PROGRAMACIÓN CONCURRENTE

EXAMEN EXTRAORDINARIO

Grado en Ingeniería de Computadores. Universidad Rey Juan Carlos

14 junio 2019

Nombre y apellidos: _____

DNI: _____

Entrega esta hoja y todas las demás que hayas necesitado. Pon el nombre en todas ellas y numéralas.

1. (1,25 punto) Explica lo que es la Ley de Moore y su posible relevancia en la actualidad.

Solución: lo esencial de las diapositivas 42, 44 y 46 del tema 1.

2. (1,25 punto) Enumera y describe las distintas arquitecturas de sistemas concurrentes.

Solución: describir lo esencial de las diapositivas 34-37, 47, y 49 del tema 1.

3. (0,5 puntos) ¿Cuál es el algoritmo que implementa la entrada y salida a una sección crítica con espera activa, para 2 procesos? ¿Y para N procesos?

El algoritmo de Dekker (2 procesos) y Lamport (N procesos). Diapositiva 90 del tema 2.

4. (1 punto) En el contexto del problema de los filósofos comilones, tenemos este código, hecho con los semáforos de SimpleConcurrent:

```
public static void filosofo(int numFilosofo) {

       while(true){
           printlnI("Pensar");

           int tIzq = numFilosofo;
           int tDer = (numFilosofo+1) % N_FILOSOFOS;

           semaforoSeccionCritica.acquire();
           tenedores[tIzq].acquire();
           tenedores[tDer].acquire();
           semaforoSeccionCritica.release();

           printlnI("Comer");

           semaforoSeccionCritica.acquire();
           tenedores[tIzq].release();
           tenedores[tDer].release();
           semaforoSeccionCritica.release();
       }
   }
```

Indica si este método (enclavado en el resto del software necesario para el problema de los filósofos) cumple o no con la funcionalidad y las restricciones típicas del problema de los filósofos. Justifica totalmente tu respuesta.

Solución: no cumple con la funcionalidad porque puede haber bloqueo mutuo. Efectivamente, si un filósofo intenta coger un tenedor que ya está siendo usado por otro filósofo, se quedará esperando a que sea liberado. Pero como se queda esperando dentro de la sección crítica, el filósofo que tiene el tenedor no puede liberarlo, al no poder entrar en la sección crítica.

Aunque resolviéramos este tema, quedaría aún un problema más, que puede seguir produciendo interbloqueo. Se trata de que todos los filósofos intenten coger a la vez su tenedor de la derecha o de su izquierda… todos quedarían bloqueados a la vez. Esto se resolvería haciendo que uno de ellos varíe el orden en que coge sus tenedores, o bien imponiendo que sólo puedan entrar al comedor NFILOSOFOS-1 a la vez.

5. (3 puntos) Implementa un programa capaz de sumar (mediante N hilos que se ejecutan de forma concurrente) todos los componentes de un array que contenga M números de tipo float. Las únicas variables compartidas serán: el propio array, un float que exprese el resultado final, y un int que exprese cuántos componentes del array hemos sumado ya. Implementa esta solución con Java, utilizando únicamente semáforos. Tanto N como M son constantes y pueden tener cualquier valor. Todos los hilos ejecutarán el mismo método. El "main" del programa tiene que ser capaz de imprimir el resultado final por pantalla (el valor de la variable compartida de tipo float, ya mencionada anteriormente, que expresa el resultado final). Se valorará que la solución dé nombres descriptivos a las variables y a los métodos, y divida el código correctamente en métodos. Se valorará el grado de concurrencia alcanzado. Te recomiendo que lo hagas en borrador y, una vez esté todo claro, lo pases a limpio (y me entregas sólo lo limpio, no el borrador).

