SEMÁFOROS

CONCURRENCIA = EJECUCIÓN SIMULTÁNEA + INDETERMINISMO + INTERACCIÓN

INTERACCIÓN = COMUNICACIÓN + SINCRONIZACIÓN

SINCRONIZACIÓN = EXCLUSIÓN MUTUA + SINCRONIZACIÓN POR CONDICIÓN

Los algoritmos de **espera activa** pueden garantizar **exclusión mutua**, pero son difíciles de implementar, difíciles de probar su corrección, malgastan la CPU y dependen de la arquitectura donde se ejecutan (entre otros muchos problemas). Por ello nace la necesidad de encontrar una mejor solución para implementar la **exclusión mutua**.

Semáforos:

Son TADs que permiten detener o dejar pasar un proceso dependiendo del valor de su contador interno. Los semáforos detienen los threads sin efectuar espera activa, permitiendo desbloquear uno de los hilos que se encuentren bloqueados. Se basan en test-and-set y son de más alto nivel que la espera activa, ya que simplifica los protocolos de E/S en la sección crítica. Por ello, decimos que los semáforos resuelven fácilmente la exclusión mutua.

Si el semáforo está "abierto", entonces se permite el paso del thread; mientras que si está "cerrado" se detiene el thread. Tiene un atributo contador de tipo entero que se usa para decidir si un proceso puede "pasar" o no. Este valor se inicializa en el constructor. Tiene dos operaciones:

await - cclib **acquire - java.util.concurrent.Semaphore**	**signal - cclib** **release - java.util.concurrent.Semaphore**

`Semaphore s;`
 	
`void await() {`
`  if(s.contador > 0)`
`  s.contador--;`
`  else`
`  block();`
`}`
	`void signal() {`
	`  if(!procesos_bloqueados)`
	`  s.contador++;`
	`  else`
	`  unblock(t);`
	`}`

Cuando el contador tiene un valor mayor que 0, únicamente decrementa el contador; cuando vale 0 entonces el proceso se queda bloqueado	Cuando no hay procesos bloqueados se incrementa el contador; cuando hay procesos bloqueados se desbloquea uno de esos procesos

La decisión del thread que se despierta en un semáforo es no-determinista (podría darse el caso de que un thread sufra inanición). En las implementaciones se suele hacer en orden de llegada para garantizar que el proceso que se haya bloqueado primero, sea el primero que se desbloquee. Esta implementación es la que se sigue en la librería `cclib`

Para razonar el comportamiento de un semáforo, debemos suponer que cualquier hilo que esté bloqueado puede ser desbloqueado en un `signal`

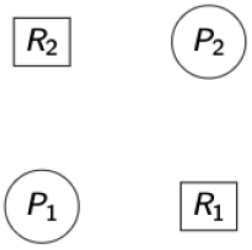
¿Verdadero o falso?

(1) Un semáforo se puede inicializar con un valor del contador negativo
(2) Un semáforo inicializado con un valor mayor que 0 puede tener s.contador < 0
(3) Si hay un thread bloqueado, entonces s.contador == 0
(4) Si s.contador == 0, entonces hay un thread bloqueado
(5) Si s.contador > 0, entonces no hay threads bloqueados
(6) A través de un semáforo se puede bloquear un thread distinto al que está ejecutando
(7) A través de un semáforo se puede desbloquear un thread distinto al que se está ejecutando
(8) El valor máximo de s.contador no está acotado

Interbloqueo:

Describe una situación en la que hay dos o más procesos bloqueados esperándose entre sí indefinidamente, es decir, hay más de un proceso compitiendo por el mismo recurso y ambos están esperando a que el otro termine con el recurso, y ninguno de ellos lo hace.

Se pueden intentar detectar con un grafo donde:
(1) Los recursos y los procesos son nodos
(2) Cuando un proceso adquiere un recurso → arista del recurso al proceso
(3) Cuando un proceso solicita un recurso → arista del proceso al recurso



a) P1 adquiere R1
b) P2 adquiere R2
c) P1 solicita R2
d) P2 solicita R1

Problema de la cena de los filósofos (dining philosophers problem):

Es un problema clásico de las ciencias de la computación propuesto por Edsger **Dijkstra** en 1965 para representar el problema de la sincronización de procesos en un SO. Cabe aclarar que la interpretación está basada en pensadores chinos, quienes comían con dos palillos, donde es más lógico que se necesite el del comensal que se siente al lado.

"Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.

Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o deadlock.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre."

