PROBLEMAS DE CONCURRENCIA

PROBLEMA 1

Se ha diseñado una sencilla aplicación que pretende mostrar por pantalla la secuencia de números enteros desde 1 hasta 10 usando 10 hilos, de forma que cada hilo realice el incremento de un contador compartido y lo muestre por pantalla.

PROBLEMA 2

Basándonos en el problema anterior, se desea modificar la aplicación de la forma siguiente:

- La clase principal, después de lanzar todos los hilos, debe esperar a que estos terminen y mostrar el valor del contador.

- Cada hilo debe incrementar el contador 5 veces.

Se pide: realizar las modificaciones oportunas de forma que los hilos se ejecuten con la máxima concurrencia.

PROBLEMA 3

Un bar quiere limitar el aforo. Para ello, se ha instalado una puerta de entrada automática, que siempre está cerrada, junto con un pulsador. Cuando un cliente quiere entrar, acciona el pulsador de la puerta de entrada y, si el bar no tiene su aforo completo, la puerta se abre y se ilumina un letrero indicando que puede entrar. En caso contrario, el cliente tendrá que esperar fuera hasta que se encienda el letrero. Cuando un cliente quiere salir, utiliza una puerta distinta a la de entrada (para evitar que algún cliente intente colarse). Esta puerta permanece cerrada y tiene un pulsador que funciona de forma análoga a la de entrada, con la diferencia de que siempre que se acciona el pulsador la puerta se abre automáticamente. Cuando sale un cliente, si había clientes esperando para entrar, se iluminará el letrero que indica que se puede entrar, y la puerta de entrada se abrirá para permitir la entrada de uno de los que están esperando entrar.

En el ordenador central se ejecuta una aplicación java multihilo para controlar el sistema descrito. Esta aplicación está compuesta por las siguientes clases:

Clase Cliente. Los clientes se representarán mediante hilos instanciados a partir de esta clase.

```
public class Cliente implements Runnable
{

Bar bar;
public Cliente (Bar bar) {
```

```
this.bar=bar;
}
public void run() {
bar.entrar(); // Esta acción simula accionar el pulsador de
// entrada y esperar a que se abra la puerta
// Estar en el bar
bar.salir(); // Esta acción simula accionar el pulsador de
// salida y esperar a que se abra la puerta.
}
}
```

Clase Bar. Al arrancar la aplicación, se instancia un objeto de esta clase. Contiene los métodos a los que invocan los hilos que representan a los clientes.

```
public class Bar
{
// Atributos para controlar el acceso al bar.
/**
* @param aforo Nº de clientes que caben en el local
*/
public Bar (int aforo)
{
// inicializar el bar
}
public void entrar()
{
// Permitir la entrada de un cliente si no se ha alcanzado
// el aforo, y quedarse bloqueado en caso contrario.
abrirPuertaE(); // Muestra el letrero de “se puede entrar” y
// abre la puerta de entrada
}
public void salir()
{
abrirPuertaS(); // Muestra el letrero de “se puede salir” y
// abre la puerta de salida
// Anotar que un cliente sale del bar, y desbloquear a
// quien pudiera estar esperando, para que entre uno más
}
}
```

Suposiciones:

- Los clientes que han pulsado el pulsador para entrar, siempre esperan (no se van) hasta que el indicador les diga que pueden entrar, y entonces entran.
- Los clientes siguen el protocolo de entrada y salida de forma estricta, es decir, siempre entran y salen de uno en uno por las puertas.

Se pide: En la clase Bar implementar los métodos entrar() y salir(), añadir los atributos que sean necesarios y añadir al constructor las acciones que sean precisas.

Nota: Por simplicidad no capturar o tratar las excepciones que arrojan las herramientas de concurrencia utilizadas para resolver el ejercicio.

