Universidad Rey Juan Carlos

Programación Concurrente
3º Grado en Ingeniería de Computadores

13 de Diciembre de 2014

Apellidos:	
Nombre:	**DNI:**

Normativa del examen

- La duración máxima del examen será de 2:30 horas.
- La hoja del examen debe contener claramente el nombre, los apellidos y el DNI del alumno.

Ejercicio 1) Preguntas cortas sobre Programación Concurrente (1.5p)

a - Describe qué son la sincronización condicional y la exclusión mutua. (0.5p)

b - ¿Qué es un semáforo? ¿Para qué se utiliza? ¿Qué operaciones tiene y cómo se comportan? (0.5p)

c - ¿Cómo se usa un semáforo para implementar la sincronización condicional y la exclusión mutua? (0.5p)

Ejercicio 2) Preguntas cortas sobre concurrencia en Java (1.5p)

a - ¿Qué son las activaciones inesperadas (*spurious wakeup*)? ¿En qué casos se pueden producir en Java? ¿Qué técnica se usa para evitar que esas activaciones afecten al correcto funcionamiento del programa? (0.5p)

b - ¿En qué se parecen y en qué se diferencian las estructuras de datos sincronizadas y las estructuras de datos concurrentes de Java? Si tenemos un mapa y queremos insertar un valor asociado a una clave sólo si no existe ya esa clave, ¿cómo se implementaría con mapa sincronizado? ¿Y con un mapa concurrente? ¿Cual de las opciones es más adecuada? (1p)

Ejercicio 3) Programa concurrente (2p)

Se pide implementar completamente un programa concurrente en Java (sin usar SimpleConcurrent) que tenga los siguientes requisitos:

1) El programa debe ejecutar 3 procesos iguales (que ejecuten el mismo método).
2) El método recibirá como parámetro un número que indicará el número de proceso.
3) Los procesos se crearán con un bucle for para pasar como parámetro el número de proceso que corresponda a cada uno de ellos.
4) Cada proceso deberá escribir en pantalla la letra ‘X’ seguida del número de proceso y quedará bloqueado a la espera de que los demás procesos también hayan escrito la letra 'X' con su número de proceso para poder continuar.
5) Los procesos escribirán la salida por pantalla de forma concurrente con los demás procesos (simulando una tarea que se puede realizar en paralelo).
6) Cuando todos los procesos hayan escrito la letra X y antes de que vuelvan a realizar esa tarea, uno de ellos deberá escribir un espacio. De esa forma, la salida de cada “fase” se separará con un espacio.
7) Las tareas que realizan los procesos se deben repetir indefinidamente.
8) Para la implementación deberá utilizarse la herramienta (o herramientas) de más alto nivel que esté disponible en la librería estándar de Java que sean adecuada para el problema.
9) Una de las posibles salidas del programa es la que se muestra a continuación (aquí se ha cortado la ejecución, pero el programa deberá mostrar este patrón indefinidamente):

```
X1X3X2 X2X1X3 X1X2X3 X3X2X1 X3X2X1 X3X1X2
```

Ejercicio 4) Programa concurrente (2p)

Se dispone de una aplicación web de juegos on-line. Para evitar que la aplicación web se sature, hay un límite máximo de partidas simultáneas, definido mediante la constante MAX_PARTIDAS. En la implementación actual, cuando se recibe una petición para crear una nueva partida, se comprueba si ya se ha alcanzado el máximo de partidas actuales. En ese caso, se devuelve un error al cliente.

Nos han pedido que modifiquemos la implementación actual para evitar en la medida de lo posible los errores enviados a los clientes. En concreto, nos piden que en el caso en que el número de partidas sea el máximo, en vez de devolver un error de forma inmediata, el hilo quede bloqueado durante un tiempo a la espera de que una de las partidas actuales finalice. Si no finaliza ninguna partida en el tiempo indicado, entonces se devolverá el error. Pero en caso de que finalice una partida antes, entonces se devolverá correctamente al cliente.

```
public class GestorMaxPartidas {

   public static final int MAX_PARTIDAS = 10;

   //Atributos necesarios

   public GestorMaxPartidas(){

   }

   //Este método será invocado cuando se quiera usar crear una nueva partida.
   //Devolverá true si se puede crear una partida (no se ha alcanzado el máximo).
   //En caso de que se haya alcanzado el máximo de partidas, se bloqueará hasta que
   //alguna partida finalice o bien se supere el tiempo de espera (waitTimeMillis).
   //Si alguna partida finaliza antes de que se supere el tiempo, se devolverá true.
   //En caso contrario, se devolverá false.
   public boolean iniciarNuevaPartida(long waitTimeMillis){

   }

   //Este método será invocado cuando una partida previamente iniciada finalice.
   public void finalizarPartida(){

   }
}
```

Se pide completar la clase anterior: incluyendo los atributos necesarios y añadiendo la implementación adecuada en los métodos y el constructor de forma que se comporte como se ha indicado previamente. No es necesario implementar ninguna clase adicional, únicamente completar la clase GestorMaxPartidas.

Ejercicio 4) Programa concurrente (3p)

Un desarrollador de C se ha pasado a Java y ha preguntado si existe algo parecido a OpenMP para Java. En concreto, a él le gustaría implementar un código como el siguiente:

```
double ProductoPunto(double* a, double* b, int size){

  double c = 0;

  #pragma omp parallel for reduction(+:c)
  for (int i = 0; i < size; ++i){
     c += a[i]*b[i];
  }

  return c;
}
```

Le decimos que no existe algo parecido a OpenMP en Java, pero que podemos implementarle un método llamado parallelForSum con el que podrá obtener un resultado parecido usando las expresiones lambda de Java 8. El anterior código C con OpenMP se podría programar de la siguiente forma en Java:

```
public double productoPunto(double[] a, double[] b, int size){

   int c = parallelForSum(size, i -> {
        return a[i] * b[i];
   });

   return c;
}
```

Le parece buena idea y nos pide que le implementemos el método parallelForSuma. Nos ponemos a ello y le implementamos primero una versión secuencial para que la empiece a usar cuanto antes:

```
public double parallelForSum(int size, Function<Integer, Double> iteration){
   double sum = 0;
   for(int i=0; i<size; i++){
      sum += iteration.apply(i);
   }
   return sum;
}
```

Tal y como se puede ver en el código, se ejecutan tantas iteraciones como haya indicado en el parámetro size. Una iteración es una invocación al método apply en el objeto iteration. El resultado del método apply se suma en la variable sum que se devuelve al finalizar el método.

Nuestro compañero no se conforma con la versión secuencial y pide que le implementemos una versión paralela del método parallelForSum. En concreto, el método tiene que comportarse de la siguiente forma:

- Tendrá que ejecutar tantas tareas en paralelo como el número de procesadores de la máquina (representado con la constante NUM_CORES).
- En cada una de esas tareas se ejecutarán una parte de las iteraciones y se sumará su resultado. Para que la implementación no quede muy complicada, supondremos que size es múltiplo de NUM_CORES, es decir, cada tarea tiene que realizar las mismas iteraciones.
- Cuando termine la ejecución de todas las tareas, se sumará el resultado de cada una de ellas y se devolverá el resultado.