¿Qué pasa en cada uno de estos escenarios posibles?

```
public static Semaphore s = new Semaphore(0);

 proceso A       proceso B     proceso C
 A1;             B1;           C1;
 s.await();      B2;           C2;
 A2;
```

```
public static Semaphore s = new Semaphore(1);

 proceso A       proceso B     proceso C
 A1;             B1;           C1;
 s.await();      B2;           C2;
 A2;
```

```
public static Semaphore s = new Semaphore(0);

 proceso A       proceso B     proceso C
 A1;             B1;           C1;
 s.signal();     B2;           C2;
 A2;
```

```
public static Semaphore s = new Semaphore(0);

 proceso A       proceso B     proceso C
 A1;             B1;           C1;
 s.signal();     s.await();    C2;
 A2;             B2;
```

```
public static Semaphore s = new Semaphore(0);

 proceso A       proceso B     proceso C
 A1;             B1;           C1;
 s.signal();     s.await();    s.await()
 A2;             B2;           C2;
```

ENTREGABLE 4

2. Garantizar exclusión mutua con semáforos

Este ejercicio, al igual que el anterior, consiste en evitar una condición de carrera. En esta ocasión tenemos el mismo número de procesos incrementadores que decrementadores que incrementan y decrementan, respectivamente, en un mismo número de pasos una variable compartida. El objetivo es asegurar la exclusión mutua en la ejecución de los incrementos y decrementos de la variable y el objetivo es hacerlo utilizando exclusivamente un semáforo de la clase `es.upm.babel.cclib.Semaphore` (está prohibido utilizar cualquier otro mecanismo de concurrencia). La librería de concurrencia `cclib.jar` puede descargarse de la página web de la asignatura.

Material a entregar: el fichero fuente a entregar debe llamarse `CC_04_MutexSem.java`

Material de apoyo: se ofrece el fichero `CC_04_MutexSem.java` con un esqueleto que debe respetarse y que muestra una condición de carrera:

```
import es.upm.babel.cclib.Semaphore;

public class CC_04_MutexSem {
 private static int N_THREADS = 2;
 private static int N_PASOS = 1000000;

 static class Contador {
       private volatile int n;
       public Contador() {
             this.n = 0;
       }
       public int valorContador() {
             return this.n;
       }
       public void inc() {
             this.n++;
       }
       public void dec() {
             this.n--;
       }
 }
 static class Incrementador extends Thread {
       private Contador cont;
       public Incrementador(Contador c) {
             this.cont = c;
       }
       public void run() {
             for(int i=0; i<N_PASOS; i++)
                   this.cont.inc();
       }
 }
```

```
	static class Decrementador extends Thread {
		private Contador cont;
		public Decrementador(Contador c) {
			this.cont = c;
		}
		public void run() {
			for(int i=0; i<N_PASOS; i++)
				this.cont.dec();
		}
	}

	public static void main(String[] args) {

		// Creacion del objeto compartido
		Contador cont = new Contador();

		// Creacion de los arrays que contendran los threads
		Incrementador[] tInc = new Incrementador[N_THREADS];
		Decrementador[] tDec = new Decrementador[N_THREADS];

		// Creacion de los objetos threads
		for(int i=0; i<N_THREADS; i++) {
			tInc[i] = new Incrementador(cont);
			tDec[i] = new Decrementador(cont);
		}

		// Lanzamiento de los threads
		for(int i=0; i<N_THREADS; i++) {
			tInc[i].start();
			tDec[i].start();
		}

		// Espera hasta terminacion de los threads
		try {
			for(int i=0; i<N_THREADS; i++) {
				tInc[i].join();
				tDec[i].join();
			}
		} catch (Exception ex) {
			ex.printStackTrace();
			System.exit(-1);
		}

		// Simplemente se muestra el valor final de la variable
		System.out.println(cont.valorContador());
		System.exit(0);
	}

}
```