Una posible solución sencilla pero válida sería la siguiente (grados menores de concurrencia tendrán penalización):

```
import java.util.concurrent.Semaphore;

public class SumaConcurrente {

	private static final int N = 5; // Numero de procesos
	private static final int M = 5000; // Tamano del buffer

	private float[] buffer = new float[M];
	private int contador = 0;
	private float resultado = 0;

	private Semaphore regionCriticaContador = new Semaphore(1);
	private Semaphore regionCriticaResultado = new Semaphore(1);
	private Semaphore numeroTerminados = new Semaphore(0);

	public void sumar() {
		try {
			while (true) {
				int posicion=0;
				boolean acabado=false;

				// Examinamos el contador en su propia sección crítica
				regionCriticaContador.acquire();
				if (contador<M) {
					posicion=contador;
					contador++;
				}
				else acabado=true;
				regionCriticaContador.release();

				// Si todo es correcto, obtenemos el dato. Si no, acabamos
				if (acabado) break;
				float dato = buffer[posicion];
```

```
                                    // Actualizamos el resultado en su propia sección crítica
                                    regionCriticaResultado.acquire();
                                    resultado = resultado + dato;
                                    regionCriticaResultado.release();
                            }
                    } catch (InterruptedException e) {}
                    finally {numeroTerminados.release();} // Notificamos que este hilo ha terminado
            }



            public void exec () throws InterruptedException {
                    // Iniciamos el buffer con numeros aleatorios
                    for (int i=0; i<M; i++) buffer[i]=(float)Math.random();

                    // Iniciamos los hilos
                    for (int i = 0; i < N; i++) {
                            new Thread(()->sumar(),"Sumador " + i).start();
                    }

                    // Esperamos hasta que todos los procesos hayan terminado, e imprimimos resultado
                    numeroTerminados.acquire(N);
                    System.out.println("Resultado: " + resultado);
            }

            public static void main(String[] args) throws InterruptedException {
                    new SumaConcurrente().exec();
            }
}
```

6. (3 puntos) Con executors (con un pool fijo de N hilos), implementar la ejecución concurrente de M tareas iguales (que ejecutan el mismo método). En concreto, realiza los siguientes dos apartados:

1. (1,5 puntos) Cada tarea se duerme un tiempo aleatorio y después imprime su número de tarea por pantalla. Cada tarea no devuelve nada.

Solución:

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Executor1 {
            public static final int N_TAREAS = 5;

            public void ejecutaTarea(int numTask) {
                    long tiempoDormida = (long) (Math.random() * 10000);
                    try {Thread.sleep(tiempoDormida);}
                    catch (InterruptedException e) {}
                    System.out.println("Tarea " + numTask + " ha estado " + tiempoDormida + " ms dormida");
            }

            public void exec() throws InterruptedException, ExecutionException {
                    ExecutorService executor = Executors.newFixedThreadPool(N_TAREAS);
                    try {
                            // Enviamos las tareas al executor
                            for (int i = 0; i < N_TAREAS; i++) {
                                    int numTask = i;
                                    executor.submit(() -> ejecutaTarea(numTask));
                            }
                    } finally {executor.shutdown();}
            }

            public static void main(String[] args) throws InterruptedException, ExecutionException {
                    new Executor1().exec();
            }
}
```

2. (1,5 puntos) Mejora el apartado anterior para que, cada tarea, también devuelva el tiempo aleatorio que ha estado dormida. El main deberá imprimir dicho tiempo aleatorio por pantalla a medida que cada tarea vaya terminando.

Solución:

```
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Executor2 {
	public static final int N_TAREAS = 10;

	public long ejecutaTarea(int numTask) {

		// Dormimos un tiempo aleatorio
		long tiempoDormida = (long) (Math.random() * 10000);
		try {Thread.sleep(tiempoDormida);}
		catch (InterruptedException e) {}

		// Imprimimos tiempo y lo devolvemos
		System.out.println("Tarea " + numTask + " ha estado " + tiempoDormida + " ms dormida");
		return (tiempoDormida);
	}

	public void exec() throws InterruptedException, ExecutionException {
		ExecutorService executor = Executors.newFixedThreadPool(N_TAREAS);
		try {
			CompletionService<Long> completionService =
					new ExecutorCompletionService<>(executor);

			// Enviamos las tareas al executor
			for (int i = 0; i < N_TAREAS; i++) {
				int numTask = i;
				completionService.submit(() -> ejecutaTarea(numTask));
			}

			// Recogemos el resultado de cada tarea a medida que van terminando
			for (int i = 0; i < N_TAREAS; i++) {
				Future<Long> completedTask = completionService.take();
				System.out.println("Task: " + completedTask.get());
			}
		} finally {executor.shutdown();}
	}

	public static void main(String[] args) throws InterruptedException, ExecutionException {
		new Executor2().exec();
	}
}
```