EXAMEN ORDINARIO DE PROGRAMACIÓN CONCURRENTE

Grado en Ingeniería de Computadores, URJC
9 de enero de 2020
Este examen cuenta 8 puntos en la nota final ordinaria de la asignatura

1. (1,5 puntos) Cuando tenemos un sólo procesador, ¿tiene alguna utilidad la programación concurrente? Razona tu respuesta.

   Solución: tiene utilidad cuando los procesos hacen uso intensivo de E/S u otros recursos por los que hay que esperar

2. (1,5 puntos) Queremos hacer una sincronización de barrera cíclica de 4 procesos ("procesoA()") que simplemente imprimen una "A" por pantalla. Razona y explica qué es lo que está mal en el siguiente código. Los semáforos "emNProcesos" y "sb" están inicializados a 1 y 0 respectivamente. La variable compartida "nProcesos" cuenta el número actual de procesos que han impreso la "A" en el ciclo actual.

```
public static void procesoA() {
  while(true){
    print("A");
    sincronizacion();
  }
}

public static void sincronizacion(){
  emNProcesos.acquire();
  nProcesos++;
  if (nProcesos < 4) {
    emNProcesos.release();
    sb.acquire();
  } else {
    println("-");
    nProcesos = 0;
    for (int i = 0; i < 3; i++) sb.release();
    emNProcesos.release();
  }
}
```

   Solución: explicar en detalle lo que se resume en la diapositiva 188 del tema 2. En este ejercicio no se pide cómo corregir el código, sino cuál es el problema del código y explicar dicho problema.

   No podemos estar seguros de que, cuando liberamos 3 veces sb, los 3 procesos estén realmente bloqueados en sb. Puede pasar que sólo uno de ellos esté bloqueado... le desbloqueamos y además incrementamos el semáforo en 2 unidades más, por lo que este proceso podría pasar otras dos veces seguidas por el sb.acquire() sin esperar a los demás.

3. (2 puntos) Implementar un programa concurrente formado por un proceso servidor, un proceso proxy y un proceso cliente. El proceso cliente genera un número aleatorio, se lo envía al proxy y espera su respuesta. Cuando la recibe, la imprime por pantalla y vuelve a empezar, generando un nuevo número aleatorio y enviándoselo al proxy. El proceso proxy no hace nada hasta que recibe una petición. Cuando la recibe, suma 1 al número enviado por el cliente y hace una petición al servidor con ese número, esperando su respuesta. Cuando el proceso servidor reciba la petición, le suma 1 y se la envía de vuelta al proxy. Cuando el proxy recibe la respuesta del servidor, se la reenvía al cliente. Es necesario usar únicamente objetos Exchanger para implementar este programa. Además, hay que hacerlo con el mínimo número posible de objetos Exchanger. Si no se cumple alguno de estos dos requisitos, el ejercicio será evaluado con cero. Se valorará el orden, la claridad, la buena estructuración y la buena nomenclatura en el código.

Solución: ejercicio 5.5.8 pero con sólo 2 objetos Exchanger: uno para las comunicaciones entre el cliente y el proxy, y otro para las comunicaciones entre el proxy y el server.

```
import java.util.concurrent.Exchanger;

public class ClienteServidorProxyN {

	private Exchanger<Double> comunicacionClienteProxy = new Exchanger<>();
	private Exchanger<Double> comunicacionProxyServer = new Exchanger<>();

	public void client() {
		try {
			for (int i = 0; i < 10; i++) {
				double datoPeticionProxy = i;
				comunicacionClienteProxy.exchange(datoPeticionProxy);
				double datoRespuestaProxy = comunicacionClienteProxy.exchange(null);
				System.out.println("Response: " + datoRespuestaProxy);
			}
		} catch (InterruptedException e) {
		}
	}

	public void proxy() {
		try {

			for (int i = 0; i < 10; i++) {
				double datoPeticionProxy = comunicacionClienteProxy.exchange(null);
				double datoPeticion = datoPeticionProxy + 1;
				comunicacionProxyServer.exchange(datoPeticion);
				double datoRespuesta = comunicacionProxyServer.exchange(null);
				comunicacionClienteProxy.exchange(datoRespuesta);
			}
		} catch (InterruptedException e) {
		}
	}

	public void server() {
		try {
			for (int i = 0; i < 10; i++) {
				double datoPeticion = comunicacionProxyServer.exchange(null);
				double datoRespuesta = datoPeticion + 1;
				comunicacionProxyServer.exchange(datoRespuesta);
			}
		} catch (InterruptedException e) {
		}
	}

	public void exec() {
		new Thread(() -> client()).start();
		new Thread(() -> proxy()).start();
		new Thread(() -> server()).start();
	}

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

4. (3 puntos) Utilizando el esquema y las herramientas de "Java Fork Join" (en caso contrario, el ejercicio se evaluará con cero), calcula el término i-ésimo de la sucesión de Fibonacci. Recuerda que el término i-ésimo de la sucesión de Fibonacci se calcula recursivamente así: F(i) = F(i-1) + F(i-2), siendo F(0)=0 y F(1)=1. Se valorará el orden, la claridad, la buena estructuración y la buena nomenclatura en el código.

Solución: Como el ejemplo de calcular el máximo, pero con Fibonacci (mucho más sencillo que calcular el máximo).

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ParallelForFibonacci {

	public class Fibonacci extends RecursiveTask<Integer> {

		private int i; // El termino de fibonacci que vamos a calcular

		public Fibonacci(int i) {
			this.i = i;
		}

		protected Integer compute() {
			if (i==0 || i==1) {
				return(i);
			} else {
```

```
                        Fibonacci i1 = new Fibonacci(i-1);
                        i1.fork();

                        Fibonacci i2 = new Fibonacci(i-2);
                        i2.fork();

                        return(i1.join()+i2.join());
                }
        }
    } // Fin clase Fibonacci

    public static void main(String[] args) throws InterruptedException {
            ForkJoinPool pool = new ForkJoinPool(4); // 4 hilos por ejemplo
            Fibonacci task = new Fibonacci(10); // Calculamos el termino 10
            System.out.println("Termino 10: " + pool.invoke(task));

    }

} // Fin del programa
```