matrices. Recordando un poco de álgebra, dos matrices se pueden multiplicar si la primera es de tamaño $m * n$ y la segunda es de tamaño $n * q$, es decir, la cantidad de columnas de la primera matriz coincide con la cantidad de filas de la segunda; siendo el resultado una matriz de $m * q$.

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} * \begin{bmatrix} b_{11} & \dots & b_{1q} \\ \dots & \dots & \dots \\ b_{n1} & \dots & b_{nq} \end{bmatrix} = \begin{bmatrix} r_{11} & \dots & r_{1q} \\ \dots & \dots & \dots \\ r_{m1} & \dots & r_{mq} \end{bmatrix}$$

donde

$$r_{ij} = \sum_{k=1}^{k=n} a_{ik} * b_{kj}$$

Esto implica que el cálculo de cada elemento de la matriz r es independiente del cálculo de los otros elementos. En consecuencia,, en principio la multiplicación de matrices podría acelerarse mediante la utilización de procesadores *multi-core*.

Para realizar los ejercicios se debe utilizar el código Java que se encuentra en la carpeta MatrixMul. A continuación se describen las clases que pueden encontrarse en dicha carpeta:

- `IMatrix`: interfaz que implementa cada matriz.
- `DenseMatrix`: implementación de la interfaz `IMatrix` que utiliza una matriz de double.
- `SparseMatrix`: implementación de la interfaz `IMatrix` que utiliza tablas de *hash* para almacenar los elementos con valor distinto de cero.
- `IMultiplication`: interfaz que define un método cuyo resultado es la multiplicación de dos matrices.
- `SimpleMultiplication`: implementación de la interfaz `IMultiplication` que multiplica matrices de forma secuencial, es decir,un elemento a la vez.
- `Utils`[5]: clase con diversos métodos útiles para la resolución de los ejercicios.

[5]Consejo: si sospecha que su solución no termina de ejecutar debido a un *deadlock* puede activar un mecanismo de detección de

Ejercicios Matrices densas

1. Utilizar el método `Utils.generateDenseSquareMatrix(int)` para generar matrices cuadradas de distintos tamaños con valores aleatorios y el método `Utils.measureTime(IMatrix, IMatrix, IMultiplication)` para medir el tiempo requerido para su multiplicación. Debido a que los tiempos de ejecución pueden variar por diversos factores, como carga del sistema, o la carga de clase, se recomienda tomar al menos 10 mediciones y calcular su promedio. Busque algún tamaño de matriz tal que su multiplicación tome entre 30 segundos y 1 minuto. Algunos valores sugeridos: 500, 1000, 1500, 1800, 2000,... (recuerde que el tiempo de ejecución crece con el cubo del tamaño de las matrices).

2. Implementar la interfaz `IMultiplication` definiendo una versión *multi-thread* de la multiplicación. Es importante tener en cuenta que el cálculo de cada elemento de la matriz resultado es independiente del cálculo correspondiente a todos los otros elementos . El el funcionamiento de la multiplicación puede ser verificado utilizando el método `Utils.verifyMultiplication(IMatrix, IMatrix, IMultiplication, int)`. El último parámetro del método indica la cantidad de veces que se realiza el test, utilice el valor 10. Se recomienda utilizar este valor debido a que en software concurrentes, los errores no necesariamente afectan todas las ejecuciones. Aunque no existe ningún número de ejecuciones que garanticen que error de concurrencias (suponiendo que existan) afecte los resultados, utilizar 10 ejecuciones aumenta la posibilidad de que este ocurra.

3. Comparar los tiempos requeridos para realizar la multiplicación utilizando la implementación secuencial y la implementación *multi-thread*. Considere qué tipo de procesador tiene la computadora en la que ejecutó la implementación *multi-thread*. ¿Cuenta con procesador *multi-core*? Si es así, ¿Cuántos cores tiene dicho procesador? ¿Cuenta el procesador con tecnología orientada a incrementar la eficiencia de programas *multi-thread* (e.g. *Intel Hyperthreading*)?

Ejercicios Matrices ralas

1. Considerar nuevamente la multiplicación secuencial de matrices, pero en este caso con matrices ralas. En este caso para la generación de las matrices se debe utilizar el método `Utils.generateSparseSquareMatrix(int, double)`, donde el segundo parámetro indica el porcentaje de ceros de la escritura. Considerar un nivel de ceros del 90 %, es decir, el segundo parámetro debe ser 0,9.