PROBLEMA 4

En el bar “Cañas gratis” hay barra libre de cerveza. Para no dedicar un camarero a servir cañas, se han colocado varios grifos conectados a un mismo barril, que los clientes pueden usar libremente. El problema que tiene el dueño del local es cómo enterarse de que se ha acabado la cerveza y que hay que cambiar el barril. Para ello, ha diseñado un mecanismo que funciona de la siguiente manera:

- Cuando un cliente se sirve su caña, el grifo informa a un ordenador, de forma que se puede llevar la cuenta de las cañas servidas.
- Cuando el barril está agotado, se enciende un letrero en el grifo con el mensaje “Esperar al cambio de barril” y se avisa al camarero para que cambie el barril.

En el ordenador central se ejecuta una aplicación java multihilo para controlar el sistema descrito. Esta aplicación está compuesta por las siguientes clases:

Clase Cliente. Los clientes se representarán mediante hilos instanciados a partir de esta clase. Como hay clientes que beben más que otros, se ha simulado su comportamiento con un argumento del constructor que indica el nº de cañas que va a tomar cada cliente:

```
public class Cliente implements Runnable
{
	Bar bar;
	int cañas;
	public Cliente (Bar bar, int cañas) {
		this.cañas=cañas;
		this.bar=bar;
	}

	public void run() {
		for (int i=0;i<cañas;i++)
		{	// Disfrutar hasta tener sed
			bar.tomarCaña(); // Esta acción simula elegir
			alguno de
		} // los grifos disponibles para servirse
			// una caña.
	}
}
```

Clase Camarero. Implementa el comportamiento del camarero encargado de reponer el barril cuando se agota. Se creará un hilo instanciado a partir de esta clase:

```
public class Camarero implements Runnable
{
	Bar bar;
	public Camarero (Bar bar) {
		this.bar=bar;
	}

	public void run()
	{
	while (true)
		bar.reponer(); // Esperar a que le avisen que se ha
		// acabado el barril y entonces reponerlo.
	}
}
```

Clase Bar. Al arrancar la aplicación, se instancia un objeto de esta clase. Se considera que, inicialmente, hay un barril lleno instalado. Contiene los métodos a los que invocan los hilos que representan a los clientes y al camarero:

```
public class Bar
{
	// Atributos para controlar las cañas que se sirven de cada barril
	// y cuándo debe ser repuesto por uno nuevo.

	/**
	* @param nCañas Nº de cañas que pueden ser servidas de un barril
	*/
	public Bar (int nCañas)
	{
		// inicializar el bar
	}

public void tomarCaña ()
{
	// Si el barril no está vacío, servirá la caña directamente.
	// En caso contrario, avisará al camarero para que lo reponga, y
	// se quedará bloqueado hasta que se haya cambiado el barril, y
	// entonces servirá la caña.
	servirCaña(); // Servir una caña
	// anotar que se ha servido una caña
}

public void reponer()
{
	// Esperar a que le avisen de que hay que cambiar el barril

cambiarBarril(); // Esta acción indica que se repone el barril
// anotar que se ha repuesto el barril
	}
}
```

Suposiciones:

- Cuando un cliente obtiene que el barril está agotado se quedará esperando hasta que se produzca el cambio de barril y luego tomará su caña.
- El camarero siempre está pendiente de que le avisen del cambio de barril y lo cambia cuando se lo soliciten.

Se pide: En la clase Bar implementar los métodos tomarCaña() y reponer(), añadir los atributos que sean necesarios y añadir al constructor las acciones que sean precisas.
Usar las herramientas Región Crítica Condicional, monitores o bien cerrojos con variables de condición.

