Nombre y apellido: Enric Gil Gallen

Nombre y apellido: Victor Granados Segara

Tiempo en casa: 1:10

Tema 04. Conceptos Básicos de Concurrencia en Java

1 Este ejercicio constituye una primera aproximación al ejercicio siguiente. Se desea mostrar en pantalla todos los números comprendidos entre 0 y $n-1$ (incluyendo ambos extremos), donde n es un dato recibido en la línea de argumentos. El orden en el que se muestran los datos no es importante y se desea emplear varias hebras.

En la línea de comandos, el primer argumento será el número de hebras y el número n será el segundo. Si el número de argumentos de la línea de argumentos no es correcto, el programa debe avisar y terminar. Asimismo, si algún argumento no es correcto (por ejemplo, se esperaba un argumento numérico y no lo es), el programa también debe avisar y terminar.

Seguidamente se muestra la versión inicial de los códigos a implementar.

```
// ================================================================
class EjemploMuestraNumeros1a {
// ================================================================

  // ----------------------------------------------------------------
  public static void main( String args[] ) {
    int  n, numHebras;

    // Comprobacion y extraccion de los argumentos de entrada.
    if( args.length != 2 ) {
      System.err.println( "Uso: java programa <numHebras> <n>" );
      System.exit( -1 );
    }
    try {
      numHebras = Integer.parseInt( args[ 0 ] );
      n         = Integer.parseInt( args[ 1 ] );
    } catch( NumberFormatException ex ) {
      numHebras = -1;
      n         = -1;
      System.out.println( "ERROR: Argumentos numericos incorrectos." );
      System.exit( -1 );
    }
    //
    // Implementacion paralela con distribucion ciclica o por bloques.
    //
    // Crea y arranca el vector de hebras.
    // ...
    // Espera a que terminen las hebras.
    // ...
  }
}
```

1.1) Implementa una versión paralela mediante el uso de hebras con una *distribución cíclica*. Realiza varias comprobaciones variando tanto el número de hebras como n, centrándote en los casos en los que n no es un múltiplo del número de hebras, por ejemplo $n = 13$ y $nH = 4$. Comprueba que aparecen todos los números deseados y que no hay repetidos.

Escribe a continuación la definición de la clase y el código a incluir en el programa principal.

```java
// ============================================================================
class EjemploMuestraNumeros1_1_ciclica {
// ============================================================================

   // ----------------------------------------------------------------------
   public static void main(String args[]) {
       int n, numHebras;

       // Comprobacion y extraccion de los argumentos de entrada.
       if (args.length != 2) {
           System.err.println("Uso: java programa <numHebras> <n>");
           System.exit(-1);
       }
       try {
           numHebras = Integer.parseInt(args[0]);
           n = Integer.parseInt(args[1]);
       } catch (NumberFormatException ex) {
           numHebras = -1;
           n = -1;
           System.out.println("ERROR: Argumentos numericos incorrectos.");
           System.exit(-1);
       }

       // Crea y arranca el vector de hebras.
       Ej1_MyThreadCiclica[] vectorHebras = new Ej1_MyThreadCiclica[numHebras];
       for (int i = 0; i < numHebras; i++) {
           vectorHebras[i] = new Ej1_MyThreadCiclica(i, n, numHebras);
           vectorHebras[i].start();
       }

       // Espera a que terminen las hebras.

       for (int i = 0; i < numHebras; i++) {
           try {
               vectorHebras[i].join();
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
       }

   }


}


public class Ej1_MyThreadCiclica extends Thread  {
   int miId, nElem, numHebdas;

   public Ej1_MyThreadCiclica(int miId, int nElem, int numHebdas) {
       this.miId = miId;
       this.nElem = nElem;
       this.numHebdas = numHebdas;
   }

   // Implementacion paralela con distribucion ciclica
```

```
    public void run() {
        int ini = miId;
        int fin = nElem;
        int inc = numHebdas;

        for (int i = ini; i < fin; i += inc) {
            System.out.print(i + " ");
        }

        System.out.println("--------------------");
    }
}
```