EJERCICIOS DE EXAMEN

(1.5 puntos) 1. Dado un programa concurrente en la que tres threads instancias de las clases A, B y C comparten una variable n:

```
volatile int n = 0;
static Semaphore s1 = new Semaphore(2);
static Semaphore s2 = new Semaphore(0);
static Semaphore s3 = new Semaphore(0);
```

```
static class A
 extends Thread {
  public void run() {
    s2.await();
    n = n * 2;
    s1.signal();
  }
}
```

```
static class B
 extends Thread {
  public void run() {
    s1.await();
    s3.await();
    n = n * n;
    s2.signal();
  }
}
```

```
static class C
 extends Thread {
  public void run() {
    s1.await();
    n = n + 2;
    s3.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?

(a) 0 y 8
(b) 4
(c) 8

(1 punto) 2. Si en el código anterior el semáforo s1 se inicializa a 1:

(a) No está garantizada la terminación de las tres tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1.5 puntos) 3. Dado un programa concurrente en la que tres threads instancias de las clases A, B y C comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(0);
static Semaphore s2 = new Semaphore(1);
static Semaphore s3 = new Semaphore(0);
```

```
static class A
 extends Thread {
  public void run() {
    s1.await();
    s2.await();
    n = 2 * n;
    s2.signal();
  }
}
```

```
static class B
 extends Thread {
  public void run() {
    s3.await();
    n = n * n;
    s2.signal();
  }
}
```

```
static class C
 extends Thread {
  public void run() {
    s2.await();
    n = n + 2;
    s1.signal();
    s3.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?

(a) 8
(b) 2
(c) 4 u 8

(1 punto) 4. Si en el código anterior los semáforos s1 y s2 se inicializa a 0 y el semáforo s3 a 1:
(a) No está garantizada la terminación de las tres tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1 punto) 5. Dado un programa concurrente en la que dos threads instancias de las clases A y B comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(0);
static Semaphore s2 = new Semaphore(1);
static Semaphore s3 = new Semaphore(2);
```

```
static class A extends Thread {
  public void run() {
    s1.await();
    s3.await();
    n = n + 2;
    s2.signal();
  }
}
```

```
static class B extends Thread {
  public void run() {
    s2.await();
    s3.await();
    n = n * 4;
    s1.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?
(a) 2
(b) 8

(1 punto) 6. Si en el código anterior el semáforo s1 se inicializa a 1:
(a) No está garantizada la terminación de ambas tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1 punto) 7. Si en el código anterior el semáforo s3 se inicializa a 1:
(a) No está garantizada la terminación de ambas tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1.5 puntos) 8. Dado un programa concurrente en la que tres threads instancias de las clases A, B y C comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
```

```
class A extends Thread {
  public void run() {
    s2.await();
    n = 2 * n;
    s1.signal();
  }
}
```

```
class B extends Thread {
  public void run() {
    s1.await();
    n = n * n;
    s2.signal();
  }
}
```

```
class C extends Thread {
  public void run() {
    s1.await();
    n = n + 2;
    s2.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?
(a) 2 o 16
(b) 4
(c) 4 o 16

(1 punto) 9. Si en el código anterior los semáforos s1 y s2 se inicializa a 1:
(a) No está garantizada la terminación de las tres tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1.5 puntos) 10. Dado un programa concurrente en la que dos threads instancias de las clases A y B comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
```

```
static class A extends Thread {
  public void run() {
    s2.await();
    n = 3 * n;
    s1.signal();
  }
}
```

```
static class B extends Thread {
  public void run() {
    s1.await();
    n = n + 2;
    s2.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?
(a) 2
(b) 6

(1 punto) 11. Si en el código anterior el semáforo s1 se inicializa a 1:
(a) No está garantizada la terminación de ambas tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1.5 puntos) 12. Dado un programa concurrente en la que tres threads instancias de las clases A, B y C comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
```

```
static class A
 extends Thread {
  public void run() {
    s2.await();
    n = 3 * n;
    s1.signal();
  }
}
```

```
static class B
 extends Thread {
  public void run() {
    s1.await();
    n = n + 2;
    s2.signal();
  }
}
```

```
static class C
 extends Thread {
  public void run() {
    s1.await();
    n = n + 4;
    s2.signal();
  }
}
```

¿Cuáles son los posibles valores de n tras terminar los tres threads?
(a) 6, 10, 14 y 18
(b) 14 y 6
(c) 10 y 14

(1 punto) 13. Si en el código anterior el semáforo s1 se inicializa a 2:
(a) No está garantizada la terminación de las tres tareas.
(b) No está garantizada la exclusión mutua en el acceso a n.

(1.5 puntos) 14. Supóngase un programa concurrente con un semáforo inicializado a 0. Tres threads invocan el método await sobre dicho semáforo y posteriormente otros dos threads invocan el método signal sobre el mismo semáforo. Supóngase que el programa se detiene sin realizar más invocaciones de métodos sobre dicho semáforo. Se pide marcar la afirmación correcta:
(a) El valor del semáforo es 0 al detener el programa.
(b) El valor del semáforo es 1 al detener el programa.
(c) El valor del semáforo es 2 al detener el programa.