PROBLEMA 5

Una empresa utiliza un ordenador con NUM_CPU procesadores para ejecutar aplicaciones de cálculo intensivo de problemas que se pueden paralelizar. Una de esas aplicaciones es el programa en Java que se describe en este ejercicio.
El programa utiliza una clase llamada **Calculadora** para realizar ciertos cálculos. Esta clase tiene un constructor sin parámetros y el siguiente método público:

```
public boolean calcularParte();
// Realiza el cálculo sobre una cierta parte de los datos.
// Devuelve true si ya ha terminado completamente el cálculo y false si
// hay que seguir calculando.
```

La codificación interna de la clase Calculadora no es relevante para este problema. En una aplicación real habría que darle a este objeto alguna información sobre lo que tienen que calcular, pero ello se omite completamente en este ejercicio por simplicidad.
Para aprovechar la potencia bruta del ordenador, la aplicación va a utilizar múltiples hilos (hasta un máximo de NUM_CPU), de manera que cada uno de ellos trabajará con un objeto Calculadora diferente. Estos hilos se implementarán mediante la clase **HiloCalculador**, y estarán en un bucle invocando al método calcularParte() hasta que este devuelva true. La codificación parcial de lo que deben realizar estos hilos es:

```
Calculadora calculadora = new Calculadora();
do {
terminar = calculadora.calcularParte();
} while (! terminar);
```

Los cálculos que realiza el método calculadora.calcularParte() son independientes, por lo que a la hora de invocar a este método no es necesario establecer ningún mecanismo de sincronización especial.
La clase Aplicación, que tiene el programa principal (método main()) se encargará de lanzar los hilos (el nº de hilos es una constante) y a continuación entrará en un bucle en el que se le muestra un menú al usuario para que decida qué hacer. El siguiente fragmento de código muestra lo comentado en este párrafo:

```
public class Aplicación {
private static final int NUM_CPU = 10;
public static void main(String[] args) {
int opcion;
// Lanzar NUM_CPU hilos

      break;
}
} while (opción!=3)
}
}
```

El método pedirOpcion() presenta un menú al usuario para que escoja entre tres opciones: suspender un hilo calculador, y reanudar todos los hilos calculadores. Este método no es relevante para el problema.
La opción de suspender un hilo calculador significa que el programa debe hacer que uno de los hilos calculadores deje de calcular hasta nueva orden. Esta opción se utiliza para reducir temporalmente la demanda de CPU de esta aplicación porque se quiere ejecutar otra aplicación en el ordenador. El hilo suspendido no debe terminar su ejecución (pues se perdería el trabajo que ha hecho hasta el momento), sino que debe terminar el último cálculo y quedarse a la espera hasta nueva orden.
La opción de reanudar todos los hilos calculadores significa que el programa debe reanudar todos los hilos calculadores que estén suspendidos, es decir, que los hilos en cuestión proseguirán con la ejecución de su bucle de cálculo. Si todos los hilos de cálculo ya estuvieran activados, esta opción no debe provocar un fallo en el programa.
El usuario puede dar varias órdenes de suspender y reanudar consecutivas antes de que algún hilo calculador procese las solicitudes del usuario. Las órdenes de suspensión son acumulativas, es decir, que dos órdenes de suspensión consecutivas implican que dos hilos deben suspenderse. Por otro lado, cada orden de reanudación dejará sin efecto todas las órdenes de suspensión pendientes.

Se pide:
Codifique completamente las clases Aplicación y HiloCalculador para satisfacer la funcionalidad descrita en el enunciado.

Nota: Para resolver este ejercicio no se deben usar los métodos Thread.suspend() y/o Thread.resume().

PROBLEMA 6

El sistema de actualización de cuentas corrientes de una entidad bancaria está implementado mediante una aplicación Java multihilo. Cada vez que se necesita realizar una operación con cuentas corrientes, se crea un nuevo hilo para ejecutarla. Se ha diseñado (parcialmente) la clase Cuenta con las siguientes características:

```
public class Cuenta {
	private final String numCuenta;
	private int saldo;
	private Cliente cliente;
	…
	public void ingreso ( int cuánto ) {
	…
	}
	public void reintegro ( int cuánto ) throws SaldoInsuficienteException {
	…
	}
	public void transferencia (int cuánto,Cuenta destino) throws
	SaldoInsuficienteException{
	…
	}
}
```

Se pide:
Codificar los métodos ingreso, reintegro y transferencia. Tenga en cuenta que se debe garantizar que estas operaciones no puedan llevar nunca a un interbloqueo por actualizaciones concurrentes de las cuentas.