1.2) Implementa una versión paralela mediante el uso de hebras con una *distribución por bloques*. Realiza varias comprobaciones variando tanto el número de hebras como n, centrándote en los casos en los que n no es un múltiplo del número de hebras, por ejemplo $n = 13$ y $nH = 4$. Comprueba que aparecen todos los números deseados y que no hay repetidos. Escribe a continuación la definición de la clase y el código a incluir en el programa principal.

```
// ============================================================================
class EjemploMuestraNumeros1_2_bloques {
// ============================================================================

    // ------------------------------------------------------------------------
    public static void main(String args[]) {
        int n, numHebras;

        // Comprobacion y extraccion de los argumentos de entrada.
        if (args.length != 2) {
            System.err.println("Uso: java programa <numHebras> <n>");
            System.exit(-1);
        }
        try {
            numHebras = Integer.parseInt(args[0]);
            n = Integer.parseInt(args[1]);
        } catch (NumberFormatException ex) {
            numHebras = -1;
            n = -1;
            System.out.println("ERROR: Argumentos numericos incorrectos.");
            System.exit(-1);
        }

        // Crea y arranca el vector de hebras.
        Ej1_MyThreadBloques[] vectorHebras = new Ej1_MyThreadBloques[numHebras];
        for (int i = 0; i < numHebras; i++) {
            vectorHebras[i] = new Ej1_MyThreadBloques(i, n, numHebras);
            vectorHebras[i].start();
        }

        // Espera a que terminen las hebras.

        for (int i = 0; i < numHebras; i++) {
```

```
            try {
                vectorHebras[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

}

public class Ej1_MyThreadBloques extends Thread{
    int miId, nElem, numHebras;

    public Ej1_MyThreadBloques(int miId, int nElem, int numHebras) {
        this.miId = miId;
        this.nElem = nElem;
        this.numHebras = numHebras;
    }

    // Implementacion paralela con distribucion ciclica
    public void run() {

        int tamBloque = (nElem + numHebras - 1) / numHebras;
        int iniElem = tamBloque * miId;
        int finElem = Math.min(iniElem + tamBloque, nElem);

        for (int i = iniElem; i < finElem; i++)
            System.out.print(i + " ");

        System.out.println("--------------------");
    }
}
```

2 Evaluación de una función en múltiples puntos.

Se dispone de un programa secuencial que evalúa una función en varios puntos. Para almacenar datos y resultados, el código emplea dos vectores de la misma dimensión: `vectorX` y `vectorY`. El código evalúa la función para cada elemento de `vectorX` dejando el resultado calculado en el correspondiente elemento de `vectorY`. Este tipo de cálculo es muy habitual y se puede realizar por ejemplo para visualizar una función en pantalla.

Tras realizar los cálculos, en lugar de visualizar los resultados en pantalla, el programa realiza y muestra en pantalla la suma de los elementos de ambos vectores para que después se puedan comprobar fácilmente los resultados con las versiones paralelas.

El programa secuencial declara e inicializa una variable con el número de hebras, pero no la emplea para nada. Dicho valor deberá ser aprovechado en las implementaciones paralelas.