(1.5 puntos) 15. Obsérvese la siguiente implementación del recurso compartido CuentaBancaria:

```
class CuentaBancaria {
  static Semaphore saldo = new Semaphore(0);
  static Semaphore atomic = new Semaphore(1);

   public void reintegro(int c) {
     atomic.await();
     for(int i=0; i<c; i++)
       saldo.await();
     atomic.signal();
   }

   public void ingreso(int c) {
     atomic.await();
     for(int i=0; i<c; i++)
       saldo.signal();
   }
}
```

La idea principal consiste en que el semáforo saldo representa el valor interno (de tipo N) del recurso. Asumiendo que se quiere atender a los procesos que quieren realizar reintegros en estricto orden de llegada, se pide señalar la respuesta correcta:
(a) Es una implementación incorrecta del recurso compartido
(b) Es una implementación correcta del recurso compartido

(1 punto) 16. Se pide señalar la respuesta correcta teniendo en cuenta que no hay otros métodos más allá del método run:
(a) El acceso a un atributo no estático y privado de tipo int de un thread desde un método run nunca es una sección crítica.
(b) El acceso a un atributo no estático y privado de tipo int de un thread desde un método run puede ser una sección crítica.

(2 puntos) 17. Dado un programa concurrente en la que tres threads instancias de las clases C, D y E comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);


static class C extends Thread {
  public void run() {
    s2.acquire();
    s1.acquire();
    n = 2 * n;
    s1.release();
  }
}
```

```
static class D extends Thread {
  public void run() {
    s1.acquire();
    n = n * n;
    s1.release();
  }
}

static class E extends Thread {
  public void run() {
    s1.acquire();
    n = n + 3;
    s2.release();
    s1.release();
  }
}
```

Se pide marcar el conjunto que contiene únicamente los posibles valores de la variable n tras la terminación de los tres threads:
(a) {3,6,18,36}
(b) {6,18,36}
(c) No está garantizada la terminación de los tres tareas
(d) {3,6,9,18,36}

(2 puntos) 18. Dado un programa concurrente en la que tres threads instancias de las clases C, D y E comparten una variable n:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);


static class C extends Thread {
  public void run() {
    s2.acquire();
    s1.acquire();
    n = 3 * n;
    s1.release();
  }
}
```

```
static class D extends Thread {
  public void run() {
    s1.acquire();
    n = n * n;
    s1.release();
  }
}

static class E extends Thread {
  public void run() {
    s1.acquire();
    n = n + 2;
    s2.release();
    s1.release();
  }
}
```

Se pide marcar el conjunto que contiene únicamente los posibles valores de la variable n tras la terminación de los tres threads:
(a) {2,12,36}
(b) {12,36}
(c) {6,12,36}
(d) {2,6,12}

(2 puntos) 19. Dadas las clases de threads T1, T2 y T3, que comparten una variable n, se lanzan sendas instancias t1, t2 y t3:

```
static int n = 3;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(1);
static Semaphore s3 = new Semaphore(0);


static class T1 extends Thread {
  public void run() {
    s1.acquire();
    s2.acquire();
    n = n + 2;
  }
}
```

```
static class T2 extends Thread {
  public void run() {
    s2.acquire();
    s3.acquire();
    n = n * n;
    s2.release();
  }
}

static class T3 extends Thread {
  public void run() {
    s1.acquire();
    n = n * n;
    s1.release();
    s2.release();
    s3.release();
  }
}
```

Se pide marcar la respuesta correcta:
(a) No está garantizada la exclusión mutua en el acceso a la variable n.
(b) Está asegurada la terminación de la tarea t3.
(c) Si las tres tareas terminan y no se viola la exclusión mutua, la variable n puede tomar el valor 100
(d) Si las tres tareas terminan y no se viola la exclusión mutua, la variable n puede tomar el valor 38

(2 puntos) 20. Dadas las clases de threads T_1, T_2 y T_3, que comparten una variable t, se lanzan sendas instancias s1, s2 y s3:

```
static int t = 3;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
static Semaphore s3 = new Semaphore(0);


static class T_1 extends Thread {
  public void run() {
    s1.acquire();
    s2.acquire();
    t = t + 2;
    s1.release();
  }
}
```

```
static class T_2 extends Thread {
  public void run() {
    s2.acquire();
    s3.acquire();
    t = t * 2;
    s2.release();
  }
}

static class T_3 extends Thread {
  public void run() {
    s3.release();
    s1.acquire();
    t = t * t;
    s1.release();
    s1.release();
    s2.release();
  }
}
```

Se pide marcar la respuesta correcta:
(a) No está garantizada la exclusión mutua en el acceso a la variable t.
(b) Está asegurada la terminación de la tarea t3.
(c) Tras terminar las tres tareas, la variable t puede tomar tanto el valor 20 como el 22.
(d) No está garantizada la terminación de las tres tareas.