2. Verificar que la implementación *multi-thread* funciona correctamente con la nueva estructura de datos. En caso de encontrar errores, reimplementar la multiplicación de forma de dar solución a el/los errores encontrados. Se sugiere la creación una nueva clase que implemente `IMultiplication`.

3. Comparar los tiempos requeridos para realizar la multiplicación utilizando la implementación secuencial y la implementación *multi-thread*. En caso de haber realizado una nueva implementación: ¿Cuál es el efecto de utilizarla con las matrices densas?

2.3. Productores y consumidores: Cuando el trabajo no lo puede hacer uno solo

Considerar el ejemplo de una aplicación del tipo productor/consumidor que se presenta en la carpeta ProdCons. A continuación se describen las clases que pueden encontrarse en dicha carpeta:

- `Producer`: Implementación de un *thread* que produce números enteros consecutivos y los deposita en un *buffer*. Además, imprime un mensaje en el error estándar cada vez que un elemento es generado.

- `Consumer`: Implementación de un *thread* que consume números enteros de un *buffer* y los imprime en la salida estándar.

- `IBuffer`: Interfaz del *buffer* utilizados por los consumidores/productores.

deadlock utilizando el método `Utils.activateDeadlockDetection()`. Limitaciones: solo detecta *deadlock* una vez que ocurren, no potenciales *deadlock*. No detecta *livelocks*, *race-conditions*, ni otros problemas de concurrencia.

Cantidad productores	Elementos a producir	Tiempo entre producción	Cantidad consumidores	Elementos a consumir	Tiempo entre consumo	Tipo de ejecución
1	100	200	1	100	200	Producen y consumen a la misma velocidad.
1	20	1000	1	20	100	Producen 10 veces más lento de lo que consume.
1	20	100	1	20	1000	Consumen 10 veces más lento de lo que produce.
1	200	100	10	20	100	Producen y consumen a la misma velocidad, pero hay múltiples consumidores.
10	20	100	1	200	100	Producen y consumen a la misma velocidad, pero hay múltiples productores.
10	60	100	10	60	100	Producen y consumen a la misma velocidad. Múltiples productores y consumidores.
10	60	1000	10	60	100	Producen 10 veces más lento de lo que consume. Múltiples productores y consumidores.
10	60	100	10	60	1000	Consumen 10 veces más lento de lo que produce. Múltiples productores y consumidores.

Cuadro 1: Configuraciones

- `CircularBuffer`: Implementación no sincronizada de un *buffer* circular de *n* elementos.
- `OneElementBuffer`: Implementación no sincronizada de un *buffer* que solo puede contener un elemento a la vez.
- `ProdConsMain`: Clase con el *main* para ejecutar el ejemplo de productores consumidores. Adicionalmente, contiene diversas variables estáticas que permiten cambiar la configuración del ejemplo. Además, existe una variable estática que permite activar la detección en ejecución de *deadlocks*.

Suponga que el desarrollador de esta aplicación no tenia conocimientos de los mecanismos de sincronización utilizados en aplicaciones *multi-thread*. Por este motivo, el desarrollador no sincronizó correctamente las estructuras de datos. En consecuencia incurrió en distintos errores que resultaron severos problemas. Probar la aplicación con las configuraciones propuestas en el Cuadro 1 y responder la guía de preguntas.

1. Utilizar alguna herramienta para ver de uso de CPU. ¿Cuál es el uso de CPU durante la ejecución? ¿Este valor se relaciona con la cantidad de trabajo útil (consumir o producir un elemento) que está realizando el programa?
2. ¿Alguna vez un consumidor consume el mismo elemento 2 veces?
3. ¿Se consume el último elemento generado por el productor?
4. Considerando el código de las clases que implementan `IBuffer`, ¿Existe algún problema con el mismo? En caso de que exista algún problema, ¿Cuál es dicho problema?

5. OPCIONAL - OPTIMIZACIONES DEL LENGUAJE. ¿Porqué las variables de instancia de las clases que implementan `IBuffer` incluyen el modificador `volatile`? ¿Qué ocurre si dicho modificador es eliminado?

Reimplementar las clases `CircularBuffer` y `OneElementBuffer`, de manera que utilicen los mecanismos correctos de sincronización provistos por Java. Adicionalmente, se debe eliminar el modificador `volatile` de las nuevas clases. Dichas clases deben llamarse `SyncCircularBuffer` y `SyncOneElementBuffer`. Finalmente, considerando la nueva implementación, responder nuevamente la guía de preguntas.