El código es el siguiente:

```
// ================================================================================
class EjemploFuncionCostosa1a {
// ================================================================================

  // --------------------------------------------------------------------------------
  public static void main( String args[] ) {
    int      n, numHebras;
    long     t1, t2;
    double   tt, sumaX, sumaY;
    double   sumaX, sumaY, ts, tc, tb;

    // Comprobacion y extraccion de los argumentos de entrada.
    if( args.length != 2 ) {
      System.err.println( "Uso: java programa <numHebras> <tamanyo>" );
      System.exit( -1 );
    }
    try {
      numHebras = Integer.parseInt( args[ 0 ] );
      n         = Integer.parseInt( args[ 1 ] );
    } catch( NumberFormatException ex ) {
      numHebras = -1;
      n         = -1;
      System.out.println( "ERROR: Argumentos numericos incorrectos." );
      System.exit( -1 );
    }

    // Crea los vectores.
    double vectorX[] = new double[ n ];
    double vectorY[] = new double[ n ];

    //
    // Implementacion secuencial.
    //
    inicializaVectorX( vectorX );
    inicializaVectorY( vectorY );
    t1 = System.nanoTime();
    for( int i = 0; i < n; i++ ) {
      vectorY[ i ] = evaluaFuncion( vectorX[ i ] );
    }
    t2 = System.nanoTime();
    ts = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Tiempo secuencial (seg.):                    " + tt );
    //// imprimeResultado( vectorX, vectorY );
    // Comprueba el resultado.
```

```
    sumaX = sumaVector( vectorX );
    sumaY = sumaVector( vectorY );
    System.out.println( "Suma del vector X:          " + sumaX );
    System.out.println( "Suma del vector Y:          " + sumaY );
/*
    //
    // Implementacion paralela ciclica.
    //
    inicializaVectorX( vectorX );
    inicializaVectorY( vectorY );
    t1 = System.nanoTime();
    // Gestion de hebras para la implementacion paralela ciclica
    // ....
    t2 = System.nanoTime();
    tc = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Tiempo paralela ciclica (seg.):          " + tc );
    System.out.println( "Incremento paralela ciclica:             " + ... );
    //// imprimeResultado( vectorX, vectorY );
    // Comprueba el resultado.
    sumaX = sumaVector( vectorX );
    sumaY = sumaVector( vectorY );
    System.out.println( "Suma del vector X:          " + sumaX );
    System.out.println( "Suma del vector Y:          " + sumaY );
    //
    // Implementacion paralela por bloques.
    //
    // ....
    //
*/

    System.out.println( "Fin del programa." );
  }

  // ---------------------------------------------------------------------
  static void inicializaVectorX( double vectorX[] ) {
    if( vectorX.length == 1 ) {
      vectorX[ 0 ] = 0.0;
    } else {
      for( int i = 0; i < vectorX.length; i++ ) {
        vectorX[ i ] = 10.0 * ( double ) i / ( ( double ) vectorX.length - 1 );
      }
    }
  }

  // ---------------------------------------------------------------------
  static void inicializaVectorY( double vectorY[] ) {
    for( int i = 0; i < vectorY.length; i++ ) {
      vectorY[ i ] = 0.0;
    }
  }

  // ---------------------------------------------------------------------
  static double sumaVector( double vector[] ) {
    double  suma = 0.0;
    for( int i = 0; i < vector.length; i++ ) {
      suma += vector[ i ];
    }
    return suma;
  }

  // ---------------------------------------------------------------------
```

```
static double evaluaFuncion( double x ) {
  return Math.sin( Math.exp( -x ) + Math.log1p( x ) );
}

// ---------------------------------------------------------------------
static void imprimeVector( double vector[] ) {
  for( int i = 0; i < vector.length; i++ ) {
    System.out.println( " vector[ " + i + " ] = " + vector[ i ] );
  }
}

// ---------------------------------------------------------------------
static void imprimeResultado( double vectorX[], double vectorY[] ) {
  for( int i = 0; i < Math.min( vectorX.length, vectorY.length ); i++ ) {
    System.out.println( "  i: " + i +
                        "  x: " + vectorX[ i ] +
                        "  y: " + vectorY[ i ] );
  }
}
```

2.1) Paraleliza el código anterior mediante el uso de hebras con una *distribución cíclica*.

Descomenta el código situado debajo de"Implementacion secuencial". Incluye la gestión de hebras que paraleliza el bucle comprendido entre la lectura de `t1` y `t2` en la versión secuencial, y la expresión que permite calcular el incremento de velocidad.

Verifica en varios casos que el nuevo código devuelve el mismo resultado que el original.