(2 puntos) 21. Dado un programa concurrente en la que tres threads instancias de las clases C, D y E comparten una variable n:

```
static int n = 1;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);


static class C extends Thread {
  public void run() {
    s2.acquire();
    s1.acquire();
    n = 2 * n;
    s1.release();
  }
}
```

```
static class D extends Thread {
  public void run() {
    s1.acquire();
    n = n * n;
    s1.release();
  }
}

static class E extends Thread {
  public void run() {
    s1.acquire();
    n = n + 3;
    s1.release();
    s2.release();
    s1.release();
  }
}
```

Se pide marcar el conjunto que contiene únicamente los posibles valores de la variable n tras la terminación de los tres threads:
(a) {4, 16, 32, 64}
(b) {4, 8, 32, 64}
(c) {8, 32, 64}
(d) {5, 8, 32}

(1 punto) 22. Obsérvese la siguiente implementación del recurso compartido MultiCont:

```
class MultiCont {
  private Semaphore permInc = new Semaphore(N);
  private Semaphore permDec = new Semaphore(0);
  private Semaphore atomicInc = new Semaphore(1);
  private Semaphore atomicDec = new Semaphore(1);

  public void inc(int n) {
    atomicInc.await();
    for(int i = 0; i < n; i++)
      permInc.await();
    atomicInc.signal();
  }
```

```
public void dec(int n) {
  atomicDec.await();
  for(int i = 0; i < n; i++)
    permDec.await();
  atomicDec.signal();
}
```

Asumiendo que se quiere atender a los procesos que quieren incrementos en estricto orden de llegada y a los que quieren realizar decrementos también en estricto orden de llegada, se pide señalar la respuesta correcta.
(a) Es una implementación correcta del recurso compartido.
(b) En una implementación incorrecta del recurso compartido.

(1 punto) 23. Se pide señalar si la siguiente afirmación es cierta o falsa:
El método signal de la clase es.upm.babel.cclib.Semaphore puede bloquear al proceso que lo invoca.
(a) Falso
(b) Cierto

(1 punto) 24. Dado el siguiente programa concurrente:

```
static class Semaforos {
  static Semaphore s = new Semaphore(0);

  static class A extends Thread {
    public void run() {
      s.await();
      System.io.println("Soy un thread A");
    }
  }

  static class B extends Thread {
    public void run() {
      s.signal();
      System.io.println("Soy un thread B");
    }
  }

  public static final void main(final
   String[] args) {
    Thread a = new A();
    Thread b = new B();
  }
}
```

Se pide señalar si la siguiente afirmación es cierta o falsa:

Soy un thread A	|	Soy un thread B
Soy un thread B	|	Soy un thread A

(a) Falso
(b) Cierto

(1 punto) 25. Supóngase un programa concurrente con un semáforo inicializado a 0. Dos procesos invocan el método await sobre dicho semáforo y posteriormente un tercer proceso invoca el método signal sobre el mismo semáforo. Supóngase que el programa se detiene sin realizar más invocaciones de métodos sobre dicho semáforo:
(a) El valor del semáforo es 1 al detener el programa
(b) El valor del semáforo es 0 al detener el programa

(1 punto) 26. Obsérvese la siguiente implementación del recurso compartido MultiCont:

```
class MultiCont {
  private Semaphore permInc = new Semaphore(N);
  private Semaphore multicont = new Semaphore(0);

  public void inc(int n) {
    for(int i = 0; i < n; i++)
      permInc.await();
    for(int i = 0; i < n; i++)
      multicont.signal();
  }

  public void dec(int n) {
    for(int i = 0; i < n; i++)
      multicont.await();
    for(int i = 0; i < n; i++)
      permInc.signal();
  }
}
```

Asumiendo que se quiere atender a los procesos que quieren incrementos en estricto orden de llegada y a los que quieren realizar decrementos también en estricto orden de llegada, se pide señalar la respuesta correcta.
(a) Es una implementación correcta del recurso compartido.
(b) En una implementación incorrecta del recurso compartido.