Escribe a continuación la parte de tu código que realiza tal tarea: la definición de la clase hebra y el código a incluir en el programa principal que permite gestionar los objetos de esta clase, así como la visualización de los resultados y prestaciones.

```
//
// Implementacion secuencial.
//
inicializaVectorX(vectorX);
inicializaVectorY(vectorY);
t1 = System.nanoTime();
for (int i = 0; i < n; i++) {
   vectorY[i] = evaluaFuncion(vectorX[i]);
}
t2 = System.nanoTime();
ts = ((double) (t2 - t1)) / 1.0e9;
System.out.println("Tiempo secuencial (seg.):                    " + ts);
//// imprimeResultado( vectorX, vectorY );
// Comprueba el resultado.
sumaX = sumaVector(vectorX);
sumaY = sumaVector(vectorY);
System.out.println("Suma del vector X:          " + sumaX);
System.out.println("Suma del vector Y:          " + sumaY);
System.out.println("\n");
```

```
public class Ej2_MyThreadBloques extends Thread{
   int miId, nElem, numHebdas;
   double vectorX[], vectorY[];

   public Ej2_MyThreadBloques(int miId, int nElem, int numHebdas, double[] vectorX,
double[] vectorY) {
       this.miId = miId;
       this.nElem = nElem;
       this.numHebdas = numHebdas;
       this.vectorX = vectorX;
       this.vectorY = vectorY;
   }

   // Implementacion paralela con distribucion ciclica
   public
```

2.2) Paraleliza el anterior código mediante el uso de hebras con una *distribución por bloques*.
Replica el código de la "Implementacion cíclica" y adapta la gestión de hebras. La nueva versión paralela debe ejecutarse tras las anteriores versiones.
Verifica en varios casos que el nuevo código devuelve el mismo resultado que el original.
Escribe a continuación la parte de tu código que realiza tal tarea: la definición de la clase hebra y el código a incluir en el programa principal que permite gestionar los objetos de esta clase, así como la visualización de los resultados y prestaciones.

```
//
// Implementacion paralela por bloques.
//
//
//

inicializaVectorX(vectorX);
inicializaVectorY(vectorY);
t1 = System.nanoTime();

// Crea y arranca el vector de hebras.
Ej2_MyThreadBloques[] vectorHebras_2 = new Ej2_MyThreadBloques[numHebras];
for (int i = 0; i < numHebras; i++) {
   vectorHebras_2[i] = new Ej2_MyThreadBloques(i, n, numHebras, vectorX, vectorY);
   vectorHebras_2[i].start();
}

// Espera a que terminen las hebras.

for (int i = 0; i < numHebras; i++) {
   try {
       vectorHebras_2[i].join();
   } catch (InterruptedException e) {
       e.printStackTrace();
   }
}

t2 = System.nanoTime();
tb = ((double) (t2 - t1)) / 1.0e9;
sp_b = ts / tb;
System.out.println("Tiempo paralela bloques (seg.):             " + tb);
System.out.println("Incremento paralela bloques:                " + sp_b );
//// imprimeResultado( vectorX, vectorY );
// Comprueba el resultado.
sumaX = sumaVector(vectorX);
sumaY = sumaVector(vectorY);
System.out.println("Suma del vector X:          " + sumaX);
System.out.println("Suma del vector Y:          " + sumaY);
```

2.3) Evalúa los códigos en el ordenador del laboratorio.

Copia los resultados en las siguientes tablas redondeándolos con dos decimales.

Ten en cuenta que, para poder calcular los incrementos de velocidad correctamente, se deben realizar las dos versiones paralelas (cíclica y por bloques) en una misma ejecución, comparando sus prestaciones con las de la versión secuencial.

Examina con detalle y justifica los resultados.

Incrementos de velocidad para $n = 1\ 000\ 000$			
Número de hebras	1	2	4
Distribución cíclica			
Distribución por bloques			

Incrementos de velocidad para $n = 10\ 000\ 000$			
Número de hebras	1	2	4
Distribución cíclica			
Distribución por bloques			

Incrementos de velocidad para n = 1 000 000			
Número de hebras	**1**	**2**	**4**
Distribución cíclica	1.025	1.758	3.118848783
Distribución por bloques	1.055	1.7067389841495768	3.18439524

Incrementos de velocidad para n = 10 000 000			
Número de hebras	**1**	**2**	**4**
Distribución cíclica	1.048120922107523	2.2321828018366547	3.498515109716784
Distribución por bloques	1.058848227303543 3	2.2134759241055137	3.534840909740823

En ambos casos cuando solo tenemos 1 hebra, las operaciones son similares. Las diferencias se observan cuando subimos el número de hebras y operaciones, se nota un gran aumento de velocidad en la resolución total de las operaciones, pero no se observa una gran diferencia entre cíclica y por bloques esto es debido a que como todas las operaciones tienen un coste similar da igual cual usemos.

2.4) Cuando la función **evaluaFuncion** del apartado anterior se aplica a un vector de números, ¿el código resultante está limitado por la CPU, por la memoria central o por la E/S?

Está limitado por CPU.

2.5) Si se utilizase la función **evaluaFuncion** que se muestra y se aplicase a un vector de números, entonces ¿el código estaría limitado por la CPU, por la memoria central o por la E/S?

```
static double evaluaFuncion( double x ) {
  return 3.5 * x;
}
```

Estaría limitado por la memoria central.

2.6) Haz una copia del programa que acabas de completar y dale el nombre de **EjemploFuncion Sencilla1a.java**. Modifícalo para que emplee la nueva función **evaluaFunción** más sencilla.

A continuación, calcula con dos decimales los incrementos de velocidad correspondientes para completar las siguientes tablas.

Examina con detalle y justifica los resultados.

Incrementos de velocidad para $n = 1\ 000\ 000$			
Número de hebras	1	2	4
Distribución cíclica			
Distribución por bloques			

Incrementos de velocidad para $n = 10\ 000\ 000$			
Número de hebras	1	2	4
Distribución cíclica			
Distribución por bloques			

Incrementos de velocidad para n = 1 000 000

Número de hebras	**1**	**2**	**4**
Distribución cíclica	1.682493922773414	0.8705214847218716	0.8368302492871191
Distribución por bloques	1.2509104779895919	0.6689099893699543	0.6828849742313844

Incrementos de velocidad para n = 10 000 000			
Número de hebras	**1**	**2**	**4**
Distribución cíclica	1.5971878666497277	0.7424892800366963	0.8180437178202103
Distribución por bloques	1.5873302932176705	1.2345193454439414	0.775367856415764

Mejor rendimiento en secuencial (cuando solo tenemos una hebra), esto pasa por que al tener operaciones sencillas la CPU casi no tiene uso y si se usan más hebras incrementa las operaciones de memoria, llegando al punto de no ser ni rentable.

2.7) ¿Los dos códigos (`EjemploFuncionCostosa1` y `EjemploFuncionSencilla1`) obtienen incrementos aceptables? Comenta los resultados.

- EjemploFuncionCostosa1 si que es aceptable puesto con 4 hebras (siendo el máximo de nuestra CPU) se Sp se acerca al número. Pero EjemploFuncionSencilla1 no es aceptable se puede observar que cuando mayor incremento se ha obtenido ha sido cuando solo teníamos una hebra y en ningún momento el Sp se acerca a 4.

2.8) Evalúa los códigos en patan.

Calcula con dos decimales los incrementos de velocidad necesarios para completar las siguientes tablas.

Recuerda que, para poder calcular los incrementos de velocidad correctamente, se deben ejecutar las dos versiones paralelas (cíclica y por bloques) en una misma ejecución, comparando sus prestaciones con las de la versión secuencial.

Examina con detalle y justifica los resultados.

Resultados para `evaluaFuncionCostosa` Incrementos de velocidad para $n = 100\ 000\ 000$				
Número de hebras	1	4	8	16
Distribución cíclica				
Distribución por bloques				

Resultados para `evaluaFuncionSencilla` Incrementos de velocidad para $n = 100\ 000\ 000$				
Número de hebras	1	4	8	16
Distribución cíclica				
Distribución por bloques				

Resultados para evaluaFuncionCostosa				
Incrementos de velocidad para n = 1 000 000 000				
Número de hebras	**1**	**2**	**4**	**8**
Distribución cíclica	0.992	1.964	3.564	4.089
Distribución por bloques	0.992	1.937	3.117	6.952

Resultados para evaluaFuncionSencilla				
Incrementos de velocidad para n = 1 000 000 000				
Número de hebras	**1**	**2**	**4**	**8**
Distribución cíclica	0.450	0.476	0.220	0.107
Distribución por bloques	0.894	0.879	0.785	0.613

La explicación es por lo mismo que antes